Machine Learning for Autonomic System Operation in the Heterogeneous Cloud-Edge Continuum

Contract Number 101092912

D6.4 MLSysOps Open Datasets

Lead Partner	UCD
Contributing Partners	ALL
Owner / Main Author	Dimitris Chatzopoulos, Theodoros Aslanidis (UCD)
Contributing Authors	*UNICAL*: Raffaele Gravina, Gianluca Aloi, Antonio Iera, Pasquale Pace, Michele Gianfelice, Francesco Pupo - *NTT DATA*: Roberto Zambetti - *UTH*: Christos Antonopoulos, Nikos Bellas, Alexandros Patras - *UCD*: Thodoris Aslanidis - *INRIA*: Valeria Loscri - *FHP*: Carlos Resende - *NVIDIA*: Dimitris Syrivelis - *NUBIS*: Anastassios Nanos - *CC*: Marcell Fehér, *UBIW*: Sofia Rosas, Ana Pereira, *AUG*: Dimitris Akridas – *TUD*: Kaitai Liang, Rui Wang
Reviewers	Dimitris Chatzopoulos (UCD)
Contractual Delivery Date	M37 (31 January 2026)
Actual Delivery Date	9 February 2026
Version	1.0
Dissemination Level	Public

HORIZON EUROPE

The research leading to these results has received funding from the European Community's Horizon Europe Programme (HORIZON) under grant no. 101092912.

Change Log

Version	Date	Summary of Changes
0.1	05/01/2026	Skeleton with indicative template for all entries
0.2	10/01/2026	Introduction ready
0.3	15/01/2026	Entries for several date sets and ML models ready
0.4	30/01/2026	Entries for all date sets and ML models ready
0.5	02/02/2026	Conclusion ready
0.6	04/02/2026	Fix formatting errors
0.7	05/02/2026	Ready for internal review
1.0	09/02/2026	Final version of the deliverable

Table of Contents

Summary

Deliverable D6.4 reports the final status of the datasets and machine learning (ML) models made publicly available within the MLSysOps project, in accordance with FAIR principles and the project's open science strategy. To ensure long-term accessibility and scientific transparency, Zenodo was selected as the primary platform to host the MLSysOps Community, serving as a persistent repository for sharing datasets, ML models, and technical reports.

This deliverable provides a comprehensive index and technical description of nineteen distinct assets:

- Eleven Machine Learning Models: These models, produced through collaborative efforts between partners, are primarily provided in the ONNX (Open Neural Network Exchange) format to ensure cross-platform interoperability and ease of deployment.
- Eight Public Datasets: These resources include raw and processed data collected from diverse sources, including real-world IoT testbeds, 5G signal monitoring, and high-fidelity system simulators.

By centralizing these resources, Deliverable D6.4 establishes a foundation for future research in autonomic system management across the cloud-edge continuum. The availability of these assets directly supports the project's goal of fostering an open research ecosystem for AI-driven infrastructure management.

Abbreviations

FAIR	Findable, Accessible, Interoperable, and Reusable
FPGA	Field-Programmable Gate Array
LSTM	Long Short-Term Memory
ML	Machine Learning
ONNX	Open Neural Network Exchange
RL	Reinforcement Learning

1 Introduction

Deliverable D6.4 provides a centralized index and technical overview of the datasets and machine learning (ML) models produced throughout the MLSysOps project. As the project focuses on designing an AI-controlled framework for autonomic system management across the cloud-edge continuum, these open-source resources are part of the practical foundation of its research and development efforts. By documenting nineteen distinct assets—comprising eight datasets and eleven machine learning models—this document serves as a comprehensive guide for anyone interested in replicating project results or building upon them. To ensure maximum impact and scientific integrity, all assets have been curated in accordance with open science standards and hosted in a persistent public repository (Zenodo).

1.1 FAIR Principles

The FAIR principles represent a set of guidelines designed to optimize the reuse of scientific data by both humans and machines. FAIR is an acronym and every letter has an important meaning. In detail:

- **F**indable: Data and metadata should be easy to find for both humans and computers. This involves using unique and persistent identifiers (like DOIs) and indexing data in searchable resources.
- **A**ccessible: Once found, users need to know how the data can be accessed, possibly including authentication and authorization. Data should be retrievable using standard communication protocols.
- **I**nteroperable: Data needs to be integrated with other data. It should use a formal, accessible, shared, and broadly applicable language for knowledge representation.
- **R**eusable: The ultimate goal of FAIR is to optimize the reuse of data. To achieve this, metadata and data should be well-described so that they can be replicated and/or combined in different settings.

The project selected Zenodo as its primary platform because it directly supports these FAIR objectives through several key features:

- Persistent Identifiers: Zenodo automatically assigns a Digital Object Identifier (DOI) to every upload. This makes the datasets and models permanently citable and "Findable".
- Community Curation: It allows for the creation of a dedicated MLSysOps Zenodo Community. This centralizes all project results -- including the eight datasets, eleven models as well as reports -- making them easier for researchers to browse in one location.
- Long-Term Preservation: As a non-commercial repository hosted by CERN, Zenodo provides a stable, long-term home for data, ensuring "Accessibility" even after the project concludes.
- Metadata Support: The platform requires structured metadata for every upload, which improves "Interoperability" and "Reusability" by providing context such as authors, descriptions, and usage instructions.
- Open Access Integration: Zenodo is built to support Open Access mandates, making it easy to comply with grant requirements while sharing work with the public

1.2 Datasets

The project has released eight public datasets hosted on Zenodo to support research in autonomic system operations. In detail:

1. Ubiwhere Smart Lamppost Dataset: Environmental and traffic data (GPU/CPU usage, noise, temperature, and person/car/motorcycle detections) from a smart lamppost in Aveiro, Portugal.
2. 5G Jamming Attack Detection Dataset: Physical layer (PHY) cellular network traces, including signal strength (RSRP, RSRQ, SINR), thermal sensors, and RF transmission power from an Android device under 5G jamming scenarios.
3. INRIA I/Q Signal Dataset: Raw radio signal traces (In-Phase/Quadrature) for RF fingerprinting and physical layer authentication, focusing on unique hardware impairments.
4. Tractor-Drone Co-Robotics Dataset: Telemetry and computer vision metrics from a smart agriculture system where an autonomous drone assists a tractor when its cameras are blinded by sun glare.
5. TUD Telemetry Dataset: Host telemetry snapshots including per-core CPU utilization (idle, iowait, irq, etc.) and various memory metrics (used, free, available) for anomaly detection.
6. Object Storage Transfer Speeds Dataset: Data documenting upload and download performance across various cloud providers and regions for analyzing network throughput.
7. Job Placement Failures Dataset: Over one million rows of data from a simulated datacenter documenting job placement attempts under varying network and fragmentation conditions.
8. FPGA ML Inference Telemetry Dataset: Performance metrics from FPGA-based ML inference, including latency breakdowns, per-thread throughput, and memory bandwidth usage.

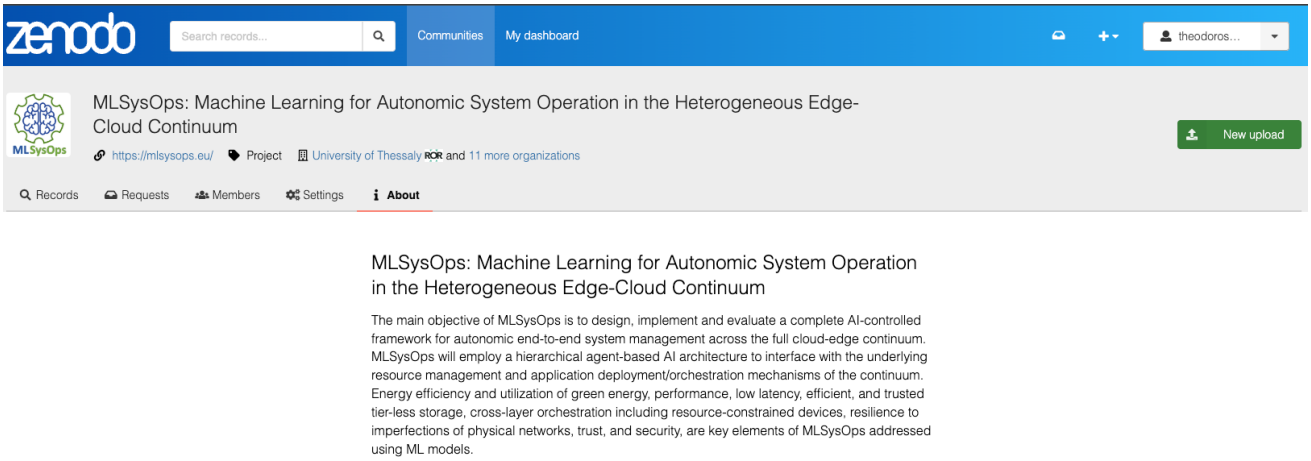
1.3 Models

The consortium has produced eleven machine learning models , primarily in ONNX format for cross-platform compatibility:

1. Cluster VM Management Model: A reinforcement learning agent that recommends creating, destroying, or doing nothing with VMs based on infrastructure state.
2. 5G Jamming Attack Detection Model: An LSTM-based neural network that processes 5G signal features to identify anomalies or intentional jamming.
3. RF Fingerprinting Models: Models designed for physical layer authentication by identifying specific authorized devices from their radio signal transients.
4. SkyFlok Latency Prediction Models: Regression models (one for each of six backends) that predict file transfer times for specific cloud storage routes.
5. Smart Lamppost Noise Prediction Model: A multivariate LSTM that estimates future environmental noise levels (dB) based on real-time traffic and pedestrian counts.
6. Drone Deployment Prediction Model: An XGBoost classifier that predicts the "should_fly" signal to trigger proactive drone deployment in agriculture.
7. 5G Latency Optimization Prediction Model: A model designed to optimize and predict latency behaviors within 5G network configurations.
8. Anomaly Detection Model: A reconstruction-based model (e.g., autoencoder) trained on host telemetry to identify system performance deviations.
9. PeakLife VM Predictor: A model that simultaneously predicts future CPU utilization and the remaining lifetime of a virtual machine for proactive resource management.
10. Job Placement Failure Predictor: A model trained on simulated datacenter data to predict the outcome of job placement attempts under varying load.
11. FPGA-Based RL Policy: A Reinforcement Learning policy specialized for selecting discrete actions in FPGA configurations based on system telemetry

1.4 The MLSysOps Zenodo Community

The MLSysOps community created at Zenodo can be found in the following link: https://zenodo.org/communities/mlsysops/about



2 MLSysOps Public Datasets

On this section we introduce in detail every dataset that is made public in the context of MLSysOps. For every entry we provide the Zenodo link, the citation that was produced by Zenodo when the dataset was uploaded as well as copied information from the Zenodo entry.

2.1 Ubiwhere Smart Lamppost Dataset

2.1.1 Link

https://zenodo.org/records/18245141

2.1.2 Citation

Ubiwhere (Portugal). (2026). Ubiwhere Smart Lamppost Dataset (v1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.18245141

2.1.3 *More details*

About the Dataset

This dataset was collected by Ubiwhere from a smart lamp post installed in the company headquarters, in city of Aveiro, Portugal. The smart lamp post is equipped with video and sound sensors (camera and microphone) and captures environmental and traffic-related data.

Time period: Data covers from 2025-08-22 13:10:00 to 2025-08-29 12:00:00.

About the Smart Lamppost.

This dataset was generated from a Smart Lamppost IoT installation by Ubiwhere in company headquarters, in the city of Aveiro, Portugal. The Smart Lamppost is a modular urban infrastructure solution that supports: intelligent LED lighting with remote management and analytics, optional electric vehicle (EV) charging capability, edge computing and telemetry, neutral hosting for 4G/5G telecom services, environmental sensing (video, sound, noise level, temperature).

For more details, see the product page: https://www.ubiwhere.com/en/products/smart-cities/smart-lamppost/

Dataset Features

- timestamp: Date and time of the observation (ISO 8601 format)
- GPU Usage: Percentage of GPU utilization of the device
- CPU Usage: Percentage of CPU utilization of the device
- Memory Used: Amount of RAM used by the device (in bytes)
- Jetson Energy: Energy consumption of the Jetson device (in joules or watt-seconds)
- Switch Energy: Energy consumption of the network switch (in joules or watt-seconds)
- Inference Time: Time taken for AI inference processing (in seconds)
- Tracking Time: Time taken for object tracking processing (in seconds)
- Noise Level: Measured environmental noise level (in decibels, dB)
- Temperature: Ambient temperature recorded by the sensor (in degrees Celsius)

- Instantaneous Person Detections: Number of persons detected in the current instant/frame
- Instantaneous Car Detections: Number of cars detected in the current instant/frame
- Instantaneous Motorcycle Detections: Number of motorcycles detected in the current instant/frame

Intended Use

- Environmental monitoring
- Traffic analysis
- Energy consumption profiling
- AI and ML model training for object detection and inference optimization
- Smart city infrastructure research

Column Summary and Data Types

Descriptor	Count	Mean	Std	Min	25%	50% (Median)	75%	Max
GPU Usage (i64)	597,002	96.2	15.0	0.0	100.0	100.0	100.0	100.0
CPU Usage (f64)	597,002	11.2	0.7	9.6	10.8	11.1	11.4	36.5
Memory Used (i64)	597,002	7.21e9	1.07e8	5.75e9	7.11e9	7.21e9	7.30e9	7.51e9
Jetson Energy (f64)	597,002	16.1	1.3	10.4	15.0	15.9	17.0	22.1
Switch Energy (f64)	597,002	13.0	2.0	9.5	11.3	11.4	15.4	16.3
Inference Time (f64)	597,002	0.2	0.0	0.1	0.2	0.2	0.2	0.3
Tracking Time (f64)	597,002	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Noise Level (f64)	597,002	12.2	7.2	-3.5	8.7	10.4	12.9	55.4
Temperature (f64)	596,972	33.8	9.7	22.1	24.7	27.3	44.7	46.6
Inst. Person Detections (i64)	597,002	0.0	1.6	0.0	0.0	0.0	0.0	777.0
Inst. Car Detections (i64)	597,002	0.0	1.3	0.0	0.0	0.0	0.0	580.0
Inst. Motorcycle Detections (i64)	597,002	0.0	0.0	0.0	0.0	0.0	0.0	7.0

2.2 5G Jamming Attack Detection Dataset

2.2.1 Link

https://zenodo.org/records/18253312

2.2.2 Citation

Xu, J., Moheddine, A., Loscri, V., Brighente, A., & Conti, M. (2026). INRIA SHIELD Framework Dataset - 5G Jamming Attack Detection (v1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.18253312

2.2.3 More details

Overview

This dataset contains physical layer (PHY) cellular network traces collected from an Android smartphone (OnePlus Nord 2T 5G) under **5G jamming attack scenarios**.

It serves as the official training and validation data for the **SHIELD Framework**.

- **Paper:** SHIELD: Scalable and Holistic Evaluation Framework for ML-Based 5G Jamming Detection
- **Source Code:** The full Android application and model training code are available on GitHub: https://github.com/mlsysops-eu/model-5g-jamming-detection

Authors

- **Jiali Xu** (Inria Centre at the University of Lille)
- **Aya Moheddine** (Inria Centre at the University of Lille)
- **Valéria Loscrì** (Inria Centre at the University of Lille)
- **Alessandro Brighente** (Department of Methematics, University of Padova)
- **Mauro Conti** (Department of Methematics, University of Padova)

File *Description*

1. Raw Data (data/raw/)

- **replay.log**: The unprocessed Android radio log captured via adb logcat -b radio. It contains mixed streams of signal reports, thermal sensors, and modem debug messages.

2. Processed Data (data/processed/)

- **fused_input.csv**: **(Recommended for Use)** The synchronized, feature-engineered dataset ready for Machine Learning.
  - **Frequency:** 1Hz (Resampled)
  - **Dimensions:** 60 Columns (+1 timestamp)
  - **Format:** Time-series matrix suited for LSTM/RNN models.

3. Configuration (config/)

- **1plus-nord2t.yaml**: Defines the Regular Expressions (Regex) used to parse the raw log file. It maps specific log tags (e.g., AT< +ECSQ) to data features.

4. Tooling (scripts/)

- **parse_data.py**: A Python script that reads 1plus-nord2t.yaml to extract raw metrics from the log into intermediate CSVs.
- **fuse_data.py**: A Python script that performs time-synchronization (linear interpolation) and feature extraction (rolling window statistics).

Dataset Schema (fused_input.csv)

The fused dataset contains **60 feature columns**. These are derived from **12 Raw Metrics** processed through **5 Statistical Aggregations** over a 10-second sliding window.

The *12 Raw Metrics*

1. **Signal Strength (3):** ssRsrp, ssRsrq, ssSinr (Standard 5G metrics).
2. **Extended Quality (6):** ecsq_idx0, ecsq_idx1, ecsq_idx2, ecsq_idx5, ecsq_idx6, ecsq_idx8 (Specific modem quality indices from AT+ECSQ).
3. **Thermal (2):** thermal_idx3, thermal_idx5 (Device internal temperature sensors).
4. **RF Transmission (1):** erftx_idx9 (Uplink transmission power state).

The 5 Aggregations (Suffixes)

For each raw metric above, the following statistics are calculated:

- _mean: Average value over the window.
- _max: Maximum value.
- _min: Minimum value.
- _std: Standard deviation (Stability indicator).
- _amplitude: Difference between Max and Min (max - min).

Total Dimensions: 12 metrics × 5 aggregations = **60 Columns**.

Example Column Names:

- ssRsrp_mean (Average Signal Power)
- ssSinr_std (Signal to Noise Stability)
- thermal_idx3_amplitude (Temperature fluctuation)

Usage Instructions

Option A: Quick Start (ML Training)

Load the pre-processed file directly into your model.

```
import pandas as pd
df = pd.read_csv("data/processed/fused_input.csv", index_col="timestep")
print(df.shape)
# Output: (Rows, 60)
```

Option B: Reproduce the Pipeline

If you wish to change the preprocessing parameters (e.g., change window size from 10s to 5s), follow these steps:

1. Create a python environment:

```
python -m venv venv
source venv/bin/activate
```

2. Install Requirements:

```
pip install pandas pyyaml
```

3. Run the Parser: Extracts the raw numbers from data/raw/replay.log using the rules in config/1plus-nord2t.yaml.

```
python scripts/parse_data.py
```

Output: Creates a parsed_data/ folder with individual CSVs.

4. Run the Fuser: Synchronizes the data to 1Hz and calculates rolling statistics.

```
python scripts/fuse_data.py
```

Output: Generates a new fused_data.csv.

2.3 I/Q Signal Dataset for RF Fingerprinting and Physical Layer Authentication

2.3.1 Link

https://zenodo.org/records/18268648

2.3.2 Citation

Alla, I., Yahia, S., Loscri, V., & eldeeb, . hossien . (2026). INRIA I/Q Signal Dataset for RF Fingerprinting and Physical Layer Authentication (v1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.18268648

2.3.3 More details

Overview

This dataset contains **Raw I/Q (In-Phase/Quadrature)** radio signal traces collected using a **BladeRF AX4** Software Defined Radio (SDR) and **GNU Radio**.

It serves as the official training and validation data for the **PLA-AP** project (Physical Layer Authentication), designed to evaluate machine learning approaches for identifying wireless devices based on their hardware impairments (RF fingerprints).

- **Paper:** Robust Device Authentication in Multi-Node Networks: ML-Assisted Hybrid PLA Exploiting Hardware Impairments
- **Source Code:** The preprocessing and model training code is available on GitHub: https://github.com/mlsysops-eu/model-physical-layer-authentication

Authors

- **Ildi Alla** (Inria Centre at the University of Lille)
- **Selma Yahia** (Inria Centre at the University of Lille)
- **Valéria Loscrì** (Inria Centre at the University of Lille)
- **Hossien Eldeeb** (University of Cambridge)

File Description

Raw Data (raw/)

This directory contains the binary signal files captured directly from the SDR.

- **Format:** Binary I/Q data (Interleaved 32-bit floats).
- **Content:** Each file captures the "burst" transmission of a specific device, including the transient (turn-on) and steady-state phases.
- **Organization:** The files are organized by Device ID (e.g., device1, device2).

Dataset Technical Specifications

The data was collected under controlled experimental conditions to ensure reproducibility.

Hardware Setup

Receiver: BladeRF AX4 (SDR) connected to a host PC running GNU Radio.

Transmitters: Various commercial-off-the-shelf (COTS) wireless devices (e.g., NRF52840 dongles or similar IoT nodes).

Signal Characteristics

- **Sampling Rate:** 20 Msps (Mega Samples Per Second).
- **Center Frequency:** 2.4 GHz (ISM Band).
- **Data Format:** Complex64 (Interleaved 32-bit floats: I, Q, I, Q...).
- **Key Feature:** The dataset specifically targets the **transient phase** (the initial signal ramp-up), which contains the most distinct hardware fingerprints.

Usage Instructions

Loading Raw I/Q Data

Since this dataset contains raw binary files without headers, you can load them using Python and NumPy.

```
import numpy as np
import matplotlib.pyplot as plt

# 1. Define File Path
filename = "data/raw/device1_trial1.bin" # Replace with actual filename

# 2. Load Binary Data (Complex64)
# BladeRF/GNU Radio saves data as interleaved float32 (I, Q, I, Q...)
# This is equivalent to numpy's complex64 type
data = np.fromfile(filename, dtype=np.complex64)

# 3. Basic Visualization
plt.figure(figsize=(10, 4))
plt.plot(np.real(data[0:1000]), label="In-Phase (I)")
plt.plot(np.imag(data[0:1000]), label="Quadrature (Q)")
plt.title("Raw I/Q Signal Snippet")
plt.legend()
plt.show()
```

Processing the Data

To transform this raw data into features suitable for machine learning (e.g., Transient Detection, Filtering, Gabor Transform) and save them in a structured HDF5 format, please refer to the the source code implemented in the official GitHub repository:

- Preprocessing Logic: https://github.com/mlsysops-eu/model-physical-layer-authentication/blob/main/src/preprocessing.py
- Data Loader & HDF5 Saving: https://github.com/mlsysops-eu/model-physical-layer-authentication/blob/main/src/dataloader.py

2.4 Tractor-Drone Co-Robotics Dataset for Weed Detection

2.4.1 Link

https://zenodo.org/records/18293250

2.4.2 Citation

Augmenta (acquired by CNH Industrial). (2026). Augmenta Tractor-Drone Co-Robotics Dataset for Weed Detection (v1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.18293250

2.4.3 More details

This dataset contains telemetry and computer vision metrics collected from a Smart Agriculture co-robotics system developed by **Augmenta (acquired by CNH Industrial)**. The system consists of a tractor equipped with a "Field Analyzer" and an autonomous drone (UAV).

The data was collected to train Machine Learning models (specifically XGBoost) to predict the should_fly event—a signal that triggers the drone to launch and assist the tractor when the tractor's onboard cameras are blinded by environmental factors (e.g., sun glare/lens flare).

This work was conducted as part of the **MLSysOps** project (EU Horizon Europe).

System Context

The Augmenta system automates the application of fertilizers and herbicides using Real-Time Computer Vision.

1. Normal Operation: The tractor cameras detect weeds and spray precisely.

2. The Problem: When the sun is at a specific angle (e.g., sunset/sunrise), it creates lens flare, blinding the tractor's camera ("Sensor Fault"). The system enters "Safe Mode" and sprays the whole field blindly, wasting chemicals.

3. The Solution: The system predicts this fault and deploys a Drone to fly ahead of the tractor. The drone sends clear weed detection coordinates back to the tractor, allowing precise spraying to continue.

Data Dictionary

The dataset consists of time-series telemetry. The core goal is to predict should_fly using the sensor and performance metrics.

Column Name	Type	Description
timestamp	datetime	ISO 8601 Timestamp of the recording.
quality_indicator_1	int	Confidence metric: Number of data correspondences between samples.
quality_indicator_2	int	Confidence metric: Number of data points used for localization.
field_indicator_1	int	The number of detected weeds in the current frame.
field_indicator_2	float	Fraction of the field frame under environmental variation.

sensor_fault_probability_1	float	**Key Feature:** Probability of the camera sensor being blinded by the sun (0.0 to 1.0).
environment_sensor_1	float	Ambient light/environmental condition measurement.
processing_performance	float	Average processing speed/performance metric of the vision unit.
success_rate	float	Fraction of successfully processed image frames (0.0 to 1.0).
should_fly	int	**Target Variable:** Binary flag (0 or 1). 1 indicates the drone should be deployed.
heading	float	Instant heading of the vehicle (radians).
velocity	float	Instant velocity of the vehicle (m/s).
latitude	float	GNSS Latitude.
longitude	float	GNSS Longitude.
altitude	float	GNSS Altitude (meters).
time_since_sensor_fault	float	Time (seconds) elapsed since the last sensor fault

Statistical Summary

Descriptor	Count	Mean	Std	Min	25%	50% (Median)	75%	Max
quality_indicator_1	1053	545.06	145.27	9.0	454.0	531.0	631.0	1051.0
quality_indicator_2	1053	413.88	119.38	64.0	324.0	396.0	492.0	835.0
field_indicator_1	1053	52.12	58.25	0.0	18.0	37.0	66.0	767.0
field_indicator_2	1053	0.015	0.010	0.001	0.008	0.013	0.021	0.086
sensor_fault_prob_1	1053	0.185	0.271	0.000	0.0002	0.086	0.384	0.999
environment_sensor_1	1053	10187.9	8416.8	808.5	2456.2	8959.3	20497.2	25678.0
processing_perf	1053	12.48	1.98	4.99	10.48	13.03	14.44	15.52
success_rate	1053	0.51	0.47	0.00	0.00	0.57	1.00	1.00
should_fly	1053	0.37	0.48	0.0	0.0	0.0	1.0	1.0
heading	1053	0.81	1.66	-3.13	-0.66	1.34	2.48	3.13
velocity	1053	2.72	0.99	0.01	2.22	2.78	3.33	5.25
altitude	1053	276.11	38.97	254.2	255.7	270.9	273.5	443.5

Collection Methodology

- **Location:** Perivlepto, Volos, Greece (Augmenta Test Field).
- **Conditions:** Data was specifically collected during sunset/sunrise to induce lens flare and trigger the "Safe Mode" (sensor fault) scenarios.
- **Equipment:**
  - **Tractor Node:** Standard agricultural tractor with Augmenta Field Analyzer (Cameras + Edge Compute).
  - **Drone Node:** Custom UAV integrated with the Augmenta control stack.
- **Protocol:** The tractor performed "Back-and-Forth" scanning of the field. As the tractor turned into the sun, the `sensor_fault_probability` spiked, triggering the `should_fly` signal for the drone.

2.5 Telemetry Dataset for Anomaly Detection

2.5.1 Link

https://zenodo.org/records/18311353

2.5.2 *Citation*

Delft University of Technology. (2026). TUD Telemetry Dataset for Anomaly Detection (v1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.18311353

2.5.3 More details

Dataset description

This dataset contains snapshots of host telemetry metrics collected during different workload conditions. It is intended for training and evaluating anomaly detection models (e.g., reconstruction-based autoencoders).

The metrics cover:

- Per-core CPU utilization breakdown by state (percent)
- Memory metrics (bytes)

Data Generation Method

1. A node was instrumented with a telemetry pipeline (e.g., Prometheus + node exporter) to collect CPU and memory metrics at a fixed sampling interval.
2. Multiple workload scenarios were executed (e.g., no load / medium load / high load).
3. Metrics were exported to CSV with a fixed column order. Each row represents one telemetry snapshot.

Notes:

- CPU values are percentages per core and CPU state. Due to sampling/aggregation, values may occasionally slightly exceed 100.
- node_memory_MemTotal_bytes is constant for a given machine (total installed memory).

Columns

All CSV files share the same schema (38 columns). Units and meanings are listed below.

CPU columns (percent)

For each core i in {0,1,2,3}, the following columns represent the percentage of time spent in the given CPU state during the sampling window:

- cpu_i_idle, cpu_i_iowait, cpu_i_irq, cpu_i_nice, cpu_i_softirq, cpu_i_steal, cpu_i_system, cpu_i_user

Memory columns (bytes)

- memory_used_bytes: used memory in bytes (as exported by the telemetry pipeline)
- node_memory_Buffers_bytes: memory used for buffers

- node_memory_Cached_bytes: memory used for page cache
- node_memory_MemAvailable_bytes: estimate of memory available for starting new applications
- node_memory_MemFree_bytes: unused memory
- node_memory_MemTotal_bytes: total installed memory

Column descriptions (full list)

Column	Unit	Description
cpu_0_idle	%	Core 0 CPU time in idle state
cpu_0_iowait	%	Core 0 CPU time waiting on I/O
cpu_0_irq	%	Core 0 CPU time servicing interrupts
cpu_0_nice	%	Core 0 CPU time for niced processes
cpu_0_softirq	%	Core 0 CPU time servicing softirqs
cpu_0_steal	%	Core 0 CPU time stolen (virtualization)
cpu_0_system	%	Core 0 CPU time in kernel space
cpu_0_user	%	Core 0 CPU time in user space
cpu_1_idle	%	Core 1 CPU time in idle state
cpu_1_iowait	%	Core 1 CPU time waiting on I/O
cpu_1_irq	%	Core 1 CPU time servicing interrupts
cpu_1_nice	%	Core 1 CPU time for niced processes
cpu_1_softirq	%	Core 1 CPU time servicing softirqs
cpu_1_steal	%	Core 1 CPU time stolen (virtualization)
cpu_1_system	%	Core 1 CPU time in kernel space
cpu_1_user	%	Core 1 CPU time in user space
cpu_2_idle	%	Core 2 CPU time in idle state
cpu_2_iowait	%	Core 2 CPU time waiting on I/O
cpu_2_irq	%	Core 2 CPU time servicing interrupts
cpu_2_nice	%	Core 2 CPU time for niced processes
cpu_2_softirq	%	Core 2 CPU time servicing softirqs
cpu_2_steal	%	Core 2 CPU time stolen (virtualization)
cpu_2_system	%	Core 2 CPU time in kernel space
cpu_2_user	%	Core 2 CPU time in user space
cpu_3_idle	%	Core 3 CPU time in idle state
cpu_3_iowait	%	Core 3 CPU time waiting on I/O
cpu_3_irq	%	Core 3 CPU time servicing interrupts
cpu_3_nice	%	Core 3 CPU time for niced processes
cpu_3_softirq	%	Core 3 CPU time servicing softirqs
cpu_3_steal	%	Core 3 CPU time stolen (virtualization)
cpu_3_system	%	Core 3 CPU time in kernel space
cpu_3_user	%	Core 3 CPU time in user space
memory_used_bytes	bytes	Used memory
node_memory_Buffers_bytes	bytes	Buffers
node_memory_Cached_bytes	bytes	Cached
node_memory_MemAvailable_bytes	bytes	MemAvailable

node_memory_MemFree_bytes	bytes	MemFree
node_memory_MemTotal_bytes	bytes	MemTotal

Summary statistics

The following table reports per-column data type and summary statistics (min / median / max).

This table was computed from the provided file.

Column	Type	Min	Median	Max
cpu_0_idle	float64	0	29.03	100.5
cpu_0_iowait	float64	0	0.02	16.43
cpu_0_irq	float64	0	0	21.77
cpu_0_nice	float64	0	0	18.78
cpu_0_softirq	float64	0	0	13.74
cpu_0_steal	float64	0	0	18.48
cpu_0_system	float64	0	0.48	22.55
cpu_0_user	float64	0	30.5	63.15
cpu_1_idle	float64	0	29	104
cpu_1_iowait	float64	0	0.02	22.61
cpu_1_irq	float64	0	0	20.17
cpu_1_nice	float64	0	0	17
cpu_1_softirq	float64	0	0	26.32
cpu_1_steal	float64	0	0	17.09
cpu_1_system	float64	0	0.43	35.32
cpu_1_user	float64	0	30.63	75.72
cpu_2_idle	float64	0	28.91	100.4
cpu_2_iowait	float64	0	0.01	19.66
cpu_2_irq	float64	0	0	14.42
cpu_2_nice	float64	0	0	19.61
cpu_2_softirq	float64	0	0	16.19
cpu_2_steal	float64	0	0	15.58
cpu_2_system	float64	0	0.45	33.33
cpu_2_user	float64	0	30.7	86.3
cpu_3_idle	float64	0	29.01	112.5
cpu_3_iowait	float64	0	0.02	14.85
cpu_3_irq	float64	0	0	17.67
cpu_3_nice	float64	0	0	19.58
cpu_3_softirq	float64	0	0	19.25
cpu_3_steal	float64	0	0	15.61
cpu_3_system	float64	0	0.44	29.21
cpu_3_user	float64	0	30.62	70
memory_used_bytes	float64	8.89095e+08	1.70806e+09	3.32244e+09
node_memory_Buffers_bytes	float64	1.05865e+08	1.16023e+08	1.18623e+08
node_memory_Cached_bytes	float64	5.08577e+09	5.39835e+09	5.57918e+09

node_memory_MemAvailable_bytes	float64	5.00084e+09	6.61521e+09	7.43418e+09
node_memory_MemFree_bytes	float64	0	1.26609e+09	1.96274e+09
node_memory_MemTotal_bytes	float64	8.32328e+09	8.32328e+09	8.32328e+09

Reproducing the statistics table

To recompute the summary statistics for one or more CSV files (e.g., all training files plus the test file), run the following locally (requires pandas and numpy):

```
python - <<"PY"
import glob
import numpy as np
import pandas as pd

# Edit paths as needed
files = glob.glob("data/*.csv") + ["telemetry.csv"]

frames = [pd.read_csv(f) for f in files]
df = pd.concat(frames, ignore_index=True, sort=False)

rows = []
for col in df.columns:
    s = df[col]
    dtype = str(s.dtype)
    if pd.api.types.is_numeric_dtype(s):
        arr = s.to_numpy(dtype=float)
        rows.append((col, dtype, np.nanmin(arr), np.nanmedian(arr),
np.nanmax(arr)))
    else:
        rows.append((col, dtype, np.nan, np.nan, np.nan))

print("| Column | Type | Min | Median | Max |")
print("|---|---:|---:|---:|---:|")
for col, dtype, mn, med, mx in rows:
    def fmt(v):
        if isinstance(v, float) and np.isnan(v):
            return ""
        av = abs(float(v))
        if av >= 1e6:
            return f"{v:.6g}"
        return f"{v:.4g}"

    print(f"| `{col}` | `{dtype}` | {fmt(mn)} | {fmt(med)} | {fmt(mx)} |")
PY
```

2.6 Object Storage Transfer Speeds Dataset

2.6.1 Link

https://zenodo.org/records/18412125

2.6.2 Citation

Fehér, M. (2026). Chocolate Cloud Object Storage Transfer Speeds Dataset (v1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.18412125

2.6.3 More details

Overview

This dataset measures upload and download performance between Fly.io gateway regions (origins) and commercial object storage backends (targets). Each row is one measurement for a specific data size, initiated from a Fly.io region and recorded against a particular backend, and is intended for studying network performance, latency-sensitive placement, and cross-region transfer behavior.

Some records use 1-byte uploads/downloads to approximate latency by activating the target service's data path with minimal payload. For each timestamp, measurements include standard sizes (1 byte, 1 MB, 10 MB, 50 MB) plus a few random sizes up to 50 MB. The dataset includes ~900.000 measurements spanning 86 days between 2024-10-31 and 2025-01-24, with a pause from 2024-11-18 to 2024-12-18. Each measurement is uniquely identified by `(timestamp, origin_fly_region, target_backend_id, size_bytes)`.

CSV Columns

- `timestamp`: UTC datetime string for the measurement (timezone-aware, ISO 8601).
- `origin_fly_region`: Fly.io gateway region code (3-letter).
- `origin_countrycode`: ISO 3166-1 alpha-2 country code (lowercase) for the Fly.io gateway.
- `origin_city`: City of the Fly.io gateway.
- `origin_lat`: Latitude of the Fly.io gateway.
- `origin_lng`: Longitude of the Fly.io gateway.
- `target_backend_id`: Internal storage backend ID.
- `target_provider`: Cloud provider name.
- `target_region`: Cloud provider region.
- `target_countrycode`: ISO 3166-1 alpha-2 country code (lowercase) for the backend location.
- `target_city`: City of the storage backend.
- `target_timezone`: Time zone name for the backend.
- `target_lat`: Latitude of the storage backend.
- `target_lng`: Longitude of the storage backend.
- `target_local_time`: Local time at the target backend for the same instant as `timestamp`.
- `distance_km`: Great-circle distance between origin and target, in kilometers (rounded int).
- `size_bytes`: Data size in bytes for the measurement.
- `upload_time_ms`: Upload time in milliseconds.
- `download_time_ms`: Download time in milliseconds.
- `upload_speed_mbps`: Upload speed in megabits per second (2 decimal places).
- `download_speed_mbps`: Download speed in megabits per second (2 decimal places).

Intended Use Examples

- Compare upload/download performance across cloud providers and regions for a fixed data size.
- Identify nearest or best-performing storage backends for a given Fly.io region.
- Analyze how geographic distance correlates with throughput.
- Build placement or replication strategies based on observed network performance.
- Use as input for predictive models of transfer time or throughput.

Notes

- Rows are sorted by `timestamp` ascending.
- City names may contain commas and are properly quoted in the CSV.
- There are no missing values

Related ML Models

Models trained on this dataset are published at: https://zenodo.org/records/18288840

These models predict transfer time for a specific Fly.io region to storage-backend route at a given time and data size. There is a separate model for six backends and the Fly.io London (`lhr`) region.

The `target_backend_id` column is the internal unique ID of a region for a commercial cloud storage provider and is consistent with the backend identifiers used in the published models.

2.7 Job Placement Failure Dataset for Simulated Datacenter Clusters with Reconfigurable Optical Networks

2.7.1 Link

https://zenodo.org/records/18485585

2.7.2 Citation

Patras, A., Syrivelis, D., & Terzenidis, N. (2026). MLNX Job Placement Failure Dataset for Simulated Datacenter Clusters with Reconfigurable Optical Networks (1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.18485585

2.7.3 More details

This dataset contains **cluster-level snapshots and job placement outcomes** generated using a simulated large-scale datacenter environment. The data is intended for training and evaluating machine learning models that predict whether a job submission will succeed or fail given the current cluster state and job resource request.

The dataset was produced as part of the MLSysOps project (EU Horizon Europe) and supports research on:

- job admission control,
- failure prediction,
- resource fragmentation,
- and network feasibility in modern datacenter architectures.

Each data sample represents a single scheduling decision and includes both:

- detailed cluster state features, and
- the observed outcome of the placement attempt.

Simulated Datacenter Architecture

The dataset is generated using a proprietary datacenter simulator modeling a hierarchical cluster composed of Scalable Units (SUs).

Cluster configuration:

- 32 Scalable Units (SUs)
- 32 servers per SU (1024 servers total)
- 8 leaf switches per SU
- 8 GPUs per server
- Leaf switches interconnected via a reconfigurable optical circuit switch (OCS)

Failure Modes Captured

Each job placement attempt can result in:

- Successful placement
- Failure due to insufficient servers
- Failure due to insufficient or infeasible uplink connectivity

While server insufficiency can be determined via simple capacity checks, uplink infeasibility is more complex, as it depends on:

- current optical circuit configurations,
- contention between jobs,
- and connectivity constraints of the OCS fabric.

The dataset explicitly captures these outcomes to support learning-based approaches for failure prediction.

Dataset Structure

- Format: Apache Parquet
- Granularity: One row per scheduling decision
- Each row contains:
  1. Job request features
  2. Cluster state features (scalar + vector)
  3. Ground-truth placement outcome label

Rows are treated as independent samples.

Ground-Truth Labels

The dataset includes a label column encoding the observed outcome of the job placement:

Value	Meaning
0	Job placement succeeded
1	Job placement failed due to insufficient servers
2	Job placement failed due to insufficient uplinks / infeasible network connectivity

Notes:

- Labels 1 and 2 both indicate job failure, but with different root causes.
- This encoding allows:
  - binary failure prediction,
  - failure cause analysis,
  - and future multi-class modeling.

Feature Description

Scalar Cluster Features

These features summarize utilization, imbalance, and fragmentation across the cluster:

Column	Description
f1_event_type	The recorded event: add, failed_server, failed_uplink
f2_mean_util	Mean server utilization
f3_diff_max_min_util	Utilization imbalance across SUs
f4_cv_util	Coefficient of variation of server utilization

f5_ratio_max_to_mean_workload	Workload skew across SUs
f6_mean_uplink_util	Mean uplink utilization
f7_diff_max_min_uplink_util	Uplink utilization imbalance
f8_cv_uplink_util	Coefficient of variation of uplink utilization
f9_mean_combined_util	Combined compute and network utilization
f10_resource_imbalance	Compute vs network mismatch
f11_bottleneck_ratio	Network-to-compute utilization ratio
f12_frag_spread_sus	Fragmentation due to SU spread
f13_frag_wasted	Fragmentation due to wasted capacity
f14_frag_su_sparseness	Intra-SU sparseness
f15_total_servers_used	Total servers in use
f16_total_sus_used	Number of active SUs
f17_total_uplink_utilized	Total uplink usage

Vector Features

Feature	Description
f18_su_server_bitmap	Binary vector (length 1024) indicating per-server usage
f19_leaf_up	Vector (length 256) indicating leaf switch uplink utilization

Job Request Feature

Column	Type	Description
f20_requested_nodes	int / float	Number of nodes requested by the job

Feature Description

Scalar Cluster Features

These features summarize utilization, imbalance, and fragmentation across the cluster:

Column	Description
f1_event_type	The recorded event: add, failed_server, failed_uplink

f2_mean_util	Mean server utilization
f3_diff_max_min_util	Utilization imbalance across SUs
f4_cv_util	Coefficient of variation of server utilization
f5_ratio_max_to_mean_workload	Workload skew across SUs
f6_mean_uplink_util	Mean uplink utilization
f7_diff_max_min_uplink_util	Uplink utilization imbalance
f8_cv_uplink_util	Coefficient of variation of uplink utilization
f9_mean_combined_util	Combined compute and network utilization
f10_resource_imbalance	Compute vs network mismatch
f11_bottleneck_ratio	Network-to-compute utilization ratio
f12_frag_spread_sus	Fragmentation due to SU spread
f13_frag_wasted	Fragmentation due to wasted capacity
f14_frag_su_sparseness	Intra-SU sparseness
f15_total_servers_used	Total servers in use
f16_total_sus_used	Number of active SUs
f17_total_uplink_utilized	Total uplink usage

Vector Features

Feature	Description
f18_su_server_bitmap	Binary vector (length 1024) indicating per-server usage
f19_leaf_up	Vector (length 256) indicating leaf switch uplink utilization

Job Request Feature

Column	Type	Description
f20_requested_nodes	int / float	Number of nodes requested by the job

Data Collection Methodology

- Environment: Simulated datacenter

- Workloads: Synthetic job traces with varying sizes and arrival patterns
- Placement policy: Simulator-internal scheduling logic
- Labeling: Determined by placement outcome (success or failure cause)

The simulator executes job placement attempts under varying load, fragmentation, and network conditions to generate diverse training examples. The simulator itself is not publicly released. Only the resulting dataset is provided.

Statistical Summary

The dataset contains a total of 1,062,943 rows, each corresponding to a single job placement attempt in the simulated cluster.

The table below summarizes the distribution of all numeric columns, including the ground-truth label.

Column Summary and Data Types

Descriptor	Type	Count	Mean	Std	Min	25%	50% (Median)	75%	Max
l1_failed	int32	1,062,943	0.6198	0.7127	0.0	0.0	0.0	1.0	2.0
f2_mean_util	float32	1,062,943	0.9129	0.1089	0.0078	0.8906	0.9404	0.9717	1.0
f3_diff_max_min_util	float32	1,062,943	0.5843	0.3332	0.0	0.2813	0.5625	1.0	1.0
f4_cv_util	float32	1,062,943	0.1982	0.3047	0.0	0.0712	0.1446	0.2500	5.5678
f5_ratio_max_to_mean_workload	float32	1,062,943	1.1865	1.2299	1.0	1.0291	1.0633	1.1228	32.0
f6_mean_uplink_util	float32	1,062,943	0.5637	0.1156	0.0	0.5195	0.5840	0.6367	0.9598
f7_diff_max_min_uplink_util	float32	1,062,943	0.9016	0.1717	0.0	0.8125	0.9063	0.9688	2.0
f8_cv_uplink_util	float32	1,062,943	0.4681	0.3022	0.0	0.3526	0.4258	0.5076	3.8730
f9_mean_combined_util	float32	1,062,943	0.7383	0.1002	0.0039	0.7139	0.7563	0.7915	0.9716
f10_resource_imbalance	float32	1,062,943	0.3493	0.1010	0.0001	0.2803	0.3350	0.4043	0.8926

f11_bottleneck_ratio	float32	1,062,943	0.6137	0.1137	0.0	0.5636	0.6345	0.6894	1.7378
f12_frag_spread_sus	float32	1,062,943	1.0713	0.0818	1.0	1.0261	1.0524	1.0922	4.0
f13_frag_wasted	float32	1,062,943	0.0713	0.0818	0.0	0.0261	0.0524	0.0922	3.0
f14_frag_su_sparseness	float32	1,062,943	0.0177	0.0176	0.0	0.0065	0.0135	0.0235	0.2589
f15_total_servers_used	int64	1,062,943	934.86	111.47	8	912	963	995	1024
f16_total_sus_used	int64	1,062,943	31.11	3.10	1	31	32	32	32
f17_total_uplink_utilized	int64	1,062,943	4617.66	946.63	0	4256	4784	5216	7863
f20_requested_nodes	int64	1,062,943	54.96	38.77	8	20	44	87	128

The l1_failed column encodes job outcomes as:

- 0: success
- 1: failure due to insufficient servers
- 2: failure due to insufficient uplinks / infeasible connectivity

Both 1 and 2 correspond to job failures.

Working with the Data

Loading the Dataset (Python)

```
import pandas as pd
df = pd.read_parquet("final_merged.parquet")
print(df.head())
```

Loading Selected Columns

```
cols = ["f20_requested_nodes", "f2_mean_util", "l1_failed"]
df = pd.read_parquet("final_merged.parquet", columns=cols)
```

Tools and Documentation

Apache Parquet specification: https://parquet.apache.org/docs

Pandas Parquet I/O: https://pandas.pydata.org/docs/reference/api/pandas.read_parquet.html

PyArrow Parquet support: https://arrow.apache.org/docs/python/parquet.html

Intended Use

This dataset is intended for:

- machine learning research on job failure prediction,
- benchmarking admission-control models,
- studying resource fragmentation and network feasibility,
- offline evaluation of scheduling heuristics.

It is not intended to represent any specific production datacenter.

Limitations

- Data is generated from a simulator, not a production system.
- The cluster topology is fixed and may not generalize to other architectures.
- Temporal dependencies between jobs are not explicitly modeled.
- Network behavior is abstracted and may differ from real optical fabrics.

2.8 FPGA Telemetry Dataset for ML Inference Experiments on AMD/Xilinx ZCU102 MPSoC Development Board

2.8.1 Link

https://zenodo.org/records/18494461

2.8.2 Citation

Patras, A. (2026). UTH FPGA Telemetry Dataset for ML Inference Experiments on AMD/Xilinx ZCU102 MPSoC Development Board (1.0.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.18494461

2.8.3 More details

This dataset contains telemetry traces from repeated machine learning inference experiments executed on a Xilinx ZCU102 FPGA platform. Each experiment corresponds to a specific DPU bitstream configuration (DPU size and number of DPU compute units), a model variant (including pruning variants), and a system workload mode applied on the ARM CPU.

The dataset is intended to support research on:

- FPGA-based ML inference performance,
- DPU scaling and configuration trade-offs,
- interaction between FPGA accelerators and ARM CPU workloads,
- power, memory bandwidth, and system-level telemetry analysis.

Each experiment produces a time-series telemetry trace recorded during a batch inference run.

Dataset Structure

The dataset is organized into two main components:

- experiments.csv — index file
  One row per experiment run, describing its configuration.
- data/<experiment_id>.csv — telemetry trace
  Time-series telemetry recorded during the corresponding experiment.

`experiments.csv` is the entry point to the dataset. Each row describes one complete experiment run on the FPGA.

Index Columns

Column	Description
Experiment ID	Unique identifier for the experiment run. Used to locate the trace at data/<experiment_id>.csv.
DPU CU#	Number of DPU compute units (parallel inference threads).
DPU Size	DPU configuration size. The dataset includes 8 sizes and 26 total configurations when combined with CU counts.

Model	Model executed during the experiment. Pruning variants are encoded in the name (e.g., resnet18-25 for 25% pruning).
Workload Mode	Background workload on the ARM CPU: None, C-H (compute-bound), or M-H (memory-bound).

Experiments Overview

Aspect	Description
Hardware platform	Xilinx ZCU102 MPSoC Development Board
DPU configurations	26 total (8 DPU sizes × multiple CU counts)
Models	12 models with pruning variants (-25, -50, or none)
Run duration	From seconds up to ~10 minutes
Execution mode	Batch inference

Each row in experiments.csv corresponds to one full experiment run on the development board.

Telemetry Traces — `data/<experiment_id>.csv`

Each telemetry file contains a time series of system and inference measurements recorded during the experiment.

Time Columns

Column	Description
timestamp	Unix timestamp (seconds, fractional) when the telemetry row was recorded.
timestamp_human	Human-readable timestamp of the same moment.

These represent the actual time at which telemetry was sampled.

Per-Thread Performance Columns (*_K)

Many columns are indexed by K, representing per-DPU compute unit (per inference thread) measurements. K ranges from 1 to DPU CU#.

Column Pattern	Description
preprocessing_time_K	Preprocessing latency for inference thread K.
inference_time_K	Inference latency for thread K.

postprocessing_time_K	Postprocessing latency for thread K.
job_id_K	Identifier of the inference job executed on thread K.
fps_K	Frames-per-second observed for thread K.

Memory Bandwidth Telemetry (ZCU102 Ports)

Column	Description
S0_read … S4_read	Read bandwidth for memory ports 0–4.
S0_write … S4_write	Write bandwidth for memory ports 0–4.

Power Telemetry

Column	Description
arm_power	Instantaneous power consumption of the ARM CPU subsystem.
fpga_power	Instantaneous power consumption of the FPGA fabric.

CPU Utilization (ARM Cortex-A53)

Column	Description
cpu_0, cpu_1, cpu_2, cpu_3	Utilization of the four ARM Cortex-A53 cores.

Memory Metrics

Column	Description
memory_available	Available system memory at sampling time.
memory_total	Total system memory.
swap_free	Free swap space.

Experiment Linkage

Column	Description

experiment_id	Experiment identifier matching the filename and the index in experiments.csv.

How to Use the Dataset

- Select an experiment from experiments.csv based on:
  - DPU configuration (DPU Size, DPU CU#)
  - model variant (Model)
  - workload mode (Workload Mode)
- Open the corresponding telemetry trace:
  - data/<experiment_id>.csv
- Analyze:
  - inference latency breakdown (pre / infer / post),
  - per-thread throughput (fps_K),
  - memory bandwidth usage (S*_read, S*_write),
  - CPU utilization and power behavior under different workloads.

Intended Use

This dataset is intended for:

- FPGA performance analysis,
- ML inference benchmarking,
- system-level telemetry studies,
- research on accelerator–CPU interaction.

It represents a controlled experimental environment, not a production deployment.

3 MLSysOps Produced Models

On this section we introduce in detail every model that is made public in the context of MLSysOps. For every entry we provide the Zenodo and the GitHub links, the citation that was produced by Zenodo when the dataset was uploaded as well as copied information from the Zenodo entry.

All models developed are openly available both in the Zenodo community and the GitHub organization.

MLSysOps Zenodo Community: https://zenodo.org/communities/mlsysops

MLSysOps GitHub Organization: https://github.com/mlsysops-eu

3.1 Cluster VM Management Model

3.1.1 Links

Zenodo: https://zenodo.org/records/18177473

Github: https://github.com/mlsysops-eu/model-cluster-vm-management

3.1.2 Citation

Aslanidis, T., & Chatzopoulos, D. (2026). UCD Cluster VM Management Model (ONNX) (v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.18177473

3.1.3 More details

This repository contains a Deep Reinforcement Learning agent (trained using Maskable PPO) for optimizing Virtual Machine placement and lifecycle management. The model is exported as a platform-independent **ONNX** file for easy deployment.

It includes a complete inference pipeline that handles raw JSON infrastructure states, serializes them for the neural network, and translates the output into human-readable actions.

Project Structure

```
.
├── model/
│   ├── vm_management_agent.onnx   # The trained Neural Network
│   └── model_config.json           # Model Card & Configuration
├── src/
│   ├── inference_engine.py        # Wraps ONNX runtime for predictions
│   ├── infra_state_serializer.py  # Converts JSON tree -> 801-dim Float Vector
│   └── action_interpreter.py       # Converts Model Output -> Human-readable
Strings
├── demo.py                                # Main entry point to run all test
scenarios
├── generate_scenarios.py          # Script to generate test JSON files
├── requirements.txt                    # Python dependencies
└── README.md
```

Limitations & Model Constraints

This agent is specialized for a specific environment configuration. The model weights are tied to these boundary conditions:

Installation

It is recommended to use a virtual environment to keep dependencies isolated.

1. Clone the Repository

```
git clone https://github.com/tgasla/MLSysOps-VM-Management-Agent.git
cd MLSysOps-VM-Management-Agent
```

2. Create and Activate Virtual Environment

Linux / macOS:

```
python3 -m venv venv
source venv/bin/activate
```

Windows (PowerShell):

```
python -m venv venv
.\venv\Scripts\Activate.ps1
```

3. Install Dependencies

```
pip install -r requirements.txt
```

Quick Start

To verify the agent is working, simply run the demo script. This will automatically check for test scenarios (and generate them if missing) and run the agent against them.

```
python demo.py
```

Output Example:

```
--- 🧪 Testing Scenario: scenario_2.json ---
📥 Input Job Req: 8 cores
🤖 Raw Action:    [1 3 2 2]

✨ Action: Create VM
   -> Location: Host 3
   -> VM Type:  Large (ID: 2)
```

Python Usage Guide

If you want to integrate this agent into your own application, here is the standard workflow:

```
import json
from src.inference_engine import MLSysOpsVMManagementAgent
from src.infra_state_serializer import InfraStateSerializer
from src.action_interpreter import ActionInterpreter

# 1. Initialize Components
# (Paths are relative to where you run the script)
config_path = "model/model_config.json"
model_path = "model/vm_management_agent.onnx"

agent = MLSysOpsVMManagementAgent(model_path, config_path)
serializer = InfraStateSerializer(config_path)
interpreter = ActionInterpreter(config_path)

# 2. Load Infrastructure State
# You can pass a dictionary or a file path
with open("scenario_1.json", "r") as f:
    state_data = json.load(f)

# 3. Prepare Inputs
# Serialize the complex tree structure into the model's expected vector
infra_vector = serializer.serialize(state_data)
# Get the pending job requirements (scalar)
job_req = state_data.get("total_job_cores_waiting", 0)

# 4. Predict
# The agent returns a raw discrete vector (e.g., [1, 12, 0, 2])
raw_action = agent.predict(infra_vector, job_req)

# 5. Interpret
# Convert raw numbers into a meaningful string
explanation = interpreter.humanify(raw_action)

print(f"Agent Recommendation: {explanation}")
```

Test Scenarios

The `generate_scenarios.py` script creates three distinct situations to test the AI's decision-making:

- **Scenario 1:** 32 Hosts (Empty) + 0 Waiting Jobs.
  - Expectation: Do Nothing.
- **Scenario 2:** 8 Hosts (Empty) + 8 Cores Waiting.
  - Expectation: Create a VM (likely Large) on any available host.
- **Scenario 3:** 16 Hosts (Populated with VMs) + 0 Waiting Jobs.
  - Expectation: Destroy a VM.

Configuration & Model Card

The file `model/model_config.json` serves as the **Model Card**. It defines the exact input/output specifications and schema the model was trained on.

Important: You are not expected to change this file. The ONNX model's weights are permanently tied to these dimensions and definitions.

- vector_length: 801 (The fixed input array size).
- action_decoding: Maps the model's integer outputs to human-readable names (e.g., ID 0 -> "Small", ID 2 -> "Large").
- hardware_definitions: Defines the VM types/flavors the model learned to manage.

3.2 5G Jamming Attack Detection Model

3.2.1 Links

Zenodo: https://zenodo.org/records/18266543

Github: https://github.com/mlsysops-eu/model-5g-jamming-detection

3.2.2 Citation

Xu, J., Moheddine, A., Loscri, V., Brighente, A., & Conti, M. (2026). INRIA 5G Jamming Attack Detection - LSTM Model (ONNX) (v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.18266543

3.2.3 More details

This model was developed by INRIA as part of the **SHIELD framework**:

SHIELD is a research framework designed to evaluate machine-learning-based approaches for detecting jamming and interference in 5G networks under realistic conditions.

For a detailed description of the framework, methodology, and experimental setup, see the:

Paper: SHIELD: Scalable and Holistic Evaluation Framework for ML-Based 5G Jamming Detection

Source Code: https://github.com/mlsysops-eu/model-physical-layer-authentication

Authors

- Jiali Xu (Inria Centre at the University of Lille)
- Aya Moheddine (Inria Centre at the University of Lille)
- Valéria Loscrì (Inria Centre at the University of Lille)
- Alessandro Brighente (Department of Mathematics, University of Padova)
- Mauro Conti (Department of Methematics, University of Padova)

Purpose

This model performs real-time detection of 5G jamming and signal degradation events based on time-series telemetry collected from a mobile device.

It outputs a probability score indicating whether the observed signal behavior is normal or anomalous, where anomalies may correspond to intentional jamming, interference, or severe radio conditions.

The model is designed to run continuously on streaming data and make decisions at fixed inference intervals.

Training Data

The model was trained on time-series logs collected from a real Android device (OnePlus Nord 2T 5G) operating under both normal and degraded radio conditions.

This dataset is publicly available on Zenodo: INRIA SHIELD Framework Dataset - 5G Jamming Attack Detection

Training data characteristics:

- Collected from live **5G operation**
- Includes periods of:
  - Normal network behavior
  - Signal degradation
  - Interference-like patterns consistent with jamming scenarios
- Derived features are computed from **four signal sources**:
  - Extended Cell Signal Quality (ECSQ)
  - Thermal sensors
  - RF transmission power
  - Radio signal metrics (RSRP, RSRQ, SINR)

Each sample consists of a **fixed-length time window** (typically 10 seconds), where multiple aggregation functions are applied to raw signals to capture both short-term dynamics and variability.

Note: Since the model is trained on data from a specific device and chipset, performance may vary on other devices without retraining or domain adaptation.

Model Architecture

The model is a sequence-based neural network built around a **Long Short-Term Memory (LSTM)** backbone, making it suitable for learning temporal dependencies in time-series signal data.

High-level architecture:

- 2 stacked LSTM layers (hidden size: 50)
- Dropout (0.4) for regularization
- Sigmoid output layer for binary classification

For more information about the model architecture, check the `model/model_config.json`

The network processes a window of aggregated signal features over time and produces a single anomaly probability score for the entire sequence.

Preprocessing (RobustScaler normalization) is integrated directly into the ONNX model, ensuring consistent behavior between training and inference.

Model Specification

Inputs & Outputs

Input:

- **Data Type:** `float32`
- **Shape:** `[batch_size, seq_len, 60]`
  - `batch_size`: Number of samples (typically 1 for real-time inference)
  - `seq_len`: Sequence length, typically 10 time steps (10 seconds with 1s resampling)

  - `60`: Number of features derived from 4 signal sources with 5 aggregation methods each
- **Feature Composition:**
  - ECSQ (Extended Cell Signal Quality): 6 features × 5 aggregations = 30 features
  - Thermal sensors: 2 features × 5 aggregations = 10 features
  - RF transmission power: 1 feature × 5 aggregations = 5 features
  - Signal strength (RSRP, RSRQ, SINR): 3 features × 5 aggregations = 15 features
- **Aggregation Methods:** mean, max, min, std, amplitude
- **Preprocessing:** RobustScaler normalization (integrated in ONNX models, separate for PyTorch models)

Output:

- **Data Type:** `float32`
- **Shape:** `[batch_size, 1]`
- **Range:** `[0.0, 1.0]` (probability score via sigmoid activation)
- **Interpretation:**
  - Score > 0.5: **ANOMALY** (potential jamming detected)
  - Score ≤ 0.5: **NORMAL** (no jamming detected)

Limitations

- **Sequence Length:** The model expects time series data with a minimum sequence length. Shorter sequences may produce unreliable results.
- **Feature Count:** Input must have exactly 60 features. Missing or extra features will cause inference to fail.
- **Data Quality:** The model assumes continuous data streams. Large gaps or missing data may affect accuracy.
- **Domain Specificity:** Trained on specific Android device logs (OnePlus Nord 2T). Performance may vary on different devices without retraining.
- **Real-time Constraints:** Inference interval (default 5s) must be longer than preprocessing + inference time to avoid queue buildup.
- **Buffer Dependencies:** Requires all 4 buffer types (ecsq, thermal, erftx, signal) to be populated for accurate predictions.

Model Execution

1. Create and activate a python virtual environment

```
python3.13 -m venv venv
source venv/bin/activate
```

2. Install dependencies

```
pip install -r requirements.txt
```

3. Run Inference

```
import onnxruntime as ort
import numpy as np

# Load model
session = ort.InferenceSession("model/lstm_jd_model.onnx")
```

```
# Prepare input (example: batch=1, seq_len=10, features=60)
input_data = np.random.randn(1, 10, 60).astype(np.float32)

# Run inference
input_name = session.get_inputs()[0].name
output = session.run(None, {input_name: input_data})[0]

# Interpret result
probability = output[0][0]
prediction = "ANOMALY" if probability > 0.5 else "NORMAL"
print(f"Prediction: {prediction} (score: {probability:.4f})")
```

3.3 RF Fingerprinting Models for Physical Layer Authentication

3.3.1 Links

Zenodo: https://zenodo.org/records/18280776

GitHub: https://github.com/mlsysops-eu/model-physical-layer-authentication

3.3.2 Citation

Alla, I., Yahia, S., Loscri, V., & eldeeb, . hossien . (2026). INRIA RF Fingerprinting Model Collection for Physical Layer Authentication (ONNX) (v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.18280776

3.3.3 More Details

This repository contains a comprehensive collection of machine learning models developed by **INRIA**.

These models implement **Physical Layer Authentication (PLA)** using **RF Fingerprinting**. They are designed to secure wireless networks by identifying devices based on the unique physical characteristics (fingerprints) of their radio hardware, rather than just their digital credentials.

This record provides a "Model Zoo" covering various experimental scenarios (Trials), machine learning architectures, and feature selection techniques.

For a detailed description of the framework, methodology, and experimental setup, see the:

Paper: Robust Device Authentication in Multi-Node Networks: ML-Assisted Hybrid PLA Exploiting Hardware Impairments

Source Code: https://github.com/mlsysops-eu/model-physical-layer-authentication

Authors

- **Ildi Alla** (Inria Centre at the University of Lille)
- **Selma Yahia** (Inria Centre at the University of Lille)
- **Valéria Loscrì** (Inria Centre at the University of Lille)
- **Hossien Eldeeb** (University of Cambridge)

Purpose

These models perform **Binary Classification** to distinguish between trusted and malicious devices:

- **Authorized (Target):** The specific device allowed to access the network.
- **Rogue (Malicious):** An attacker, imposter, or unknown device trying to mimic a trusted node.

The models are exported in **ONNX format** (Opset 18) to ensure interoperability and are designed to run on edge devices for real-time authentication.

Repository Structure

Since this repository contains multiple models, the files are organized using the following directory structure:

```
/models
    ├── /trial_1                # Experimental Scenario 1
```

```
    │     ├── /random_forest
    │     │     ├── /pca         # Feature Selection Method: PCA
    │     │     │     └── model_device3.onnx
    │     │     └── /anova       # Feature Selection Method: ANOVA
    │     │           └── ...
    │     └── /xgb
    │           └── ...
    ├── /trial_2                 # Experimental Scenario 2
    └── ...
```

- **Trial:** Corresponds to specific experimental setups (specific sets of authorized vs. rogue devices).
- **Model Type:** The architecture used (e.g., xgb, svc, knn).
- **Feature Selection:** The method used to reduce the input vector size (e.g., pca, anova, mutual_info).
- **Filename:** Indicates the specific device ID the model was trained to authenticate (e.g., model_device3.onnx protects Device 3).

Training Data

The models were trained on raw I/Q signal data collected using a BladeRF AX4 Software Defined Radio (SDR) and GNU Radio. The dataset captures the "transient" phase of RF signals (the turn-on/turn-off characteristics), which contains the most distinctive hardware impairments used for fingerprinting.

This dataset is publicly available on Zenodo:

INRIA I/Q Signal Dataset for RF Fingerprinting and Physical Layer Authentication

Training data characteristics:

- **Source Hardware:** BladeRF AX4 (20 MHz sampling rate).
- **Signal Processing:** Transient detection, Low-pass filtering, Discrete Gabor Transform (DGT).
- **Input Features:** Statistical moments (Variance, Skewness, Kurtosis, Entropy) extracted from diagonal patches of the spectrogram.

Model Architecture

This collection includes the following architectures, all converted to standard ONNX format:

- **XGBoost (XGB):** Gradient boosting for high-performance classification.
- **Random Forest (RF):** Ensemble learning method using decision trees.
- **Support Vector Classifier (SVC):** Polynomial kernel SVM.
- **K-Nearest Neighbors (KNN):** Instance-based learning.
- **Logistic Regression:** Linear model for baseline comparison.

Specific hyperparameters (e.g., number of trees, kernel type) for each architecture can be found in the models/model_config.json file included in this record.

Model Specification (Common Interface)

All models in this collection share the same input/output interface specifications:

Inputs

- **Data Type:** float32
- **Shape:** [batch_size, n_features]
- **batch_size:** Number of samples (typically 1 for real-time inference).
- **n_features: Dynamic**. This depends on the specific model file selected (e.g., a PCA model might expect 20 features, while an ANOVA model might expect 50).
- *Note*: You must check the expected input shape of the specific .onnx file before feeding data.

Outputs

- **Data Type:** int64 (Label) and float32 (Probabilities)
- **Shape:** [batch_size, 1]
- **Interpretation:**
  - **Label 0: ROGUE / MALICIOUS DEVICE** (Access Denied)
  - **Label 1: AUTHORIZED / TRUSTED DEVICE** (Access Granted)

Limitations

- **Feature Dependency:** The input must be a feature vector extracted using the specific Gabor Transform & Statistical parameters defined in the paper. Raw I/Q samples cannot be used directly.
- **Device Specificity:** A model named model_device3.onnx is trained only to recognize Device 3. It will treat all other devices (even other trusted ones) as "Rogue" relative to Device 3.

Usage Demo

To run any model from this collection, use the provided inference_demo.py script.

1. Setup Environment

```
python3.13 -m venv venv
source venv/bin/activate
pip install -r requirements
```

2. Run Inference

Select a specific model file from the folder structure and run:

```
import onnxruntime as ort
import numpy as np

# 1. Select your model file
model_path = "./models/trial_1/xgb/pca/model_device3.onnx"
session = ort.InferenceSession(model_path)

# 2. Check how many features this specific model needs
input_meta = session.get_inputs()[0]
n_features = input_meta.shape[1]
print(f"Selected model expects {n_features} features.")
```

```
# 3. Generate input (Replace with actual calculated features)
# Shape: [1, n_features]
dummy_input = np.random.rand(1, n_features).astype(np.float32)

# 4. Run Inference
outputs = session.run(None, {input_meta.name: dummy_input})
predicted_label = outputs[0][0]

# 5. Interpret
if predicted_label == 1:
    print("✅ ACCESS GRANTED: Authorized Device Detected")
else:
    print("🚨 ALERT: Rogue/Malicious Device Detected")
```

3.4 SkyFlok Latency Prediction Models

3.4.1 Links

Zenodo: https://zenodo.org/records/18288840

GitHub: https://github.com/mlsysops-eu/model-storage-gateway-speed-prediction

3.4.2 Citation

University College Dublin, & Chocolate Cloud. (2026). Chocolate Cloud SkyFlok Latency Prediction: Gradient Boosting Models (ONNX) (v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.18288840

3.4.3 More details

This repository contains a collection of machine learning models developed in collaboration between **University College Dublin (UCD)** and **Chocolate Cloud (CC)**.

These models are deployed within the **SkyFlok** Gateway component (hosted in London). They perform **Latency Prediction** to estimate the time required to retrieve a file from specific cloud storage backends.

By predicting download times based on temporal patterns and file size, these models enable the Gateway to intelligently route download requests to the fastest available storage region. This minimizes retrieval latency and optimizes network efficiency for the end user.

The models are exported in **ONNX format** (Opset 18) to ensure high-performance, low-latency inference within the real-time routing logic.

Purpose

These models perform **Regression** to predict a continuous value:

- **Input:** File size and detailed timestamps (Hour, Day, Time of Day).
- **Output:** Estimated Transfer Time (Latency) in milliseconds.

Repository Structure & Backend Mapping

Since latency characteristics vary between cloud providers, a separate model is trained for each storage backend. Use the table below to identify which model corresponds to which cloud provider/region.

Backend ID	Model Filename	Cloud Provider	Region	Location
4	model_backend_id_4.onnx	Google Cloud	europe-west1	St. Ghislain, Belgium
20	model_backend_id_20.onnx	AWS	eu-west-1	Dublin, Ireland
39	model_backend_id_39.onnx	Microsoft Azure	WEST EUROPE	Amsterdam, Netherlands
79	model_backend_id_79.onnx	OVH Cloud	GRA	Gravelines, France
137	model_backend_id_137.onnx	Exoscale	Geneva	Geneva, Switzerland
144	model_backend_id_144.onnx	Scaleway	Warsaw	Warsaw, Poland

Directory Layout

```
/models
    ├── model_backend_id_4.onnx            # Model for Backend ID 4
    ├── model_backend_id_20.onnx           # Model for Backend ID 20
    ├──       model_backend_id_39.onnx                    #       ...
    ├──                                          model_backend_id_79.onnx
    ├──                                         model_backend_id_137.onnx
    └──                                         model_backend_id_144.onnx
    └── model_config.json          # Hyperparameters & metadata
```

Training Data

The models were trained on historical transfer logs collected from the SkyFlok platform.

The dataset captures real-world network performance metrics across different times of day and days of the week.

Features Used:

- **Workload:** File size (bytes).
- **Temporal:** Time of day (categorical: morning, afternoon, etc.), Hour, Minute, Second, Day of Week.

Model Architecture

These models utilize a Scikit-Learn Pipeline architecture, fully converted to ONNX:

1. Preprocessing: A ColumnTransformer that handles mixed data types (One-Hot Encoding for strings, pass-through for numbers).

2. Regressor: Gradient Boosting Regressor (2000 trees, Max Depth 12).

This architecture allows the model to capture complex, non-linear relationships between network congestion (time of day) and transfer speeds.

Model Specification (Common Interface)

All models in this collection share the same input/output interface specifications.

Inputs

The models accept a dictionary of standard Python lists.

Input Name	Type	Shape	Description	Example
time_of_day	String	[batch, 1]	Categorical time block	"morning", "night"
hour	Int64	[batch, 1]	Hour of the day (0-23)	14
minute	Int64	[batch, 1]	Minute of the hour	30
second	Int64	[batch, 1]	Second of the minute	45

day_of_week	String	[batch, 1]	Full day name	"Monday", "Friday"
size	Float32	[batch, 1]	File size in Bytes	100000.0

Outputs

- **Name:** predicted_latency
- **Data Type:** float32
- **Shape:** [batch_size, 1]
- **Unit:** Milliseconds (ms)

Limitations

- **Backend Specificity:** model_backend_id_4.onnx is trained specifically on the performance history of Backend #4. It should not be used to predict latency for other backends.
- **Historical Bias:** Predictions are based on historical trends; sudden network outages or unprecedented congestion events may result in prediction errors.

Usage Demo

To run any model from this collection, follow the steps below.

1. Setup Environment

```
python3.13 -m venv venv
source venv/bin/activate
pip install onnxruntime
```

2. Run Inference

```
python inference_demo.py
```

If you prefer to integrate it into your own application, here is the minimal code required:

```
import onnxruntime as ort
from datetime import datetime

# --- Configuration ---
backend_id = 4
file_size = 100000.0  # 100 KB

# --- Helper ---
def get_time_of_day(hour):
    if 5 <= hour < 12: return 'morning'
    elif 12 <= hour < 17: return 'afternoon'
    elif 17 <= hour < 21: return 'evening'
    return 'night'

# --- 1. Prepare Input ---
# Note: Double brackets [[ ]] create the required batch dimension (Batch=1)
t = datetime.now()
inputs = {
```

```
    'time_of_day': [[get_time_of_day(t.hour)]],
    'hour':        [[t.hour]],
    'minute':      [[t.minute]],
    'second':      [[t.second]],
    'day_of_week': [[t.strftime('%A')]],
    'size':        [[file_size]]
}

# --- 2. Run Inference ---
session = ort.InferenceSession(f"models/model_backend_id_{backend_id}.onnx")
result = session.run(None, inputs)

print(f"Predicted Latency: {result[0][0][0]:.2f} ms")
```

3.5 Smart Lamppost Noise Prediction Model

3.5.1 Links

Zenodo: https://zenodo.org/records/18290725

GitHub: https://github.com/mlsysops-eu/model-smart-lamppost-noise-prediction

3.5.2 Citation

Moti, M. H., Aslanidis, T., & Ubiwhere (Portugal). (2026). Ubiwhere Smart Lamppost Noise Prediction LSTM Model (ONNX) (v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.18290725

3.5.3 More details

This repository contains a machine learning model developed in collaboration between **University College Dublin (UCD)** and **Ubiwhere** as part of the **MLSysOps** project.

The model is deployed on edge devices within **Smart Lampposts** in Aveiro, Portugal. It performs **Noise Level Prediction** to estimate future environmental noise levels based on real-time traffic and pedestrian activity.

By predicting noise levels in advance, this model enables proactive urban management, such as dynamic lighting adjustment or alerting city operators to potential noise pollution events before they escalate.

The model is exported in **ONNX format (Opset 18)** to ensure high-performance, low-latency inference on edge hardware (e.g., NVIDIA Jetson).

Purpose

This model performs Time-Series Regression to predict a continuous value:

- **Input:** A sequence of past 30 readings (Noise Level, Cars, Motorcycles, People).
- **Output:** Predicted Noise Level (dB) for the next time step.

Repository Structure

The repository provides the trained model and its configuration for easy deployment.

```
.
├── inference_demo.py      # Full inference script (loads config and model)
├── model/                 # Directory containing the ONNX model and config
│   ├── noise_model.onnx
│   └── model_config.json
├── requirements.txt       # Python dependencies
└── README.md              # Project documentation
```

Training Data

The model was trained on real-world sensor data collected by Ubiwhere from a **Smart Lamppost** installed in Aveiro, Portugal.

The Smart Lamppost is a modular urban infrastructure equipped with video and sound sensors (camera and microphone) to capture environmental and traffic-related data.

Time Period: 2025-08-22 to 2025-08-29

The complete training dataset is publicly available on Zenodo: Ubiwhere Smart Lamppost Dataset

Features Used

The model uses a multivariate approach, correlating traffic density with noise levels:

1. Noise Level (Target): Measured environmental noise (dB).

2. Car Detections: Count of cars detected.

3. Motorcycle Detections: Count of motorcycles detected.

4. Person Detections: Count of pedestrians detected.

Note: Other metrics present in the raw dataset (CPU Usage, Jetson Energy, Temperature, etc.) were excluded to focus purely on the noise-traffic relationship

Model Architecture

This model utilizes a Multivariate Long Short-Term Memory (LSTM) network, fully converted to ONNX:

1. Input Layer: Accepts a sequence of shape (Batch, 30, 4) (30 time steps, 4 features).

2. LSTM Layers: Two stacked LSTM layers with 64 hidden units each to capture temporal dependencies.

3. Output Layer: A Linear layer that maps the final hidden state to a single scalar prediction (Noise Level).

This architecture allows the model to understand "inertia" (e.g., noise tends to stay high) and "causality" (e.g., a spike in cars leads to a spike in noise).

Model Specification

Inputs

The model accepts a single tensor representing a history window.

Input Name	Shape	Type	Description
input	[batch_size, 30, 4]	float32	Normalized history of the last 30 time steps.

Feature Order (Last Dimension):

1. Noise Level
2. Car Detections
3. Motorcycle Detections
4. Person Detections

Outputs

Output Name	Shape	Type	Description
input	[batch_size, 1]	float32	Normalized predicted noise level for the next step.

Limitations

- **Normalization Required:** The model expects input values normalized between 0 and 1. Raw sensor data must be scaled using the min/max values found in model_config.json before inference.
- **Location Specific:** This model is trained on data from a specific street in Aveiro. Deploying it in a different environment (e.g., a highway or a quiet park) may require fine-tuning.

Usage Demo

To run this model, follow the steps below.

1. Setup Environment

```
python3.13 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

2. Run Inference Script

Alternatively, you can directly run the script included in this Zenodo record to see a demonstration:

```
python inference_demo.py
```

3.6 Drone Deployment Prediction Model

3.6.1 Links

Zenodo: https://zenodo.org/records/18299548

GitHub: https://github.com/mlsysops-eu/model-drone-deployment-prediction

3.6.2 Citation

Chouliaras, A., Aslanidis, T., & Augmenta (acquired by CNH Industrial). (2026). Augmenta Drone Deployment Prediction Model (ONNX) (v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.18299548

3.6.3 More details

This repository contains a machine learning model developed by **University College Dublin (UCD)** for **Augmenta** (acquired by **CNH Industrial**) as part of the **MLSysOps** project, focusing on drone deployment prediction.

The model predicts the **should_fly** signal for drone operations, leveraging temporal sensor and flight data to anticipate deployment needs ahead of time. This enables proactive drone management, accounting for operational delays and improving decision-making in real-world scenarios.

The model is exported in **ONNX format (Opset 15)** for efficient inference on edge or cloud devices.

Purpose

This model performs Time-Series Classification to predict a binary signal:

- **Input:** A vector of features including temporal lagged variables and flight parameters (e.g., sensor fault probability, success rate, velocity, heading).
- **Output:** Predicted binary signal should_fly indicating if the drone should deploy or not at the forecast horizon.

Repository Structure

The repository provides the trained model and its configuration for easy deployment.

```
.
├── inference_demo.py      # Full inference script
├── model/                 # Directory containing the ONNX model and config
│   ├── drone_deployment_xgboost_model.onnx
│   └── model_config.json
├── requirements.txt       # Python dependencies
└── README.md              # Project documentation
```

Training Data

The model was trained on drone deployment data capturing sensor readings and flight parameters with temporal dependencies engineered as lag features.

- **Data Characteristics:** Time-stamped data with features such as sensor fault probability, success rate, processing performance, velocity, and heading.
- **Prediction Horizon:** Forecasts the should_fly signal several time steps ahead to mimic real deployment delays.

The complete training dataset is publicly available on Zenodo: [Augmenta Tractor-Drone Co-Robotics Dataset for Weed Detection]

Features Used

The model uses a rich feature set including:

- Temporal lags of:
  - sensor_fault_probability_1
  - success_rate
  - processing_performance
  - velocity
  - heading
  - Time metadata: year, month, hour
  - Time since last sensor fault and heading changes
  - Median fixed heading value

Model Architecture

This model utilizes an XGBoost classifier:

- **Boosting rounds:** 200 estimators
- **Max tree depth:** 5
- **Learning rate:** 0.1
- **Objective:** Binary logistic regression (binary classification)

The model captures complex temporal and non-linear relationships in sensor data to predict drone deployment signals accurately.

Model Specification

Inputs

The model accepts a single tensor representing the feature vector.

Input Name	Shape	Type	Description
float_input	[batch_size, 44]	float	Vector of features including lags & metadata

Feature Order (Last Dimension):

List of 44 feature names is included in the model/model_config.json under "features": {"names": [...]}.

Outputs

Output Name	Shape	Type	Description
label	[batch_size]	int64	Predicted class (0 or 1)
probabilities	[batch_size, 2]	float32	Class probabilities

Limitations

- **No scaling applied:** Model expects raw or preprocessed feature vectors matching training distributions.
- **Domain Specific:** Trained specifically for the drone deployment dataset and operational settings used; transfer to other drone types or environments may require retraining.

Usage Demo

Setup Environment

```
python3.13 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Run Inference Script

```
python inference_demo.py
```

This script loads the model and performs prediction on sample input data.

3.7 5G Latency Optimization Prediction Model

3.7.1 Links

Zenodo: https://zenodo.org/records/18303750

GitHub: https://github.com/mlsysops-eu/model-5g-network-optimization

3.7.2 Citation

Pazienza, A. (2026). NTT DATA 5G Latency Optimization RL Prediction Model (ONNX) (v1.0.0). Zenodo. https://doi.org/10.5281/zenodo.18303750

3.7.3 More details

This repository contains a machine learning model developed by **NTT DATA.**

This artifact provides an **ONNX export (opset 18)** of a **Deep Q-Network (DQN)** agent trained to **select the best data center (among 3)** for a client request in a 5G/MEC setting, optimizing latency-related outcomes (and incorporating carbon intensity as a feature).

Given the current state of three candidate data centers, the model outputs **Q-values** for each possible selection and chooses the data center with the highest Q-value.

Purpose

This model performs a Reinforcement Learning agent that predict a discrete value:

- **Input:** A normalized tensor representing the current state of **three candidate data centers**, shaped as *(3 × 10)*, where each row corresponds to a data center and each column represents a feature such as client identifier, resource utilization, network metrics, latency statistics, packet loss, and carbon intensity. The input is MinMax-scaled using parameters learned during training and includes a label-encoded client identifier.
- **Output:** A discrete action in {0, 1, 2} corresponding to the **selection of the optimal data center** among the three candidates, computed as the index with the highest predicted Q-value.

Repository Structure

A typical layout (as used in the accompanying repository/bundle):

- model/
  - 5g_latency_opt_dqn_model.onnx — ONNX model (DQN Q-network)
  - model_config.json — model metadata, I/O specs, feature order, preprocessing parameters
- src/
  - state_serializer.py — builds the model input tensor from JSON scenarios and applies preprocessing
  - minmax_scaler.py — lightweight MinMax scaling implementation (training-fitted parameters)

    - inference_engine.py — ONNXRuntime inference wrapper (argmax over Q-values)
    - action_interpreter.py — converts predicted action to a human-readable decision
- demo.py — end-to-end demo (JSON → preprocess → ONNX inference → decoded action)
- requirements.txt — minimal Python dependencies

Training Data

The model was trained on a tabular dataset containing per-data-center telemetry and network metrics. Each decision step groups **3 rows** (one per candidate data center) into a single observation.

Features Used

The training preprocessing:

- **MinMax normalization** for numeric features:
    - cpu_usage_percent
    - memory_usage_percent
    - disk_usage_percent
    - net_in_percent
    - net_out_percent
    - latency_avg
    - latency_mdev
    - lost_percent
    - carbon_intensity
- **Label encoding** for:
    - client_id
- Dropped columns:
    - start_time, end_time, net_in_absolute, net_out_absolute,
    - latency_min, latency_max

Important: the inference pipeline must reuse the **same MinMaxScaler parameters (data_min/data_max)** and the **same client_id encoding mapping** fitted during training.

Model Architecture

The exported ONNX model contains the **SB3 DQN policy Q-network** (MLP-based Q-function). The network maps a **(3×10)** observation (three candidate data centers, ten features each) to **Q-values** for the three discrete actions (select DC0/DC1/DC2).

At inference time, the recommended decision is argmax(Q-values).

Model Specification

Inputs

- **Name:** observation
- **Type:** float32
- **Shape:** (batch_size, 3, 10)

where:

- dimension 1 = candidate data centers (always 3)
- dimension 2 = feature vector per data center

Feature Order (Last Dimension)

The last dimension (size 10) follows this exact order:

1. client_id (label-encoded)
2. cpu_usage_percent (MinMax-scaled)
3. memory_usage_percent (MinMax-scaled)
4. disk_usage_percent (MinMax-scaled)
5. net_in_percent (MinMax-scaled)
6. net_out_percent (MinMax-scaled)
7. latency_avg (MinMax-scaled)
8. latency_mdev (MinMax-scaled)
9. lost_percent (MinMax-scaled)
10. carbon_intensity (MinMax-scaled)

Outputs

- **Name:** q_values
- **Type:** float32
- **Shape:** (batch_size, 3)
- **Meaning:** Q-values for the three actions:
  - action 0 → select Data Center 0 (Milan)
  - action 1 → select Data Center 1 (Rome)
  - action 2 → select Data Center 2 (Cosenza)

Limitations

- The model is trained for **exactly 3 candidate data centers**; input shape is fixed to (3,10).
- Correct behavior depends on **identical preprocessing**:
  - MinMax scaling must use **training-fitted** min/max values
  - client_id must be encoded using the **training-fitted** mapping (unknown IDs should be handled explicitly)
- Generalization outside the training distribution (different telemetry ranges, unseen client populations, different operational conditions) is not guaranteed.
- This model provides a decision policy but does **not** guarantee optimality; it should be validated in the target deployment setting before use.

Usage Demo

Setup Environment

Create a Python environment and install dependencies:

```
python -m venv .venv
source .venv/bin/activate # (Linux/macOS)
# .venv\Scripts\activate # (Windows)
pip install -r requirements.txt
```

At minimum, the runtime requires:

- onnxruntime
- numpy
- Pandas

Run Inference Script

```
python demo.py
```

The demo will:

1. Load a JSON scenario containing dataCenterStates (3 entries)
2. Apply preprocessing (MinMax scaling + client_id encoding)
3. Run ONNX inference via ONNXRuntime
4. Print the chosen data center index (argmax over Q-values)

3.8 Anomaly Detection Model

3.8.1 Links

Zenodo: https://zenodo.org/records/18302802

GitHub: https://github.com/mlsysops-eu/model-anomaly-detection

3.8.2 Citation

Delft University of Technology. (2026). TUD Anomaly Detection Model (ONNX). Zenodo. https://doi.org/10.5281/zenodo.18302802

3.8.3 More details

This repository contains a trained Autoencoder-based anomaly detection model developed in the context of the MLSysOps project (Machine Learning for Autonomic System Operation in the Heterogeneous Edge-Cloud Continuum), funded by the European Union's Horizon Europe research and innovation programme under grant agreement No. 101092912.

The model is exported in ONNX format for efficient inference on edge or cloud devices.

Purpose

This model performs **unsupervised anomaly detection** on node/VM telemetry metrics by learning to reconstruct normal observations.

- **Input:** A feature vector of telemetry metrics (float values), normalized with Min-Max scaling.
- **Output:** The reconstructed feature vector.
- **Anomaly score:** RMSE between input and reconstruction.
- **Decision rule:** anomaly if RMSE > threshold (threshold stored in model_config.json).

Repository Structure

The repository provides the trained model and its configuration for easy deployment.

```
.
├── demo.py                 # Inference script (ONNXRuntime)
├── model/
│   ├── autoencoder.onnx    # ONNX model
│   └── model_config.json   # Model configuration (features, normalization,
threshold)
├── requirements.txt        # Python dependencies
└── README.md               # Documentation
```

Training Data

The model was trained on telemetry data representing normal system behavior. The training dataset is not included in this Zenodo record unless explicitly provided in the uploaded files.

Important: The inference input must use the same feature ordering as the training data.

Features Used (Feature Order)

The expected feature order (last dimension of the input tensor) is:

1. cpu_0_idle
2. cpu_0_iowait
3. cpu_0_irq
4. cpu_0_nice
5. cpu_0_softirq
6. cpu_0_steal
7. cpu_0_system
8. cpu_0_user
9. cpu_1_idle
10. cpu_1_iowait
11. cpu_1_irq
12. cpu_1_nice
13. cpu_1_softirq
14. cpu_1_steal
15. cpu_1_system
16. cpu_1_user
17. cpu_2_idle
18. cpu_2_iowait
19. cpu_2_irq
20. cpu_2_nice
21. cpu_2_softirq
22. cpu_2_steal
23. cpu_2_system
24. cpu_2_user
25. cpu_3_idle
26. cpu_3_iowait
27. cpu_3_irq
28. cpu_3_nice
29. cpu_3_softirq
30. cpu_3_steal
31. cpu_3_system
32. cpu_3_user
33. memory_used_bytes
34. node_memory_Buffers_bytes
35. node_memory_Cached_bytes
36. node_memory_MemAvailable_bytes
37. node_memory_MemFree_bytes
38. node_memory_MemTotal_bytes

(These names must match model/model_config.json)

Model Architecture

This model is a fully-connected Autoencoder with ReLU activations:

- Encoder dims: feature_size -> int(0.75*feature_size) -> int(0.5*feature_size) -> int(0.25*feature_size) -> int(0.1*feature_size)
- Decoder dims: symmetric back to feature_size

Model Specification

Inputs

- **Input name:** x
- **Shape:** [batch_size, 38]
- **Type:** float32
- **Description:** Min-Max normalized feature vector

Preprocessing

- x_norm = (x - min) / (max - min)
- If a feature has max == min (constant feature in training), normalization must avoid division by zero (recommended: set the normalized feature to 0.0).
- Optionally clamp x_norm to [0, 1] if desired (configurable via model_config.json).

Outputs

- **Output name:** reconstruction
- **Shape:** [batch_size, 38]
- **Type:** float32
- **Description:** Reconstructed feature vector

Post-processing (Anomaly Detection)

- rmse = sqrt(mean((x_norm - reconstruction)^2)) per sample
- anomaly = 1 if rmse > threshold else 0
- threshold is stored in model/model_config.json

Limitations

- **Feature order & dimension are fixed:** Inputs must have exactly 38 features in the specified order.
- **Normalization is training-dependent:** Min/Max parameters are derived from the training data distribution; out-of-distribution inputs may yield unreliable anomaly scores.
- **Constant features:** Features with max == min require special handling during normalization (avoid division by zero).
- **ONNX output is reconstruction only:** The anomaly score/label is computed in the inference script.

Usage Demo

1. Setup Environment

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

2. Run Inference Script

```
python demo.py --model model/autoencoder.onnx --config model/model_config.json --
csv telemetry.csv --row 0
```

CSV Format Requirements

- CSV must include a header row.
- Numeric columns only (or ensure the numeric columns match the 38 features exactly).
- Column order must match the feature list and model_config.json.

3.9 VM Utilization and Remaining Lifetime Predictor Model

3.9.1 Link

GitHub: https://github.com/mlsysops-eu/model-peaklife-predictive-vm-management

Zenodo: https://zenodo.org/records/18422649

3.9.2 Citation

Bowen, S., D. Antonopoulos, C., Smirni, E., Ren, B., Bellas, N., & Lalis, S. (2026). # PeakLife (ONNX) — VM Utilization + Remaining Lifetime Predictor (Version 1.0.0) [Computer software]. https://github.com/mlsysops-eu/model-peaklife-predictive-vm-management

3.9.3 More details

This repository contains **PeakLife**, a lightweight neural model exported to **ONNX** for portable inference. Given the historic utilization information for a VM, PeakLife predicts:

- **Future CPU utilization**: **AvgCPU** and **MaxCPU** (normalized)
- **Remaining lifetime**: normalized remaining lifetime (and seconds via scaling)

The repo includes a minimal demo pipeline that loads a small CSV (demodata.csv), preprocesses it to the model's expected inputs, runs ONNX inference, and prints the results.

Project Structure

```
.
├── model/
│   ├── peaklife.onnx            # ONNX model
│   └── model_config.json        # Model card & Configuration
├── src/ (Optional)              # Helpers for data preparation
│   ├── DataUtil.py
│   └── prepare_demodata.py
├── demo.py                      # Main entry point for inference demo
├── demodata.csv                 # Demo dataset
├── requirements.txt             # Python dependencies for inference
└── README.md
```

Limitations & Model Constraints

This ONNX model is tied to a specific input contract and normalization:

Pre-set history length: input_length = 288 time steps by default.

- Forecast horizon: forecast_length = H.
- Signals: CPU utilization-only (AvgCPU, MaxCPU). Other resources (RAM/disk/net) are not modeled in this version.
- Normalization:
    - CPU values are expected in 0–100 in CSV and normalized by cpu_divisor (usually 100.0).

    - Remaining lifetime is normalized by max_lifetime_seconds from model_config.json.
- Output ranges: the model outputs are bounded to [0, 1] (Sigmoid heads), so it will not produce values outside this range.
- Data schema requirement for demo: demodata.csv must contain at least:
    - VMID
    - AvgCPU, MaxCPU
    - time_relative_seconds, lifetime_seconds
    - optionally TimeStamp and MaxCPU_so_far (if missing, demo falls back to last MaxCPU)

Installation

It is recommended to use a virtual environment to keep dependencies isolated.

1. Clone the Repository

```
git clone https://github.com/mlsysops-eu/model-peaklife-predictive-vm-management
cd model-peaklife-predictive-vm-management
```

2. Create and Activate Virtual Environment

Linux / macOS:

```
python3 -m venv venv
source venv/bin/activate
```

Windows (PowerShell):

```
python -m venv venv
.\venv\Scripts\Activate.ps1
```

3. Install Dependencies

```
pip install -r requirements.txt
```

Quick Start

Run the demo script.

```
python demo.py
```

Output Example:

```
--- Model Loaded ---
--- Demo Dataset Ready ---
--- PeakLife Demo ---
VMID: QdbZeJFmsJ3euIQ4lwW63NwFEP+QIirT4QbI0jEGr4dpkOet8p3iQSHAEm1gKWnR
inputs shape: (1, 288, 2) | aux shape: (1, 2)
pred_util shape: (1, 1, 2) | pred_life shape: (1, 1)

--- Utilization Prediction (AvgCPU, MaxCPU) ---
Pred (normalized): 3.912 , 5.861 | 0.039123 , 0.058608
True (normalized): 4.356 , 6.186 | 0.043559 , 0.061863
```

```
--- MAPE (avg, max, and combined) ---
util_mape(avg)=0.101832 | util_mape(max)=0.052609 | util_mape(combined)=0.077221

--- Remaining Lifetime Prediction ---
Pred remaining_lifetime_norm=0.893175 | Pred
remaining_lifetime_seconds=1544747.0s
True remaining_lifetime_norm=0.950043 | True
remaining_lifetime_seconds=1643100.0s

--- Lifetime MAPE ---
life_mape=0.059858
```

Configuration & Model Card

The file model/model_config.json serves as the Model Card and includes:

```
+ input_length

+ forecast_length

+ normalization.cpu_divisor

+ normalization.max_lifetime_seconds

+ input/output names and shapes (if you record them)
```

Important: The ONNX weights are tied to these dimensions and normalization constants.

3.10 ML Model for Predicting Job Placement Failures in Datacenter Clusters

3.10.1 Links

https://zenodo.org/records/18486169

https://github.com/mlsysops-eu/model-peaklife-predictive-vm-management

3.10.2 Citation

Patras, A., Syrivelis, D., & Terzenidis, N. (2026). MLNX ML Model for Predicting Job Placement Failures in Datacenter Clusters (1.0.0). Zenodo. https://doi.org/10.5281/zenodo.18486169

3.10.3 More details

This repository contains a trained binary classification model, exported to ONNX, that predicts whether a submitted job will fail or run successfully, given:

- the current state of a simulated datacenter cluster, and
- the resource request of an incoming job.

The model was developed within the MLSysOps research project and is intended for offline analysis, benchmarking, and integration into scheduling or admission-control pipelines.

Problem Statement

Modern large-scale clusters must decide whether to admit a job under uncertainty. Poor placement decisions can lead to job failures, even when aggregate resources appear sufficient.

In this work, a job failure can occur due to two distinct causes:

1. Insufficient compute resources (servers)
   If the cluster does not have enough free servers to satisfy the job request, failure can be determined through a simple availability check.
2. Insufficient or infeasible network connectivity (uplinks)
   Even when the total number of uplinks appears sufficient, the job may still fail because the required connectivity cannot be realized.

The latter case arises from the presence of a reconfigurable optical circuit switch (OCS) interconnecting leaf switches. Although OCS-based fabrics provide high bandwidth and flexibility, they introduce topological and temporal constraints: not all feasible matchings between leaf switches can be realized simultaneously, and reconfiguration constraints may prevent forming the necessary end-to-end paths.

As a result, uplink feasibility is not a simple counting problem, but a combinatorial one that depends on:

- the current circuit configuration,
- contention with existing jobs,
- and connectivity constraints imposed by the optical fabric.

Goal:
The model learns to predict whether a job will fail due to either compute insufficiency or network infeasibility, based on a snapshot of the cluster state and the job request.

Dataset

The model was trained and evaluated using a large-scale simulated dataset of job placement attempts.

Dataset repository (Zenodo): https://zenodo.org/records/18485585

The dataset repository provides:

- detailed system context,
- feature descriptions,
- ground-truth label semantics,
- statistical summaries,
- and usage examples.

Note: The dataset is released separately and is required to reproduce training or evaluation results.

Model Summary

- Task: Binary classification (job failure prediction)
- Framework: PyTorch
- Training orchestration: Ray Train / Ray Tune
- Export format: ONNX
- Inference backend: ONNX Runtime

The model consumes tabular features plus fixed-length vectors describing cluster utilization. Although the dataset distinguishes between different failure causes, the released model produces a binary output:

- not failed
- failed

Inputs and Preprocessing

- The model expects:
  - scalar numeric features describing cluster utilization and fragmentation,
  - fixed-length vector features representing server and uplink utilization.
- All preprocessing steps are defined in bundle.json, including:
  - feature column order,
  - normalization parameters (StandardScaler),
  - vector dimensions.

bundle.json must always be treated as the authoritative source of truth for model inputs.

Quick Start

Prerequisites

Install the required Python dependencies:

```
pip install numpy pandas pyarrow onnxruntime
```

or

```
pip install -r requirements.txt
```

Basic Usage

The `src/inference_runtime.py` script loads the ONNX model and preprocessing bundle, reads rows from a parquet file, and outputs predictions.

Run inference on the first 1000 rows

```
python src/inference_runtime.py \
  --onnx model/model.onnx \
  --bundle model/bundle.json \
  --parquet model/data.parquet \
  --n 1000
```

Output format (per row):

```
0    not failed    proba=0.023456
1    failed        proba=0.987654
2    not failed    proba=0.012345
```

Evaluate metrics (if ground-truth labels are available)

If your parquet file includes the ground-truth label column, you can compute evaluation metrics:

```
python src/inference_runtime.py \
  --onnx model/model.onnx \
  --bundle model/bundle.json \
  --parquet model/data.parquet \
  --n 1000 \
  --label_col l1_failed
```

Additional output:

Metrics on loaded rows:

```
accuracy=0.925980
precision=0.933392
```

```
recall=0.910949
f1=0.922034
```

Command-Line Arguments

The inference script (inference_runtime.py) supports the following command-line arguments:

Argument	Required	Default	Description
--onnx	Yes	—	Path to the model.onnx file
--bundle	Yes	—	Path to bundle.json containing preprocessing metadata
--parquet	Yes	—	Path to the input Parquet file
--n	No	1000	Number of rows to load from the Parquet file
--label_col	No	None	Name of the ground-truth label column (used only for metrics)

If --label_col is not provided, the script performs inference only and does not compute evaluation metrics.

Note: The exact feature column order and normalization parameters are stored in bundle.json.

Model Constraints

The released model is subject to several explicit constraints that must be respected for correct and meaningful use.

Fixed Input Schema

- The model expects a fixed set of input features:
  - scalar numeric features,
  - a server utilization bitmap of fixed length,
  - a leaf-switch utilization vector of fixed length.
- The exact feature order, normalization parameters, and vector lengths are defined in bundle.json.

Fixed Cluster Topology Assumption

- The model is trained assuming a specific cluster architecture:
  - 32 Scalable Units (SUs),
  - 32 servers per SU (1024 total servers),
  - 8 leaf switches per SU (256 total leaf uplinks).
- The server and uplink vectors are not dynamically resizable.
- Applying the model to clusters with:
  - different numbers of servers,
  - different SU layouts,
  - or different network topologies
    requires retraining or careful feature remapping and validation.

Binary Output Only

- Although the dataset distinguishes between:
  - server-related failures, and
  - uplink-related failures, the released model produces a binary output only:
    - failed
    - not failed
- The model does not indicate why a failure is predicted.

Probabilistic Predictions

- The model outputs a probability of failure, not a deterministic decision.
- The default classification threshold is 0.5, but:
  - different operational settings may require different thresholds,
  - threshold tuning should consider false-positive vs false-negative trade-offs.
- Predictions should be interpreted as risk estimates, not guarantees.
- It is intended to be used as a decision-support component, not as a standalone scheduler.

Users integrating this model into larger systems should ensure that all constraints above are satisfied and validated before relying on predictions in operational workflows.

3.11 Reinforcement Learning Policy Model for Dynamic FPGA DPU Configuration Selection

3.11.1 Links

GitHub: https://github.com/mlsysops-eu/model-peaklife-predictive-vm-management

Zenodo: https://zenodo.org/records/18494559

3.11.2 Citation

Patras, A., Lalis, S., Antonopoulos, C., & Bellas, N. (2026). UTH Reinforcement Learning Policy Model for Dynamic FPGA DPU Configuration Selection (0.1.0). Design, Automation, and Test in Europe (DATE). Zenodo. https://doi.org/10.5281/zenodo.18494559

3.11.3 More details

This repository provides a trained reinforcement-learning policy model, exported to ONNX, that selects an FPGA DPU configuration—defined by DPU size and number of DPU compute units (instances)—given an observation vector describing the current system and job context.

The model is intended for offline analysis, benchmarking, and decision support in FPGA-based ML inference pipelines, where selecting an appropriate DPU configuration is critical for performance and efficiency.

Problem Statement

Modern FPGA platforms support multiple DPU bitstream configurations, trading off parallelism, resource usage, and performance. Selecting an optimal configuration at runtime is non-trivial due to:

- varying model characteristics,
- changing workload intensity,
- contention between FPGA and ARM CPU resources,
- and complex performance–power trade-offs.

This repository addresses the problem of DPU configuration selection as a discrete decision-making task, learned via reinforcement learning from prior experimentation.

Goal: Given an observation vector describing the system/job state, predict the most suitable DPU configuration from a fixed action space.

Dataset

The policy model was trained using telemetry and experiment data collected from repeated ML inference runs on a Xilinx ZCU102 FPGA platform.

Dataset repository (Zenodo): https://zenodo.org/records/18494461

The dataset provides:

- experiment configurations,
- time-series telemetry,

- performance, power, and system metrics,
- and serves as the basis for training and evaluating this policy.

Model Summary

- Task: Discrete action selection (RL policy inference)
- Framework: Reinforcement Learning (trained offline)
- Export format: ONNX
- Inference backend: ONNX Runtime
- Output: Action index corresponding to a DPU configuration

The model outputs logits over a discrete action space. At inference time, the selected action is obtained via argmax.

Action Space (DPU Configurations)

Each model output corresponds to one action in a **fixed action space**. An action maps to:

- **DPU Size** (bitstream size), and
- **CU count** (number of DPU compute units / instances).

The **canonical mapping** is defined in:

```
action_interpreter.py → DEFAULT_ACTION_CONFIGS
```

This mapping is part of the **model contract** and must remain consistent with training and export.

Model Input/Output

Note: Training used normalized observations

All numeric features must be normalized to **[0, 1]** using fixed caps/bounds (see below). If you pass raw telemetry units directly (e.g., MB/s, W, CPU%), the model output will not be meaningful.

Observation layout (22 features total)

The model input is a single flat vector with this exact order:

System telemetry (16)

```
cpu_0, cpu_1, cpu_2, cpu_3
S0_read, S1_read, S2_read, S3_read, S4_read
S0_write, S1_write, S2_write, S3_write, S4_write
fpga_power, arm_power
```

Job static characteristics (5)

```
gmac, ldfm, ldwb, stfm, parameters
```

Constraints (1)

```
target_fps
```

Normalization caps / bounds (training contract)

System caps:

```
max_bw_mb_s = 7000
max_power_w = 20.0
```

Job static bounds:

```
canonical_depth: (18.0, 152.0) (not used in this release’s observation vector)
gmac: (0.3, 12.303)
ldfm: (0.792, 91.787)
ldwb: (3.326, 65.66)
stfm: (0.192, 61.125)
parameters: (3.5, 60.2)
```

Dynamic bounds (used for internal training instrumentation; not part of the released observation vector):

```
total_fps: (10.0, 100.0)
ppw: (1.0, 100.0)
```

Normalization formula:

```
x_norm = clip((x - low) / (high - low), 0, 1)
```

Outputs

Logits tensor of shape:

```
(1, num_actions) or (batch_size, num_actions)
```

The inference examples select:

```
action_idx = argmax(logits)
```

Usage

1. Prerequisites - install required dependencies:

```
onnxruntime
numpy
pandas
```

2. Action Interpretation

   action_interpreter.py provides the ActionInterpreter utility, which:

   - maps action_idx → (dpu_size, cu_count),
   - produces human-readable descriptions,

- enables consistent decoding of model outputs,
- supports action-space inspection and statistics.

Note: Changing the action ordering will silently invalidate predictions.

3. Programmatic Inference (Recommended)

   Use inference_wrapper.py to integrate the model into other scripts or services.

   - Load the ONNX model
   - Pass one or more **normalized** observation vectors
   - Receive decoded DPU configuration and confidence

The following example shows how to load the model and run inference on a single observation vector.

```
import numpy as np

from src.inference_wrapper import FPGARLInference, normalize_observation_row

# 1) Load the ONNX policy
model = FPGARLInference("fpga_rl_policy.onnx", verbose=True)

# 2) Option A (recommended): provide RAW features in dataset column names,
# then normalize using the same caps/bounds as training.
raw_features = {
    "cpu_0": 12.5,
    "cpu_1": 10.2,
    "cpu_2": 8.8,
    "cpu_3": 9.7,
    "S0_read": 1200.0,
    "S1_read": 1180.0,
    "S2_read": 150.0,
    "S3_read": 120.0,
    "S4_read": 90.0,
    "S0_write": 220.0,
    "S1_write": 210.0,
    "S2_write": 35.0,
    "S3_write": 28.0,
    "S4_write": 20.0,
    "fpga_power": 3.10,
    "arm_power": 1.85,
    "gmac": 1.82,
    "ldfm": 25.40,
    "ldwb": 14.20,
    "stfm": 9.10,
    "parameters": 11.70,
    "target_fps": 30.0,
}
```

```
# Normalize to the 22-feature vector expected by the model
obs_norm = normalize_observation_row(raw_features)          # shape: (22,)
result = model.predict(obs_norm)                            # runs ONNX
inference

print(model.format_result(result))
print("Action index:", result["action_idx"])

# 3) Option B: pass an already-normalized observation vector (float32, 22 dims).
# (Only do this if you *know* your normalization matches training.)
obs_norm_direct = obs_norm.astype(np.float32)
result2 = model.predict(obs_norm_direct, return_logits=True)
print("Top logit:", float(result2["logits"][result2["action_idx"]]))

# 4) Batch inference example (N x 22)
batch_raw = [raw_features, {**raw_features, "cpu_0": 35.0, "target_fps": 60.0}]
batch_obs = np.stack([normalize_observation_row(r) for r in batch_raw], axis=0)
# (N, 22)

batch_results = model.predict_batch(batch_obs)
for i, r in enumerate(batch_results):
    print(f"Sample {i}: {r['action_str']} (confidence={r['confidence']:.2%})")
```

CSV-Based Inference Demo

The `onnx_inference_example.py` provides an end-to-end example that:

- loads the ONNX model,
- reads observations from a CSV file,
- runs inference row-by-row,
- prints selected actions and summary statistics.

This is useful for offline replay, inspection, and experimentation.

```
python3 src/onnx_inference_example.py \
 --onnx model/fpga_rl_policy.onnx \
 --csv sample_data.csv \
 --out inference_results.csv
```

Model Constraints

- The action space is **fixed** and cannot be extended without retraining.
- The model produces **one discrete decision only** (no multi-objective output).
- Predictions assume consistent feature preprocessing and observation semantics.
- Inference is **stateless**; temporal dependencies are not modeled.

4 Conclusions

Deliverable D6.4 documents the successful finalization and public release of the MLSysOps Open Datasets and Models, representing the project's commitment to open science. By curating and publishing eight public datasets and eleven machine learning models, the consortium has provided a robust foundation for future research in autonomic system management within the cloud-edge continuum.

The utilization of Zenodo as a central repository has ensured that all project outputs adhere to the FAIR principles—making them Findable, Accessible, Interoperable, and Reusable for the broader scientific community.

By providing these models primarily in the ONNX format, the project ensures cross-platform compatibility, allowing researchers to deploy and test these solutions across varying hardware environments. Ultimately, the MLSysOps Zenodo Community serves as a permanent, citable archive that will continue to support the evolution of AI-controlled frameworks long after the project's formal conclusion.

END OF DOCUMENT