UNIVERSITY OF
THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE & TELECOMMUNICATIONS

DETECTION OF ATTACKS IN AUTONOMOUS VEHICLES THROUGH MACHINE LEARNING

TSIMPAS MICHAIL

FINAL THESIS

ADVISOR

KONSTANTINOS KOLOMVATSOS
ASSOCIATE PROFESSOR

Lamia July year 2026

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί ***χωρίς να τα περικλείω σε εισαγωγικά*** *και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

2. Δέχομαι ότι η αυτολεξεί ***παράθεση χωρίς εισαγωγικά****, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ.), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: ……/…../20……

Ο – Η Δηλ.

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

ΠΕΡΙΛΗΨΗ

Τα μη επανδρωμένα εναέρια οχήματα (drones) αναπτύσσονται ολοένα και περισσότερο για εργασίες όπως η παρακολούθηση υποδομών, η εφοδιαστική αλυσίδα και η περιβαλλοντική ανίχνευση. Καθώς αυτά τα δίκτυα αυξάνονται σε μέγεθος και επιχειρησιακή σημασία, αυξάνεται αντίστοιχα και ο κίνδυνος κυβερνοεισβολών που στοχεύουν τα επίπεδα επικοινωνίας και αισθητήρων που τα διατηρούν λειτουργικά. Ο εντοπισμός τέτοιων εισβολών χωρίς να τίθεται σε κίνδυνο η ιδιωτικότητα των δεδομένων που διακινούνται μεταξύ των drone αποτελεί ένα πρόβλημα που οι κεντρικοποιημένες προσεγγίσεις μηχανικής μάθησης (centralized machine learning approaches) αδυνατούν να αντιμετωπίσουν αποτελεσματικά, καθώς απαιτούν τη μεταφορά ακατέργαστων δεδομένων αισθητήρων μέσω του δικτύου και τη συσσώρευσή τους σε έναν και μόνο κόμβο επεξεργασίας.

Η παρούσα διπλωματική εργασία προτείνει και αξιολογεί ένα σύστημα ομοσπονδιακής μάθησης (federated learning) για τον εντοπισμό εισβολών σε ένα προσομοιωμένο δίκτυο πέντε drone, το οποίο εκτελείται σε περιβάλλον προσομοίωσης πραγματικού χρόνου, από την παραγωγή δεδομένων στα drone έως την ολική συνάθροιση του μοντέλου στον διακομιστή. Κάθε drone λειτουργεί ως ανεξάρτητος πελάτης στο χρησιμοποιούμενο Flower framework, εκπαιδεύοντας ένα πλήρως συνδεδεμένο νευρωνικό δίκτυο πέντε επιπέδων σε δεδομένα αισθητήρων που αποθηκεύονται τοπικά και μεταδίδονται μέσω του Apache Kafka. Μετά από κάθε τοπικό κύκλο εκπαίδευσης, μόνο οι ενημερώσεις βαρών και παραμέτρων του μοντέλου διαβιβάζονται σε έναν κεντρικό διακομιστή, ο οποίος τις συναθροίζει με τον αλγόριθμο FedAvg και αναδιανέμει ένα ενιαίο, βελτιωμένο καθολικό μοντέλο. Το κεντρικό επιχείρημα αυτής της εργασίας είναι ότι τα ακατέργαστα δεδομένα των αισθητήρων δεν εγκαταλείπουν ποτέ το drone που τα παρήγαγε.

Αξιολογήθηκαν δύο πειραματικές διαμορφώσεις όπου στην μία κάθε drone εκπαιδεύτηκε σε 20.000 εγγραφές αισθητήρων, δηλαδή το σύνολο των δεδομένων, και μία στην οποία η εκπαίδευση ξεκινούσε μόλις μετά τη συλλογή 2.000 εγγραφών, ενώ οι υπόλοιπες εξακολουθούσαν να καταναλώνονται. Και οι δύο διαμορφώσεις επέτυχαν καθολική ακρίβεια 100% και τέλειο βαθμό recall σε όλα τα δείγματα εισβολής του συνόλου αξιολόγησης, που σημαίνει ότι καμία επίθεση δεν πέρασε απαρατήρητη σε καμία από τις δύο εκτελέσεις. Επιπλέον, μια συγκριτική ανάλυση έναντι ενός κεντρικοποιημένου συστήματος εντοπισμού εισβολών, βασισμένου στην ίδια υποδομή Kafka, αναδεικνύει το πλεονέκτημα ιδιωτικότητας που προσφέρει η ομοσπονδιακή προσέγγιση.

Τα αποτελέσματα επιβεβαιώνουν ότι η ομοσπονδιακή μάθηση αποτελεί μια πρακτικά βιώσιμη και απόρρητη εναλλακτική λύση έναντι του κεντρικοποιημένου εντοπισμού εισβολών για δίκτυα πολλαπλών drone. Η ποιότητα εντοπισμού είναι τουλάχιστον εξίσου ισχυρή με αυτήν που μπορεί να προσφέρει ένα κεντρικοποιημένο μοντέλο, ενώ οι αρχιτεκτονικές ιδιότητες της ομοσπονδίας καθιστούν το σύστημα σημαντικά πιο ανθεκτικό και λιγότερο εκτεθειμένο σε επιθέσεις επιπέδου δεδομένων.

ABSTRACT

Unmanned aerial vehicles are increasingly deployed in group for tasks such as infrastructure monitoring, logistics and environmental sensing. As these networks grow in size and operational importance, so does the risk of cyber intrusions targeting the communication and sensor layers that keep them functional. Detecting such intrusions without compromising the privacy of the data flowing between drones is a problem that centralized approaches struggle to solve well, since they require raw sensor readings to travel over the network and accumulate on a single processing node.

This thesis proposes and evaluates a federated learning system for intrusion detection in a simulated five-drone network, while it runs on a real time simulation network, from the drone production to the global model aggregation on the server. Each drone operates detection in a simulated five-drone network. Moreover, each unit functions as an independent Flower client, training a five layer fully connected neural network on locally buffered sensor data streamed through Apache Kafka. After each local training epoch, only the model weight updates are transmitted to a central Flower server, which aggregates them using the FedAvg algorithm and redistributed a single improved global model. The key argument of this work is that raw sensor readings never leave the drone that produced them.

Two experimental configurations were evaluated, one in which each drone trained on 20.000 sensor record, which is the whole dataset, and one in which training began only after 2.000 collected records while the remaining were still being consumed. Both configurations reached a global accuracy of 100% and achieved perfect recall across all intrusion samples in the evaluation set, meaning no attack went undetected in either run. Additionally, a comparative analysis against a centralized intrusion detection system build on the same Kafka infrastructure demonstrates the privacy advantage of the federated approach.

The results confirms that federated learning is a practically viable and privacy-preserving alternative to centralized intrusion detection for multi-drone networks. The detection quality is at least as strong as what a centralized model can offer, while the architectural properties of federation make the system substantially more resilient and less exposed to data level attacks.

Table of Contents

CHAPTER 1

Introduction

Autonomous aerial vehicles and unmanned aerial systems generate large volumes of data that are essential for tasks such as decision-making and operational efficiency. Ensuring the efficient processing and protection of transmitted data is crucial, as ordinary centralized approaches often require sending raw information to a central server, producing latency, communication overhead and vital privacy risks. To address these challenges, this study presents a framework based on federated learning, wherein aerial unmanned vehicles, such as drones, collaboratively train machine learning models with a central server without the necessity of sharing their raw data. This decentralized approach reduces communication costs, preserves privacy, and enhances security during data exchange. The primary objective of this research is to develop and evaluate a secure data transmission framework utilizing federated learning for unmanned aerial systems. The study seeks to illustrate that federated learning can sustain accuracy while reducing data exposure and network overhead.

1.1: Machine Learning

Machine Learning (ML) is one of the most significant developments in modern computing, allowing systems to improve their performance by learning from data rather than relying solely on explicit programming. Unlike traditional algorithms that follow predefined instructions, machine learning models are built to identify patterns and relationships within data and make predictions based on past experiences. This capability transformed how computers handle complex tasks, enabling them to adapt, generalize, and make informed decisions in dynamic environments. Today, machine learning forms the foundation of many intelligent systems, from speech recognition and computer vision to autonomous vehicles and cybersecurity solutions. This section will give a concise introduction to the different types of machine learning.

The concept of machine learning has evolved over several decades, growing from simple rule-bases systems into one of the most influential areas of modern computer science. Early artificial intelligence research focused on giving machines logical instructions to mimic human reasoning. However, these systems struggled with complexity and uncertainty in real-world data. As computing power and data availability increased, researchers began to design algorithms that could automatically improve trough experience. This shift from programmed logic to data-driven learning marked the beginning of modern machine learning. Over

time, with advances in statistical modeling, neural networks, and big data processing, machine learning has become essential to applications that require adaptability and autonomous decision-making.

The fundamental idea behind machine learning is that by training a model to perform well on a set of tasks that closely mirror the real-world problems it will encounter, the model can then accurately predict outcomes when presented with new data in its intended application.(*What Is Machine Learning? | IBM*, n.d.) The process generally consists of three key phases: training, validation, and testing. In the training phase, the model learns from a dataset that includes examples with known outcomes, adjusting its internal parameters to reduce prediction errors. The validation phase helps fine-tune the model and avoid overfitting, which occurs when the system performs well on familiar data but struggles to generalize to new data. Finally, the testing phase evaluates the model's performance on a separate dataset to determine its effectiveness in real-world scenarios. The quality and amount of data used in each phase are essential, as biased or insufficient data can result in unreliable or inaccurate predictions.

Machine Learning can be categorized into three main types: supervised, unsupervised and reinforcement learning. Supervised is the most common form and relies on labeled datasets, where each input is paired with a known output. The model learns to map inputs to outputs and can then make predictions on new data.

1.1.1: Supervised Learning

Supervised Learning is the groundwork of most practical machine learning systems and is used whenever labeled data is available. In this approach, the model is trained using datasets that include both input features and their corresponding expected outputs, often referred to as labels. The goal of the procedure is to learn the mapping function that best links inputs to outputs so that it can predict results for new, unseen data. Common supervised learning algorithms contain decision trees, support vector machines, random forests, and neural networks. Their methods are widely applied in tasks such as image recognition, fraud detection and predictive maintenance. In autonomous systems, supervised learning enables vehicles to recognize road signs, detect obstacles, or classify terrain types based on labeled sensor information. However, the effectiveness of supervised learning depends heavily on the quality and diversity of the labeled dataset. Collecting and labeling large volumes of precise data can be time-consuming and costly, which is one of the main limitations of this approach.

In supervised learning, the data is labeled, including examples of inputs, known as features, and their corresponding correct outputs, called labels. Algorithms examine a vast dataset of these training pairs to determine the expected output value for new data predictions. For example, if you want to train a model to recognize tree images, you supply a labeled dataset with various tree types and their species names. The algorithm attempts to identify the characteristics that

define each tree based on these labeled outputs. You can then evaluate the model by presenting it with a tree image and asking it to identify the species. If the model's response is incorrect, you can further train it and adjust its parameters using additional examples to enhance its accuracy and reduce errors. Once the model is satisfactorily trained and tested, it can be used to predict unknown data based on the knowledge it has acquired. (*What Is Supervised Learning? | Google Cloud*, n.d.)

Supervised learning in machine learning is usually categorized into classification and regression. Classification algorithms categorize data by forecasting a categorical label or output variable from the input data. This method is applied when the output variables are categorical, indicating two or more classes. A typical example of a classification algorithm is the spam filter in your email inbox. In this case, a supervised learning model is trained to determine if an email is spam using a dataset with labeled examples of both spam and genuine emails. The algorithm analyzes various aspects of each email, such as the sender, subject line, and body text, to identify patterns and assign a score that suggests whether an email is authentic or spam.

Regression algorithms are designed to predict a real or continuous value by identifying a relationship between two or more variables. A common regression task could involve predicting a salary based on work experience. For example, a supervised learning algorithm would receive inputs related to work experience, such as duration, industry, location, etc., along with the corresponding salary. Once the model is trained, it can predict the average salary based on the given work experience. (*What Is Supervised Learning? | Google Cloud*, n.d.)

1.1.2: Unsupervised Learning

Unsupervised Learning represents a more exploratory side of machine learning, where the model works without any predefined labels or outcomes. Unlike supervised learning, which learns by example, unsupervised methods aim to discover the natural structure or organization within data. This approach is particularly valuable when labeled algorithm searches for similarities, correlations, and hidden relationships between data points, often revealing patterns that might not be obvious to human observers. Through this process, unsupervised learning can uncover insights that support decision-making, improve data understanding, or even generate new hypotheses for further study. Two of the most common techniques in this field are clustering and dimensionality reduction.

Imagine you have a dataset containing patient details from a healthcare system, which is complicated and includes both categorical and numerical attributes. You aim to identify patterns and similarities within this dataset. The strategy you could use for this purpose is called Clustering. This is an unsupervised machine learning method that organizes unlabeled examples according to their similarity. In contrast, if the examples are labeled, this type of grouping is referred to as

Classification. Consider a theoretical patient study aimed at evaluating a new treatment protocol. Throughout the study, patients log the frequency and intensity of their symptoms each week. Researchers can utilize clustering analysis to categorize patients with comparable treatment responses into distinct groups. The following figure illustrates one potential arrangement of simulated data into three clusters.

By examining the unlabeled data on the left of the figure bellow, one might suppose that it forms three clusters, even without a formal definition of similarity between data points. In practical applications, however, it is necessary to clearly define a similarity measure, or the metric used to compare samples, based on the dataset's features. When examples have only a few features, visualizing and measuring similarity is straightforward. However, as the number of features increases, combining and comparing features becomes less intuitive and more complex. Different similarity measures may be suitable for various clustering scenarios. Once clustering is complete, each group is given a unique label known as a cluster ID. Clustering is powerful because it can simplify large, complex datasets with numerous features into a single cluster ID. (*What Is Clustering? | Machine Learning | Google for Developers*, n.d.)

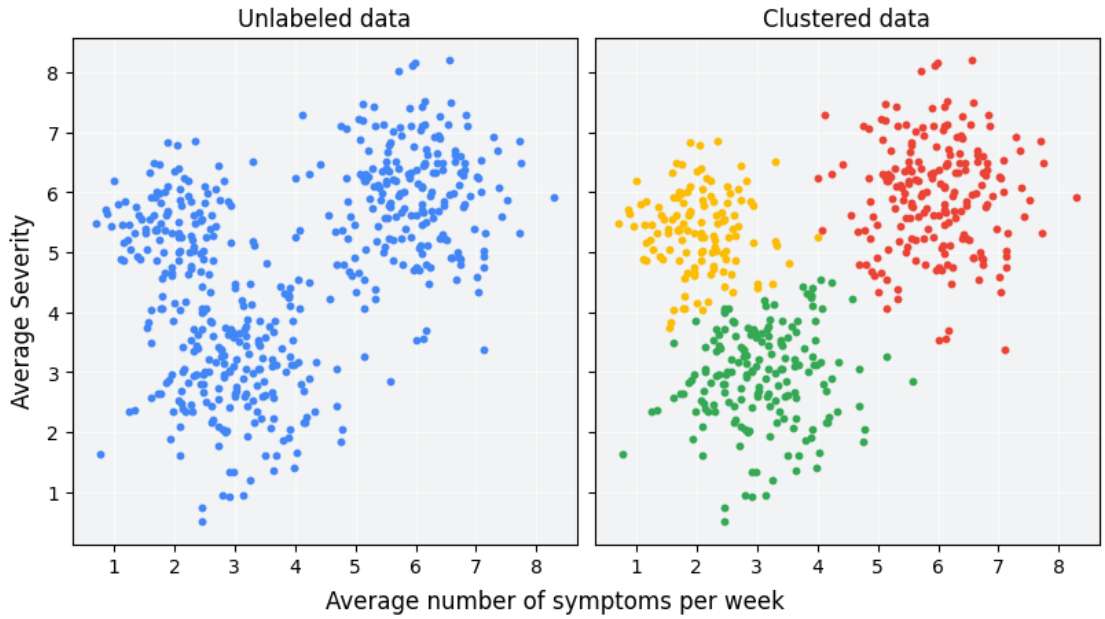

Figure 1: This figure demonstrates the deference between unlabeled and clustered data on visual diagrams (What Is Clustering? | Machine Learning | Google for Developers, n.d.)

1.1.3: Reinforcement Learning

Reinforcement learning, a dynamic field of machine learning, draws inspiration from the way humans and animals acquire knowledge through interaction and feedback. Instead of relying on labeled datasets, a reinforcement learning model learns by taking actions within an environment and receiving feedback in the form

of rewards or penalties. The objective is to discover an optimal strategy – known as a policy – that maximizes the cumulative reward over time.

The system continuously balances exploration by trying new actions and exploitation using known successful actions, to improve its performance. Key components of reinforcement learning include the agent which is the learner or decision-maker, the environment where it acts, actions, states, and rewards. Classic algorithms such as Q-Learning and Deep Q-Networks (DQN) have demonstrated remarkable success in complex tasks, from playing games to managing robotic control.

Reinforcement Learning (RL) is about figuring out how to make choices. Think of an agent, which might be a software program, a robot, or even an unmanned vehicle in our study, moving around in an agricultural aria. This setting could be a real place, a virtual game world, or even a market. The agent makes moves in this location, and these moves can result in different outcomes, some better than others. The agent's aim is to collect as many rewards as possible over time. It does this by learning a strategy, which is basically a plan that guides it on what move to make in any situation. This plan gets better through many rounds of interacting with the setting. For example, think of a chess-playing AI. The agent's moves are the steps it takes on the chessboard. The setting is the current state of the game, and the reward is winning the game. By playing repeatedly and getting feedback on its moves, the RL agent learns which moves are more likely to lead to winning. (*What Is Reinforcement Learning (RL)? | Google Cloud*, n.d.)

In autonomous aerial systems, reinforcement learning is particularly valuable because these environments are often uncertain, dynamic, and require real-time decision-making. A drone, for instance, can use reinforcement learning to optimize flight paths, avoid obstacles, or adjust its behavior in response to changing wind conditions without needing explicit programming for every possible scenario. Over time, the system becomes more capable and adaptive, learning from both successes and mistakes. However, reinforcement learning also presents challenges: it can require large amounts of trial-and-error interaction, which is costly and sometimes unsafe in physical systems. As a result, reinforcement learning in autonomous vehicles is often combined with simulation environments or hybrid learning methods to train models safely and efficiently. Reinforcement learning can be broadly categorized into two main types: model-based and model-free.

In model-based reinforcement learning, the agent tries to create a mental picture of its surroundings. This mental picture helps the agent guess what will happen if it takes certain actions, allowing for a more thoughtful and planned way of acting. Think of a robot trying to find its way through a maze. A model-based RL agent would work on forming a mental map of the maze's design. It would then use this map to plan a route, trying out different actions and predicting their results before actually moving.

Model-free reinforcement learning, however, doesn't depend on taking a clear mental picture of the environment. Instead, it focuses on learning the best way to

act by linking actions with values based on the rewards it gets. Going back to the maze example, a model-free agent wouldn't try to map out the whole maze. Instead, it would learn which actions, like turning left or right at certain points, are more likely to lead to the exit based only on its past experiences and the rewards it has received. (Terven, 2025)

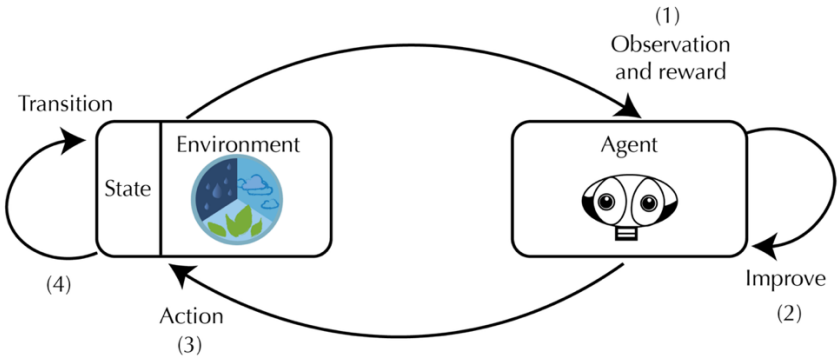

Figure 2: Showing how reinforcement learning works between the environment and the agent. This figure is inspired by [Morales, M. Grokking Deep Reinforcement Learning; Manning Publications: Shelter Island, NY, USA, 2020. Google Scholar]

In summary, machine learning provides the foundation for systems capable of learning from experience and improving their performance over time without explicit programming. The three main learning paradigms each contribute distinct methods for analyzing and understanding data. In the context of unmanned aerial vehicles, these approaches enable drones and vehicles to perceive their surroundings, make informed decisions, and adapt to changing conditions. Building on these principles, the next section explored federated learning, a decentralized approach designed to enhance data privacy and efficiency when multiple devices collaborate to train machine learning models.

1.2: Federated Learning

Federated Learning is a modern approach in machine learning that enables multiple devices to cooperate on training a mutual model without exchanging their raw data. Instead of transferring entire datasets to a central server, each device processes its own local information and sends only model updates, such as learned parameters or gradients, to a central coordinator. The server aggregates these updates, often by averaging them, to refine a global model, which is then sent back to the devices for further training. This cycle continues until the model achieves a stable level of accuracy. By keeping the data decentralized, Federated Learning offers an approach to preserve confidentiality, reduce communication overhead, and limit dependence on centralized storage. It is particularly valuable in distributed environments where large amounts of data are generated endlessly,

such as smartphones, IoT devices, and autonomous systems, and were sharing sensitive information directly would raise ethical or security concerns.

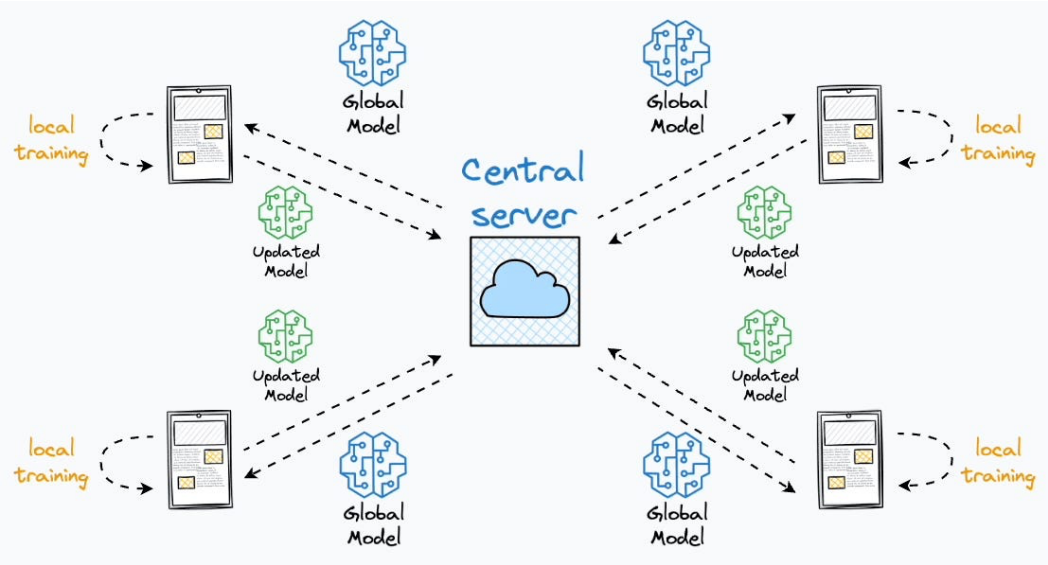

Figure 3: How a federated learning central server works with its client, showing the data transmission between them (Introduction to Federated Learning - by Avi Chawla, n.d.)

1.2.1: Motivation and Background

Federated Learning emerged as a response to the growing worries over data privacy and the limitations of traditional centralized machine learning approaches. In 2016, researchers at "Google LLC" introduced the concept to enable collaborative model training across multiple devices without the need to share raw data. This innovation aimed to address the challenges posed by data silos and the risks associated with centralized data storage. By allowing devices to train models locally and share only model updates, federated learning preserves user privacy and reduces the need for extensive data transmission. The approach has since gained traction in various domains, including mobile applications, healthcare, and autonomous systems, as it aligns with strict data protection regulations and the increasing demand for privacy-preserving techniques. In detail, the basic idea of federated learning appeared as a response to the practical limitations of traditional centralized machine learning, especially in large distributed systems. In many typical approaches, data that is produced in a single or multiple devices, is collected in a particular central location where the final training process takes place.

The technique mentioned, can have significant advantages especially in situations where system security and data distribution are vital considerations. By constraining the data to be localized on devices that are functioning as clients, this technique can reduce the risk of sensitive information exposure during data transmition or storage. This approach naturally strengthens confidentiality and

enables organizations to meet strict data protection requirements. (*Federated Learning: What It Is and How It Works | Google Cloud*, n.d.) As data increasingly contain sensitive information, transferring it outside the environment in which it was produced raises ethical and regulatory challenges. In many cases, legal frameworks and organizational policies restrict how data can be stored and shared. Federated Learning was proposed as a way to address these concerns by allowing learning to take place without direct access to the underlying data. By keeping data on local devices and sharing only limited training information, the overall learning process becomes more aligned with modern privacy requirements.

1.2.2: Advantages for Use in Autonomous Aerial Vehicles

This work leverages federated learning in order to address key challenges encountered in distributed autonomous aerial systems. One of the main advantages is the reduction of data transmission requirements. These autonomous systems continuously generate valuable information that is being sent to a central server which can be inefficient and unreliable, especially under limited or unstable network conditions. By allowing each unit to train locally and share only essential training updates, this technique reduces communication overhead and supports more efficient system operation.

Another key advantage lies in the protection of valuable information. Data collected from units such as LiDAR Sensors, GPS modules, radar systems and Inertial Measurement Units (IMUs) that include operational or environmental details, should remain confined to the local system and limit direct data exposure by keeping the raw data on each device throughout the training process. This approach lowers the risk associated with centralized information storage and aligns well with modern privacy and security requirements.

An additional advantage of the technique mentioned is its ability to support scalable and resilient learning in distributed aerial environments. As the number of autonomous units increases, centralized approaches often struggle to manage the associated computational and coordination needs. Federated learning allows each component to contribute independently to the training process, allowing the system to scale without needing extensive centralized resources. In addition, since local training does not depend on constant connectivity, temporary communication disturbances do not prevent participation altogether. This characteristic makes federated learning suitable for use in autonomous aerial systems as they operate under dynamic network conditions.

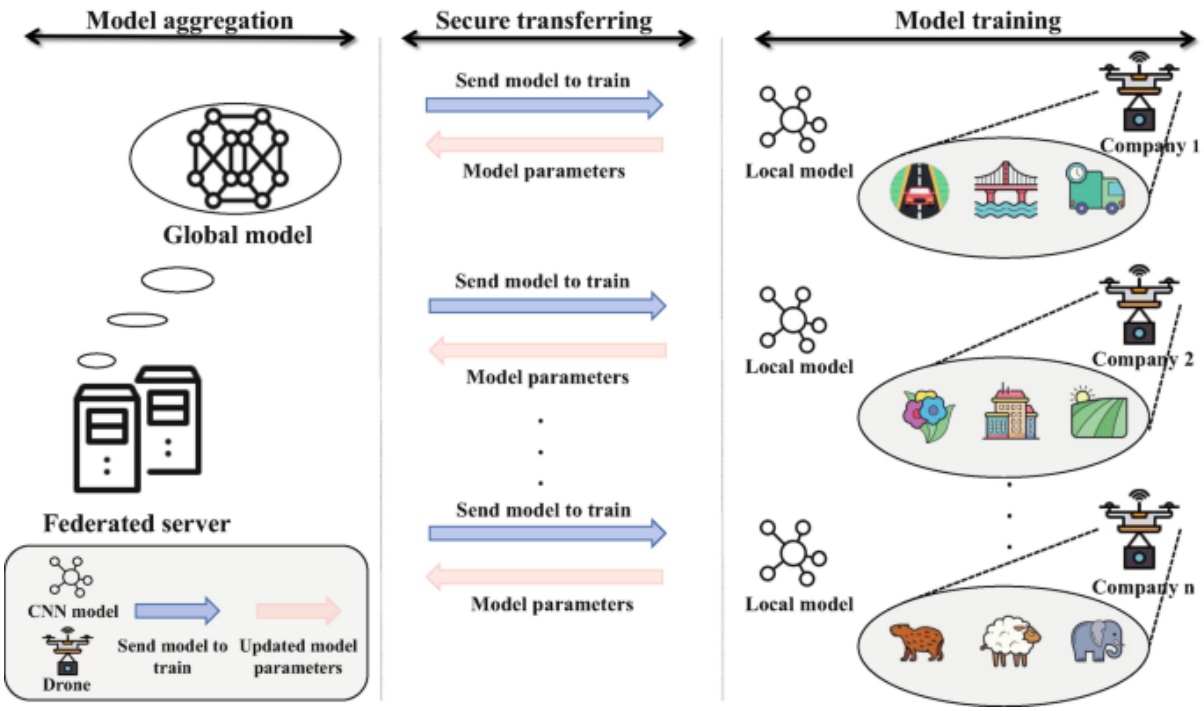

Figure 4: Illustrating a typical federated learning workflow applied to autonomous aerial units, where local model training is performed independently and aggregated through a central coordinator to form a global model. Having different units each one with their own information, are implementing local training. After the successful training each entity sends their model parameters on the federated server and then to the global model. The model itself and the model parameters are then sent back and forth repeatedly from the server to the drone and vice versa until the optimal process is completed. (P. Li et al., 2024)

1.2.3: The Importance of Enhanced Security Measures in UAVs

As unmanned aerial vehicles become increasingly autonomous and interconnected with various devices, the appropriate protection of those systems becomes a central requirement rather than an optional development. These devices are working on open environments having as an example a wheat field that a drone needs to consume valuable information from the soil for further examination, relying heavily on wireless communication and frequently performing tasks that are sensitive or mission-critical that any interruption or data loss could loss major complications. Any weakness in the communication process or the learning framework may not only disturb data reliability but also clutter with real-time operational behavior. Therefore, having a strong security mechanism in device-based architecture is essential for ensuring trustworthy system performance on both data and unit control.

Unlike conventional ground-based infrastructures, UAVs are working on dynamic network conditions and potential outer intrusions. The are transmitting data valuable on the device's navigation system as well as positioning information, environmental measurements, and in certain occasions, visual or strategic

content. If any of the information above is interrupted on any means such as theft or signal loss, the cost of this may extend beyond data loss and directly affect fling performance, coordination between units, or decision-making accuracy. In scattered learning settings, cooperated model updates could progressively influence the global model. Affecting the behavior of all participating units.

Furthermore, the growing use of unmanned aerial vehicles in arias such as environmental monitoring, infrastructure review, defense procedures and emergency response increases the importance of reliability. On applications like the ones mentioned, consistent performance is essential because even the slightest disruption can cause significant consequences. Thus, designing a framework that ensures emphasizes on secure data handling, seamless communication and robust learning procedures is obligatory. In the context of this study, Federated Learning is explored as a technique that will not only have efficient collaboration, but also as a step toward strengthening the overall security posture of distributed aerial systems.

CHAPTER 2

Literature Review

This chapter presents a review of the existing literature related to federated learning and its application in distributed and autonomous systems. The intention of this review is to examine prior research efforts, identify common approaches, and highlight the challenges that remain unresolved in current methodologies. Particular emphasis is placed on studies involving decentralized machine learning, privacy-preserving training techniques, and autonomous aerial platforms, as these areas are directly related to the scope of this thesis. By analyzing relevant works within these domains, this chapter establishes the theoretical and technological foundation upon which the proposed research is built.

2.1: Evolution from Centralized to Decentralized Machine Learning

As intelligent systems continue to evolve and become embedded in a wide range of real-world applications, the approach that machine learning models are trained has gradually shifted away from strictly centralized settings. In many modern-day scenarios, data is no longer produced within a single controlled environment but is instead generated continuously throughout numerous independent devices that operate in parallel and often under varying conditions. This revolution has led to the development of distributed machine learning approaches, in which the training process is shared among several computing units that may differ in computational capability, network availability, and operational constraints. The primary objective of these approaches is to enable collective learning from geographically and logically scattered data sources while maintaining acceptable performance and system efficiency.

Centralized machine learning remains a widely accepted approach due to its simplicity and ease of implementation, however, its limitations become increasingly evident as systems scale in size and complexity. The requirement to transfer large volumes of raw data to a central processing unit can place significant strain on communication infrastructure, particularly in environments where bandwidth is limited or connectivity is unstable. In addition to communication overhead, centralized data collection introduces delays that may obstruct timely decision-making in systems that rely on real-time or near-real-time responses. Furthermore, gathering data in a single location increases

exposure to security risks and creates a dependency on centralized resources, which may not always be reliable or continuously available.

In response to these challenges, decentralized learning strategies have been introduced with the aim of distributing computation closer to the data sources themselves. Under this paradigm, individual nodes participate in the learning process by performing local computations while contributing selectively to a shared objective. Such an approach allows systems to reduce reliance on continuous data transmission, improve resilience to partial network failures, and better accommodate heterogeneous devices operating under different constraints. Decentralized learning methods therefore provide a flexible foundation for large-scale intelligent systems, particularly in dynamic environments where data generation is continuous and system conditions cannot be fully controlled. Federated learning represents a structured realization of this paradigm, offering a coordinated mechanism for decentralized training that aligns well with privacy-aware and communication-efficient system design.

2.1.1: Limitation and Challenges for Multi-Node Learning Systems

As intelligent systems continue evolving, the shift from centralized to distributed learning has become gradually relevant. While centralized data offers more simplicity and power over the training process, it struggles to adapt scale, diversity and data generation rate in more modern applications. While on the other hand, decentralized approaches, distribute computation closer to the birthplace of the data, allowing individual devices or nodes to contribute on the model development without transferring all the raw information on a single place. Nevertheless, as well as other training techniques like centralized, the proposed method has its own set of challenges that needs to be faced.

One of the most crucial considerations in decentralized systems is heterogeneity. Every device has its own setting that makes them special and defer from each other on computational power, memory volume and energy availability to complete a task, which can directly affect how each participant contributes to the overall learning process. Furthermore, these varying devices can generate data with different structures or distributions, creating potential inconsistencies that complicate the aggregation of local updates into the global model. Ensuring that the system can resolve these differences while maintaining model accuracy involves careful synchronization and adaptive aggregation strategies to guarantee the best possible final model.

Connectivity and communication restrictions could also play a crucial role. Unlike centralized systems, decentralized learning relies heavily on periodic exchange of model updates between participants and directors. In real-world environments network bandwidth may be limited, latency can vary and connections can be sporadically lost. These conditions can cause complications such as time-consuming aggregation of models, decrease the precision of learning updates and

in some cases even result in inconsistent model parameters if devices fail to broadcast updates on schedule. Therefore, a robust system is necessary for preserving reliable performance in distributed learning.

Additional challenges lie in the confidentiality and security of the data that is valuable to the unit. Whilst raw data are not being transmitted due to decentralized learning, the need for proper handling is obligatory because it can resolve on data leakage and reveal sensitive information about local datasets. Techniques such as secure aggregation, differential privacy or encrypted parameter sharing are often employed to mitigate those risks, in contrary the techniques mentioned are giving additional complexity and computational overhead. Balancing privacy protection with model efficiency remains a key design consideration, especially on environments where regulatory compliance and user dependance are critical.

The previews challenges become more noticeable on especially on autonomous aerial units. This is happening because UAVs are highly demanding and are working on a highly dynamic and unpredictable environments where connectivity can be intermittent, as for instance a sudden wind gust whilst on flight, the unpredictable change on the drones transmitted signal due to obstacles such as trees or the exceeding drone distance, resulting in a weaker signal. UAVs have limited energy resources as the battery that powers the unit has its capacity limits, and on-board processing power is constrained. Each unit is equipped with a continuous stream of sensor data, including visual imagery for its handling, LiDAR scans, GPS signals for its position on the map and inertial measurements, all of which vary in type, scale and quality. Integrating this heterogeneous data into a shared learning model requires mechanisms that can adapt to diverse local conditions while preserving the accuracy and consistency of the global model. Security concerns in UAV networks are particularly demanding. The interception, manipulation or corruption of transmitted updated can compromise both model reliability and operational safety. Additionally, the flexibility that benefits the architecture of UAVs is that they may temporarily leavy the network or rejoin under varying conditions. Therefore, Federated Learning offers an assuring framework in this context, that supports local training on UAVs while sharing only essential model information and updates, towards reducing exposure to external threats and enhancing system toughness.

Overall, using decentralized learning in Unmanned Aerial Vehicles represents a delicate balance between efficiency, reliability, and security. By understanding these challenges in both general distributed systems and UAV-specific scenarios, researchers and engineers can design federated learning structures that are robust, scalable and capable of supporting autonomous aerial operations under real-world constraints. This understanding lays the foundation for the subsequent section of this thesis, which explore concrete methodologies and solutions tailored to UAV networks.

2.2: Federated Learning Approaches in Distributed Environments

Unmanned aerial vehicles continue to evolve in order to operate into complex operational environments, for this reason, their security and reliability have become a critical research concern. Modern UAV systems are no longer isolated platforms performing simple tasks; instead, they often operate as part of larger networks that rely on continuous communication, sensor data. Exchange and coordinated decision-making. These capabilities enable advances application s such as environmental monitoring, disaster response, infrastructure inspection and autonomous navigation. Nevertheless, the connectivity that enhances UAV functionality, introduces potential vulnerabilities likewise, as communication links and onboard systems may become targets for malicious interference or unauthorized access.

As a response to the challenges mentioned, a growing part of research has focused on the development of techniques that are designed to detect, prevent and mitigate cyber threats in UAV environments. Among these approaches, machine learning has emerged as a large-scale tool due to its ability to work with large volumes of operational and network data in order to identify abnormal patterns and behavior. Structures similar to federated learning, such as other intrusion and anomaly detection systems as well as distributed learning strategies have all been explored as potential solutions for improving UAV security and reliance. By reviewing the existing literature in the specified field, we can gain insights into the current advancements in UAV system protection, as well as the ongoing limitations and challenges that drive further research.

2.2.1: Architecture and Distributed Training

Researchers have progressively focused on collaborative learning frameworks that allow multiple devices to train machine learning models without transferring their local datasets to a central location. One comprehensive study by J. Liu et al. (Liu et al., 2022) examines the evolution from traditional distributed machine learning approaches toward federated learning systems. The authors explain in their work that federated learning supports several participants to contribute to a shared model while keeping their training data locally stored, which helps address concerns related to privacy and data ownership. In this approach, the model training happens on each unit locally for secure training with its own data, then they periodically send updated model parameters to a central server where they are aggregated to develop the global model. The study also highlights several technical aspects that influence the effectiveness of federated learning systems. These include communication efficiency between participating devices and the server, the capability of the system to handle devices with altered computational capabilities and the challenge of maintaining model performance when data distributions differ among participants. Another aspect of this study is the analysis of how federated learning can be applied across various domains where

sensitive data cannot be easily centralized. The authors examination delivers a broad overview of the design principles and challenges associated with federated learning, offering beneficial insights for research areas that rely on distributed and privacy-aware machine learning environments.

Another study by Saba F. Lameh, Wade Noble et al. (Lameh et al., 2020) investigates how federated learning can be effectively applied to a network of edge devises, each with its own computational and storage boundaries. The research emphasizes how allocating the training process across multiple nodes can reduce the need for sending raw data to a central server, which not only saves bandwidth but also sustains data secrecy. The paper also discusses strategies to balance the workload among heterogeneous devices, ensuring that units with lower capabilities are not overwhelmed while still contributing significantly to the overall model. A key focus of the work is on the challenges that arise when devices operate under different network conditions. Since the network between devices and the central coordinator can be inconsistent, the study examines methods to manage asynchronous updated and reduce delays in aggregating model parameters. It emphasizes the importance of a flexible architecture that can adapt to varying connectivity, warranting that the global model continues to improve even if some devices temporarily drop out of connection or experience a slower performance. By keeping the data localized and sharing only model updates and parameters, federated learning minimizes exposure of private information and thus this arranges privacy and security answer in distributed learning environments. The study also points out the need for secure aggregation techniques to prevent leakage across transmitted updates. These insights offer practical guidance for using federated learning in UAV networks, where devices work independently, produce varied data, and need to stay both efficient and safe in shifting and potentially risky conditions.

The studies discussed are using architectures implemented with federated learning that enable collaborative model training while sustaining data decentralization and privacy. By allowing devices to perform local training and interchange only model updates, these systems reduce the need for centralized data collection and improve scalability in distributed environments. At the same time, the literature highlights several practical challenges, including devices heterogeneity, communication boundaries and the need for effective aggregation mechanisms. These factors are especially relevant for UAV networks, where individual units may operate under diverse computational and connectivity conditions. These influences are particularly relevant for UAV networks, where individual units may operate under different computational and connectivity conditions. While the federated learning technique offers a promising framework for distributed intelligence, additional considerations such as system reliability and security must also be addressed. The following subsection will examine the confidentiality and the security challenges that are associated with federated learning systems.

2.2.1: Threats and Vulnerabilities in Federated Learning Systems

As federated learning offers significant advantages for distributed environments, the collaborative nature of this approach also introduces several security concerns. Since model training is performed across multiple independent participants, the overall system depends on the integrity of each contributing node and the reliability of the communication process. In practical deployments, especially in large-scale networks, some devices may behave unpredictably or even maliciously. These conditions create opportunities for adversaries to interfere with the training process, compromise model performance, or attempt to obtain sensitive information from participating units. As a result, understanding potential risks associated with federated learning systems has become an important research topic.

A comprehensive review by S. Almutairi et al. (Almutairi & Barnawi, 2023) examines the security risks associated with federated learning systems and analyzes how these distributed frameworks can become vulnerable to different types of attacks. Even though the federated learning technique is designed to protect user data by keeping it on local devices, the collaborative training process introduces new security concerns. Since model updates are exchanged between multiple participants and a coordinating server, malicious nodes may attempt to manipulate these updates in order to influence the global model or disrupt the training process. In the work mentioned (Almutairi & Barnawi, 2023) the authors classify several categories of threats that can affect the federated learning technique. Poisoning attacks are one of the discussed concerns where compromised participants intentionally send manipulated model parameters to degrade the accuracy of the shared model. Another risk comes from inference-based attacks, where an adversary attempts to extract sensitive information about the training data by analyzing the exchanged model updates. These attacks demonstrate that even when raw data are not shared, private information may still be indirectly exposed through the learning process. In the case oof poisoning attacks, malicious participants aim to influence the training procedure by submitting altered model updates during the aggregation phase. Because federated learning requires contribution from varying distributed devices, the system often assumes that these participants behave honorably. However, if a targeted attack becomes successful and the invader gain control over one or more nodes, this can gradually affect the global model, reducing its accuracy or causing it to behave incorrectly in certain situations. Such interference can become a great concern in application where reliable decision-making is critical. Inference attacks represent another important category of threats within federated learning environments. The main idea of these attacks is not on altering the training process, but the objective is to obtain information about the data that is being used by other participants. By carefully analyzing the model parameters which are shared across the device and the server during the training rounds, an opposition may attempt to infer characteristics of the underlying datasets, such as specific patterns or sensitive attributes for its own benefit. This prospect shows that even though federated learning limits direct data sharing, additional protection

mechanism are still a necessity to ensure that confidential information remains secure.

Although poisoning and inference attacks represent significant threats in federated learning, researchers have identified several additional risks that also arise from the distributed nature of the training process. Because this framework relies on the collaboration of many independent units that communicate with a coordinating server, attackers may attempt to exploit weaknesses in participation, communication or model aggregation. Understanding these potential attack strategies is important for ensuring that federated learning systems remain reliable and trustworthy when deployed in real-world applications.

One possible threat that involves malicious entities attempting to break into the system is by presenting themselves as legitimate participants (Zakaria & Khalid, n.d.). In such situations, an attacker may introduce a fraudulent device into the training network and begin contributing updates to the final model. If this node becomes accepted by the system, it can influence the collaborative training process by sending misleading or carefully crafted parameters based on its advantages. Over time, these manipulated contributions may affect the overall behavior of the global model, potentially reduce its accuracy or alter the way it responds to specific inputs. Another concern relates to attacks that interfere with the communicational process between participating devices and the coordinating server (Zakaria & Khalid, n.d.). Federated learning depends heavily on regular exchanges of model updates during training rounds, and any disruption in this communication can negatively affect system performance. Delay transmission may be caused by attackers or interruptions to the exchange of updates. Such disruptions can slow down the training procedure because they are time costly and create instability to the overall training resulting in an unstable global model. These attacks often appear in environments where network conditions are already limited. A target for adversarial actions could also became the aggregation phase. During this stage, the central server combines the updates received from multiple participants in order to construct a new version of the shared model. Federated learning may face a variety of threats that can affect different stages of the collaborative training process. For a more detailed understanding of these risks, the table below outlines some of the primary types of attacks recognized in federated learning settings.

Attack Type	Description	Attack Surface	Timing
Membership Inference	Determines if a specific data point was in the training set	Client	Post-training
Property Inference	Infers statistical properties of client data	Client	Post-training
Model Poisoning	Injects malicious updates to corrupt the global model	Client	During training
Backdoor Attacks	Embeds hidden misbehaviour triggered by specific inputs	Client	During training
Gradient Leakage	Reconstructs raw data from shared gradient updates	Server	During training
Sybil Attacks	Uses fake identities to influence model updates disproportionately	Client	During training
Free-Rider Attacks	Participates in FL without contributing useful updates	Client	During training
Collusion Attacks	Multiple adversaries collaborate to subvert learning	Multiple	During training
Adaptive Multi-Vector	Dynamic use of multiple attack vectors to evade defences	Multiple	Dynamic

Figure 5: Listing all attack types with each attack surface and timing (Zakaria & Khalid, n.d.)

Federated learning presents several benefits for environments such as distributed machine learning, but the collaborative nature of the training process also shaped new security concerns that needs to be resolved. As discussed in the literature, attacks do not target only one aspect on the system they hit many parts of the unit or the process including the integrity of model updates, the privacy of local data and the communication among participating devices. Vulnerabilities like these demonstrate that even though federated learning switches form centralized to decentralized data collection, additional protection mechanisms are required to ensure a secure operation. Addressing these challenges is particularly important in environments where distributed devices must operate reliably and securely, such as networks of autonomous aerial vehicles.

2.2.2: Approaches for Intrusion Detection Systems

Intrusion Detection Systems (IDS) have an important role in protection computer networks and distributed systems from malicious activities. As more networks become interconnected and complex, traditional rule-based security mechanisms are not enough for the protection needed in order to distinguish new or evolving cyber threats. Therefore, many machine learning techniques have been explored for improving the ability of an IDS to identify suspicious activity. The main advantage is that machine learning models are capable of analyzing large volumes of network data and recognize patterns that may indicate abnormal or potentially harmful activity. Be learning from historical data, these systems can alter and adapt to changing network conditions and detect attacks that may not follow previously known signatures.

One example of this research direction can be seen in a work that investigates the use of machine learning techniques for detecting cyber threats on an Internet of Things environment (Kantharaju et al., 2024). Networks composed of multiple connected devices generate large amounts of communication data, in this scenario, identifying malicious activity becomes particularly challenging when using conventional security mechanisms. The authors of this work (Kantharaju et al.,

2024) explain how learning-based models can analyze patterns within network traffic in order to distinguish normal system behavior from activities that may indicate a potential intrusion. The main idea of these systems is to reduce the manual effort needed for monitoring the federated learning process, while improving the speed and accuracy of threat detection. By combining multiple processing stages, these frameworks can focus on the most relevant information and adapt to changing network conditions. Machine learning-based detection methods are exceptionally effective at recognizing both known threats and novel, previously unseen behaviors. These systems learn from traffic patterns and operational behavior, rather than relying solely on predefined signatures, which allows for a more flexible and dynamic protection. In environments where devices are heterogeneous this is very essential while they continuously generate data and furthermore demand rapid responses for detecting suspicious activity which can prevent broader system compromise. The use of these intelligent models also helps balancing the computational efficiency with performance based on fraud detection, ensuring that security measures do not overwhelm the system's resources. In distributed and autonomous systems, such as networks of UAVs, the techniques mentioned take on an even greater importance. These networks operate in dynamic environments with limited onboard resources and intermittent connectivity, making real-time detection challenging. Incorporating machine learning-based intrusion detection into these systems allows individual units to analyze local data while sharing only the essential insights with the central coordinator or with other participating units. By using this approach, network security gains far-reaching improvement while also preserving data discretion, as sensitive information does not need to leave each unit. Ultimately, combining distributed learning and intelligent detection creates a more resilient and adaptive security framework suitable for modern autonomous systems (Kantharaju et al., 2024).

Buyuktanir et al. (Buyuktanir et al., 2025) offer a broad survey of how federated learning has been adapted for intrusion detection systems across distributed environments. The core argument of their work is that classical IDS architectures, despite strong classification performance, introduce privacy risks when they rely on centralizing raw network traffic. By keeping data local and only share model updates with a global aggregation server, FL-based IDS frameworks allow participating nodes to collaboratively build detection models without surrendering control over their own data. This iterative process of local training followed by severe-side aggregation makes federated learning particularly compatible to sectors where regulatory and confidentiality make direct data sharing impractical. The study analyzes many federated learning techniques including FedAvg, FedProx, FedOpt and FedAdam across established cybersecurity datasets such as UNSW-NB15 and Edge-IIoTset. Among the reviewed systems, a federated multi class IDS designed for software-defined networking environments achieved approximately 98.6% detection accuracy on the Edge-IIoTset dataset. Another approach called FEDGAN-IDS, combined generative adversarial networks with a federated learning scheme, reaching upward of 99% accuracy in binary classification tasks on the UNSW-NB15, KDD99 along with other datasets. Furthermore, the inclusion of GAN is especially relevant in distributed settings,

as it helps address the scarcity and class imbalance of labeled attack samples that individual nodes commonly experience. Beyond these specific systems, the authors examine a range of architectural choices that influence how well FL-based IDS solutions perform in practice. The most common configuration for network-level detection is horizontal federated learning, where nodes share the same feature spaces bit hold different traffic samples.

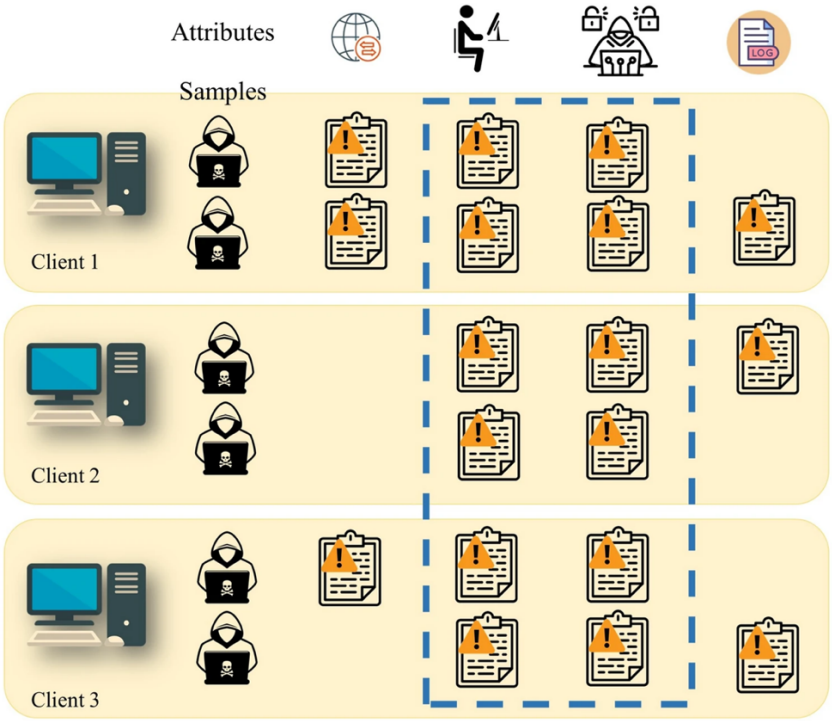

Figure 6: The horizontal FL technique showing the deviation of data (Buyuktanir et al., 2025).

A biologically inspired direction is also reviewed, in which an artificial immune system-based IDS was deployed within a privacy-preserving edge intelligence framework, demonstrating the gained performance over competing models when evaluated on the KDD-99 dataset. These varied approaches reflect the breadth of design strategies explored as the field matures.

Several challenges that continue to limit real-world deployments have also taking a great part on this work (Buyuktanir et al., 2025). One of the most frequent obstacles in federated learning is the distribution of non-IID data across clients. As statistical divergence between local datasets can hinder global model governance. Communication overhead from repeated model exchanges and adversarial threats such as model poisoning are similarly flagged as open problems. In order to reinforce privacy beyond what the federated architecture alone provides, techniques such as differential privacy and homomorphic encryption are discussed as practical and complementary safeguards.

A more technically focused contribution to this area comes from Li et al. (J. Li et al., 2023), who approached the federated IDS problem from a different angle, driven by the practical limitations that individual organizations face when trying

to build reliable detection models on their own. The main observation on their study is straightforward but important, a singly enterprise collects occasionally high-quality labeled attack traffic and uses that data to train a model which is capable of handling the full spectrum of modern threats. While pooling data across organizations would solve this, but it raises exactly the confidentiality concerns that make doing so infeasible in practice. To keep this tension in control, the authors developed a system called "Dynamix Weighted Aggregation Federated Learning" also referred to as "DAFL", which builds on the standard federated paradigm but introduces two mechanisms designed to make the aggregation step considerably more intelligent than conventional approaches. Before the aggregation process takes place, the first mechanism is a dynamic filtering stage that runs on the side of the server. Rather than treating all incoming local model updates as equally valid contributions, the system actively evaluates the quality of each update and excludes these that fall below an acceptable threshold. This is more important that it might initially appear, in real federated deployments, some participants will inevitably have noisy, imbalanced or otherwise unreliable local data, and blindly incorporating their updates into the global model which can drag down the overall model performance. The DAFL technique reduces the influence of weak participants without excluding them from the federation entirely by filtering the contributors at the aggregation stage. The second mechanism assigns weights to each accepted update proportional to its assessed contribution quality, meaning that the nodes with more informative local data have a proportionally greater influence on the resulting global model. These two steps together will replace the equal weighting assumption that inspires the most standard aggregation strategies, including FedAvg, with something considerably better suited to the uneven data conditions found in real-world deployments. Beyond the detection performance gains, one of the more practically significant outcomes reported by Li et al. (J. Li et al., 2023) concerns communication efficiency. The federated learning technique involves repeated rounds of model interaction within clients and the main global server, and in large scale or bandwidth limited environments, this overhead can become a great obstacle to deployment. The filtering that the DAFL technique uses as well as the weighting approach, was shown to reduce the volume of transmitted data between participants and the server by 33% to 71% by concentrating model updates toward higher quality contributors and discarding lower quality ones early in each round, depending on the experimental configuration. This is a non-trivial reduction not only in terms of network load, but in terms of the time cost per round, which directly affects how quickly the global model converges to a usable state. This kind of efficient gain carries real operational weight especially in security critical deployments where rapid adaptation to new attack patterns matters.

DAFL is particularly relevant to the broader federated IDS literature as it demonstrates these improvements without compromising the fundamental privacy guarantees that motivate using federated learning in the first place. Decisions such as filtering and weighting, are made on the server using only the model update vectors instead of the raw traffic data that produced them, so the local confidentiality of each participant's records is maintained. The work mentioned (J. Li et al., 2023) speaks to a challenge that several other contributors

in this space have left unaddressed: How to preserve both privacy benefits and the practical efficiency of federated training while simultaneously improving robustness against the degrading effects of heterogeneous data quality across clients.

Across the works reviewed in the above section, federated learning consistently proves to be more than a simple privacy preserving substitute for centralized IDS. It represents a fundamentally different way of building, updating and maintaining detection models in environments where data cannot or should not leave its source. The contributions in the works mentioned, address this from different angles, ranging from broad algorithmic benchmarking to targeted improvements in aggregation quality and communication efficiency. However, the environments considered in most of these works, including enterprise networks, IoT deployments, and software defined infrastructures, still operate under relatively stable conditions compared to other distributed settings. When the participating nodes are mobile, intermittently connected, and severely constrained in terms of onboard computing resources, the challenges surrounding federated intrusion detection can become considerably more complex. This is precisely the reality faced in networks of unmanned aerial vehicles, where security and adaptability must coexist under strict operational limitations.

2.2.3: Federated Learning for UAV Network Security

The deployment of unmanned aerial vehicles in coordinated multi-unit formation raises a distinct set of security concerns that differ meaningfully from those encountered in static or ground based distributed systems. Unmanned Aerial Vehicles collect continuous streams of imagery, sensor readings, and positional data though their abundance of instruments and cameras. These structures carry privacy implications, and the collaborative nature of multi-drone operation means that this information must be shared in some form to support coordinated decision-making. A work by Gupta S. et al. (Gupta et al., 2025) addresses this tension directly, disputing that conventional machine learning approaches to UAV security are fundamentally limited by their reliance on centralizing raw data, and that federated learning offers a structurally superior alternative for environments where bot security and data confidentiality need to be maintained simultaneously. The framework that this study proposes, integrates federated learning with convolutional neural networks, a combination chosen deliberately to balance detection capability with the computational constraints typical of drone platforms. Convolutional Neural Networks are well suited to pattern recognition tasks within complex multi-dimensional data, and their relatively efficient inference characteristics make them a more practical choice for onboard processing compared to deeper or more parameter heavy architectures. In the proposed structure, each UAV trains a local model using its own collected data independently from each other unit, and only the resulting model parameters are transmitted to a central aggregation point. This means that the underlying raw data collected by each unit, whether environmental sensor readings or operational

logs, never leaves the individual drone, which directly highlights one of the core vulnerabilities in centralized learning architectures.

On the other hand, the authors also recognize that transmitting model parameters rather than raw data does not entirely eliminate privacy risks. Gradient and weight information can under certain conditions be reverse engineered to expose characteristics of the training data that generated them. In order to encounter this problem, the suggested framework incorporates a secure aggregation mechanism that encrypts the local model updates before they are sent upstream. The combination of these encrypted contributions takes place in the server without ever accessing their individual content, which preserves the confidentiality of each drone's local data even during the aggregation step. This layered approach, that associates the architectural privacy of federated training with cryptographic protection at the communication stage, represents a more thorough treatment of the privacy problem than most comparable system offer. The simulation results after training show measurable gains across a range of evaluation metrics, including accuracy, precision, recall and F1-score, when comparing the proposed federated framework against non-federated baselines. Beyond raw detection performance, the work highlights that the synchronous federated approach used in their design avoids data loss during training rounds, a practical advantage over asynchronous alternatives that trade speed for completeness. For multi-UAV systems that operate in environments where coordination failures or security breaches can have immediate operational consequences, this kind of reliability during the learning process matters as much as the final model quality. The contribution therefore is not simply a privacy mechanism grafted onto an existing detection pipeline, but a design choice that shapes the reliability, security and operational suitability of the entire system (Gupta et al., 2025).

A dedicated contribution to federated anomaly detection in UAV networks is presented by Bathi et al. (Bithi et al., 2026), who explore the security challenges unique to drone communication systems that have not been adequately addressed by existing federated learning frameworks. Their standing point is that UAVs are entering more and more into the fields of agriculture, environmental monitoring, spanning logistics and more over critical operations and defense, and as their deployment grows, so does their exposure to adversarial exploitation. The environment that drones operate are particularly difficult to secure and that occurs not only by the sensitivity of the data that these units produce, but by the structural heterogeneity of the data. Each drone functions under distinct mission conditions, encounters deferent threat patterns and generates traffic that reflects its specific operational context. This means that the non-IID problem, which is aa known challenge across federated learning generally, manifests here in a sharper and more consequential form than it does in more homogeneous network environments.

To address this, the authors propose BANCO-FL, a framework specifically built around the realities of UAV traffic rather than adapted from general purpose federated learning systems. The detection model at its core is a lightweight artificial neural network fully connected with two layers, batch normalization, and

dropout regularization, a deliberate architectural choice made after finding that more complex sequence-based models such as LSTMs and CNNs offered no classification advantage on the tubular UAV traffic features while demanding substantially more memory during training. BANCO-FL offers a more distinctive contribution that lies in its aggregation strategy. Instead of committing to a single aggregation rule, the proposed framework was evaluated across five different methods, including FedAvg, FedProx, FedAdam, FedMedian, and a clustering-based approach called ClusterAvg. Moreover, each client receives a proportional allocation of soft samples calibrated to its attack subset size, which prevents the class imbalance inherent in real drone traffic from undermining the global model during the aggregation stage. These experiments were conducted using the ISOT Drone dataset from the University of Victoria, a real-world collection of approximately 2.35 million traffic records capturing both normal communications and eight distinct attack categories including denial of service, man-in-the-middle, IP spoofing, payload manipulation and replay attacks. Two federated configurations were tested. The first configuration operated with three clients under strong heterogeneity, and the alternative option with nine clients where each node held exactly one attack type, representing a more granular and demanding non-IID setup. Across both configurations FedAdam, ClusterAvg, and FedMedian consistently outperformed the simpler aggregation strategies as shown in the diagram bellow. In the three-client setting, these methods achieved accuracy, precision, recall and F1-scores all approaching 99%, while FedAvg and FedProx showed more pronounced sensitivity to distributional mismatches across clients. The alternate nine client setup strengthened this pattern, with FedAdam and ClusterAvg upholding near-perfect classification even for the most heterogeneous clients, while FedAvg and FedProx logged more noticeable performance drops under the same conditions.

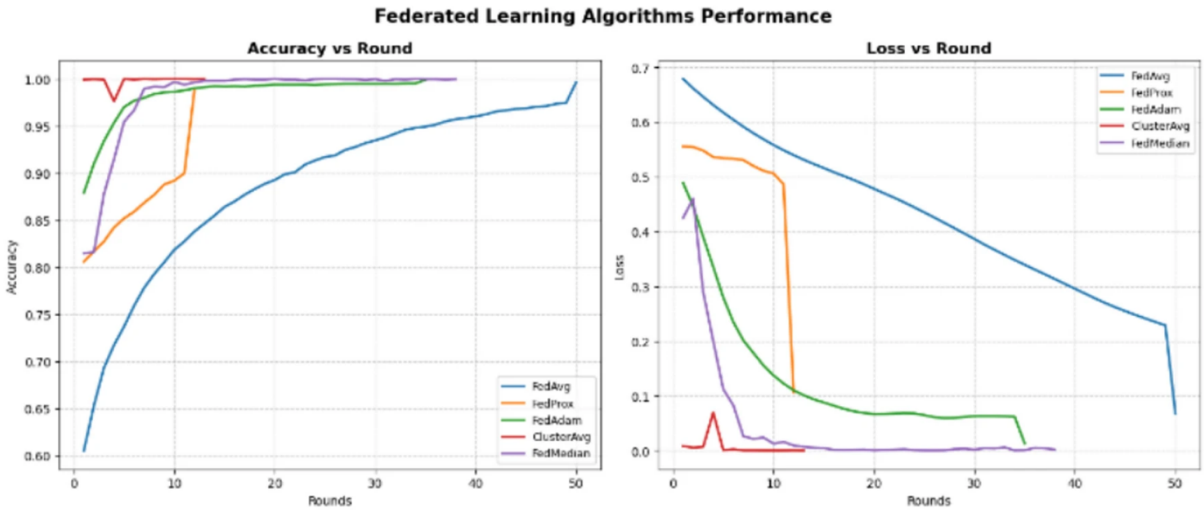

Figure 7: Training Progress (Accuracy & Loss): Shows convergence of all five aggregation methods over rounds (Bithi et al., 2026).

Beyond raw classification metrics, the authors also analyzed convergence behavior across training rounds, and the results here carry practical significance for real deployments. ClusterAvg and FedMedian reached near-optimal accuracy within approximately 33 communication rounds, while FedAdam stabilized shortly thereafter around round 40. FedAvg on the other hand, required close to 70 rounds to approach equivalent performance levels. In resource-constrained drone networks with limited bandwidth, where every unnecessary communication round carries a real operational cost, this gap in convergence efficiency is not merely a performance statistic but a deployment consideration. Taken together, BANCO-FL demonstrates that designing the aggregation with purpose and careful client-level data balancing can make federated anomaly detection genuinely viable in UAV settings, even when the participating nodes observe fundamentally different portions of the threat landscape (Bithi et al., 2026).

A further contribution to federated learning for UAV network security is presented by Zami M et al. (Zami et al., 2025), where the problem is approached from the standpoint of trust rather than only detection performance. The authors argue that in distributed aerial systems, it is not sufficient to assume that all participating nodes behave honestly during the training process. Since UAVs operate in open and potentially adversarial environments, compromised or malicious nodes may attempt to inject misleading updates into the global model. In order for this issue to be addressed, the proposed framework integrates federated learning with a verification mechanism that allows each participant to prove the correctness of its contribution without exposing the underlying data. This setting is achieved through the use of zero knowledge proof techniques, which enable a UAV to demonstrate that its locally trained model update follows the expected training procedure without revealing any sensitive information. To put it differently, the system presents an additional layer of trust that operates alongside the privacy guarantees already provided by federated learning. Moreover, the use of a digital twin representation for each participating UAV is a key component of the work mentioned (Zami et al., 2025). Instead of interacting directly with the physical device during the training process, the scheme preserves a virtual counterpart that mirrors its operational state, data characteristics and learning behaviors. This digital representation allows the system to monitor and evaluate of each UAV's contribution before it is accepted into the global aggregation stage. By comparing the expected behavior of the digital twin with the actual updates received, the framework can detect irregularities that may indicate flawed function or malicious interference. This approach effectively presents a validation layer that operates prior to aggregation, reducing the risk of corrupted updates influencing the global model. In highly dynamic UAV networks, where straightforward verification of each node may not always be feasible, the utilization of digital twins provides a structured way to maintain oversight of the learning process.

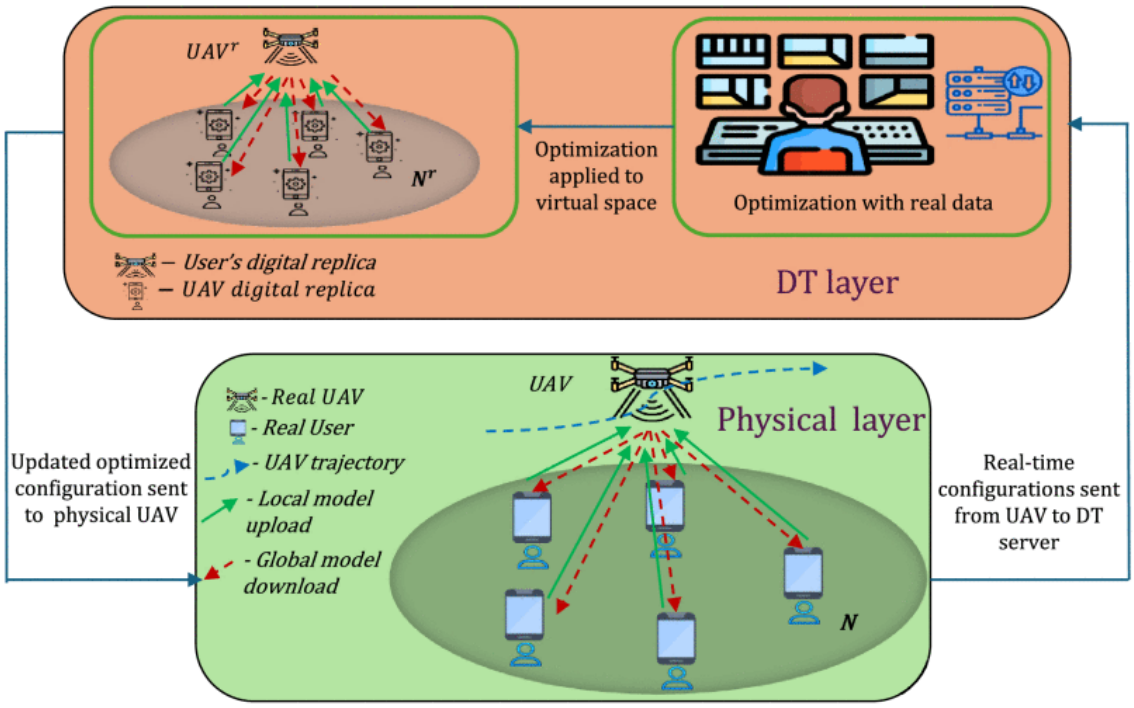

Figure 8: Demonstration the use of the digital twin that the research mentioned uses for additional trustworthiness (Zami et al., 2025)

The combination of digital twin and zero-knowledge verification revolutions how security is handled in federated UAV systems. Instead of relying solely on post-training evaluation or defensive aggregation strategies, the framework actively filters and validates updates before they are incorporated into the model. This reduces the likelihood of poisoning attacks and strengthens the overall reliability of the system, while simultaneously the design maintains the decentralized nature of federated learning, as raw data remains local and verification does not require exposing sensitive information. The work demonstrated that introducing trust-aware mechanisms at the protocol level can considerably enhance the robustness of collaborative learning in UAV networks, especially in scenario where nodes cannot be assumed to be fully trustworthy. This shifts the focus from purely improving detection accuracy, to ensuring that the learning procedure itself remains secure and demandable under adversarial conditions.

To conclude, the works examined above, collectively illustrate that federated learning has evolved from a privacy preserving alternative in centralized training into a comprehensive framework capable of addressing the broader security and operational demands of UAV networks. Early approaches emphasis on enabling distributed detection while protecting sensitive data, whereas more recent works extend this foundation by tackling issues such as data heterogeneity, aggregation reliability and the practical constraints of aerial environments. The introduction of adaptive aggregation toward more resilient and dependable systems. Despite these advancements, it also becomes evident that no single approach fully resolves all challenges associates with multi-UAV security. Questions remain regarding how to simultaneously achieve efficient communication, robust detection

performance and strong guarantees of trust under highly dynamic conditions. Tis highlights an open research direction where federated learning must continue to evolve, not only as a learning paradigm but as an integrated system that harmonizes privacy, reliability and security. The works presented in this chapter provide a solid and well-structured foundation, upon which further exploration and refinement can be thoughtfully developed.

CHAPTER 3

Methodology and System Architecture

This chapter describes the design and implementation of the proposed federated learning system for intrusion detection across a simulated multi-drone network. The core object of this work is to allow a group of five drone nodes to collaboratively build a shared intrusion detection model without any of them ever transmitting their raw sensor readings to a central location. Rather than treating security as something enforced from the outside, the system is designed so that privacy and detection capability are both built into the structure of the training process itself. The sections that follow cover each component of this system in turn, starting from how data is generated and consumed, moving through the local training procedure on each drone, the aggregation logic on the server side, and finally the evaluation mechanism used to assess how well the resulting model distinguishes between normal and malicious traffic.

3.1: System Overview

The overall architecture of the system follows the federated learning pattern, where multiple independent clients participate in one shared training process without ever pooling their data in one place. Each of the five drone nodes in this structure operates as a fully autonomous client that generates its own stream of sensor data, trains a local neural network on that data, and forwards only the resulting model parameters to a central aggregation server. The server collects these updates from all five participating drones, combining them into a single global model through a weighted averaging procedure, and finally it sends the updated parameters back to each client to begin the next round of training. This cycle runs for a fixed number of rounds, after which the best-performing version of the global model is saved on the disk and used for further evaluation.

From an infrastructure perspective, the system is built around two key technologies. "*Apache Kafka*" (*Apache Kafka*, n.d.) that handles the data production and consumption pipeline, acting as the communication layer throughout each drone's sensor readings before they reach the training module. Moreover, to run the entire system as a collection of isolated containers, we operate with the "Docker" platform (*Docker: Accelerated Container Application Development*, n.d.), while having each drone client and the central server operating in its individual environment whilst communicating over a shared internal network. This containerized design means that no component has direct access to another node's data, which reflects the privacy boundary that federated learning is intended to

enforce. The combination of *Kafka* for live data streaming and Docker for process isolation gives the system a structure that is reasonably close to how a real distributed deployment across physical devices would behave, even though in this implementation everything runs on a single machine.

Apache *Kafka* (*Apache Kafka*, n.d.) is an open-source distributed event streaming platform originally developed at "LinkedIn" and later released as an Apache Software Foundation project. At its core, *Kafka* operates as a messaging system that allows one or more producers, where in our case we have 5 producers for each drone, in order to publish streams of records to names channels called topics, which one or more consumers can then subscribe to and read from at their own pace. Unlike traditional messages for a configurable period, which makes it well suited for scenarios where data needs to be replayed, buffered or even consumed by multiple independent receivers. In the context of this system, *Kafka* acts as the intermediary layer between the aggregation process and the drone training clients, ensuring that sensor messages produced at one rate can be consumed and processed at another without either side being directly dependent on the timing of the other.

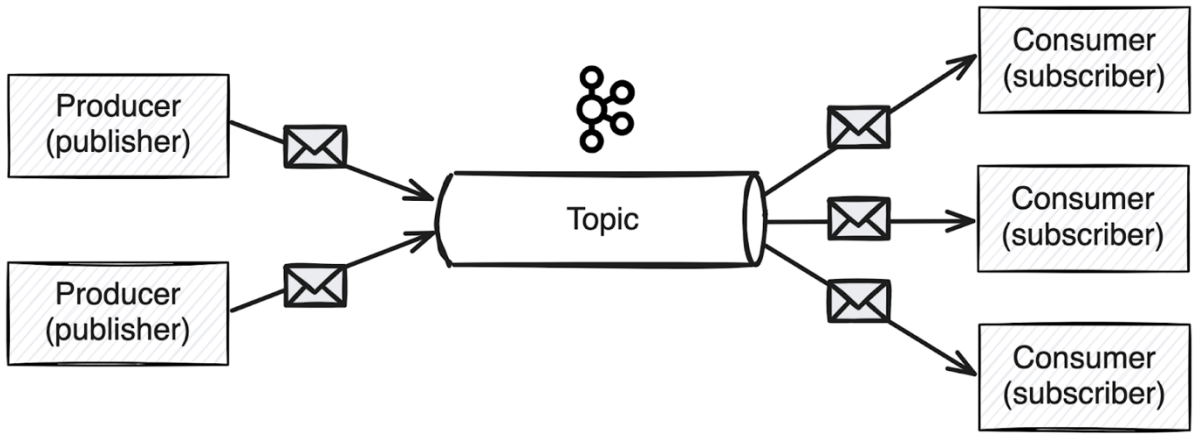

Figure 9: The diagram above illustrates the role of Kafka topic in assisting data transfer between producers and consumers (Introduction to the Apache Kafka Messaging System | Redpanda, n.d.).

Docker is an open-source containerization platform that allows applications and their dependencies to package into self-contained units called containers. Each container runs in isolation from the rest of the host system, with its own filesystem, environment variables and network interfaces, but shares the underlying operating system kernel, which makes containers considerably more lightweight than full virtual machines. This isolation is particularly valuable in distributed system development because it allows multiple independent components to run on the same physical machine while behaving as though they were separate nodes on a network. In this project, Docker is used to deploy each drone client and the central aggregation server as individual containers, ensuring that no component has unintended access to another node's data or processes. This setup closely mimics the boundary conditions of a real multi-drone deployment,

where each unit would operate as a physically separate device with no direct access to its peers.

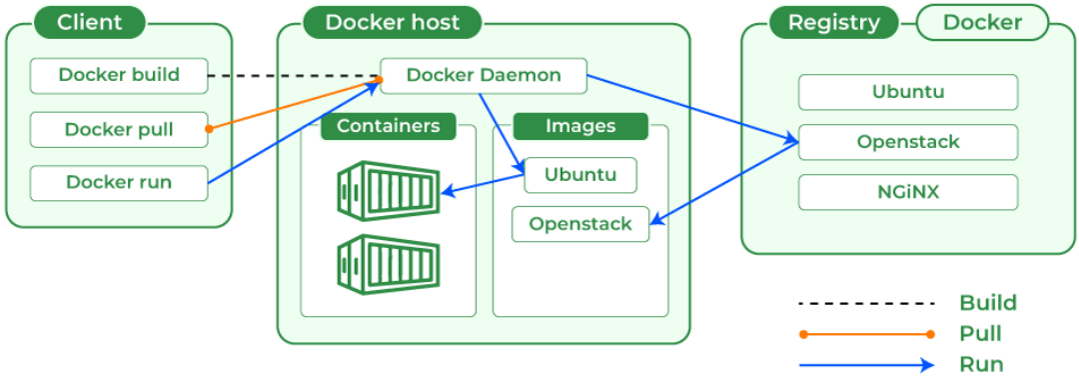

Figure 10: The diagram illustrates the core architecture of Docker, showing how a client issues commands to the Docker Daemon running on the host, which either pulls pre-built images from a remote registry or uses locally available ones to spin up running containers (What Is Docker? - GeeksforGeeks, n.d.).

3.1.1: Data Generation and Streaming Pipeline

The data generation layer of the system is built on Apache *Kafka*, which serves as the messaging backbone where each drone independently produces and consumes its own continuous stream of sensor readings. *Kafka* was chosen for this role because of its ability to decouple the data production process from the data consumption process, meaning that the rate at which messages are generated does not need to match the rate at which the training client reads and processes them. This buffering capability is particularly relevant in a simulated multi-drone environment where different components of the system may operate at different speeds, and where the training process should not be blocked or slowed down by momentary variations in message delivery timing.

For each drone, a dedicated *Kafka producer* instance runs independently and generates a fixed sequence of 20.000 messages over the course of its execution. Each message represents a single sensor observation and contains five fields: a Unix timestamp recording when the message was produced, and four numerical sensor readings covering temperature, humidity, pressure, and vibration. These four features were selected to represent the kinds of physical and environmental measurements that a real drone would potentially collect during operation, and their normal value ranges, were defined to reflect plausible stable flight conditions. Temperature readings under normal conditions are drawn from the range of 15 to 35 degrees, humidity from 30 to 80 percent, pressure from 900 to 1100 units, and vibration from 0 to 5 units. Each message also carries a binary label field that identifies whether the reading represents normal traffic or an intrusion event, with a value of zero indicating normal information and a value of one indicating an intrusion. The intrusion messages are generated using a distinct

set of values designed to simulate the kind of anomalous sensor behavior that might accompany a cyberattack or physical interference event on a drone. Intrusion temperature readings fall between 40 and 50 degrees, humidity between 90 and 150, pressure between 600 and 800, and vibration between 7 and 12. These ranges overlap partially with the normal operating ranges in some features but deviate significantly in others, which creates a realistic detection challenge where some intrusion samples are relatively easy to identify while others sit closer to the boundary between the two classes. This partial overlap is intentional, as a dataset where intrusions and normal readings were perfectly separable by simple threshold rules would not require a learned model at all and would not reflect the ambiguity present in real-world attack scenarios. Out of every 20,000 messages produced per drone, exactly 2,000 are labeled as intrusions, corresponding to a 10% intrusion rate across the full message stream. These ratios cover real-life intrusion percentage in order to mark the simulation as close to real threat measurements. The indices of these intrusion messages are selected randomly at the start of each producer run using a random sampling procedure without replacement, which ensures that attack events are scattered unpredictably throughout the sequence rather than appearing in any fixed pattern. This random placement reflects the reality that intrusions in operational systems do not arrive at predictable intervals and cannot be anticipated based on position alone. Each drone also appends its own identifier to every message it produces, allowing the downstream consumer to confirm which node a given reading originated from, though in the current implementation each client consumes only its own topic, and this field primarily serves a logging and debugging purpose. The producer is configured with several settings designed to maximize throughput and reliability during message generation. Acknowledgment is set to require confirmation from all in-sync replicas before a message is considered successfully delivered, which prevents data loss in cases where a broker failure occurs during production. Batching is enabled with a maximum of 10,000 messages per batch, and a minimal linger time is applied to allow small batches to accumulate before transmission, reducing the message overhead at high volumes. Furthermore, compression is applied to all outgoing messages, which reduces the network bandwidth consumed during transmission and decreases the storage footprint of retained messages on the *Kafka broker*. A small sleep interval of one millisecond is introduced between consecutive message productions to avoid overwhelming the broker with requests at very high rates, and the producer flushes its internal buffer explicitly every 5,000 messages to ensure that accumulated batches are committed to the broker in a timely fashion rather than waiting until the end of the full run.

On the consumer side, each drone client subscribes to its designated *Kafka topic* using a consumer group identifier that is configurable through an environment variable, allowing multiple instances of the same client script to be deployed with different group memberships if needed. The consumer polls for new messages in a loop with a one-second timeout on each poll call, appending each successfully decoded message to a local in-memory buffer as JSON. Rather than waiting for all 20,000 messages to arrive before beginning the training process, the system is configured with a minimum message threshold of 2,000, meaning that once the buffer contains at least this many readings the client will proceed to the training

stage. A maximum waiting period of 1,200 seconds is enforced as a safety timeout, after which the client will proceed with however many messages it has collected regardless of whether the threshold has been met. This early-start design is a practical trade-off that prioritizes system responsiveness, accepting that the first training round may be based on a partial view of the full data stream in exchange for avoiding indefinite blocking in cases where message delivery is slow or the broker experiences delays.

3.1.2: Data Preprocessing and Normalization

Before the collected sensor readings are passed into the neural network, they undergo a preprocessing stage that transforms the raw numerical values into a standardized form that is more suitable for the training of the model. This is done though z-score normalization, where each of the four input features is shifted by a predetermined global mean and divided by a predetermined global standard deviation. The specific normalization constants applied across all five drone clients are fixed values established in advance, with values that have being set to 25.637, 54.533, 998.012, and 2.578 for temperature, humidity, pressure, and vibration respectively, and the corresponding standard deviations set to 7.906, 15.396, 69.412, and 1.688. A small epsilon term of 1e-8 is added to each standard deviation during division to prevent numerical instability in cases where variance approaches zero. After the normalization stage is complete, the mean of each feature across the training samples should approach zero and the standard deviation should approach one, then the system prints these values explicitly after each normalization step as a verification check.

Using shared global statistics rather than normalization that has been computer per client, values locally as a deliberate and important design decision. If each drone normalized its own data independently, the resulting feature representations would differ across clients in ways that reflect the specific distribution of that client's local data rather than a common reference frame. When it's time for the server to aggregate the model parameters from all five clients, it would be combining weight matrices that were trained to operate on incompatible numerical scales, which would degrade the quality of the global model. By fixing the normalization constants globally and applying them identically across all participating nodes, the system ensures that every drone's local model operates in the same input space and that the aggregated parameters remain meaningful when combined.

After normalization, the whole dataset collected by each drone, is converted into *PyTorch tensors*, with the feature matrix cast as a float32 tensor and the label vector as a long integer tensor to satisfy the requirements of the cross-entropy loss function. These tensors are wrapped together in a *TensorDataset* object, which allows *PyTorch* to treat the paired features and labels as a single iterable dataset. This combined dataset is then partitioned into a training subset and a validation subset using a fixed 80/20 split, where the split index is computed as 80% of the total number of collected samples and the division is performed using *PyTorch's*

random_split utility. The training set is used exclusively for computing gradient updates during local training, while the validation set is held out and used only for monitoring generalization performance and triggering early stopping. Both subsets are wrapped in DataLoader objects configured with a batch size of 256, with the training loader using shuffling to randomize sample order across epochs and the validation loader also using shuffling to prevent any ordering effects during evaluation.

3.1.3: Local Model Architecture

The neural network that is being used by every drone client is a fully connected feedforward architecture implemented as a custom *PyTorch* module called Net. The network consists of five sequential linear transformation layers with the following structure. The first layer maps the four input features to 256 hidden units, the second reduces this to 128 units, then the third to 64 units, the fourth to 32 units and with the final output layer mapping the 32-dimentional representation to two output logits corresponding to the normal and intrusion datasets. Each of the first four layers is followed by a ReLU activation function, which introduces non-linearity into the model and allows it to learn complex decision boundaries that would not be possible with purely linear transformations. The output layer produces raw unnormalized scores rather than probabilities, as the cross-entropy loss function used during training applies a softmax transformation internally.

This specific narrowing of the hidden layer dimensions, from 256 down to 128, 64 and 32, is a standard design pattern in feedforward networks for tabular data. The model can construct rich intermediate representations of the input features, while the narrower later layers force the network to compress these representations into increasingly abstract summaries before making its final classification decision. All the above can be completed due to the wider early layers that are being used on the formula. For a four-feature input space, the architecture provides more than sufficient capacity to learn the separation between normal and intrusion patterns while remaining computationally lightweight enough to train efficiently within a containerized environment with limited resources. Adam optimizer is the technique used for the training, which is configured with a learning rate of 1e-4 and a weight decay coefficient of the same value. The weight decay acts as an L2 regularization penalty that discourages the model from assigning unreasonably large magnitudes to individual weight parameters, which helps prevent overfitting particularly when the local training dataset is small relative to the complexity of the model. The loss function that is being used throughout the training process is a weighted cross-entropy criterion with class weights set to 1.0 for the normal class and 9.0 for the intrusion class. This asymmetric weighting is a direct response to the class imbalance in the data, where intrusions account for only 10% of all messages. Without this correction, a model trained on this distribution would be strongly encouraged to predict the majority class on every sample, achieving a raw accuracy of around 90% while completely failing at the actual detection task. Be assigning a ninefold higher penalty to intrusion

misclassifications, the loss function rebalances the effective contribution of each class during training and pushes the model to treat rare attack events as genuinely important. Each local training session is capped at a maximum number of epochs specified by the server configuration but also incorporates an early stopping mechanism that can terminate the training before reaching the model limits. Then after each epoch, the model is switched to evaluation mode, and the validation loss is computed over the held-out validation set without any gradient computation. A patience counter is incremented then if this validation loss fails to improve compared to the best value recorded so far into training. Once this counter reaches the number of three consecutive epochs without improvement, training is halted and the best validation loss recorded up to that point es reposted. The patience-based early stopping that was chosen to be used on this work, ensures that each drone submits a model that has not simply memorized its local training data, and also prevents unnecessary computation in rounds where the model converges quickly.

At each hidden layer, the transformation applied to an input vector is expressed as:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Formula 1.0

where **W**is the learned weight matrix and **b**is the bias vector associated with that layer. For a network with *L*layers, the full forward computation can be expressed as a composition of these transformations applied sequentially:

$$\mathbf{h}^{(l)} = f\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right), l = 1, \dots, L-1$$

Formula 1.1

where $\mathbf{h}^{(0)} = \mathbf{x}$ is the raw input vector and $\mathbf{h}^{(l)}$ denotes the activation vector produced at layer l. The ReLU activation function f applied after each hidden layer is defined element wise as:

$$f(z) = \max(0, z)$$

Formula 1.2

The final output layer omits the activation function and produces the raw logits directly:

$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$$

Formula 1.3

These logits are passed into the softmax function, which converts them into a valid probability distribution over the two classes:

$$P(\text{class} = k \mid \mathbf{x}) = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

Formula 1.4

where the denominator sums over both output units, ensuring the resulting values sum to one (*CS231n Deep Learning for Computer Vision*, n.d.). The weighted cross-entropy loss is then computed as:

$$\mathcal{L} = -\sum_k w_k \cdot y_k \cdot \log P(\text{class} = k \mid \mathbf{x})$$

Formula 1.5

where $y_k$is a one-hot indicator for the true class and $w_k$is the class weight, set to 1.0 for normal and 9.0 for intrusion (*CrossEntropyLoss — PyTorch 2.11 Documentation*, n.d.). The gradients of this loss with respect to the model parameters are computed through backpropagation, which applies the chain rule of calculus recursively from the output layer back toward the input. For the weight matrix at layer l, the gradient takes the form:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \cdot \left(\mathbf{h}^{(l-1)}\right)^{\top}$$

Formula 1.6

where $\partial \mathcal{L} / \partial \mathbf{z}^{(l)}$is the error signal propagated back from the layer above and $\left(\mathbf{h}^{(l-1)}\right)^{\top}$is the transpose of the incoming activation vector (Nielsen, 2015). These gradients are then consumed by the Adam optimizer to update the parameters.

The Adam optimizer tracks two running statistics for each parameter. The first moment $m_t$represents a moving average of past gradients, and the second moment $v_t$represents a moving average of their squared values:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

Formula 1.7

Because both quantities are initialized at zero, bias-corrected estimates are computed at each step to account for the underestimation that would otherwise occur early in training:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Formula 1.8

The parameter update rule then becomes:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \varepsilon}$$

Formula 1.9

where $\eta = 10^{-4}$is the learning rate and $\varepsilon$is a small constant included for numerical stability (Kingma & Ba, 2014). The L2 regularization introduced by the weight decay term modifies the effective loss function that the optimizer minimizes to:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2}\sum_i \theta_i^2$$

Formula 1.10

where $\lambda = 10^{-4}$is the weight decay coefficient and the sum runs over all trainable parameters in the network (Ng, 2004). The early stopping condition applied after each epoch can be stated as:

$$\mathcal{L}_{\text{val}}(t) \geq \mathcal{L}_{\text{val}}(\text{best})\text{for three consecutive values of } t$$

where the validation loss at epoch $t$is computed as:

$$\mathcal{L}_{\text{val}}(t) = -\frac{1}{N_{\text{val}}}\sum_i \sum_k w_k \cdot y_{i,k} \cdot \log\, P(\text{class} = k \mid \mathbf{x}_i)$$

Formula 1.11

and $N_{\text{val}}$is the total number of validation samples. When this condition is met, training is halted regardless of whether the maximum epoch count has been reached (Prechelt, 2012).

3.1.4: Federated Aggregation

The server-side section of the system is implemented using the *Flower* federated learning framework and manages the fill coordination of the training process across all five drone clients. Rather than using the default FedAvg strategy provided by *Flower* directly, the systems introduce a custom extension called FedAvgWithGlobalAccuracy, which adds two important capabilities on top of the standards weighted averaging procedure. The first is the ability to track the best global accuracy recorded across all training rounds, and the second is the ability to save model checkpoints to disk at two different points in the training lifecycle. The server of this scheme is configured with a set of strict participation requirements that must be satisfied before any aggregation can take place. Both

the *fraction_fit* and *fraction_evaluate* parameters are set to 1.0, and the minimum number of clients required for both training and evaluation phases is set to five. With that, the server will not proceed with the aggregation process in any round unless all five drone clients have submitted their updated and will not record an evaluation result unless all five have also returned their local validation metrics. This propositions a fully synchronous configuration, ensuring that the global model is always constructed from the contributions of the complete federation rather than a subset, which is important for maintaining fairness and consistency across a small number of clients.

At the start of each round, the server sends a configuration dictionary to every participating client that specifies the learning rate the number of local training epochs, and the early stopping patience value to be used during that round. For all ten training rounds, the learning rate is fixed at 1e-4, and the number of local epochs is set to one and the patience is set to three. The number assigned to the local epochs means that each client performs only a single pass over its local dataset before returning its updated parameters to the server, which is a conventional choice that keeps local updates small and reduces the risks of client drift, a phenomenon where individual models diverge too far from the global model through extended local training. After receiving and aggregating the updates from all five clients, the server reconstructs a full *PyTorch* model from the aggregated parameter arrays and saves it to disk, overwriting the previous checkpoint. Separately, after the evaluation phase of each round, the server computes aa weighted global accuracy across all clients by combining each client's reported accuracy proportionally to the size of its validation dataset. If this global accuracy exceeds the best value recorded in any previous round, the current aggregated model is saved as a second checkpoint. This ensures that regardless of what happens in the final rounds of training, the highest-performing version of the model is always preserved and available for evaluation.

Before the first federated round begins, the server initializes a fresh instance of the neural network and extracts its parameter arrays using the same architecture defined on the client side. These initial parameters are converted into a Flower compatible format and passed to the strategy as the starting global model, ensuring that all five clients begin round one from an identical weight initialization rather than from independently randomized starting points. This matters because if clients began from different random initializations, the first aggregation would average parameters that have no meaningful correspondence to one another, producing a global model that is essentially noise. Starting from a common point guarantee that the averaging operation is meaningful from the very first round.

The communication between the server and each client runs over gRPC, which Flower uses by default to handle the serialization and transmission of parameter tensors. At the start of each round, the server broadcasts the current global parameters and the configuration dictionary to all five clients simultaneously. Each client deserializes the received parameters, loads them into its local model using PyTorch's state dictionary mechanism, runs local training, and then

serializes the updated parameters back into a format that Flower can transport. The server collects the responses and, once all five submissions have arrived, passes them to the aggregation function. If a client fails to respond within the round, the server does not fall back to a partial aggregation; it simply does not proceed, which is a direct consequence of the minimum client requirements being set equal to the total number of participants.

The server-side section of the system is implemented using the *Flower* federated learning framework and manages the full coordination of the training process across all five drone clients. The system introduces a custom extension of the standard *FedAvg* strategy called *FedAvgWithGlobalAccuracy*, which adds checkpoint saving and global performance tracking on top of the core aggregation procedure. The aggregation itself follows the standard *FedAvg* formulation, where the global model parameters at round $t + 1$are computed as a weighted average of the local parameter vectors submitted by all participating clients:

$$\theta^{(t+1)} = \sum_{k=1}^{K} \frac{n_k}{N} \cdot \theta_k^{(t)}$$

Formula 2.0

where $\theta_k^{(t)}$denotes the local parameter vector submitted by client $k$at round t, $n_k$is the number of training samples held by client k, and $N = \sum_{k=1}^{K} n_k$is the total number of training samples across the full federation (Brendan McMahan et al., 2016). This weighting ensures that clients with larger local datasets contribute proportionally more to the resulting global model.

The server is configured with strict participation requirements that must be satisfied before aggregation proceeds. Both fraction_fit and fraction_evaluate are set to 1.0, and the minimum number of required clients for both phases is set to $K = 5$. This enforces a fully synchronous training regime where every round satisfies:

$$| \mathcal{S}^{(t)} | = K = 5$$

where $\mathcal{S}^{(t)}$denotes the set of clients that have submitted updates in round t. If any client fails to respond, aggregation does not take place for that round.

At the start of each round t, the server broadcasts a configuration dictionary to all participating clients specifying the learning rate $\eta = 10^{-4}$, the number of local epochs $E = 1$, and the early stopping patience $p = 3$. Setting $E = 1$means that each client performs a single local gradient update before returning its parameters. This limits the degree to which any individual client can diverge from the global model, a phenomenon known as client drift. The drift experienced by client $k$after E local epochs can be bounded by the divergence between its local objective $F_k(\theta)$and the global objective $F(\theta)$:

$$F(\theta) = \sum_{k=1}^{K} \frac{n_k}{N} F_k(\theta)$$

Formula 2.1

where $F_k(\theta)$ is the local loss function minimized by client k on its own data (Karimireddy et al., 2019). When local data distributions differ significantly across clients, the gap between $F_k(\theta)$ and $F(\theta)$ grows, and extended local training amplifies this divergence. Keeping E small directly controls this effect.

Each client computes its local parameter update by minimizing its local loss through one pass of gradient descent. The update sent to the server by client k at round t takes the form:

$$\Delta\theta_k^{(t)} = \theta_k^{(t)} - \theta^{(t)}$$

where $\theta^{(t)}$ is the global model distributed at the start of round t and $\theta_k^{(t)}$ is the locally updated model after E epochs of training. The server then reconstructs the new global model by applying the weighted aggregation:

$$\theta^{(t+1)} = \theta^{(t)} + \sum_{k=1}^{K} \frac{n_k}{N} \cdot \Delta\theta_k^{(t)}$$

Formula 2.2

which is algebraically equivalent to the weighted average stated earlier but makes explicit the additive nature of the aggregation step (Brendan McMahan et al., 2016).

After the evaluation phase of each round, the server computes a weighted global accuracy across all clients proportional to their validation dataset sizes:

$$\mathrm{Acc}_{\mathrm{global}}^{(t)} = \sum_{k=1}^{K} \frac{m_k}{M} \cdot \mathrm{Acc}_k^{(t)}$$

Formula 2.3

where $\mathrm{Acc}_k^{(t)}$ is the accuracy reported by client k during round t, m_k is the number of validation samples held by client k, and $M = \sum_{k=1}^{K} m_k$ is the total number of validation samples across the full federation. The best global accuracy recorded over all $T = 10$ rounds is tracked as:

$$\mathrm{Acc}_{\mathrm{best}} = \max_{t \in \{1,\ldots,T\}} \mathrm{Acc}_{\mathrm{global}}^{(t)}$$

Formula 2.4

Whenever the condition:

$$\mathrm{Acc}_{\mathrm{global}}^{(t)} > \mathrm{Acc}_{\mathrm{best}}$$

is satisfied, the current aggregated parameter vector $\theta^{(t+1)}$is saved as a separate best model checkpoint. This guarantees that the highest-performing version of the global model across the entire training process is always preserved and available for the subsequent evaluation stage, regardless of whether performance improves or degrades in the final rounds (*Strategy - Flower Framework*, n.d.).

3.1.5: Intrusion Prediction and Evaluation

Once the federated training process is complete, the saved global model then is evaluated using a dedicated prediction script that applies a more sophisticated assessment strategy than straightforward single-pass classification. The evaluation used on this work is built around the idea that combining information from the trained model with information derived directly from the structure of the raw sensor readings, produces more reliable final judgement than either source alone. This is particularly relevant in intrusion detection, where the cost of a missed attack typically outweighs the cost of a false alarm, and where additional corroborating signals can help resolve ambiguous cases that fall near the classifier's decision boundary.

To understand why a single forward pass is not enough, it helps to think about what a neural network is actually doing when it produces an output probability. The softmax values that come out of the final layer are a function of the input features, the learned weights, and the initialization-dependent quirks that survived training. For inputs that sit clearly inside the normal region or clearly inside the intrusion region, the output is stable and a single pass is sufficient. For inputs near the boundary, small numerical differences in how the input is represented can push the output slightly above or below the decision threshold. Test-time augmentation addresses this by sampling from the neighborhood of the input rather than evaluating the model at a single point, which gives a more representative picture of how the model behaves across the local input region and reduces the chance that a borderline sample is misclassified due to a spurious numerical effect rather than a genuine signal.

The first component of the evaluation is the classifier probability, which is obtained by passing a normalized sensor reading through the trained neural network and applying a softmax transformation to the output logits. Rather than performing this forward pass once on every sample, the system applies test-time augmentation by running each input through the model sixteen times, each time adding a small amount of Gaussian noise with a standard deviation of 0.01 to the normalized feature vector before inference. The intrusion probability reported for that sample is then the mean of the sixteen individual softmax outputs, and the standard deviation across these runs is recorded as a measure of prediction uncertainty. The choice of sixteen augmentation passes reflects a balance between

stability and computational cost. At very low counts, say two or three passes, the mean probability estimate can still be noisy enough that repeated evaluations of the same sample would give noticeably different results. At higher counts, the estimate stabilizes but the inference time grows proportionally. Sixteen passes proved sufficient in practice to produce consistent estimates across repeated runs of the same input, and the standard deviation of the individual pass outputs dropped to values below 0.005 for well-separated samples in both experimental configurations. For samples near the decision boundary the standard deviation remains higher, which is exactly the behavior the uncertainty measure is intended to capture. This repeated noisy inference approach makes the final probability estimate more robust to small perturbations in the input and provides a sense of how confidently and consistently the model responds to a given reading. Samples that produce high variance across augmentation runs can be treated with more caution than those where the model responds consistently.

The second component is a feature-based anomaly score that measures how unusual a reading from the sensor is relative to the known distribution of normal drone operation, computed without any involvement from the trained model. This score is derived using a Mahalanobis calculation in which the raw feature vector is first transformed into z-scores using the same global normalization constants applied during training, and the sum of squared z-scores is then computed across all four features. This quantity captures how far the reading is from the center of the normal distribution in a standardized space where each feature contributes proportionally to its expected variance. The resulting distance value is then mapped to a bounded score between zero and one using a scaled exponential transformation, where higher distances produce scores closer to one. A reading that falls well within the expected ranges for temperature, humidity, pressure, and vibration will receive a score close to zero, while a reading with extreme values in one or more features will receive a score approaching one regardless of how the classifier responds to it.

These two signals are then combined into a single composite score using a weighted linear combination in which the classifier probability contributes 75% and the feature anomaly score contributes the remaining 25%, controlled by a parameter called alpha set to 0.75. A composite score greater than 0.5 results in the sample being classified as an intrusion, while anything below this threshold is classified as normal. The choice of alpha equal to 0.75 was not arbitrary, but it was also not derived from formal optimization. The reasoning behind it is that the trained neural network, having seen thousands of labeled examples from five different drones across ten federated rounds, should carry substantially more information about the intrusion pattern than a distance measure computed from four numbers and a set of global statistics. The feature anomaly score is useful precisely because it operates independently of the training history, which means it can flag readings that look structurally unusual even when the classifier is uncertain, but it is a cruder signal that knows nothing about the specific boundaries learned during federation. Setting alpha at 0.75 reflects this asymmetry: the classifier drives the decision, and the anomaly score acts as a second opinion that can shift the composite score without overriding the classifier

entirely. A value closer to 1.0 would effectively ignore the anomaly score, while a value closer to 0.5 would give it equal weight, which is hard to justify given how much more information the classifier contains.

The heavier weighting toward the classifier probability reflects the expectation that the trained model, having learned from thousands of labeled examples across all five drone clients, carries more discriminative information than the raw distance measure alone. The feature anomaly score nevertheless plays a meaningful supporting role, acting as a structural check that can reinforce or temper the classifier's output based on how anomalous the raw readings appear relative to the normal operating envelope. The alpha value of 0.75 was set based on design judgment informed by the relative reliability of each component, and systematic calibration of this parameter against a labeled validation set using techniques such as Youden's J statistic represents a straightforward avenue for further refinement.

The classifier probability for a given sample is estimated through test-time augmentation, where the input is passed through the trained network N_{tta} = 16times, each time with a small perturbation drawn from a Gaussian distribution added to the normalized feature vector:

$$\tilde{\mathbf{x}}^{(i)} = \mathbf{x} + \boldsymbol{\epsilon}^{(i)}, \boldsymbol{\epsilon}^{(i)} \sim \mathcal{N}(0, \sigma^2\mathbf{I}), \sigma = 0.01$$

Formula 3.0

where $\tilde{\mathbf{x}}^{(i)}$is the perturbed input at augmentation run $i$and $\mathbf{I}$is the identity matrix. The classifier probability reported for that sample is then the mean softmax output across all augmentation runs:

$$\hat{p} = \frac{1}{N_{tta}} \sum_{i=1}^{N_{tta}} P(\text{intrusion} \mid \tilde{\mathbf{x}}^{(i)})$$

Formula 3.1

The prediction uncertainty is quantified by the standard deviation of these individual probability estimates:

$$\hat{\sigma}_p = \sqrt{\frac{1}{N_{tta}} \sum_{i=1}^{N_{tta}} (P(\text{intrusion} \mid \tilde{\mathbf{x}}^{(i)}) - \hat{p})^2}$$

Formula 3.2

A high value of $\hat{\sigma}_p$indicates that the model responds inconsistently to small input perturbations, which signals low prediction confidence for that particular sample (Lakshminarayanan et al., 2016).

The second component of the evaluation is the feature-based anomaly score, which is computed independently of the trained model using a Mahalanobis-style distance measure. The raw feature vector $\mathbf{x}_{\text{raw}}$ is first transformed into a standardized z-score representation using the global normalization statistics:

$$\mathbf{z} = \frac{\mathbf{x}_{\text{raw}} - \boldsymbol{\mu}}{\boldsymbol{\sigma} + \varepsilon}$$

Formula 3.3

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the global mean and standard deviation vectors applied during training and $\varepsilon = 10^{-8}$ is a small stability constant. The squared Mahalanobis distance under a diagonal covariance assumption is then computed as the sum of squared z-scores across all $d = 4$features:

$$D(\mathbf{x}_{\text{raw}}) = \sum_{j=1}^{d} z_j^2 = \| \mathbf{z} \|^2$$

Formula 3.4

This quantity measures how far the raw reading sits from the center of the known normal operating distribution in a variance-normalized space (De Maesschalck et al., 2000). A large value of Dindicates that the reading is structurally anomalous regardless of the classifier's response. This distance is then mapped to a bounded score in the interval [0,1)using a scaled exponential transformation:

$$s_{\text{feat}}(\mathbf{x}_{\text{raw}}) = 1 - \exp\left(-\frac{D(\mathbf{x}_{\text{raw}})}{d \cdot 2} \right)$$

Formula 3.5

where the scaling factor $d \cdot 2$normalizes the distance relative to the dimensionality of the feature space, ensuring that the mapping behaves consistently regardless of how many features are involved.

The two signals are then combined into a single composite score through a weighted linear combination controlled by a mixing parameter $\alpha = 0.75$:

$$s_{\text{combined}} = \alpha \cdot \hat{p} + (1 - \alpha) \cdot s_{\text{feat}}(\mathbf{x}_{\text{raw}})$$

Formula 3.6

The final classification decision is made by applying a fixed threshold $\tau = 0.5$to this composite score:

$$\hat{y} = \begin{cases} 1 & \text{if } s_{\text{combined}} \geq \tau \\ 0 & \text{if } s_{\text{combined}} < \tau \end{cases}$$

Formula 3.7

where $\hat{y} = 1$denotes an intrusion and $\hat{y} = 0$denotes normal traffic (Fawcett, 2006). The value of $\alpha$was set based on design judgment reflecting the expectation that the trained classifier carries more discriminative information than the raw distance measure alone, and systematic calibration of this parameter against a labeled validation set using techniques such as Youden's J statistic represents a natural direction for further refinement.

3.2: Implementation Environment and Tools

The local model training component of the system is built on *PyTorch*, an open-source deep learning library developed by *Meta's AI Research* team that has become one of the most widely adopted frameworks in both academic research and practical machine learning development. *PyTorch* builds its computational graphs dynamically at runtime rather than compiling them statically in advance, which means that the structure of the network can be inspected, modified and debugged at any point during execution in the same way as ordinary Python code. This characteristic makes it considerably more transparent than statically compiled alternatives and particularly well suited to research contexts where the model architecture or training procedure may need to be adjusted iteratively. In this project, *PyTorch* is used to define the fully connected neural network on each drone unit, managing the tensor operations involved in forward and backward passes, implement the weighted cross-entropy loss function, handle the *Adam optimizer* updates and organize the training and validation data through its *DataLoader* and *TensorDataset* utilities.

The federated learning coordination layer is managed by *Flower*, an open-source framework that originated from research at the University of Oxford and was designed from the ground up to make federated learning practical across heterogeneous and resource-constrained environments. Before frameworks such as *Flower*, researchers who wanted to implement federated learning had to build their own parameter serialization logic, communication protocols, round management systems, and aggregation procedures from scratch, which represented a noteworthy engineering effort on top of the actual research contribution. *Flower* addresses this by providing a clean client-server abstraction where the low-level details of connection management, message passing, and protocol handling are taken care of internally, leaving the researcher free to focus entirely on the learning logic. The client-side interface requires implementing only a small number of methods covering how the model receives parameters, how it

trains locally, and how it returns updates, while the server side allows the aggregation strategy to be defined and extended with minimal extra code. *Flower* is also entirely framework oriented, meaning it places no restriction on which machine learning library is used on the client side, which is what allows *PyTorch* to be used seamlessly as the local training backend without any compatibility issues.

NumPy is used throughout the whole system for numerical array operation that sit outside the *PyTorch* computation graph. While *PyTorch* handles all tensor operations during model training and inference, there are several parts of the pipeline where ray numerical data needs to be processed before it enters the model or after it leaves it. On the client side, the sensor readings that has been collected from the *Kafka* buffer are first assembled into *NumPy* arrays before being converted into *PyTorch* tensors, and the global normalization constants used during the preprocessing stage are defined and applied as *NumPy* arrays. On the evaluation side, the intrusion prediction script relies heavily on *NumPy* for computing the feature-based anomaly score, including the z-score transformation, the squared distance calculation, and the exponential mapping to a bounded score. *NumPy* is also used in this project to compute the mean and standard deviation of the classifier probability estimates across the test-time augmentation runs, and to generate the random Gaussian noise vectors added to each input during augmentation. Its close iteration ability with *PyTorch*, where *NumPy* arrays can be converted to tensors and back with minimal effort, makes it a natural companion library for handling the parts of the pipeline that do not require gradient computation.

Python's standard threading module is used in the client script to run the *Kafka* consumer in a background thread while the main execution thread monitors the message buffer and waits for the minimum threshold to be reached before proceeding to training. This separation is necessary because the *Kafka* polling loop is a blocking operation that would otherwise prevent any other logic from executing and running it in a dedicated daemon thread, that allows the rest of the program to remain responsive. The socket module from the Python standard library is used to verify that the *Flower* server is reachable before the client attempts to establish a federated connection, implementing a simple retry loop that checks port availability at regular intervals and exits gracefully if the server does not become ready within the allowed number of attempts. Finally, *JSON* serialization is used as the message format for all *Kafka* communications, with each sensor reading encoded as a *JSON* object by the producer and decoded back into a Python dictionary by the consumer before being appended to the local data buffer. These three components are part of the Python standard library and require no additional installation on the project, but their roles in keeping the system reliable and well-coordinated during runtime are worth noting.

3.3: System Deployment and Communication Flow

The entire system that is implemented, is deployed as a collection of *Docker* containers that run simultaneously on the same host machine while remaining isolated from one another at the process and network level. Each container operates within a shared *Docker* network, which gives the components a way to address and communicate with each other using service names rather than raw IP addresses, while still maintaining the boundary that prevents any container from accessing another's filesystem or environment directly. The implementation consists of a *Kafka* broker container, a *Flower* server container, and five drone client containers, each of which runs as an independent instance of the same client script configured with a different identity through environment variables. The environment variables passed to each drone container control its client identifier, the *Kafka* topic it produces and consumes from, the consumer group it belongs to, and the address of the Flower server it will connect to. This approach means that the same codebase serves all five drone nodes without any modification, and the behavioral differences between them arise entirely from their runtime configuration rather than from changes to the underlying logic.

The startup sequence of the system follows a strict dependency order that must be respected for the components to connect successfully. The Kafka broker must be fully operational before any producer or consumer can interact with it in order for the system to work correctly, since both sides depend on the broker being available to register topics, accept incoming messages, and serve buffered records to subscribers. Once Kafka is running, all five drone producer instances can begin generating and publishing their sensor message streams to their respective topics. The Flower server must also be started and listening on its designated port before any of the drone clients attempt to initiate a federated connection, since a client that tries to connect before the server is ready will either fail immediately or enter a never-ending retry loop. To handle this dependency, each drone client runs a pre-connection check using Python's socket module that repeatedly attempts to open a TCP connection to the server's host and port at regular intervals. If the connection is successful, the client moves on with the federated training process. If the server does not become reachable within the configured number of retries, the client logs a failure message and exits rather than hanging indefinitely.

The communication between the *Flower* server and each drone client takes place over gRPC, a high-performance remote procedure call protocol that *Flower* uses internally to exchange model parameters and configuration messages between the server and its clients. From the perspective of the system implementation, this communication layer is entirely managed by the *Flower* framework and does not require any explicit networking code in either the client or server scripts. At the start of each federated round, the server serializes the current global model parameters into a compatible format with the *Flower* protocol and sends them to all connected clients along with a configuration dictionary specifying the learning rate, number of local epochs, and early stopping patience for that round. Each client deserializes the received parameters, loads them into its local *PyTorch*

model, runs the local training procedure over its own dataset, and then serializes the updated parameters back into the *Flower* protocol format before returning them to the server alongside its local dataset size and training loss. The server collects these responses from all five clients, applies the weighted *FedAvg* aggregation to produce the updated global model, saves a checkpoint to disk, and then moves to the evaluation phase where each client evaluates the new global model on its local validation set and reports its accuracy back to the server. The server computes the weighted global accuracy, updates the best model checkpoint if a new peak has been reached, and then begins the next round by distributing the updated parameters again. This cycle repeats for all ten configured rounds, after which the server shuts down, and the best performing version of the global model remains saved on disk ready for the evaluation stage described in section 3.1.5. The final algorithm used on this project then deploys in order to show its truthfulness with ten data paradigms showing the spot that the decision of normal or intruded data label is being taken.

The overall flow from system starting point to the final model persistence can therefore be summarized as a linear sequence of stages. The infrastructure layer starts first with *Kafka* and *Docker* establishing the communication backbone. The data layer is being activated next as each drone producer begins streaming sensor messages to its topic and each client consumer begins collecting them. The learning layer then takes over as the clients reach their pre-determined message threshold, connect to the *Flower* server, and begin the ten-round federated training cycle. Finally, the evaluation layer operates independently on the saved model once training is complete, applying the combined classifier and anomaly scoring procedure to assess detection performance. Each of these stages depends on each other, and the design choices made at each layer, from the *Kafka* buffering threshold to the synchronous aggregation requirement to the best model checkpointing logic, are all oriented toward ensuring that this sequence runs reliably and produces a global model that genuinely reflects the contributions of all five participating drone units.

The work presented in this chapter provides a complete description of the system designed and implemented for this thesis, covering every layer starting from the raw data generation pipeline through to the final intrusion scoring mechanism. The five drone nodes, each operating as an independent federated client with its own Kafka data stream, local neural network, and isolated container environment, they all demonstrate together how privacy-preserving distributed learning can be applied to a security-sensitive autonomous system without centralizing any of the fundamental sensor data. The architectural decisions discussed throughout this chapter, including the choice of synchronous aggregation, the global normalization constants shared across all clients, the asymmetric class weighting, and the combined anomaly scoring approach used during evaluation, were all made with the specific operational limitations of a multi-drone environment in mind rather than adopted as generic defaults. Together, they form a system that is not only technically coherent, but also practical for using machine learning in places with limited resources and where the data is spread out. Having established the full design and implementation of the proposed system, the following chapter presents

the experimental results obtained by running the whole project, examining how the global model performs across the ten federated training rounds and evaluating the effectiveness of the intrusion detection mechanism against both normal and malicious sensor readings.

CHAPTER 4

Results and Performance Assessments

This chapter presents the results obtained from two separate experimental runs of this work's federated learning system. Both runs are very similar to each other, the only difference is in the number of Kafka messages each drone client waits to collect before starting the training process. The first configuration limits each drone unit to 2.000 messages per drone, which gives each client roughly one tenth of the full dataset before starting the training process. This technique is used in order to save time on the complete run of the algorithm. The second configuration starts training later while waiting for the full dataset of 20.000 messages per drone. Both experimental runs are trained for 10 federated rounds with the same neural network architecture, the same aggregation strategy and the same intrusion detection evaluation pipeline. The goal of comparing them is to understand how much the amount of local data affects both the training dynamics and the final detection quality.

4.1: Experimental Setup

The entire system runs inside Docker containers, with one container per drone client and one container for the Flower server, alongside the Kafka broker and ZooKeeper. Five drone nodes participate in each experiment. Each producer publishes 20.000 sensor messages to a dedicated Kafka topic, randomly placing 2.000 of them as intrusion events, which gives a 10% intrusion rate across the fill stream. The sensor data covers four features: temperature, humidity, atmospheric pressure and vibration. Normal readings follow the ranges used during training, while intrusion readings present abnormal conditions such as elevated temperature, low humidity, pressure drops and high vibration values. Each client uses the same five layer fully connected neural network with input dimension 4 expanding to 256 neurons in the first hidden layer and then compressing through 128, 64 and 32 neurons before the two-class output. The class weights used during training are 1.0 for the normal class and 9.0 for the intrusion class, which compensates for the imbalance in the data. Local training in each federated round runs for a single epoch with early stopping set at patience 3. The Adam optimizer is used with a learning rate of 0.0001 and weight decay of 0.0001. Global normalization constants are applied to all feature vectors before training, computed offline from a representative dataset.

In the 20.000 message configuration, clients start connecting to the Flower server only after they have collected the full amount of messages from Kafka. On the other hand, on the 2.000 message configuration, clients begin training after only after the 2.000 messages have been buffered. This means that in the first run, each client has approximately 1.600 training samples among with 400 validation samples, compared to roughly 16.000 and 4.000 respectively in the first run. Everything else, including the FedAvg aggregation, the 10 round schedule, and the evaluation pipeline, stays the same between the two runs.

4.2: Federated Training Convergence

The most visible difference between the two configurations appears in how quickly the global loss drops across the 10 federated rounds. The following figure (Figure 4.2) plots the global loss on a logarithmic scale for both runs, and the gap between them is quite noticeable from the very first round.

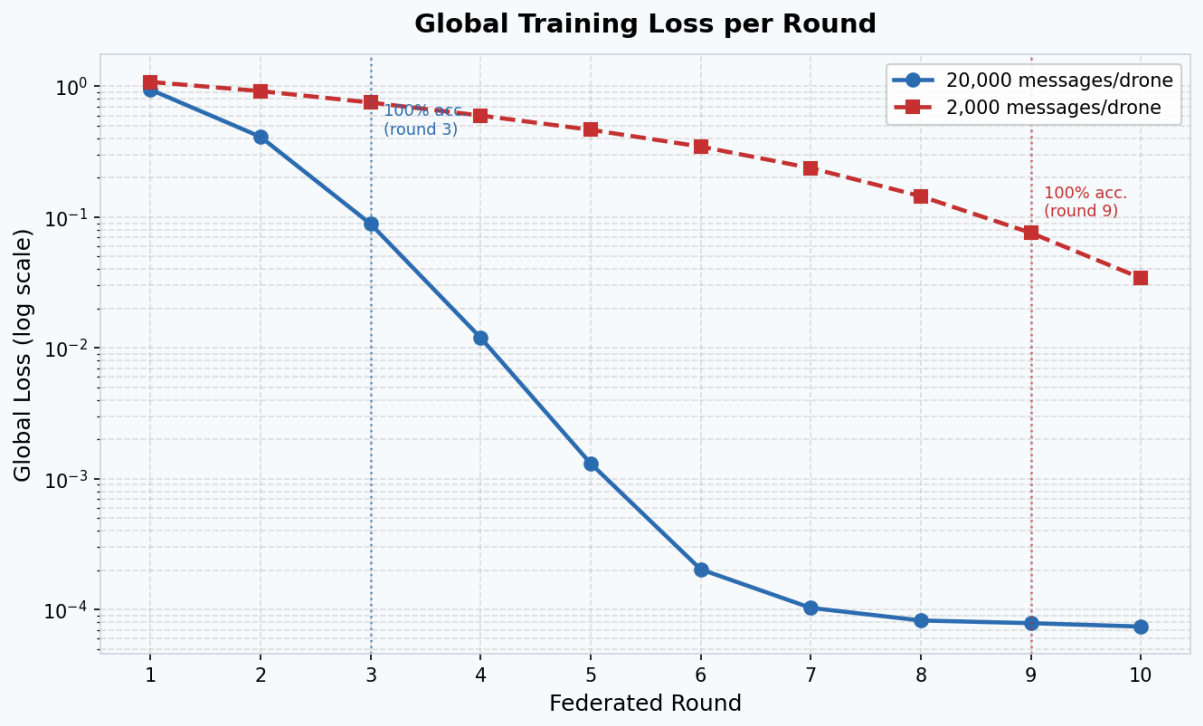

Figure 4.1: Global training loss per federated round (logarithmic scale). The 20.000 message configuration (blue) converges dramatically faster than the 2.000 message configuration (red)

In the 20.000 message run, the loss starts at 0.94455 in round 1 and drops all the way to 0.0890 by round 3, a reduction of roughly 90% in just two additional rounds. From round 3 onwards, the model is already at 100% global accuracy and the loss continues declining, reaching 0.000074 by the end of the 10th round. This kind of near-exponential drop is characteristic of a well-initialized federated model with

sufficient data on each client. Because each drone has 16.000 training samples, a single local epoch is enough to produce gradients that mode the global model meaningfully in each round.

The 2.000 message run on the other hand, starting from the first round loss at 1.0813, which is higher than the 20k counterpart, and it decreases slowly though rounds 2 to 8 following a roughly linear movement rather than an exponential one. The model does eventually converge, reaching 100% accuracy at the end of the 9th round and closing at a loss of 0.0342 in round 10. That final value is about 462 times larger than the 20k run's final loss. This is not a failure of the federated learning process itself because both runs present outstanding results, therefore it is a natural consequence of each client contributing updates from a much smaller sample, which results in noisier gradient estimates and slower movement per round.

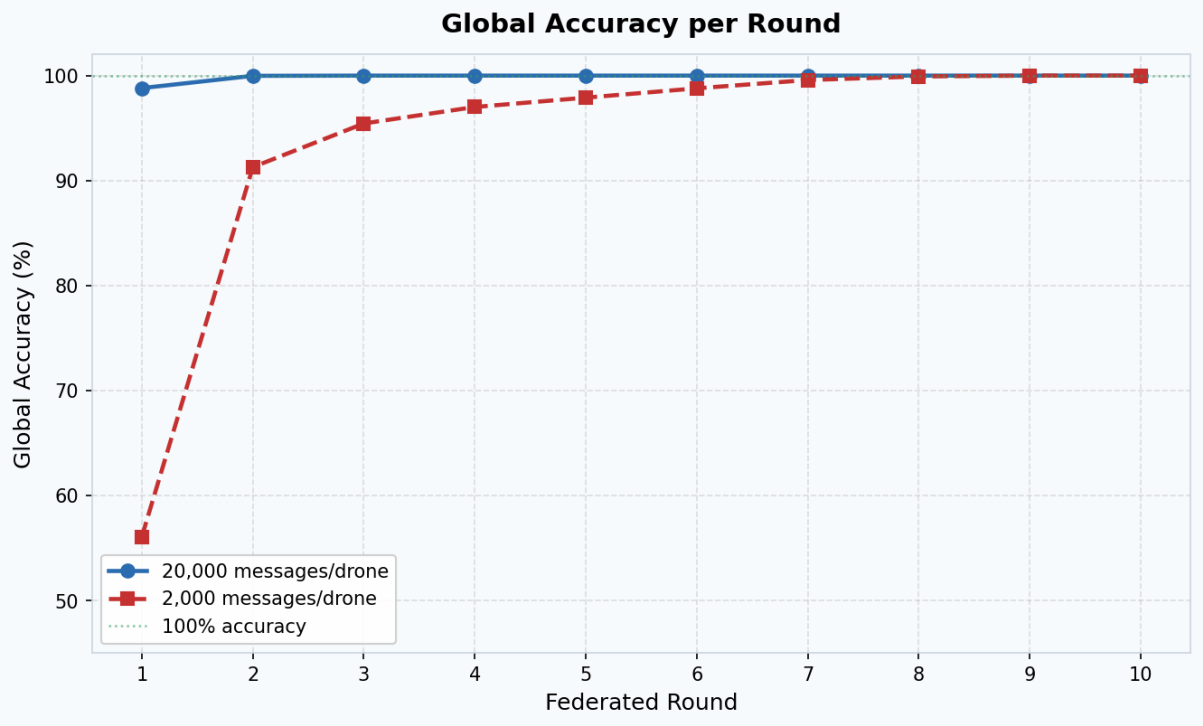

Figure 4.2: Global accuracy per federated round. The 20.000 message model reaches 100% at round 3, while the smaller 2.000 batch model requires until round 9 to reach peak accuracy.

The figure above shows the accuracy progression more clearly. The 20k model's accuracy jumps from 98.81% in round 1 to 99.98% in round 2 showing that with only two rounds the model is extremely accurate, therefore in round 3 onwards the percentage reaches 100%. The 2k model starts at only 56.07% in round 1, which is barely better than random for a two-class problem, whereas in round 2 climbs gradually at 91.28% and 95.44% at round 3 while reaching 99.58% only by round 7. This slow climb reflects the fact that with so few samples, the local models in early rounds have not learned to separate the two classes reliably and the

aggregated global model inherits this uncertainty. Table 4.1 provides the complete per-round figures for both configurations.

Per-Round Training Metrics — Both Configurations

Round	Loss (20k)	Accuracy (20k)	Loss (2k)	Accuracy (2k)
1	0.945500	98.81%	1.081300	56.07%
2	0.412200	99.98%	0.920300	91.28%
3	0.089000	100.00%	0.752600	95.44%
4	0.012000	100.00%	0.598700	97.01%
5	0.001310	100.00%	0.465500	97.90%
6	0.000203	100.00%	0.346400	98.79%
7	0.000103	100.00%	0.237500	99.58%
8	0.000082	100.00%	0.144600	99.92%
9	0.000079	100.00%	0.075800	100.00%
10	0.000074	100.00%	0.034200	100.00%

20,000 messages/drone 2,000 messages/drone Both reached 100% accuracy

Table 4.3: Complete per-round training loss and accuracy for both configurations. Green rows indicate rounds where both models had reached 100% accuracy.

One observation worth marking here is that the 20k run required 130.94 seconds to complete the full 10 round training, while the 2k run finished in 58.65 seconds. This is a direct consequence of the larger datasets while having more batches per local epoch means more computation per round, even though the number of rounds is the same. The trade-off between training time and final model quality is therefore real and the choice of when to start training has measurable consequences on both ends.

4.3: Intrusion Detection Performance

After the completion of the training process, the best saved model from each run is evaluated using the intrusion prediction pipeline. This pipeline applies 16 forward passes with small Gaussian noise perturbations on each input sample, averages the resulting class probabilities to obtain a stable classifier probability, computes a Mahalanobis bases anomaly score from the raw feature values and in the end combines the two using the weighted formula with alpha set to 0.75. A sample is classified as an intrusion when the combined score exceeds the value of 0.50.

The evaluation is run on a set of 1.000 synthesized test samples, of which 98 are labeled intrusions and 902 are labeled normal. Figure 4.3 demonstrated the confusion matrices for both runs side by side.

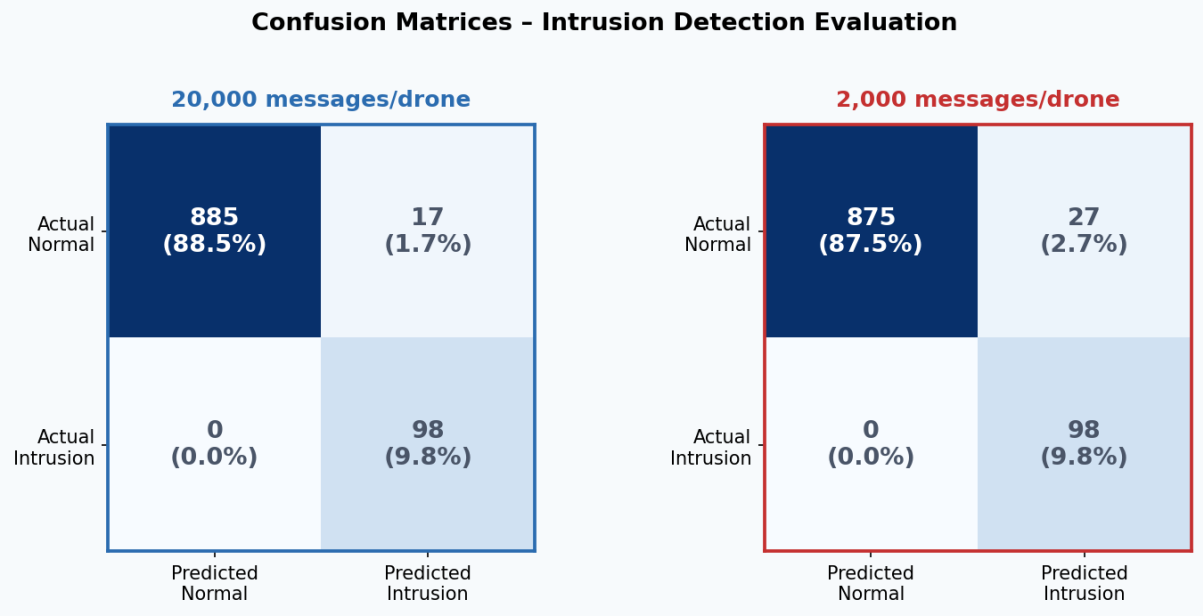

Figure 4.4: Confusion matrices for the 20.000 message model (left) and the 2.000 message model (right). Both models archive zero false negatives, meaning no intrusion was missed.

Both models correctly detected all 98 intrusion samples, giving a recall score of 1.0000 in both cases. Missing a real attack is considerably more dangerous than raising a false alarm, as it is the most critical property for an intrusion detection system. The difference between the two runs appears entirely in the false positives. The 20k model incorrectly flags 17 normal samples as intrusions, while the 2k model flags 27. This translates to precision values of 0.8522 and 0.7840 respectively and F1 scores of 0.9202 versus 0.8789.

The fixed samples used throughout this simulation normal test, with values of temperature 25.0, humidity 55.0, pressure 1000.0 and vibration 1.2, produced combined scores of 0.0806 for the 20k model and 0.1018 for the 2k model. Both are well below the 0.50 threshold. The fixed intrusion samples with temperature value of 90.0, humidity 5.0, pressure 600.0 and lastly vibration of 12.0, produced combined scores of 0.9645 and 0.9854 respectively, both comfortably above the threshold, as a result of the models to agree clearly on samples that sit fat from the decision boundary

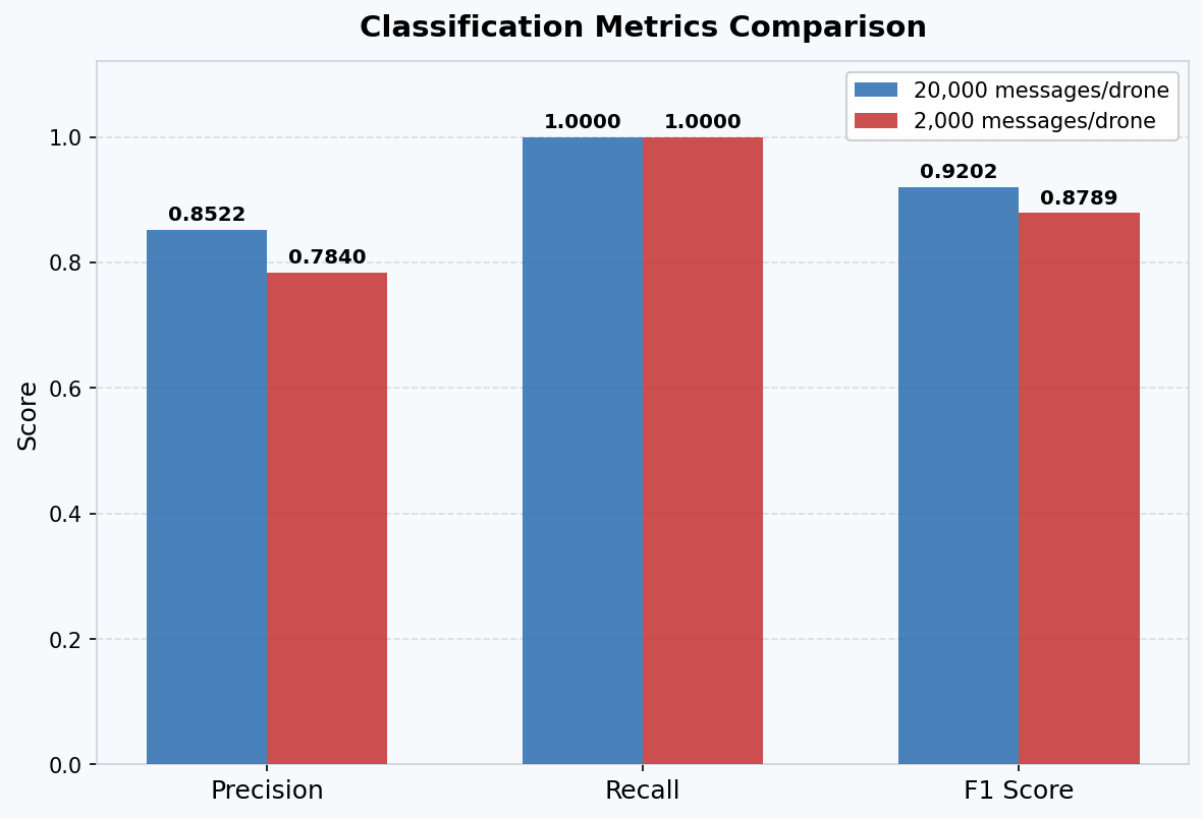

Figure 4.5: Precision, recall and F1 score for both configurations. Recall is identical at 1.0000 while precision and F1 are higher for the 20.000 message model

Figure 4.5 makes the metric comparison even easier to read. The difference in precision of 0.8522 value versus 0.7840 is a gap of about 8.6 percentage points in favor of the 20k model. For an intrusion detection scenario, this means the 20k model raises fewer unnecessary alarms while still catching every actual intrusion. The additional false positives from the 2k model could in practice trigger unnecessary responses and increase operator workload, even though the underlying threat detection is equally complete, so that is something that needs to be fixed in future work

4.4: Decision Boundary Analysis

To understand where exactly the decision boundary sits for each model, the evaluation script runs a boundary sweep that linearly interpolates all four sensor features between a fixed normal reading and another fixed intrusion reading. The interpolation parameter t runs from values on 0.30 to 0.49 in steps of 0.01 where t = 0 would be fully normal and t = 1 would be fully anomalous. The combined score id computed for each blend and the first point where the score exceeds 0.50 defined the effective trigger point.

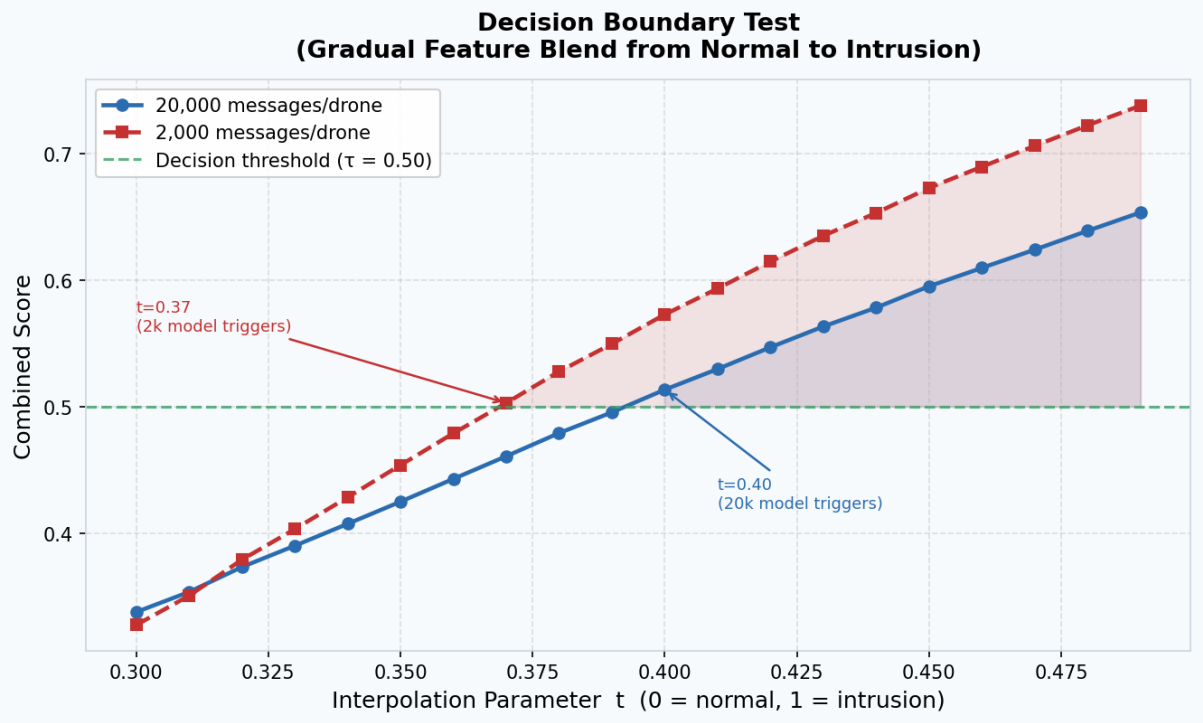

Figure 4.6: Decision boundary test. The combined score is plotted against the interpolation parameter t for both models. The 20.000 message model (in blue) triggers at = 0.40 while the 2.000 model (red) triggers earlier at t = 0.37.

The above figure demonstrated the results of this sweep. The 2k model crosses the 0.50 threshold at t = 0.37, where the blended features correspond to a temperature of around 32.4, humidity near 79.0, pressure around 889.0 and vibration near 5.1. The 20k model does not trigger until t = 0.40 where the values are those of temperature 33.0, humidity at 81.0, pressure 880.0 and a vibration value at 5.3. The 20k model therefore requires a more extreme deviation from normal conditions before raising an alarm. This behavioral difference is consistent with the confusion matrix results. A model that is more conservative about calling something an intrusion will naturally produce fewer false positives on normal samples that happen to fall near the boundary region. At the same time, both models still catch every actual intrusion sample in the evaluation set, that is a result on the true intrusion samples that sit far from the boundary, in a region where both models agree. The boundary difference only matters for ambiguous inputs that lie somewhere between clearly normal and clearly anomalous data.

One of the random test samples illustrates this point directly. Sample number 9, with raw values of temperature 29.48, humidity 63.79, pressure 820.75, and vibration 6.15, has a true label of 0 meaning normal data, even though it is classified as an intrusion by both models. The 3l model gives it a combined score of 0.5359 and the other model gives 0.5178. Taking a look at the feature values, the pressure of 820.75 is below the normal range of 900 to 1100 and the vibration of 6.15 is slightly above the normal ceiling of 5.0. So, while the ground truth label is normal, the sensor readings are genuinely ambiguous. The Mahalanobis feature score reflects this by returning a value of 0.3473 for this sample. Neither model is

clearly wrong to flag it, and it highlights that the threshold of 0.50 may need calibration on a leveled validation set if false positive reduction is a priority

4.5: Configuration Comparison and Trade-offs

The following table brings together the key metrics from both configurations into a single summary, with green highlighting used to mark the stringer result in each row.

Final Evaluation Summary — Both Configurations

Metric	20,000 messages/drone	2,000 messages/drone
Training duration	130.94 seconds	58.65 seconds
First round at 100% accuracy	Round 3	Round 9
Final global loss (round 10)	0.000074	0.034164
True Positives (TP)	98	98
False Positives (FP)	17	27
False Negatives (FN)	0	0
True Negatives (TN)	885	875
Precision	0.8522	0.7840
Recall	1.0000	1.0000
F1 Score	0.9202	0.8789
Boundary trigger point	t = 0.40	t = 0.37

Better result Weaker result

Table 4.7: Final evaluation summary comparing both configurations. Green cells indicate the better results for each metric, while red cells indicate weaker results.

The data in the table above shows a consistent pattern with the 20.000 message configuration to wins on nearly every quality metric, while the 2.000 message configuration wins on training speed. The 20k model converges to 100% accuracy six rounds earlier, finishes with a loss that is 460 times smaller, produces ten fewer false positives, and achieves a higher F1 score. The 2k model on the other hand, completes training in 58.65 seconds compared to 130.94 which is a meaningful difference in a real deployment where network conditions or computational resources may be constrained.

Whether this speed advantage justifies the quality reduction depends on the application context. For a drone swarm operating in an environment where attacks are frequent and detection latency is critical, starting the training process earlier may be acceptable because the model still achieves perfect recall even with limited data. The federation process ultimately brings the 2k model to 100% accuracy by round 9, so the quality gap disappears for the training accuracy metric. The gap that remains is in the false positive rate and this is where the additional data in the 20k configuration provides the clearest benefit. The better calibrated decision boundary as illustrated in Figure 4.6, is a direct product of richer local training data, which allows each drone to develop a more precise

understanding of what constitutes a normal operating state before contributing to the federated update. Both configurations demonstrate that the federated learning approach is fundamentally sound for this application. No communication of raw sensor data is required between drones, and yet a globally shared model that detects intrusions with perfect recall emerges from the aggregation process in both cases. The FedAvg strategy, combined with local early stopping and class-weighted loss, handles the data imbalance well enough that the system does not collapse into predicting only the majority class, which is a common failure in intrusion detection with skewed datasets.

The results also confirm that the combination of the neural network classifier with the Mahalanobis anomaly score feature adds value beyond the classifier alone. On the fixed normal sample, the classifier probability for the 20k model is 0.0950, which is already well below the threshold, but the feature score of 0.0374 pulls the combined score even lower to 0.0806. On ambiguous samples near the boundary, the feature score acts as a second opinion that can either reinforce or moderate the classifier's output. Tuning the alpha weighting between the two components, currently fixed at 0.75, would be a logical next step toward reducing the residual false positive count.

4.6: Comparison with the Centralized Intrusion Detection Approach

This section compares the system developed in this thesis against a centralized intrusion detection system built on the same Kafka based sensor data infrastructure. The introduced system in this work, referred to here as the centralized approach, was implemented by a student and serves as a direct reference point because the two works share the same data generation logic and the same general application domain. Both systems are designed to detect anomalous behavior in drone sensor readings, and both use Apache Kafka as the data transport layer running inside a Docker composed environment with ZooKeeper for coordination. Beyond these shared foundations however, the architectural decisions, the machine learning methods and the security of both systems diverge significantly.

4.6.1: Overview of the two architectures

The centralized system follows a straightforward pipeline. A single Kafka producer generates sensor readings and publishes them to one shared topic. Then a single consumer node reads every message, buffers a large enough batch for the training process and then fits three unsupervised models in sequence and starts classifying incoming records one by one. Everything happens on one machine. The raw sensor payload that travels across the broker includes seven fields per record.

These are a Unix timestamp, the four physical readings and an engine check flag that works as an intrusion flag and finally an idle time value. Any node that can subscribe to the Kafka topic can read the full contents of every message. The federated system takes the opposite structural approach. Five drone containers each, run a local Kafka consumer and collect sensor data independently. The raw data never moves beyond the container that produced it. Each drone trains a local neural network on its own buffered data and then shares only the resulting weight matrices with the Flower aggregation server. The server applies weighted averaging across the five submissions and redistributes a single global model that incorporates what all five drones have learned. From there, each drone uses the updated global weights as the next training round while the raw readings never leave the drone throughout the whole process.

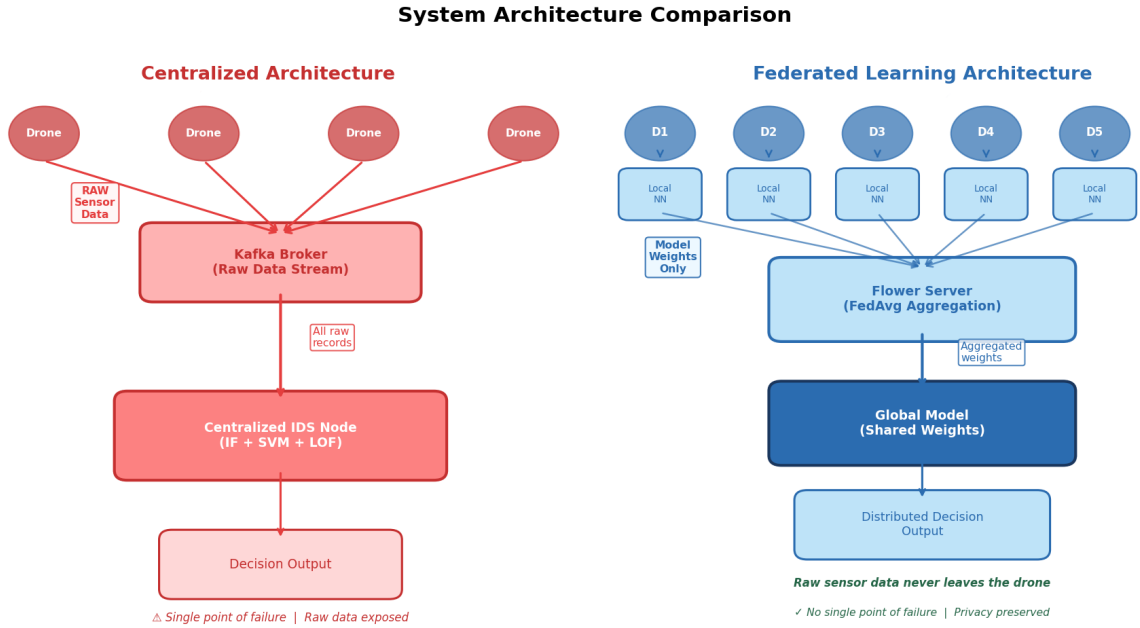

Table 4.8: Architectural comparison between the centralized system (left on red color) and the federated system (right on blue color). In the centralized design, raw sensor data flows to a single processing node. In the federated design, only model weights travel over the network.

The figure presented above illustrates this structural contrast. The left side of the graphic shows the data path in the centralized system, where arrows from each drone converge on a single Kafka broker that then feeds a central IDS node. The right side of the graph shows the federated round cycle, where each drone hosts its own local model and the only thing that crosses the network boundary is the trained weight tensor.

4.6.2: Security and Data Privacy

The most consequential difference between the two systems is what gets transmitted over the network. In the centralized design every sensor reading from every drone is serialized to JSON and pushed through Kafka as a raw record. This includes the engine check status, the idle time and all four previously mentioned physical sensor measurements. A possible attacker who can intercept or subscribe to the Kafka topic which in a plaintext listener configuration requires no credentials, gains complete visibility into the operational state of every drone in the fleet in real time. Beyond passive spying, this architecture also keeps the entire dataset on one done. So that a single successful compromise of that node gives an attacker access to every record that has ever been collected.

On the other hand, the federated architecture removes this exposure almost entirely at the data level because the raw readings are only ever held in the memory of the drone container that produced them whereas there is no Kafka message containing sensor values that a third party could intercept. What travels over the network are only the neural network weight matrices, which are floating point tensors with no direct semantic connection to individual sensor readings. Reconstructing the original measurements from the shared weights is a model inversion problem that requires both the model architecture and training data access and even then, it produces approximate statistics rather than exact records. The privacy improvements are therefore structural and does not depend on encryption, access controls or any additional security layer being configured correctly.

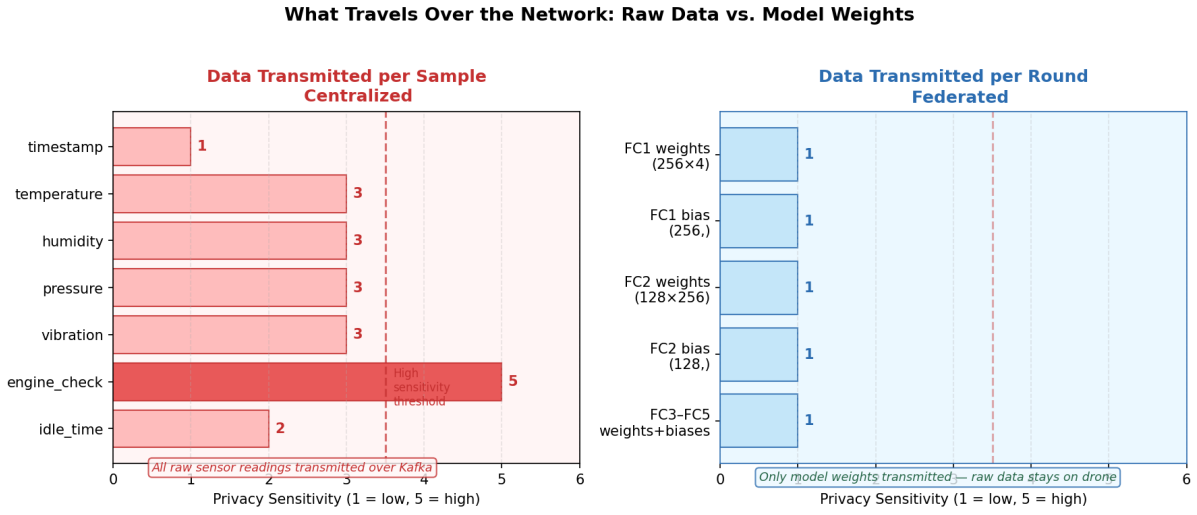

Figure 4.9: Comparison of what is transmitted over the network in each system. The centralized system (the one with the red color) sends all seven raw fields per record, including sensitive operational flags. The federated system (in blue color) sends only model weight tensors, which carry no directly interpretable sensor values.

The above figure visualizes the difference mentioned. On the left graph, each of the seven fields in the centralized producer's JSON payload is rated for privacy sensitivity. The engine check flag which explicitly signals a mechanical fault condition receives the highest sensitivity score because its exposure could allow an adversary to identify and target a compromised drone. On the right, the federated system's transmitted objects are the neural network layer weights each rated at the lowest sensitivity level because they contain no directly readable operational data. This is the fundamental privacy argument for federated learning in this context, which it the information that needs to be protected is sensor data that never leave its birthplace.

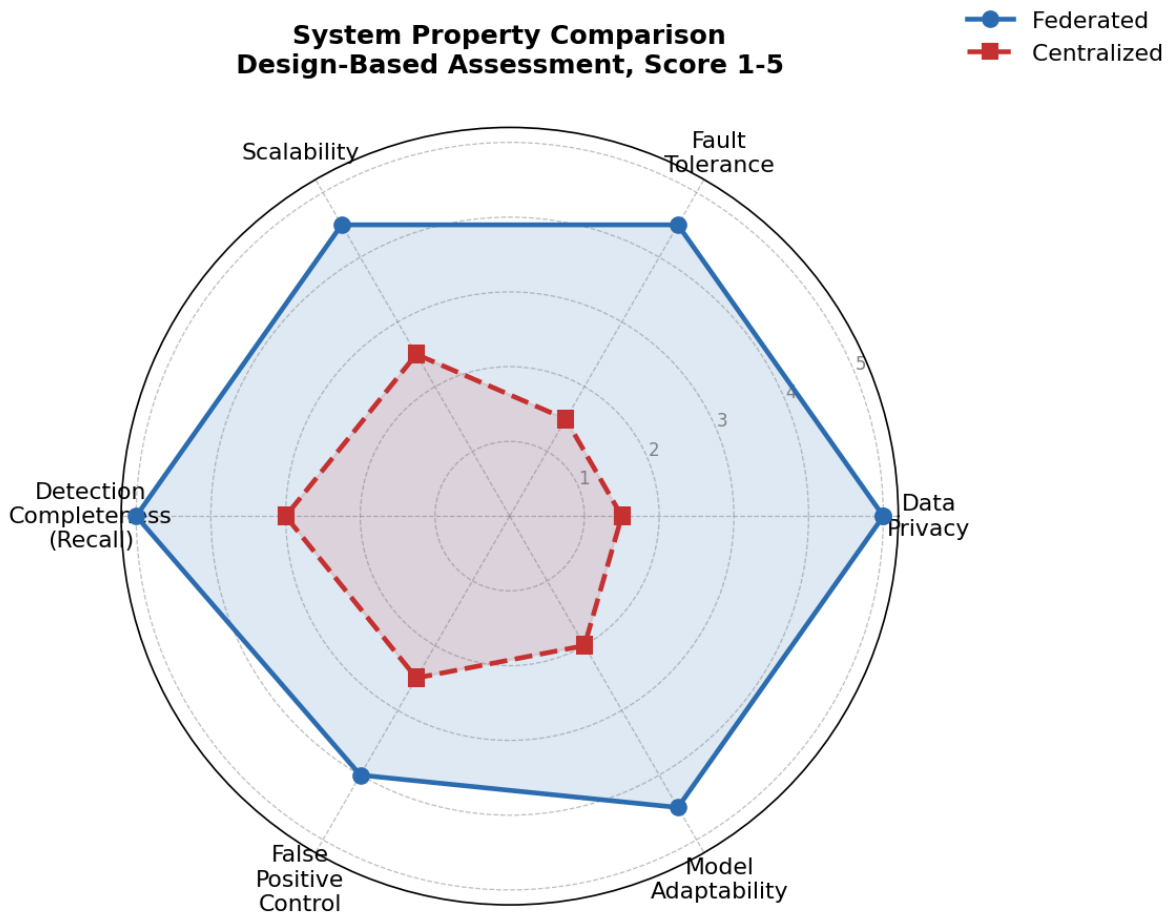

Figure 4.10: Radar chart comparing the two systems across six design properties. Scores are based on architectural analysis of both implementations. The federated system (blue) outperforms the centralized system (red) on every security-relevant dimension.

This graph extends the comparison to six design properties scored on a scale from 1 to 5. The most visually difference is in the data privacy score, where the centralized system scores close to 1.5 because raw data is continuously broadcast over Kafka, while the federated system scores 5.0 because of the non-transmitted data. Fault tolerance is the second largest gap. In the centralized design, if the

single IDS node fails or is taken offline, the detection capability of the entire network disappears immediately. In the federated design, the Flower server plays a coordinating role but the local models on each drone continue to function; removing the server mid-deployment stops further aggregation but does not disable detection on the drones that already hold the latest global model. Scalability also favors the federated approach, as adding a new drone in the centralized system means routing its entire data stream through the broker to the central node, increasing bandwidth and processing load linearly, whereas a new federated client simply joins the aggregation pool and contributes weight updates without increasing the data volume at the server.

4.6.2: Machine Learning Methodology

The two systems also differ fundamentally in how they approach the detection problem from a machine learning perspective. The centralized approached system uses three unsupervised algorithms, being Isolation Forest, One-Class SVM and a Local Outlier Factor. A voting mechanism requires at least two of the three models to agree before an input is classified as an intrusion. This is a reasonable design choice for a scenario where no label training data is available, since all three methods learn the shape of the normal behavior without being told what anomalies look like. The tradeoff is that unsupervised models are inherently approximate. Their decision boundaries depend on the contamination and nu hyperparameters, which have to be guessed rather than learned from labeled examples, and the threshold for what counts as an anomaly is derived from percentile calculations on the training distribution rather than from any ground truth.

The federated approached system uses a supervised five layer fully connected neural network with labeled sensor records where 0 flags normal readings while 1 marks intrusion messages. Because the labels are known during training, the model can learn a much more precise decision boundary than any of the three unsupervised methods. The class weighted cross-entropy loss function explicitly penalizes missed intrusions nine times more heavily than false alarms, which directly shapes the model toward the operating point that matters most in a security context. The results confirmed in the evaluation stage discussed in earlier sections of this chapter, is a recall of 1.0 in both experimental configurations, meaning that the model does not miss any of the intrusion samples in the test set.

Detection Pipeline Comparison

Property	Centralized	Federated
Learning paradigm	Unsupervised	Supervised
Algorithm(s)	IF + SVM + LOF (voting)	5-layer Neural Network
Number of features	5 (incl. idle_time)	4 (sensor readings)
Training labels used	No (anomaly scoring)	Yes (0/1 class labels)
Training location	Central node only	Each drone independently
Model update frequency	Once (batch)	Every federated round
Data seen by server	All raw records	None (weights only)
Supports new drones	Requires retraining	New client joins pool
Failure impact	Full system down	Remaining clients continue

Federated Learning Round Cycle

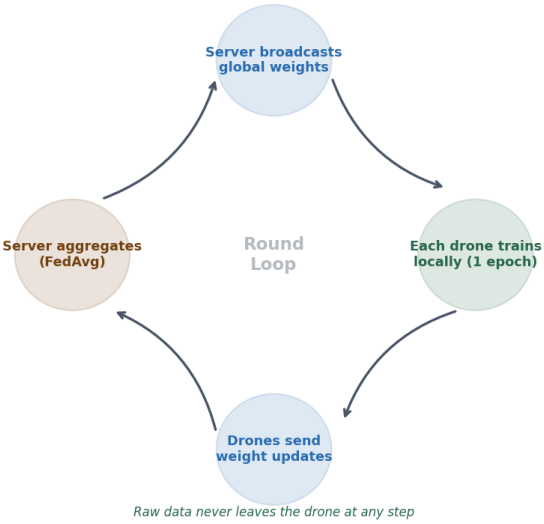

Figure 4.11: On the left we're seeing a pipeline property comparison table between the two systems. On the right side the federated learning round cycle, illustrating how model updates circulate without any raw data leaving the drones.

The figure above summarizes the methodological differences in a side by side table. A few entries are worth discussing in more detail. The centralized system uses five features per sample because it includes the idle time field from the Kafka producer, while the federated system uses only four physical sensor readings and drops idle time for convenient. This difference exists because the federated system was designed around what the sensor hardware would realistically measure during flight, without relying on application layer flags. The engine check flag used as a ground truth label in the centralized system is also worth noting. This is a Boolean value set by the producer logic itself, so the system is effectively learning to detect the pattern that the producer defines an anomalous rather than independently discovering it. The federated system uses a separate binary label field that is set at data generation time and never included in the feature vector, keeping the detection signal clean. Another difference is lying on the training frequency. The centralizes system trains all three models once on a fixed batch of records, after which the models do not update as new data arrives. The federated system retrains the local model weights in every round which means that the global model can adapt as the federation sees more data over time. In a real deployment where the nature of attacks evolves periodic retraining is considerably more valuable than a one-time fit, and the federated approach supports this naturally through its round based structure.

4.6.2: Summary of the two compared architectures

The two systems solve the same surface problem, which is detecting anomalies in drone sensor streams, but they represent meaningfully different points on the

spectrum between simplicity and security. The centralized approach is straightforward to reason about and straightforward to deploy as it contains one producer, one topic, one consumer and three off the shelf anomaly detectors. It works well enough as a prototype and demonstrates that the Kafka pipeline can carry sensor data reliably. Its weakness is that it was not designed with confidentiality as a requirement. The raw sensor data, including operational status flags that reveal the health of each drone, flows freely though the broker and accumulates on a single node.

The federated system on the other hand, accepts more architectural complexity in exchange for a fundamentally different security posture. Be keeping raw data on each drone and transmitting only model weights, it eliminates the most obvious attack surface in the centralized design without requiring any encryption layer or access control policy. The de The detection quality also improves: the supervised neural network, combined with the Mahalanobis anomaly score and the test-time augmentation pipeline, produces better-calibrated decisions than the voting ensemble, and the class-weighted training objective directly optimizes for zero missed intrusions. The cost of this is a more complex deployment with five separate clients, a Flower server, and a multi-round training protocol, none of which exist in the centralized setup.
Whether that added complexity is justified depends on the deployment context. For a research prototype running on a local machine, the centralized approach is faster to get running. For a system intended to operate on real hardware across an untrusted network, where the sensor readings might reveal commercially or operationally sensitive information about the drones, the federated approach provides a security guarantee that the centralized design structurally cannot offer.

CHAPTER 5

Conclusion and Future Works

6.1: Conclusion

This thesis set out to investigate whether federated learning could serve as a practical and privacy-preserving approach to intrusion detection in a multi-drone network. The proposed system was built around five simulated drone nodes, each running a local neural network trained on sensor data streamed through Apache Kafka. The Flower framework handled the federated aggregation using a weighted averaging strategy, and the full pipeline ran inside a Docker Compose environment that kept each drone isolated from the others at the operating system level. Two experimental configurations were tested where in the first one each drone waited for the full 20.000 message dataset before continuing on with the training procedure, while the other waiting for only 2.000 messages and consumed the remaining while the system continued on the training stage for time efficiency. Both configurations eventually reached to 100% global accuracy, which in itself confirms that the federated aggregation proves works correctly even when local datasets are small.

The most interesting breakthrough came from the evaluation phase. Both models achieved perfect recall, meaning not a single intrusion event was missed in either run. The 20.000 message model produced fewer false positives compared to the 2.000 run while scoring 17 compared to 27 respectively while also reaching 100% accuracy sic federated rounds earlier. This confirms the intuitive expectation that richer local training data produces a better calibrated model, but it also shows that the federated approach does not strictly require large per-client datasets to be useful. A drone that starts training with limited data and participates in the federation can still contribute and benefit from the aggregated global model. The comparison with the centralized intrusion detection system, which used the same Kafka infrastructure but transmitted raw sensor records to a single processing node, reinforced the privacy argument for federation. The centralized approach exposes every sensor reading, including operational status flags, to any node that can subscribe to the broker topic. The federated system transmits only floating-point weight tensors, which carry no directly interpretable sensor values. This is a structural privacy improvement that does not depend on any encryption or access control policy being configured correctly, and it is arguably more robust because it removes the sensitive data from the network path entirely rather than trying to protect it in transit.

Taken together, the results suggests that federated learning is a genuinely viable alternative to centralized processing for this kind of application. The detection quality is at least as good, the privacy properties are substantially better, and the architecture is more resilient to single-node failures. The added complexity of the federated setup, specifically the multi-client orchestration and the multi-round training protocol, is real but manageable and it comes with benefits that are difficult to achieve any other way.

6.1: Future Works

Several directions stand out as natural extensions of this presented work, ranging from changes to the machine learning core to broader questions about real-world deployment.

The most significant methodological limitation of the current system is that it relies on labeled training data. Each sensor record is presented with a label that is tagged as normal or intrusion data at the generation time by the Kafka producer, making the entire system work as a supervised machine learning through the neural network. In a real deployment, producing accurate labels at scale is expensive and sometimes impossible, particularly for a novel attack pattern that have not been seen before. Replacing the supervised classifier with an unsupervised or self-supervised approach would remove this dependency entirely. Federated versions of anomaly detection methods, such as federated autoencoders or federated one-class classification, could learn the distribution of normal sensor behavior without ever seeing a labeled intrusion, making the system capable of detecting attack types that were not anticipated at training time. This transition from supervised to unsupervised learning within the federated framework is probably the most impactful single improvement that could be made to the system. A closely related extension is threshold calibration. The combined score threshold of 0.50 used in the current evaluation pipeline was fixed manually and not tuned against a labeled validation set. The boundary test results in Chapter 4 showed that both models trigger at slightly different interpolation points and the residual false positives in both runs could potentially be reduced by finding the optimal threshold on a held out set. This is a relatively small change, but it would make the intrusion prediction pipeline more principled and easier to justify in a deployment context.

The most obvious gap between this thesis and a real-world system is that all five drones are simulated containers running on the same physical machine. Testing the system on actual embedded hardware, whether small single board computers carried by physical drones or edge computing nodes attached to a ground station, would introduce network latency, hardware heterogeneity and resource constraints that the containerized setup does not capture. In particular, the assumption that all five clients complete their local training and report back to the server within the same federated round becomes harder to guarantee when clients run on different hardware under real network condition. Asynchronous

federated learning protocols, where the server aggregates updates as they arrive rather than waiting for all clients, would be worth exploring in that setting.

Extending the system to handle a larger and more diverse fleet is another worthwhile direction. The current setup uses five homogeneous drones all producing data from the same distribution. A more realistic scenario would involve drones with different sensor suites, flying different missions and potentially experiencing very different baseline readings. Personalized federated learning approaches, where the global model is fine-tuned locally for each drone rather than applied identically, could improve drone performance in this kind of heterogeneous setting without sacrificing the privacy benefits of the federated architecture.

Finally, the security of the federated protocol itself deserves attention. The current implementation uses an unsecured gRPC connection between the Flower clients and the server, which is acceptable for a research prototype but in real life applications, it would need to be hardened before deployment. Beyond transport security, the aggregation process itself is potentially vulnerable to poisoning attacks, where a compromised drone submits manipulated weigh updates designed to degrade the global model's performance on specific inputs. Robust aggregation strategies that can detect and exclude outlier updates, such as coordinate-wise median or Krum algorithm, would strengthen the system against this class of threat and would complement the data privacy guarantees that federated learning already provides.

CHAPTER 6

6.1: Kafka Producer Python Script

```
from confluent_kafka import Producer
import logging
import random
import json
import time
import os

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

KAFKA_SERVER = os.environ.get("KAFKA_SERVER", "kafka:9092")
KAFKA_TOPIC = os.environ.get("KAFKA_TOPIC", "sensor_data_drone1")
DRONE_ID = os.environ.get("DRONE_ID", "drone-1")

producer = Producer({
    'bootstrap.servers': KAFKA_SERVER,
    'acks': 'all',  # Ensure full acknowledgment
    'batch.num.messages': 10000, # -
    'linger.ms': 0.1,
    'compression.codec': 'gzip',
})

def generate_sensor_data(anomaly=False):

    # Generate normal or anomalous sensor data
    if anomaly:
        return {
            "timestamp": int(time.time()),  # Add a timestamp
            "temperature": round(random.uniform(40.0, 50.0), 2), # Extreme high temperature
            "humidity": round(random.uniform(90.0, 150.0), 2), # Abnormally low humidity
            "pressure": round(random.uniform(600.0, 800.0), 2), # Abnormal pressure drop
            "vibration": round(random.uniform(7.0, 12.0), 2), # High vibration/possible
mechanical failure
            "label": 1, # Force an anomaly
        }
    else:
        return {
            "timestamp": int(time.time()),  # Add a timestamp
            "temperature": round(random.uniform(15.0, 35.0), 2),
            "humidity": round(random.uniform(30.0, 80.0), 2),
            "pressure": round(random.uniform(900.0, 1100.0), 2),
            "vibration": round(random.uniform(0.0, 5.0), 2),
            "label": 0,
        }

def delivery_report(err, msg):
    """Callback for delivery reports."""
    if err is not None:
        logging.error(f"Message delivery failed: {err}")

def produce_messages(num_messages=20000, num_intrusions=2000
```

```
    # Produce sensor data messages with exactly num_intrusions randomly distributed
    intrusion_indices = set(random.sample(range(num_messages), num_intrusions))
    logging.info(f"Intrusion indices selected: {sorted(list(intrusion_indices))[:10]}...")

    count = 0
    intrusion_count = 0

    while count < num_messages:
        anomaly = count in intrusion_indices
        data = generate_sensor_data(anomaly=anomaly)
        data["drone_id"] = DRONE_ID

        if anomaly:
            intrusion_count += 1
        try:
            producer.produce(KAFKA_TOPIC, json.dumps(data), callback=delivery_report)
            producer.poll(0)  # Ensure message is processed
        except Exception as kafka_error:
            logging.error(f"Kafka produce error: {kafka_error}")
            break
        count += 1

        if count % 5000 == 0:
            logging.info(f"Produced {count} messages...")
            producer.flush()  # Flush every 5000 messages

        time.sleep(0.001)  # Avoid overloading Kafka

    producer.flush()
    logging.info(f"Finished producing {num_messages} messages.")
    logging.info(f"Total intrusions produced: {intrusion_count}")

if __name__ == "__main__":
    logging.info(f"Producing messages to topic: {KAFKA_TOPIC} (DRONE_ID={DRONE_ID})")
    try:
        produce_messages(num_messages=20000, num_intrusions=2000)
    except Exception as e:
        logging.error(f"Error in producer: {e}")
```

6.2: Client Python Script

```
from confluent_kafka import Consumer, KafkaError, KafkaException
from torch.utils.data import TensorDataset, DataLoader
from collections import OrderedDict
from flwr.client import NumPyClient
import torch.nn.functional as F
from typing import List
import torch.nn as nn
import numpy as np
import threading
import logging
import socket
import torch
import flwr
import time
import json
import os

CLIENT_ID = int(os.environ.get("CLIENT_ID", 0))
DEVICE = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
print(f"Starting Training on {DEVICE}")
print(f"Flower {flwr.__version__} / PyTorch {torch.__version__}")

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Kafka Configuration
KAFKA_SERVER = os.environ.get("KAFKA_SERVER", "kafka:9092")
KAFKA_TOPIC = os.environ.get("KAFKA_TOPIC", "sensor_data_drone1")
BATCH_SIZE = 256
NORMALIZE_FEATURES = True

GLOBAL_MEAN = np.array([27.041304, 61.775265, 969.56335,  3.2226298], dtype=np.float32)
GLOBAL_STD  = np.array([8.085047,  24.433617, 105.70269,  2.5380642], dtype=np.float32)

# Consuming Messages
class KafkaConsumerApp:
    def __init__(self, kafka_topic, kafka_server):
        CLIENT_GROUP = os.environ.get("KAFKA_GROUP", "sensor_data_group")

        self.kafka_consumer = Consumer({
            'bootstrap.servers': kafka_server,
            'group.id': CLIENT_GROUP,
            'auto.offset.reset': 'earliest'
        })
        self.kafka_consumer.subscribe([kafka_topic])
        self.data_buffer = []

    def consume_data(self):
        logging.info("Starting to consume messages...")
        message_count = 0
        max_messages = 20000
        try:
            while message_count < max_messages:
                msg = self.kafka_consumer.poll(timeout=1.0)  # Poll for messages
                if msg is None:
                    continue
                if msg.error():
                    if msg.error().code() == KafkaError._PARTITION_EOF:
                        continue
                    else:
                        raise KafkaException(msg.error())
```

```
                collected_data = msg.value().decode('utf-8')
                data_dict = json.loads(collected_data)
                self.data_buffer.append(data_dict)  # Store messages in the buffer
                message_count += 1
                if message_count % 5000 == 0:
                    logging.info(f"Client {CLIENT_ID} received {message_count}
messages...")

        except KeyboardInterrupt:
            logging.info("Stopping Consumer...")
        finally:
            self.kafka_consumer.close()

def load_datasets_from_kafka(data_buffer, batch_size=BATCH_SIZE):

# Extracting the 4 features from each message
# Convert Kafka bugger into PyTorch DataLoaders for training and validation

    features = []
    labels = []

    for item in data_buffer:
        feat = [item["temperature"],
                item["humidity"],
                item["pressure"],
                item["vibration"],
        ]   # Numeric features
        features.append(feat)
        labels.append(item["label"]) # Supervised label

    features = np.array(features, dtype=np.float32)
    labels = np.array(labels, dtype=np.int64) # Ensure labels are integers

    if NORMALIZE_FEATURES:
        print(f"[Client {CLIENT_ID}] Normalization ON (GLOBAL stats)")
        features = (features - GLOBAL_MEAN) / (GLOBAL_STD + 1e-8) # Using standard
normalization for same scale features

        print(f"[Client {CLIENT_ID}] Feature mean after: {features.mean(axis=0)}")
        print(f"[Client {CLIENT_ID}] Feature std  after: {features.std(axis=0)}")
    else:
        print(f"[Client {CLIENT_ID}] Normalization OFF")

    x = torch.tensor(features, dtype=torch.float32)
    y = torch.tensor(labels, dtype=torch.long)
    dataset = TensorDataset(x, y)

    split_idx = int(0.8 * len(dataset))
    train_dataset, val_dataset = torch.utils.data.random_split(
        dataset, [split_idx, len(dataset) - split_idx]
    )
    trainloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    valloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)

    print(f"[Client] Number of training samples: {len(train_dataset)}") # Debug: Training
samples
    print(f"[Client] Number of validation samples: {len(val_dataset)}") # Debug: Validation
samples
    return trainloader, valloader

# Net Function
class Net(nn.Module):   # Fivr layer fully connected neural
    def __init__(self, input_dim=4, num_classes=2):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_dim, 256) # Increased number of neurons
```

```
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 32)
        self.fc5 = nn.Linear(32, num_classes)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = self.fc5(x)
        return x

# Parameters Function
def get_parameters(net) -> List[np.ndarray

    # Converts the models parameters from PyTorch tensors → NumPy arrays
    params = [val.cpu().numpy() for _, val in net.state_dict().items()]

    print(f"[Client] Sending parameters to server: {[p.shape for p in params]}") # Debug:
Parameter shapes
    return params

def set_parameters(net, parameters: List[np.ndarray]):
    """ Converts the received NumPy arrays from the server → PyTorch tensors. This lets
your local model update its weights before continuing training. """
    print(f"[Client] Received parameters from server: {[p.shape for p in parameters]}") #
Debug: Parameter shapes

    params_dict = zip(net.state_dict().keys(), parameters)
    state_dict = OrderedDict({k: torch.tensor(v) for k, v in params_dict})
    net.load_state_dict(state_dict, strict=True)

# Training and Testing function
def train_local(net, trainloader, valloader, epochs, lr, weight_decay, patience, device):

    # Train the network on the training set with early stopping
    net.to(device)
    class_weights = torch.tensor([1.0, 9.0]).to(device)
    criterion = torch.nn.CrossEntropyLoss(weight=class_weights)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr, weight_decay=weight_decay)

    best_loss = float("inf")
    no_improve = 0
    avg_loss = 0.0

    for epoch in range(epochs):
        net.train()
        epoch_loss = 0.0
        for x, y in trainloader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            loss = criterion(net(x), y)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
        avg_loss = epoch_loss / len(trainloader)

        # Validation for early stopping
        net.eval()
        val_loss = 0.0
        with torch.no_grad():
            for x_val, y_val in valloader:
                x_val, y_val = x_val.to(device), y_val.to(device)
                val_loss += criterion(net(x_val), y_val).item()
        val_loss /= max(1, len(valloader))
```

```
        net.train()
        print(f"[Client] Epoch {epoch} | Train Loss: {avg_loss:.4f} | Val Loss:
{val_loss:.4f}")

        if val_loss < best_loss:
            best_loss = val_loss
            no_improve = 0
            print(f"[Client] Validation loss improved to {best_loss:.4f}")
        else:
            no_improve += 1
            print(f"[Client] No improvement for {no_improve} epoch(s).")
        if no_improve >= patience:
            print(f"[Client] Early stopping at epoch {epoch + 1}.")
            break

    return avg_loss

def test(net, testloader, device):

    # Evaluate the network on the validation set
    net.to(device)
    net.eval()
    class_weights = torch.tensor([1.0, 9.0]).to(device)
    criterion = torch.nn.CrossEntropyLoss(reduction="sum", weight=class_weights)
    total_loss, correct, total = 0.0, 0, 0

    with torch.no_grad():
        for x, y in testloader:
            x, y = x.to(device), y.to(device)
            outputs = net(x)
            total_loss += criterion(outputs, y).item()
            correct += (outputs.argmax(dim=1) == y).sum().item()
            total += y.size(0)

    loss = total_loss / max(1, total)
    accuracy = correct / max(1, total)
    print(f"[Client] Evaluation completed. Loss: {loss:.4f}, Accuracy: {accuracy:.4f}")
    return loss, accuracy

# Flwr Client
# Client App
class FlowerClient(NumPyClient):
    def __init__(self, partition_id, net, trainloader, valloader):
        self.partition_id = partition_id
        self.net = net
        self.trainloader = trainloader
        self.valloader = valloader

        print(f"[Client {self.partition_id}] FlowerClient initialized with partition_id:
{self.partition_id}")  # Debug: Initialization
        print(f"[Client {self.partition_id}] Model structure: {self.net}") # Debug: Model
structure
        print(f"[Client {self.partition_id}] Training loader size:
{len(self.trainloader.dataset)}") # Debug: Training dataset size
        print(f"[Client {self.partition_id}] Validation loader size:
{len(self.valloader.dataset)}") # Debug: Validation dataset size

    def get_parameters(self):
        print(f"[Client {self.partition_id}] get_parameters called") # Debug: Method call
        params = get_parameters(self.net)
        print(f"[Client {self.partition_id}] Parameters sent to server: {[p.shape for p in
params]}") # Debug: Parameter shapes
        return params

    def fit(self, parameters, config):
        print(f"[Client {self.partition_id}] fit() called with config: {config}")
```

```
        set_parameters(self.net, parameters)

        avg_loss = train_local(
            net=self.net,
            trainloader=self.trainloader,
            valloader=self.valloader,
            epochs=config.get("local_epochs", 10),
            lr=config.get("lr", 1e-4),
            weight_decay=1e-4,
            patience=config.get("patience", 3),
            device=DEVICE
        )

        updated_params = get_parameters(self.net)
        return updated_params, len(self.trainloader.dataset), {"train_loss": avg_loss}

    def evaluate(self, parameters, config):
        print(f"[Client {self.partition_id}] evaluate() called")
        set_parameters(self.net, parameters)

        loss, accuracy = test(self.net, self.valloader, DEVICE)
        return float(loss), len(self.valloader.dataset), {"accuracy": float(accuracy)}


# Main Programm
if __name__ == '__main__':
    consumer_app = KafkaConsumerApp(kafka_topic=KAFKA_TOPIC, kafka_server=KAFKA_SERVER)
    threading.Thread(target=consumer_app.consume_data, daemon=True).start()

    # Wait for Kafka messages to arrive
    print("⏳ Waiting for Kafka data...")
    max_wait = 1200  # seconds
    start_time = time.time()

    TARGET = 2000 # This is the message buffer, 20.000 for the other run
    while len(consumer_app.data_buffer) < TARGET:   # wait for at least 2.000 messages
        if time.time() - start_time > max_wait:
            print(f"⚠️ Timeout: Only {len(consumer_app.data_buffer)} messages collected.
\nProceeding anyway...")
            break
        print(f"Waiting... collected {len(consumer_app.data_buffer)} messages")
        time.sleep(1)

    print(f"✅ Collected {len(consumer_app.data_buffer)} messages from Kafka, starting
training...")   Debug: Number of messages collected


    trainloader, valloader = load_datasets_from_kafka(consumer_app.data_buffer)
    net = Net().to(DEVICE)


    print(f"[Client] Model initialized: {net}") # Debug: Model structure
    print(f"[Client] Training loader size: {len(trainloader.dataset)}") # Debug: Training
dataset size
    print(f"[Client] Validation loader size: {len(valloader.dataset)}") # Debug: Validation
dataset size

    client_instance = FlowerClient(
        partition_id=CLIENT_ID,
        net=net,
        trainloader=trainloader,
        valloader=valloader
        )
```

```
    print(f"[Client] FlowerClient initialized with partition_id:
{client_instance.partition_id}") # Debug: Client initialization

    # Start the Flower client
    FLOWER_SERVER = os.environ.get("FLOWER_SERVER", "server:5000")
    MAX_RETRIES = 20
    RETRY_INTERVAL = 5

    def wait_for_server(host, port, retries=MAX_RETRIES, interval=RETRY_INTERVAL):
        for i in range(retries):
            try:
                with socket.create_connection((host, port), timeout=2):
                    print(f"[Client {CLIENT_ID}] Server is ready!")
                    return True
            except (OSError, ConnectionRefusedError):
                print(f"[Client {CLIENT_ID}] Server not ready, retrying
({i+1}/{retries})...")
                time.sleep(interval)
        print(f"[Client {CLIENT_ID}] Server did not become ready after {retries}
attempts.")
        return False

    host, port = FLOWER_SERVER.split(":")
    port = int(port)
    if wait_for_server(host, port):
        flwr.client.start_client(server_address=FLOWER_SERVER,
client=client_instance.to_client())
    else:
        print(f"[Client {CLIENT_ID}] Exiting, cannot connect to server.")
```

6.3: Server Python Script

```
from flwr.server import ServerAppComponents, ServerConfig
from flwr.common import ndarrays_to_parameters, parameters_to_ndarrays
from flwr.server.strategy import FedAvg
from client import Net, get_parameters
from collections import OrderedDict
import torch
import flwr

params = get_parameters(Net())

class FedAvgWithGlobalAccuracy(FedAvg):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.best_accuracy = 0.0
        self.current_parameters = None # track latest aggregated params

    def aggregate_fit(self, rnd, results, failures):
        aggregated = super().aggregate_fit(rnd, results, failures)
        if aggregated is None:
            return None

        parameters, metrics = aggregated
        self.current_parameters = parameters # store for aggregate_evaluate to use

        ndarrays = parameters_to_ndarrays(parameters)
        net = Net()
        state_dict = OrderedDict(
            (k, torch.tensor(v)) for k, v in zip(net.state_dict().keys(), ndarrays)
        )
        net.load_state_dict(state_dict, strict=True)

        torch.save(net.state_dict(), "global_model.pth")
        print(f"[Server] Saved global_model.pth (round {rnd})")
        return parameters, metrics

    def aggregate_evaluate(self, rnd, results, failures):
        if not results:
            print(
                f"\n🌍 [Server] Round {rnd} GLOBAL RESULTS\n"
                f"Global Loss: N/A (no evaluation results, failures={len(failures)})\n"
                f"Global Accuracy: N/A\n"
            )
            return float("inf"), {"global_accuracy": 0.0, "eval_failed": 1.0}

        total_correct = 0.0
        total_samples = 0
        total_loss = 0.0

        for _, res in results:
            num_samples = res.num_examples
            loss = res.loss
            accuracy = float(res.metrics.get("accuracy", 0.0))
            total_loss += loss * num_samples
            total_correct += accuracy * num_samples
            total_samples += num_samples

        global_loss = total_loss / total_samples
        global_accuracy = total_correct / total_samples

        print(
            f"\n🌍 [Server] Round {rnd} GLOBAL RESULTS\n"
```

```
            f"Global Loss: {global_loss:.6f}\n"
            f"Global Accuracy: {global_accuracy:.4f}\n"
        )

        # Save best model based on accuracy
        if global_accuracy > self.best_accuracy:
            self.best_accuracy = global_accuracy
            print(f"[Server] New best accuracy {global_accuracy:.4f} — saving
best_model.pth")
            ndarrays = parameters_to_ndarrays(self.current_parameters)
            net = Net()
            state_dict = OrderedDict(
                (k, torch.tensor(v)) for k, v in zip(net.state_dict().keys(), ndarrays)
            )
            net.load_state_dict(state_dict, strict=True)
            torch.save(net.state_dict(), "best_model.pth")

        return global_loss, {"global_accuracy": global_accuracy, "eval_failed": 0.0}

def server_fn() -> ServerAppComponents:
    def fit_config(rnd: int):
        return {"lr": 1e-4, "local_epochs": 1, "patience": 3, "round": rnd}

    strategy = FedAvgWithGlobalAccuracy(
        fraction_fit = 1.0,
        fraction_evaluate = 1.0,
        min_fit_clients = 5,
        min_evaluate_clients = 5,
        min_available_clients = 5,
        initial_parameters = ndarrays_to_parameters(params),
        on_fit_config_fn = fit_config,
    )

    print("[Server] FedAvg strategy initialized")
    return ServerAppComponents(strategy=strategy)

if __name__ == "__main__":
    print("[Server] Starting Flower server...")
    flwr.server.start_server(
        server_address="0.0.0.0:5000",
        config=ServerConfig(num_rounds=10),
        strategy=server_fn().strategy,
    )
```

6.4: Kafka Producer Python Script

```
import torch.nn.functional as F
from typing import Tuple
import torch.nn as nn
import numpy as np
import torch
import os

MODEL_PATH = os.getenv("MODEL_PATH", "best_model.pth")

class Net(nn.Module):
    def __init__(self, input_dim=4, num_classes=2):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_dim, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 32)
        self.fc5 = nn.Linear(32, num_classes)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        return self.fc5(x)

model = Net()
model.load_state_dict(torch.load(MODEL_PATH, map_location="cpu"))
model.eval()

# Normalization stats (must match training)
MEAN = np.array([27.041304, 61.775265, 969.56335,  3.2226298], dtype=np.float32)
STD  = np.array([8.085047,  24.433617, 105.70269,  2.5380642], dtype=np.float32)

torch.manual_seed(0)
np.random.seed(0)

def normalize(x: np.ndarray) -> np.ndarray:
    return (x - MEAN) / (STD + 1e-8)

# Predict intrusion probability with test-time augmentation (TTA)
def predict_proba(x: np.ndarray, tta: int = 8, noise_scale: float = 0.01) -> float:
# Runs the model 16 times on the same input, each time adding tiny random noise
# Returns the average probability + the ± std deviation

    model.eval()
    probs = []
    with torch.no_grad():
        for _ in range(tta):
            # Add small gaussian noise in normalized space to emulate TTA
            aug = x + np.random.normal(scale=noise_scale, size=x.shape).astype(np.float32)
            inp = torch.tensor(aug).unsqueeze(0)  # shape (1, features)
            logits = model(inp)
            p = torch.softmax(logits, dim=1).cpu().numpy()[0, 1]
            probs.append(float(p))
    return float(np.mean(probs)), float(np.std(probs))

def feature_anomaly_score(x_raw: np.ndarray) -> float:
    z = (x_raw - MEAN) / (STD + 1e-8)
    mahal = float(np.sum(z * z))  # Measures how far a sample is from the known normal
distribution using z-scores
    # Convert to probabilistic-like score via scaled exponential
```

```
    scale = x_raw.size * 2.0
    score = 1.0 - np.exp(-mahal / scale)
    return float(np.clip(score, 0.0, 1.0))

def combined_score(classifier_prob: float, feature_score: float, alpha: float = 0.7) ->
float:
    return alpha * classifier_prob + (1.0 - alpha) * feature_score

def detect_from_score(score: float, threshold: float = 0.5):
    return "INTRUSION 🚨" if score >= threshold else "NORMAL ✅"

# Optional utility: compute best threshold from labeled arrays (requires scikit-learn)
def optimal_threshold_from_labels(y_true: np.ndarray, scores: np.ndarray):
    try:
        from sklearn.metrics import roc_curve
    except Exception:
        raise RuntimeError("sklearn required for threshold calibration: pip install scikit-
learn")
    fpr, tpr, thr = roc_curve(y_true, scores)
    j_scores = tpr - fpr
    idx = np.argmax(j_scores)
    return float(thr[idx])

def random_sensor_sample(anomaly_prob: float = 0.1) -> Tuple[np.ndarray, int]:
    is_anomaly = np.random.rand() < anomaly_prob

    if is_anomaly:
        x_raw = np.array([
            np.random.uniform(40.0,  50.0),   # temperature
            np.random.uniform(90.0, 150.0),   # humidity
            np.random.uniform(600.0, 800.0),  # pressure
            np.random.uniform(7.0,  12.0),    # vibration
        ], dtype=np.float32)
        return x_raw, 1
    else:
        x_raw = np.array([
            np.random.uniform(15.0, 45.0),    # temperature
            np.random.uniform(20.0, 80.0),    # humidity
            np.random.uniform(800.0, 1100.0), # pressure
            np.random.uniform(0.0, 7.0),      # vibration
        ], dtype=np.float32)
        return x_raw, 0

normal_sample    = np.array([25.0,  55.0, 1000.0, 2.5], dtype=np.float32)
intrusion_sample = np.array([45.0, 120.0,  700.0, 9.5], dtype=np.float32)

normal_norm = normalize(normal_sample)  # Normalize
intrusion_norm = normalize(intrusion_sample)

normal_prob, normal_std = predict_proba(normal_norm, tta=16, noise_scale=0.01)  # Predict
with TTA (Test-Time Augmentation)
intr_prob, intr_std = predict_proba(intrusion_norm, tta=16, noise_scale=0.01)

normal_feat_score = feature_anomaly_score(normal_sample) # Feature anomaly scores
intr_feat_score = feature_anomaly_score(intrusion_sample)

alpha = 0.75  # rely mostly on classifier, tune alpha on validation set
normal_comb = combined_score(normal_prob, normal_feat_score, alpha=alpha)
intr_comb = combined_score(intr_prob, intr_feat_score, alpha=alpha)
# Threshold: default 0.5, but should be calibrated on validation set using
optimal_threshold_from_labels
THRESHOLD = 0.5

print("=== Normal sample ===")
print(f"classifier_prob={normal_prob:.4f} ±{normal_std:.4f}")
print(f"feature_score={normal_feat_score:.4f}")
```

```
print(f"combined_score={normal_comb:.4f} -> {detect_from_score(normal_comb, THRESHOLD)}\n")
print("=== Intrusion sample ===")
print(f"classifier_prob={intr_prob:.4f} ±{intr_std:.4f}")
print(f"feature_score={intr_feat_score:.4f}")
print(f"combined_score={intr_comb:.4f} -> {detect_from_score(intr_comb, THRESHOLD)}\n")

def range_based_truth(x_raw: np.ndarray) -> str:
    t, h, p, v = x_raw.tolist()  # <-- now unpacks 4 values
    is_intr = (40 <= t <= 50) and (90 <= h <= 150) and (600 <= p <= 800) and (7 <= v <= 12)
    is_norm = (15 <= t <= 35) and (30 <= h <= 80) and (900 <= p <= 1100) and (0 <= v <= 5)
    if is_intr:
        return "intrusion-by-ranges"
    if is_norm:
        return "normal-by-ranges"
    return "mixed/undefined-by-ranges"

print("=== Random samples ===")
for i in range(10):
    x_raw, true_label = random_sensor_sample(anomaly_prob=0.1)
    x_norm = normalize(x_raw)

    p, p_std = predict_proba(x_norm, tta=16, noise_scale=0.01)
    fs = feature_anomaly_score(x_raw)
    cs = combined_score(p, fs, alpha=alpha)

    pred = detect_from_score(cs, 0.5)

    print(f"\n--- Sample #{i} (true_label={true_label}) ---")
    print(f"x_raw={x_raw}")
    print(f"range_truth={range_based_truth(x_raw)}")
    print(f"classifier_prob={p:.4f} ±{p_std:.4f}")
    print(f"feature_score={fs:.4f}")
    print(f"combined_score={cs:.4f} -> {pred}")

print("\n=== Zoomed Boundary Testing (t=0.45 to t=0.55) ===")
print("Gradually blending all features from normal → intrusion\n")

for t in [round(x * 0.01, 2) for x in range(30, 50)]:
    temp      = 25.0   + t * (45.0  - 25.0)
    humidity  = 55.0   + t * (120.0 - 55.0)
    pressure  = 1000.0 + t * (700.0 - 1000.0)
    vibration = 2.5    + t * (9.5   - 2.5)

    x_raw = np.array([temp, humidity, pressure, vibration], dtype=np.float32)
    prob, _ = predict_proba(normalize(x_raw), tta=16, noise_scale=0.01)
    fs = feature_anomaly_score(x_raw)
    cs = combined_score(prob, fs, alpha=0.75)
    print(f"t={t:.2f} | temp={temp:.1f}, hum={humidity:.1f}, pres={pressure:.1f},
vib={vibration:.1f} → prob={prob:.4f}, combined={cs:.4f} → {detect_from_score(cs,
THRESHOLD)}")

# Precision & Recall
print("\n=== Precision & Recall ===")

TP = 0  # Predicted intrusion, actually intrusion
FP = 0  # Predicted intrusion, actually normal
FN = 0  # Predicted normal,    actually intrusion
TN = 0  # Predicted normal,    actually normal

for _ in range(1000):
    x_raw, true_label = random_sensor_sample(anomaly_prob=0.1)
    prob, _ = predict_proba(normalize(x_raw), tta=16, noise_scale=0.01)
    fs = feature_anomaly_score(x_raw)
    cs = combined_score(prob, fs, alpha=0.75)
    predicted = 1 if cs >= THRESHOLD else 0
```

```
    if predicted == 1 and true_label == 1:
        TP += 1
    elif predicted == 1 and true_label == 0:
        FP += 1
    elif predicted == 0 and true_label == 1:
        FN += 1
    else:
        TN += 1

precision = TP / (TP + FP) if (TP + FP) > 0 else 0.0
recall    = TP / (TP + FN) if (TP + FN) > 0 else 0.0
f1        = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0
else 0.0

print(f"TP={TP}, FP={FP}, FN={FN}, TN={TN}")
print(f"Precision = {precision:.4f}")
print(f"Recall    = {recall:.4f}")
print(f"F1 Score  = {f1:.4f}")
```

References

Almutairi, S., & Barnawi, A. (2023). Federated learning vulnerabilities, threats and defenses: A systematic review and future directions. *Internet of Things*, *24*, 100947. https://doi.org/10.1016/J.IOT.2023.100947

Apache Kafka. (n.d.). Retrieved 25 March 2026, from https://kafka.apache.org/

Bithi, M., Masud, Md. E., & Hossain, Md. A. (2026). A new adaptive federated learning approach for privacy preserving UAV anomaly detection under non-IID distributions. *Scientific Reports 2026 16:1*, *16*(1), 8451-. https://doi.org/10.1038/s41598-026-38732-z

Brendan McMahan, H., Moore, E., Ramage, D., Hampson, S., & Agüera y Arcas, B. (2016). Communication-Efficient Learning of Deep Networks from Decentralized Data. *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017*, *54*. https://arxiv.org/abs/1602.05629v4

Buyuktanir, B., Altinkaya, Ş., Karatas Baydogmus, G., & Yildiz, K. (2025). Federated learning in intrusion detection: advancements, applications, and future directions. *Cluster Computing*, *28*(7), 473-. https://doi.org/10.1007/S10586-025-05325-W/TABLES/4

CrossEntropyLoss — PyTorch 2.11 documentation. (n.d.). Retrieved 28 March 2026, from https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

CS231n Deep Learning for Computer Vision. (n.d.). Retrieved 28 March 2026, from https://cs231n.github.io/linear-classify/#softmax

De Maesschalck, R., Jouan-Rimbaud, D., & Massart, D. L. (2000). The Mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems*, *50*(1), 1–18. https://doi.org/10.1016/S0169-7439(99)00047-7

Docker: Accelerated Container Application Development. (n.d.). Retrieved 25 March 2026, from https://www.docker.com/

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, *27*(8), 861–874. https://doi.org/10.1016/J.PATREC.2005.10.010

Federated learning: what it is and how it works | Google Cloud. (n.d.). Retrieved 28 October 2025, from https://cloud.google.com/discover/what-is-federated-learning

Gupta, S. K., Sharma, A., Dogra, K., & Gupta, P. (2025). Federated Learning for Secure Multi- UAV Coordination . *SECURITY AND PRIVACY*, *8*(6). https://doi.org/10.1002/SPY2.70134

Introduction to Federated Learning - by Avi Chawla. (n.d.). Retrieved 23 December 2025, from https://blog.dailydoseofds.com/p/introduction-to-federated-learning

Introduction to the Apache Kafka messaging system | Redpanda. (n.d.). Retrieved 25 March 2026, from https://www.redpanda.com/guides/kafka-use-cases-messaging-system

Kantharaju, V., Suresh, H., Niranjanamurthy, M., Ansarullah, S. I., Amin, F., & Alabrah, A. (2024). Machine learning based intrusion detection framework for detecting security attacks in internet of things. *Scientific Reports 2024 14:1*, *14*(1), 30275-. https://doi.org/10.1038/s41598-024-81535-3

Karimireddy, S. P., Kale, S., Mohri, M., Reddi, S. J., Stich, S. U., & Suresh, A. T. (2019). SCAFFOLD: Stochastic Controlled Averaging for Federated Learning. *37th International Conference on Machine Learning, ICML 2020*, *PartF168147-7*, 5088–5099. https://arxiv.org/abs/1910.06378v4

Kingma, D. P., & Ba, J. L. (2014). Adam: A Method for Stochastic Optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. https://arxiv.org/abs/1412.6980v9

Lakshminarayanan, B., Pritzel, A., & Blundell, C. (2016). Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles. *Advances in Neural Information Processing Systems*, *2017-December*, 6403–6414. https://arxiv.org/abs/1612.01474v3

Lameh, S. F., Noble, W., Amannejad, Y., & Afshar, A. (2020). Analysis of Federated Learning as a Distributed Solution for Learning on Edge Devices. *2020 International Conference on Intelligent Data Science Technologies and Applications, IDSTA 2020*, 66–74. https://doi.org/10.1109/IDSTA50958.2020.9264060

Li, J., Tong, X., Liu, J., & Cheng, L. (2023). An Efficient Federated Learning System for Network Intrusion Detection. *IEEE Systems Journal*, *17*(2), 2455–2464. https://doi.org/10.1109/JSYST.2023.3236995

Li, P., Liu, Z., Chang, L., Peng, J., & Wu, Y. (2024). *FedBA: Non-IID Federated Learning Framework in UAV Networks*. 121–131. https://doi.org/10.1007/978-3-031-51097-7_11

Liu, J., Huang, J., Zhou, Y., Li, X., Ji, S., Xiong, H., & Dou, D. (2022). From distributed machine learning to federated learning: a survey. *Knowledge and Information Systems 2022 64:4*, *64*(4), 885–917. https://doi.org/10.1007/S10115-022-01664-X

Ng, A. Y. (2004). Feature selection, L1 vs. L2 regularization, and rotational invariance. *Proceedings, Twenty-First International Conference on Machine Learning, ICML 2004*, 615–622. https://doi.org/10.1145/1015330.1015435

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com

Prechelt, L. (2012). Early Stopping — But When? *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7700 LECTURE NO*, 53–67. https://doi.org/10.1007/978-3-642-35289-8_5

strategy - Flower Framework. (n.d.). Retrieved 28 March 2026, from https://flower.ai/docs/framework/ref-api/flwr.serverapp.strategy.html

Terven, J. (2025). Deep Reinforcement Learning: A Chronological Overview and Methods. *AI 2025, Vol. 6, Page 46*, *6*(3), 46. https://doi.org/10.3390/AI6030046

What is clustering? | Machine Learning | Google for Developers. (n.d.). Retrieved 22 October 2025, from https://developers.google.com/machine-learning/clustering/overview

What is Docker? - GeeksforGeeks. (n.d.). Retrieved 25 March 2026, from https://www.geeksforgeeks.org/devops/introduction-to-docker/

What is Machine Learning? | IBM. (n.d.). Retrieved 22 October 2025, from https://www.ibm.com/think/topics/machine-learning

What is reinforcement learning (RL)? | Google Cloud. (n.d.). Retrieved 23 October 2025, from https://cloud.google.com/discover/what-is-reinforcement-learning

What is Supervised Learning? | Google Cloud. (n.d.). Retrieved 22 October 2025, from https://cloud.google.com/discover/what-is-supervised-learning

Zakaria, F., & Khalid, S. K. (n.d.). A Review of Federated Learning Attacks: Threat Models and Defence Strategies. *IJACSA) International Journal of Advanced Computer Science and Applications*, *16*(7), 2025. Retrieved 8 March 2026, from www.ijacsa.thesai.org

Zami, M. B. Al, Uddin, M. R., & Nguyen, D. C. (2025). Secure UAV-assisted Federated Learning: A Digital Twin-Driven Approach with Zero-Knowledge Proofs. *IEEE Internet of Things Journal*. https://doi.org/10.1109/JIOT.2025.3643468