Bachelor of Science Thesis

CONTAINERIZED MACHINE LEARNING TRAINING WIZARD: DESIGN AND IMPLEMENTATION OF A DATA PROCESSING AND MODEL TRAINING PIPELINE

Clerides Christodoulos - Glafkos

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2026

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Containerized Machine Learning Training Wizard: Design and Implementation of a Data Processing and Model Training Pipeline

Clerides Christodoulos – Glafkos

Supervisor

Dr. Vasos Vasiliou

A thesis submitted in partial fulfillment of the requirements for the Bachelor of Science degree in Computer Science at the University of Cyprus.

May 2026

"I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research, and with full accountability on my part for the accuracy, integrity, and validity of all content."

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Dr. Vasos Vassiliou, for his guidance, support, and valuable feedback throughout the development of this thesis. His academic insight and encouragement played an important role in shaping both the direction and quality of this project.

Abstract

The increasing adoption of the Internet of Things (IoT) and Cyber-Physical Systems (CPS) has created a growing need for structured and reproducible machine learning workflows capable of supporting security analysis, anomaly detection, and malicious activity identification. Although machine learning techniques are widely applied to intrusion detection and telemetry-based anomaly detection, the practical process of preparing datasets, configuring experiments, executing models, storing artifacts, and visualizing results often remains fragmented and difficult to reproduce. This thesis addresses these challenges through the design and implementation of a containerized Machine Learning Training Wizard for IoT and CPS security datasets.

The proposed system integrates dataset preprocessing, dataset registration, experiment configuration, asynchronous training execution, artifact generation, and result visualization into a unified web-based platform. The implementation combines a Laravel backend API, a Vue.js frontend interface, Python-based preprocessing and training services, MySQL database storage, and Docker Compose containerization. Through a guided wizard workflow, users can select or register datasets, configure supervised classification or telemetry-oriented anomaly detection experiments, define algorithm parameters, submit training runs, and inspect generated results through a centralized interface.

The platform was evaluated through practical case studies involving a KDD-style network intrusion detection dataset and the Secure Water Treatment (SWaT) telemetry dataset. The supervised classification workflow demonstrates binary benign and malicious traffic analysis using algorithms such as Logistic Regression and Random Forest, while the anomaly detection workflow applies Isolation Forest to telemetry data and generates interactive time-series visualizations. The results confirm that the implemented system can successfully support end-to-end machine learning experimentation, including configuration tracking, metric generation, confusion matrix visualization, anomaly score inspection, and artifact management.

Overall, this thesis contributes a practical, modular, and extensible platform for guided machine learning experimentation in IoT and CPS security contexts. The system demonstrates how containerized deployment, modern web technologies, and structured workflow design can improve accessibility, reproducibility, and maintainability in machine learning-based security analysis.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1. Background

The Internet of Things (IoT) and cyber-physical systems (CPS) have become important components of modern computing environments. IoT refers to interconnected devices that collect, exchange, and process data through communication networks, while CPS combine computational elements with physical processes and real-world operations. These technologies are increasingly used in industrial automation, healthcare, smart infrastructure, transportation, and other environments where digital systems interact with the physical world [16].

The growth of IoT and CPS environments has introduced significant cybersecurity challenges. Many connected devices operate with limited computational resources, restricted storage, and long deployment lifecycles, making it difficult to apply traditional security mechanisms consistently. As a result, IoT systems are often exposed to weaknesses related to authentication, software updates, device management, communication security, and data protection [13].

Security risks in IoT and CPS environments are particularly important because attacks can affect not only digital services but also physical processes. Compromised sensors, actuators, industrial controllers, or networked devices may lead to incorrect measurements, operational disruption, privacy exposure, or unsafe system behaviour. Existing research on IoT vulnerabilities and CPS

security challenges shows that these environments are difficult to protect due to their heterogeneity, connectivity, and interaction with physical systems [6, 12, 31].

To address these challenges, machine learning has become increasingly relevant in IoT and CPS security research. Machine learning techniques can be used to analyse network traffic, telemetry data, and system behaviour in order to identify suspicious patterns, classify malicious activity, and detect anomalies. Intrusion detection systems and anomaly detection methods are therefore important research directions for improving the monitoring and analysis of security-related data in connected environments [10, 20].

Recent research on deep learning-based attack detection for CPS further demonstrates the importance of applying intelligent models to security monitoring tasks. Such approaches can learn complex patterns from telemetry and operational data, which makes them suitable for detecting abnormal behaviour in industrial and cyber-physical environments. However, their practical use also increases the need for structured preprocessing, model configuration, evaluation, and experimentation workflows [44].

At the same time, applying machine learning in this context is not only an algorithmic problem. A complete machine learning workflow requires dataset preparation, preprocessing, feature extraction, model configuration, training execution, evaluation, and result visualization. These stages are often handled through separate scripts, tools, and environments, which can make experimentation difficult to reproduce and maintain. Reproducibility has been identified as an important challenge in machine learning research, especially when experiments depend on specific software versions, datasets, configurations, and execution environments [32].

This thesis focuses on the design and implementation of a containerized machine learning training wizard for IoT and CPS security experimentation. The proposed system combines dataset preprocessing, model training, experiment configuration, and result visualization into a unified web-based platform. By using containerized services, the system aims to improve workflow organization, reproducibility, and accessibility for classification and telemetry-oriented anomaly detection experiments.

1.2. Motivation

The motivation for this thesis originates from the increasing need for intelligent and reproducible security analysis workflows in IoT and CPS environments. As connected systems continue to expand, the amount of telemetry and network traffic data generated by these environments also increases.

This makes manual inspection and traditional rule-based analysis more difficult, especially when the goal is to identify abnormal behaviour or malicious activity across heterogeneous datasets.

IoT and CPS security research highlights that these systems are subject to many risks because of their distributed nature, limited device capabilities, and interaction with physical processes. Security issues in such environments are not limited to data confidentiality but may also affect availability, reliability, safety, and operational continuity [6, 31]. Therefore, effective monitoring and analysis mechanisms are required in order to support the identification of suspicious activity and potential attacks.

Machine learning-based intrusion detection and anomaly detection techniques offer a promising direction for this problem. Intrusion detection systems can analyse network or system behaviour to identify malicious activity, while anomaly detection methods can detect unusual patterns that differ from expected behaviour [10, 20]. These techniques are especially relevant for IoT and CPS datasets, where both labeled attack traffic and telemetry-oriented time-series data may be used for security analysis.

Despite the potential of machine learning, building a complete experiment workflow remains challenging. Users often need to manually prepare datasets, configure algorithms, execute training scripts, manage generated artifacts, and interpret results. Existing studies on machine learning reproducibility show that ML experiments often depend on software versions, datasets, configurations, and execution environments, making consistent experimentation difficult [7].

Containerization provides a practical way to address part of this problem by packaging software dependencies and execution environments into isolated containers. Docker-based environments can improve portability and reduce dependency conflicts between machines, making experiments easier to reproduce and share [32]. This motivates the development of a platform that combines

machine learning experimentation with containerized deployment in a guided and accessible workflow.

1.3. Problem Statement

One of the main problems addressed in this thesis is the difficulty of moving from raw or processed IoT/CPS security datasets to structured and reproducible machine learning experiments. Security datasets may contain different formats, feature types, labels, timestamps, and preprocessing requirements. Without a consistent workflow, users must manually adapt scripts and configurations for each dataset, increasing the likelihood of errors and reducing reproducibility.

Another challenge is that machine learning workflows for intrusion detection and anomaly detection are often fragmented. Dataset preprocessing, model training, evaluation, artifact generation, and visualization may be implemented using separate tools that are not integrated into one coherent platform. This fragmentation makes it harder to track experiments, reproduce results, compare configurations, and inspect generated outputs effectively [32].

In addition, IoT and CPS security analysis requires support for different types of machine learning tasks. Supervised classification may be used when labeled benign and malicious samples are available, while anomaly detection is useful when identifying abnormal behaviour in telemetry or time-series data. Existing research shows that both intrusion detection and anomaly detection are important techniques for cybersecurity analysis, while deep learning-based CPS attack detection further demonstrates the need for suitable preprocessing and evaluation pipelines when applying ML to telemetry data [20, 44].

Therefore, there is a need for a system that simplifies the process of configuring and executing machine learning experiments for IoT and CPS security datasets. Such a system should support dataset preparation, guided experiment configuration, training execution, result generation, and visualization while maintaining reproducibility through consistent execution environments.

This thesis addresses this problem through the development of a containerized ML Training Wizard. The system is designed to provide a structured workflow for IoT security experimentation by integrating data preprocessing, orchestration, experimentation, database tracking, and visualizations into a single platform.

1.4. Aim and Objectives

The primary aim of this thesis is to design and implement a containerized machine learning training wizard for IoT and cyber-physical system security experimentation. The proposed system is intended to support classification and telemetry-oriented anomaly detection in a unified, reproducible web-based environment.

To achieve this aim, the thesis pursues the following objectives:

1. To design a modular architecture that integrates dataset preprocessing, experiment configuration, model training, evaluation, and visualization into a single workflow.
2. To implement supervised classification and telemetry-oriented anomaly detection workflows using IoT and CPS security datasets.
3. To provide a guided frontend interface that simplifies dataset selection, algorithm configuration, training execution, and result inspection.
4. To encapsulate the backend, frontend, database, and training services in containerized environments in order to improve portability, deployment consistency, and reproducibility [7].
5. To store experiment metadata, training configurations, execution statuses, and generated artifacts so that completed runs can be reviewed and compared.
6. To evaluate the developed platform through practical classification and anomaly detection case studies using representative security datasets.

1.5. Scope of the Thesis

The scope of this thesis is limited to the design, implementation, and evaluation of a prototype platform for machine learning-based IoT and cyber-physical system security experimentation. The work focuses on creating an integrated environment where supported datasets can be prepared, registered, selected, used for training, and inspected through generated evaluation outputs as discussed earlier.

The implemented system focuses on offline experimentation using static datasets. It does not perform real-time monitoring of deployed IoT devices, live traffic inspection, or online model

adaptation. The platform also does not aim to propose a new machine learning algorithm; instead, it applies existing machine learning methods within a structured and reproducible workflow.

The current version supports supervised classification and telemetry-oriented anomaly detection, while other learning tasks such as regression, clustering, forecasting, and advanced deep learning workflows remain outside the implementation scope. Similarly, the deployment environment is based on local Docker Compose orchestration and does not include Kubernetes, cloud-native scaling, managed MLOps services, or production-grade multi-user and multi-tenant functionality.

Therefore, the scope of the thesis is centred on demonstrating how a modular, containerized, and guided workflow can support reproducible machine learning experimentation for IoT and CPS security datasets, rather than delivering a complete production intrusion detection system.

1.6. Thesis Outline

The thesis is organized into six chapters:

Chapter 1 which is the introduction to this thesis giving background information, reason of conducting the research, the research problem, and the objectives of the research in an effort to justify the need for intelligent IoT security monitoring and visualization.

Chapter 2 is the literature review section of this thesis where literature on machine learning-based IoT and cyber-physical system security as well as existing monitoring solutions are described and analysed. It also outlines the technologies used in creating the ML Training Wizard such as Laravel framework, Vue.js, Python and Docker.

Chapter 3 provides an overview of the proposed system, requirement analysis, its workflow design and architecture. The chapter comprehensively discusses the architecture of a containerized machine learning training wizard for telemetry-oriented anomaly detection and classification workflows in IoT and CPS security environments.

Chapter 4 focuses on the implementation process of the developed system starting from environment set up and backend development, preprocessing pipelines and containerization as well as frontend development. This chapter further describes some of the practical aspects involved in creating the proposed system using current Web technologies.

Chapter 5 is dedicated to practical cases for anomaly detection and classification training runs using the developed system. It is mainly focused on the evaluation of the system usability, effectiveness and performance in fortifying ML-Based IoT Security Monitoring.

Chapter 6 presents the conclusion and highlights of the study, overall significance of the work, and limitations in the course of investigating the problems as well as recommendations for further research. Additionally, it raises questions about the consequences of solutions proposed in the context of intelligent IoT security and discusses possible prospects for further research and development.

By offering practical solutions to emerging issues challenging organizations' IoT ecosystems or security, this thesis will contribute to improving IoT security management and visualization.

Chapter 2

Literature Review

2.1 IoT and Cyber-Physical System Security

The Internet of Things and cyber-physical systems have become important parts of modern computing environments. IoT systems consist of connected devices that collect and exchange data through communication networks, while cyber-physical systems combine computation, communication, and physical processes. These systems are used in many sectors, including industry, healthcare, smart buildings, transportation, and critical infrastructure. Their purpose is to improve monitoring, automation, decision-making, and operational efficiency by connecting physical devices with software-based services [16].

Although IoT and CPS technologies provide many advantages, they also introduce major cybersecurity challenges. Many IoT devices are deployed with limited processing power, memory, and energy capacity. These limitations make it difficult to apply traditional security mechanisms in the same way they are applied in conventional computer systems. NIST highlights several cybersecurity capabilities that IoT devices should support, such as device identification, secure configuration, software updates, data protection, and vulnerability management [13]. However, in practice, many IoT deployments remain difficult to manage securely because of device

heterogeneity, long device lifecycles, inconsistent update mechanisms, and exposure to evolving attack surfaces [15]. ENISA similarly emphasizes that IoT security must be addressed across the full device and software lifecycle, from design and development through to deployment, maintenance, and decommissioning [12]. IoT cyber risk management also requires consideration of communication security, data protection, device lifecycle, and organizational risk exposure [23].

Cyber-physical systems introduce an additional level of risk because attacks may affect not only data and services, but also physical processes. A compromised sensor or actuator may lead to incorrect measurements, unsafe control decisions, or disruption of an industrial process. CPS security must therefore consider both cyber and physical dependencies, because failures or attacks in these systems may have real-world consequences [6, 19]. This makes monitoring and detection especially important in IoT and CPS environments.

Operational Technology is also relevant to this area. OT refers to systems that monitor and control physical equipment, processes, and industrial operations. Unlike traditional IT systems, OT environments often prioritize availability, safety, and reliability because interruptions may affect production, infrastructure, or human safety. NIST guidance on OT security emphasizes that such systems require protection strategies that consider operational continuity and physical process behaviour [38].

The security problem in IoT is not limited to a single device or protocol. IoT security spans multiple layers, including devices, communication networks, cloud services, applications, and user interaction [31]. From a data perspective, risks may also appear during data collection, transmission, storage, processing, and use, especially when sensitive or privacy-related information is generated by connected devices [18, 45]. This data-oriented view is relevant to the proposed platform, which focuses on preparing, analysing, and visualising security-related datasets rather than on securing individual devices.

2.2 Intrusion Detection Systems and Machine Learning-Based Detection

Intrusion Detection Systems are used to identify malicious or suspicious activity within networks and computing environments. Traditional IDS approaches are divided into signature-based and anomaly-based methods. Signature-based systems compare activity against known attack patterns, while anomaly-based systems attempt to detect behaviour that differs from what is considered

normal. Both approaches have advantages and limitations: signature-based methods are effective for known threats, but anomaly-based methods are more suitable for detecting unknown or evolving attacks [20].

Machine learning has become widely used in intrusion detection because it can learn patterns from data instead of relying on manually created rules, with earlier survey work showing the relevance of data mining and ML methods for cybersecurity intrusion detection [9]. In supervised learning, models are trained using labeled examples of normal and malicious behaviour, allowing them to classify new observations based on features extracted from network traffic or system activity. This approach is especially useful when datasets contain clear attack labels. More recent IDS studies also show that both machine learning and deep learning approaches are widely applied to network intrusion detection, although their performance depends strongly on the selected features, datasets, and evaluation setup [1].

Several studies have applied machine learning to IoT security monitoring. Meidan et al. proposed N-BaIoT, a system that uses deep autoencoders to detect IoT botnet attacks from device-level network behaviour [27]. Mirsky et al. proposed Kitsune, an online network intrusion detection system based on an ensemble of autoencoders capable of operating without prior knowledge of attack types [29]. Deep learning has also been investigated for distributed IoT attack detection, although such approaches may introduce additional complexity in training and deployment [11]. These systems demonstrate the value of machine learning for IoT security monitoring, but they are designed around specific detection architectures rather than providing a general experimentation workflow.

IDS performance also depends heavily on dataset quality and evaluation methodology. A model may produce strong results on one dataset but generalise poorly if the data distribution, feature extraction method, or attack types differ. Dataset quality, class imbalance, false positive rates, and evaluation consistency are all recognised challenges in IDS research [20]. This supports the case for structured experimentation platforms where preprocessing, configuration, and results are managed consistently rather than assembled differently for each experiment.

2.3 Anomaly Detection and Telemetry-Based Security Analysis

Anomaly detection is concerned with identifying observations that do not conform to expected behaviour [10]. In network security, anomaly detection techniques are commonly used to identify suspicious traffic patterns that may not match known attack signatures [2]. In IoT and CPS environments, anomalies may represent faults, misconfigurations, abnormal sensor behaviour, or cyberattacks. This makes anomaly detection useful when labeled attack data is not available or when unusual behaviour must be identified continuously over time.

Telemetry-based anomaly detection is especially relevant in cyber-physical and industrial environments. Telemetry data includes sensor readings, actuator states, timestamps, controller signals, and process measurements. Unlike tabular records, telemetry has a time-based structure where the order and timing of values carry meaning. Deep learning-based CPS attack detection approaches exploit these temporal and structural patterns to identify abnormal process behaviour [44]. Studies comparing different time-series anomaly detection models for industrial control systems further show that preprocessing, normalization, and windowing directly affect model performance. This suggests that detection quality cannot be separated from the quality of the overall data preparation workflow [21].

Isolation Forest is one practical algorithm for anomaly detection. It isolates unusual observations through random partitioning rather than constructing an explicit model of normal behaviour, making it well suited for detecting outliers in numerical data [24]. In this thesis, Isolation Forest serves as a lightweight baseline for telemetry-oriented anomaly detection workflows. Random Forest is used on the classification side as a supervised ensemble method that combines multiple decision trees to improve robustness and performance across tabular security datasets [8]. Together, these algorithms represent two distinct workflow paths; unsupervised anomaly detection and supervised classification, without requiring the thesis to propose new model architectures.

2.4 Security Datasets and Preprocessing

Datasets are central to machine learning-based security research, and dataset quality directly affects the validity of experimental results. Early benchmark datasets such as KDD Cup 99 became widely used for comparing IDS methods, but were later found to contain significant redundancy and distributional problems that can produce misleadingly high detection performance [39]. This

criticism demonstrated that dataset popularity does not guarantee experimental reliability and motivated subsequent efforts to establish more rigorous dataset design practices. Shiravi et al. proposed a systematic methodology for generating intrusion detection benchmarks, emphasizing the importance of realistic traffic profiles, complete capture, clear labelling, and representative attack scenarios [36].

Later datasets attempted to address these limitations. CICIDS2017 provides labeled flow-level features and a wider range of attack categories generated in a controlled environment [35]. UNSW-NB15 incorporates more contemporary traffic characteristics and attack types [30]. Both datasets are useful for supervised classification experiments, though they remain benchmark environments with their own assumptions regarding traffic generation, feature extraction, and attack coverage. Results obtained from them should therefore be interpreted as indicative rather than directly representative of real deployment conditions.

For cyber-physical system security, the Secure Water Treatment dataset is particularly relevant. SWaT is a water treatment testbed designed for industrial control system security research, containing process telemetry from sensors and actuators recorded under both normal and attack conditions [25]. Unlike network-only datasets, SWaT captures how attacks propagate through physical processes over time, making it especially suitable for telemetry-oriented anomaly detection workflows.

Preprocessing is required before most datasets can be used for machine learning. This includes handling missing values, encoding categorical features, normalising numerical values, mapping labels, balancing class distributions, and splitting data appropriately. Data quality in machine learning goes beyond basic cleaning: the suitability of transformations and feature representations for the intended learning task also matters [17]. For IoT and CPS security datasets in particular, preprocessing is non-trivial because datasets from different sources rarely share the same format, feature naming conventions, or label structures. This is one of the main reasons why the proposed system treats dataset registration and preprocessing as first-class workflow components rather than peripheral steps. It also supports the need for storing dataset metadata, task type information, and processed dataset outputs in a form that can be reused across training runs.

2.5 Machine Learning Workflows and Reproducibility

A complete machine learning workflow involves more than training a model. It includes dataset preparation, preprocessing, feature extraction, algorithm selection, parameter configuration, training execution, evaluation, artifact generation, and result visualization. When data dependencies, configuration settings, and system interactions are not managed carefully, ML systems accumulate hidden technical debt — complexity that makes experiments difficult to maintain, reproduce, or explain over time [34]. Delivering ML-enabled systems also requires broader coordination than conventional software development, spanning data management, model development, evaluation, deployment, and monitoring concerns [3].

Reproducibility is a major concern in machine learning research. Reliable experimentation requires clear documentation of datasets, code versions, hyperparameters, computational environments, and evaluation procedures [32]. Without this, the same experiment may produce different results when software versions, preprocessing scripts, or execution environments differ — a problem that becomes more acute when a single platform is used to run many experiments across different datasets and configurations.

MLflow addresses part of this problem by providing experiment tracking, run management, parameter logging, metric recording, and artifact storage [43]. These concepts are directly relevant to the design of the proposed system. However, MLflow is a general-purpose ML lifecycle tool. It does not provide a guided interface specifically oriented toward IoT and CPS security datasets, wizard-style experiment configuration, or telemetry-oriented visualization.

2.6 Containerization and Supporting Technologies

Containerization improves portability and consistency between execution environments. Docker packages software dependencies, runtime libraries, and system configuration into isolated units that can be executed more consistently across machines [7, 28]. For machine learning in particular, this addresses a common source of experimental failure: the inability to recreate the same software environment on a different host or at a later time.

In a multi-service platform, containerization is especially useful because different components — a frontend application, backend service, database, and training environment — often require different technology stacks and dependency versions. Isolating these concerns through containers

supports more disciplined system composition and makes repeated execution easier to manage. Containerization should nonetheless be understood as one part of a broader reproducibility strategy. Environment consistency reduces one source of variability, but reproducibility also depends on transparent dataset handling, configuration tracking, artifact persistence, and documented evaluation procedures [32].

2.7 Existing Monitoring and Machine Learning Workflow Solutions

Existing work in IoT and CPS security can be grouped into two broad categories. The first focuses on detection systems. N-BaIoT demonstrates botnet detection from IoT device network behaviour using deep autoencoders [27], while Kitsune demonstrates online intrusion detection using an ensemble of lightweight autoencoders [29]. SWaT-based anomaly detection studies show how process telemetry can be used to identify attacks affecting physical processes [21, 25]. These works are valuable for demonstrating the feasibility of machine learning-based security monitoring, but they are primarily detection systems. They do not provide a reusable workflow where users can register datasets, configure experiments, execute training runs, and inspect artifacts through a guided interface.

The second category consists of ML workflow platforms. MLflow provides experiment tracking, run management, and artifact logging, making it relevant to reproducible experimentation [43]. Apache Airflow organizes workflows as directed acyclic graphs of dependent tasks, which is well suited to scheduled data engineering pipelines but adds unnecessary orchestration complexity for a lightweight guided experimentation prototype [5]. Kubeflow supports reusable ML pipelines and scalable execution on Kubernetes, but targets production-oriented and cloud-native deployments rather than local, containerized research environments [22]. MLOps literature more broadly shows that these platforms address experiment tracking, pipeline orchestration, model management, and deployment — capabilities that are useful but designed for general and large-scale contexts [14].

The reviewed work therefore reveals a consistent separation between two streams: detection research that develops models and benchmarks for IoT and CPS security, and workflow research that develops reproducibility, orchestration, and lifecycle tooling. What remains less directly addressed is their intersection — a lightweight, guided, containerized platform specifically oriented toward IoT and CPS security datasets, where users can move from preprocessing to

training execution and result visualization without assembling the workflow manually from disconnected tools.

2.8 Summary and Identified Gap

The reviewed literature shows that IoT, CPS, and OT environments are difficult to secure because they combine connected devices, physical processes, heterogeneous data sources, and operational constraints [16, 38]. Monitoring and detection are therefore necessary complements to preventive controls, since attacks and abnormal behaviour may occur during normal system operation. Intrusion detection and anomaly detection are two important machine learning-based responses to this challenge. Supervised classification is useful when labeled benign and malicious data are available, while anomaly detection is appropriate when abnormal behaviour must be identified from telemetry or time-series data [10, 20]. In both cases, model performance depends not only on algorithm selection but also on dataset quality, preprocessing, configuration, and evaluation methodology.

Benchmark datasets including KDD Cup 99, CICIDS2017, UNSW-NB15, and SWaT provide useful reference points for security experimentation, but each carries assumptions and limitations that require careful preprocessing and metadata handling [25, 30, 35, 39]. Machine learning workflows more broadly require environment consistency, configuration tracking, artifact management, and reproducible evaluation procedures [7, 32, 34].

Existing detection systems and workflow platforms each address part of this problem. Detection research provides algorithms, benchmarks, and monitoring approaches, while tools such as MLflow, Airflow, and Kubeflow provide orchestration, tracking, and lifecycle management [43, 5, 22]. However, these two directions are typically treated separately, and there remains a need for lightweight platforms that combine them for guided IoT and CPS security experimentation.

The gap identified from the literature is therefore the need for a guided, containerised, and reproducible platform that integrates dataset preprocessing, experiment configuration, model execution, run tracking, artifact generation, and result visualisation for machine learning-based IoT and CPS security workflows. This gap motivates the ML Training Wizard presented in the following chapter.

Chapter 3

Overview of the System and its Design

3.1 Introduction

This chapter presents the design and architecture of the proposed containerized ML training wizard and discusses the main development decisions made during the implementation process. The system was designed as an end-to-end workflow environment for IoT and cyber-physical system security datasets, supporting dataset preparation and preprocessing, while simplifying model configuration, training execution, result evaluation and artifact visualization through a guided and centralized interface [3, 17, 34]. By Utilizing modern technologies such as Laravel, Vue.js, Python and Docker the platform provides functionality for dataset management, automated preprocessing pipelines, configurable training workflows and machine learning experiment monitoring within an integrated environment [7, 28].

3.2 System Requirements

The system requirements were defined based on the needs of reproducible machine learning experimentation in IoT and cyber-physical system security contexts [32, 34]. Since the proposed platform is designed as an end-to-end ML training workflow the requirements focus on dataset management, preprocessing, model configuration, training execution, result evaluation and

containerized deployment. The requirements are divided into functional and non-functional requirements

3.2.1 Functional Requirements

The functional requirements describe the core capabilities and services that the platform must provide in order to support the pipeline workflows.

1. Dataset Management: The platform must allow users to view available datasets and support the registration of processed datasets together with their metadata, including task type, features columns, target column, time column and label distribution [17, 26].
2. Dataset Upload and Processing: The system must support dataset upload and automated preprocessing workflows that transform raw or user-provided data into formats suitable for machine learning training evaluation [17].
3. Guided ML Workflow: The platform must provide a guided wizard interface that assist users through the main stages of an experiment, including task selection, algorithm selection and parameter configuration [43].
4. Training Run Execution: The system must allow users to launch training runs based on selected datasets, algorithms and training parameters. Training execution should be handled asynchronously to prevent long-running processes from blocking the application interface.
5. Support Multiple Learning Tasks: The platform should support supervised binary classification and telemetry-oriented anomaly detection, allowing the system to be applied to various IoT and cyber-physical system security scenarios [10, 20, 44].
6. Model and Parameter Configuration: Users must be able to select supported machine learning algorithms and configure relevant training parameter before execution.
7. Results and Artifact Visualization: The system must present experiment results in a clear and understandable form, including evaluation metrics, classification reports, confusion matrices, generated artifacts and time-series anomaly visualizations whenever applicable [43].

8. Experiment Tracking: The platform should store training run metadata, execution status, selected configuration parameters and generated artifacts for the experiments to be reviewed and evaluated.

9. API -Based Communication: Every layer should be containerized and the communication across layers should be through well-defined API endpoints enabling clear separation between the user interface, the orchestration layer and the model execution layer [7, 28].

3.2.2 Non-functional Requirements

The non-functional requirements describe the quality attributes and architectural characteristics that influence the design and operation of the platform.

1. Reproducibility: The system should be able to consistently reproduce training workflows by standardizing dataset preprocessing, storing run configurations, and generating persistent evaluation artifacts [32].

2. Usability: The platform interface should remain simple and guided, allowing users to execute machine learning experiments without manually running scripts or directly interacting with backend services.

3. Modularity: The architecture should separate the frontend, backend API, database, queue worker, preprocessing logic, and trainer service so that each component can be maintained or extended independently [3, 34].

4. Extensibility: The system should support future integration of additional datasets, learning tasks, algorithms, and visualization methods without requiring major architectural changes.

5. Scalability: The performance of the system should be scalable thus able to accommodate larger datasets, additional training runs and future model types by using a separate training service.

6. Performance: The system should process datasets and training runs efficiently enough for practical experimentation while keeping the user interface responsive.

7. Reliability: The platform should be able to handle failures during preprocessing or training execution and provide accurate error details through the logs for troubleshooting.

8. Portability: The system should operate in a plug-and-play nature across different environments through Docker-based containerization, reducing dependency issues [7, 28].
9. Maintainability: the overall architecture should be designed in a way that supports future development and system maintenance through clear separation between services [3, 34].

3.3 System Architecture

The ML training wizard focuses on machine learning experimentation for IoT and cyber-physical system security datasets and is built on a modular, layered architecture [34]. The system utilizes Docker for containerization, Vue.js for the frontend user interface, Laravel for the backend API and orchestration logic and Python for dataset preprocessing, and model training. This architecture allows the platforms to support dataset management, automated preprocessing, training execution and machine learning result visualization within a containerized full-stack workflow environment [7, 28].

3.3.1 High-Level Architecture

The system architecture consists of the following main components:

1. Dataset Input Layer: This layer represents the datasets used by the system, including IoT and cyber-physical system security datasets. The datasets act as the starting point of the machine learning workflow and are processed before training and evaluation [25, 35, 39].
2. Data processing Layer: This layer is responsible for transforming datasets into a standardized machine learning format. The processing pipeline performs operations such as cleaning, normalization, feature extraction and selection, train/evaluation splitting and metadata generation .
3. Database Layer: In this layer all the metadata, training configurations and features of the processed datasets are stored as well as execution statuses, generated metrics and artifact references. The database supports training run tracking and reproducibility within the platform [32, 43].
4. Backend API and Orchestration Layer: This layer contains the logic of the system and coordinates communication between the frontend, preprocessing pipeline, database, queue

worker, and trainer service. It handles dataset management, training run creation, validation and result retrieval.

5. Asynchronous Job Layer: This layer handles long-running operations such as preprocessing and training execution. Training runs are dispatched to queue workers to prevent blocking the user interface during execution.

6. Training Execution Layer: This layer is responsible for machine learning model execution and evaluation. The trainer service loads processed datasets, executes selected model and generates outputs such as metrics, reports, confusion matrices and time-series anomaly visualizations [10, 20].

7. Frontend Application layer: The user interacts with the system through the Graphical User Interface. It is composed of the user interface, which is the graphical layer of an application that users directly interact with. The interface includes the guided workflow for dataset selection, training configuration, run monitoring, and result visualization.

8. Containerization and Deployment Layer: This layer manages the deployment of the different system services using Docker Compose. Containerization improves portability, reproducibility and dependency management across different environments [28].

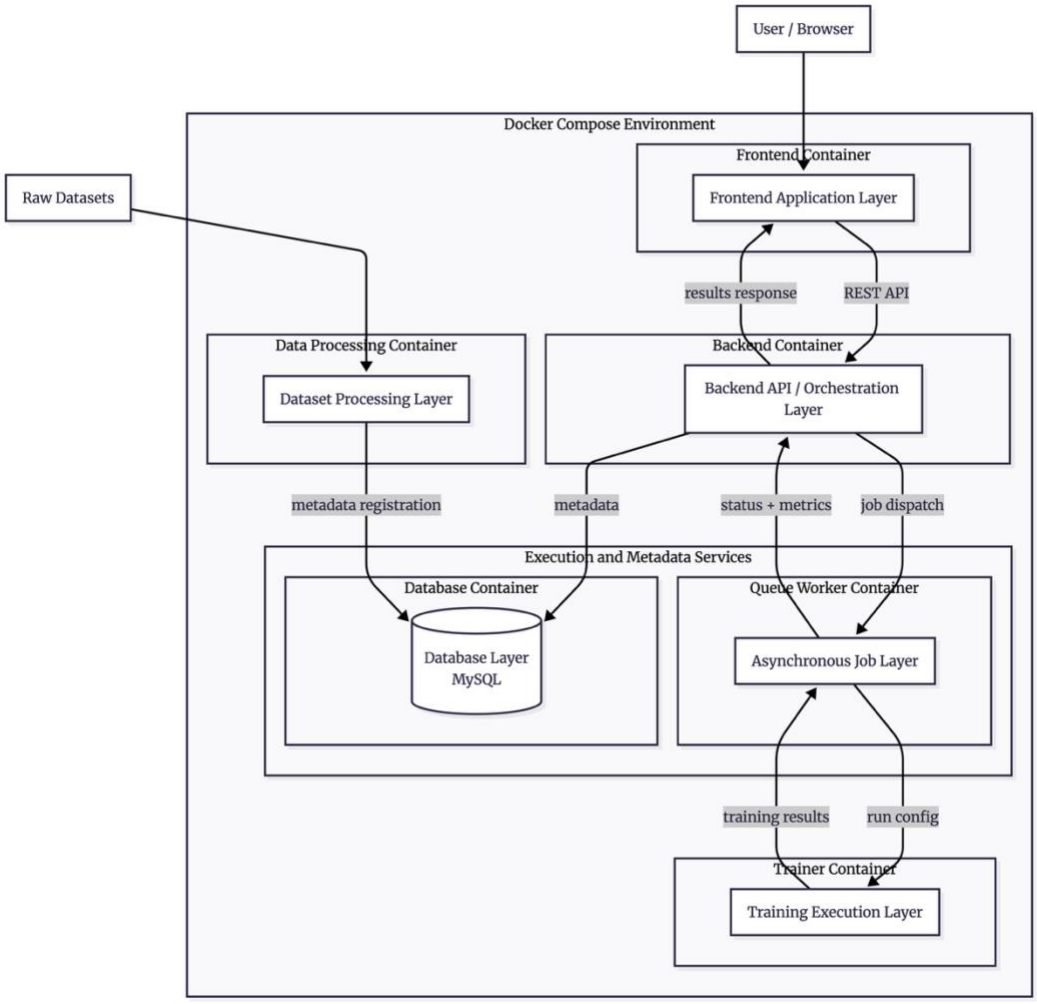

Figure 1: Containerized architecture of the proposed ML Training Wizard

3.3.2 Dataset Input Layer

This first layer represents the starting point of the machine learning workflow and handles raw IoT and cyber-physical system security datasets used for training and evaluation tasks. This layer includes:

1. Dataset Sources: Supports datasets collected from public repositories, research datasets, and prepared local files.
2. Dataset Organization: Organizes dataset files into a consistent structure used by the preprocessing and training pipeline.

3. Dataset Validation: Performs basic validation checks to ensure datasets follow the required format before processing.

The purpose of this layer is to provide controlled and reproducible input data for the machine learning workflow [17].

3.3.3 Data Processing Layer

This layer is responsible for preparing datasets for machine learning training and evaluation. It includes:

1. Data Cleaning: Handles missing values, invalid records, and inconsistent dataset structures.
2. Feature Processing: Performs feature selection, normalization, and label mapping operations.
3. Time Column Handling: Preserves timestamp information for telemetry-oriented datasets when required.
4. Dataset Splitting: Generates training and evaluation datasets together with metadata files.

The preprocessing pipeline transforms different datasets into a standardized format that can be used repeatedly and consistently by the training workflow [17, 26].

3.3.4 Database Layer

The database layer stores the structured information required for dataset management and training run tracking. This layer includes:

1. Dataset Metadata Storage: Stores dataset information such as task type, feature columns, target columns, and label distribution.
2. Training Run records: Stores selected configurations, execution status, metrics, timestamps, and generated artifacts.
3. Experiment Tracking: Allows previous training runs to be reviewed and reproduced [43].
4. Status Monitoring: Maintains the current state of each training run during execution.

The database stores metadata and artifact references while large dataset files are managed separately by the platform.

3.3.5 Backend API and Orchestration Layer

Implemented using Laravel, the backend API and orchestration layer acts as the central coordination component of the platform. This layer includes:

1. REST API Endpoints: Handle dataset retrieval, dataset upload, training execution, and result access.
2. Request Validation: Validates selected datasets, task types, algorithms, and training parameters.
3. Training Run Management: Stores training configurations and manages execution status updates.
4. Job Dispatching: Dispatches preprocessing and training jobs to the asynchronous execution layer.

This layer coordinates communication between the frontend, database, preprocessing pipeline, queue worker, and trainer service.

3.3.6 Asynchronous Job Layer

The asynchronous job layer is responsible for handling long-running operations outside the normal request lifecycle. This layer includes:

1. Queue Workers: Execute preprocessing and training jobs asynchronously.
2. Job Coordination: Prepares and dispatches training configurations to the trainer service.
3. Status and Error Handling: Updates execution status and stores error information during failures.

This design prevents the frontend interface from blocking during training execution and improves workflow reliability.

3.3.7 Training Execution Layer

The training execution layer performs machine learning training and evaluation tasks. This layer includes:

1. Model Execution: Executes the selected machine learning models using Python-based training services.
2. Evaluation Generation: Produces metrics, reports, confusion matrices, and anomaly outputs.
3. Artifact Generation: Generates files and visual outputs used by the frontend visualization layer.
4. Execution Isolation: Separates machine learning dependencies from the backend application environment.

This separation improves modularity and allows the training pipeline to evolve independently from the web application layer [3, 34].

3.3.8 Frontend Application Layer

The frontend application layer is implemented using Vue.js and provides the graphical user interface of the platform. This layer includes:

1. Guided Workflow Interface: Assists users during dataset selection, model configuration, and training execution.
2. Dataset Browsing: Displays available datasets and their metadata.
3. Run Monitoring: Allows users to monitor execution progress and review completed runs.
4. Result Visualization: Displays metrics, reports, confusion matrices, and anomaly visualizations.

The frontend communicates with the backend layer to retrieve data and display generated results.

3.3.9 Containerization and Deployment Layer

The containerization and deployment layer manages the execution of the platform services using Docker Compose. This layer includes:

1. Service Containerization: Runs the frontend, backend, database, queue worker, and trainer services in isolated containers.
2. Environment Reproducibility: Ensures consistent execution environments across different machines [7, 32].
3. Dependency Management: Separates application and machine learning dependencies into independent services.

The deployment environment consists of five separate containers, one for each component of the system: the frontend application, backend API, MySQL database, queue worker, and Python trainer service. Containerization improves portability, maintainability, and reproducibility within the platform [28].

3.4 System Workflow Design

The system workflow follows these steps:

1. Raw datasets are registered and stored within the platform for preprocessing and training purposes.
2. The preprocessing pipeline prepares the datasets by performing cleaning, normalization, feature selection, metadata generation, and train/evaluation splitting.
3. The processed dataset metadata is stored in the database and made available through the backend API layer.
4. The user selects the dataset, task type, algorithm, and training parameters through the frontend wizard interface.
5. The backend validates the selected configuration and creates a training run record.
6. The training job is dispatched asynchronously through the queue worker and forwarded to the trainer service for execution.
7. The trainer service executes the selected machine learning model and generates metrics, reports, confusion matrices, and visualization artifacts.
8. The generated outputs are stored and later retrieved through the backend API layer.

9. The frontend application retrieves the generated results and displays them through the visualization interface.

This workflow architecture separates preprocessing, execution, storage, and visualization into independent stages, improving maintainability, scalability, and reproducibility within the platform [3, 34].

3.5 Scalability and Future Extensibility

The architecture of the platform was designed in a way that allows new dataset insertion in the pipeline as well as new machine learning workflows. The concept of containerization was adopted in order new preprocessing pipelines, ML algorithms and training types to be easily and independently integrated in the system without affecting the overall workflow [28]. In addition, asynchronous job execution is used during preprocessing and training operations to keep the system responsive during long-running tasks, larger training workloads and database migrations.

3.6 Security and Access Considerations

Security was considered during the design of the platform to ensure controlled access to datasets, preprocessing operations, and training execution. Dataset uploads and user inputs are validated before processing in order to reduce invalid configurations and unsupported operations [13]. Sensitive configuration values such as database credentials and service settings are managed through environment variables. Docker containerization contributes by isolating the frontend, backend, database, preprocessing pipeline, and trainer service into separate environments [7]. This separation improves reliability and limits direct interaction between system components.

3.7 Conclusion

This chapter presented the design and architecture of the ML training wizard together with the main workflow and system components. The chapter discussed the system requirements, architecture layers, workflow design, scalability considerations, security measures, and future extensibility of the platform.

The next chapter presents the implementation details of the system and explains the technologies and components used during development.

Chapter 4

System Implementation

4.1 Introduction

This chapter explains how the ML Training Wizard was implemented and deployed. The system is developed with modern web technologies: Laravel was selected for the backend services, Vue.js is used for the development of the frontend interface and Python for dataset preprocessing and machine learning training workflows. Each framework operates within its own dedicated container, and Docker Compose is used for connecting all the services together and handling service orchestration. This approach allows for a scalable, responsive, maintainable and modular solution capable of handling the complex requirements of the ML training pipeline [7, 34, 28].

4.2 Development Environment and Repository Structure

The repository of the system was organized and categorized so each directory corresponds to a specific part or service of the ML training wizard. The structure separates the frontend, backend, trainer service, preprocessing scripts, datasets, and infrastructure and deployment configurations, allowing better orientation and organization of the components of the system.

The repository is divided into four main areas.

- The `apps` directory: Contains the main application and execution services of the platform.
- The `datasets` directory: Acts as the shared workspace used by the preprocessing and training workflow.
- The `infra` directory: Contains the infrastructure and deployment configuration, mainly the Docker Compose file used to start the local stack.
- The `scripts` directory: Contains automation and utility scripts used for starting the environment, processing datasets and validating the workflow.

The figure below presents the main repository structure of the system:

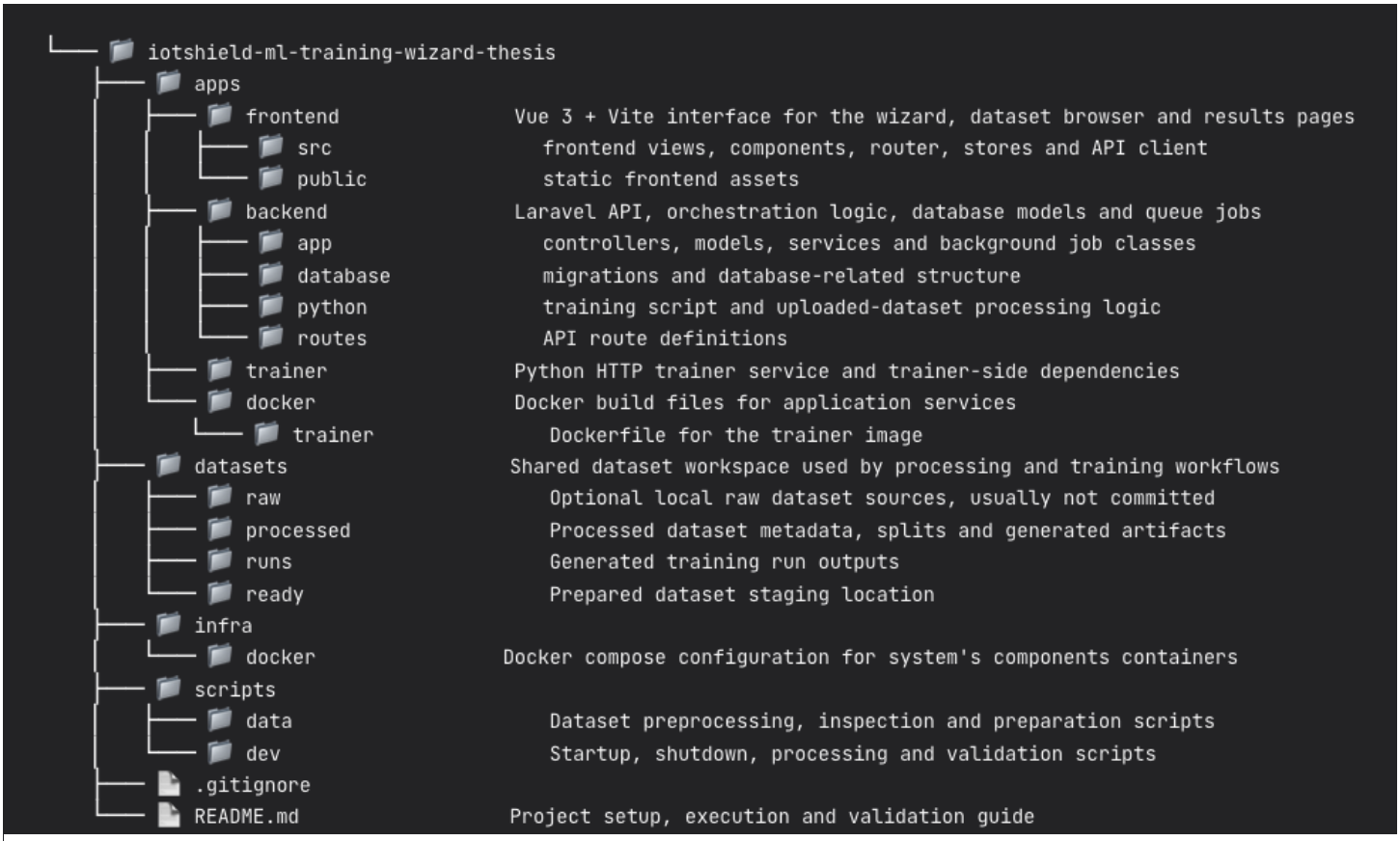

Figure 2: System's repository structure

The `apps/backend` directory contains the Laravel backend implementation, including controllers, queue jobs, database models, migrations, and API services. The `apps/frontend` directory contains the Vue.js frontend wizard and result visualization interface, while the `apps/trainer` directory contains the Python-based trainer service responsible for machine learning execution. Dataset preprocessing scripts are located in the `scripts/data` directory, while deployment configuration files are stored in the `infra/docker` directory. The datasets directory is used as the shared workspace for processed datasets, metadata, generated metrics, reports, confusion matrices, and visualization artifacts.

Docker Compose runs the development environment from the repository root, starting all required services together and reducing host-level dependency setup across different machines [7, 28].

4.3 Backend Implementation

On the Backend of the system, Laravel 12.X is used. Laravel was selected because it provides built-in support for routing, request validation, queues, migrations, and structured REST API development [37]. The backend exposes REST API, coordinates preprocessing and training execution, manages asynchronous jobs, stores workflow metadata and communicates with the trainer container.

The implementation of the backend framework is located in the `apps/backend` directory and is organized around controllers, services, queue jobs, models and API routes.

The structure of the backend directory is presented in the figure below:

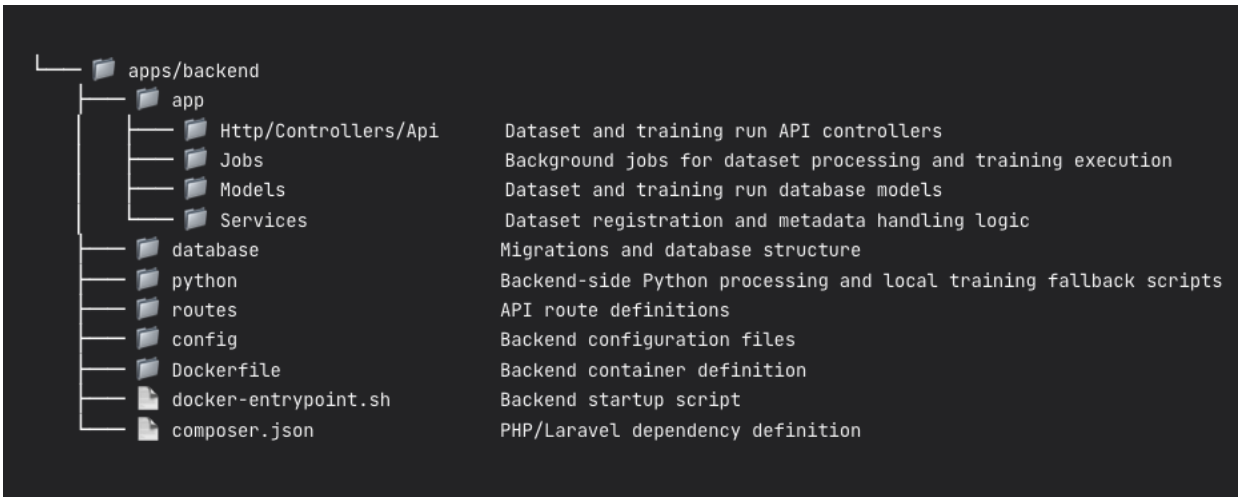

Figure 3: Backends' directory structure

The directory consists of 4 main subdirectories:

- The `Controllers` directory: contains the API endpoints used by the frontend wizard.
- The `Jobs` directory: contains preprocessing and training jobs executed through Laravel queues.
- The `Services` directory: contains reusable orchestration and trainer communication logic.
- The `Models` directory: contains the Eloquent entities used for dataset and training run persistence.

Thus, the backend operates as the central orchestration layer between the frontend, the database, the data preprocessing layer and the trainer service.

4.3.1 Laravel Backend Environment Setup

Laravel is executed and setup entirely inside a Docker container; the backend container, avoiding a manually configured PHP local environment. The backend container is based on PHP 8.4 and installs Composer dependencies, and defines the required PHP extensions for MySQL communication, archive handling, and Python runtime support used by the training workflow.

/apps/backend/Dockerfile

```
FROM php:8.4-cli

RUN apt-get update && apt-get install -y \
    libzip-dev \
    unzip \
    git \
    python3 \
    python3-pip \
    && docker-php-ext-install pdo pdo_mysql zip

COPY --from=composer:2 /usr/bin/composer /usr/bin/composer

WORKDIR /var/www/html
COPY composer.json composer.lock ./
RUN composer install --no-dev --no-scripts --no-autoloader

COPY . .
RUN composer dump-autoload --optimize
```

Figure 4: Laravel setup on Dockerfile.

Python 3.13.9 and pip 25.3 are also installed inside the backend container because the platform supports a local fallback execution mode where the training script can run directly from the Laravel environment in case the dedicated trainer service is unavailable.

The backend framework modifies the default PHP upload limits `upload_max_filesize` and `post_max_size` that are default to 200MB in order to support larger datasets uploaded through the API or the user interface.

The application is configured using environment variables in the .env file that are injected through Docker Compose [7].

These variables configure:

- database communication,
- queue execution,
- trainer execution mode,
- dataset paths,
- trainer service URL,
- timeout values

The database connection is configured using the internal Docker service name of the MySQL container.

/apps/backend/.env.docker

```
DB_CONNECTION=mysql
DB_HOST=db
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=laravel
DB_PASSWORD=secret
```

Figure 5: Database environment configuration.

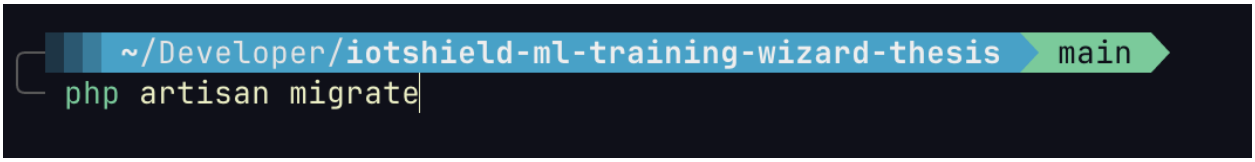

Figure 6: Database migration command.

Database tables are created through Laravel migrations using the following command:

The migration process creates the tables required for the whole end-to-end pipeline workflow. From the datasets to training runs and queue execution.

This containerized setup keeps the backend reproducible because all runtime dependencies are defined inside Docker instead of depending on host machine configuration [28].

4.3.2 API Route Definition

The frontend communicates with the backend exclusively through REST API endpoints defined in `routes/api.php`. The API is organized around two main resources:

1. Datasets
2. Training runs

Dataset routes are responsible for dataset upload, registration, and listing, while training routes manage run creation, status tracking, and artifact retrieval.

/apps/backend/routes/api.php

```
Route::get('/datasets', [DatasetController::class, 'index']);
Route::post('/datasets/upload', [DatasetController::class, 'upload']);
Route::post('/datasets/register', [DatasetController::class, 'register']);

Route::get('/runs', [TrainingRunController::class, 'index']);
Route::post('/runs', [TrainingRunController::class, 'store']);
Route::get('/runs/{run}', [TrainingRunController::class, 'show']);

Route::get('/runs/{run}/artifacts/{filename}',
    [TrainingRunController::class, 'artifact']);
```

Figure 7: Main backend API routes.

The frontend does not directly access the database, queue workers, dataset files, or trainer container. All workflow operations pass through the backend API.

The API is responsible for:

- exposing processed datasets to the frontend wizard
- accepting uploaded datasets
- dispatching preprocessing jobs
- creating training runs
- returning execution status
- exposing generated artifacts and metrics

4.3.3 Dataset API and Registration Service

The dataset management is implemented through `DatasetController` and `DatasetRegistrationService`. The controller handles dataset listing, upload, registration, and validation operations, while the service layer manages dataset metadata extraction and database persistence.


```
{
  "data": [
    {
      "id": 2,
      "name": "network_intrusion_detection",
      "task_type": "classification",
      "target_column": "label",
      "feature_columns": [
        "duration",
        "src_bytes",
        "dst_bytes",
        "...",
        "flag_SH"
      ],
      "time_column": null,
      "train_path": "processed/network_intrusion_detection/train.csv",
      "test_path": "processed/network_intrusion_detection/val.csv",
      "train_path_absolute": "/var/www/html/storage/datasets/processed/network_intrusion_detection/train.csv",
      "test_path_absolute": "/var/www/html/storage/datasets/processed/network_intrusion_detection/val.csv",
      "n_rows": 25192,
      "n_features": 118,
      "label_distribution": {
        "benign": 13449,
        "malicious": 11743
      },
      "notes": "NIDS: binary intrusion classification (labels reported as benign/malicious; "
    }
  ]
}
```


Figure 8: A dataset API response returned by /api/datasets.

Dataset uploads are validated before preprocessing begins. The backend validates the uploaded file type, dataset slug, target column, ingestion profile, optional positive class, and optional time column. Supported upload formats include CSV and Excel files.

After validation, the uploaded dataset is stored inside the raw dataset directory, and a preprocessing queue job is dispatched asynchronously.

/apps/backend/app/Http/Controllers/Api/DatasetController.php

```
$request->validate([
            'file' => ['required', 'file', 'max:204800'],
            'slug' => ['required', 'string', 'regex:/^[a-z][a-z0-9_]{1,63}$/'],
            'target_column' => ['required', 'string', 'max:255'],
            'dataset_name' => ['sometimes', 'string', 'max:255'],
            'positive_class' => ['sometimes', 'nullable', 'string', 'max:255'],
            'ingest_profile' => ['sometimes','string',
                                 'in:binary_tabular,telemetry_tabular,
                                  ton_iot_csv,network_kdd'],
            'time_column' => ['sometimes', 'nullable', 'string', 'max:255'],
        ]);
```

Figure 9: Dataset upload validation logic.

Dataset registration is handled through `DatasetRegistrationService`. The service reads the `metadata.json` file generated by the preprocessing pipeline and stores the extracted metadata in the database.

The registration workflow extracts:

- dataset name
- task type
- feature columns
- target column
- training and evaluation dataset paths
- row count
- label distribution
- preprocessing notes

Separating the registration logic into a dedicated service keeps the controller lightweight and allows dataset registration to be reused by API endpoints, scripts, and command-line workflows.

4.3.4 Training Run Creation and Validation

Training runs are created through `TrainingRunController`. When a user starts an experiment through the wizard's interface, the selected dataset, algorithm, and training parameters are submitted to the backend API that validates the request before the training run gets created and executed.

Specifically, the backend validates:

- dataset existence
- supported algorithm selection
- parameter structure

The implemented algorithms include Random Forest and Logistic Regression for classification tasks, and Isolation Forest for anomaly detection.

/apps/backend/app/Http/Controllers/Api/TrainingRunController.php

```php
$validated = $request->validate([
    'dataset_id' => [
        'required',
        'integer',
        'exists:datasets,id'
    ],
    'algorithm' => [
        'required',
        'string',
        'in:random_forest,logistic_regression,isolation_forest'
    ],
    'params' => ['sometimes', 'array'],
]);

$run = TrainingRun::create([
    'dataset_id' => $validated['dataset_id'],
    'algorithm' => $validated['algorithm'],
    'params' => $validated['params'] ?? [],
    'status' => 'queued',
]);
```

Figure 10: Training run validation and creation

After validation, a new `TrainingRun` record is created with the status queued. The record stores:

- The selected dataset
- The selected algorithm for that run
- The training parameters of the run
- The execution status
- Generated artifact references such as confusion matrices
- Timestamps

Finally, the controller also validates whether the selected dataset matches the seleceted training run type. Classification runs required labeled datasets, while anomaly detection runs support telemetry type datasets and timeseries inputs. Since the implemented confusion matrix workflow is designed around binary evaluation, the backend also checks the dataset label distribution where applicable. Once the training run is created, the backend dispatches the `ExecuteTrainingRunJob` queue job responsible for asynchronous execution. If the queue is configured as asynchronous, the job executes immediately; otherwise, it is handled by the queue worker. This makes the backend flexible for both instant and long term job execution.

In case the validation fails, the controller stops the request before a training job is created and Laravel returns an error response preventing unsupported runs to reach the queue worker or the trainer service.

/apps/backend/app/Http/Controllers/Api/TrainingRunController.php

```php
if (! in_array($dataset->task_type, ['classification', 'anomaly_detection'], true)) {
    return response()->json([
        'message' => 'Only classification and anomaly detection runs are supported.', ], 422);
}

if ($nClasses !== null && $nClasses !== 2) {
    return response()->json([
        'message' => 'Unsupported classification type.',
        'dataset' => $dataset->name,
        'n_classes' => $nClasses, ], 422);
}
```

Figure 11: Validation failure handling during training run creation

4.3.5 Queue Job and Pipeline Control logic

The main orchestration logic is implemented inside `ExecuteTrainingRunJob`. This job transforms a stored training run into an executable machine learning configuration.

The execution process perfroms the following steps:

1. Loads the training run and dataset metadata from the database
2. Resolves the paths of the training and evaluation datasets
3. Verifies that the required dataset exists
4. Validates the extracted feature columns and dataset configuration
5. Updates the training run status to “running”
6. Creates the output directory used for the generated artifacts
7. Generates the structured training cofniguration
8. Writes the run_config.json execution file
9. Executes the Python trainer service
10. Stores the generated metrics, reports, and visualization artifacts
11. Updates the training status to either “completed” or “failed”

/apps/backend/app/Jobs/ExecuteTrainingRunJob.php

```php
class ExecuteTrainingRunJob implements ShouldQueue
  {
    public function handle()
      {
        $this->run->update(['status' => 'running']);

        $config = $this->buildConfiguration();

        $this->writeConfig($config);

        $this->executeTrainer($config);

        $this->storeArtifacts();

        $this->run->update(['status' => 'completed']);
    }
}
```

Figure 12: Simplified training job execution flow

The generated `run_config.json` file contains all the information required by the Python trainer service. It includes dataset file paths, target column, feature columns, selected algorithm, parameters, task type, time column and the output directory.

/apps/backend/app/Jobs/ExecuteTrainingRunJob.php

```php
$config = [
    'train_csv' => $trainAbs,
    'eval_csv' => $evalAbs,
    'target_column' => $dataset->target_column,
    'feature_columns' => $features,
    'algorithm' => $run->algorithm,
    'params' => $run->params ?? [],
    'task_type' => $dataset->task_type,
    'time_column' => $dataset->time_column,
    'output_dir' => $outDir,
];

$configPath = $outDir.DIRECTORY_SEPARATOR.'run_config.json';
File::put($configPath, json_encode($config, JSON_THROW_ON_ERROR | JSON_PRETTY_PRINT));
```

Figure 13: Structure of a generated run_config.json file

The generated configuration preserves the exact execution parameters of each run, this supports reproducibility and makes debugging painless, since the input settings of the training run can be inspected after it execution [32, 43]. The artifacts are stored inside shared Docker volumes mounted between the backend layer and the trainer container. This allows both services to access datasets, configuration files, generated metrics, plots and reports.

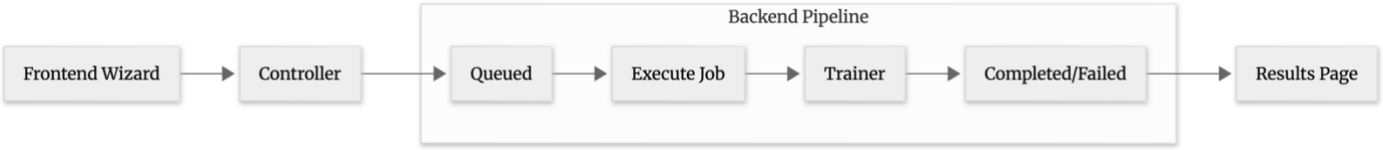

Figure 14: The backend training execution workflow

4.3.6 Trainer Service Communication

The backend service of the IoT ML training platform supports two execution modes, HTTP trainer execution and local Python fallback execution, where the HTTP mode represents the default containerized path and the local fallback provides a practical recovery option during debugging or

trainer service unavailability. The execution mode is controlled through the `TRAINING_EXECUTOR` environment variable.

In HTTP mode, the backend sends the generated configuration to the trainer container through an internal Docker network request. The trainer service receives the configuration, executes the Python training pipeline, generates the required artifacts and returns the execution result.

/apps/backend/app/Jobs/ExecuteTrainingRunJob.php

```php
if ($executor === 'http') {
    $url = (string) env('TRAINER_HTTP_URL', '');

    $seconds = (int) env('TRAINER_HTTP_TIMEOUT_SECONDS', (string) min($this->timeout, 900));
    $seconds = $seconds > 0 ? $seconds : min($this->timeout, 900);

    $response = Http::timeout($seconds)
        ->acceptJson()
        ->asJson()
        ->post($url, $config);

    if (! $response->successful()) {
        $body = $response->body();

        $this->failRun($run,trim('Trainer HTTP failed: HTTP'.$response->status()."\n".$body));
    }
}
```

Figure 15: HTTP trainer service communication

The HTTP request is configured with a timeout to prevent the backend from waiting indefinitely for long or failed training executions. If the trainer service returns an unsuccessful response, the backend marks the run as failed and stores the returned error message, helping the frontend to display meaningful and coordinative feedback instead of silently failing.

In local mode, the backend executes the Python training script directly using Symfony Process rather than sending the configuration to the HTTP trainer service. The generated `run_config.json` file is passed as an argument to the script, allowing the same training logic and configuration structure to be reused in both execution modes.

/apps/backend/app/Jobs/ExecuteTrainingRunJob.php

```
$python = env('PYTHON_EXECUTABLE', 'python3');
$script = base_path('python/train_run.py');

$process = new Process([$python, $script, $configPath]);
$process->setTimeout($this->timeout);
$process->run();

if (! $process->isSuccessful()) {
    $this->failRun(
        $run,
        trim($process->getErrorOutput() ?: $process->getOutput() ?:'Training process failed.')
    );

    return;
}
```

Figure 16: Local trainer fallback execution

The fallback implementation was retained for debugging and recovery purposes because it allows the workflow to execute and operate even when the trainer container is unavailable. If the local process fails, the backend captures the process or error output, marks the run as failed, and stores the failure message for display through the frontend similarly with the HTTP mode. This separation keeps the Laravel backend focused on orchestration while machine learning execution remains isolated inside the Python environment [34].

4.3.7 Error Handling and Artifact Access

Training executions move through the following states:

- “queued”
- “running”
- “completed”
- “failed”

In case a required dataset is missing, the trainer service fails, or the Python execution returns an error, the training run is considered failed. Thus, the backend marks the run as “failed” and stores the corresponding error message.

Before execution starts, the backend validates the selected dataset’s existence as a registered database record and checks the availability of the required training file, evaluation file, and feature column configuration. These checks prevent invalid workflows from reaching the trainer layer. Generated artifacts are exposed through controlled API endpoints where only predefined filenames are accessible through the artifact route.

The allowed artifacts include:

- `metrics.json`
- `classification_report.json`
- `confusion_matrix.png`
- `timeseries.png`
- `timeseries.json`
- `run_config.json`

If the requested file does not exist or is not included in the allowed artifact list, the backend returns a 404 response. After successful execution, artifact references are stored in the training run record and transformed into API URLs returned to the frontend. These URLs are used by the results interface for displaying metrics, confusion matrices, anomaly visualizations, generated reports and downloadable artifacts.

The backend therefore acts as the central orchestration component connecting the frontend wizard, asynchronous execution pipeline, trainer service, database persistence layer, and result visualization workflow.

4.4 Database Implementation

The database layer of the ML Training Wizard was implemented using MySQL 8.0 and Laravel migrations. In this system, the database is not used for storing full data set files or generated output

files directly. Instead, it stores the structured metadata required by the application in order to operate end-to-end. The actual dataset splits, configuration files, metrics, confusion matrices, plots, and time-series outputs are stored in the shared dataset workspace, while MySQL stores the information required to locate, interpret, and expose these resources through the backend API. This separation keeps the database focused on application state and experiment tracking, while the filesystem stores large dataset and artifact files to prevent storage overhead and unsastainability.

4.4.1 Schema Design and Conventions

The database schema is defined through Laravel migration files located in the backend application. Migrations allow the schema to be recreated in a clean environment by running the migration command [Figure 6], rather than using SQL to manually create tables reducing complexity of new dataset insertions. This technique contributes on achieving the requirement of keeping the project generalized, as a new installation can create the same database structure during setup [43].

The main application tables are:

1. `datasets` : Stores metadata about processed datasets available to the wizard
2. `training_runs` : Stores the experiment records that are created when a training run is launched

The database also includes standard Laravel tables for framework features such as caching, sessions and jobs. Finally, the schema follows common Laravel conventions. Each table uses an auto-incrementing `id` field as its primarty key. Therefore, fields ending in `_id` represent foreing key relationships. JSON collumns are used for structured values such as training parameters, artifact references and label distributions. Timestamp fields are used for tracking creation, updates and training execution events.

4.4.2 Datasets Table

The `datasets` table acts as the dataset catalog of the system. Each row represents a processed dataset that has been registered and can be selected through the frontend wizard. The table does not store the actual CSV data, only the descriptive metadata and file references required by the backend and training workflow.

Important fields in this table include:

1. the dataset name: The name of the dataset
2. `task_type` : Identifies whether the dataset is used for classification or anomaly detection.
3. `target_column` : Identifies the label column.
4. `feature_columns` : Defines the input columns used by the model.
5. `train_path` : Points to the path of the processed training files in the dataset workspace.
6. `test_path` : Points to the path of evaluation files inside the shared dataset workspace.
7. `n_rows` : Stores the number of dataset rows
8. `n_features` : Stores the number of extracted features
9. `label_distribution` : Stores the class distribution as structured metadata,
10. `time_column` : Stores the timestamp column for telemetry workflows
11. `notes` : Stores optional preprocessing notes

These fields allow the frontend to display dataset information and allow the backend to construct the configuration required for model training.

Field	Type	Length	Unsign...	Zerofill	Bin...	Allow N...	Key	Default
id	BIGINT		☑	☐	☐	☐	PRI	
name	VARCHAR	255	☐	☐	☐	☐	UNI	
task_type	VARCHAR	255	☐	☐	☐	☐		
target_column	VARCHAR	255	☐	☐	☐	☑		NULL
feature_columns	TEXT		☐	☐	☐	☑		
train_path	VARCHAR	255	☐	☐	☐	☐		
test_path	VARCHAR	255	☐	☐	☐	☑		NULL
n_rows	INT		☑	☐	☐	☑		NULL
n_features	INT		☑	☐	☐	☑		NULL
label_distribution	JSON		☐	☐	☑	☑		NULL
time_column	VARCHAR	255	☐	☐	☐	☑		NULL
notes	TEXT		☐	☐	☐	☑		
created_at	TIMESTAMP		☐	☐	☐	☑		NULL
updated_at	TIMESTAMP		☐	☐	☐	☑		NULL

Figure 17: Structure of the datasets table

4.4.3 Training Runs Table

The `training_runs` table stores the execution records of the training experiments. Each row represents a training run created through the wizard's interface. This table allows the system to track the lifecycle of an experiment from creation to completion or failure.

- `dataset_id` : Links the training run to the dataset used for execution
- `algorithm` : Stores the selected machine learning algorithm
- `params` : Stores the selected training parameters as structured JSON metadata
- `status` : Stores the execution state of the run, such as queued, running, completed, or failed
- `error_message` : Stores the failure message if execution fails
- `artifacts` : Stores references to generated output files
- `started_at` : Stores the timestamp when execution begins
- `finished_at` : Stores the timestamp when execution finishes

When a user starts a training run, the backend first creates a record with the status queued. Once the queue worker begins execution, the status becomes running. After successful execution, the status is updated to completed, and generated artifact references are stored. If execution fails, the status becomes failed and the error message is recorded. The `params` field stores user-selected training parameters, while the `artifacts` field stores references to outputs such as `metrics.json`, `confusion_matrix.png`, `timeseries.png`, `timeseries.json`, and `run_config.json`. These references are later transformed by the backend into API URLs so the frontend can display the generated results

Field	Type	Length	Unsigned	Zerofill	Bin...	Allow Null	Key	Default	Extra
id	BIGINT		☑	☐	☐	☐	PRI		auto_increme...
dataset_id	BIGINT		☑	☐	☐	☐	MUL		None
algorithm	VARCHAR	255	☐	☐	☐	☐			None
params	JSON		☐	☐	☑	☑		NULL	None
status	VARCHAR	255	☐	☐	☐	☐		'queued'	None
error_message	TEXT		☐	☐	☐	☑			None
artifacts	JSON		☐	☐	☑	☑		NULL	None
started_at	TIMESTAMP		☐	☐	☐	☑		NULL	None
finished_at	TIMESTAMP		☐	☐	☐	☑		NULL	None
created_at	TIMESTAMP		☐	☐	☐	☑		NULL	None
updated_at	TIMESTAMP		☐	☐	☐	☑		NULL	None

Figure 18: Structure of the training_runs table

4.4.4 Dataset and Training Run Relationship

The relationship between datasets and training runs is implemented through the `dataset_id` foreign key inside the `training_runs` table. This field references the id field of the datasets table. As a result, one dataset can be associated with multiple training runs, while each training run belongs to exactly one dataset.

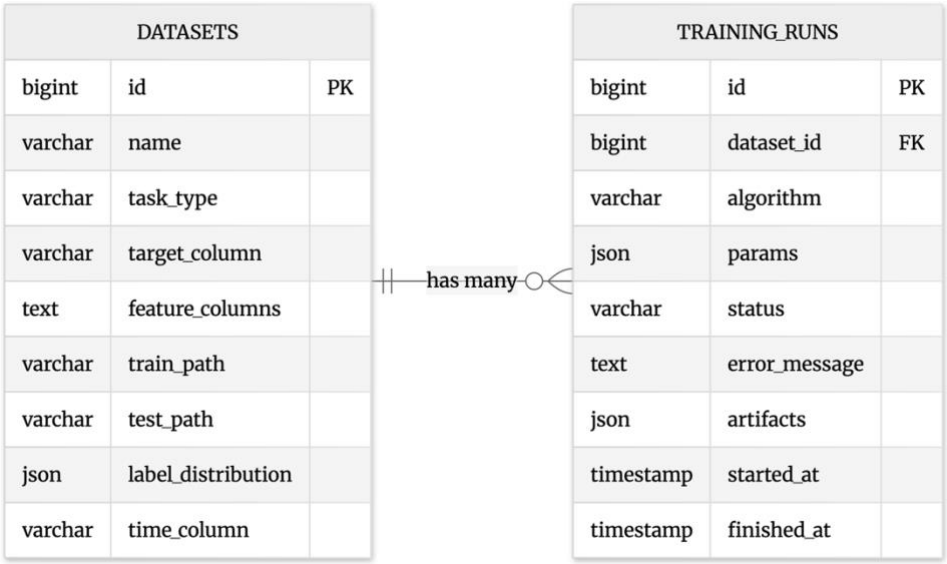

Figure 19: Relationship between datasets and training runs

This relationship supports repeated experimentation. The same dataset can be trained with differnet algorithms or parameter configurations, and each experiment remains stored as a separate run record. This makes it possible to inspect the history of training attempts and compare their outputs. The foreign key relationship is defined through Laravel Migrations.

apps/backend/database/migrations/2026_04_26_000001_create_training_runs_table.php

```
Schema::create('training_runs', function (Blueprint $table) {
            $table->id();
            $table->foreignId('dataset_id')->constrained('datasets')->cascadeOnDelete();
```

Figure 20: Laravel migration defining the dataset-to-training_run relationship

The `cascadeOnDelete()` behaviour prevents orphaned training run records from remaining in the database if the related dataset is removed.

4.5 Dataset Processing Pipeline Implementation

The dataset processing layer is responsible for transforming raw IoT/CPS datasets into a standardized structure that can be used consistently by the backend, trainer service, and frontend wizard. This layer is required because the supported datasets do not follow a common format. They may use different column names, label values, timestamp fields, feature sets, and train/evaluation split requirements. Instead of requiring the backend or trainer service to understand every raw dataset format individually, the system uses dataset-specific preprocessing scripts [17, 26]. Each script converts a dataset into a common processed structure containing training and evaluation files, and a metadata.json file. This metadata file becomes the contract between the preprocessing layer, backend registration logic, database, trainer service, and frontend.

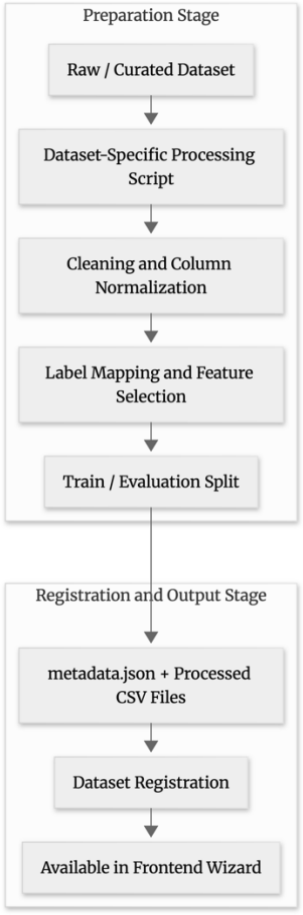

Figure 21: Dataset preprocessing pipeline

4.5.1 Dataset Input and Processing Workflow

The system supports two dataset input paths. The first path uses curated datasets processed through predefined scripts, which is the main approach used for the thesis datasets. The second path supports user-uploaded datasets submitted through the backend API and processed using ingestion profiles. For the curated workflow, raw files are placed under `datasets/raw`, while processed outputs are written under `datasets/processed` using one directory per dataset. Each processed dataset directory contains the generated training file, evaluation file, and `metadata.json` file. After processing, the metadata can be registered by the backend so that the dataset becomes available through the frontend wizard.

4.5.2 Dataset-Specific Preprocessing Scripts

Dataset-specific preprocessing scripts are located inside the `scripts/data` directory. Each script is responsible for understanding the structure of one dataset and converting it into the format expected by the rest of the system. This keeps dataset-specific assumptions outside the generic backend and trainer code. The final implementation includes processors for the main supported dataset types, such as network intrusion detection, SwaT telemetry anomaly detection and spacecraft telemetry anomaly detection. For example, the network intrusion detection processor prepares tabular network traffic data for supervised binary classification. It normalizes column names, maps raw class labels into benign/malicious labels, encodes categorical features, creates train/evaluation splits, and writes the metadata file used by the backend. Telemetry-oriented processors, such as the TS Training SWaT and spacecraft processors, handle timestamped sensor data. These scripts preserve timestamp columns for later visualization, convert sensor values into numeric features, map labels into normal/anomaly format, and generate processed files for anomaly detection workflows. In both classification and telemetry processors, the output format remains consistent even though the raw datasets differ. This consistency allows the backend and trainer service to consume all processed datasets through the same metadata contract instead of implementing a dataset-specific logic during training execution [17, 26].

4.5.3 Label Mapping and Feature Selection

A major responsibility of the preprocessing layer is label normalization. Different datasets represent normal and anomalous behaviour using different naming conventions. Some datasets use values such as normal and anomaly, while others use Normal and Attack or numeric labels.

The processing scripts convert these dataset-specific labels into a consistent binary representation. In the implemented workflow, normal behaviour is represented as "0", while anomalous, malicious, or attack behaviour is represented as "1". This allows the backend and trainer service to operate across different datasets without embedding dataset-specific label logic inside the generic training pipeline.

For example, the TS Training SWaT processor maps the Normal/Attack column into a binary label:

scripts/data/process_ts_training_swat.py

```
lab = df["normal_attack"].astype(str).str.strip().str.lower()
y = lab.map(lambda v: 0 if v == "normal" else 1)

df = df.drop(columns=["normal_attack"])
df[TARGET] = y.astype(int).values
```

Figure 22: Label mapping in the TS Training SWaT processor

Feature selection is also performed during preprocessing. The scripts identify which columns should be used as model inputs and which should be excluded. Target label columns are excluded because they represent the expected output. Timestamp columns are also excluded from `feature_columns` in telemetry datasets as they are used for temporal ordering and visualization and not as direct sensor inputs. The resulting feature list is stored in `metadata.json` under `feature_columns`.

This approach reinforces the system's generalization and dynamic processing by keeping feature selection consistent, since it allows the backend and trainer service to use the same feature definition without recomputing it during training execution. It also reduces the risk of training-time inconsistencies caused by different datasets using different column names, label formats or even timestap conventions [17].

4.5.4 Train and Evaluation Splitting

The preprocessing layer generates the training and evaluation datasets used by the trainer service. For classification datasets, a standard train/evaluation strategy is used. In the network intrusion detection processor, the split is stratified so that both benign and malicious classes remain represented in the training and evaluation subsets.

scripts/data/process_network_intrusion_detection.py

```
X_train, X_val, y_train, y_val = train_test_split(
    X,
    y,
    test_size=TEST_SIZE,
    random_state=RANDOM_SEED,
    stratify=y,
)
```

Figure 23: Stratified train/evaluation split for classification

Telemetry anomaly detection datasets require additional handling because temporal order matters. Randomly shuffling time-series data can make later anomaly visualization misleading, because anomaly scores would no longer align naturally with the original event sequence. For this reason, telemetry processors preserve timestamp ordering and use time-aware splitting where appropriate [10, 21].

In the TS Training SWaT processor, the dataset is sorted by timestamp and split into a time-contiguous training and evaluation window:

scripts/data/process_ts_training_swat.py

```
df["timestamp"] = pd.to_datetime(df["timestamp"], errors="coerce")
df = df.sort_values("timestamp")

cut = max(1, int(len(df) * (1.0 - TEST_SIZE)))
train_df = df.iloc[:cut].copy()
test_df = df.iloc[cut:].copy()
```

Figure 24: Time-contiguous split for telemetry anomaly detection

This is where the time_column field becomes important. Timestamp columns such as timestamp or ts are preserved in the processed CSV files but excluded from feature_columns. The

preprocessing layer stores the timestamp field inside metadata.json using time_column, allowing later anomaly detection and visualization steps to align generated outputs with their original temporal order.

4.5.5 Metadata Generation

Each preprocessing script generates a metadata.json file together with the processed dataset files. This metadata file defines how the dataset should be interpreted by the backend and trainer service. It contains the dataset name, task type, target column, feature columns, row count, number of features, label distribution, file paths, notes, and, for telemetry datasets, the time column. A typical processed dataset directory contains the processed train/evaluation files and `metadata.json`. This structure allows different datasets to be handled through the same contract even when their raw formats are different.

Figure 25: Screenshot of datasets/processed/ts_training_swat dataset directory

datasets/processed/ts_training_swat/metadata.json


```
{
  "dataset_name": "ts_training_swat",
  "version": "ts_training_v1",
  "task_type": "anomaly_detection",
  "target_column": "label",
  "time_column": "timestamp",
  "feature_columns": [
    "fit101","lit101",...,"p101","p102","ait201"],
  "n_rows": 449919,
  "n_features": 51,
  "label_distribution": { "normal": 395298,"attack": 54621 },
  "created_at": "2026-05-05",
  "files": {
    "train": "processed/ts_training_swat/train.csv",
    "test": "processed/ts_training_swat/test.csv"
  },
  "notes": "TS Training SWaT.csv (Normal/Attack). Timestamp kept for plotting; features
exclude timestamp."
}
```


Figure 26: Screenshot of datasets/processed/ts_training_swat/metadata.json file

A shortened metadata generation example is shown below:

/scripts/data/process_ts_training_swat.py

```
meta = {
    "dataset_name": DATASET_SLUG,
    "task_type": "anomaly_detection",
    "target_column": TARGET,
    "time_column": TIME_COLUMN,
    "feature_columns": feature_columns,
    "n_rows": int(len(df)),
    "n_features": int(len(feature_columns)),
    "label_distribution": label_dist,
    "files": {
        "train": f"processed/{DATASET_SLUG}/train.csv",
        "test": f"processed/{DATASET_SLUG}/test.csv",
    },
}
```

Figure 27: Metadata generation for telemetry datasets

The metadata file improves reproducibility because the dataset carries the information required to train and evaluate it. The backend does not need to infer which column is the label, which columns are features, or where the processed files are located. These decisions are made during preprocessing and stored explicitly [17, 32].

4.5.6 Telemetry Dataset Handling

Telemetry datasets require additional handling because their records are collected over time. For this reason, the preprocessing scripts preserve the timestamp column in the processed CSV files and store its name in `metadata.json` through the `time_column` field. This design allows later training stages to generate time-aligned outputs such as `timeseries.json`, where anomaly scores, timestamps, and labels remain connected to the original telemetry sequence [10, 21]. As a result, the frontend can display not only whether anomalies were detected, but also when they occurred during the evaluation window, in a time series matter.

Overall, the dataset processing implementation standardizes heterogeneous IoT/CPS security datasets into a unified machine learning format shared by the backend, trainer service, and frontend wizard making the system sustainable and dynamic providing reproducibility and generalization instead of limitations.

4.6 ML Training Implementation

The machine learning training layer is responsible for executing the selected model, evaluating the result, and generating the artifacts displayed by the frontend. While the dataset processing layer prepares standardized train and evaluation files, the training layer consumes these files through the configuration generated by the backend. The core training logic is implemented in `/apps/backend/python/train_run.py`. The script receives a generated `run_config.json` file, loads the selected datasets, builds the required model, evaluates predictions, and writes the output artifacts such as `metrics.json`, `confusion_matrix.png`, `timeseries.json`, and `timeseries.png`.

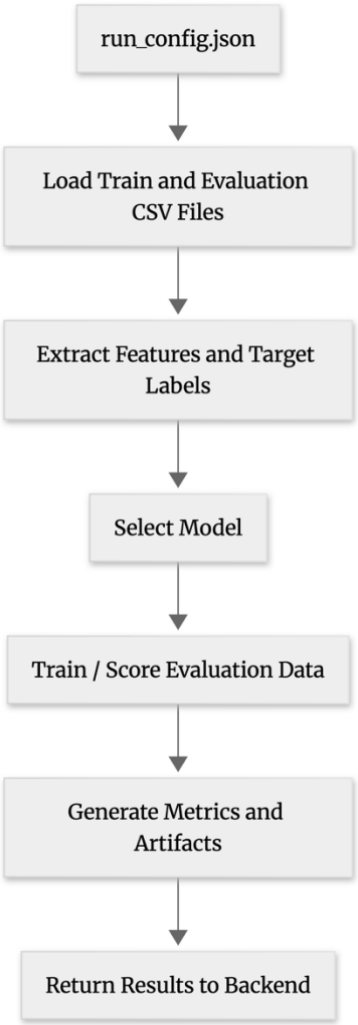

Figure 28: Machine learning training pipeline

4.6.1 Training Configuration and Model Selection

Training begins from the `run_config.json` file produced by the backend job. This file contains all the information required by the Python trainer, including dataset paths, target column, feature columns, selected algorithm, task type, optional time column, model parameters, and output directory.

/apps/backend/python/train_run.py

```
cfg = _load_config(sys.argv[1])
train = pd.read_csv(cfg["train_csv"])
ev = pd.read_csv(cfg["eval_csv"])

target = cfg["target_column"]
features = cfg["feature_columns"]
algorithm = cfg["algorithm"]
params = cfg.get("params") or {}
task_type = str(cfg.get("task_type") or "classification").strip().lower()
time_column = cfg.get("time_column")
```

Figure 29: Loading the training configuration

The trainer supports three algorithms: Random Forest, Logistic Regression, and Isolation Forest. Random Forest and Logistic Regression are used for supervised classification, while Isolation Forest is used for anomaly detection. The selected algorithm and its parameters are read from the configuration, allowing the same training script to support multiple experiment types without changing the source code for each run [8, 24].

4.6.2 Classification and Anomaly Detection Workflows

For classification tasks, the trainer loads the processed training and evaluation files, extracts the configured feature columns, encodes target labels when necessary, and trains the selected classifier. The implementation builds a preprocessing pipeline where numeric values are imputed and scaled, while categorical values are imputed and one-hot encoded. This allows the classification workflow to handle tabular datasets containing mixed feature types. For anomaly detection tasks, the trainer uses Isolation Forest algorithm that identifies anomalous observations by isolating samples that differ significantly from the majority of the data. In this workflow, labels are not used to train the model; they are used only for evaluation and visualization. The implementation derives binary anomaly predictions by calculating anomaly scores and applying a threshold based on the

contamination parameter. This follows the anomaly-scoring style used in many notebook-based anomaly detection workflows.

/apps/backend/python/train_run.py

```python
if str(algorithm).lower().strip() == "isolation_forest":
    pipe.fit(X_train)
    score_train = (-pipe.decision_function(X_train)).astype(float)
    score_ev = (-pipe.decision_function(X_ev)).astype(float)
    frac = _contamination_fraction(params, default=0.05)
    threshold = float(np.quantile(score_train, 1.0 - frac))
    y_pred = (score_ev >= threshold).astype(int)
else:
    pipe.fit(X_train, y_train)
    y_pred = pipe.predict(X_ev)
```

Figure 30: Classification and anomaly execution logic

before generating visualization data. This ensures that anomaly scores remain aligned with the original telemetry sequence [21].

4.6.3 Metrics and Artifact Generation

After training and evaluation, the script writes the generated outputs to the run output directory. The main artifacts are `metrics.json`,`confusion_matrix.png`, `timeseries.json`, `timeseries.png`, and the original `run_config.json`. The `metrics.json` file stores structured evaluation results such as accuracy, algorithm name, confusion matrix values, label order, label names, and classification report values. The confusion matrix is also exported as an image so it can be exported directly from the frontend. For anomaly detection runs,as mentioned earlier the trainer also generates `timeseries.json`. This file contains anomaly score values, timestamps when available, and ground-truth labels used for evaluation. The output is bounded to avoid generating overly large frontend payloads while still covering the evaluation window.

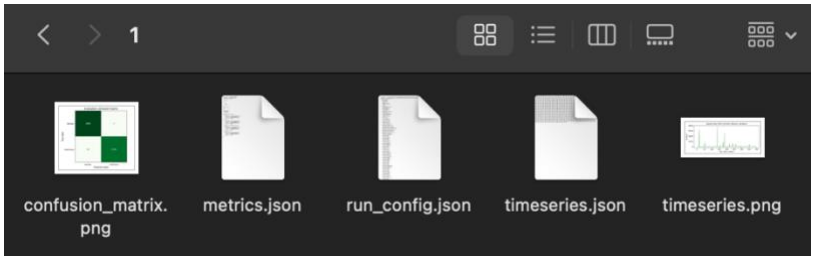

Figure 31: Screenshot of datasets/runs/1 directory

4.6.4 Trainer Service Execution

The trainer is implemented as a small Python HTTP server whose only job is to turn an accepted training request into a bounded subprocess run of `train_run.py`. At runtime it exposes a minimal surface: a readiness probe (`/health`) and a training entrypoint(`POST /run`) that accepts the JSON payload forwarded from the orchestration layer. When `/run` is invoked, the handler validates that the body parses as a JSON object, then normalizes paths in the configuration. Paths that refer to the datasets tree as seen from the host-side application are rewritten to the path of the same files inside the trainer container, using `HOST_DATASETS_ROOT` and `TRAINER_DATASETS_ROOT`. This step exists solely to reconcile bind-mount layout with unchanged training semantics: the training script always receives paths that exist in its own filesystem view. Before launching work, the service enforces a write sandbox on `output_dir`: it must resolve under the shared datasets `runs` hierarchy so training artifacts cannot be written to arbitrary locations inside the container. After validation, the normalized configuration is written to a temporary JSON file on disk; the service then starts `train_run.py` as a subprocess, passing that file path as the sole CLI argument. Process `stdout/stderr` are captured; a non-zero exit code is reported back as a failed execution with a truncated diagnostic excerpt, while a clean exit yields a short JSON success payload. The temporary config file is deleted in a finally path so the runner does not leave per-request configuration on disk. Viewed end-to-end, the trainer service is therefore not a second training implementation: it is an execution envelope—HTTP parsing, mount-aware path adjustment, output-directory policy, subprocess isolation, and structured completion reporting—for the shared Python training script.

4.7 Containerization and Deployment Implementation

The ML Training Wizard is deployed as a Docker multi-container environment using Docker Compose. Containerization keeps the Laravel backend, Vue.js frontend, MySQL database, queue worker, Python trainer service and optional preprocessing tools isolated while allowing them to communicate over a shared Compose network and through common volume mounts for the datasets workspace [7, 28].

This section documents how the platform is packaged and run. Explaining Dockerfiles, the Compose stack, ports and service naming, the inter-service communication within the pipeline, and persistent or shared storage; Aiming to emphasize on deployment mechanics that make that

behaviour repeatable in a new machine, as backend orchestration, HTTP and local fallback trainers, and training scripts behaviour are discussed in Sections 4.3 and 4.6. Figure 31 provides a visual overview of the containerized service layout.

At a glance, the default runtime layout uses a straightforward host configuration, where the browser loads the wizard's UI from the mapped frontend port and talks to the Laravel API on a separate mapped API port, while containers inside Compose use service names and the internal bridge network. In this stack, the trainer listens on 8088 only inside Compose and is not published to the host. The SPA sends REST calls directly to http://localhost:8000 using `VITE_API_BASE_URL`, so the Vite dev container does not proxy Laravel.

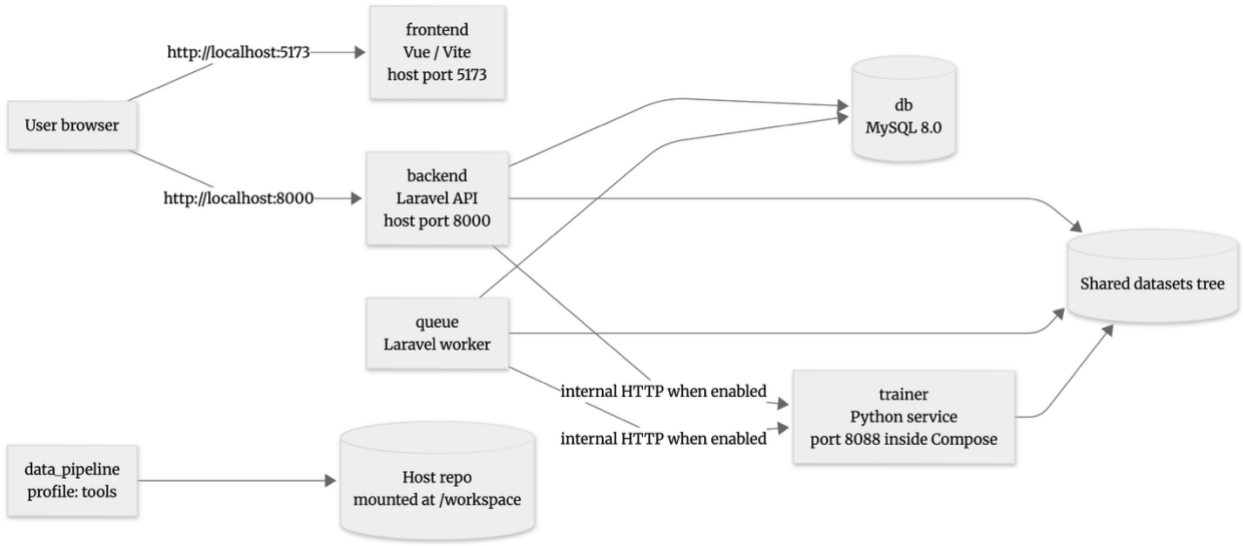

Figure 32: Containerized service layout of the ML Training Wizard

4.7.1 Dockerfiles

The project builds three custom images; the backend, data preprocessing, and trainer images. The frontend is not built from a no repository Dockerfile but uses a stock Node image declared in Compose, allowing development to remain lightweight yet efficient. The Laravel backend image (`apps/backend/Dockerfile`) yields the PHP 8.4 image for the Laravel API and its Composer-managed dependencies. It enables the PDO/MySQL and ZIP extensions needed for persistence and dataset packaging, applies PHP upload and execution limits appropriate for larger transfers, and installs Python 3 with `python/requirements-train.txt` so `train_run.py` can execute in this image when training is not delegated to the dedicated trainer process. For deployment, Docker compose uses the same Dockerfile to build and run the same image for the API and the worker containers, responsible for the backend and queue services. These two containers differ only in start command and role. The trainer image is built using `apps/docker/trainer/Dockerfile`. That Dockerfile defines a Python 3.12 image used for the training service. It installs `apps/trainer/requirements.txt`, copies `server.py` and `apps/backend/python/train_run.py` into the image so containerised training runs use the same script, thus logic as the path executed from the backend environment.ENV entries declare `TRAINER_DATASETS_ROOT` and `HOST_DATASETS_ROOT`, which support path rewriting between backend and trainer mount layouts (see section 4.6.4).

/apps/docker/trainer/Dockerfile

```
FROM python:3.12-slim-bookworm
WORKDIR /app
COPY trainer/requirements.txt /tmp/requirements.txt
RUN pip install --no-cache-dir -r /tmp/requirements.txt
COPY trainer/server.py /app/server.py
COPY backend/python/train_run.py /app/train_run.py
ENV TRAINER_DATASETS_ROOT=/datasets
ENV HOST_DATASETS_ROOT=/var/www/html/storage/datasets
EXPOSE 8088
CMD ["python", "/app/server.py"]
```

Figure 33: Trainer service Dockerfile

The data preprocessing image is built through the `/scripts/data/dockerfile` for `scripts/data/process_*.py. The build context is the repository root; `PYTHONNUNBUFERED=1` gives predictable console logging. The service is being started through Compose's `tools` profile rather than the default wizard stack.

/scripts/data/Dockerfile

```
FROM python:3.12-slim-bookworm
WORKDIR /workspace
COPY scripts/data/requirements.txt /tmp/requirements.txt
RUN pip install --no-cache-dir -r /tmp/requirements.txt
ENV PYTHONUNBUFFERED=1
```

Figure 34: Data pipeline Dockerfile

4.7.2 Docker Compose Configuration

The stack is declared in `infra/docker/compose.yml`. Compose builds custom images, starts services, maps host ports, injects environment values, attaches volumes, and defines startup order. The table below lists the main Compose services.

Service	Role
db	MySQL 8.0 with health check and persistent volume
backend	Laravel API; 8000:8000 on the host
queue	Same image as backend; runs `php artisan queue:work` (via shell command in Compose)
frontend	Node 20 Alpine; Vite on 5173:5173
trainer	Python trainer HTTP service (8088 on the internal network only)
data_pipeline	Optional tooling; profile `tools`; full repo at `/workspace`

Table 1: Main Docker Compose services and roles

The backend and queue services share one image but different commands. `QUEUE_CONNECTION` is set to `sync` in the commited Compose file so short demo runs can execute jobs without relying on a long-running worker loop for basic behaviour; the queue container is still included so the topology matches an asynchronous deployment without changing the volume or environment layout.

/infra/docker/compose.yml

```
backend:
  build:
    context: ../../apps/backend
    dockerfile: Dockerfile
  ports:
    - "8000:8000"
  depends_on:
    db:
      condition: service_healthy
    trainer:
      condition: service_started

queue:
  build:
    context: ../../apps/backend
    dockerfile: Dockerfile
  command: sh -c "composer install
           --no-interaction 2>/dev/null;
           php artisan queue:work"
  depends_on:
    trainer:
      condition: service_started
    backend:
      condition: service_started
```

Figure 35: Backend and queue services

/infra/docker/compose.yml

```
data_pipeline:
  profiles:
    - tools
  build:
    context: ../..
    dockerfile: scripts/data/Dockerfile
  working_dir: /workspace
  volumes:
    - ../../:/workspace
```

Figure 36: Data pipeline profile

4.7.3 Inter-Service Communication

Communication inside the platform operates on two levels:

- **Browser-to-service communication:** The user accesses the frontend through the Vue.js application running on port `517`, while the backend API is exposed on port `8000`. Since the browser operates outside the Docker network, the frontend uses the configured `VITE_API_BASE_URL` variable to route API requests to the backend service.
- **Container-to-container communication:** Internal services communicate through Docker Compose service names. The Laravel backend and queue worker connect to MySQL using the `db` service name, while training requests are forwarded to the trainer container through the internal `trainer` service URL when HTTP trainer execution is enabled, as discussed earlier in Section 4.3.6.

/infra/docker/compose.yml

```
frontend:
  environment:
    VITE_API_BASE_URL: http://localhost:8000

backend:
  environment:
    DB_HOST: db
    TRAINING_EXECUTOR: ${TRAINING_EXECUTOR:-local}
    TRAINER_HTTP_URL: ${TRAINER_HTTP_URL:-http://trainer:8088/run}
    TRAINER_HTTP_TIMEOUT_SECONDS: ${TRAINER_HTTP_TIMEOUT_SECONDS:-840}
```

Figure 37: Service communication environment variables

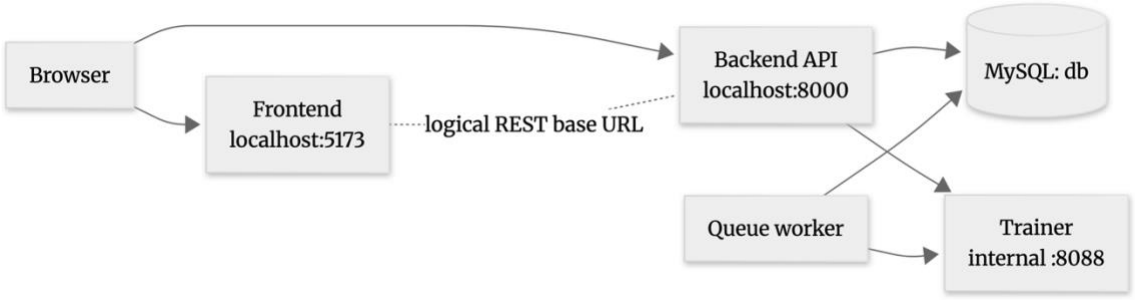

Figure 38: Communication paths

4.7.4 Environment Variables and Volumes

Environment variables configure database connectivity, the queue driver, the datasets root inside the Laravel containers, browser-side API routing, and trainer delegation (`TRAINING_EXECUTOR`,`TRAINER_HTTP_URL`,`TRAINER_HTTP_TIMEOUT_SECONDS`). Trainer listen defaults (TRAINER_HTTP_HOST, `TRAINER_HTTP_PORT`) and image-level dataset roots (`TRAINER_DATASETS_ROOT`, `HOST_DATASETS_ROOT`) appear in `compose.yml` for the trainer service where relevant.

Volume/ mount	Purpose
db_data	Persists MySQL data across restarts
frontend_node_modules	Keeps Node dependencies when the app tree is bind-mounted
`../../datasets`→ `/var/www/html/storage/datasets`	Thesis datasets workspace as seen by Laravel paths
`../../datasets` → `/datasets` (trainer)	Same files with trainer-side paths (rewriting in Section 4.6.4)
`../../`→ `/workspace` (data_pipeline)	Full repository for preprocessing scripts

The shared datasets tree is central: processed CSVs, `runs/` outputs, and other artifacts must be readable and writable from the services that produce or serve them (see Sections 4.3, 4.5, and 4.6). Bootstrap scripts and defaults. The Compose file expands `TRAINING_EXECUTOR` with default local when the variable is unset on the host. The repository's `scripts/dev/up.sh` and `scripts/dev/up_full.sh` export `TRAINING_EXECUTOR=http` (and related trainer URL and timeout) before `docker compose up`, so the documented thesis cold-start path uses the HTTP trainer unless the operator overrides it. `up_full.sh` additionally runs migrations and dataset registration after the stack is up.

Figure 39: Startup script environment configuration

4.8 Frontend Implementation

The frontend implementation was developed on `apps/frontend` using Vue.js and Vite and acts as the user interaction layer of the ML Training Wizard [40, 42]. The frontend is implemented as a single-page application where the main interface is loaded once and navigation between views is handled through Vue Router without requiring full page reloads. The frontend layer is responsible for the guided workflow, dataset browsing, training configuration, run monitoring, and result visualization. The frontend communicates with the backend through REST API requests using Axios. All dataset management, training execution, and result retrieval operations are performed through the backend API layer, allowing the frontend to remain independent from the database and machine learning execution environment.

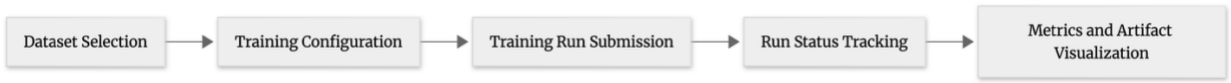

Figure 40: Frontend pipeline of the ML Training Wizard

4.8.1 Technology Stack and Project Layout

The technology stack of the frontend layer consists of various modern technologies and frameworks. The implementation is based on Vue 3 together with the Vite build toolchain [42, 40]. Vue Router is used for navigation across pages, while Pinia is used for managing application state during the wizard's workflow [33, 41]. Regarding the optical aspect of the interface, Apache Echarts is used for metric and time-series for telemetry anomalies visualization, while Tailwind CSS is used for styling, and achieving responsiveness and layout management throughout the application's interface [4]. The frontend source code is organized under the `/src` directory which contains:

- `main.js`: application bootstrap
- `App.vue`: application shell and main layout
- `router/`: route definitions
- `api/`: Axios API client
- `stores/`: Pinia stores
- `views/`: page-level Vue components
- `components/`: reusable interface components

- `composables/`: reusable frontend logic

4.8.2 Application Shell and Routing

The application is initialized through `src/main.js`, which registers Vue Router, Pinia, and the main application shell before mounting the Vue application. `App.vue` defines the main interface layout consisting of the header, sidebar, and the central content region rendered through `RouterView`. The wizard layout applies route-specific behaviour to keep the multi-step workflow inside a controlled viewport. The routing structure includes views for `datasets`, `training runs`, `results`, `settings`, and the guided wizard workflow. The wizard itself is implemented using nested routes where each step is represented as a separate view. This structure allows users to move between workflow stages while preserving the current configuration state throughout the process.

/apps/frontend/src/router/index.js

```
{
  path: '/wizard',
  component: () => import('../views/WizardLayout.vue'),
  children: [
    { path: '', redirect: 'phase' },
    { path: 'phase', component: () => import('../views/WizardPhase.vue') },
    { path: 'task', component: () => import('../views/WizardTask.vue') },
    { path: 'datasets', component: () => import('../views/WizardDatasets.vue')
},
    { path: 'params', component: () => import('../views/WizardParams.vue') },
    { path: 'review', component: () => import('../views/WizardReview.vue') },
  ]
},
```

Figure 41: Wizard nested route structure

4.8.3 API Communication and Backend Integration

Communication between the frontend and backend is implemented using Axios through a dedicated API client module, located in `src/api/client.js`. The API base URL is configured through environment variables, allowing the frontend to communicate with different backend environments without modifying the application source code directly.

The frontend uses dedicated API helpers for:

- dataset retrieval
- dataset upload,
- training run creation
- training run listing
- result retrieval

The frontend layer does not directly access the database, filesystem, or machine learning services. All operations are performed through backend API endpoints, allowing the backend to act as the central orchestration layer of the workflow as discussed earlier in Section 4.3.

4.8.4 Wizard Workflow and State Management

The machine learning workflow is implemented as a guided multi-step wizard. The guided wizard is the main interaction component of the platform and is responsible for guiding users through the pipeline. A shared workflow state is stored inside the Pinia store located in `src/stores/wizard.js`.

The stored state includes the selected task type, dataset, algorithm, and training parameters. This approach allows the frontend to maintain workflow consistency while users navigate between different stages of the wizard.

The implemented wizard views include:

- `WizardPhase.vue`
- `WizardTask.vue`
- `WizardDatasets.vue`
- `WizardParams.vue`
- `WizardReview.vue`

Each view is responsible for a specific workflow step. The dataset selection view retrieves available datasets and their metadata, the parameter configuration view dynamically exposes algorithm-specific parameters depending on the selected learning task. For classification workflows, the interface exposes parameter configuration for Logistic Regression and Random

Forest algorithms, while for anomaly detection workflows, the interface exposes Isolation Forest parameters including contamination and max_samples. The final review step summarizes the selected configuration before submitting the training request to the backend API.

4.8.5 Run Monitoring and Result Visualization

The frontend includes dedicated views for monitoring training runs and visualizing generated outputs. `Runs.vue` retrieves the available training runs from the backend API and displays their execution status while linking each run to its corresponding results page. `Results.vue` retrieves the selected run information and displays evaluation metrics, generated artifacts, visualization outputs, and error messages when a training run fails. For classification workflows, the frontend displays scalar metrics such as accuracy, precision, recall, and F1-score together with the generated confusion matrix returned by the backend. For telemetry-oriented anomaly detection workflows, the frontend loads `timeseries.json` and renders time-series visualizations using Apache ECharts. The generated charts visualize anomaly scores and labels aligned with the original time axis produced during preprocessing and training. The visualization layer is implemented independently from the training workflow, allowing generated artifacts and results to be displayed without directly coupling the frontend to the machine learning execution logic.

4.8.6 Additional Views and User Utilities

Additional frontend views such as are implemented for dataset browsing, settings management, and user guidance. The dataset view allows users to inspect available datasets outside the guided workflow, while the settings and user guide views provide additional usability and navigation support within the platform. The frontend structure was designed to keep presentation logic separated from preprocessing, training execution, and backend orchestration responsibilities, allowing the interface to remain modular and maintainable throughout the implementation.

4.9 Conclusion

The implementation of the ML Training Wizard combines modern web technologies, containerized deployment practices, and machine learning workflows into a unified platform for IoT and cyber-physical system security datasets. Laravel, Vue.js, Python, MySQL, and Docker Compose were integrated together in order to support the end-to-end wizard’s workflow.

The integration of a well-established backend and interactive frontend enhanced supervised classification and telemetry-oriented anomaly detection workflows to ensure that the system can meet the strict demands of IoT security visualizations. Container technology and the modern way of deploying the application makes it easy to host and scale the application [7, 10, 20, 28].

As IoT and cyber-physical systems continue to evolve, the implemented system demonstrates a robust foundation on which further evolution and expansion can ocur as new security analysis, anomaly detection, and malicious activity identification requirements emerge [6, 16, 31].

Chapter 5

Practical Use and System Evaluation

5.1. Introduction

This chapter discusses the deployment and assessment of the IoT ML Training Wizard as proposed in this research. To be specific, real SWAT datasets of the iTrust Singapore Programme along with some public KDD-style NIDS datasets are used to show how the system can effectively work with various types of IoT data and generate informative visualization for IoT security and anomaly detection analysis [25, 39].

5.2 Testing and Evaluation Strategy

The testing and evaluation process focuses on verifying the correct operation of the platform under a complete end-to-end workflow. Validation included service startup, database initialization, dataset registration, training execution, artifact generation, and frontend result visualization. The primary startup workflow uses the `scripts/dev/up.sh` script together with `scripts/dev/up_full.sh` for a completely clean “cold-started” environment.

These scripts start the Docker Compose stack, initialize the database, execute Laravel migrations, and register the processed datasets used by the platform.

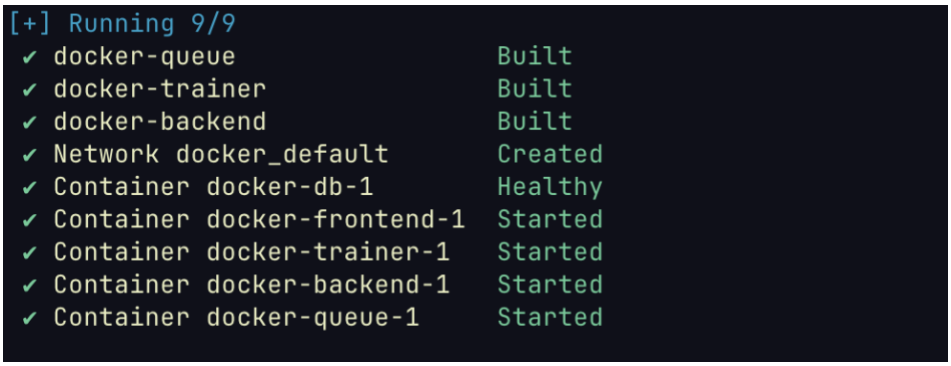

Figure 42: Execution of the up.sh script

After startup, the backend API can be validated through the `/api/datasets` endpoint, while the frontend can be accessed through the configured frontend service port. Automated backend tests were implemented using the Laravel testing framework in order to validate dataset management and training run workflows.

These tests verify API responses, request validation, and training run execution logic during development. End-to-end validation was performed through both classification and anomaly detection workflows executed from the frontend wizard. Successful validation required the training runs to complete correctly, update their execution status, and generate the expected output artifacts inside the `datasets/runs` directory. Validation also included failure handling scenarios where invalid configurations or failed executions return appropriate error messages through both the backend API and the frontend interface.

5.3 Case Study – Supervised Classification

This case study demonstrates a full supervised classification run using a processed Network Intrusion Detection dataset that contains binary benign and malicious traffic labels. The case study aims to evaluate the end-to-end execution of the system [39].

5.3.1 Main Dashboard

The IoT ML Training Wizard dashboard is depicted in figure 42:

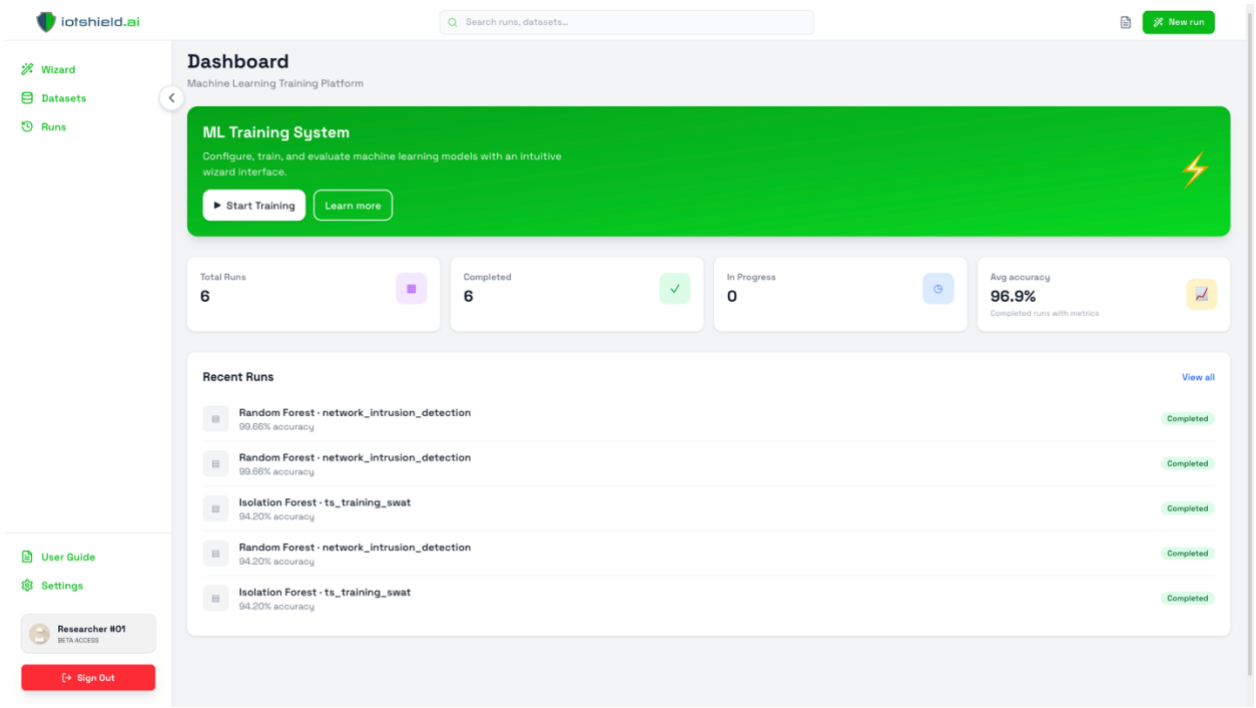

Figure 43: Main dashboard landing page

The dashboard is the landing page of the system and provides a quick overview of the executed runs displaying useful insightss such as the amount of submitted runs, how many of those were completed, how many are still in progress along with their overall average accuracy. In addition, the dashboard presents the last five runs along with their status, while allowing users to start a new execution through the “Start Training” button or read the systems handbook through the “learn more” button.

5.3.2 Experiment Type and ML Task Selection

The first step of submitting a new experiment is selecting whether it should be trainer or evaluation run, For this case study, a training run was selected.

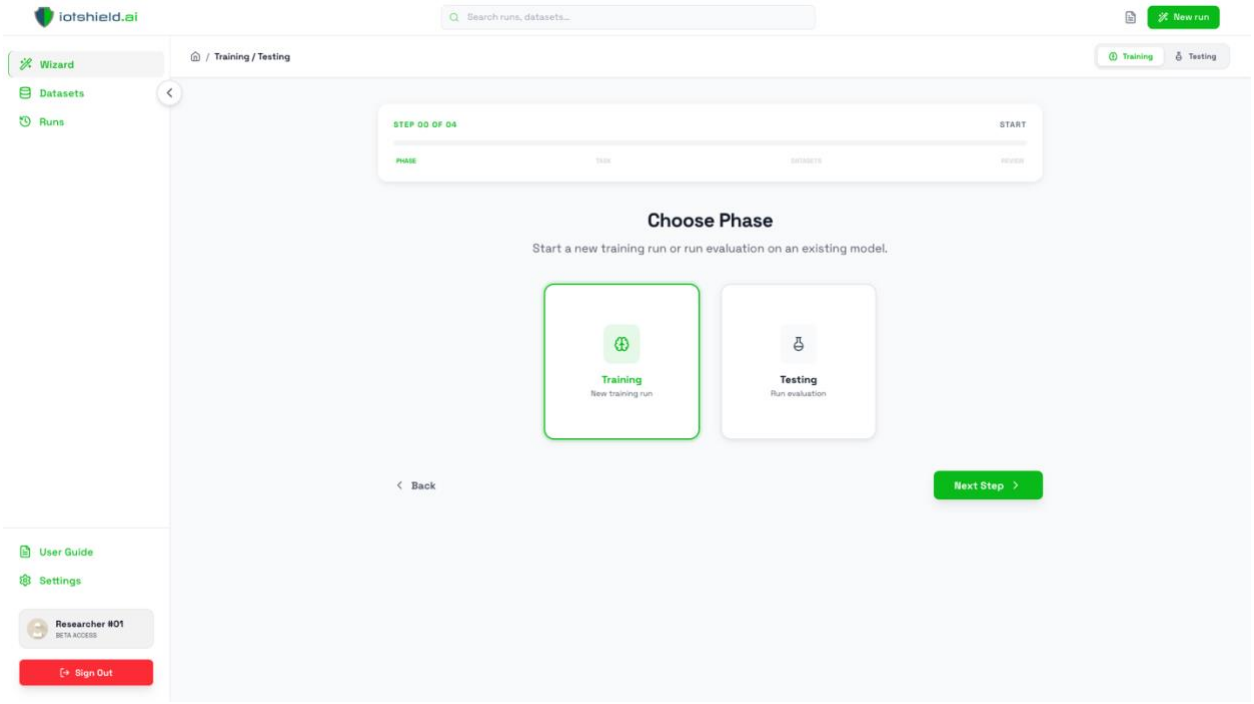

Figure 44: Experiment type selection

After selecting the experiment type, the user must then determine the appropriate machine learning task for the experiment. This version of the system supports classification and anomaly detection, while regression and clustering options are added but appear unavailable for future work implementation. Since the case study focuses on supervised execution, classification is selected as shown in the figure 44 below.

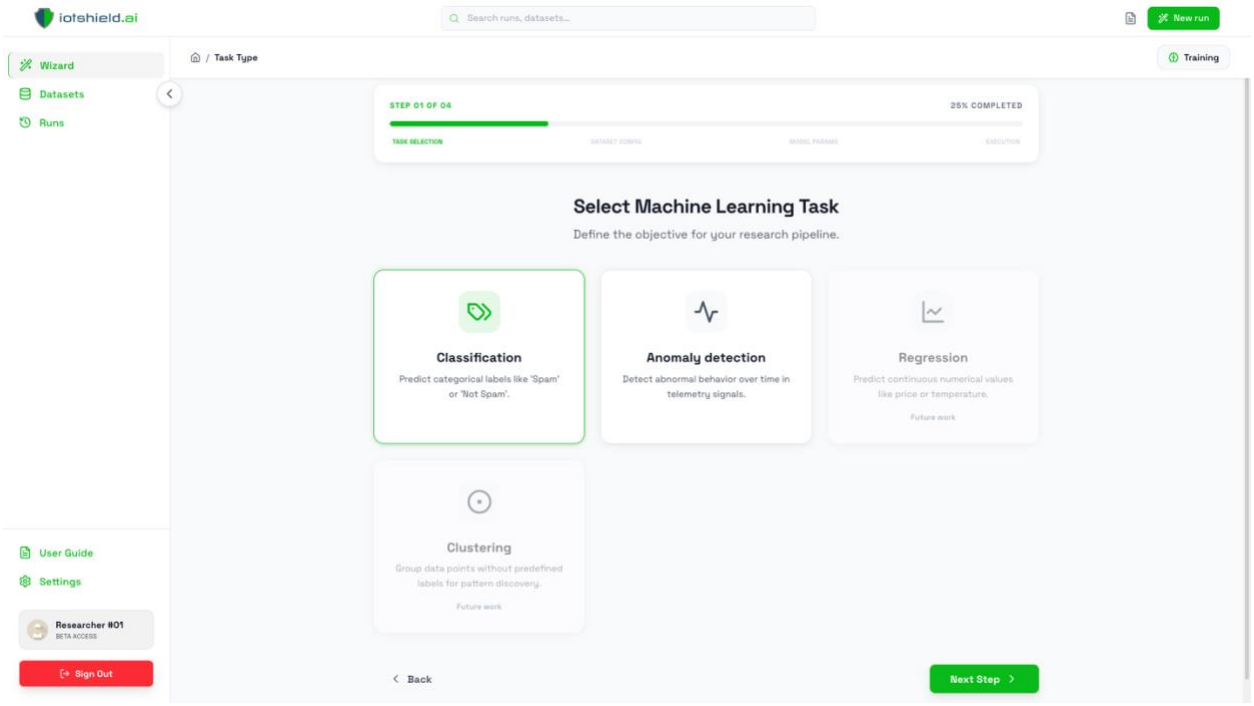

Figure 45: Machine learning task selection

5.3.3 Dataset Selection

The next stage of the workflow focuses on dataset selection and inspection. The dataset selection interface retrieves the registered and already processed datasets from the backend API and displays their associated metadata such as task type, their features, target column label distribution and their description notes giving the user the ability to easily differentiate and select the most suitable or desired dataset.

In addition to selecting existing datasets, the platform also provides a dedicated dataset management interface where users can upload and register new datasets directly through the frontend application, in case any new datasets are needed. The dedicated `datasets` page can be access through the wizard's sidebar. During dataset registration, the user provides the dataset file together with configuration parameters such as the dataset identifier, display name, target column, and positive class label where required.

The figures below demonstrate the dataset selection phase and the `datasets` page accordingly:

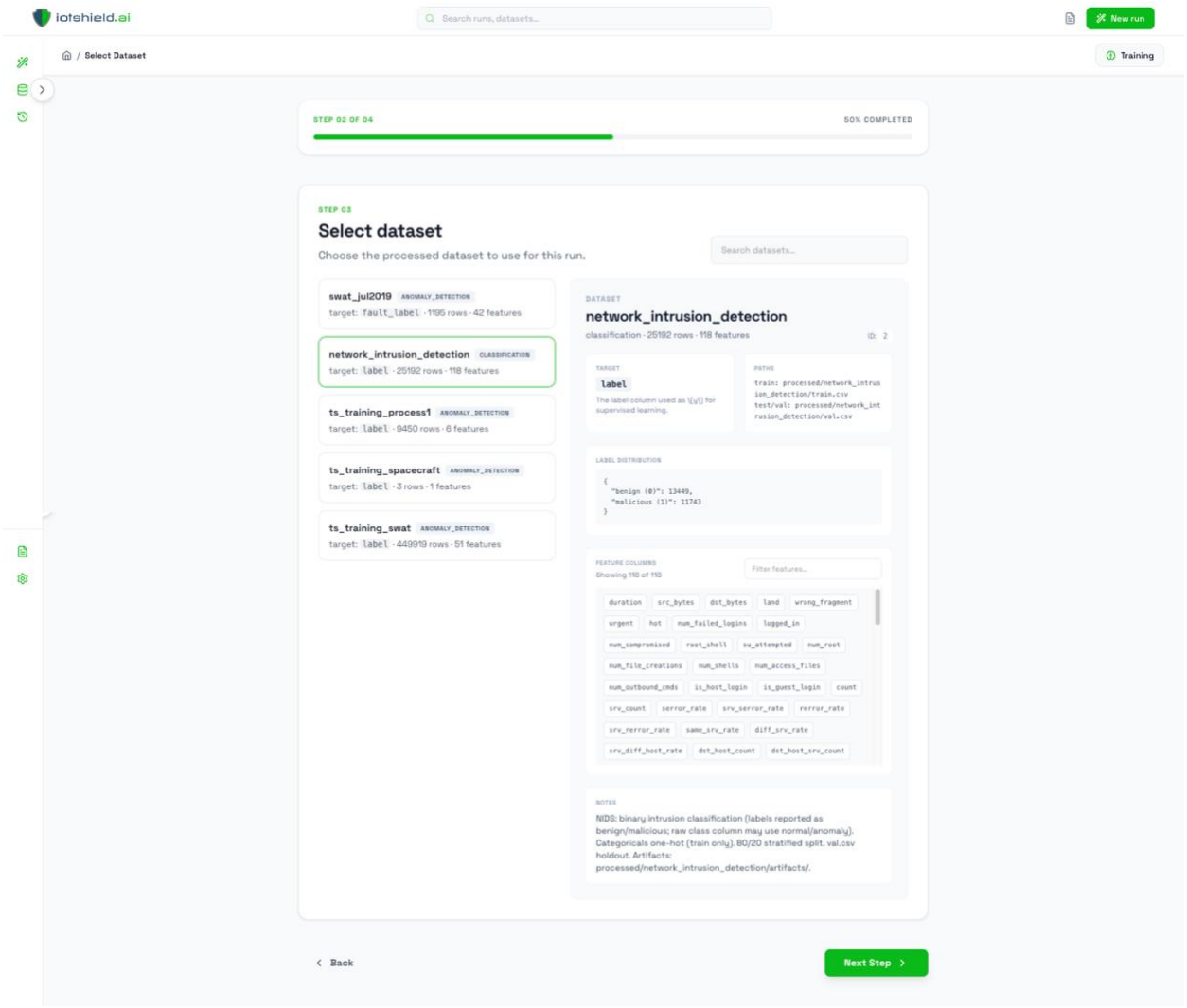

Figure 46: Dataset selection phase

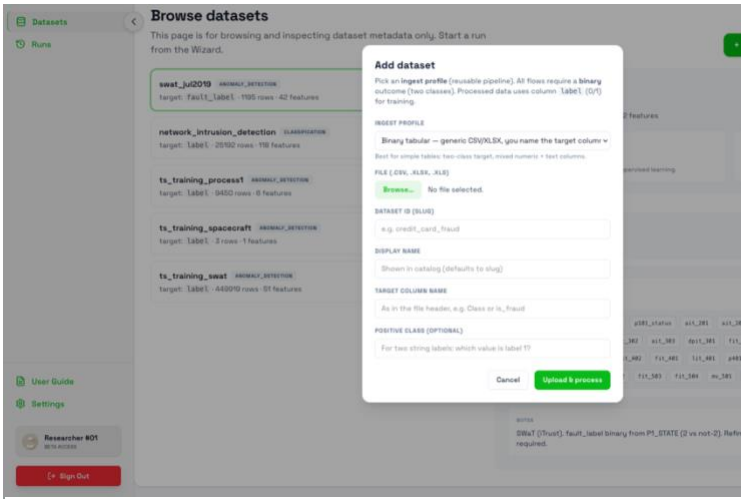

Figure 47: Dataset addition

5.3.4 Experiment Configuration and Review

Proceeding through the dataset selection phase, the third step of the experiment workflow consists of the final experiment configuration. The user is called to choose the desired machine learning algorithm to be applied on the experiment. For Classification tasks Random Forest and Logistic Regression algorithms are available. In addition to the classifier, users are available to set their own hyperparameters or use the default values as shown in the figure below.

This case study follows Random Forest [8].

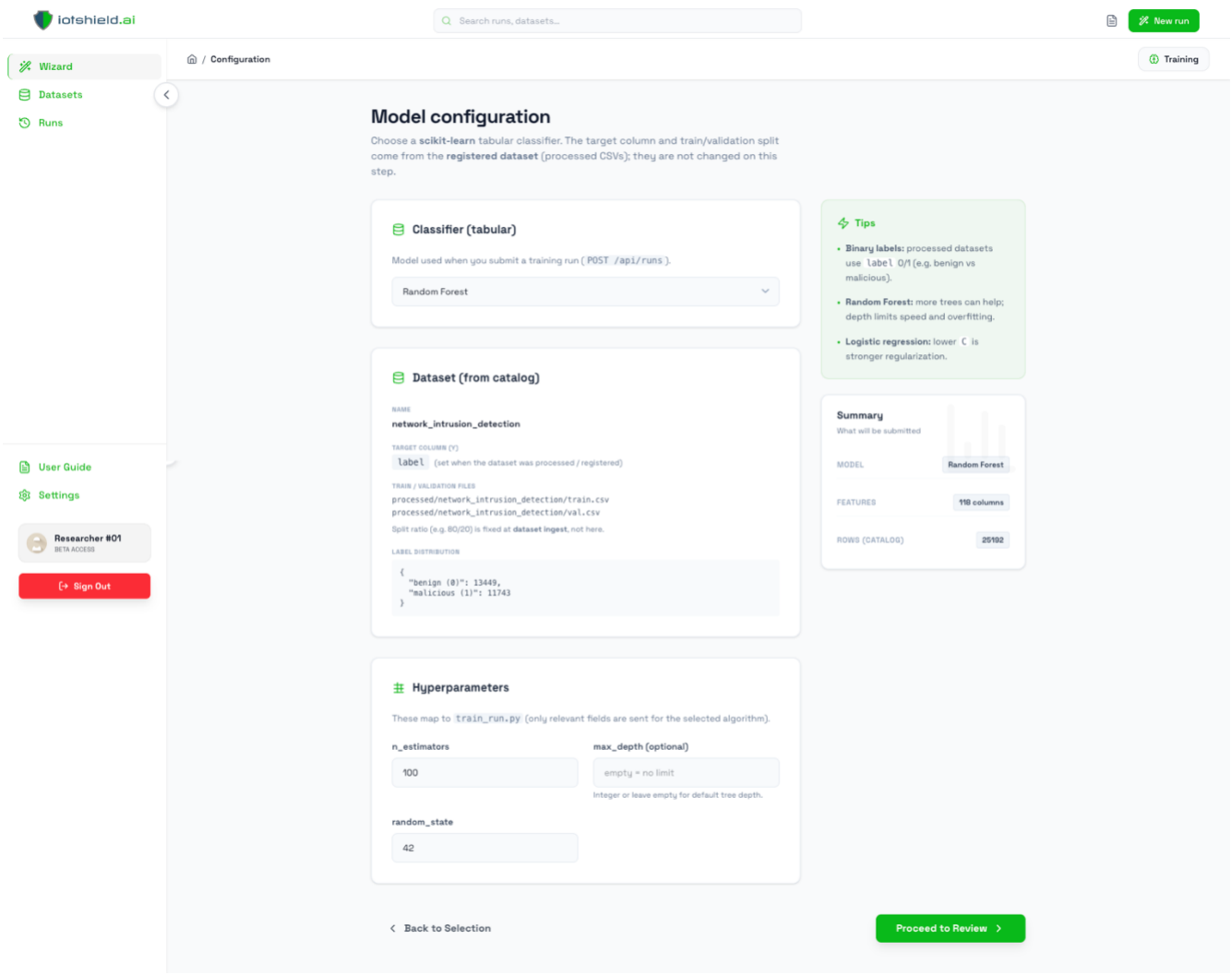

Figure 48: Model configuration step

Once the experiment has been configured, the user reaches the final stage of the workflow, namely the final review page. This interface allows the user to verify the selected experiment type, machine learning algorithm, dataset, and execution configuration before submitting the training run. The page acts as a confirmation step prior to job execution, reducing the possibility of incorrect experiment configuration before asynchronous training begins.

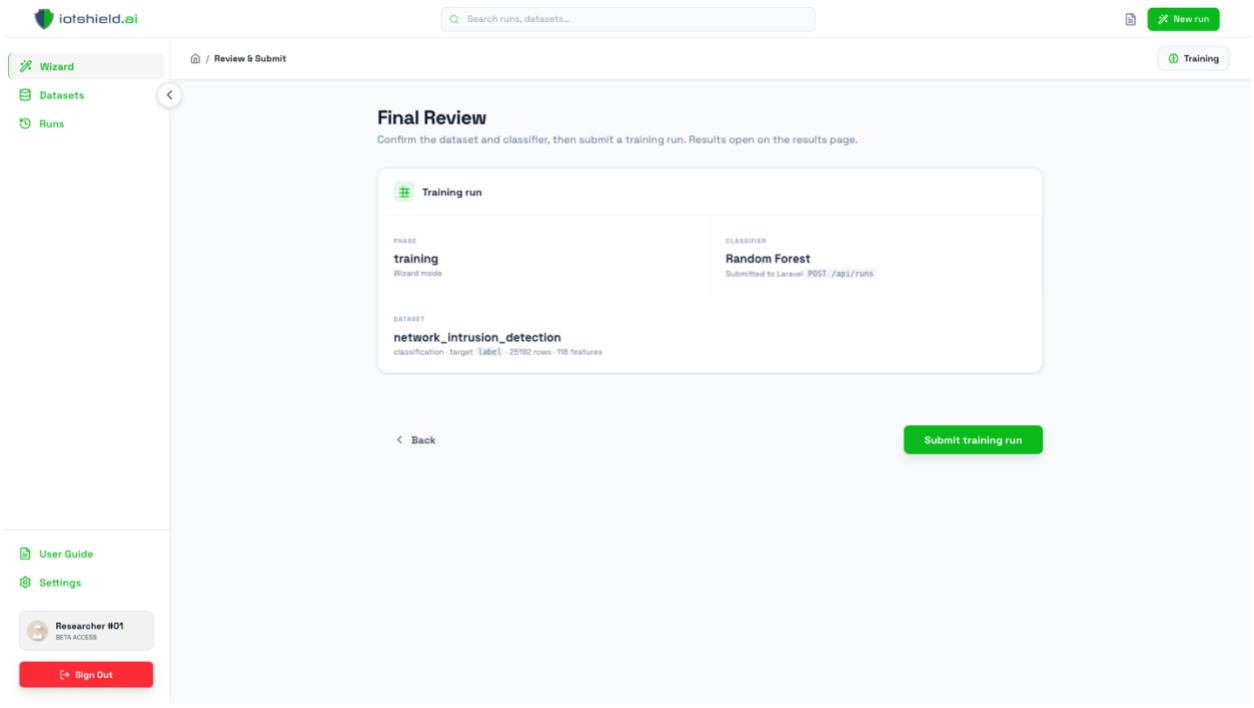

Figure 49: Final review page

5.3.5 Experiment Results

After the training run is submitted and successfully completed, the platform redirects the user to the results interface where the generated outputs and evaluation metrics are presented. The results page provides a summarized overview of the executed experiment together with the generated artifacts and execution metadata. For the supervised classification workflow, the interface displays scalar evaluation metrics including accuracy, precision, recall, and F1-score. In addition, the generated confusion matrix visualizes the relationship between correctly and incorrectly classified benign and malicious samples, allowing the classification performance of the model to be interpreted more effectively.

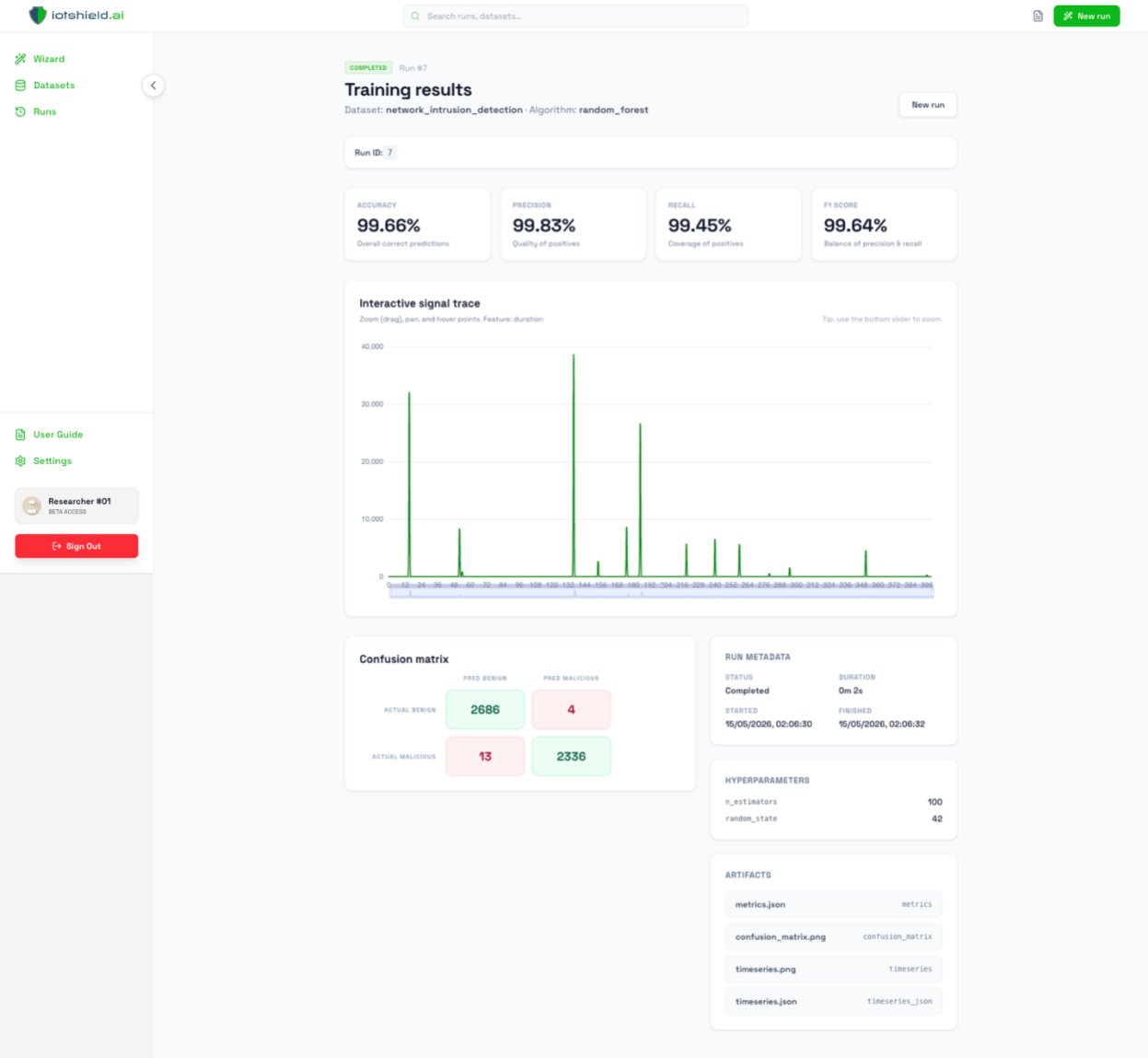

Figure 50: Classification results page

The results interface also presents additional execution information such as run status, execution duration, selected hyperparameters, and generated artifacts including metrics files, confusion matrix images, and visualization outputs. Interactive visualizations generated from the processed dataset are also displayed in order to provide additional insight into the behaviour of the evaluated data. Overall, the results page centralizes the outputs of the machine learning workflow into a single interface, allowing users to inspect experiment performance, review generated artifacts, and evaluate the effectiveness of the executed classification model.

5.4. Case Study – Telemetry Anomaly Detection

This case study demonstrates a telemetry-oriented anomaly detection workflow using a Secure Water Treatment (SWaT) dataset, attained from the iTrust cybersecurity research centre at the Singapore University of Technology and Design (SUTD). Similar to the supervised classification workflow, the purpose of this case study is to evaluate the end-to-end execution of the system, including dataset selection, experiment configuration, asynchronous execution, artifact generation, and result visualization for telemetry type datasets [25].

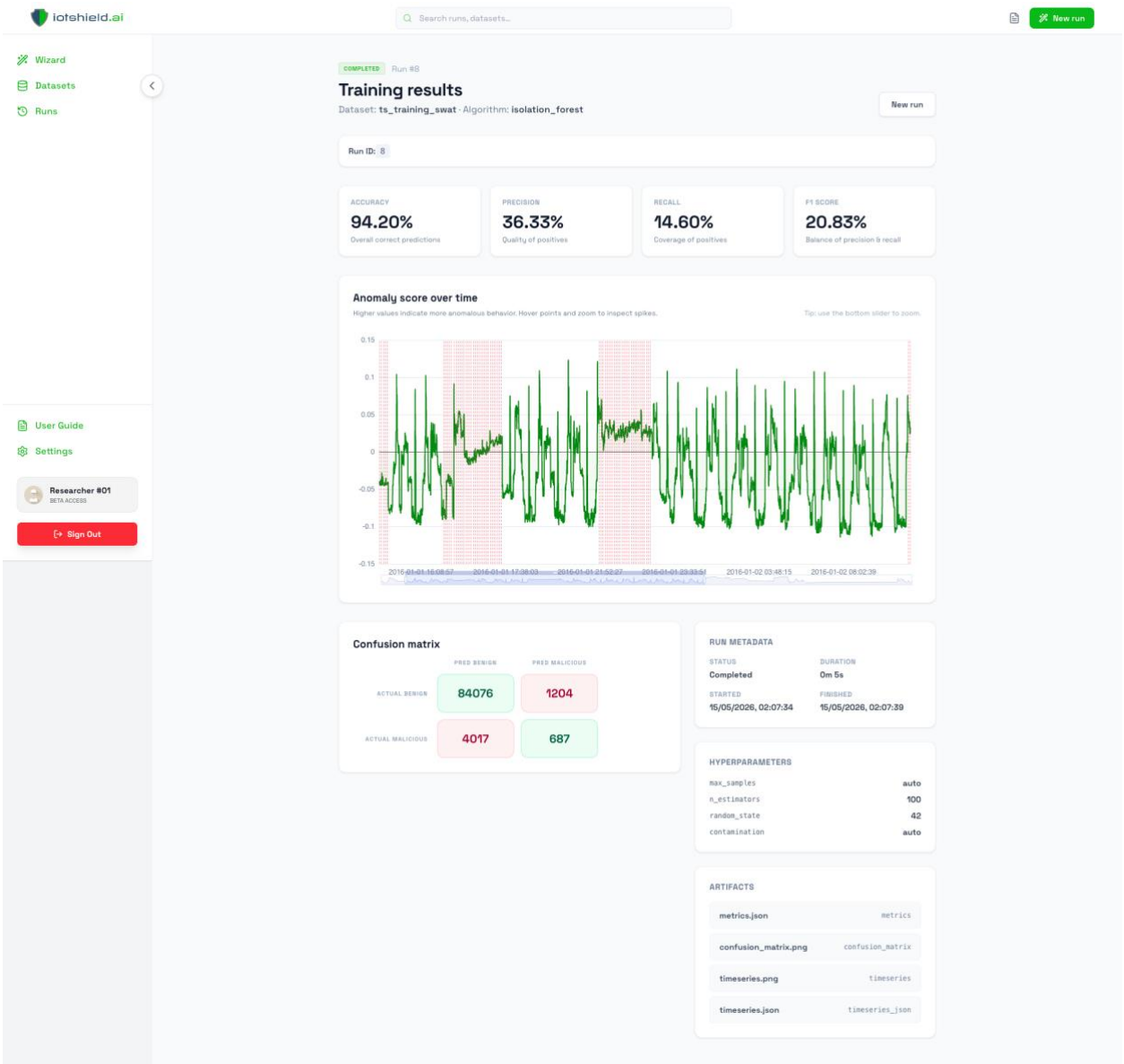

Figure 51: Anomaly detection results page

The anomaly detection workflow follows the same guided wizard process presented in the previous classification case study. The main difference lies in the selected machine learning task, algorithm configuration, and generated outputs. For anomaly detection workflows, the platform currently supports the Isolation Forest algorithm together with its corresponding hyperparameter configuration, including contamination and sampling parameters [24].

After successful execution, the platform redirects the user to the results interface where the generated anomaly detection outputs are visualized. Similar to the supervised classification workflow, the interface displays scalar evaluation metrics including accuracy, precision, recall, F1-score, and the generated confusion matrix. In addition, the anomaly detection workflow provides an interactive time-series visualization that enhances observability and inspection capabilities across the telemetry sequence. The generated graph supports zooming and horizontal navigation, allowing users to inspect specific time periods in greater detail. Vertical red markers are also used to indicate detected anomaly periods, improving the interpretability of anomalous behaviour throughout the dataset [10].

5.5 System Evaluation

5.5.1 Usability

The implemented platform demonstrates high usability through the guided multi-step wizard workflow. The frontend interface organizes dataset selection, algorithm configuration, parameter configuration, execution monitoring, and result visualization into separate structured stages, reducing workflow complexity for the user.

The separation between classification and anomaly detection workflows also improves usability by dynamically exposing only the relevant datasets, algorithms, and parameter configurations required for each experiment type.

5.5.2 Effectiveness

The evaluation is demonstrating that the system successfully supports both supervised classification and telemetry-oriented anomaly detection workflows. The platform was capable of executing preprocessing, training, asynchronous orchestration, artifact generation, and result visualization through a centralized workflow.

The generated confusion matrices, evaluation metrics, anomaly scores, and time-series visualizations confirm that the implemented workflow can support IoT and cyber-physical system malicious activity detection and security analysis [10, 20].

5.5.3 Reproducibility

The platform demonstrates reproducibility through the use of containerization, structured dataset registration, generated configuration files, and shared artifact management. Docker Compose allows the same environment to be recreated across different systems, while generated `run_config.json` files preserve the exact execution configuration of each training run. The standardized preprocessing and metadata generation workflow also ensures that datasets can be reused consistently across multiple experiments without modifying the backend or training pipeline [7, 28, 32].

5.5.4 Limitations

Although the implemented platform successfully demonstrates the proposed workflow, several limitations remain. The current implementation supports a limited number of machine learning algorithms and focuses primarily on supervised classification and Isolation Forest anomaly detection workflows.

In addition, the deployment environment is currently based on a local Docker Compose architecture and does not include distributed execution, cloud-native orchestation, or large-scale production deployment features. The evaluation was also limited to the supported datasets included within the scope of the thesis.

5.6 Alignment with Research Objectives

The implemented system successfully aligns with the research objectives defined in Chapter 1. The platform provides a centralized and reproducible environment for IoT and cyber-physical system machine learning workflows through dataset preprocessing, training orchestration, asynchronous execution, result visualization, and containerized deployment.

The frontend wizard simplifies machine learning experimentation through a structured workflow, while the backend and trainer services support extensibility for additional datasets, preprocessing pipelines, algorithms, and visualization workflows. The implemented anomaly detection and

classification pipelines demonstrate the capability of the system to support IoT security analysis and malicious activity detection through an integrated environment.

5.7 Conclusion

The practical evaluation of the ML Training Wizard demonstrates that the implemented platform can successfully support supervised classification and telemetry-oriented anomaly detection workflows within a unified machine learning environment. The integration of preprocessing, backend orchestration, asynchronous execution, visualization, and containerized deployment allows the platform to support reproducible IoT and cyber-physical system security experimentation.

The generated metrics, confusion matrices, anomaly scores, and telemetry visualizations confirm the effectiveness of the implemented workflows for malicious activity detection and security analysis. Overall, the implemented system establishes a robust and extensible foundation for future IoT/CPS machine learning and anomaly detection research.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis addressed the growing need for reproducible and centralized machine learning workflows for IoT and cyber-physical system security datasets. As IoT infrastructures continue to expand across domains such as industrial automation, healthcare, smart environments, and critical infrastructure, the volume and complexity of generated telemetry and network data continue to increase [16, 31]. This creates a growing demand for systems capable of supporting preprocessing, machine learning experimentation, anomaly detection, and result visualization through structured and scalable workflows [10, 20, 32]. The main objective of this research was the design and implementation of a containerized ML Training Wizard capable of supporting supervised classification and telemetry-oriented anomaly detection workflows through a unified frontend and backend environment. The implemented platform combines dataset preprocessing, asynchronous execution, machine learning training, artifact generation, and visualization into a single workflow designed to simplify machine learning experimentation on heterogeneous IoT/CPS security datasets.

The methodology of this research included the analysis of existing machine learning workflow limitations, the design of a modular architecture, the implementation of a multi-service platform using modern web and machine learning technologies, and the evaluation of the system through classification and anomaly detection case studies.

The implementation illustrated that Laravel, Vue.js, Python, Docker, and MySQL can be effectively integrated together in order to support end-to-end machine learning experimentation through a guided frontend interface and asynchronous backend execution pipeline. The resulting platform allows datasets to be processed, registered, trained, visualized, and evaluated through a reproducible and extensible environment [3, 7, 34].

6.2 Key Findings and Contributions

The implementation and evaluation of the ML Training Wizard produced several important findings and contributions related to IoT and cyber-physical system security experimentation.

6.2.1 Unified Machine Learning Workflow

One of the main contributions of this work is the integration of preprocessing, training orchestration, artifact generation, and visualization into a centralized workflow. The implemented platform demonstrates that heterogeneous machine learning stages can be combined into a single guided environment without requiring direct interaction with multiple isolated tools or services [34, 43].

6.2.2 Support for Classification and Anomaly Detection

The system successfully illustrated both supervised classification and telemetry-oriented anomaly detection workflows. Classification experiments supported benign and malicious traffic analysis through Logistic Regression and Random Forest algorithms, while telemetry-oriented anomaly detection workflows were implemented using Isolation Forest together with interactive time-series visualizations [8, 10, 24].

6.2.3 Containerized and Reproducible Architecture

The implementation demonstrated the advantages of containerized machine learning environments for reproducibility and deployment consistency. Docker Compose allowed the frontend, backend, trainer service, database, and queue workers to operate as isolated but connected services while simplifying dependency management across different environments.

6.2.4 Interactive Frontend Workflow

The guided frontend wizard simplified machine learning experimentation by organizing dataset selection, algorithm configuration, execution monitoring, and result visualization into structured workflow stages. The integration of scalar metrics, confusion matrices, and telemetry visualizations improved the interpretability of the generated outputs [7, 28].

6.2.5 Extensible System Design

The modular architecture of the platform allows additional datasets, preprocessing pipelines, machine learning algorithms, and visualization workflows to be integrated without requiring major architectural changes. This provides a flexible foundation for future experimentation and system expansion [3].

6.3 Practical Significance of the Study

The findings of this research demonstrate the practical applicability of centralized and containerized machine learning workflows for IoT and cyber-physical system security analysis.

6.3.1 Simplification of Machine Learning Experimentation

The implemented platform reduces the complexity of machine learning experimentation by integrating preprocessing, training, and visualization into a guided workflow. This simplifies the execution of security-oriented experiments and reduces the operational overhead associated with managing separate machine learning environments manually [32, 43].

6.3.2 Support for Security Analysis and Malicious Activity Detection

The supervised classification and anomaly detection workflows demonstrated the capability of the system to support IoT and cyber-physical system malicious activity detection through machine learning experimentation and telemetry visualization [20, 44].

6.3.3 Reproducibility and Deployment Consistency

The use of Docker Compose and containerized services improves reproducibility and deployment consistency across different development environments. This allows the same workflow to be recreated without manually configuring multiple technology ecosystems and dependencies [7, 32].

6.3.4 Foundation for Future ML Workflow Platforms

The architecture implemented in this research demonstrates how modern web technologies and machine learning services can be integrated into modular experimentation platforms capable of supporting future extensions and larger-scale workflows [3, 14].

6.4 Limitations of the Study

Despite the successful implementation and evaluation of the platform, several limitations remain.

6.4.1 Limited Algorithm Support

The current implementation supports a limited number of machine learning algorithms and primarily focuses on Logistic Regression, Random Forest, and Isolation Forest workflows. Additional supervised and unsupervised algorithms were outside the scope of this research.

6.4.2 Limited Workflow Types

The implemented platform primarily focuses on supervised classification and telemetry-oriented anomaly detection. Other machine learning workflows such as regression, clustering, and advanced forecasting were not fully implemented within the current version of the system.

6.4.3 Local Deployment Environment

The deployment environment is currently based on Docker Compose and local container orchestration. The system was not evaluated under distributed or production-scale cloud deployments.

6.4.4 Limited Dataset Coverage

Although the platform successfully illustrated classification and telemetry-oriented workflows, the evaluation was limited to the datasets included within the scope of the thesis and does not cover the full diversity of real-world IoT/CPS environments [6, 31].

6.4.5 Limited Multi-User Functionality

Authentication, authorization, tenant isolation, and advanced multi-user management capabilities were not fully implemented within the current platform version and remain outside the scope of the prototype.

6.5 Future Work

Several future research and development directions emerge from the findings and limitations of this study.

6.5.1 Additional Machine Learning Workflows

Future versions of the platform could extend support to additional workflow types such as regression, clustering, forecasting, and advanced anomaly detection techniques together with their corresponding evaluation metrics and visualization methods.

6.5.2 Extended Algorithm Support

Additional supervised and unsupervised machine learning algorithms could be integrated into the trainer service in order to support broader experimentation and comparative evaluation across different IoT/CPS security datasets.

6.5.3 Kubernetes and Cloud-Native Deployment

Future work could focus on migrating the deployment environment from Docker Compose to Kubernetes in order to support production-scale orchestration, workload scaling, resource management, and cloud-native deployment scenarios [22].

6.5.4 Real-Time and Streaming Telemetry Processing

The current implementation primarily focuses on static datasets. Future research could extend the platform toward real-time telemetry ingestion, streaming anomaly detection, and continuous monitoring workflows for live IoT environments [21, 26].

6.5.5 Multi-User and Multi-Tenant Support

Future platform extensions could include authentication, authorization, dataset isolation, quota management, audit logging, and collaborative experiment management features suitable for institutional or multi-user environments.

6.5.6 Explainability and Advanced Visualization

Additional explainability techniques and advanced visualization approaches could improve the interpretability of generated machine learning results and anomaly detection outputs for security analysts and researchers.

6.6 Concluding Remarks

IoT and cyber-physical systems continue to evolve rapidly, increasing both the scale of generated telemetry data and the complexity of associated security challenges [6, 16, 31].

As machine learning becomes increasingly important for anomaly detection and malicious activity analysis, the demand for reproducible and extensible experimentation platforms will continue to grow [10].

This research demonstrated that modern web technologies, containerized deployment environments, and machine learning workflows can be effectively integrated into a unified platform capable of supporting preprocessing, training execution, telemetry visualization, and experiment management for IoT/CPS security datasets.

The implemented ML Training Wizard establishes a modular foundation for future machine learning experimentation and anomaly detection research. Through further extension, scalability improvements, and integration of additional machine learning capabilities, platforms of this type can contribute toward more effective IoT and cyber-physical system security analysis in both research and real-world environments.

References

[1] Z. Ahmad, A. S. Khan, C. W. Shiang, J. Abdullah, and F. Ahmad, “Network intrusion detection system: A systematic study of machine learning and deep learning approaches,” Transactions on Emerging Telecommunications Technologies, vol. 32, no. 1, Art. no. e4150, 2021, doi: 10.1002/ett.4150.

[2] M. Ahmed, A. N. Mahmood, and J. Hu, “A survey of network anomaly detection techniques,” Journal of Network and Computer Applications, vol. 60, pp. 19–31, 2016, doi: 10.1016/j.jnca.2015.11.016.

[3] S. Amershi et al., “Software engineering for machine learning: A case study,” in Proc. IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 291–300, doi: 10.1109/ICSE-SEIP.2019.00042.

[4] Apache ECharts, Apache ECharts Documentation. [Online]. Available: https://echarts.apache.org/en/index.html

[5] Apache Software Foundation, Apache Airflow Documentation, 2026. [Online]. Available: https://airflow.apache.org/docs/

[6] Y. Ashibani and Q. H. Mahmoud, “Cyber physical systems security: Analysis, challenges and solutions,” Computers & Security, vol. 68, pp. 81–97, 2017, doi: 10.1016/j.cose.2017.04.005.

[7] C. Boettiger, “An introduction to Docker for reproducible research,” ACM SIGOPS Operating Systems Review, vol. 49, no. 1, pp. 71–79, 2015, doi: 10.1145/2723872.2723882.

[8] L. Breiman, “Random forests,” Machine Learning, vol. 45, no. 1, pp. 5–32, 2001, doi: 10.1023/A:1010933404324.

[9] A. L. Buczak and E. Guven, “A survey of data mining and machine learning methods for cyber security intrusion detection,” IEEE Communications Surveys & Tutorials, vol. 18, no. 2, pp. 1153–1176, 2016, doi: 10.1109/COMST.2015.2494502.

[10] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” ACM Computing Surveys, vol. 41, no. 3, pp. 1–58, 2009, doi: 10.1145/1541880.1541882.

[11] A. A. Diro and N. Chilamkurti, “Distributed attack detection scheme using deep learning approach for Internet of Things,” Future Generation Computer Systems, vol. 82, pp. 761–768, 2018, doi: 10.1016/j.future.2017.08.043.

[12] European Union Agency for Cybersecurity, Good Practices for Security of IoT: Secure Software Development Lifecycle, ENISA, 2019. [Online]. Available: https://www.enisa.europa.eu/publications/good-practices-for-security-of-iot-1

[13] M. Fagan, K. N. Megas, K. Scarfone, and M. Smith, IoT Device Cybersecurity Capability Core Baseline, NIST Interagency/Internal Report 8259A, National Institute of Standards and Technology, 2020, doi: 10.6028/NIST.IR.8259A.

[14] L. Faubel and K. Schmid, “A systematic analysis of MLOps features and platforms,” in Proc. Euromicro Conference Series on Software Engineering and Advanced Applications / DSD-SEAA Works in Progress Session, 2024.

[15] M. Frustaci, P. Pace, G. Aloi, and G. Fortino, “Evaluating critical security issues of the IoT world: Present and future challenges,” IEEE Internet of Things Journal, vol. 5, no. 4, pp. 2483–2495, 2018, doi: 10.1109/JIOT.2017.2767291.

[16] C. Greer, M. Burns, D. Wollman, and E. Griffor, Cyber-Physical Systems and Internet of Things, NIST Special Publication 1900-202, National Institute of Standards and Technology, 2019, doi: 10.6028/NIST.SP.1900-202.

[17] V. Gudivada, A. Apon, and J. Ding, “Data quality considerations for big data and machine learning: Going beyond data cleaning and transformations,” International Journal on Advances in Software, vol. 10, no. 1–2, pp. 1–20, 2017.

[18] J. Hou, L. Qu, W. Shi, and G. Li, “A survey on Internet of Things security from data perspectives,” Computer Networks, vol. 148, pp. 295–306, 2019, doi: 10.1016/j.comnet.2018.11.026.

[19] A. Humayed, J. Lin, F. Li, and B. Luo, “Cyber-physical systems security—A survey,” IEEE Internet of Things Journal, vol. 4, no. 6, pp. 1802–1831, 2017, doi: 10.1109/JIOT.2017.2703172.

[20] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, “Survey of intrusion detection systems: Techniques, datasets and challenges,” Cybersecurity, vol. 2, no. 20, 2019, doi: 10.1186/s42400-019-0038-7.

[21] B. Kim et al., “A comparative study of time series anomaly detection models for industrial control systems,” Sensors, vol. 23, no. 3, Art. no. 1310, 2023, doi: 10.3390/s23031310.

[22] Kubeflow, Kubeflow Documentation, 2026. [Online]. Available: https://www.kubeflow.org/docs/

[23] I. Lee, “Internet of Things (IoT) cybersecurity: Literature review and IoT cyber risk management,” Future Internet, vol. 12, no. 9, Art. no. 157, 2020, doi: 10.3390/fi12090157.

[24] F. T. Liu, K. M. Ting, and Z.-H. Zhou, “Isolation Forest,” in Proc. IEEE International Conference on Data Mining, 2008, pp. 413–422, doi: 10.1109/ICDM.2008.17.

[25] A. P. Mathur and N. O. Tippenhauer, “SWaT: A water treatment testbed for research and training on ICS security,” in Proc. International Workshop on Cyber-Physical Systems for Smart Water Networks (CySWater), 2016, pp. 31–36, doi: 10.1109/CySWater.2016.7469060.

[26] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du, “Data ingestion for the connected world,” in Proc. Conference on Innovative Data Systems Research (CIDR), 2017.

[27] Y. Meidan et al., “N-BaIoT: Network-based detection of IoT botnet attacks using deep autoencoders,” IEEE Pervasive Computing, vol. 17, no. 3, pp. 12–22, 2018, doi: 10.1109/MPRV.2018.03367731.

[28] D. Merkel, “Docker: Lightweight Linux containers for consistent development and deployment,” Linux Journal, no. 239, 2014.

[29] Y. Mirsky et al., “Kitsune: An ensemble of autoencoders for online network intrusion detection,” in Proc. Network and Distributed System Security Symposium (NDSS), 2018, doi: 10.14722/ndss.2018.23204.

[30] N. Moustafa and J. Slay, “UNSW-NB15: A comprehensive data set for network intrusion detection systems,” in Proc. Military Communications and Information Systems Conference (MilCIS), 2015, pp. 1–6, doi: 10.1109/MilCIS.2015.7348942.

[31] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, “Demystifying IoT security: An exhaustive survey on IoT vulnerabilities and a first empirical look on Internet-scale IoT exploitations,” IEEE Communications Surveys & Tutorials, vol. 21, no. 3, pp. 2702–2733, 2019, doi: 10.1109/COMST.2019.2910750.

[32] J. Pineau et al., “Improving reproducibility in machine learning research,” Journal of Machine Learning Research, vol. 22, no. 164, pp. 1–20, 2021.

[33] Pinia, Pinia Documentation. [Online]. Available: https://pinia.vuejs.org/

[34] D. Sculley et al., “Hidden technical debt in machine learning systems,” in Advances in Neural Information Processing Systems, vol. 28, 2015.

[35] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, “Toward generating a new intrusion detection dataset and intrusion traffic characterization,” in Proc. International Conference on Information Systems Security and Privacy (ICISSP), 2018, pp. 108–116, doi: 10.5220/0006639801080116.

[36] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, “Toward developing a systematic approach to generate benchmark datasets for intrusion detection,” Computers & Security, vol. 31, no. 3, pp. 357–374, 2012, doi: 10.1016/j.cose.2011.12.012.

[37] M. Stauffer, Laravel: Up & Running: A Framework for Building Modern PHP Apps, 2nd ed. O'Reilly Media, 2019.

[38] K. Stouffer et al., Guide to Operational Technology (OT) Security, NIST Special Publication 800-82 Revision 3, National Institute of Standards and Technology, 2023, doi: 10.6028/NIST.SP.800-82r3.

[39] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in Proc. IEEE Symposium on Computational Intelligence for Security and Defense Applications, 2009, pp. 1–6, doi: 10.1109/CISDA.2009.5356528.

[40] Vite, Vite Documentation. [Online]. Available: https://vite.dev/guide/

[41] Vue Router, Vue Router Documentation. [Online]. Available: https://router.vuejs.org/

[42] Vue.js, Vue.js Documentation. [Online]. Available: https://vuejs.org/guide/introduction.html

[43] M. Zaharia et al., "Accelerating the machine learning lifecycle with MLflow," IEEE Data Engineering Bulletin, vol. 41, no. 4, pp. 39–45, 2018.

[44] J. Zhang, L. Pan, Q.-L. Han, C. Chen, S. Wen, and Y. Xiang, "Deep learning based attack detection for cyber-physical system cybersecurity: A survey," IEEE/CAA Journal of Automatica Sinica, vol. 9, no. 3, pp. 377–391, 2022, doi: 10.1109/JAS.2021.1004261.

[45] J. H. Ziegeldorf, O. G. Morchon, and K. Wehrle, "Privacy in the Internet of Things: Threats and challenges," Security and Communication Networks, vol. 7, no. 12, pp. 2728–2742, 2014, doi: 10.1002/sec.795.