FalconEye

Functional prototype for monitoring critical embedded systems

Real-time supervision for critical environments.

August 2025

Contents

1 Introduction

1.1 Overview

FalconEye is a functional prototype for monitoring critical embedded systems, designed to simulate real-time supervision of sensitive components in a fighter jet, including the engine, hydraulics, airframe, and navigation systems. It is built on a distributed, secure, and modular architecture, reflecting the industrial constraints commonly encountered in the aerospace and defense sectors.

1.2 Motivations

Having previously worked on projects focused on networking and cryptography, I aimed to tackle a more ambitious challenge with a systemic and industrial approach.

The project's objective is to meet the standards of professional technical development, encompassing architecture design, implementation of security measures, unit and integration testing, comprehensive technical documentation, and reproducible demonstrations.

1.3 Simulation Context

In this project, the fighter jet is modeled as an embedded system composed of multiple critical sensors. These sensors continuously transmit data over a secure internal network bus, mirroring the constraints of real-world onboard communications.

The data collected from these sensors is then:

- gathered and centralized through a secure communication protocol,
- stored in a time-series database for efficient access,
- visualized in real-time via interactive dashboards,
- analyzed to automatically detect anomalies or potential failures,
- managed by an automated alerting system that notifies operators instantly through channels such as Discord or email.

2 Project Modules

2.1 Module `sensor-engine`

The `sensor-engine` module manages the simulation of critical embedded systems in a fighter jet. It models in real time several essential subsystems while adhering to the levels of criticality defined by the DO-178C (software) and DO-254 (hardware) standards.

Simulated Critical Systems and Levels of Criticality According to the Design Assurance Levels (DAL) classification defined in the DO-178C and DO-254 standards, the following systems are simulated:

- **Engine System (Engine Control Unit - ECU)**
  - *DAL A*: safety-critical engine control.
  - Simulated parameters: temperature (40°C to 1200°C), oil pressure (0 to 10 bar), vibrations (0 to 50 g).
- **Hydraulic Systems**
  - *DAL B*: control of flight surfaces and brakes.
  - Simulated parameters: hydraulic pressure (0 to 350 bar), pump status.
- **Airframe and Mechanical Integrity**
  - *DAL C*: monitoring of vibrations and mechanical stress.
  - Simulated parameters: accelerations (0 to 20 g).
- **Avionics Navigation Systems**
  - *DAL A*: GPS and airspeed data critical for flight.
  - Simulated parameters: GPS coordinates, airspeed (0 to 2500 km/h).
- **Flight Control Systems**
  - *DAL A*: angle-of-attack sensors, control surface positions.
  - Simulated parameters: angle of attack (15° to +25°), control surface positions (30° to +30°).

Standards References The criticality levels are based on:

- RTCA DO-178C for embedded software.
- RTCA DO-254 for electronic hardware components.

2.2 Module `mqtt-broker`

The `mqtt-broker` module serves as the central communication hub between the simulated sensors, analysis modules, and the visualization interface. It is based on the MQTT (Message Queuing Telemetry Transport) protocol, known for its lightweight design, low latency, and suitability for distributed embedded systems, such as those found in aerospace environments.

Functionality

- **Message reception**: Simulated sensors (`sensor-engine`) periodically publish messages on various topics (e.g., `aircraft/engine/temp`).
- **Routing to consumers**: Modules subscribed to the corresponding topics (visualization, anomaly analysis) receive data in real time.
- **Fine-grained client management**: The broker handles authentication, access control (ACL), message retention, and session management.
- **Support for QoS (Quality of Service) levels**:
  - QoS 0: At most once (best effort)
  - QoS 1: At least once (acknowledged delivery)
  - QoS 2: Exactly once (maximum reliability)

Security Given the sensitivity of the exchanged data (simulated as critical for a fighter jet), the module implements several cybersecurity measures:

- **Client authentication** via X.509 certificates or username/password.
- **TLS encryption** to secure MQTT traffic (port 8883).
- **Access control** (ACL) to restrict publishing and subscribing to specific topics only.
- **Abuse protection**: rate limiting, connection limits, and log monitoring.

Implementation The module relies on an open-source broker (Mosquitto or EMQX), configured locally and scriptable for integration into the simulation pipeline. Messages can also be monitored in real time using tools such as MQTT Explorer or Node-RED.

Example Topics Used

```
aircraft/engine/temp
aircraft/hydraulics/pressure
aircraft/structure/vibrations
aircraft/nav/gps
alerts/critical
```

Conclusion Although simple, the MQTT protocol proves particularly well-suited for efficient communication between sensors and analyzers in a simulated embedded environment. When combined with proper security measures, it meets the requirements of critical military systems while maintaining high integration flexibility. However, it would not be suitable for real-world deployment.

2.3 Module `data-collector`

The `data-collector` module is responsible for centralizing, validating, and persistently storing all data emitted by the simulated sensors and received via the `mqtt-broker`. It serves as a critical entry point in the monitoring chain, enabling subsequent analysis, visualization, or alert triggering.

Functionality

- **Data subscription**: The `data-collector` is an MQTT client that subscribes to all topics simulating sensor data (`aircraft/#`).
- **Message validation**: It verifies the received JSON structure, mandatory fields (`timestamp`, `value`, `unit`, etc.), and their consistency.
- **Time series storage**: Measurements are persisted in an InfluxDB database, specialized for timestamped data.
- **Intelligent structuring**: Each sensor becomes an InfluxDB "measurement" enriched with *tags* such as `source`, `system`, `unit`, etc.
- **Data exposure**: The collected data can then be accessed by the graphical interface or downstream analysis modules.

Why InfluxDB InfluxDB was chosen for its ability to:

- efficiently handle thousands of points per second with low latency,
- automatically compress data for optimized long-term storage,
- provide a high-performance query language (`Flux`) for temporal aggregations,
- offer a native HTTP API enabling easy integration with other components.

Example of Stored Data

```
measurement: engine_temperature
tags:
    source: sensor-01
    unit: Celsius
    aircraft_zone: rear
fields:
    value: 725.0
time: 2025-07-26T10:30:00Z
```

The `data-collector` ensures reliable and efficient historization of simulated critical measurements. It supports traceability, long-term analysis, and visualization, while respecting the structure of a modern industrial monitoring system.

2.4 Module `anomaly-detector`

The `anomaly-detector` module is responsible for automatically detecting abnormal behavior from the data stored in InfluxDB. Its goal is to identify any critical deviation, classify its severity level, and trigger an alert mechanism accordingly.

General Operation The module regularly queries InfluxDB to analyze the latest measurements collected by the `data-collector`. These analyses are performed according to predefined rules based on:

- **Static thresholds** (e.g., engine temperature ¿ 800°C),
- **Dynamic deviations** (e.g., sudden temperature rise),
- **Sliding time windows** (trend analysis).

Adaptive frequency based on criticality Each aircraft subsystem is associated with a *criticality level* (inspired by DO-178C/DO-254). The module adjusts its analysis frequency according to this level:

- **Criticality A (e.g., engine)**: analysis every 2 seconds.
- **Criticality B (e.g., hydraulics)**: every 5 seconds.
- **Criticality C-D (structure, navigation)**: every 10–15 seconds.

This approach optimizes system performance while complying with safety requirements.

Detection and Alerting When an anomaly is detected, a message is immediately published on a dedicated MQTT topic for the relevant system. Example:

```
Topic : alerts/engine/overheat
Payload : {
    "timestamp": "2025-07-26T10:30:00Z",
    "system": "engine",
    "type": "temperature",
    "value": 825,
    "level": "CRITICAL",
    "message": "Engine temperature too high"
}
```

These messages can be consumed by the `grafana` module to be relayed via Discord, email, or other alert channels.

2.5 Module `grafana`

The `grafana` module serves as the main interface for visualizing and monitoring the collected data and alerts generated by the system. Connected to the InfluxDB database, it allows the creation of interactive real-time dashboards, facilitating the monitoring of simulated critical parameters.

Data Visualization Grafana retrieves both historical and real-time data stored in InfluxDB. It offers a wide variety of graphical panels (line charts, gauges, histograms) to track the evolution of measurements such as engine temperature, hydraulic pressure, or structural vibrations.

Alert Management Although Grafana cannot directly consume MQTT messages, it receives alerts emitted by the `anomaly-detector` module through an intermediate channel (e.g., inserting an entry into a dedicated InfluxDB alerts measurement or via an HTTP webhook). These alerts are then displayed on dashboards and can trigger notifications to various channels (email, Discord, Slack).

Advantages and Limitations

- **Advantages:**
  - Rich and configurable graphical interface,
  - Native support for time-series databases like InfluxDB,
  - Ability to define simple alert thresholds with built-in notifications,
  - Easy access to historical data and correlation between metrics.
- **Limitations:**
  - Depends on a third-party module to inject alerts into the database.

Conclusion The `grafana` module plays a central role in visual supervision and simple alert management. Combined with the `anomaly-detector` module, it enables efficient monitoring of simulated critical systems while providing a modern and user-friendly interface.

2.6 Module `security-layer`

The `security-layer` module ensures the security of communications and access to the various services of the FalconEye system, particularly those exposed via web interfaces like Grafana. This module is essential to guarantee confidentiality, integrity, and authentication in a critical environment.

Securing Access to Grafana An NGINX reverse proxy is deployed in front of Grafana to:

- Terminate TLS/SSL connections, ensuring encrypted exchanges between users and Grafana.
- Implement prior authentication (e.g., Basic Auth, OAuth, or LDAP) to restrict access to dashboards.
- Apply filtering rules, such as blocking suspicious IP addresses or rate limiting, to protect against denial-of-service attacks or other network threats.
- Centrally manage TLS certificates, simplifying security maintenance and deployment.

Cross-Service Security Beyond Grafana, the `security-layer` module also protects all critical system services (MQTT broker, APIs, databases) using similar mechanisms, ensuring a centralized and consistent security policy.

Complementarity with MQTT Broker Security Although the MQTT broker includes its own security mechanisms (TLS, client authentication), the `security-layer` module provides an additional, system-wide layer, notably via the NGINX reverse proxy, which acts as a single secure entry point for all exposed components.

Conclusion Implementing the `security-layer` module ensures that sensitive communications are encrypted, access is controlled, and the system as a whole is better protected against attacks and intrusions—essential in the context of critical embedded systems.

2.7 Module `tests`

The `tests` module groups all automated tests designed to validate the quality and reliability of the FalconEye system. It includes:

- **Unit Tests**: verify the correct functioning of individual functions and components (simulators, data collector, anomaly detector).
- **Integration Tests**: check interactions between modules, including MQTT communication, writing to InfluxDB, and alert generation.
- **Functional Tests**: simulate complete scenarios to validate overall system behavior and compliance with requirements.
- **Performance Tests**: assess performance under high load conditions, ensuring the system can process data in real time.

These tests are automated using appropriate tools (e.g., pytest, JUnit) and run regularly to ensure code stability during development.

2.8 Module `documentation`

The `documentation` module consolidates all project-related documents, essential for understanding, using, and maintaining the system:

- **Architecture**: detailed description of modules, their interactions, and technical choices.
- **Specifications**: definition of functional and non-functional requirements.
- **User Guides**: instructions for installing, configuring, and using the interfaces (e.g., Grafana dashboards).
- **Test Documentation**: test protocols, results, and validation procedures.

3 Architecture

3.1 Architecture - Simulators

Figure 1 shows the overall architecture of the `sensor-engine` module, which simulates the different critical embedded systems of the fighter aircraft.

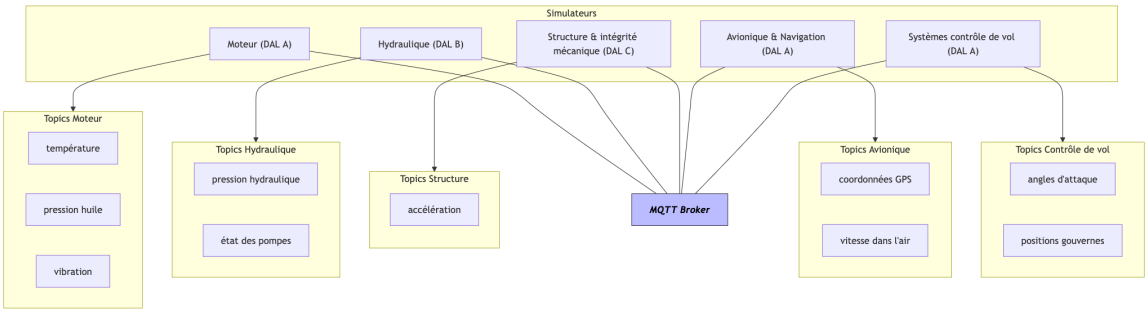

Figure 1: Architecture of the `sensor-engine` module simulating critical embedded systems

3.2 Architecture - Data Communication

Figure 2 illustrates the secure architecture of the `mqtt-broker` module. This module centralizes communication between the simulators (`sensor-engine`) and the analysis, visualization, and alerting components. Security is ensured through a layer integrating TLS and an authentication system.

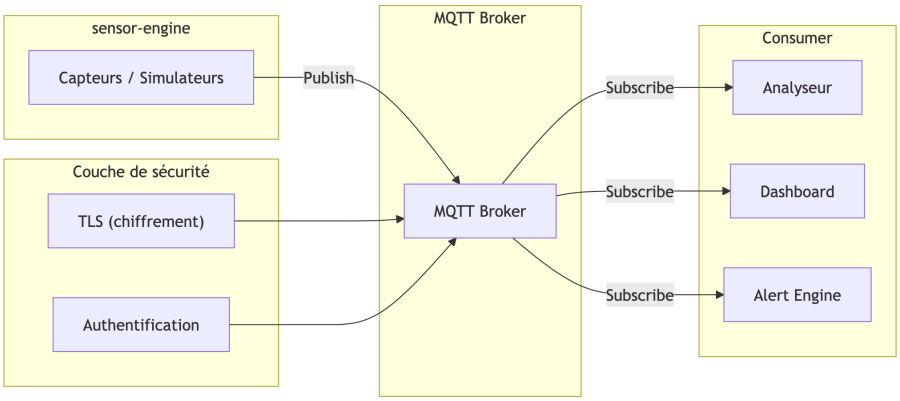

Figure 2: Secure architecture of the MQTT Broker module with publish/subscribe

3.3 Architecture – Data Management

Figure 3 shows the functioning of the `data-collector` module and its interaction with the infrastructure for storing and analyzing critical data. This module plays a central role in receiving, validating, and persisting simulated measurements from onboard sensors.

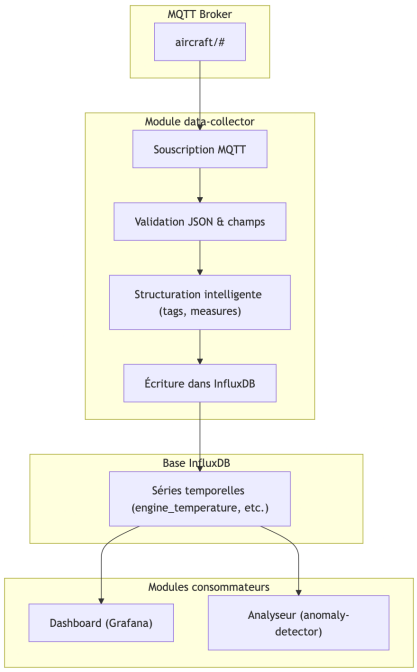

Figure 3: Data management architecture via the `data-collector` module and InfluxDB

3.4 Architecture – Alert System

Figure 4 shows the operation of the `anomaly-detector` module, responsible for automatic monitoring of simulated critical parameters. This component regularly queries the InfluxDB database to analyze collected measurements according to three types of rules: static thresholds, dynamic variations, and temporal trends.

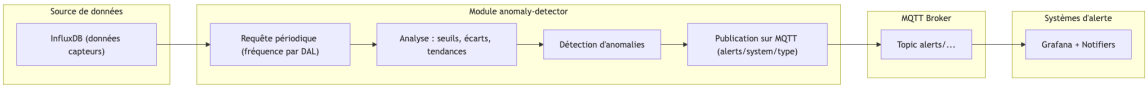

Figure 4: Architecture of the `anomaly-detector` module for anomaly detection

3.5 Architecture – Dashboard

Figure 5 illustrates the role of the `grafana` module in the FalconEye supervision interface. This component enables real-time visualization of critical data and alerts generated by the system.

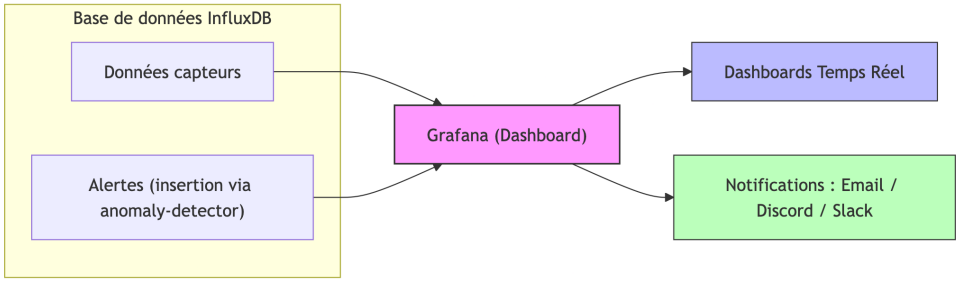

Figure 5: Integration architecture of the `grafana` module for supervision

3.6 Architecture – Security

Figure 6 shows the architecture of the `security-layer` module, which ensures the security of communications and access to critical components of the FalconEye system. It relies on an NGINX reverse proxy configured as a single secure entry point.

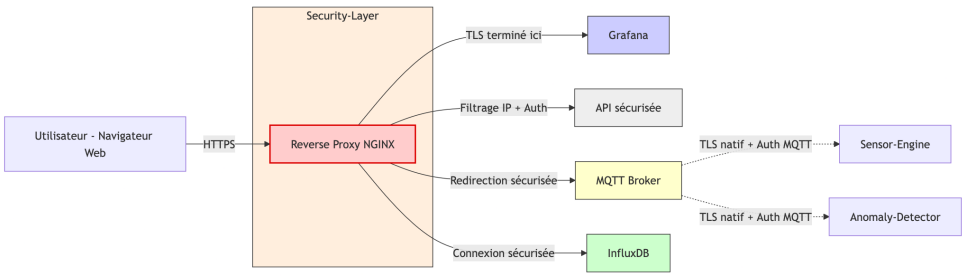

Figure 6: Security architecture via the `security-layer` module (NGINX)

4 Infrastructure Setup

4.1 Falconnet Network

To effectively isolate services and ensure reliable communication, a dedicated Docker network named *falconnet* has been set up.

The three main services are:

- **Mosquitto**: the MQTT broker, essential for communication between sensors and data consumers;
- **InfluxDB**: the time-series database where all sensor-transmitted values are stored;
- **Grafana**: the visualization platform used to create supervision dashboards.

Each onboard system of the aircraft (see Figure 1) is encapsulated in a separate Docker container, ensuring strong software isolation and improving maintainability.

A specific container, named *mqtt-collector-influxdb*, handles the reception, validation, and transfer of JSON messages sent by sensors to the database. It thus acts as a trusted gateway between the sensor world and data storage.

An integration test script, `check_network.sh`, verifies both the presence of all containers in the *falconnet* network and the connectivity between critical services, notably along the path **Mosquitto → MQTT Collector → InfluxDB**.

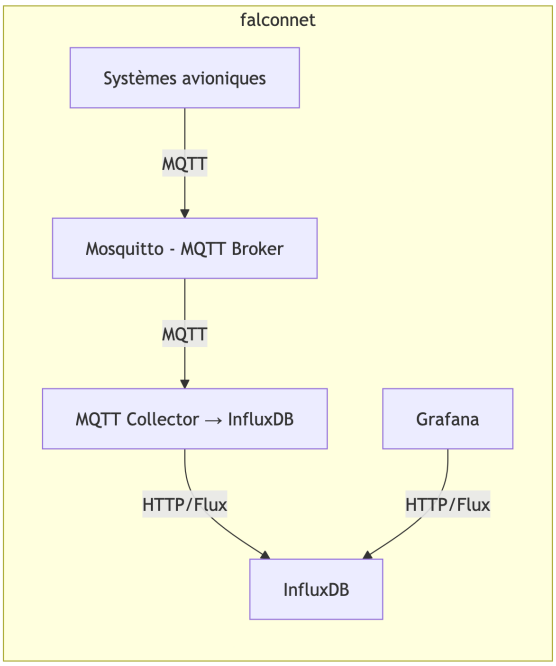

Figure 7: FalconEye network architecture

4.2 Simulator Operation

The simulators play a crucial role in generating data for the various avionics systems. Their primary goal is to realistically reproduce the behavior of onboard sensors and instruments to test the platform's integration and overall functioning.

Each simulator is encapsulated in a separate Docker container, allowing strict isolation between systems and facilitating independent maintenance and module evolution. These simulators continuously generate navigation data, such as latitude, longitude, speed, and temperature, with a configurable publication interval.

The simulation process relies on realistic evolution models for the measured parameters: for instance, the GPS position is updated each cycle according to the simulated speed, ensuring coherent movement through space. Random anomalies are also introduced to test system robustness against unexpected events, such as sudden position jumps or extreme speed variations.

Messages produced by the simulators are formatted in JSON and published on dedicated MQTT topics via the Mosquitto broker. This lightweight, asynchronous protocol allows efficient communication between simulators and the data collection system.

This modular architecture ensures not only realistic simulation but also great flexibility for integrating new sensors or adjusting simulation models as needed.

Simulation code excerpt:

```
import time
import json
import random
import paho.mqtt.client as mqtt
from datetime import datetime, timezone

mqtt_client = mqtt.Client()
mqtt_client.connect("mosquitto", 1883)

latitude = 48.8566
longitude = 2.3522
airspeed = 800.0

def evolve_position(lat, lon, speed_kmh, interval_s=1):
    # Approximate movement in degrees (1 deg  111 km)
    distance_km = speed_kmh * (interval_s / 3600)
    delta_deg = distance_km / 111
    # Simulate movement north-east for simplicity
    lat += delta_deg * (random.uniform(0.8, 1.2))
    lon += delta_deg * (random.uniform(0.8, 1.2))
```

```
    return round(lat, 6), round(lon, 6)

while True:
    latitude, longitude = evolve_position(latitude, longitude, airspeed)
    message = {
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "latitude": latitude,
        "longitude": longitude,
        "airspeed": airspeed,
        "unit": "degrees",
        "source": "nav-sensor-01"
    }
    mqtt_client.publish("aircraft/navigation/position/1", json.dumps(message))
    time.sleep(1)
```

4.3 MQTT Broker Deployment

The MQTT broker is essential for communication from sensors to the *InfluxDB* database. To enable this communication, ports **8883** (MQTT over TLS) and **9001** (MQTT via WebSocket over TLS) have been opened.

For **TLS** setup, an internal Certificate Authority was created to sign both the server certificate and the certificates for the various clients (sensors, database, etc.). The script *create_client_cert.sh* automates the creation of these client certificates.

The configuration requires that any client wishing to connect to the broker present a certificate signed by the same Certificate Authority as the server, using the *require_certificate* parameter.

Communications are encrypted using the **ECDHE** (Elliptic Curve Diffie-Hellman Ephemeral) cipher suite, which allows ephemeral key exchanges for each connection. This approach ensures *Perfect Forward Secrecy* (PFS): even if a session key is compromised, past communications remain protected.

In addition to encryption, an **Access Control List** (ACL) mechanism has been implemented to restrict actions that each client is authorized to perform on specific topics, further enhancing overall system security.

4.4 Access Control Lists (ACL)

In addition to TLS certificate-based authentication, the MQTT broker uses **Access Control Lists** (ACLs) to limit client actions on specific topics. For example, ACLs can restrict a motor sensor to publishing only on topics `aircraft/engine/#` and prevent access to topics reserved for other subsystems.

The original configuration aimed to use Mosquitto's *use_identity_as_username* option,

which automatically maps the identity (CN) from the client certificate to the username for ACL purposes. However, this mechanism could not be implemented because the broker was unable to correctly extract the CN during the TLS connection.

Faced with this limitation, we opted for username/password authentication via a `passwd` file generated with the `mosquitto_passwd` tool. Each client connects with specific credentials, and ACLs are applied based on this username.

While this solution still ensures effective access control, combining certificate-based identity with password control would have enabled **two-factor authentication**, further strengthening security.

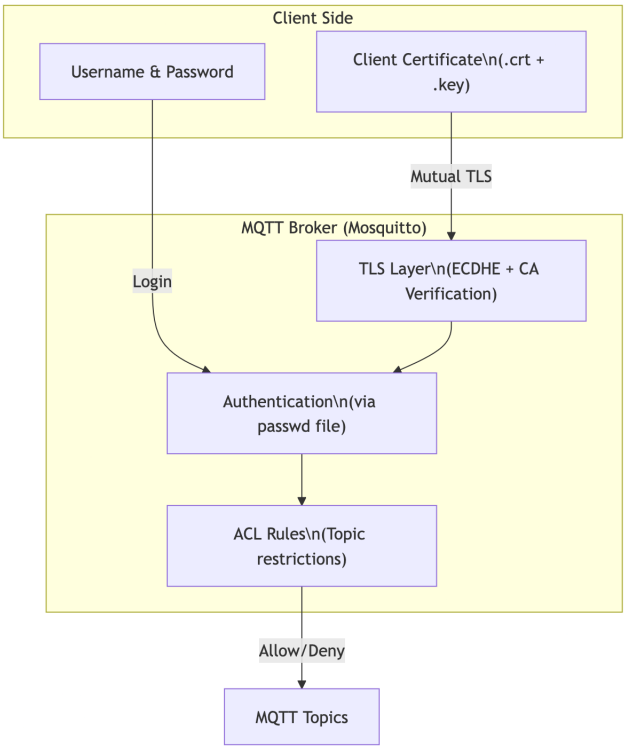

Figure 8: ACL/TLS setup in the Mosquitto system

4.5 Data Collector Deployment

The data collector bridges the **Mosquitto MQTT broker** and the **InfluxDB** database. It subscribes to various aircraft-related topics and inserts received messages as points that can be exploited by Grafana.

Broker Connection. The collector uses the `paho-mqtt` library to establish a secure connection with the Mosquitto broker. Authentication is performed using TLS certificates, and the collector subscribes to the global topic `aircraft/#`, enabling it to receive all messages published by onboard sensors.

Message Reception and Processing. When a message is received, it is decoded from `JSON`. The collector then determines the type of data by analyzing the MQTT topic structure:

- **Alerts**: messages under `aircraft/alerts/...` contain critical information (e.g., engine temperature, anomalies). They are enriched with metadata such as severity, aircraft zone, and unit.
- **GPS Position**: messages under `aircraft/position/...` provide latitude and longitude. These data are stored with appropriate fields (geographical coordinates and unit in degrees).
- **Standard Sensor Data**: for other sensors, the collector records the measured value, unit, and contextual information (source, aircraft zone).

Writing to InfluxDB. Each message is transformed into a `Point` (InfluxDB native format) containing:

- *tags*: qualitative information used for filtering (sensor name, unit, aircraft zone, severity, etc.);
- *fields*: numeric or textual measured values (e.g., temperature, latitude, alert level);
- *timestamp*: precise instant provided by the sensor.

These points are then synchronously inserted into the **InfluxDB bucket**, ensuring data persistence and temporal consistency.

Operational Loop. Once initialized, the collector continuously listens to the MQTT broker. It automatically records any new data published by the aircraft sensors, ensuring a continuous data flow to the time-series database.

Summary. This component is essential for transforming MQTT messages into data usable by InfluxDB. It ensures **reliable collection, normalization, and secure storage** of flight information, enabling Grafana to generate real-time dashboards from this data.

4.6 Grafana and InfluxDB Deployment

The deployment of the **FalconEye** data visualization and storage infrastructure is based on Docker Compose. The main services are:

- **InfluxDB**: time-series storage of raw sensor data and alerts generated by the collector. The instance is configured using the following environment variables:

  - `DOCKER_INFLUXDB_INIT_USERNAME`, `DOCKER_INFLUXDB_INIT_PASSWORD`: administrator user (stored in plaintext here for simplification)
  - `DOCKER_INFLUXDB_INIT_ORG`, `DOCKER_INFLUXDB_INIT_BUCKET`: initial organization and bucket
  - `DOCKER_INFLUXDB_INIT_ADMIN_TOKEN`: access token

  Authentication is thus enabled, but since passwords are stored in plaintext, this setup should not be used in production.

- **Grafana**: data visualization and dashboard creation. Each simulator has a dedicated dashboard to monitor its variables, and an additional dashboard is planned for alerts. Access is secured by an administrator user and password configured via:
  - `GF_SECURITY_ADMIN_USER`
  - `GF_SECURITY_ADMIN_PASSWORD`

  However, the alerts dashboard is not currently functional, as multi-sensor alert handling and false positive reduction have not yet been fully implemented.

- **Collector**: an intermediary service that retrieves MQTT messages, validates their format, and injects them into InfluxDB. It depends on the `mosquitto` and `influxdb` services.

Network Architecture All services communicate through the Docker network `falconnet`, ensuring isolation and connectivity between simulators, the MQTT broker, the collector, InfluxDB, and Grafana.

Volumes and Persistence InfluxDB and Grafana data are mounted on local volumes to ensure persistence when containers are restarted.

Current Limitations

- Passwords and tokens are stored in plaintext in Docker Compose, representing a security risk.
- The alerts dashboard is not yet operational, and sensor-based alerts have not been implemented to avoid a massive flow of false positives due to the simulators' random behavior.
- Grafana allows observation of historical data and sensor values, but alerts must currently be consulted directly in InfluxDB.

5 Anomaly Detection

The **FalconEye** system integrates two complementary analysis modes for generating alerts from avionics sensor data:

- **Real-time analysis via MQTT**: each message published by sensors is processed as soon as it is received. A filtering logic (EMA + threshold rules + derivative detection) is applied immediately. Alerts are published to a dedicated channel (`aircraft/alerts/...`) if an anomaly is detected.
- **Backfill analysis (InfluxDB)**: raw data are first stored in InfluxDB. A periodic script queries recent historical data (e.g., the last 10 minutes) and applies the same detection rules. Alerts are emitted with a slight time delay but in a more robust manner, less sensitive to transient fluctuations.

5.1 Mathematical Choices

1. Exponential Moving Average (EMA) Both scripts use an *exponential moving average* to reduce noise:

$$EMA_t = \alpha \cdot x_t + (1 - \alpha) \cdot EMA_{t-1}$$

where:

- x_t : raw value at time t
- α : smoothing factor ($\alpha = 0.25$ in our case)
- EMA_t : smoothed value

This choice attenuates random fluctuations while remaining responsive to real trends.

2. Derivative Detection (Rate of Change) The **rate of change** between two successive timestamps is calculated as:

$$rate = \frac{EMA_t - EMA_{t-\Delta t}}{\Delta t}$$

An alert is generated if $|rate|$ exceeds a predefined threshold ($rate_{thresh}$).

Example: for engine vibration ($max = 50$, $rate_{thresh} = 5$), an increase of 20 units in 3 seconds gives:

$$rate = \frac{20}{3} \approx 6.67 > 5 \quad \Rightarrow \quad slopealert$$

3. Threshold Verification Each measurement has bounds $[min, max]$. If a value falls outside this range, a `threshold` alert is generated:

$$alert = (x_t < min) \ \vee \ (x_t > max)$$

Example: if pressure exceeds 350 PSI, an immediate alert is triggered.

5.2 Comparison of Approaches

Criterion	**Real-time (MQTT)**	**Delayed (InfluxDB)**
Responsiveness	Very high (instant alert)	Less responsive (10-min window)
Noise Robustness	Low (risk of false positives)	Higher (historical analysis)
Resource Usage	Continuous (infinite loop)	Periodic (heavier queries)
Deployment	Simple (MQTT + local rules)	More complex (InfluxDB required)
Use Case	In-flight safety (real-time)	Post-flight validation / analysis

5.3 Limitations and Possible Optimizations

- **Excessive alert generation**: simulators produce noisy signals ⇒ many false positives. *Possible solutions*: adjust α, increase $rate_{thresh}$, introduce temporal persistence (e.g., alert only if anomaly persists for n seconds).
- **Simple per-key cooldown**: currently, an alert is blocked for $30s$ after emission for a given sensor. *Optimization*: make the cooldown adaptive, proportional to the anomaly amplitude.
- **Lack of inter-sensor correlation**: rules are single-sensor. *Optimization*: introduce multi-source rules (e.g., engine temperature + oil pressure).
- **Merging both approaches**: use **real-time MQTT** for immediate detection and **InfluxDB delayed** for confirming or invalidating alerts afterward.

5.4 Sensor Presence Monitoring (Bash Script)

In addition to the two approaches (real-time via MQTT and delayed via InfluxDB), a **Bash script** was implemented to check that all sensor simulator containers are present

in the `falconnet` network.

The script runs in a loop every 5 seconds, queries Docker network status, and compares the list of expected services (engine, hydraulics, navigation, etc.) with those actually connected. If a service is missing, a notification is sent directly to the operator's machine (via `notify-send` on Linux or `osascript` on macOS).

Advantages:

- Simple to implement (lightweight script, no complex dependencies).
- Immediate responsiveness (visual/local alert within seconds).
- Independent of InfluxDB and Grafana (works even if the monitoring stack is stopped).

Disadvantages:

- Local notifications only: alerts are not integrated into the **Grafana dashboard**, so they are not visible in the main aircraft interface.
- No centralized history: unlike InfluxDB alerts, no record is stored for later analysis.
- Risk of visual overload if multiple sensors are missing simultaneously.

Discussion: This mechanism addresses a basic supervision need but lacks integration into the FalconEye ecosystem. Ideally, these alerts should be sent to InfluxDB and then displayed in Grafana. Currently:

- **Presence alerts** (missing sensors) are visible only via this script.
- **Value-based alerts** (threshold exceedances, derivatives) are not displayed in Grafana because the dashboard is still incomplete and noise-related false positives made the solution unusable.
- It is still possible to consult **raw alerts in InfluxDB**, which centralizes detected events.

A future improvement would be to **merge this Bash script with the Grafana alerting logic** to provide a complete dashboard (anomalous values + missing sensors) in the monitoring interface.

6 Integration of an NGINX Reverse Proxy

In the current architecture, sensors send their data to a TLS-secured Mosquitto *broker*, then a collector processes the information and inserts it into **InfluxDB**. Finally, **Grafana** is used to visualize dashboards built from this data.

6.1 Role of a Reverse Proxy

A **reverse proxy** such as **NGINX** sits in front of one or more services to serve as a single entry point. It is mainly used to:

- Terminate SSL/TLS connections (centralized certificates).
- Apply additional security rules (authentication, IP filtering).
- Unify access to services (URL- or subdomain-based routing).
- Cache or compress certain responses to improve performance.

6.2 Relevance in Our Context

In our project, *MQTT* communications between sensors and Mosquitto are already TLS-secured. Additionally, Grafana is intended to be used only within a restricted network (dashboard for a fighter jet), which limits the immediate usefulness of a reverse proxy.

However, if we wanted to expose Grafana or the InfluxDB API to remote users (for example, in a ground control center), NGINX could be deployed as a *single exit point*. It would ensure secure access and centralized SSL configuration.

6.3 Example Setup with Docker

A typical `docker-compose` configuration could define three services: InfluxDB, Grafana, and NGINX. NGINX would be the only service exposing ports externally (80/443) and would forward requests to the internal services.

Listing 1: Simplified docker-compose.yml example

```
version: "3.9"
services:
  influxdb:
    image: influxdb:2.7
    networks: [backend]

  grafana:
    image: grafana/grafana:10.0.0
    networks: [backend]

  nginx:
    image: nginx:latest
    ports:
      - "80:80"
      - "443:443"
```

```
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on: [influxdb, grafana]
    networks: [backend]

networks:
  backend:
    driver: bridge
```

Listing 2: Simplified NGINX configuration example

```
http {
    server {
        listen 80;

        location /grafana/ {
            proxy_pass http://grafana:3000/;
        }

        location /influx/ {
            proxy_pass http://influxdb:8086/;
        }
    }
}
```

In this setup, the external user communicates only with NGINX, which relays the request to Grafana or InfluxDB depending on the URL. This model makes the architecture more modular and secure in case of external access.

6.4 Conclusion

Integrating an NGINX reverse proxy is therefore not essential within the current scope of the project but represents a valuable enhancement for any future extension toward remote or multi-user access. In that case, it would centralize TLS certificate management and strengthen the overall platform security.

7 Deployment Automation

Within the project, the architecture (Mosquitto, collector, InfluxDB, Grafana) was deployed using `docker-compose`. An alternative would have been to automate this deployment with a tool like **Ansible**.

Specifically, an Ansible playbook could have allowed to:

- automatically install Docker and Docker Compose on the target machine;
- create the necessary directories and copy configuration files (TLS certificates, `docker-compose.yml` etc.);
- launch the complete stack with a single command;
- easily redeploy everything in case of failure or update.

A minimal example of an Ansible playbook could look like this:

```
---
- hosts: avionics_hosts
  become: true
  tasks:
    - name: Install Docker
      apt:
        name: docker.io
        state: present
        update_cache: yes

    - name: Install Docker Compose
      apt:
        name: docker-compose
        state: present

    - name: Create directories for the FalconEye stack
      file:
        path: "/opt/falconeye/{{ item }}"
        state: directory
        mode: '0755'
      loop:
        - docker/mqtt
        - docker/influxdb
        - docker/grafana
        - config/certs

    - name: Copy configuration files
      copy:
        src: "{{ item.src }}"
        dest: "{{ item.dest }}"
        owner: root
        group: root
```

```
      mode: '0644'
    loop:
      - { src: 'docker-compose.yml', dest: '/opt/falconeye/docker-compose.yml' }
      - { src: 'config/certs/', dest: '/opt/falconeye/config/certs/' }

  - name: Launch the Docker Compose stack
    command: docker-compose up -d
    args:
      chdir: /opt/falconeye
```

This approach would have been particularly useful in an aeronautical context, where **reproducibility**, **rapid redeployment**, and **reduction of human errors** are essential. Although not implemented here (the code is already versioned on GitHub), it represents a potential improvement to professionalize and secure deployment.

8 Securing Communications and Access

Security is a central concern in any distributed architecture, especially when services are exposed outside a local network. Several complementary mechanisms can be considered to protect the infrastructure.

8.1 TLS on Mosquitto

In this project, **TLS** encryption was implemented on the **Mosquitto** broker. This mechanism ensures the confidentiality and integrity of data exchanged between sensors and the collector, preventing third parties from intercepting or modifying MQTT messages. The use of certificates also allows authentication between different entities of the system.

8.2 Firewall to Reduce Attack Surface

Another common security measure is the use of a **firewall** (e.g., `ufw` on Linux) to restrict incoming traffic. In a real deployment, it would be relevant to:

- Allow only strictly necessary ports (e.g., 8883 for Mosquitto, 3000 for Grafana);
- Block all other ports to reduce the attack surface;
- Limit access to certain trusted IP addresses when possible.

In this project, this configuration was not implemented because the environment remains experimental and the services run locally. Nevertheless, in a production context, a firewall is an essential barrier to limit intrusion risks.

8.3 Fail2ban for Dynamic Attack Response

Another mechanism that could be added is **Fail2ban**. This tool analyzes service logs (e.g., Mosquitto or Grafana login failures) and automatically applies filtering rules to block suspicious IP addresses for a given period.

In a real architecture, Fail2ban would allow to:

- Temporarily or permanently block IP addresses performing repeated login attempts;
- Reduce the impact of brute-force password attacks;
- Add an adaptive defense layer in addition to a static firewall.

8.4 Chosen Approach in This Project

Only **TLS on Mosquitto** was implemented in this project, as it represented the most relevant measure to protect the confidentiality of data transmitted by the sensors. Adding a firewall and a system like Fail2ban would have limited benefit in a controlled and restricted environment like the project.

However, in an operational deployment (e.g., on an online server), combining the three mechanisms — TLS encryption, port restriction via firewall, and dynamic defense via Fail2ban — would constitute a robust approach to secure the infrastructure.

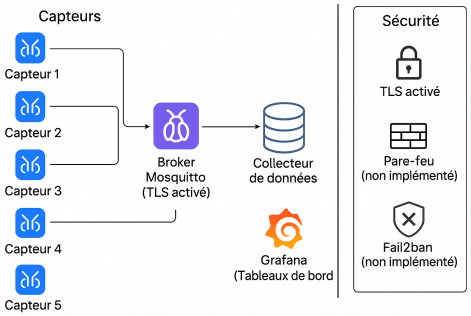

Figure 9: Communications Security

9 Testing and Validation

To ensure the robustness and security of the **FalconEye** architecture, several categories of tests were implemented. These tests verify the **system's resilience to errors**, the **security of communications**, and the **continuity of data flow** between components.

9.1 Message Integrity Test

A dedicated container was added to deliberately send malformed messages to the MQTT broker (for example, replacing a numeric value with a string). The **collector** is then

responsible for rejecting these invalid messages and logging an error. This test validates **protection against data corruption** and **compliance with the expected format**.

9.2 Network Penetration Test

A `Kali Linux` container was integrated into the infrastructure to simulate network attacks. The main scenarios tested include:

- **Port scanning** (using `nmap`) to identify exposed services;
- **TLS checks** with `openssl` to verify certificate validity;
- **Unauthorized publishing attempts** on Mosquitto using fake credentials.

These simulated attacks confirm that the infrastructure **rejects any unauthorized access**.

9.3 Certificates and ACLs Test

A dedicated script checks the proper functioning of the trust chain and access rules:

1. Publish and subscribe with a valid certificate on an authorized topic (expected success).
2. Publish with a valid certificate on a forbidden topic (expected failure).
3. Connect with an invalid password (expected rejection).
4. Connect without a client certificate (expected rejection).

These tests ensure that **mutual TLS**, **ACLs**, and password authentication function according to requirements.

9.4 Network Connectivity Test

A second script automates the following checks:

- verify that all services (`mosquitto`, `influxdb`, `grafana`, simulators, etc.) are connected to the Docker network `falconnet`;
- **ping tests** between critical containers (e.g., collector → broker and collector → InfluxDB);
- check **InfluxDB service health** via its HTTP endpoint `/health`;
- test the **open ports** on the host (8883 for MQTT, 8086 for InfluxDB, 3000 for Grafana, 9001 for MQTT WebSockets).

These tests confirm that the **infrastructure is properly interconnected** and that essential services are reachable.

9.5 End-to-End Test

Finally, a full test was implemented: a simulator sends data to the MQTT broker, which is collected by the collector and stored in InfluxDB. An HTTP request queries InfluxDB to ensure the data has been persisted. This test validates the **integrity of the end-to-end data processing chain**, from simulated sensors to Grafana dashboards.

10 Conclusion

This project enabled the deployment of a complete avionics simulation architecture based on Docker containers and orchestrated with `docker-compose`. The main objective was achieved: understanding and deploying an infrastructure that integrates a secure MQTT broker (Mosquitto), a data collector, a time-series database (InfluxDB), and a visualization tool (Grafana).

The most educational aspect was the **implementation of the network and software architecture** and the **securing of communications** (TLS, certificates, ACLs). Simulator code played a secondary role: most work involved configuration, learning Docker, and project structuring (folder organization, automation scripts).

10.1 Potential Improvements

Several areas for improvement were identified:

- **Documentation**: enrich user and developer documentation to simplify project onboarding and deployment.
- **Testing**: extend existing integrity and security tests with unit tests on simulators. Using `pytest`, one could automatically verify the validity of generated messages (types, units, value ranges).
- **Continuous Integration**: implement a branch dedicated to automated tests, executed via GitHub Actions. This approach would ensure infrastructure reliability and prevent regressions.

10.2 Grafana Dashboards

A central element of the project is the creation of dashboards in Grafana, allowing real-time visualization of simulated sensor data: navigation, flight controls, hydraulics, engine, and structural metrics. These dashboards provide a clear and dynamic overview of avionics operations.

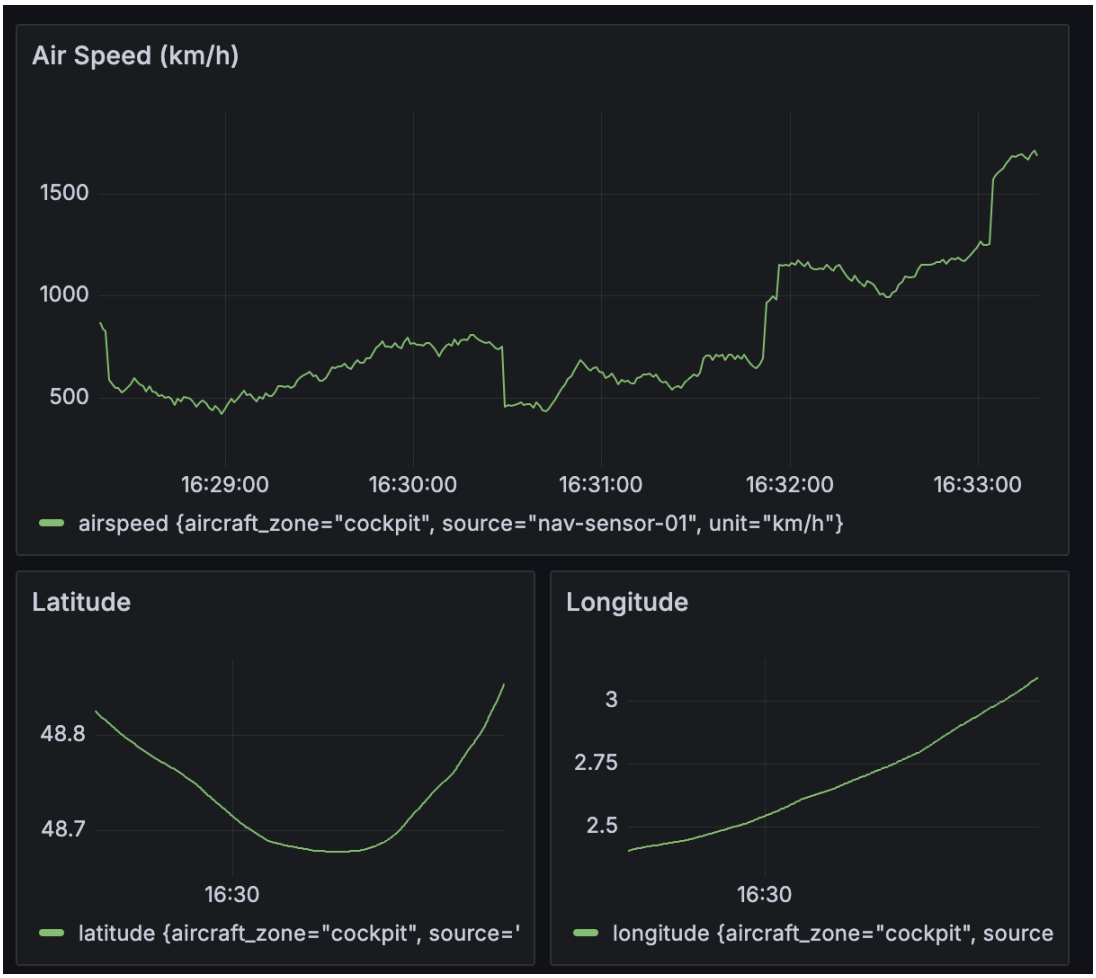

Figure 10: Example Grafana dashboard for engine monitoring.

In conclusion, this project provided comprehensive experience in deploying a distributed, secure, and visualizable architecture from scratch. It establishes a solid foundation that can be extended with automated testing, enhanced documentation, and continuous integration to move toward a more professional environment.