MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

PLATAFORMA DE INGESTA Y MONITORIZACIÓN DE TELEMETRÍA PARA SISTEMAS CRÍTICOS CON DETECCIÓN DE ANOMALÍAS

Autor: Sergio Rodríguez García

Director: Atilano Ramiro Fernández-Pacheco Sánchez-Migallón

Madrid

Junio de 2026

Declaración de originalidad

Declaro bajo mi responsabilidad que el Proyecto presentado con el título plataforma de ingesta y monitorización de telemetría para sistemas críticos con detección de anomalías e la ETS de Ingeniería – ICAI de la Universidad Pontificia Comillas en el curso académico 2ºMIT es de mi autoría y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Uso de Inteligencia Artificial[1]

Declaro bajo mi responsabilidad que (indicar la opción correcta):

☐ No he utilizado Inteligencia Artificial en la elaboración del presente documento.

∗ He utilizado Inteligencia Artificial en la elaboración del presente documento y/o del Anexo B siempre en las condiciones permitidas por la Universidad Pontificia Comillas, es decir, aplicando el Nivel 2 de la Escala de Evaluación de Perkins et al. (2024): "*La IA puede utilizarse para actividades previas a la tarea, como la lluvia de ideas, la descripción y la investigación inicial. Este nivel se centra en el uso de la IA para la planificación, las síntesis y la generación de ideas, pero las evaluaciones deben hacer hincapié en la capacidad de desarrollar y refinar estas ideas de forma independiente*". En concreto, las Inteligencia Artificial ha sido empleada para:

- Ayuda con la planificación del desarrollo de la plataforma (fases y pasos a seguir) y con la resolución de errores de esta - Revisión de la escritura de la memoria - Investigación de referencias para el estado del arte

[1] Esta declaración se refiere al uso de la Inteligencia Artificial generativa para realizar los documentos del Proyecto (Anexo B y Memoria). No aplica a Proyectos donde, por su naturaleza, deban emplear inteligencia artificial como parte de los mismos (aplicación de técnicas de aprendizaje automático, redes neuronales, análisis de datos…)


Firmado (alumno): Sergio Rodríguez García
Fecha: 07/07/2026

Autorización para la entrega del Proyecto

El Director del Proyecto	El co-Director del Proyecto (si aplica)
	
Fdo: Atilano Ramiro Fernández-Pacheco Sánchez-Migallón	Fdo:
Fecha: 07/07/2026	Fecha:

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

PLATAFORMA DE INGESTA Y MONITORIZACIÓN DE TELEMETRÍA PARA SISTEMAS CRÍTICOS CON DETECCIÓN DE ANOMALÍAS

Autor: Sergio Rodríguez García

Director: Atilano Ramiro Fernández-Pacheco Sánchez-Migallón

Madrid

Junio de 2026

Agradecimientos

Quiero agradecer a mi director, Atilano Ramiro Fernández-Pacheco Sánchez-Migallón, su orientación y disponibilidad a lo largo de todo el desarrollo del proyecto, así como a mi familia y compañeros por su apoyo durante estos 6 años.

PLATAFORMA DE INGESTA Y MONITORIZACIÓN DE TELEMETRÍA PARA SISTEMAS CRÍTICOS CON DETECCIÓN DE ANOMALÍAS

Autor: Rodríguez García, Sergio.
Director: Fernández-Pacheco Sánchez-Migallón, Atilano Ramiro.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

Este trabajo diseña, implementa y valida una plataforma de ingesta, almacenamiento, monitorización y detección de anomalías en tiempo real para datos de telemetría de vuelo. El sistema combina un servicio de ingesta con contrato de datos cerrado, persistencia en PostgreSQL, observabilidad mediante Prometheus y Grafana, y un módulo de detección basado en una red LSTM multiclase capaz de identificar en tiempo real el estado concreto de una aeronave, no solo si su comportamiento es anómalo, sino cuál es la anomalía, integrado en un panel de visualización con alarma.

Palabras clave: telemetría; ingesta de datos en tiempo real; detección de anomalías; aprendizaje automático; LSTM; monitorización; sistemas críticos

1. Introducción

Los sistemas críticos que generan telemetría de forma continua, entre ellos las aeronaves, requieren infraestructuras capaces de ingerir, almacenar y analizar datos a alta velocidad y con baja latencia, así como de detectar comportamientos anómalos antes de que deriven en incidencias graves. Las plataformas comerciales existentes en el sector aeronáutico suelen operar como ecosistemas cerrados orientados a la visualización de tráfico o al mantenimiento predictivo, dejando un hueco para soluciones abiertas y extensibles centradas específicamente en la identificación del estado concreto de una aeronave en tiempo real.

2. Definición del proyecto

El objetivo general del proyecto es diseñar, implementar y validar una plataforma abierta de ingesta de telemetría en tiempo real y detección de anomalías aplicada a sistemas críticos, empleando la telemetría de vuelo como dominio de validación. Sus objetivos específicos incluyen diseñar una arquitectura modular y escalable, validar la calidad de los datos recibidos, almacenar la telemetría de forma eficiente, monitorizar el estado del sistema y detectar comportamientos anómalos de la aeronave en tiempo real, identificando su tipo concreto.

3. Descripción del modelo/sistema/herramienta

El sistema se compone de un servicio de ingesta (FastAPI) que valida cada punto de telemetría frente a un contrato de datos cerrado (Pydantic v2) antes de persistirlo en PostgreSQL; un módulo de observabilidad basado en Prometheus y Grafana; un módulo de detección de anomalías que compara cuatro modelos (un baseline estadístico, Isolation Forest, Local Outlier Factor y una red LSTM) y despliega el mejor de ellos, la

LSTM, reformulada como clasificador multiclase, integrado en el propio flujo de ingesta; y un panel de visualización en React con mapa en tiempo real y sistema de alarmas.

4. Resultados

- La LSTM multiclase obtiene la mejor precisión y AP de los cuatro modelos comparados (F1 = 0,816 en la vista binaria) y es la única capaz de identificar el estado concreto de la aeronave. Tras corregir tres discrepancias entre el comportamiento offline y en producción, la detección en tiempo real alcanza una precisión del 100% durante el ascenso, del 73,7% durante el crucero y del 94,8% durante una maniobra de fallo de motor simulada, validada de forma end-to-end sobre una demostración de diez vuelos.

5. Conclusiones

El proyecto demuestra que es posible construir, con herramientas de código abierto, una plataforma completa de monitorización inteligente de telemetría capaz de identificar en tiempo real el estado concreto de un sistema crítico. Quedan como líneas de trabajo futuro la conexión con fuentes de datos de vuelo reales y el reentrenamiento del modelo sobre un conjunto de datos sintéticos de mayor tamaño.

6. Referencias

[1] S. Hochreiter y J. Schmidhuber, “Long Short-Term Memory,” Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[2] F. T. Liu, K. M. Ting, y Z.-H. Zhou, “Isolation Forest,” Proc. IEEE International Conference on Data Mining, 2008.https://technet.microsoft.com/en-us/magazine/hh509051.aspx

[3] A. Chandola, A. Banerjee, y V. Kumar, “Anomaly Detection: A Survey,” ACM Computing Surveys, vol. 41, no. 3, pp. 1–58, 2009.http://blogthinkbig.com/big-data-emprendedor/

INGESTION AND MONITORING PLATFORM FOR TELEMETRY IN CRITICAL SYSTEMS WITH ANOMALY DETECTION

Author: Rodríguez García, Sergio.
Supervisor: Fernández-Pacheco Sánchez-Migallón, Atilano Ramiro.
Collaborating Entity: ICAI – Universidad Pontificia Comillas

ABSTRACT

This work designs, implements and validates a real-time ingestion, storage, monitoring and anomaly detection platform for flight telemetry data. The system combines an ingestion service with a closed data contract, persistence in PostgreSQL, observability through Prometheus and Grafana, and a detection module based on a multiclass LSTM network capable of identifying, in real time, the specific state of an aircraft, not only whether its behaviour is anomalous, but which anomaly it is, integrated into a real-time visualization panel with alarms.

Keywords: telemetry; real-time data ingestion; anomaly detection; machine learning; LSTM; monitoring; critical systems

1. Introduction

Critical systems that continuously generate telemetry, aircraft among them, require infrastructures capable of ingesting, storing and analysing data at high speed and with low latency, as well as detecting anomalous behaviour before it leads to serious incidents. Existing commercial platforms in the aviation sector tend to operate as closed ecosystems oriented towards traffic visualization or predictive maintenance, leaving room for open, extensible solutions focused specifically on identifying the concrete state of an aircraft in real time.

2. Project definition

The project's general objective is to design, implement and validate an open platform for real-time telemetry ingestion and anomaly detection applied to critical systems, using flight telemetry as the validation domain. Its specific objectives include designing a modular and scalable architecture, validating the quality of incoming data, storing telemetry efficiently, monitoring the system's health, and detecting anomalous aircraft behaviour in real time, identifying its specific type.

3. Description of the model/system/tool

The system comprises an ingestion service (FastAPI) that validates every telemetry point against a closed data contract (Pydantic v2) before persisting it in PostgreSQL; an observability module based on Prometheus and Grafana; an anomaly detection module that compares four models (a statistical baseline, Isolation Forest, Local Outlier Factor and an LSTM network) and deploys the best of them, the LSTM, reformulated as a multiclass classifier, integrated into the ingestion flow itself; and a React-based visualization panel with a real-time map and an alarm system.

4. Results

The multiclass LSTM achieves the best precision and AP among the four compared models (F1 = 0.816 in its binary view) and is the only one able to identify the aircraft's specific state. After correcting three discrepancies between offline and production behaviour, real-time detection reaches 100% accuracy during climb, 73.7% during cruise and 94.8% during a simulated engine-failure manoeuvre, validated end-to-end over a ten-flight demonstration.

5. Conclusions

The project shows that it is possible to build, using open-source tools, a complete intelligent telemetry monitoring platform capable of identifying, in real time, the specific state of a critical system. Connecting to real flight data sources and retraining the model on a larger synthetic dataset remain as future work.

6. References

[1] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[2] F. T. Liu, K. M. Ting, and Z.-H. Zhou, “Isolation Forest,” Proc. IEEE International Conference on Data Mining, 2008.https://technet.microsoft.com/en-us/magazine/hh509051.aspx

[3] A. Chandola, A. Banerjee, and V. Kumar, “Anomaly Detection: A Survey,” ACM Computing Surveys, vol. 41, no. 3, pp. 1–58, 2009.http://blogthinkbig.com/big-data-emprendedor/

ÍNDICE DE LA MEMORIA

ÍNDICE DE FIGURAS

CAPÍTULO 1 – INTRODUCCIÓN

La creciente complejidad de los sistemas tecnológicos actuales, junto con su progresiva digitalización, ha impulsado la necesidad de disponer de mecanismos avanzados de supervisión y control capaces de proporcionar información fiable y en tiempo real sobre su estado operativo. En este contexto, la telemetría se ha consolidado como una herramienta fundamental para la observación continua de sistemas distribuidos, permitiendo recoger y analizar datos generados por sensores, dispositivos y componentes software a lo largo del tiempo.

La correcta gestión de estos datos resulta clave para garantizar aspectos como la eficiencia, la disponibilidad y la seguridad de los sistemas, especialmente en aquellos entornos en los que un fallo puede tener consecuencias significativas. No obstante, el aumento en el volumen, la velocidad y la diversidad de los datos de telemetría plantea retos técnicos relevantes que requieren soluciones específicas desde el punto de vista de la ingeniería de sistemas y de datos.

Este capítulo introduce el marco general en el que se desarrolla el proyecto, contextualizando la problemática asociada a la gestión de telemetría y estableciendo las

bases conceptuales que justifican la necesidad de una solución estructurada y robusta. A partir de esta introducción, se analizan los principales desafíos inherentes a la ingesta y gestión de datos de telemetría, se presentan las motivaciones que impulsan el desarrollo del trabajo y se describe, a alto nivel, el enfoque adoptado para dar respuesta a dichas problemáticas.

1.1 CONTEXTO Y PROBLEMÁTICA DE LA GESTIÓN DE TELEMETRÍA

La gestión de telemetría es hoy un elemento clave en muchos sistemas tecnológicos, especialmente en aquellos que son complejos, distribuidos y críticos. La digitalización de procesos industriales, infraestructuras, transportes y servicios ha incrementado de forma notable la cantidad de datos generados por sensores, dispositivos embebidos y sistemas software. Estos datos, producidos de manera continua y con dependencia temporal, son esenciales para supervisar el funcionamiento del sistema, analizar su comportamiento y apoyar la toma de decisiones en tiempo real.

Sin embargo, disponer de grandes volúmenes de datos no es suficiente. Para que la telemetría sea realmente útil, es necesario contar con mecanismos sólidos de ingesta, validación, almacenamiento y monitorización que garanticen la calidad, coherencia y trazabilidad de la información. Uno de los principales retos es la heterogeneidad de las fuentes, ya que en un mismo sistema pueden coexistir dispositivos con distintos formatos, frecuencias de muestreo y niveles de fiabilidad, lo que dificulta su tratamiento uniforme.

A esto se suman otros problemas como las limitaciones de conectividad, los fallos de comunicación o los errores en los dispositivos emisores, que pueden generar pérdidas de datos, duplicidades e inconsistencias temporales. Además, al tratarse de flujos continuos y en muchos casos de alta frecuencia, los sistemas de gestión deben procesar la información con baja latencia y capacidad de escalado, algo especialmente exigente en entornos críticos

donde un retraso o una pérdida de datos puede afectar a la seguridad o al funcionamiento correcto del sistema.

Por todo ello, la gestión de telemetría debe abordarse de forma integral, incluyendo no solo la recepción y almacenamiento de datos, sino también su validación, la observabilidad del propio sistema y la detección temprana de anomalías. Esta necesidad justifica el desarrollo de plataformas de gestión de telemetría orientadas a sistemas críticos, donde la fiabilidad, la escalabilidad y la trazabilidad son requisitos fundamentales.

1.2 MOTIVACIONES DEL PROYECTO

El desarrollo de este proyecto surge de la combinación de distintos factores que han puesto de manifiesto el interés y la oportunidad de abordar una solución centrada en la telemetría de vuelo en tiempo real. La creciente importancia de disponer de sistemas capaces de ingerir, validar y supervisar datos de forma continua, especialmente en entornos técnicos complejos, ha motivado el planteamiento de un trabajo orientado al diseño de una plataforma modular, observable y preparada para la detección de anomalías.

Además, este proyecto responde tanto a inquietudes formativas como a necesidades de carácter práctico. Por un lado, permite profundizar en tecnologías, arquitecturas y metodologías actuales dentro del ámbito de la ingeniería de telecomunicación y los sistemas distribuidos. Por otro, ofrece una aplicación concreta en un contexto donde la fiabilidad, la monitorización y el análisis temprano de comportamientos anómalos resultan especialmente relevantes. A partir de ello, las motivaciones del proyecto pueden agruparse en tres dimensiones principales: técnica, académica y aplicada.

1.2.1 MOTIVACIONES TÉCNICAS

Desde una perspectiva técnica, el desarrollo de plataformas de telemetría en tiempo real plantea retos relacionados con la arquitectura, el tratamiento de datos y la supervisión de sistemas complejos. El gran volumen de información generado por múltiples fuentes

obliga a diseñar soluciones capaces de ingerir, procesar y almacenar datos de forma eficiente, sin comprometer la fiabilidad ni la escalabilidad.

Uno de los principales desafíos es garantizar la calidad y coherencia de los datos durante todo su ciclo de vida. Errores de formato, inconsistencias temporales o valores fuera de rango pueden reducir notablemente la utilidad de la telemetría, por lo que es necesario incorporar mecanismos de validación y control desde las primeras etapas de ingesta. Además, estas plataformas deben gestionar flujos continuos de información con baja latencia, persistencia y acceso eficiente, lo que exige arquitecturas orientadas al procesamiento en tiempo real.

También resulta fundamental monitorizar el propio sistema de gestión de telemetría. La observabilidad permite analizar el comportamiento de la plataforma, detectar cuellos de botella e identificar incidencias en los procesos de ingesta o almacenamiento. A ello se suma la incorporación de técnicas de detección de anomalías, que facilitan la identificación automática de comportamientos inusuales y la detección temprana de fallos o degradaciones.

Estas cuestiones muestran el interés técnico del proyecto, que se orienta al estudio y desarrollo de una arquitectura capaz de integrar la ingesta de datos, la validación de la información, la persistencia estructurada y los mecanismos de observabilidad y análisis dentro de una misma plataforma.

1.2.2 MOTIVACIONES ACADÉMICAS

Desde el punto de vista académico, este proyecto permite aplicar de forma práctica conocimientos adquiridos en programación, bases de datos, arquitectura de sistemas y tratamiento de datos temporales. La construcción de una plataforma de gestión de telemetría exige combinar conceptos de distintos ámbitos, lo que favorece una visión más completa e integrada de los sistemas de información actuales.

Uno de sus principales intereses académicos es el estudio de arquitecturas orientadas al procesamiento de datos continuos, cada vez más relevantes en sistemas distribuidos.

Además, el proyecto permite profundizar en el análisis de datos temporales, abordando retos relacionados con su almacenamiento, organización y explotación. A esto se suma la introducción de técnicas de detección de anomalías, que conectan fundamentos teóricos del aprendizaje automático y la minería de datos con su aplicación en un sistema real.

El trabajo contribuye al desarrollo de competencias en el diseño e implementación de sistemas software complejos. La definición de una arquitectura clara, la integración de distintos componentes y la validación mediante pruebas y simulaciones convierten el proyecto en una experiencia formativa útil para consolidar conocimientos y adquirir una perspectiva más práctica del desarrollo tecnológico.

1.2.3 MOTIVACIONES APLICADAS

Además de su interés técnico y académico, el proyecto tiene una clara motivación aplicada, ligada a la importancia creciente de la gestión de datos de telemetría en muchos sectores tecnológicos. Actualmente, numerosos sistemas complejos dependen de la recopilación y análisis continuo de información para supervisar su funcionamiento, detectar incidencias y optimizar su rendimiento.

La telemetría se emplea en ámbitos como el transporte, la industria, las infraestructuras críticas o los sistemas aeronáuticos, donde resulta esencial disponer de información actualizada sobre el estado del sistema. En este contexto, las plataformas de ingesta y monitorización permiten centralizar datos de múltiples fuentes, ofrecer una visión integrada del comportamiento del sistema y facilitar su análisis a lo largo del tiempo. Además, la detección automática de anomalías mejora la capacidad de respuesta ante fallos y reduce la dependencia de la supervisión manual.

El proyecto se ajusta a estas necesidades al proponer una plataforma que integra ingesta de datos, almacenamiento estructurado, monitorización y análisis básico de anomalías. Aunque parte de datos simulados, la arquitectura está pensada para adaptarse a escenarios más cercanos a aplicaciones reales, lo que refuerza su carácter práctico.

1.3 ENFOQUE GENERAL DE LA SOLUCIÓN PROPUESTA

La solución propuesta se plantea como una plataforma modular capaz de recibir telemetría de vuelo en tiempo real, validarla frente a un contrato de datos cerrado, almacenarla de forma estructurada y ofrecer visibilidad tanto sobre el estado de las aeronaves simuladas como sobre el propio funcionamiento del sistema. El diseño se apoya en componentes independientes y desplegables de forma aislada, comunicados mediante interfaces bien definidas, lo que facilita su evolución y su sustitución individual a medida que el proyecto avance.

1.3.1 FILOSOFÍA DE DISEÑO

El diseño del sistema se guía por un conjunto de principios orientados a la separación de responsabilidades y a la robustez frente a datos incorrectos. En primer lugar, se adopta un enfoque de contrato cerrado de telemetría: toda la información que entra al sistema debe ajustarse a una estructura y a un conjunto de reglas de negocio definidas de antemano, validadas de forma explícita antes de ser aceptada. En segundo lugar, se persigue una validación en profundidad, reforzando a nivel de base de datos las mismas reglas ya comprobadas en la capa de aplicación, de modo que ningún dato inconsistente pueda persistirse aunque falle alguna de las comprobaciones anteriores. Por último, cada componente del sistema (ingesta, almacenamiento, generación de telemetría, observabilidad) se diseña como una pieza independiente, ejecutada en su propio contenedor, lo que permite razonar, probar y sustituir cada parte del sistema por separado.

1.3.2 VISIÓN DE ALTO NIVEL

A alto nivel, el sistema se organiza como una tubería de datos que comienza en los emisores sintéticos de telemetría, encargados de generar vuelos físicamente plausibles e inyectar anomalías controladas siguiendo el contrato definido. Estos datos se envían mediante peticiones HTTP a un servicio de ingesta, que valida cada punto recibido y lo persiste en una base de datos relacional junto con la información de vuelo y de segmentación

asociada. En paralelo, el propio servicio de ingesta expone métricas operativas que son recogidas y visualizadas mediante una pila de observabilidad, permitiendo supervisar tanto el volumen y la validez de los datos recibidos como el estado general del sistema. Esta misma infraestructura de datos, una vez almacenada de forma estructurada y con anomalías etiquetadas de forma controlada, sienta las bases para incorporar en fases posteriores un módulo de detección automática de comportamientos anómalos.

1.3.3 ALCANCE DEL TRABAJO

El alcance de este trabajo se centra en el diseño, la implementación y la validación de la plataforma de ingesta, almacenamiento y observabilidad de telemetría, así como en los emisores sintéticos capaces de generar datos de vuelo realistas con anomalías controladas. Se incluye también el despliegue reproducible de todo el sistema mediante contenedores y la definición de una suite de pruebas automatizadas que valida el comportamiento del conjunto. Quedan fuera del alcance directo de este trabajo, y se plantean como líneas de continuación, la migración del almacenamiento a una base de datos especializada en series temporales, la integración de una fuente de datos de vuelo real, el desarrollo de una interfaz web propia y la implementación y comparación de los modelos de detección de anomalías, cuya base de datos etiquetada queda ya preparada por el sistema desarrollado.

CAPÍTULO 2 – DEFINICIÓN DEL TRABAJO

Este capítulo delimita con precisión el trabajo desarrollado: en primer lugar, se justifica la necesidad del proyecto a partir de las carencias detectadas en las soluciones actuales de gestión de telemetría; a continuación, se establecen el objetivo general y los objetivos específicos que guían el desarrollo del sistema; y, por último, se describe la metodología de trabajo seguida a lo largo del proyecto.

2.1 JUSTIFICACIÓN

Resulta necesario justificar de forma concreta la conveniencia de abordar el proyecto propuesto. En este apartado se analiza, desde una perspectiva más aplicada, por qué el desarrollo de una plataforma orientada a la gestión y análisis de datos de telemetría puede aportar valor frente a las soluciones existentes.

Para ello, se examinan en primer lugar las necesidades actuales relacionadas con la gestión de telemetría, posteriormente se analizan algunas limitaciones presentes en las plataformas de monitorización disponibles, y finalmente se expone la propuesta del proyecto y los elementos que constituyen su principal valor diferencial.

2.1.1 NECESIDAD DE SOLUCIONES AVANZADAS PARA LA GESTIÓN DE TELEMETRÍA

En muchos sistemas tecnológicos actuales, la telemetría es esencial para supervisar el funcionamiento y analizar el comportamiento del sistema. Sin embargo, el aumento del volumen de datos y la complejidad de los entornos monitorizados hace necesario contar con soluciones más avanzadas para gestionar esta información de forma eficiente.

Muchos de estos sistemas generan datos continuos y con componente temporal, por lo que se necesitan plataformas capaces de ingerirlos, procesarlos y almacenarlos sin afectar a la fiabilidad. Además, su utilidad no depende solo de recopilar información, sino de poder analizarla de forma estructurada para extraer conocimiento útil.

Por ello, resulta necesaria la existencia de soluciones integradas que centralicen datos de distintas fuentes, faciliten su explotación y refuercen las capacidades de supervisión y diagnóstico en entornos donde disponer de información fiable y actualizada es especialmente importante.

2.1.2 LIMITACIONES DE LAS PLATAFORMAS ACTUALES DE MONITORIZACIÓN

Aunque existen muchas herramientas para monitorizar sistemas y seguir datos en tiempo real, gran parte de ellas presentan limitaciones cuando se analiza la gestión de telemetría de forma global. En muchos casos se centran en recopilar y visualizar información, pero no incorporan capacidades avanzadas de análisis sobre los datos obtenidos.

Esto hace que la detección de anomalías o degradaciones siga dependiendo en gran medida de la interpretación humana. Además, muchas veces la ingesta, el almacenamiento, la monitorización y el análisis se resuelven con herramientas separadas, lo que obliga a realizar integraciones adicionales y dificulta una gestión unificada del ciclo de vida de los datos.

Todo esto, justifica el desarrollo de plataformas que reúnan en un mismo entorno la recogida de datos, su almacenamiento estructurado, la supervisión del sistema y el análisis automático de comportamientos anómalos, ofreciendo así una solución más completa y coherente.

2.1.3 PROPUESTA DEL PROYECTO Y VALOR DIFERENCIAL

El proyecto propone desarrollar una plataforma para la gestión y análisis de datos de telemetría aeronáutica, integrando en un mismo sistema la ingesta, el almacenamiento, la monitorización y el análisis de información generada de forma continua. La idea es disponer de una solución que permita recoger datos de vuelo, estructurarlos y utilizarlos después en tareas de supervisión.

Además, la plataforma incorpora mecanismos de observabilidad para controlar el funcionamiento del propio sistema y detectar incidencias durante el tratamiento de los datos. También plantea la aplicación de técnicas de detección de anomalías sobre la telemetría recopilada, con el fin de identificar comportamientos inusuales de forma automática.

Todo ello se apoya en una arquitectura modular, diseñada para conectar de forma ordenada las distintas fases del ciclo de vida de los datos, desde su recepción y validación hasta su almacenamiento y análisis posterior. Este enfoque permite integrar diferentes componentes tecnológicos dentro de una misma solución, facilitando una gestión más coherente de la información y una mayor capacidad de supervisión sobre los datos de telemetría aeronáutica.

2.2 OBJETIVOS

El presente proyecto se articula en torno a un objetivo general, que resume el propósito último del trabajo, y a un conjunto de objetivos específicos que concretan las capacidades técnicas necesarias para alcanzarlo.

2.2.1 OBJETIVO GENERAL

Diseñar, implementar y validar una plataforma modular para la ingesta, el almacenamiento, la monitorización y el análisis de datos de telemetría aeronáutica, capaz de garantizar la calidad de los datos recibidos, ofrecer observabilidad sobre su propio funcionamiento y sentar las bases para la detección automática de comportamientos anómalos.

2.2.2 OBJETIVOS ESPECÍFICOS

2.2.2.1 DISEÑAR UNA ARQUITECTURA MODULAR Y ESCALABLE

Diseñar una plataforma organizada en componentes independientes (ingesta, almacenamiento, generación de telemetría y observabilidad), desplegables de forma aislada mediante contenedores y comunicados a través de un contrato de datos bien definido, de modo que el sistema pueda evolucionar y escalar sin necesidad de rediseñar su arquitectura.

2.2.2.2 VALIDAR LA CALIDAD DE LOS DATOS RECIBIDOS

Definir un contrato de telemetría cerrado y validarlo en profundidad, tanto en la capa de aplicación mediante reglas declarativas y validadores explícitos, como en la base de datos mediante restricciones que refuercen la misma lógica, garantizando que únicamente se almacenan datos coherentes con la estructura y las reglas de negocio establecidas.

2.2.2.3 ALMACENAR TELEMETRÍA DE FORMA EFICIENTE

Diseñar un modelo de datos relacional capaz de almacenar de forma estructurada la telemetría recibida, la información de los vuelos y su segmentación por escenario, incorporando la indexación necesaria para soportar las consultas del sistema y previendo su evolución hacia una base de datos optimizada para series temporales a medida que crezca el volumen de ingestión.

2.2.2.4 MONITORIZAR EL ESTADO DEL SISTEMA

Instrumentar el servicio de ingesta para exponer métricas operativas sobre el volumen y la validez de los datos recibidos, y desplegar una pila de observabilidad que permita visualizar dichas métricas, facilitando la supervisión continua del estado del sistema y la detección temprana de problemas de funcionamiento.

2.2.2.5 DETECTAR COMPORTAMIENTOS ANÓMALOS

Generar de forma controlada telemetría con comportamientos anómalos etiquetados, correctamente caracterizados por tipo, metadato y puntuación de confianza, de manera que el sistema disponga de un conjunto de datos fiable sobre el que diseñar, entrenar y comparar modelos de detección automática de anomalías.

2.3 METODOLOGÍA

El desarrollo del proyecto ha seguido una metodología de carácter iterativo, organizada en ciclos sucesivos de diseño, implementación, simulación y validación. Cada ciclo se apoya en los resultados obtenidos en el anterior: los hallazgos surgidos durante la simulación y la validación de un componente retroalimentan su diseño e implementación, permitiendo refinar progresivamente tanto la arquitectura como el comportamiento del sistema a medida que avanza el proyecto.

2.3.1 DISEÑO MODULAR

El sistema se ha estructurado desde el inicio en componentes claramente separados (el servicio de ingesta, el modelo de datos, los emisores sintéticos de telemetría y la infraestructura de despliegue y observabilidad), cada uno responsable de una única función y comunicado con el resto a través de interfaces explícitas, como el contrato de telemetría o el esquema de base de datos. Esta separación ha permitido avanzar en el desarrollo de cada componente de forma independiente y sustituir o ampliar partes concretas del sistema sin afectar al resto.

2.3.2 IMPLEMENTACIÓN

La implementación del sistema se ha llevado a cabo de forma incremental, añadiendo funcionalidad en pasos sucesivos y verificables. Esto se refleja, por ejemplo, en la evolución del esquema de base de datos mediante migraciones incrementales que han ido incorporando de forma progresiva la gestión de metadatos de anomalías, la política de coherencia entre puntuación y confianza, el contexto necesario para el futuro entrenamiento de modelos y el seguimiento de las ejecuciones de simulación, así como en la evolución de los propios emisores sintéticos de telemetría hacia una arquitectura de escenarios reutilizable.

2.3.3 SIMULACIÓN

Ante la dificultad de disponer desde el inicio de una fuente de datos de vuelo real con anomalías etiquetadas, se ha optado por desarrollar emisores sintéticos capaces de generar vuelos físicamente plausibles e inyectar de forma controlada distintos tipos de anomalías, siguiendo en todo momento el contrato de telemetría definido. Esta estrategia ha permitido disponer de un conjunto de datos con anomalías conocidas y etiquetadas de antemano, imprescindible tanto para validar el correcto funcionamiento del sistema de ingesta como para preparar, en fases posteriores, el entrenamiento y la evaluación de modelos de detección de anomalías.

2.3.4 VALIDACIÓN

La validación del sistema se ha apoyado en una suite de pruebas automatizadas que comprueba tanto el contrato de los datos generados por los emisores sintéticos como el comportamiento del pipeline completo de ingesta, incluyendo pruebas de carga que simulan condiciones de mayor volumen de tráfico. Estas pruebas se han complementado con validaciones manuales de extremo a extremo sobre el sistema desplegado, comprobando la coherencia de los datos almacenados y el correcto funcionamiento de las métricas expuestas.

Los resultados de esta validación no se han empleado únicamente para confirmar el correcto funcionamiento del sistema, sino también como punto de partida del siguiente ciclo de

diseño e implementación, cerrando así la naturaleza circular de la metodología seguida a lo largo del proyecto (Figura 1).

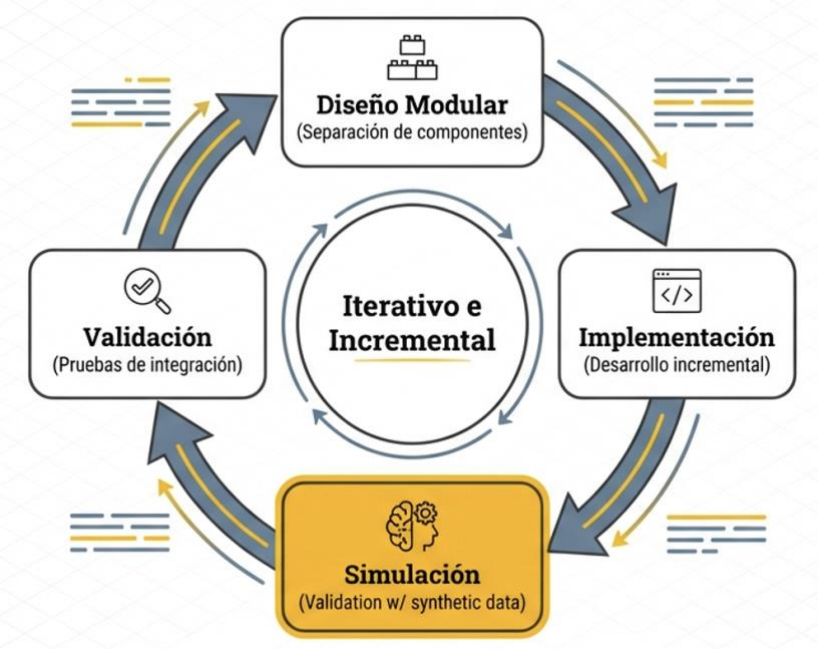

Figura 1. Ciclo iterativo e incremental de la metodología del proyecto

CAPÍTULO 3 – ESTADO DE LA CUESTIÓN

El desarrollo de sistemas capaces de procesar grandes volúmenes de datos se ha convertido en un elemento fundamental en numerosas aplicaciones tecnológicas actuales. En ámbitos como la monitorización de infraestructuras, los sistemas industriales, las plataformas digitales o los entornos de análisis avanzado, es necesario gestionar flujos continuos de información generados por múltiples fuentes y procesarlos de forma eficiente. Este contexto ha impulsado el desarrollo de arquitecturas orientadas a datos que permiten capturar, procesar y analizar información a gran escala. En este sentido, los sistemas modernos se caracterizan por su capacidad para manejar grandes volúmenes de datos, alta velocidad de generación y una gran diversidad de formatos, lo que exige arquitecturas flexibles y escalables capaces de garantizar un procesamiento eficiente [1].

El aumento de la disponibilidad de datos y la necesidad de procesarlos con baja latencia ha favorecido la adopción de arquitecturas basadas en eventos, en las que los sistemas reaccionan de manera inmediata ante la llegada de nueva información. Este enfoque permite diseñar aplicaciones capaces de operar sobre flujos continuos de datos, facilitando la construcción de sistemas de procesamiento en tiempo real. Las arquitecturas orientadas a

eventos han demostrado ser especialmente adecuadas para entornos donde los datos se generan de forma constante y deben procesarse de manera inmediata para apoyar procesos de decisión o monitorización [2].

Paralelamente, el avance de la computación en la nube ha permitido desplegar infraestructuras altamente escalables capaces de gestionar grandes cargas de procesamiento. Las plataformas cloud proporcionan recursos de almacenamiento y computación bajo demanda, lo que facilita el desarrollo de sistemas capaces de adaptarse dinámicamente a variaciones en el volumen de datos procesados. Este paradigma ha contribuido significativamente al desarrollo de plataformas de procesamiento distribuido capaces de operar a gran escala, permitiendo construir sistemas capaces de gestionar flujos de datos provenientes de múltiples fuentes [3].

Además, en los últimos años ha surgido el paradigma de la computación en el borde o edge computing, que busca acercar el procesamiento de datos al lugar donde estos se generan. Este enfoque resulta especialmente relevante en entornos donde la latencia es un factor crítico o donde la transmisión continua de grandes volúmenes de datos hacia centros de datos centrales no resulta eficiente. La combinación de infraestructuras cloud y edge permite diseñar arquitecturas híbridas que equilibran capacidad de procesamiento, latencia y consumo de recursos [4].

Las primeras aproximaciones al procesamiento masivo de datos en sistemas distribuidos se desarrollaron en el contexto del procesamiento por lotes, donde los datos se procesaban en grandes conjuntos mediante técnicas de computación distribuida. Un ejemplo representativo de este enfoque es el modelo MapReduce, que permitió simplificar el procesamiento de grandes conjuntos de datos en clústeres de máquinas distribuidas [5]. Sin embargo, este modelo presenta limitaciones cuando se requiere procesar información de forma continua o con baja latencia.

Para abordar estas limitaciones surgieron nuevas arquitecturas de procesamiento de datos que combinan distintos modelos de análisis. Entre ellas destaca la denominada

arquitectura Lambda, que propone la integración de procesamiento por lotes y procesamiento en tiempo real con el objetivo de equilibrar precisión y latencia en el tratamiento de datos [6]. Posteriormente, se han desarrollado modelos más avanzados orientados específicamente al procesamiento de flujos de datos, como el modelo Data Flow, que proporciona un marco conceptual para gestionar correctamente el equilibrio entre latencia, consistencia y coste en sistemas de procesamiento masivo de datos en streaming [7].

En este contexto, el creciente grado de complejidad de los sistemas distribuidos ha puesto de manifiesto la necesidad de contar con mecanismos avanzados de monitorización y análisis que permitan comprender el comportamiento interno de las plataformas de procesamiento de datos. Los enfoques tradicionales de monitorización han evolucionado hacia el concepto de observabilidad, que busca proporcionar una visión más completa del funcionamiento de los sistemas mediante la recopilación y análisis de diferentes tipos de información operativa [8].

A partir de este contexto tecnológico, este capítulo se organiza en dos grandes bloques. En primer lugar, se revisan los fundamentos técnicos del estado de la cuestión: las principales plataformas de ingesta de datos en tiempo real, los sistemas de monitorización y observabilidad utilizados en arquitecturas distribuidas, las soluciones para la gestión de telemetría en sistemas críticos y las técnicas empleadas para la detección de anomalías en datos generados de forma continua. En segundo lugar, se complementa este análisis técnico con una perspectiva más aplicada, revisando la existencia de productos y plataformas comerciales de telemetría, en primer lugar en el ámbito general de la ingeniería y, a continuación, de forma específica en el sector aeronáutico.

3.1 FUNDAMENTOS TÉCNICOS DE LA TELEMETRÍA Y LA INGESTA DE DATOS

Esta primera parte del capítulo revisa los fundamentos técnicos que sustentan el diseño del sistema desarrollado: las plataformas de ingesta de datos en tiempo real, los sistemas de monitorización y observabilidad empleados en arquitecturas distribuidas, la gestión de telemetría en sistemas críticos y las técnicas utilizadas para la detección de anomalías en datos generados de forma continua.

3.1.1 PLATAFORMAS DE INGESTA DE DATOS EN TIEMPO REAL

Las plataformas de ingesta de datos en tiempo real constituyen uno de los componentes fundamentales en las arquitecturas modernas de procesamiento de información. En numerosos sistemas actuales, los datos se generan de forma continua a partir de múltiples fuentes, como sensores, aplicaciones distribuidas, dispositivos conectados o servicios digitales. Esta naturaleza dinámica de los datos requiere infraestructuras capaces de capturar, transportar y procesar flujos de información con baja latencia, garantizando al mismo tiempo escalabilidad y tolerancia a fallos [1].

El procesamiento de datos en streaming ha surgido como una respuesta a estas necesidades, permitiendo analizar y transformar datos a medida que se generan en lugar de hacerlo únicamente mediante procesos por lotes. A diferencia del procesamiento tradicional basado en grandes conjuntos de datos almacenados previamente, los sistemas de streaming operan sobre flujos continuos de información, lo que permite detectar eventos, generar alertas o realizar análisis en tiempo casi real. Este enfoque resulta especialmente relevante en aplicaciones donde la rapidez en la toma de decisiones es crítica, como la monitorización de sistemas, la analítica operativa o la gestión de infraestructuras tecnológicas [2].

Para soportar este tipo de procesamiento, se han desarrollado diferentes arquitecturas y plataformas especializadas que facilitan la ingesta y el tratamiento de grandes volúmenes

de datos en movimiento. Entre ellas destacan los sistemas de mensajería distribuida y las plataformas de procesamiento de flujos, que permiten desacoplar la generación de datos de su procesamiento y facilitan el diseño de sistemas altamente escalables. Estas tecnologías permiten construir pipelines de datos capaces de gestionar flujos provenientes de múltiples fuentes, garantizando mecanismos de persistencia, replicación y tolerancia a fallos [9].

El desarrollo de estas plataformas ha estado estrechamente ligado a la evolución de los modelos de procesamiento distribuido. Sistemas como Apache Kafka, Apache Flink o Spark Streaming han permitido consolidar arquitecturas capaces de procesar grandes volúmenes de datos con latencias reducidas, combinando escalabilidad horizontal con mecanismos avanzados de gestión de flujos de datos [9][10][11]. Estas soluciones se apoyan en modelos conceptuales que buscan equilibrar aspectos como la consistencia de los resultados, la latencia del procesamiento y el coste computacional asociado al tratamiento de los datos [7].

En este contexto, las plataformas de ingesta y procesamiento de datos en tiempo real se han convertido en un elemento clave dentro de las arquitecturas modernas de análisis de datos. Su desarrollo ha permitido construir sistemas capaces de gestionar flujos de información continuos de forma fiable y eficiente. A continuación, se revisan las principales arquitecturas de procesamiento de datos en streaming, las tecnologías utilizadas para la ingesta de datos y los modelos conceptuales que sustentan este tipo de sistemas.

3.1.1.1 PROCESAMIENTO DE DATOS EN STREAMING

Las arquitecturas de procesamiento de datos en streaming permiten analizar y transformar flujos continuos de información a medida que se generan. A diferencia de los sistemas tradicionales de procesamiento por lotes, en los que los datos se almacenan previamente y se procesan posteriormente en grandes conjuntos, las arquitecturas de streaming trabajan directamente sobre flujos de eventos que llegan de forma constante. Este enfoque permite reducir significativamente la latencia en el análisis de datos y facilita la

construcción de aplicaciones capaces de reaccionar rápidamente ante cambios en el estado de un sistema [1].

El diseño de estas arquitecturas se basa generalmente en modelos de procesamiento distribuido que permiten escalar el sistema horizontalmente mediante la incorporación de nuevos nodos de computación. En este contexto, los sistemas de mensajería distribuida desempeñan un papel fundamental, ya que actúan como intermediarios entre los productores de datos y los componentes encargados de su procesamiento. Un ejemplo representativo de este enfoque es Apache Kafka, que permite gestionar flujos de eventos mediante un sistema de colas distribuidas basado en particiones, proporcionando alta escalabilidad y tolerancia a fallos [9].

En muchas arquitecturas modernas, el procesamiento de datos en streaming se estructura mediante pipelines compuestos por diferentes etapas de transformación. En cada una de estas etapas se aplican operaciones sobre los datos, como filtrado, agregación o enriquecimiento, antes de enviarlos a la siguiente fase del flujo de procesamiento. Este enfoque permite dividir el procesamiento en múltiples componentes independientes, lo que facilita el desarrollo de sistemas modulares y escalables [1].

Uno de los enfoques más conocidos para organizar arquitecturas de procesamiento de datos es la arquitectura Lambda, que combina dos tipos de procesamiento: un procesamiento por lotes que garantiza la exactitud de los resultados y un procesamiento en tiempo real que permite obtener resultados con baja latencia. Este modelo busca equilibrar precisión y velocidad mediante la integración de ambos tipos de procesamiento dentro de una misma arquitectura [6].

Posteriormente, han surgido modelos más avanzados que buscan simplificar esta aproximación. Entre ellos destaca el modelo Dataflow, que propone una arquitectura orientada al procesamiento continuo de datos en streaming. Este modelo introduce conceptos como el manejo de eventos fuera de orden, la definición de ventanas temporales y la gestión

del tiempo de evento y tiempo de procesamiento, aspectos fundamentales para garantizar la consistencia de los resultados en sistemas de procesamiento masivo de datos [7].

Además de estos modelos conceptuales, diversas plataformas han contribuido al desarrollo práctico de arquitecturas de streaming. Sistemas como Spark Streaming introdujeron el modelo de discretized streams (D-Streams), en el que los flujos de datos se procesan como pequeñas secuencias de micro-batches, permitiendo aprovechar infraestructuras de procesamiento distribuido originalmente diseñadas para procesamiento por lotes [10]. Por otro lado, motores de procesamiento como Apache Flink han sido diseñados específicamente para el procesamiento continuo de flujos de datos, proporcionando mecanismos avanzados para la gestión del estado, tolerancia a fallos y procesamiento de eventos en tiempo real [11].

3.1.1.2 ARQUITECTURAS ORIENTADAS A EVENTOS PARA LA INGESTA DE DATOS

Las plataformas de ingesta de datos constituyen el primer componente dentro de las arquitecturas de procesamiento de datos en tiempo real. Su función principal consiste en capturar los datos generados por diferentes fuentes y transportarlos de forma fiable hacia los sistemas encargados de su procesamiento y almacenamiento. Estas plataformas deben ser capaces de manejar grandes volúmenes de información, garantizar la entrega de mensajes y mantener un alto grado de escalabilidad, ya que en muchos casos los datos se generan de forma continua y a gran velocidad [1].

Una de las tecnologías más extendidas en este ámbito son los sistemas de mensajería distribuida, que permiten desacoplar los componentes productores de datos de los sistemas encargados de procesarlos. En este tipo de arquitecturas, los productores publican eventos o mensajes en una plataforma intermedia, mientras que los consumidores pueden leer estos datos de forma independiente. Este modelo facilita el desarrollo de sistemas distribuidos escalables y permite que múltiples aplicaciones puedan consumir simultáneamente los mismos flujos de datos [9].

Entre las plataformas más utilizadas para la ingesta de datos en tiempo real destaca Apache Kafka, un sistema de mensajería distribuido diseñado para gestionar grandes volúmenes de eventos mediante una arquitectura basada en particiones y replicación. Kafka permite almacenar flujos de eventos de forma persistente y distribuirlos entre múltiples consumidores, lo que lo convierte en una pieza central en muchas arquitecturas modernas de procesamiento de datos [9]. Su diseño permite manejar altas tasas de escritura y lectura, manteniendo al mismo tiempo mecanismos de tolerancia a fallos y escalabilidad horizontal.

Además de Kafka, existen otras plataformas diseñadas específicamente para el procesamiento de flujos de datos en tiempo real. Apache Storm es uno de los primeros sistemas desarrollados para el procesamiento distribuido de streams, permitiendo definir topologías de procesamiento compuestas por diferentes nodos que ejecutan operaciones sobre los flujos de datos entrantes [12]. Este tipo de sistemas facilita la construcción de pipelines de procesamiento donde los datos pueden ser transformados, filtrados o agregados a medida que circulan por la arquitectura.

En entornos basados en computación en la nube también han surgido soluciones gestionadas que simplifican el despliegue y la operación de infraestructuras de ingesta de datos. Un ejemplo de ello es Amazon Kinesis Data Streams, que permite capturar y procesar grandes cantidades de datos en tiempo real mediante una infraestructura escalable gestionada por el proveedor cloud. Este tipo de servicios facilita la integración de aplicaciones distribuidas con sistemas de análisis y almacenamiento de datos en la nube [13].

La elección de una plataforma de ingesta depende de diversos factores, como el volumen de datos generado, los requisitos de latencia, la necesidad de persistencia de los mensajes o el nivel de integración con otras tecnologías dentro de la arquitectura. En muchos sistemas modernos, estas plataformas se combinan con motores de procesamiento de streaming y bases de datos especializadas para construir pipelines completos de procesamiento de datos en tiempo real capaces de soportar aplicaciones de monitorización, analítica operativa o procesamiento de telemetría [1][11].

3.1.1.3 PLATAFORMAS Y TECNOLOGÍAS PARA INGESTIÓN DE DATOS EN TIEMPO REAL

El procesamiento de flujos de datos se apoya en diferentes modelos conceptuales que permiten organizar y gestionar de manera eficiente la ejecución de operaciones sobre datos que llegan de forma continua. Estos modelos definen cómo se representan los flujos de datos, cómo se estructuran las operaciones de transformación y cómo se gestionan aspectos fundamentales como la latencia, la consistencia de los resultados o la tolerancia a fallos. En el contexto de los sistemas modernos de procesamiento distribuido, estos modelos constituyen la base sobre la que se diseñan las plataformas de streaming y los motores de procesamiento de eventos [1].

Uno de los enfoques más influyentes en el desarrollo de sistemas de procesamiento de datos en tiempo real es el modelo Dataflow. Este modelo describe el procesamiento como un grafo dirigido de operaciones en el que los datos fluyen entre diferentes nodos que ejecutan transformaciones sobre ellos. El modelo introduce además conceptos clave como el tiempo de evento, el tiempo de procesamiento y la gestión de ventanas temporales, que permiten tratar adecuadamente flujos de datos que pueden llegar desordenados o con retrasos. Este enfoque proporciona un marco conceptual que permite equilibrar precisión, latencia y coste computacional en sistemas de procesamiento masivo de datos [7].

Otra aproximación relevante al procesamiento de flujos de datos es el modelo de discretized streams o D-Streams, utilizado en sistemas como Spark Streaming. En este modelo, los flujos continuos de datos se dividen en pequeños micro-batches que se procesan de forma periódica mediante operaciones distribuidas. Esta estrategia permite aprovechar infraestructuras originalmente diseñadas para procesamiento por lotes, adaptándolas al tratamiento de datos en tiempo casi real. Aunque este enfoque introduce una ligera latencia adicional asociada al tamaño de los micro-batches, facilita la reutilización de mecanismos de procesamiento distribuido ya existentes [10].

El desarrollo de arquitecturas de streaming también ha estado influido por modelos híbridos que combinan procesamiento por lotes y procesamiento en tiempo real. Un ejemplo

de ello es la arquitectura Lambda, que propone la coexistencia de una capa de procesamiento batch para garantizar la precisión de los resultados y una capa de procesamiento en streaming que proporciona respuestas rápidas ante la llegada de nuevos datos. Este enfoque busca resolver las limitaciones de los sistemas puramente batch cuando se requiere baja latencia en el análisis de datos [6].

Desde una perspectiva más general, los sistemas de procesamiento de datos modernos suelen organizarse como pipelines compuestos por diferentes etapas de transformación. Cada etapa realiza operaciones específicas sobre los datos, como filtrado, agregación o enriquecimiento, y transmite los resultados a la siguiente fase del flujo de procesamiento. Este modelo de procesamiento en pipeline permite construir arquitecturas modulares y escalables, en las que cada componente puede evolucionar de manera independiente y adaptarse a cambios en los requisitos del sistema [1].

Además, en el ámbito del procesamiento de grandes flujos de datos han surgido técnicas destinadas a reducir el consumo de memoria y el coste computacional asociado al análisis continuo de información. Entre estas técnicas destacan los algoritmos de resumen de flujos de datos o stream summarization, que permiten mantener estimaciones aproximadas de ciertas métricas sin necesidad de almacenar todos los datos generados. Un ejemplo representativo es el algoritmo Count-Min Sketch, que permite estimar frecuencias de elementos en grandes flujos de datos utilizando estructuras de datos compactas y eficientes [15].

Estos modelos y técnicas constituyen la base teórica sobre la que se desarrollan los sistemas modernos de procesamiento de datos en streaming. Su evolución ha permitido diseñar arquitecturas capaces de manejar grandes volúmenes de datos con latencias reducidas, garantizando al mismo tiempo escalabilidad y fiabilidad en entornos distribuidos [5].

3.1.1.4 RETOS EN LA INGESTA DE TELEMETRÍA

El desarrollo de sistemas capaces de procesar datos en tiempo real plantea diversos desafíos técnicos relacionados con la escalabilidad, la latencia, la consistencia de los resultados y la tolerancia a fallos. A diferencia de los sistemas tradicionales de procesamiento por lotes, en los que los datos pueden analizarse sin restricciones estrictas de tiempo, los sistemas de procesamiento en streaming deben operar continuamente sobre flujos de información que llegan de manera constante. Esta característica introduce una serie de complejidades en el diseño de las arquitecturas y en la gestión del procesamiento distribuido [1].

Uno de los principales retos en este tipo de sistemas es garantizar una baja latencia en el procesamiento de los datos. En muchas aplicaciones, como la monitorización de infraestructuras, la detección de anomalías o la analítica operativa, es necesario que los resultados se generen con un retraso mínimo desde el momento en que se produce un evento. Para ello, las arquitecturas de procesamiento deben optimizar el flujo de datos a través de los diferentes componentes del sistema, evitando cuellos de botella y garantizando una distribución eficiente de la carga de procesamiento entre los distintos nodos del sistema [7].

La escalabilidad constituye otro aspecto fundamental en los sistemas de procesamiento en tiempo real. A medida que aumenta el volumen de datos generado por las fuentes de información, la infraestructura debe ser capaz de adaptarse mediante la incorporación de nuevos recursos de computación. Las plataformas modernas de procesamiento de streaming permiten escalar horizontalmente mediante la distribución del procesamiento entre múltiples nodos, lo que facilita la gestión de grandes volúmenes de datos sin comprometer el rendimiento del sistema [11].

Además, la tolerancia a fallos representa un requisito esencial en arquitecturas distribuidas. En entornos donde múltiples componentes trabajan de forma coordinada, la caída de uno de los nodos puede afectar al procesamiento de los flujos de datos. Por este motivo, los sistemas de streaming incorporan mecanismos de replicación, persistencia de

datos y recuperación del estado que permiten continuar el procesamiento incluso cuando se producen fallos en algunos componentes de la infraestructura [1].

Otro desafío importante está relacionado con la correcta gestión del tiempo en el procesamiento de flujos de datos. En muchos sistemas distribuidos, los eventos pueden llegar fuera de orden debido a retrasos en la red o diferencias en los tiempos de generación de los datos. Esto obliga a los sistemas de procesamiento a implementar mecanismos que permitan gestionar adecuadamente el tiempo de evento y el tiempo de procesamiento, garantizando la coherencia de los resultados obtenidos a partir de los flujos de datos [7].

Por último, el crecimiento de las infraestructuras basadas en computación en la nube y en el borde ha introducido nuevos desafíos en la gestión de sistemas de procesamiento en tiempo real. La integración de fuentes de datos distribuidas geográficamente, junto con la necesidad de reducir la latencia en determinadas aplicaciones, ha impulsado el desarrollo de arquitecturas híbridas que combinan recursos de procesamiento en centros de datos cloud con capacidades de procesamiento cercanas al origen de los datos. Estas arquitecturas permiten optimizar el uso de recursos y mejorar los tiempos de respuesta en aplicaciones sensibles a la latencia [3][4].

3.1.2 SISTEMAS DE MONITORIZACIÓN Y OBSERVABILIDAD

El crecimiento de los sistemas distribuidos y de las arquitecturas basadas en microservicios ha incrementado significativamente la complejidad en la gestión y operación de infraestructuras tecnológicas. En este tipo de entornos, las aplicaciones suelen estar compuestas por múltiples componentes que interactúan entre sí a través de redes y servicios intermedios. Esta complejidad hace que comprender el comportamiento interno del sistema y detectar posibles fallos resulte cada vez más difícil, lo que ha impulsado el desarrollo de herramientas y metodologías avanzadas de monitorización y análisis operativo [1].

Tradicionalmente, los sistemas de monitorización se han centrado en la recopilación de métricas relacionadas con el rendimiento de la infraestructura, como el uso de CPU,

memoria o tráfico de red. Estas métricas permitían detectar problemas básicos de rendimiento o disponibilidad, pero resultaban insuficientes para analizar el comportamiento de sistemas distribuidos complejos en los que múltiples servicios interactúan de manera dinámica. Como consecuencia, han surgido nuevos enfoques que amplían el alcance de la monitorización tradicional y permiten obtener una visión más completa del estado de los sistemas [8].

En este contexto, el concepto de observabilidad ha ganado relevancia como una evolución de las prácticas de monitorización. La observabilidad se refiere a la capacidad de comprender el estado interno de un sistema a partir de la información que este genera durante su funcionamiento. Este enfoque se basa en la recopilación y análisis de diferentes tipos de datos operativos, como métricas, registros de eventos y trazas distribuidas, que permiten analizar el comportamiento del sistema y diagnosticar problemas de manera más eficaz [8].

Para implementar estas capacidades, han surgido diversas herramientas y plataformas especializadas en la recopilación, almacenamiento y visualización de datos operativos. Sistemas como Prometheus permiten almacenar métricas en forma de series temporales y realizar consultas sobre su evolución a lo largo del tiempo, facilitando la monitorización del rendimiento de los sistemas [17]. Por su parte, plataformas como Grafana proporcionan interfaces de visualización que permiten analizar estos datos mediante paneles interactivos y herramientas de exploración gráfica [18].

Además de las métricas tradicionales, las trazas distribuidas se han convertido en un elemento fundamental para comprender el funcionamiento de sistemas complejos. Sistemas de trazado como Dapper permiten seguir el recorrido de una solicitud a través de múltiples servicios dentro de una arquitectura distribuida, proporcionando información detallada sobre el tiempo empleado en cada etapa del procesamiento y facilitando la identificación de cuellos de botella o fallos en el sistema [16].

Asimismo, algunas organizaciones han desarrollado sistemas de telemetría a gran escala para monitorizar infraestructuras distribuidas en entornos de producción. Un ejemplo

de ello es el sistema Atlas desarrollado por Netflix, que permite recopilar y analizar grandes volúmenes de métricas operativas generadas por miles de servicios distribuidos [19]. Estas soluciones demuestran la importancia de disponer de infraestructuras de observabilidad robustas para garantizar la fiabilidad y el correcto funcionamiento de sistemas de gran escala.

3.1.2.1 EVOLUCIÓN DE LA MONITORIZACIÓN HACIA LA OBSERVABILIDAD

La monitorización de sistemas informáticos ha sido tradicionalmente una práctica orientada a supervisar el estado y el rendimiento de los componentes de una infraestructura tecnológica. En sus primeras etapas, las herramientas de monitorización se centraban en la recopilación de métricas básicas relacionadas con el uso de recursos del sistema, como la utilización de CPU, memoria, almacenamiento o tráfico de red. Estas métricas permitían detectar problemas evidentes de rendimiento o disponibilidad, pero ofrecían una visión limitada del comportamiento interno de sistemas cada vez más complejos [1].

Con la evolución de las arquitecturas de software hacia sistemas distribuidos y entornos basados en microservicios, las limitaciones de los enfoques tradicionales de monitorización se hicieron más evidentes. En este tipo de sistemas, una única operación puede implicar la interacción de múltiples servicios distribuidos, lo que dificulta la identificación de la causa de un fallo o de un problema de rendimiento utilizando únicamente métricas de infraestructura. Como consecuencia, surgió la necesidad de adoptar nuevas estrategias que permitieran comprender el funcionamiento interno de estos sistemas de manera más completa [8].

En este contexto aparece el concepto de observabilidad, que amplía el enfoque de la monitorización tradicional. La observabilidad se define como la capacidad de inferir el estado interno de un sistema a partir de la información que este genera durante su funcionamiento. A diferencia de la monitorización clásica, que suele basarse en indicadores predefinidos, la observabilidad permite explorar el comportamiento del sistema incluso en situaciones no previstas inicialmente, facilitando el diagnóstico de problemas complejos [8].

Uno de los avances clave que impulsaron esta transición fue el desarrollo de sistemas de trazado distribuido. Estos sistemas permiten seguir el recorrido de una solicitud a través de múltiples servicios dentro de una arquitectura distribuida, proporcionando información detallada sobre las interacciones entre componentes. Un ejemplo destacado es el sistema Dapper desarrollado por Google, que introdujo un modelo de trazado distribuido capaz de capturar información sobre las diferentes etapas del procesamiento de una solicitud en sistemas a gran escala [16].

3.1.2.2 MÉTRICAS, LOGS Y TRAZAS EN SISTEMAS DISTRIBUIDOS

En los sistemas distribuidos modernos, la observabilidad se apoya principalmente en tres tipos fundamentales de datos operativos: métricas, logs y trazas. Estos tres elementos proporcionan diferentes perspectivas sobre el comportamiento de un sistema y, cuando se utilizan de forma conjunta, permiten obtener una visión más completa del estado y funcionamiento de las aplicaciones. La combinación de estas fuentes de información facilita la detección de errores, el análisis del rendimiento y la identificación de problemas en entornos complejos [8].

Las métricas representan medidas cuantitativas del estado de un sistema a lo largo del tiempo. Normalmente se almacenan como series temporales y permiten analizar la evolución de variables como el uso de recursos, la latencia de las operaciones o el número de solicitudes procesadas por un servicio. Este tipo de información resulta especialmente útil para detectar anomalías en el comportamiento del sistema o identificar tendencias que puedan indicar problemas de rendimiento. En muchas infraestructuras modernas, las métricas se recopilan y almacenan mediante sistemas especializados de monitorización diseñados para gestionar grandes volúmenes de datos temporales [17].

Los logs, por su parte, consisten en registros detallados de eventos generados por los diferentes componentes de una aplicación. Estos registros suelen incluir información textual sobre las operaciones realizadas por el sistema, mensajes de error, advertencias o información de depuración. A diferencia de las métricas, que ofrecen una visión agregada

del comportamiento del sistema, los logs proporcionan información más detallada sobre eventos específicos, lo que resulta útil para investigar problemas concretos o analizar el comportamiento de determinadas operaciones dentro de una aplicación.

Las trazas distribuidas constituyen el tercer elemento clave dentro de los sistemas de observabilidad. Este tipo de información permite seguir el recorrido completo de una solicitud a través de los diferentes servicios que componen una arquitectura distribuida. Cada vez que una solicitud pasa por un componente del sistema, se genera un registro que describe el tiempo empleado en esa etapa del procesamiento. Al combinar todos estos registros es posible reconstruir el flujo completo de ejecución de una operación, lo que facilita la identificación de cuellos de botella y problemas de rendimiento en sistemas compuestos por múltiples servicios interconectados. El sistema Dapper desarrollado por Google es uno de los ejemplos más conocidos de infraestructura de trazado distribuido en entornos de gran escala [16].

Para facilitar el análisis de estos datos operativos, se utilizan plataformas especializadas que permiten almacenar, consultar y visualizar la información recopilada. Sistemas de monitorización como Prometheus están diseñados para gestionar grandes volúmenes de métricas en forma de series temporales, mientras que herramientas de visualización como Grafana permiten representar estos datos mediante paneles interactivos que facilitan el análisis del estado del sistema y la detección de posibles problemas [17][18].

3.1.2.3 HERRAMIENTAS Y PLATAFORMAS DE OBSERVABILIDAD

La implementación práctica de sistemas de observabilidad requiere el uso de herramientas capaces de recopilar, almacenar, analizar y visualizar grandes volúmenes de datos operativos generados por las aplicaciones y la infraestructura. Estas plataformas permiten centralizar información procedente de diferentes fuentes, como métricas, logs y trazas distribuidas, facilitando el análisis del comportamiento de los sistemas y la detección de problemas en entornos complejos [17].

Una de las herramientas más utilizadas en la monitorización de sistemas distribuidos es Prometheus. Este sistema está diseñado para recopilar y almacenar métricas en forma de series temporales, permitiendo realizar consultas sobre la evolución de diferentes indicadores de rendimiento. Prometheus emplea un modelo de recopilación basado en scrapping, mediante el cual obtiene periódicamente métricas expuestas por los distintos servicios del sistema. Además, proporciona un lenguaje de consulta especializado que permite analizar y agregar datos de monitorización de forma flexible [17].

Para complementar el análisis de métricas, es habitual utilizar herramientas de visualización que permitan representar gráficamente la información recopilada. Grafana es una de las plataformas más extendidas en este ámbito, ya que permite crear paneles interactivos que facilitan la exploración y el análisis de datos de monitorización. Estos paneles pueden integrar información procedente de múltiples fuentes de datos, lo que permite visualizar de forma conjunta diferentes métricas y analizar el comportamiento del sistema en tiempo real [18].

Además de las métricas, muchas plataformas de observabilidad incorporan sistemas de trazado distribuido que permiten analizar el recorrido de las solicitudes a través de los distintos componentes de una arquitectura distribuida. Estas herramientas resultan especialmente útiles en entornos basados en microservicios, donde una única operación puede implicar la interacción de múltiples servicios independientes. El sistema Dapper desarrollado por Google introdujo uno de los primeros modelos de trazado distribuido a gran escala, permitiendo analizar el tiempo empleado en cada etapa del procesamiento de una solicitud y facilitando la identificación de cuellos de botella en sistemas complejos [16].

3.1.3 GESTIÓN DE TELEMETRÍA EN SISTEMAS CRÍTICOS

La telemetría desempeña un papel fundamental en la monitorización y operación de sistemas complejos, especialmente en aquellos entornos donde la fiabilidad, la disponibilidad y la capacidad de diagnóstico son factores críticos. En estos sistemas, la recopilación continua de datos operativos permite conocer el estado de los distintos

componentes de la infraestructura, detectar posibles anomalías y facilitar la toma de decisiones en tiempo real. La telemetría se utiliza en una amplia variedad de ámbitos, incluyendo infraestructuras tecnológicas, sistemas industriales, plataformas cloud o aplicaciones distribuidas de gran escala [1].

Los sistemas modernos generan grandes cantidades de datos de telemetría procedentes de múltiples fuentes, como sensores, aplicaciones software, dispositivos conectados o componentes de infraestructura. Esta información suele incluir métricas de rendimiento, eventos de funcionamiento o datos sobre el estado interno de los sistemas. La correcta gestión de estos datos requiere infraestructuras capaces de capturar, almacenar y procesar flujos continuos de información sin comprometer el rendimiento ni la disponibilidad del sistema [3].

Además del procesamiento de datos de monitorización, la telemetría suele requerir sistemas de almacenamiento capaces de gestionar grandes volúmenes de datos de series temporales generados continuamente. Las bases de datos especializadas en este tipo de información permiten almacenar métricas y eventos operativos de manera eficiente, facilitando su consulta y análisis posterior. Un ejemplo de este tipo de soluciones es Apache IoTDB, una base de datos diseñada específicamente para gestionar grandes cantidades de datos temporales generados por dispositivos y sistemas conectados [14].

Por otra parte, el crecimiento de infraestructuras distribuidas y el uso combinado de recursos cloud y edge han ampliado el alcance de los sistemas de telemetría. En muchas aplicaciones, los datos pueden generarse en ubicaciones distribuidas geográficamente y requerir procesamiento cercano al origen para reducir la latencia o el volumen de información transmitido. Este tipo de arquitecturas híbridas plantea nuevos desafíos en la gestión y análisis de datos operativos, ya que requiere coordinar múltiples fuentes de información distribuidas en diferentes niveles de la infraestructura [4].

En este contexto, la gestión eficiente de la telemetría se ha convertido en un elemento clave para garantizar la operación fiable de sistemas críticos. La capacidad de recopilar,

almacenar y analizar datos operativos permite mejorar la monitorización del sistema, optimizar el rendimiento de las aplicaciones y facilitar la detección temprana de posibles fallos o comportamientos anómalos. En las siguientes subsecciones se analizan con mayor detalle las características de la telemetría en sistemas críticos y las arquitecturas utilizadas para su almacenamiento y gestión.

3.1.3.1 CARACTERÍSTICAS Y REQUISITOS DE LA TELEMETRÍA EN SISTEMAS CRÍTICOS

En los sistemas críticos, la telemetría desempeña un papel esencial para garantizar el correcto funcionamiento de la infraestructura y permitir una supervisión continua del estado de los distintos componentes del sistema. Este tipo de sistemas suele operar en entornos donde la fiabilidad y la disponibilidad son factores fundamentales, por lo que resulta necesario disponer de mecanismos que permitan recopilar información operativa de manera constante y analizarla con rapidez para detectar posibles fallos o comportamientos anómalos [1].

Una de las características principales de la telemetría en sistemas críticos es la generación continua de grandes volúmenes de datos. Los diferentes componentes de la infraestructura, como aplicaciones, servicios o dispositivos conectados, producen de forma constante métricas, eventos y registros que reflejan su estado de funcionamiento. La gestión eficiente de estos datos requiere sistemas capaces de procesar flujos de información de alta frecuencia sin afectar al rendimiento del sistema monitorizado [3].

Otro requisito fundamental es la capacidad de procesar la información con baja latencia. En muchos sistemas críticos, la detección temprana de problemas resulta esencial para evitar fallos de mayor impacto. Por este motivo, los sistemas de telemetría deben ser capaces de recopilar y analizar los datos generados por la infraestructura prácticamente en tiempo real, permitiendo identificar anomalías o degradaciones en el rendimiento del sistema de forma inmediata [2].

Además, en muchas aplicaciones modernas los datos de telemetría se generan en entornos distribuidos geográficamente, lo que introduce nuevos retos en su recopilación y procesamiento. La incorporación de infraestructuras basadas en edge computing permite realizar parte del procesamiento cerca del origen de los datos, reduciendo la latencia y optimizando el uso de recursos de red. Este enfoque resulta especialmente útil en sistemas donde la transmisión continua de grandes volúmenes de información hacia centros de datos centrales puede resultar ineficiente o introducir retrasos en el análisis de los datos [4].

3.1.3.2 ARQUITECTURAS PARA ALMACENAMIENTO Y GESTIÓN DE DATOS DE TELEMETRÍA

La gestión de datos de telemetría requiere infraestructuras capaces de almacenar y procesar grandes volúmenes de información generados de forma continua por múltiples fuentes. En muchos sistemas críticos, los datos de telemetría se generan con alta frecuencia y deben conservarse durante largos periodos para permitir su análisis histórico, la detección de anomalías o la evaluación del rendimiento del sistema. Por este motivo, las arquitecturas utilizadas para gestionar estos datos deben garantizar escalabilidad, fiabilidad y eficiencia en el acceso a la información [1].

Una de las características principales de los datos de telemetría es su naturaleza temporal. La mayoría de la información recopilada consiste en métricas que describen el estado del sistema en un momento determinado, lo que da lugar a grandes conjuntos de datos organizados como series temporales. Para gestionar este tipo de información de manera eficiente, se han desarrollado bases de datos especializadas en el almacenamiento de series temporales, diseñadas para manejar grandes volúmenes de datos generados de forma continua. Un ejemplo de este tipo de soluciones es Apache IoTDB, una base de datos optimizada para el almacenamiento y consulta de datos temporales generados por dispositivos y sistemas conectados [14].

En muchas arquitecturas modernas, los datos de telemetría se integran dentro de pipelines de procesamiento que permiten capturar, almacenar y analizar la información de forma continua. Estos pipelines suelen estar compuestos por sistemas de ingesta de datos,

motores de procesamiento y sistemas de almacenamiento diseñados para gestionar grandes volúmenes de información. Este enfoque permite construir plataformas capaces de analizar datos operativos en tiempo real y mantener al mismo tiempo históricos de información para análisis posteriores [1].

Otro aspecto relevante en estas arquitecturas es la integración con infraestructuras de computación en la nube. Las plataformas cloud permiten desplegar sistemas de almacenamiento distribuidos capaces de adaptarse dinámicamente al crecimiento del volumen de datos generado por la infraestructura monitorizada. Este enfoque facilita la construcción de sistemas de telemetría escalables, en los que los recursos de almacenamiento y procesamiento pueden ampliarse según las necesidades del sistema [3].

Asimismo, en algunos casos los datos de telemetría deben integrarse con sistemas de almacenamiento relacional o plataformas de análisis que permiten realizar consultas complejas sobre la información recopilada. Las bases de datos distribuidas diseñadas para entornos cloud, como Amazon Aurora, han introducido arquitecturas que separan los componentes de almacenamiento y procesamiento, permitiendo mejorar el rendimiento y la escalabilidad en sistemas que manejan grandes volúmenes de datos [20].

3.1.3.3 TELEMETRÍA AERONÁUTICA Y VARIABLES RELEVANTES PARA LA MONITORIZACIÓN

En el ámbito aeronáutico, la telemetría desempeña un papel fundamental en la supervisión del estado y del comportamiento de las aeronaves durante el vuelo. A través de distintos sistemas de comunicación y vigilancia, es posible obtener información sobre la posición, la trayectoria y diversos parámetros operativos de las aeronaves en tiempo casi real. Estos datos constituyen la base para múltiples aplicaciones, entre las que se incluyen la gestión del tráfico aéreo, el seguimiento de vuelos, el análisis de trayectorias y la detección de comportamientos anómalos [33], [34].

Uno de los mecanismos más utilizados para la obtención de información de vuelo es el sistema Automatic Dependent Surveillance–Broadcast (ADS-B). Este sistema permite que las aeronaves transmitan periódicamente información sobre su posición y estado mediante señales emitidas por el transpondedor a bordo. Dichas señales pueden ser recibidas por estaciones terrestres o por otros sistemas aeronáuticos, lo que facilita la construcción de redes de vigilancia aérea distribuidas capaces de proporcionar información detallada sobre el tráfico aéreo [33], [35].

Los datos transmitidos a través de sistemas como ADS-B incluyen un conjunto de variables que describen tanto la posición como el comportamiento dinámico de la aeronave. Entre las variables más relevantes se encuentran la latitud y longitud, que permiten determinar la posición geográfica del avión; la altitud, que indica su posición vertical en el espacio aéreo; y la velocidad sobre el suelo, que describe la rapidez con la que la aeronave se desplaza respecto al terreno. Asimismo, otras variables como el rumbo o heading proporcionan información sobre la dirección de desplazamiento del avión, mientras que parámetros como la velocidad vertical permiten analizar ascensos o descensos durante el vuelo [33], [34].

Además de estas variables relacionadas con la navegación, los sistemas de telemetría aeronáutica suelen incluir información adicional que permite identificar y contextualizar el vuelo. Por ejemplo, es habitual que los mensajes transmitidos incorporen identificadores únicos de aeronave, códigos de vuelo o marcas temporales que facilitan la reconstrucción de la trayectoria completa de la aeronave a lo largo del tiempo. Esta información resulta especialmente útil para el análisis histórico de vuelos y para la correlación de datos procedentes de distintas fuentes [34].

El análisis conjunto de estas variables permite detectar patrones de comportamiento que pueden indicar situaciones anómalas o potencialmente problemáticas. Por ejemplo, cambios bruscos en la altitud, desviaciones significativas en el rumbo previsto o variaciones inesperadas en la velocidad pueden ser indicativos de incidencias técnicas, condiciones

meteorológicas adversas o maniobras fuera de lo habitual. De este modo, la telemetría aeronáutica proporciona un conjunto de datos estructurados que resulta especialmente adecuado para la aplicación de técnicas de monitorización y detección de anomalías.

En consecuencia, la disponibilidad de este tipo de información abre la posibilidad de desarrollar plataformas capaces de recoger, almacenar y analizar datos de vuelo de forma sistemática. Estas plataformas pueden aprovechar los datos transmitidos por sistemas de vigilancia aérea para construir modelos que permitan caracterizar el comportamiento normal de las aeronaves y detectar desviaciones respecto a dicho comportamiento. Por ello, el estudio de las variables presentes en la telemetría aeronáutica constituye un elemento clave en el diseño de sistemas orientados a la monitorización y análisis del tráfico aéreo.

3.1.4 TÉCNICAS DE DETECCIÓN DE ANOMALÍAS

La detección de anomalías constituye una técnica fundamental en el análisis de datos generados por sistemas complejos. En numerosos ámbitos tecnológicos, como la monitorización de infraestructuras, la seguridad informática, los sistemas industriales o las plataformas de análisis de datos, resulta necesario identificar comportamientos que se desvían significativamente del funcionamiento normal del sistema. Estos comportamientos anómalos pueden indicar fallos técnicos, errores en los datos, degradaciones del rendimiento o incluso actividades maliciosas, por lo que su detección temprana resulta esencial para garantizar la fiabilidad y seguridad de los sistemas [21].

El análisis de anomalías ha sido ampliamente estudiado en el ámbito de la minería de datos y el aprendizaje automático. Diversos trabajos han propuesto métodos para identificar observaciones atípicas en conjuntos de datos mediante el análisis de patrones estadísticos, distribuciones de probabilidad o relaciones entre variables. Estos enfoques permiten detectar valores que difieren significativamente del comportamiento esperado del sistema, facilitando la identificación de posibles incidencias o irregularidades en los datos [21][22].

En los sistemas modernos de procesamiento de datos, la detección de anomalías se aplica con frecuencia sobre grandes volúmenes de información generados de forma continua. En este contexto, el análisis debe realizarse de manera eficiente y, en muchos casos, en tiempo real. Esto ha impulsado el desarrollo de técnicas capaces de operar sobre flujos de datos o series temporales, permitiendo identificar cambios inesperados en el comportamiento de las métricas monitorizadas [31].

Las técnicas de detección de anomalías pueden clasificarse en diferentes categorías según el enfoque utilizado. Entre las aproximaciones más comunes se encuentran los métodos estadísticos tradicionales, los algoritmos basados en aprendizaje automático y las técnicas especializadas en el análisis de series temporales. Cada uno de estos enfoques presenta ventajas y limitaciones dependiendo del tipo de datos analizados, del volumen de información disponible y de los requisitos del sistema en el que se aplican [22].

En entornos donde los datos se generan de forma continua, como en sistemas de monitorización o telemetría, las anomalías suelen manifestarse como cambios abruptos en la evolución temporal de ciertas métricas. Por este motivo, el análisis de series temporales se ha convertido en una herramienta clave para detectar comportamientos anómalos en sistemas distribuidos. Diversos métodos han sido desarrollados para identificar patrones inusuales en datos temporales, combinando técnicas estadísticas con algoritmos de aprendizaje automático capaces de adaptarse al comportamiento dinámico de los sistemas monitorizados [27][28][29].

En las siguientes subsecciones se analizan con mayor detalle los principales enfoques utilizados para la detección de anomalías. En primer lugar, se revisan los métodos estadísticos tradicionales utilizados para identificar valores atípicos en conjuntos de datos. Posteriormente, se describen las técnicas basadas en aprendizaje automático y, finalmente, se presentan los métodos específicos utilizados para detectar anomalías en series temporales.

3.1.4.1 MÉTODOS ESTADÍSTICOS PARA DETECCIÓN DE ANOMALÍAS

Los métodos estadísticos constituyen una de las aproximaciones más tradicionales para la detección de anomalías en conjuntos de datos. Estos enfoques se basan en el análisis de las propiedades estadísticas de los datos con el objetivo de identificar observaciones que se desvían significativamente del comportamiento esperado. En general, una anomalía se define como un valor que presenta una probabilidad muy baja de ocurrir según el modelo estadístico que describe el comportamiento normal del sistema [21].

Uno de los enfoques más comunes consiste en modelar la distribución estadística de los datos y detectar valores que se encuentran fuera de determinados umbrales definidos a partir de parámetros como la media y la desviación estándar. Por ejemplo, en distribuciones aproximadamente normales, es habitual considerar como anómalos aquellos valores que se encuentran a más de tres desviaciones estándar de la media. Este tipo de técnicas se utilizan con frecuencia en el control estadístico de procesos y en la supervisión de sistemas industriales, donde permiten identificar desviaciones respecto al comportamiento esperado del sistema [23].

Otra categoría de métodos estadísticos se basa en el análisis de densidad o en la estimación de la probabilidad de los datos. En estos enfoques, las observaciones que se encuentran en regiones de baja densidad dentro del espacio de datos se consideran potenciales anomalías. Estos métodos resultan especialmente útiles cuando el comportamiento normal del sistema puede representarse mediante modelos probabilísticos que permiten estimar la distribución de los datos observados [21].

En el caso de datos que evolucionan a lo largo del tiempo, los métodos estadísticos también pueden aplicarse mediante el análisis de series temporales. En este contexto, las anomalías suelen identificarse como desviaciones significativas respecto a patrones temporales esperados, como tendencias o comportamientos estacionales. Diversos estudios han analizado técnicas estadísticas para detectar valores atípicos en datos temporales,

teniendo en cuenta la dependencia entre observaciones consecutivas y la evolución del sistema a lo largo del tiempo [22].

Además, algunos métodos combinan técnicas estadísticas con modelos adaptativos que permiten ajustar los parámetros del sistema a medida que se reciben nuevos datos. Este tipo de enfoques resulta especialmente útil en entornos donde el comportamiento del sistema puede cambiar con el tiempo. En estos casos, los modelos deben ser capaces de actualizarse dinámicamente para mantener la capacidad de detectar desviaciones significativas en el comportamiento del sistema [28].

3.1.4.2 TÉCNICAS BASADAS EN APRENDIZAJE AUTOMÁTICO

El uso de técnicas basadas en aprendizaje automático ha adquirido una gran relevancia en la detección de anomalías, especialmente en entornos donde los datos presentan patrones complejos o no lineales que resultan difíciles de modelar mediante métodos estadísticos tradicionales. Estos enfoques permiten construir modelos capaces de aprender el comportamiento normal de un sistema a partir de los datos observados y detectar posteriormente desviaciones significativas respecto a dicho comportamiento [21].

Uno de los enfoques más utilizados dentro de esta categoría es el aprendizaje no supervisado, en el que el modelo se entrena únicamente con los datos disponibles sin necesidad de disponer de etiquetas que indiquen si una observación es normal o anómala. Este enfoque resulta especialmente útil en escenarios reales, donde la disponibilidad de datos etiquetados suele ser limitada. Entre los algoritmos desarrollados bajo este paradigma destaca Isolation Forest, un método diseñado para identificar anomalías mediante la construcción de árboles de partición aleatorios que aíslan rápidamente las observaciones atípicas dentro del conjunto de datos [24].

Otra línea de investigación relevante se basa en el uso de modelos de redes neuronales capaces de capturar relaciones complejas entre variables. Entre estos modelos destacan las redes neuronales recurrentes, que han demostrado ser especialmente adecuadas

para analizar datos que presentan dependencia temporal. El modelo Long Short-Term Memory (LSTM) es uno de los ejemplos más conocidos dentro de esta categoría, ya que permite aprender patrones temporales complejos y capturar dependencias a largo plazo dentro de secuencias de datos [25].

El uso de redes LSTM ha sido ampliamente explorado en el ámbito de la detección de anomalías en series temporales. Diversos estudios han demostrado que estos modelos pueden aprender el comportamiento normal de una serie temporal y detectar anomalías cuando las predicciones del modelo difieren significativamente de los valores observados. Este tipo de enfoques resulta especialmente útil en sistemas de monitorización y telemetría, donde las métricas evolucionan a lo largo del tiempo y pueden presentar patrones dinámicos complejos [27].

Además de los modelos específicos utilizados para detectar anomalías, el desarrollo de infraestructuras de aprendizaje automático a gran escala ha permitido aplicar estas técnicas sobre grandes volúmenes de datos generados por sistemas distribuidos. El crecimiento de las capacidades de computación y almacenamiento ha facilitado la aplicación de algoritmos de aprendizaje profundo en entornos de análisis masivo de datos, permitiendo construir modelos cada vez más sofisticados para el análisis automático de información operativa [26].

3.1.4.3 DETECCIÓN DE ANOMALÍAS EN SERIES TEMPORALES

En muchos sistemas de monitorización y telemetría, los datos generados se organizan en forma de series temporales, donde cada observación está asociada a un instante de tiempo específico. Este tipo de datos aparece con frecuencia en métricas de rendimiento, registros de sensores o indicadores operativos de sistemas informáticos. La detección de anomalías en series temporales consiste en identificar comportamientos que se desvían significativamente de los patrones temporales esperados, como cambios bruscos en la tendencia, variaciones inesperadas en la estacionalidad o valores que no siguen la evolución habitual de la serie [22].

Una de las aproximaciones más utilizadas para abordar este problema consiste en modelar el comportamiento normal de la serie temporal y detectar desviaciones significativas respecto a dicho modelo. En muchos casos, las anomalías se identifican mediante el análisis del error entre los valores observados y los valores estimados por el modelo. Cuando la diferencia entre ambos supera un determinado umbral, la observación se considera potencialmente anómala. Este tipo de técnicas resulta especialmente útil en entornos donde las métricas presentan patrones regulares que pueden ser aprendidos a partir de los datos históricos [30].

En los últimos años, se han desarrollado diversos sistemas diseñados específicamente para detectar anomalías en series temporales a gran escala. Un ejemplo de ello es el sistema utilizado por Microsoft para el análisis automático de grandes volúmenes de datos temporales generados por infraestructuras tecnológicas. Este tipo de plataformas combina técnicas estadísticas y modelos de aprendizaje automático para identificar patrones anómalos en flujos de datos generados de forma continua [31].

Otra línea de investigación relevante en este ámbito se basa en el uso de modelos de aprendizaje profundo para analizar series temporales complejas. En particular, las redes neuronales recurrentes basadas en arquitecturas LSTM han demostrado ser eficaces para capturar dependencias temporales y detectar anomalías en secuencias de datos. Estos modelos pueden aprender patrones de comportamiento a largo plazo dentro de una serie temporal y detectar desviaciones cuando los datos observados difieren significativamente de las predicciones generadas por el modelo [27].

Además de los modelos basados en aprendizaje profundo, también se han propuesto métodos diseñados específicamente para detectar anomalías en grandes conjuntos de datos temporales generados por sistemas distribuidos. Algunos de estos enfoques combinan técnicas de análisis estadístico con algoritmos capaces de procesar grandes volúmenes de datos de forma eficiente. Por ejemplo, se han desarrollado frameworks escalables para la

detección automática de anomalías que permiten analizar grandes colecciones de series temporales generadas por sistemas de monitorización [28].

Asimismo, diversos trabajos han propuesto técnicas orientadas a detectar anomalías de largo plazo en sistemas de monitorización cloud, teniendo en cuenta cambios graduales en el comportamiento del sistema y patrones complejos que pueden evolucionar con el tiempo. Estas aproximaciones permiten identificar desviaciones persistentes que no necesariamente se manifiestan como cambios abruptos en los datos, sino como alteraciones progresivas en el comportamiento de la serie temporal [29].

3.1.4.4 DETECCIÓN DE ANOMALÍAS EN DATOS DE VUELO

La detección de anomalías en datos de vuelo constituye una línea de investigación cada vez más relevante dentro del ámbito del análisis de datos aeronáuticos. El aumento en la disponibilidad de información procedente de sistemas de vigilancia aérea, como ADS-B, ha permitido acceder a grandes volúmenes de datos sobre la posición y el comportamiento de las aeronaves durante el vuelo. Esta disponibilidad de información ha abierto la posibilidad de aplicar técnicas de análisis de datos para identificar comportamientos inusuales o potencialmente anómalos en las trayectorias de vuelo.

Los datos de vuelo presentan características que los hacen especialmente adecuados para el análisis mediante técnicas de detección de anomalías. En particular, muchas de las variables que describen el comportamiento de una aeronave, como la altitud, la velocidad, el rumbo o la velocidad vertical, evolucionan de forma continua a lo largo del tiempo y suelen seguir patrones relativamente predecibles durante las distintas fases del vuelo. Las desviaciones significativas respecto a estos patrones pueden indicar la presencia de situaciones atípicas, como cambios bruscos de trayectoria, descensos o ascensos inesperados, o comportamientos inconsistentes con las rutas habituales [38].

Diversos trabajos han explorado el uso de técnicas de análisis de trayectorias y aprendizaje automático para identificar anomalías en el tráfico aéreo. Estos enfoques suelen

basarse en la construcción de modelos que representan el comportamiento normal de las aeronaves a partir de datos históricos. Una vez definido este comportamiento esperado, es posible comparar nuevas observaciones con el modelo aprendido y detectar aquellas trayectorias o patrones de movimiento que presentan desviaciones significativas respecto a lo habitual. Este tipo de métodos permite identificar tanto anomalías puntuales en variables concretas como comportamientos anómalos en la trayectoria completa de un vuelo [39].

Asimismo, el análisis de datos de vuelo puede combinar técnicas de detección de anomalías con modelos de series temporales o con métodos basados en aprendizaje automático. La aplicación de estos enfoques permite analizar la evolución temporal de las variables de vuelo y detectar cambios inesperados en el comportamiento de la aeronave. Este tipo de análisis resulta especialmente útil en sistemas de monitorización donde se dispone de grandes volúmenes de datos generados de forma continua y donde la detección temprana de comportamientos atípicos puede aportar información valiosa para el análisis del tráfico aéreo y la supervisión del sistema.

3.2 PLATAFORMAS Y PRODUCTOS COMERCIALES DE TELEMETRÍA

Además del análisis de los fundamentos técnicos presentados en las secciones anteriores, resulta relevante examinar en qué medida estas tecnologías se traducen en soluciones disponibles en el mercado. Con este objetivo, en las siguientes secciones se revisa el panorama de productos y plataformas comerciales orientados a la telemetría y la monitorización, abordando en primer lugar el ámbito general de la ingeniería y, posteriormente, su aplicación específica al sector aeronáutico. Este análisis permite contextualizar el proyecto desarrollado respecto a las soluciones existentes en la industria y valorar su aportación diferencial.

3.2.1 PLATAFORMAS COMERCIALES DE TELEMETRÍA EN EL ÁMBITO DE LA INGENIERÍA

Más allá de las tecnologías de código abierto descritas anteriormente, el mercado ofrece un amplio catálogo de plataformas comerciales orientadas a la adquisición, el almacenamiento y la visualización de datos de telemetría en entornos industriales y de ingeniería. Entre las soluciones con mayor trayectoria destaca el sistema PI System, desarrollado originalmente por OSIsoft e integrado actualmente en el catálogo de AVEVA, que constituye uno de los historiadores de datos industriales más utilizados para capturar, almacenar y analizar series temporales procedentes de procesos productivos y sistemas de control [40].

En el ámbito del Internet de las Cosas industrial, plataformas como Siemens Insights Hub, anteriormente conocida como MindSphere, y PTC ThingWorx ofrecen infraestructuras en la nube diseñadas para conectar equipos industriales, recopilar telemetría de sensores y construir modelos de análisis y mantenimiento predictivo sobre dicha información [41], [42]. Estas plataformas se caracterizan por integrar capacidades de ingesta, almacenamiento y visualización dentro de un mismo ecosistema propietario, orientado principalmente a clientes corporativos con infraestructuras industriales complejas.

Los principales proveedores de computación en la nube también han desarrollado servicios específicos para la gestión de telemetría industrial. Es el caso de AWS IoT SiteWise, orientado a la recopilación y el modelado de datos procedentes de equipos industriales, y de Azure Digital Twins, que permite construir representaciones virtuales de sistemas físicos a partir de los datos de telemetría recibidos [43], [44]. De forma complementaria, plataformas como Datadog ofrecen soluciones comerciales de observabilidad como servicio, extendiendo las capacidades de herramientas de código abierto como Prometheus o Grafana mediante funcionalidades adicionales de análisis, alertado y correlación automática [45].

En conjunto, estas plataformas comerciales proporcionan infraestructuras robustas y ampliamente adoptadas en el ámbito de la ingeniería industrial. No obstante, se trata generalmente de soluciones propietarias, orientadas a casos de uso genéricos y con modelos de licenciamiento cerrados, lo que dificulta su adaptación a dominios específicos que requieran una lógica de validación de datos, un modelo de almacenamiento o unas técnicas de detección de anomalías adaptadas a las particularidades de un sector concreto, como ocurre en el caso de la aviación.

3.2.2 PLATAFORMAS Y PRODUCTOS COMERCIALES DE TELEMETRÍA EN EL SECTOR AERONÁUTICO

La disponibilidad de datos de telemetría aeronáutica ha favorecido el desarrollo de diversas plataformas destinadas al seguimiento y análisis del tráfico aéreo. Estas plataformas permiten recopilar información procedente de múltiples fuentes de datos, agregándola en sistemas centralizados capaces de ofrecer una visión global del comportamiento de las aeronaves en el espacio aéreo. En muchos casos, estas soluciones combinan redes distribuidas de sensores con infraestructuras de procesamiento de datos que permiten almacenar y analizar grandes volúmenes de información generada de forma continua [34], [37].

Uno de los ejemplos más representativos de este tipo de plataformas es OpenSky Network, una red colaborativa de receptores ADS-B diseñada para recopilar datos de vuelo a gran escala y ponerlos a disposición de la comunidad investigadora. Esta plataforma se basa en una infraestructura distribuida formada por estaciones receptoras que capturan las señales emitidas por las aeronaves y las envían a un sistema central donde los datos son procesados, almacenados y puestos a disposición de los usuarios mediante interfaces de acceso específicas [34], [36]. Gracias a este enfoque, OpenSky permite analizar patrones de tráfico aéreo, estudiar trayectorias de vuelo y desarrollar investigaciones relacionadas con la monitorización del espacio aéreo.

Junto a estas iniciativas de carácter investigador, existen numerosos servicios comerciales orientados al público general que ofrecen seguimiento de vuelos en tiempo real a partir de redes de receptores ADS-B. Entre los más conocidos se encuentran Flightradar24 y FlightAware, plataformas que agregan datos procedentes de miles de receptores distribuidos por todo el mundo para ofrecer visualización de tráfico aéreo, información histórica de vuelos y servicios de alerta básicos [46], [47]. De forma similar, ADS-B Exchange se ha consolidado como una red colaborativa que distribuye datos de vuelo sin filtrar, incluyendo aeronaves que otras plataformas excluyen por motivos comerciales o de privacidad [48].

Más allá de los servicios de seguimiento orientados al público general, los principales fabricantes aeronáuticos han desarrollado plataformas propias de telemetría orientadas a la gestión de flotas y al mantenimiento predictivo. Airbus, por ejemplo, ofrece Skywise, una plataforma de datos que integra información de telemetría de las aeronaves con el objetivo de optimizar operaciones de mantenimiento y mejorar la eficiencia operativa de las aerolíneas [49]. Boeing dispone de una propuesta equivalente a través de su plataforma AnalytX y del sistema Airplane Health Management (AHM), orientado a la monitorización en tiempo real del estado de los sistemas de la aeronave [50].

Otros proveedores del sector, como Honeywell, ofrecen soluciones específicas como Honeywell Forge, orientada a la optimización de la eficiencia de vuelo a partir del análisis de datos de telemetría, mientras que fabricantes de sistemas embarcados como Teledyne Controls proporcionan sistemas de monitorización de la condición de la aeronave (Aircraft Condition Monitoring System, ACMS) encargados de capturar y transmitir telemetría detallada durante el vuelo [51], [52].

No obstante, tanto las iniciativas de investigación como la mayoría de las plataformas comerciales descritas se centran principalmente en la agregación, el seguimiento y la visualización de datos de vuelo, sin incorporar necesariamente mecanismos avanzados de análisis automatizado sobre los datos recogidos. Además, las plataformas orientadas a la

gestión de flotas presentan un carácter cerrado y propietario, fuertemente ligado a un fabricante o proveedor concreto. En conjunto, la detección automática de anomalías en trayectorias o en el comportamiento de las aeronaves, dentro de una arquitectura abierta y extensible que combine ingesta, almacenamiento estructurado y análisis configurable, como la planteada en el presente trabajo, sigue siendo un área no cubierta de forma satisfactoria por el estado del arte actual.

3.2.3 SÍNTESIS Y POSICIONAMIENTO DEL PROYECTO FRENTE AL ESTADO DEL ARTE

El análisis conjunto de las secciones técnica y aplicada de este capítulo permite identificar una brecha entre, por un lado, las tecnologías y arquitecturas de ingesta, almacenamiento y observabilidad disponibles en el ámbito de la ingeniería de datos, y, por otro, las plataformas comerciales existentes en el sector aeronáutico, generalmente orientadas a la visualización de tráfico aéreo o al mantenimiento predictivo dentro de ecosistemas cerrados propios de cada fabricante.

En este contexto, el sistema desarrollado en el presente trabajo se sitúa en la intersección de ambos ámbitos, proponiendo una arquitectura abierta y modular que combina la ingesta de telemetría en tiempo real, su almacenamiento estructurado mediante bases de datos de series temporales y la aplicación de técnicas de detección de anomalías configurables, replicando a menor escala capacidades presentes en plataformas comerciales como Skywise o Honeywell Forge, pero con un enfoque orientado a la experimentación, la extensibilidad y el aprendizaje, en línea con los objetivos académicos definidos en el Capítulo 2.

Dentro del ámbito académico, la propuesta de arquitecturas de procesamiento de telemetría en tiempo real ha sido también objeto de estudio en trabajos previos, como el de Aibar Álvarez, centrado en el diseño de una arquitectura de procesamiento de datos en tiempo real para sistemas de monitorización [32]. El presente trabajo se diferencia de dicho enfoque al incorporar, además de la ingesta y el procesamiento, un módulo de detección de anomalías

basado en distintos modelos de aprendizaje automático y una interfaz de visualización en tiempo real orientada específicamente al dominio aeronáutico.

CAPÍTULO 4 – TECNOLOGÍAS EMPLEADAS

4.1 TECNOLOGÍAS PARA EL DESARROLLO DE SERVICIOS BACKEND

4.1.1 FRAMEWORKS PARA DESARROLLO DE APIS

4.1.1.1 PYTHON 3.11

Python 3.11 es el lenguaje de programación utilizado tanto para el servicio de ingesta como para los emisores sintéticos de telemetría. Se eligió por la madurez de su ecosistema para el desarrollo de servicios web y el procesamiento de datos, así como por su compatibilidad con las librerías empleadas en el proyecto, como FastAPI, Pydantic y psycopg. La versión 3.11 se fija de forma explícita como imagen base de los contenedores Docker del proyecto (python:3.11-slim), garantizando un entorno de ejecución reproducible [53].

4.1.1.2 FastAPI

FastAPI es el framework empleado para implementar el servicio de ingesta de telemetría (service/main.py). Se seleccionó por su soporte nativo para programación asíncrona, su integración directa con Pydantic para la validación automática de los datos recibidos y la generación automática de documentación OpenAPI. Sobre FastAPI se han definido los endpoints /ingest, para la recepción de puntos de telemetría, /health, para comprobaciones de disponibilidad, y /metrics, para la exposición de métricas operativas del servicio. La versión empleada en el proyecto es la 0.115.2 [54].

4.1.1.3 Uvicorn

Uvicorn es el servidor ASGI utilizado para ejecutar la aplicación FastAPI del servicio de ingesta. Se ha configurado con las dependencias adicionales del extra [standard], que incluyen componentes optimizados como uvloop y httptools, mejorando el rendimiento del bucle de eventos y del análisis de peticiones HTTP respecto a la configuración por defecto. La versión utilizada es la 0.30.6 [55].

4.1.1.4 requests (para realizar peticiones HTTP desde el emisor sintético)

La librería requests se emplea en los emisores sintéticos de telemetría (flight_emitter.py y scenario_emitter.py) para enviar los puntos generados al servicio de ingesta mediante peticiones HTTP POST al endpoint /ingest. Su sencillez de uso y su amplia adopción la convierten en una opción adecuada para un cliente HTTP síncrono como el que emplean los emisores, que generan y envían telemetría de forma secuencial [56].

4.1.2 Validación y modelado de datos

4.1.2.1 Pydantic v2

Pydantic v2 se utiliza para definir el modelo de datos TelemetryIn, que constituye el contrato cerrado de telemetría del sistema (service/models.py). Mediante restricciones declarativas sobre cada campo (por ejemplo, rangos válidos para la latitud, la longitud, el rumbo o la altitud) y validadores personalizados, Pydantic garantiza que únicamente se

acepten puntos de telemetría que cumplan la estructura y las reglas de negocio definidas, incluyendo la coherencia entre la etiqueta de anomalía, su tipo, su metadato asociado y su score. La versión empleada es la 2.9.2 [57].

4.1.2.2 PYTHON-DOTENV (GESTIÓN DE VARIABLES DE ENTORNO)

La librería python-dotenv permite cargar variables de entorno definidas en archivos .env durante el desarrollo local, facilitando la configuración de credenciales de base de datos, claves de API del servicio de ingesta y parámetros de los emisores sintéticos sin necesidad de modificarlos directamente en el código. La versión empleada es la 1.0.1 [58].

4.2 TECNOLOGÍAS DE ALMACENAMIENTO DE DATOS

4.2.1 SISTEMAS DE BASES DE DATOS RELACIONALES

4.2.1.1 POSTGRESQL 15

PostgreSQL 15 es el sistema de gestión de bases de datos relacional utilizado como almacén persistente del sistema. Sobre él se definen las tablas flights, flight_segments, telemetry_points, ml_training_points y simulation_runs, mediante un conjunto de migraciones SQL incrementales (infra/sql) que, además de definir el esquema, incorporan restricciones CHECK que refuerzan a nivel de base de datos el contrato de telemetría ya validado en la capa de aplicación mediante Pydantic, aportando una segunda barrera de consistencia de los datos [59].

4.2.2 ACCESO A BASES DE DATOS DESDE APLICACIONES

4.2.2.1 PSYCOPG3

psycopg3 es el adaptador utilizado para ejecutar consultas SQL parametrizadas contra PostgreSQL desde el servicio de ingesta (service/db.py). Frente a su predecesor psycopg2, ofrece soporte nativo para tipos de datos modernos de Python y una API más adecuada para su uso combinado con frameworks asíncronos como FastAPI [60].

4.2.2.2 PSYCOPG_POOL

psycopg_pool proporciona un mecanismo de pool de conexiones reutilizables hacia PostgreSQL, evitando el coste de abrir una nueva conexión en cada operación de escritura o lectura. En el servicio de ingesta se configura un pool de tamaño reducido (hasta 5 conexiones simultáneas), compartido por las distintas operaciones que intervienen en el procesamiento de cada punto de telemetría recibido [60].

4.2.3 BASES DE DATOS DE SERIES TEMPORALES

Tecnologías utilizadas para el almacenamiento eficiente de datos generados continuamente.

En la implementación actual del sistema, los datos de telemetría se almacenan en tablas convencionales de PostgreSQL, indexadas por marca temporal y por vuelo. Está prevista la migración de estas tablas hacia una base de datos optimizada para series temporales cuando el volumen de ingestión en tiempo real lo justifique, con el fin de mejorar el rendimiento de las consultas y la gestión del ciclo de vida de los datos históricos.

4.2.3.1 TIME SERIES DATABASES (TSDB)

Las bases de datos de series temporales (Time Series Databases, TSDB) son sistemas de almacenamiento optimizados para datos indexados por marca temporal, como es el caso de la telemetría de vuelo. Frente a una base de datos relacional convencional, una TSDB incorpora mecanismos específicos de compresión, particionado automático por rangos de tiempo y funciones de agregación temporal, lo que mejora el rendimiento de las consultas sobre grandes volúmenes de datos históricos y facilita la aplicación de políticas de retención y purgado de información antigua.

4.2.3.2 TIMESCALEDB

TimescaleDB es una extensión de PostgreSQL que añade capacidades de base de datos de series temporales mediante hypertables, que particionan automáticamente las tablas por rangos temporales. Se ha seleccionado como la solución prevista para evolucionar el almacenamiento de telemetry_points cuando el sistema pase a operar con mayores

volúmenes de ingestión en tiempo real, dado que permite mantener la compatibilidad con SQL estándar y con las herramientas ya utilizadas en el proyecto (psycopg, Grafana). En el estado actual del desarrollo, la base de datos se despliega como PostgreSQL 15 estándar, sin la extensión TimescaleDB habilitada, por lo que esta integración se plantea como trabajo futuro [61].

4.3 TECNOLOGÍAS DE MONITORIZACIÓN Y OBSERVABILIDAD

4.3.1 SISTEMAS DE MONITORIZACIÓN BASADOS EN MÉTRICAS

4.3.1.1 PROMETHEUS CLIENT

El servicio de ingesta se instrumenta mediante la librería prometheus-client, que expone métricas operativas en formato Prometheus a través del endpoint /metrics. Entre las métricas expuestas se incluyen contadores del número de puntos de telemetría válidos e inválidos recibidos (ingest_valid_points_total, ingest_invalid_points_total) y contadores de peticiones HTTP desglosados por código de estado, método y ruta. Estas métricas son recogidas periódicamente por un servidor Prometheus desplegado como contenedor independiente (prom/prometheus, versión v2.52.0), configurado para realizar scraping del servicio cada 10 segundos [17].

4.3.2 PLATAFORMAS DE VISUALIZACIÓN DE DATOS

4.3.2.1 GRAFANA 11

Grafana 11 se utiliza como plataforma de visualización de las métricas y de los datos almacenados en el sistema. Se despliega mediante la imagen oficial grafana/grafana en su versión 11.1.0 y se configura con datasources aprovisionados automáticamente hacia PostgreSQL y hacia el servidor Prometheus del proyecto. Los paneles se definen como dashboards versionados en el repositorio (vuelos, estado de las simulaciones y tablas de la base de datos), lo que permite reconstruir el entorno de visualización de forma reproducible en cualquier despliegue [18] (Figura 2).

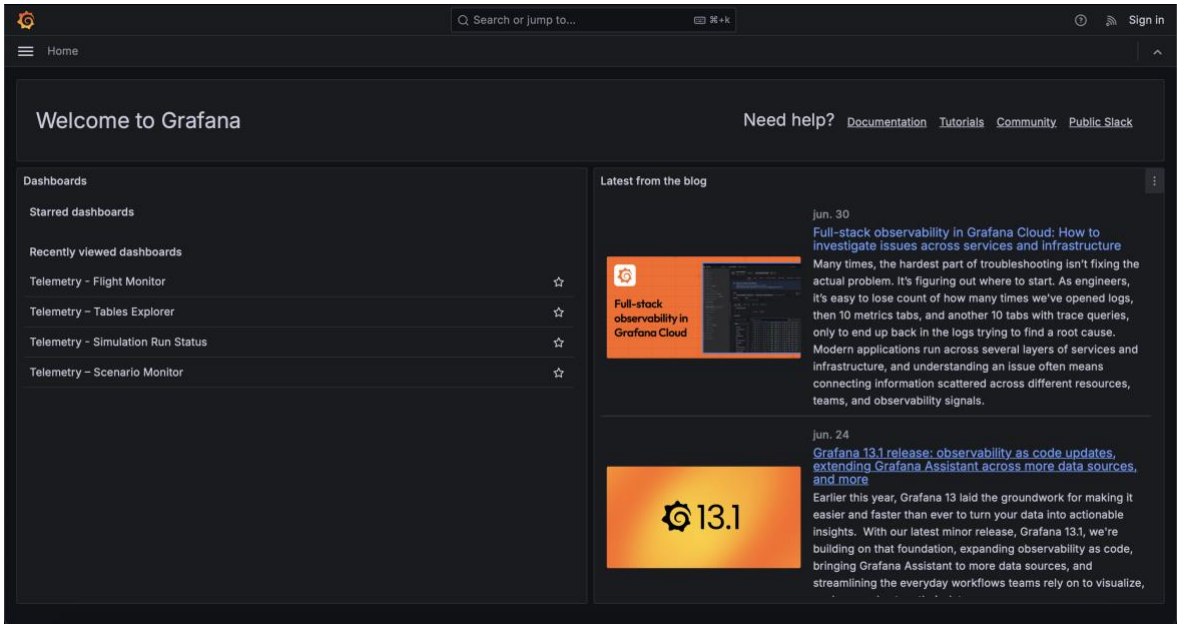

Figura 2. Interfaz de Grafana (pantalla de bienvenida / listado de dashboards)

4.4 TECNOLOGÍAS DE CONTENERIZACIÓN Y DESPLIEGUE

4.4.1 CONTENEDORES SOFTWARE

4.4.1.1 DOCKER

Docker se utiliza para empaquetar y ejecutar de forma aislada los distintos componentes del sistema. Tanto el servicio de ingesta como el emisor sintético de telemetría disponen de su propio Dockerfile, lo que permite construir imágenes reproducibles con las dependencias exactas necesarias para cada componente, independientemente del entorno del host en el que se ejecuten [62].

4.4.1.2 IMAGEN BASE PYTHON:3.11-SLIM

Ambos Dockerfiles del proyecto (servicio de ingesta y emisor sintético) emplean como imagen base python:3.11-slim, una variante reducida de la imagen oficial de Python que incluye únicamente los paquetes necesarios para ejecutar aplicaciones Python. Esta elección reduce el tamaño final de las imágenes generadas y limita la superficie de

componentes instalados en los contenedores, en línea con buenas prácticas de contenerización orientadas a la eficiencia y a la seguridad del despliegue.

4.4.2 ORQUESTACIÓN DE SERVICIOS

4.4.2.1 DOCKER COMPOSE

Docker Compose se utiliza para orquestar de forma conjunta los servicios que componen el sistema: la base de datos PostgreSQL, el servicio de ingesta, el emisor sintético de telemetría, el servidor Prometheus y Grafana. La definición del stack completo en un único archivo docker-compose.yml permite levantar y detener el entorno de desarrollo o de pruebas de extremo a extremo mediante un único comando, garantizando además que las dependencias entre servicios, como la disponibilidad de la base de datos antes de iniciar el servicio de ingesta, se gestionan de forma declarativa [63].

4.4.2.2 DOCKER COMPOSE V2.20.0

El proyecto fija la versión de Docker Compose utilizada en 2.20.0, correspondiente a la especificación Compose V2 integrada como plugin de la CLI de Docker. Fijar esta versión facilita la reproducibilidad del entorno entre distintas máquinas de desarrollo y evita incompatibilidades derivadas de cambios en la sintaxis del archivo de definición del stack entre versiones.

4.5 TECNOLOGÍAS DE DESARROLLO Y CONTROL DE CALIDAD

4.5.1 TESTING AUTOMATIZADO

4.5.1.1 PYTEST

Pytest se utiliza como framework de pruebas automatizadas del proyecto. La suite de pruebas, ubicada en tests/, incluye pruebas de contrato sobre los datos generados por los emisores sintéticos (test_emitter_contract.py), pruebas de extremo a extremo sobre el

pipeline completo de ingesta (test_pipeline.py) y pruebas de carga que simulan condiciones de mayor volumen de tráfico (test_pipeline_load.py). La versión empleada es la 7.4.0 [64].

4.5.1.2 MARCA SLOW DE PYTEST PARA PRUEBAS DE CARGA

El archivo pytest.ini define un marcador personalizado, slow, utilizado para identificar aquellas pruebas de carga o de estrés que requieren más tiempo de ejecución o un mayor consumo de recursos, como test_pipeline_load.py. Esta marca permite ejecutar de forma selectiva únicamente las pruebas rápidas durante el desarrollo habitual, reservando la ejecución de las pruebas marcadas como slow para validaciones más exhaustivas.

4.5.2 INTEGRACIÓN CONTINUA

4.5.2.1 GITHUB ACTIONS

GitHub Actions se utiliza como plataforma de integración continua del proyecto, mediante un workflow definido en .github/workflows/github-actions.yml. Este workflow automatiza la ejecución de la suite de pruebas ante cada cambio en el repositorio, permitiendo detectar de forma temprana regresiones en el comportamiento del servicio de ingesta, de los emisores sintéticos o del esquema de base de datos [65].

4.6 TECNOLOGÍAS PARA INTERFACES WEB

4.6.1 FRAMEWORKS DE FRONTEND MODERNOS

4.6.1.1 REACT

React es una biblioteca de JavaScript de código abierto, mantenida por Meta, orientada a la construcción de interfaces de usuario mediante la composición de componentes reutilizables y un modelo de renderizado declarativo basado en un DOM virtual. En este proyecto se emplea para desarrollar el panel de visualización en tiempo real del estado de las aeronaves simuladas (Capítulo 5), apoyándose en su ecosistema de librerías complementarias, como react-leaflet para la representación cartográfica de las posiciones de vuelo, lo que permite

construir una interfaz reactiva que se actualiza automáticamente conforme llegan nuevos datos de telemetría [66].

4.7 TECNOLOGÍAS DE ADQUISICIÓN DE DATOS AERONÁUTICOS

4.7.1 SISTEMAS DE VIGILANCIA AÉREA

4.7.1.1 ADS-B (AUTOMATIC DEPENDENT SURVEILLANCE-BROADCAST)

El sistema Automatic Dependent Surveillance–Broadcast (ADS-B), ya descrito en el Capítulo 3, constituye la referencia real sobre la que se ha diseñado el contrato de telemetría del proyecto. Las variables incluidas en el modelo TelemetryIn (posición, altitud, velocidad respecto al suelo, rumbo, identificador de aeronave, entre otras) replican el tipo de información que un receptor ADS-B obtiene habitualmente de una aeronave, lo que permite que los datos generados por los emisores sintéticos del proyecto sean estructuralmente compatibles con datos ADS-B reales, aunque su origen actual sea simulado [33].

4.8 TECNOLOGÍAS PARA DETECCIÓN DE ANOMALÍAS

4.8.1 ESTRATEGIA DE EVALUACIÓN DE MODELOS DE DETECCIÓN DE ANOMALÍAS

Antes de seleccionar un modelo concreto de detección de anomalías, resulta necesario definir una estrategia de evaluación que permita comparar de forma objetiva distintos enfoques sobre los datos de telemetría generados por el sistema. Esta estrategia se apoya en el conjunto de datos etiquetado que el propio sistema genera de forma nativa: cada punto de telemetría almacenado en ml_training_points incorpora una etiqueta de anomalía, su tipo y una puntuación de confianza asignados de forma controlada por el emisor sintético, lo que permite evaluar modelos de forma supervisada sin depender de un proceso de etiquetado manual.

4.8.1.1 VENTAJAS DE LA COMPARACIÓN DE MÚLTIPLES MODELOS

Comparar varios modelos de detección de anomalías, en lugar de adoptar directamente uno de ellos, permite contrastar distintos supuestos sobre la naturaleza de los datos de telemetría: mientras que los métodos estadísticos asumen una distribución conocida del comportamiento normal, los métodos basados en aprendizaje automático pueden capturar relaciones no lineales entre variables sin necesidad de definirlas explícitamente. Esta comparación resulta especialmente relevante en un problema con clases desbalanceadas, como la detección de anomalías, en el que la proporción de puntos anómalos es reducida frente al total de la telemetría generada.

4.8.1.2 CRITERIOS GENERALES PARA LA COMPARACIÓN DE MODELOS

La tabla ml_training_points incorpora una columna split, ya prevista en el esquema de base de datos, orientada a soportar particiones de entrenamiento, validación y prueba sobre el conjunto de datos generado. Los criterios previstos para comparar los modelos incluyen métricas orientadas a problemas desbalanceados, como la precisión, la exhaustividad (recall) y el área bajo la curva precisión-exhaustividad, además de aspectos prácticos como el coste computacional de la inferencia y la interpretabilidad de los resultados, relevante en un dominio como el aeronáutico.

4.8.2 MODELOS DE APRENDIZAJE AUTOMÁTICO PARA DETECCIÓN DE ANOMALÍAS

En este apartado se describen los modelos seleccionados para el estudio, presentando sus fundamentos teóricos, su funcionamiento general y su posible aplicación a datos de telemetría. Cada uno de ellos se analiza de forma individual con el fin de mostrar sus características principales y su utilidad dentro del problema planteado.

4.8.2.1 BASELINE ESTADISTICO

Como primer candidato se considera un modelo estadístico basado en el control de procesos, que modela el comportamiento normal de cada variable de telemetría mediante

límites de control calculados a partir de su media y desviación estándar histórica, señalando como anómalo cualquier valor que se aleje significativamente de dicho comportamiento. Se trata de un enfoque interpretable y de bajo coste computacional, que sirve como referencia de partida (baseline) frente a los modelos más complejos considerados [23].

4.8.2.2 ISOLATION FOREST

Como segundo candidato se considera Isolation Forest, un modelo de aprendizaje no supervisado que detecta anomalías en función de la facilidad con la que un punto puede aislarse del resto de las observaciones mediante particiones aleatorias del espacio de variables. Su carácter no supervisado resulta adecuado para explorar la telemetría generada sin depender en exceso de las etiquetas de anomalía, y su coste computacional relativamente reducido lo hace apropiado para conjuntos de datos de tamaño considerable [24].

4.8.2.3 LSTM

Como tercer candidato se considera un modelo basado en redes LSTM (Long Short-Term Memory), capaz de capturar dependencias temporales entre observaciones consecutivas de un mismo vuelo y detectar desviaciones respecto al comportamiento esperado de la serie temporal. Este enfoque resulta especialmente adecuado para variables con una fuerte componente secuencial, como la evolución de la altitud o de la velocidad a lo largo de un vuelo, aunque su entrenamiento requiere un mayor volumen de datos y de recursos computacionales que los modelos anteriores [25], [27].

4.8.2.4 LOCAL OUTRIDER FACTOR

Como cuarto candidato se considera un modelo basado en la densidad local de los datos, como Local Outlier Factor, que identifica como anómalos aquellos puntos cuya densidad de vecinos es significativamente menor que la de su entorno. Este enfoque complementa a los anteriores al aportar una perspectiva basada en la vecindad de los datos, en lugar de en su distribución global o en su evolución temporal, resultando de interés para detectar anomalías puntuales en variables como la posición o la altitud instantánea [21].

CAPÍTULO 5 – SISTEMA DESARROLLADO

Este capítulo describe en detalle el sistema efectivamente implementado: su arquitectura, los componentes que lo forman, el diseño del modelo de datos, el proceso de ingesta y validación de telemetría, la persistencia en base de datos y la instrumentación de observabilidad desplegada. Cada apartado se apoya en referencias directas al código del repositorio (ruta de archivo, función y línea concreta), de modo que la descripción arquitectónica quede trazada de forma precisa sobre la implementación real, y no sobre un diseño meramente conceptual.

5.1 DISEÑO DEL SISTEMA

El sistema se ha diseñado como un conjunto de servicios independientes, desplegados como contenedores Docker y orquestados mediante Docker Compose (infra/docker-compose.yml), comunicados entre sí a través de interfaces HTTP y de un esquema de base de datos compartido. Esta sección describe la arquitectura general de la plataforma, sus componentes principales, el flujo de funcionamiento extremo a extremo y los requisitos funcionales y no funcionales que ha debido satisfacer.

5.1.1 ARQUITECTURA GENERAL DE LA PLATAFORMA

La plataforma se compone de cinco servicios definidos en infra/docker-compose.yml. El servicio db (imagen postgres:15) actúa como almacén persistente, con un volumen nombrado db_data para conservar los datos entre reinicios y un bind mount de infra/sql sobre /docker-entrypoint-initdb.d, de forma que Postgres ejecuta automáticamente 001_init.sql al arrancar sobre un volumen vacío. El servicio service (construido a partir de service/Dockerfile) expone la API de ingesta en el puerto 8000 y depende de db. El servicio emitter (construido a partir de synthetic/Dockerfile) ejecuta flight_emitter.py y depende de service, al que envía la telemetría generada mediante peticiones HTTP; se define con restart: "no", de forma que una simulación finalizada no se reinicia automáticamente. El servicio prometheus (imagen prom/prometheus:v2.52.0) depende de service y realiza scraping periódico de su endpoint /metrics. Por último, grafana (imagen grafana/grafana:11.1.0) depende de db y de prometheus, y expone en el puerto 3000 los dashboards provisionados automáticamente desde infra/grafana/provisioning. El grafo de dependencias resultante (emitter → service → db; service → prometheus → grafana; grafana → db) garantiza que, en un arranque desde cero, la base de datos y la API estén disponibles antes de que el emisor sintético o el resto de la pila de observabilidad intenten utilizarlas (Figura 3).

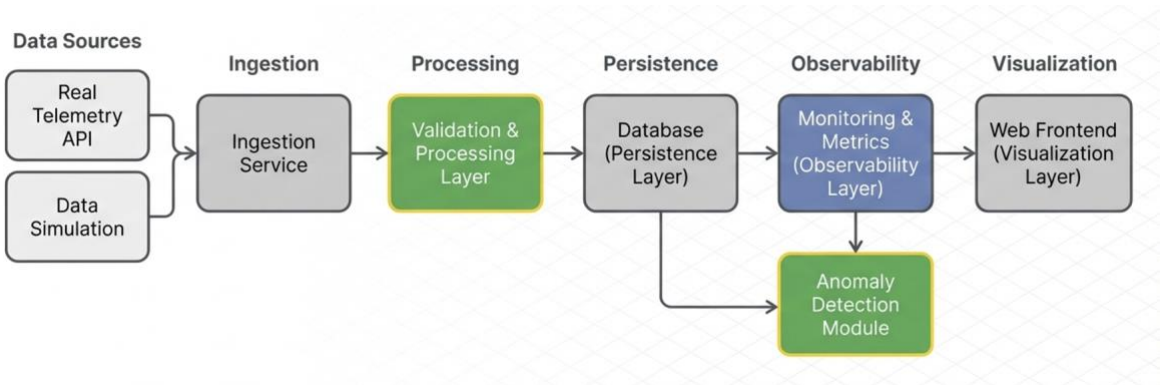

Figura 3. Arquitectura del sistema

5.1.2 COMPONENTES PRINCIPALES DEL SISTEMA

A nivel de código, el sistema se organiza en cuatro módulos con responsabilidades disjuntas. El módulo service/ contiene el servicio de ingesta: main.py define la aplicación FastAPI y sus rutas (/health, /metrics, /ingest, /simulation-run/status); models.py define el contrato de datos mediante los modelos Pydantic TelemetryIn y SimulationRunStatusIn; db.py concentra todo el acceso a PostgreSQL a través de un pool de conexiones psycopg_pool.ConnectionPool y expone las funciones ensure_flight, upsert_flight_segment, upsert_simulation_run, insert_telemetry_point e insert_ml_training_point. El módulo infra/sql/ contiene el esquema de base de datos: 001_init.sql define el esquema completo para una base de datos nueva, mientras que 002_anomaly_metadata.sql a 007_route_identity_fields.sql son migraciones incrementales e idempotentes (basadas en ADD COLUMN IF NOT EXISTS y bloques DO $$ que comprueban la existencia de cada CONSTRAINT antes de crearlo) pensadas para aplicarse sobre una base de datos ya existente. El módulo synthetic/ contiene los emisores sintéticos: flight_emitter.py genera vuelos físicamente plausibles a partir de route_catalog.py (catálogo de rutas y perfiles de rendimiento por tipo de ruta) e inyecta anomalías mediante el paquete synthetic/scenarios/ (contracts.py define la taxonomía de escenarios y anomalías; phases.py construye el payload final aplicando el contrato de telemetría; anomalies.py implementa cada generador de anomalía; registry.py expone el registro de escenarios disponibles). Por último, el módulo infra/ agrupa la configuración de despliegue y observabilidad: docker-compose.yml, los Dockerfile de cada servicio, infra/prometheus/prometheus.yml y infra/grafana/provisioning/ (datasources y dashboards).

5.1.3 FLUJO GENERAL DE FUNCIONAMIENTO

El flujo de funcionamiento extremo a extremo comienza en el emisor sintético, que construye un payload conforme al contrato de telemetría (aproximadamente treinta campos, incluyendo posición, actitud, estado del transpondedor y metadatos de escenario/anomalía) y lo envía mediante una petición HTTP POST a /ingest, autenticada con la cabecera x-api-

key. FastAPI valida automáticamente el cuerpo de la petición contra el modelo TelemetryIn antes de que se ejecute el cuerpo de la función ingest(): si el payload no es válido, se devuelve un 422 sin que el código de la aplicación llegue a ejecutarse. Si la validación es correcta, el manejador definido en service/main.py (líneas 55-131) ejecuta de forma secuencial cuatro operaciones sobre la base de datos: primero ensure_flight, que crea o actualiza el registro del vuelo; después upsert_flight_segment, que mantiene la segmentación por escenario y tipo de anomalía; a continuación insert_telemetry_point, que persiste el punto de telemetría y devuelve su id; y finalmente insert_ml_training_point, que inserta la réplica del punto en la tabla orientada a entrenamiento de modelos. Si las cuatro operaciones se completan sin excepción, se incrementa el contador Prometheus de puntos válidos y se devuelve un 202 con el id generado y la marca de tiempo del servidor. Si alguna de ellas lanza una excepción, se incrementa el contador de puntos inválidos y se devuelve un error 500. El siguiente extracto muestra el manejador completo, incluyendo el cálculo de ingest_latency_ms cuando el cliente no lo proporciona:

```
@app.post("/ingest", status_code=202, dependencies=[Depends(check_key)])
def ingest(t: TelemetryIn, request: Request):
    with LAT.labels("/ingest", "POST").time():
        server_ts = datetime.now(timezone.utc)
        try:
            ensure_flight(
                flight_id=str(t.flight_id),
                icao24=t.icao24.lower(),
                start_ts=t.ts,
                end_ts=t.ts,
                scenario=t.scenario,
                scenario_params={"source": t.source, "schema_version": t.schema_version},
            )
            upsert_flight_segment(
                flight_id=str(t.flight_id), ts=t.ts,
                scenario=t.scenario, anomaly_type=t.anomaly_type,
            )

            ingest_latency_ms = t.ingest_latency_ms
            if ingest_latency_ms is None:
                ingest_latency_ms = int((server_ts - t.ts).total_seconds() * 1000)

            # payload = { ... ~30 campos del contrato v2 ... }

            new_id = insert_telemetry_point(payload)
            insert_ml_training_point(new_id, payload)
            VALID.inc()
```

```
            REQS.labels("/ingest", "POST", "202").inc()
            return {"id": new_id, "server_ts": server_ts.isoformat()}
        except Exception as e:
            INVALID.inc()
            REQS.labels("/ingest", "POST", "500").inc()
            raise HTTPException(status_code=500, detail=str(e))
```

(service/main.py:55-131, extracto)

Este extracto corresponde al estado inicial del manejador; en la implementación actual, tras insert_ml_training_point se añade además la puntuación en tiempo real por los modelos de detección de anomalías, descrita en detalle en el apartado 5.6 de este mismo capítulo, envuelta en su propio bloque try/except para que un fallo del modelo nunca afecte al resultado de la ingesta.

5.1.4 REQUISITOS FUNCIONALES Y NO FUNCIONALES

5.1.4.1 REQUISITOS FUNCIONALES

1. Recibir telemetría de vuelo mediante un endpoint HTTP autenticado.
2. Validar cada punto recibido frente a un contrato de datos cerrado, rechazando cualquier campo no reconocido (TelemetryIn se define con model_config = ConfigDict(populate_by_name=True, extra="forbid") en service/models.py:44) y cualquier combinación de campos que viole las reglas de negocio del contrato de anomalías.
3. Persistir de forma estructurada los vuelos, sus segmentos por escenario/anomalía, los puntos de telemetría y su réplica orientada a entrenamiento de modelos.
4. Exponer métricas operativas en formato Prometheus sobre el volumen y la validez de los datos ingeridos.
5. Permitir generar telemetría sintética parametrizable (tasa de emisión, número de vuelos, proporción de valores extremos, semilla aleatoria, tipos de ruta) y reportar el estado de cada ejecución de simulación a través del endpoint /simulation-run/status.
6. Visualizar los datos almacenados y las métricas del sistema mediante dashboards de Grafana provisionados automáticamente.

5.1.4.2 REQUISITOS NO FUNCIONALES

En cuanto a requisitos no funcionales: la escalabilidad de la capa de persistencia se apoya en un pool de conexiones configurado con min_size=1 y max_size=5 (service/db.py:14-15), dimensionado para el volumen de tráfico sintético actual, no para una carga de producción; la reproducibilidad del entorno se garantiza fijando la versión de todas las imágenes empleadas (postgres:15, grafana/grafana:11.1.0, prom/prometheus:v2.52.0, python:3.11-slim como base de los servicios propios); la seguridad de acceso a la API se resuelve con un mecanismo mínimo de autenticación por clave estática, comparada en texto plano en la cabecera x-api-key (service/main.py:30-37), suficiente para un entorno de desarrollo pero sin rotación de credenciales ni control de acceso granular; y la trazabilidad de los datos se apoya en la restricción UNIQUE(flight_id, seq) de telemetry_points y en las claves foráneas entre tablas, descritas en el apartado de persistencia. No se han implementado actualmente comprobaciones de salud (healthcheck) explícitas entre servicios en docker-compose.yml, por lo que la disponibilidad del sistema ante un arranque en frío depende únicamente del orden declarado en depends_on.

5.2 MÓDULO DE INGESTA DE DATOS

5.2.1 USO DE DATOS SIMULADOS

La totalidad de los datos procesados actualmente procede de los emisores sintéticos del proyecto: synthetic/flight_emitter.py y synthetic/scenario_emitter.py. Ambos construyen vuelos a partir de route_catalog.py, que define ROUTE_CATALOG (un conjunto de rutas origen-destino) y ROUTE_PROFILES (rangos de altitud y velocidad de crucero por tipo de ruta), de forma que la cinemática generada sea coherente con el tipo de trayecto simulado. Sobre esa base, el paquete synthetic/scenarios/ inyecta anomalías de forma controlada: contracts.py define la taxonomía cerrada de escenarios y su tipo de anomalía asociado (SCENARIO_ANOMALY_TYPE), su nivel de confianza (SCENARIO_CONFIDENCE) y su score base (ANOMALY_BASE_SCORE); anomalies.py implementa los generadores

concretos (engine_failure, gnss_drift, turbulence, wind_shear, dropout, spike, frozen, route_deviation, irregular_sampling, multi); y phases.py (función enrich_payload) combina el punto cinemático base con los metadatos de anomalía exigidos por el contrato antes de enviarlo. El emisor admite los parámetros, rate (frecuencia de emisión),, flights (número total de vuelos),, batch-size (vuelos simulados de forma concurrente),, extreme-ratio (proporción de valores forzados a condiciones extremas) y, seed (semilla para reproducibilidad). La concurrencia se implementa mediante una lista de generadores activos (active_streams): cada vuelo es un generador Python que produce un punto por iteración; cuando un generador se agota (StopIteration), se sustituye por uno nuevo hasta alcanzar, flights vuelos en total, manteniendo siempre, batch-size vuelos emitiendo en paralelo:

```
while active_streams:
    i = 0
    while i < len(active_streams):
        gen = active_streams[i]
        try:
            payload = next(gen)
        except StopIteration:
            if next_flight_idx < args.flights:
                active_streams[i] = spawn_stream(next_flight_idx)
                next_flight_idx += 1
            else:
                active_streams.pop(i)
            continue

        r = requests.post(args.url, json=payload,
                          headers={"x-api-key": args.api_key}, timeout=3)
        if r.status_code >= 400:
            print("error", r.status_code, r.text)
        else:
            emitted_points += 1
```

(synthetic/flight_emitter.py:396-420, extracto)

5.2.2 DISEÑO DEL PROCESO DE INGESTA

El proceso de ingesta se estructura en tres capas claramente separadas: enrutado y autenticación (service/main.py), validación (service/models.py) y persistencia (service/db.py). La aplicación FastAPI se instancia una única vez a nivel de módulo (app = FastAPI(title="TFM Telemetry Service", version="0.1.0")) y la autenticación se implementa como una dependencia de FastAPI reutilizable, aplicada mediante

dependencies=[Depends(check_key)] en la propia decoración de la ruta, de forma que la comprobación de la clave se ejecuta antes de que FastAPI intente siquiera parsear y validar el cuerpo de la petición:

```python
def check_key(request: Request):
    key = request.headers.get("x-api-key")
    if key != API_KEY:
        raise HTTPException(status_code=401, detail="invalid api key")
```

(service/main.py:30-37, extracto)

5.2.3 RECEPCIÓN DE DATOS

Actualmente el servicio expone un único canal de recepción: peticiones HTTP POST síncronas contra /ingest, una por cada punto de telemetría (no existe agregación en lotes a nivel de API, aunque el emisor sí agrupa varios vuelos concurrentes mediante, batch-size en el lado del cliente). La frecuencia de llegada de datos está controlada por el emisor, no por el servicio:, rate fija un retardo delay = 1.0 / args.rate entre ticks de cada generador de vuelo (synthetic/flight_emitter.py, función main()), y cada petición se envía con un timeout de 3 segundos (requests.post(..., timeout=3)).

La gestión de errores durante la recepción es actualmente asimétrica entre cliente y servidor. En el servidor, cualquier excepción durante la fase de persistencia se traduce en un HTTP 500 y en el incremento del contador ingest_invalid_points_total (service/main.py:128-131); los errores de validación del contrato (por ejemplo, un campo fuera de rango) se resuelven en la capa de FastAPI/Pydantic con un HTTP 422 antes de que el código de la aplicación se ejecute, por lo que no incrementan ese contador. En el cliente, el emisor sintético registra por consola cualquier respuesta con código igual o superior a 400 (print("error", r.status_code, r.text)), pero no reintenta el envío ni aplica ningún tipo de backoff: el punto rechazado se descarta y el generador continúa con el siguiente tick.

5.3 VALIDACIÓN Y PROCESAMIENTO DE DATOS

El sistema aplica un enfoque de contrato cerrado de telemetría: todo punto recibido debe ajustarse exactamente a la estructura y a las reglas de negocio definidas en TelemetryIn (service/models.py), sin campos adicionales ni combinaciones de valores inconsistentes. Esta sección describe el mecanismo de validación declarativa y semántica aplicado en la capa de aplicación, así como las transformaciones que sufre el dato antes de persistirse.

5.3.1 VALIDACIÓN DE DATOS

La validación de formato y estructura se resuelve de forma declarativa sobre los propios campos del modelo Pydantic, combinando tipos y restricciones de rango (Field(..., ge=..., le=..., lt=...)). Por ejemplo, la latitud, la longitud, la altitud, la velocidad respecto al suelo y el rumbo se acotan a sus rangos físicamente válidos directamente en la definición del modelo:

```
lat: float = Field(..., ge=-90.0, le=90.0)
lon: float = Field(..., ge=-180.0, le=180.0)
alt_m: float = Field(..., ge=0.0)
ground_speed_mps: float = Field(..., ge=0.0)
heading_deg: float = Field(..., ge=0.0, lt=360.0)
signal_quality: float | None = Field(None, ge=0.0, le=1.0)
msg_gap_ms: int | None = Field(None, ge=0)
```

(service/models.py:55-69, extracto)

Más allá del formato de cada campo por separado, la comprobación de consistencia y calidad de los datos se resuelve mediante un validador de modelo completo (@model_validator(mode="after")) que aplica el contrato de anomalías: si anomaly_label es falso, exige que anomaly_type sea 'none', que anomaly_metadata esté vacío, que anomaly_score sea exactamente 0 y que anomaly_source sea 'none'; si es verdadero, exige lo contrario (tipo distinto de 'none', metadato no vacío, score estrictamente positivo y origen distinto de 'none'), y además delega en _validate_anomaly_metadata la comprobación de que el contenido del metadato coincide exactamente con el esquema de claves esperado para ese tipo concreto de anomalía (por ejemplo, wind_shear exige delta_speed_mps y

delta_heading_deg, mientras que dropout exige gap_ms y missed_points), rechazando tanto claves ausentes como claves inesperadas:

```python
@model_validator(mode="after")
def validate_anomaly_contract(self):
    metadata = self.anomaly_metadata
    has_metadata = isinstance(metadata, dict) and len(metadata) > 0

    if not self.anomaly_label:
        if abs(self.anomaly_score) > 1e-9:
            raise ValueError("anomaly_score must be 0 when anomaly_label is false")
        if self.anomaly_type != "none":
            raise ValueError("anomaly_type must be 'none' when anomaly_label is false")
        if metadata not in (None, {}):
            raise ValueError("anomaly_metadata must be empty when anomaly_label is false")
        if self.anomaly_source != "none":
            raise ValueError("anomaly_source must be 'none' when anomaly_label is false")
        return self

    if self.anomaly_score <= 0:
        raise ValueError("anomaly_score must be > 0 when anomaly_label is true")
    if not has_metadata:
        raise ValueError("anomaly_metadata is required when anomaly_label is true")
    self._validate_anomaly_metadata(self.anomaly_type, metadata)
    return self
```

(service/models.py:96-129, extracto)

5.3.2 TRANSFORMACIÓN Y PREPARACIÓN DE LOS DATOS

Antes de persistirse, cada punto sufre dos transformaciones en el servidor. La primera es el cálculo de ingest_latency_ms cuando el emisor no lo proporciona explícitamente, derivado de la diferencia entre la marca de tiempo del servidor y la marca de tiempo del evento (t.ts):

```python
ingest_latency_ms = t.ingest_latency_ms
if ingest_latency_ms is None:
    ingest_latency_ms = int((server_ts - t.ts).total_seconds() * 1000)
```

(service/main.py:82-85)

La segunda es la aplicación de valores por defecto para los campos opcionales del contrato antes de construir la sentencia de inserción, mediante un único diccionario de valores por defecto (_TELEMETRY_DEFAULTS, service/db.py:165-189) fusionado con el payload recibido: payload = {**_TELEMETRY_DEFAULTS, **point}. Este mismo

diccionario se reutiliza sin modificaciones tanto en insert_telemetry_point como en insert_ml_training_point, garantizando que ambas tablas reciban exactamente los mismos valores por defecto para los campos que el emisor no informe explícitamente.

5.4 PERSISTENCIA Y GESTIÓN DE BASE DE DATOS

La persistencia se implementa sobre PostgreSQL 15 mediante psycopg3 y un pool de conexiones compartido. Esta sección describe el modelo de datos, la estructura de almacenamiento e indexación, las funciones de inserción y consulta, y los mecanismos de integridad y trazabilidad aplicados sobre los datos persistidos.

5.4.1 DISEÑO DEL MODELO DE DATOS

El esquema se organiza en cinco tablas (infra/sql/001_init.sql). flights almacena un registro por vuelo simulado (flight_id UUID como clave primaria, icao24, ventana temporal start_ts/end_ts, escenario dominante y scenario_params en JSONB). flight_segments referencia a flights por flight_id y registra los tramos temporales en los que el par (scenario, anomaly_type) se mantiene constante dentro de un vuelo, permitiendo reconstruir la secuencia de episodios anómalos sin recorrer punto a punto la tabla de telemetría. simulation_runs registra el ciclo de vida de cada ejecución del emisor (run_id, status restringido a 'ongoing'/'done'/'failed', total_flights, emitted_points). telemetry_points es la tabla operativa central, con un registro por punto de telemetría ingerido y treinta y dos columnas que cubren identidad, navegación, actitud, estado del transpondedor, etiquetas de escenario/anomalía y metadatos de contexto para el futuro entrenamiento de modelos. Por último, ml_training_points es una réplica desnormalizada de telemetry_points (vinculada mediante telemetry_point_id) pensada como superficie de consulta estable para el futuro consumo por parte de los modelos de detección de anomalías, desacoplada de la tabla operativa. El siguiente extracto muestra la definición de telemetry_points, incluyendo las columnas de identidad/navegación y dos de sus restricciones CHECK:

```
CREATE TABLE telemetry_points (
  id BIGSERIAL PRIMARY KEY,
  icao24 VARCHAR(8) NOT NULL,
  flight_id UUID NOT NULL REFERENCES flights(flight_id),
  seq BIGINT NOT NULL,
  ts TIMESTAMPTZ NOT NULL,
  lat DOUBLE PRECISION NOT NULL,
  lon DOUBLE PRECISION NOT NULL,
  alt_m DOUBLE PRECISION NOT NULL,
  ground_speed_mps DOUBLE PRECISION NOT NULL,
  heading_deg DOUBLE PRECISION NOT NULL,
  anomaly_label BOOLEAN NOT NULL DEFAULT FALSE,
  anomaly_type TEXT NOT NULL DEFAULT 'none',
  anomaly_metadata JSONB NOT NULL DEFAULT '{}'::jsonb,
  anomaly_score DOUBLE PRECISION NOT NULL DEFAULT 0.0,

  CONSTRAINT uq_flight_seq UNIQUE (flight_id, seq),
  CONSTRAINT ck_lat CHECK (lat BETWEEN -90 AND 90),
  CONSTRAINT ck_anomaly_contract CHECK (
    (anomaly_label = FALSE AND anomaly_type = 'none' AND anomaly_metadata = '{}'::jsonb)
    OR
    (anomaly_label = TRUE AND anomaly_type <> 'none' AND anomaly_metadata <> '{}'::jsonb)
  )
);
```

(infra/sql/001_init.sql:42-119, extracto)

El diagrama entidad-relación completo del esquema se muestra en la Figura 4.

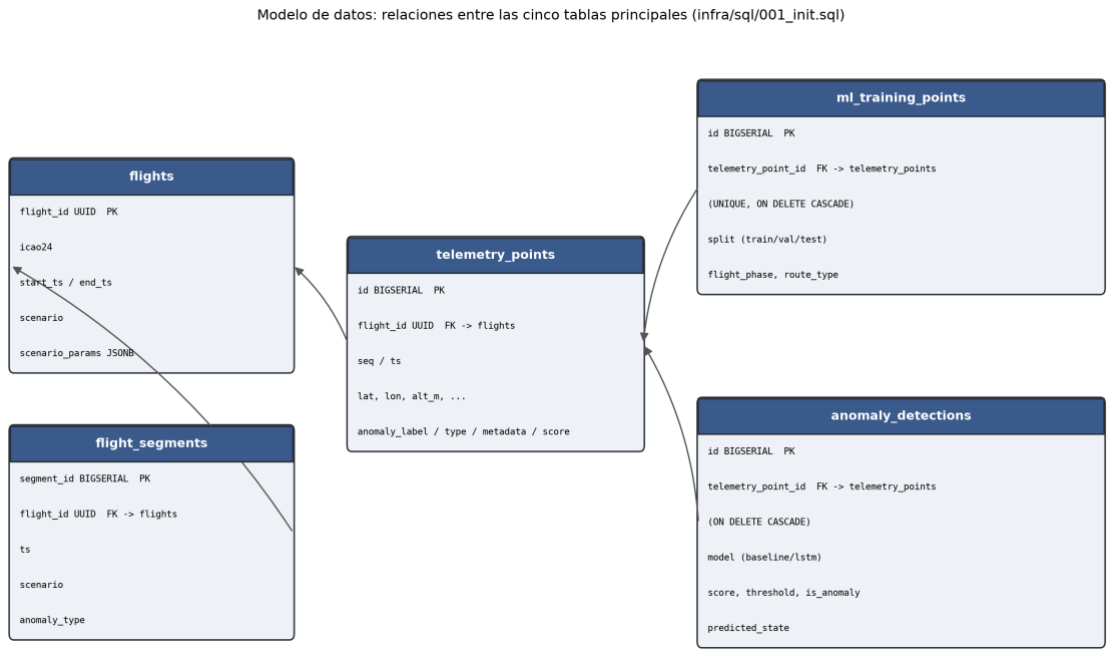

Figura 4.Diagrama entidad-relación del modelo de datos

5.4.2 ESTRUCTURA DE ALMACENAMIENTO

El almacenamiento actual es PostgreSQL 15 estándar, sin la extensión TimescaleDB habilitada (como ya se indica en el Capítulo 4, esa migración queda planteada como trabajo futuro), por lo que la eficiencia de las consultas depende íntegramente de la estrategia de indexación B-tree definida sobre las tablas. Sobre telemetry_points se definen cuatro índices: idx_tp_icao_ts e idx_tp_flight_ts (compuestos, para recuperar eficientemente la serie temporal de una aeronave o de un vuelo concreto), idx_tp_scenario (para filtrar por escenario) e idx_tp_ts (para consultas por ventana temporal global). Sobre ml_training_points se definen otros cuatro: idx_mltp_ts e idx_mltp_flight_ts (equivalentes a los anteriores), idx_mltp_label (compuesto sobre anomaly_label y anomaly_type, pensado para las consultas de selección de ejemplos por clase que requerirá el entrenamiento de modelos) e idx_mltp_route_phase (compuesto sobre route_type y flight_phase, para análisis segmentados por tipo de ruta o fase de vuelo). Sobre flights y flight_segments se definen índices adicionales (idx_flights_icao_start, idx_flights_scenario, idx_seg_flight_time) orientados a las mismas consultas por aeronave y por ventana temporal:

```
CREATE INDEX idx_tp_icao_ts ON telemetry_points (icao24, ts);
CREATE INDEX idx_tp_flight_ts ON telemetry_points (flight_id, ts);
CREATE INDEX idx_tp_scenario ON telemetry_points (scenario);
CREATE INDEX idx_tp_ts ON telemetry_points (ts);

CREATE INDEX idx_mltp_label ON ml_training_points (anomaly_label, anomaly_type);
CREATE INDEX idx_mltp_route_phase ON ml_training_points (route_type, flight_phase);
```

(infra/sql/001_init.sql:121-124, 166-167)

5.4.3 INSERCIÓN Y CONSULTA DE INFORMACIÓN

Toda la escritura se realiza mediante consultas parametrizadas (marcadores %(nombre)s de psycopg3), lo que evita construcción de SQL por concatenación e inyección de código. ensure_flight ilustra el patrón de escritura idempotente empleado en el proyecto: una sentencia INSERT ... ON CONFLICT (flight_id) DO UPDATE que, ante un vuelo ya existente, únicamente extiende end_ts hacia adelante mediante GREATEST() y evita sobrescribir scenario_params si ya se había establecido, de forma que llamadas repetidas

para el mismo flight_id (una por cada punto de ese vuelo) sean seguras y no pierdan información previamente escrita:

```
def ensure_flight(flight_id, icao24, start_ts, scenario="normal",
                  scenario_params=None, end_ts=None):
    with POOL.connection() as conn, conn.cursor() as cur:
        cur.execute(
            """
            INSERT INTO flights (flight_id, icao24, start_ts, end_ts, scenario,
scenario_params)
            VALUES (%(flight_id)s, %(icao24)s, %(start_ts)s, %(end_ts)s, %(scenario)s,
%(scenario_params)s)
            ON CONFLICT (flight_id) DO UPDATE SET
              end_ts = COALESCE(
                GREATEST(flights.end_ts,        EXCLUDED.end_ts),        flights.end_ts,
EXCLUDED.end_ts
              ),
              scenario_params = CASE
                WHEN flights.scenario_params = '{}'::jsonb THEN EXCLUDED.scenario_params
                ELSE flights.scenario_params
              END;
            """,
            {"flight_id": flight_id, "icao24": icao24, "start_ts": start_ts,
             "end_ts": end_ts, "scenario": scenario,
             "scenario_params": Json(scenario_params or {})},
        )
```

(service/db.py:19-56, extracto)

insert_telemetry_point ejecuta una única sentencia INSERT ... RETURNING id sobre las treinta y dos columnas de telemetry_points, y devuelve el identificador generado para encadenar la inserción del registro espejo en ml_training_points. insert_ml_training_point reutiliza ese id como clave foránea y aplica ON CONFLICT (telemetry_point_id) DO NOTHING, de forma que un reintento accidental sobre el mismo punto no produzca duplicados en la tabla de entrenamiento. upsert_flight_segment sigue un patrón distinto: primero bloquea con SELECT ... FOR UPDATE el último segmento del vuelo, y decide en código Python si debe extender ese segmento (si scenario y anomaly_type no han cambiado) o crear uno nuevo, evitando condiciones de carrera cuando llegan puntos consecutivos del mismo vuelo.

5.4.4 GESTIÓN DE INTEGRIDAD Y TRAZABILIDAD DE LOS DATOS

La integridad de los datos se refuerza en tres capas independientes. La primera es la validación de Pydantic descrita en el apartado anterior, que impide que un payload inconsistente llegue siquiera a construirse como llamada a base de datos. La segunda es un conjunto de restricciones CHECK definidas directamente en el esquema (ck_lat, ck_lon, ck_heading, ck_anomaly_metadata_obj, ck_anomaly_contract, ck_anomaly_source_policy, ck_forced_extreme_policy, ck_anomaly_score_policy), que reproducen a nivel de base de datos las mismas reglas de negocio ya comprobadas por Pydantic: esta redundancia intencionada (defensa en profundidad) garantiza que ningún dato inconsistente pueda persistirse aunque exista un error en la capa de aplicación que sortee la validación de Pydantic. La tercera es la trazabilidad relacional: la restricción UNIQUE(flight_id, seq) impide duplicar o reordenar puntos dentro de un mismo vuelo, y las claves foráneas (flight_segments.flight_id y telemetry_points.flight_id hacia flights.flight_id; ml_training_points.telemetry_point_id hacia telemetry_points.id, con ON DELETE CASCADE) garantizan que ningún segmento, punto o registro de entrenamiento pueda existir sin su vuelo o punto de origen, y que la eliminación de un punto de telemetría propague automáticamente la eliminación de su réplica en ml_training_points:

```
CONSTRAINT uq_flight_seq UNIQUE (flight_id, seq)

telemetry_point_id BIGINT NOT NULL UNIQUE
  REFERENCES telemetry_points(id) ON DELETE CASCADE
```

(infra/sql/001_init.sql:89-128, extracto)

Una limitación conocida del diseño actual es que estas cuatro escrituras (ensure_flight, upsert_flight_segment, insert_telemetry_point, insert_ml_training_point) no comparten una única transacción: cada función abre y confirma su propia conexión del pool de forma independiente (with POOL.connection() as conn, conn.cursor() as cur:), por lo que la escritura de un punto no es atómica de extremo a extremo. Si una excepción se produce después de que telemetry_points ya se haya confirmado pero antes de que se complete la inserción en ml_training_points, ambas tablas quedan temporalmente desincronizadas para

ese punto, algo a corregir en una futura iteración envolviendo las cuatro llamadas en una única transacción explícita.

La tabla anomaly_detections, introducida junto con el módulo de detección de anomalías (apartado 5.6), sigue el mismo patrón de integridad: clave foránea hacia telemetry_points con ON DELETE CASCADE, y una restricción UNIQUE(telemetry_point_id, model) que garantiza que un mismo punto no pueda tener dos veredictos distintos del mismo modelo, incluso si la puntuación en tiempo real se reintentase por error.

5.5 MONITOREO Y OBSERVABILIDAD DEL SISTEMA

La observabilidad del sistema se apoya en la instrumentación directa del servicio de ingesta mediante la librería prometheus-client, un servidor Prometheus dedicado que realiza scraping periódico, y Grafana como capa de visualización, con datasources y dashboards provisionados de forma automática y reproducible.

5.5.1 MÉTRICAS MONITORIZADAS

El servicio define cuatro métricas Prometheus a nivel de módulo (service/main.py:24-27): REQS, un Counter etiquetado por path, method y code que cuenta cada petición HTTP atendida; LAT, un Histogram etiquetado por path y method que mide la duración de la petición mediante el gestor de contexto LAT.labels(...).time(); VALID, un Counter sin etiquetas que se incrementa una vez por cada punto de telemetría persistido correctamente; e INVALID, un Counter que se incrementa ante cualquier excepción durante la fase de persistencia. Es importante señalar que INVALID no contabiliza los rechazos por validación de Pydantic (HTTP 422), puesto que esos casos nunca llegan a ejecutar el cuerpo de la función ingest(): mide exclusivamente los fallos ocurridos una vez el payload ya ha sido aceptado como sintácticamente válido.

```
REQS = Counter("http_requests_total", "HTTP requests", ["path", "method", "code"])
LAT = Histogram("http_request_duration_seconds", "Request latency", ["path", "method"])
VALID = Counter("ingest_valid_points_total", "Valid telemetry points")
INVALID = Counter("ingest_invalid_points_total", "Invalid telemetry points")
```

(service/main.py:24-27)

Junto a estas cuatro métricas originales, la integración del módulo de detección de anomalías añadió otras dos: anomaly_detections_total, un contador etiquetado por modelo que registra cuántas puntuaciones en tiempo real se han realizado, y anomaly_flagged_total, que cuenta específicamente cuántas de ellas resultaron en un veredicto anómalo, permitiendo vigilar en Grafana y Prometheus la tasa de activación de cada modelo sin necesidad de consultar directamente la tabla anomaly_detections.

5.5.2 INSTRUMENTACIÓN Y SUPERVISIÓN DEL SISTEMA

Estas métricas se exponen en formato de texto Prometheus a través del endpoint GET /metrics (service/main.py:47-53), que delega en prometheus_client.generate_latest(). Un servidor Prometheus independiente (imagen prom/prometheus:v2.52.0) realiza scraping de ese endpoint cada diez segundos, según la configuración declarada en infra/prometheus/prometheus.yml:

```
global:
  scrape_interval: 10s
  evaluation_interval: 30s

scrape_configs:
  - job_name: "telemetry-service"
    static_configs:
      - targets: ["service:8000"]
        labels:
          component: "ingest_api"
```

(infra/prometheus/prometheus.yml)

Grafana se provisiona automáticamente al arrancar el contenedor, mediante los ficheros bind-montados en infra/grafana/provisioning: datasources/datasource.yml declara dos fuentes de datos (una conexión directa a Postgres para consultar las tablas del sistema, y una conexión al servidor Prometheus para las métricas operativas), y dashboards/ contiene cuatro dashboards ya construidos y versionados en el repositorio: dashboard_flights.json

(series temporales de altitud, velocidad, rumbo, velocidad vertical, confianza de escenario y score de anomalía para un vuelo seleccionado, junto con una tabla de violaciones de contrato), dashboard_simulation_status.json (estado en vivo de la última ejecución de simulación registrada en simulation_runs), y dashboard_tables.json (explorador de las tablas principales, con recuentos de violaciones de contrato y distribución de escenarios/tipos de anomalía en los últimos treinta minutos). Al estar versionados como JSON en el repositorio, estos dashboards se reconstruyen de forma idéntica en cualquier entorno donde se despliegue el sistema, sin configuración manual en la interfaz de Grafana.

5.6 DETECCIÓN DE ANOMALÍAS

Esta sección describe la implementación real del módulo de detección de anomalías: la preparación del conjunto de datos etiquetado, el diseño y la comparación de los cuatro modelos candidatos planteados en el Capítulo 4, la implementación del modelo finalmente seleccionado y su integración como servicio de puntuación en tiempo real dentro de la arquitectura de ingesta descrita en este mismo capítulo.

5.6.1 INTEGRACIÓN DEL MÓDULO DE DETECCIÓN DENTRO DE LA ARQUITECTURA

El módulo de detección se implementa como un componente adicional dentro del propio servicio de ingesta (service/detector.py), en lugar de como un servicio independiente, para minimizar la complejidad operativa en esta fase del proyecto. Los resultados de cada modelo se persisten en una tabla nueva, anomaly_detections, deliberadamente separada de las columnas anomaly_label/anomaly_type de telemetry_points: estas últimas registran la verdad terreno generada por el emisor sintético, mientras que anomaly_detections registra lo que el modelo desplegado cree en el momento de la ingesta, evitando confundir ambos conceptos. La tabla incluye una restricción UNIQUE(telemetry_point_id, model) y una clave foránea con ON DELETE CASCADE hacia telemetry_points, siguiendo el mismo patrón de integridad ya descrito para ml_training_points.

La puntuación se invoca desde el propio manejador de /ingest, inmediatamente después de persistir el punto de telemetría, pero envuelta en su propio bloque try/except independiente del resto de la función: un fallo del modelo o de su almacenamiento nunca debe impedir que la ingesta se complete ni contabilizarse como punto inválido:

```
new_id = insert_telemetry_point(payload)
insert_ml_training_point(new_id, payload)

# Deteccion de anomalias en tiempo real: best-effort, nunca debe
# tumbar la ingesta si el modelo o su almacenamiento fallan.
try:
    scoring_point = {**payload, "ts_epoch": t.ts.timestamp()}
    for det in detector.score_point(str(t.flight_id), scoring_point):
        insert_anomaly_detection(new_id, str(t.flight_id), t.seq, det)
        DETECTIONS.labels(det["model"]).inc()
        if det["is_anomaly"]:
            ANOMALIES_FLAGGED.labels(det["model"]).inc()
except Exception:
    pass
```

(service/main.py:127-140)

Se sirven dos modelos simultáneamente: baseline, que no requiere historial y puntúa desde el primer punto de cada vuelo, y lstm, el mejor de los cuatro candidatos evaluados, que necesita 50 puntos de contexto antes de emitir su primera predicción. El estado necesario para reconstruir ese contexto (la ventana de los últimos puntos de cada vuelo) se mantiene en memoria del propio proceso, mediante un diccionario ordenado con expulsión LRU acotada a 2000 vuelos simultáneos (service/detector.py), evitando un crecimiento de memoria sin límite en un servicio de larga duración. Los pesos entrenados, el escalador de normalización y el umbral calibrado se cargan una única vez al arrancar el servicio, desde un volumen de solo lectura montado en el contenedor (code/ml/artifacts, ver infra/docker-compose.yml), de modo que reentrenar un modelo no exige reconstruir la imagen del servicio, solo reiniciar el contenedor. Esta integración tiene un coste arquitectónico real: la imagen del servicio de ingesta ha dejado de ser tan ligera como se describía en capítulos anteriores, al incorporar PyTorch como dependencia; separar la detección en un servicio propio quedaría como evolución natural si se quisiera recuperar esa ligereza.

5.6.2 PREPARACIÓN DE DATOS PARA EL ANÁLISIS DE ANOMALÍAS

Antes de entrenar cualquier modelo fue necesario resolver dos problemas de preparación de datos: cómo particionar el conjunto en entrenamiento/validación/prueba sin filtrar información entre conjuntos, y qué representación numérica de cada punto de telemetría alimentar a los modelos.

La partición (code/ml/prepare_split.py) se realiza a nivel de vuelo completo, nunca de punto suelto: mezclar puntos de un mismo vuelo entre entrenamiento y prueba filtraría la trayectoria de ese vuelo entre ambos conjuntos. Dado que un vuelo puede contener varios tipos de anomalía distintos (el emisor encadena escenarios, por ejemplo turbulence → wind_shear → route_deviation → engine_failure), la partición no puede basarse únicamente en el tipo dominante de cada vuelo: se clasifica cada tipo de anomalía en un nivel de cobertura según en cuántos vuelos distintos aparece (tier A si aparece en tres o más vuelos, con al menos uno garantizado en cada conjunto; tier B con dos vuelos, garantizando train y test pero renunciando explícitamente a val; tier C con un único vuelo, donde ningún algoritmo puede garantizar más de un conjunto). En el dataset actual, dropout y spike resultaron ser tier C: su único vuelo se reserva íntegramente a test, visible en la comparación final pero sin ningún ejemplo de entrenamiento, una limitación del dataset documentada explícitamente, no oculta:

```python
def classify_tiers(type_flight_count):
    tiers = {}
    for t, n in type_flight_count.items():
        if n >= TIER_A_MIN_FLIGHTS:
            tiers[t] = "A"
        elif n == 2:
            tiers[t] = "B"
        else:
            tiers[t] = "C"
    return tiers
```

(code/ml/prepare_split.py:83-91)

Sobre esa partición, code/ml/features.py construye dos representaciones distintas del mismo dato. La primera, plana, es un vector por punto con doce variables físicas y cuatro derivadas (delta de altitud, velocidad, rumbo —con manejo circular para el salto 0°/360°—

y tiempo respecto al punto anterior del mismo vuelo), usada por el baseline estadístico, Isolation Forest y Local Outlier Factor. La segunda, secuencial, agrupa esas mismas dieciséis variables en ventanas deslizantes de 50 puntos consecutivos por vuelo, usada por la LSTM. Ambas se normalizan con media y desviación típica calculadas únicamente sobre el conjunto de entrenamiento.

Durante la validación del sistema en un dashboard de Grafana en vivo se detectó un problema real en esta preparación: el primer punto de cada vuelo, al no tener punto anterior, recibía un delta de tiempo de 0.0 como valor centinela. Dado que en este dataset los puntos llegan con una cadencia prácticamente constante de un segundo, esa variable tiene una desviación típica casi nula en entrenamiento, y un 0.0 en esa escala producía un z-score de aproximadamente -146 en el primer punto de cualquier vuelo, disparando falsos positivos sistemáticos sin relación con ninguna anomalía real. La corrección consistió en imputar, tanto en el entrenamiento como en el servicio en tiempo real, con la mediana de los deltas realmente observados en lugar de un valor centinela arbitrario, coherente con el mismo criterio ya usado para las variables opcionales ausentes:

```
if state.prev_raw is None:
    # No previous point yet: impute with the train-median delta instead
    # of a raw 0.0, which would land ~146 std devs from delta_t_s's
    # near-constant ~1s cadence and manufacture a false extreme score
    # on every flight's very first point.
    delta_alt_m = _SCALER["fill_values"]["delta_alt_m"]
    delta_speed = _SCALER["fill_values"]["delta_ground_speed_mps"]
    delta_heading = _SCALER["fill_values"]["delta_heading_deg"]
    delta_t = _SCALER["fill_values"]["delta_t_s"]
```

(service/detector.py:121-129)

5.6.3 DISEÑO Y COMPARATIVA DE DIFERENTES MODELOS

Los cuatro candidatos comparten un arnés de evaluación común (code/ml/metrics.py): el umbral de decisión se calibra únicamente sobre el conjunto de validación, maximizando F1, y las métricas que se reportan (precisión, exhaustividad, F1 y área bajo la curva precisión-exhaustividad, más adecuada que el área ROC dado el desbalanceo de clases) se calculan sobre el conjunto de prueba con ese umbral ya fijado, sin

volver a tocarlo. Además de las métricas globales, se calcula la exhaustividad desglosada por tipo de anomalía, para que un modelo que solo acierte en los tipos más frecuentes no parezca mejor de lo que realmente es.

El Modelo 1 (baseline estadístico) no requiere entrenamiento: su puntuación es directamente el máximo valor absoluto de z-score entre las dieciséis variables normalizadas de un punto. El Modelo 2 (Isolation Forest, scikit-learn, 200 árboles) se entrena sobre los 4,55 millones de puntos de entrenamiento en menos de 2 segundos. El Modelo 4 (Local Outlier Factor) no es viable sobre el conjunto completo por su coste $O(n^2)$, por lo que se entrena con novelty=True sobre una muestra estratificada de 50.000 puntos que conserva la proporción de anomalías del conjunto original. El Modelo 3 (LSTM) se entrena sobre las ventanas secuenciales completas del conjunto de entrenamiento en unos 35 segundos, acelerado por GPU (Metal, en Apple Silicon).

Resultado final de la comparativa (umbral calibrado en validación, métricas en el conjunto de prueba):

modelo	precision	recall	f1	AP	fit(s)
modelo1_baseline_estadistico	0.527	0.421	0.468	0.380	0.0
modelo2_isolation_forest	0.610	0.313	0.414	0.397	1.9
modelo4_local_outlier_factor	0.236	0.435	0.306	0.302	0.8
modelo3_lstm	0.989	0.695	0.816	0.956	35.5

El desglose por tipo de anomalía explica el porqué de esta diferencia. El baseline y Isolation Forest rinden bien en anomalías de magnitud evidente (engine_failure: recall 0,95 y 0,77; multi: 0,997 y 0,92), donde un valor puntual ya se sale claramente de rango, pero fallan casi por completo en anomalías gruduales o contextuales (gnss_drift: 0,001 y 0,001; wind_shear: 0,145 y 0,026), que solo son visibles observando la evolución temporal de la variable, no su valor instantáneo. La LSTM, al procesar la secuencia completa, domina precisamente en esas categorías temporales (wind_shear: 0,970; route_deviation: 0,862), que además concentran la mayoría del volumen de anomalías del dataset. Local Outlier Factor,

limitado por la muestra reducida sobre la que puede entrenarse, obtiene resultados intermedios y más irregulares entre tipos (Figura 5).

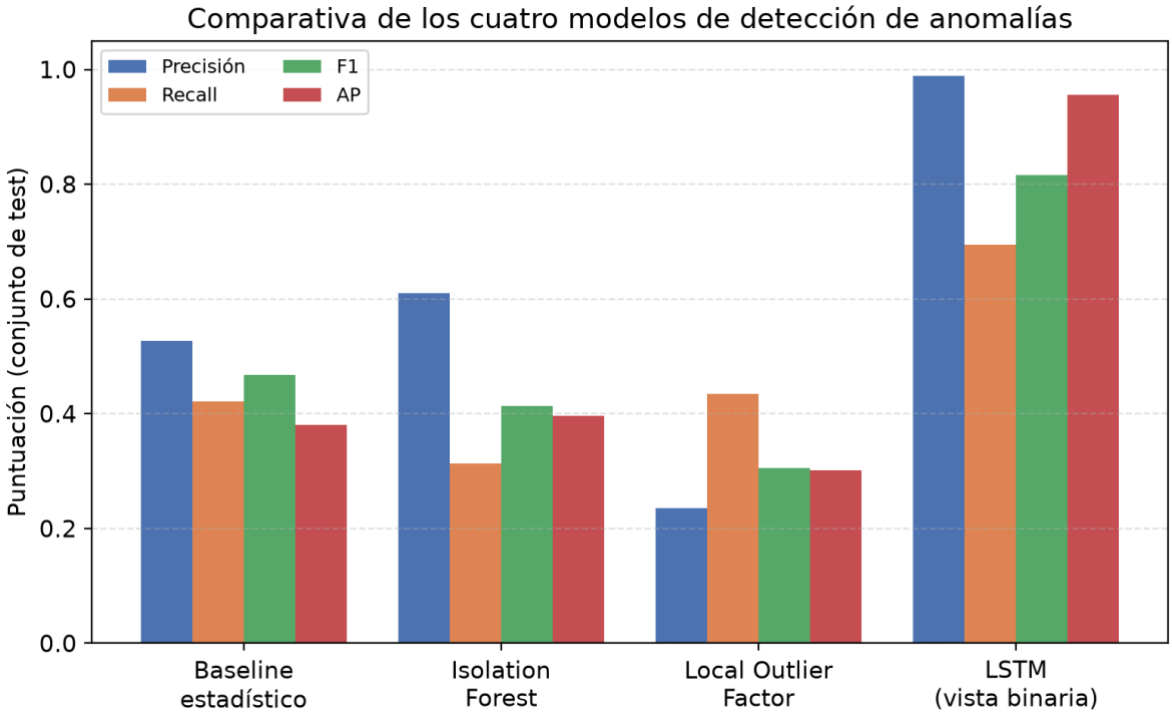

Figura 5. Comparativa de precisión, recall, F1 y AP de los cuatro modelos

5.6.4 IMPLEMENTACIÓN DEL MODELO SELECCIONADO

El modelo seleccionado es la LSTM, por ser el único de los cuatro capaz de aprovechar la estructura secuencial de los datos y el que mejor resultado global obtiene. A diferencia de un detector binario que solo señala si un punto es anómalo, se implementó como clasificador multi-clase: predice directamente en qué estado concreto se encuentra el vuelo (uno de nueve: 'none' más los ocho tipos de anomalía observados), no solo si algo va mal. La arquitectura es una única capa LSTM con 64 unidades ocultas, seguida de una capa lineal que proyecta el último estado oculto sobre las nueve clases:

```python
class AnomalyLSTM(nn.Module):
    def __init__(self, n_features, n_classes, hidden_size, num_layers):
        super().__init__()
        self.lstm    =    nn.LSTM(n_features,    hidden_size,    num_layers=num_layers,
batch_first=True)
        self.head = nn.Linear(hidden_size, n_classes)

    def forward(self, x):
        _, (h_n, _) = self.lstm(x)
        return self.head(h_n[-1])  # logits, shape (batch, n_classes)
```

(code/ml/model_lstm.py:42-51)

El entrenamiento usa CrossEntropyLoss ponderada por la frecuencia inversa de cada clase (para compensar que 'none' representa más del 95% de las ventanas), optimizador Adam y ocho épocas, seleccionando al final el punto de entrenamiento con mejor área bajo la curva precisión-exhaustividad en validación, no simplemente la última época. Los pesos del modelo se guardan junto con la lista de clases y la configuración de la arquitectura en un único fichero de checkpoint (lstm_state.pt), que es lo que el servicio de ingesta carga en tiempo real.

El informe de clasificación sobre el conjunto de prueba muestra una exactitud global del 94,8%, inflada por el peso de la clase 'none'; las métricas relevantes son la F1 macro (0,372, sin ponderar por frecuencia) y el desglose por clase, que confirma que el modelo generaliza razonablemente sobre las clases con suficientes ejemplos, pero es honestamente incapaz de predecir dropout y spike: al ser tipos tier C sin ningún ejemplo en entrenamiento (ver el apartado anterior), su precisión, exhaustividad y F1 son exactamente 0,000 en el informe, no un resultado pobre sino una consecuencia directa y esperada de la escasez de datos, ya anticipada al diseñar la partición.

5.6.5 LÓGICA DE DETECCIÓN Y GENERACIÓN DE RESULTADOS

En el servicio, cada punto de telemetría recibido se transforma en el mismo vector de dieciséis variables usado en el entrenamiento (incluidas las cuatro derivadas), utilizando el estado en memoria del vuelo (último punto conocido, para calcular los deltas) y el escalador entrenado. El baseline puntúa ese vector inmediatamente, sin necesitar historial. La LSTM

lo añade a la ventana deslizante del vuelo y, en cuanto esta alcanza cincuenta puntos, calcula una distribución de probabilidad sobre las nueve clases mediante softmax, toma la de mayor probabilidad como estado predicho y esa misma probabilidad como medida de confianza:

```python
if len(state.window) == _WINDOW:
    with torch.no_grad():
        x = torch.tensor([list(state.window)], dtype=torch.float32)
        logits = _LSTM_MODEL(x)
        probs = torch.softmax(logits, dim=-1)[0]
        pred_idx = int(torch.argmax(probs).item())
        confidence = float(probs[pred_idx].item())
    predicted_state = _CLASSES[pred_idx]
    detections.append({
        "model": "lstm",
        "score": confidence,
        "threshold": None,
        "is_anomaly": predicted_state != "none",
        "predicted_state": predicted_state,
    })
```

(service/detector.py:172-186)

Antes de acumular cincuenta puntos, un vuelo recién iniciado solo dispone del veredicto del baseline: es la zona ciega de la LSTM ya discutida al plantear la viabilidad del tiempo real, inherente a cualquier modelo que necesite contexto histórico. Se midió la latencia real de puntuar una única ventana (el caso de uso en producción, no un lote de entrenamiento): entre 0,36 ms en CPU y 0,9 ms en GPU, muy por debajo de cualquier restricción de tiempo real razonable para el volumen de tráfico actual del sistema; el cuello de botella de la detección en vivo no es el cómputo del modelo, sino el propio proceso de ingesta HTTP.

Cada resultado se persiste en anomaly_detections con el estado predicho, la confianza y el veredicto binario derivado, y se refleja en el dashboard de Grafana descrito en la siguiente sección mediante paneles dedicados: la evolución temporal del score de ambos modelos, el último veredicto de cada uno, una tabla con las detecciones recientes y una línea de tiempo por colores que muestra cómo evoluciona el estado predicho por la LSTM a lo largo del vuelo seleccionado.

5.7 VISUALIZACIÓN E INTERFAZ DE USUARIO

El sistema incorpora un panel de visualización en tiempo real, desarrollado en React, que permite observar el estado de las aeronaves actualmente simuladas y ser alertado de inmediato si alguna de ellas entra en un estado anómalo. Esta sección describe su diseño, su composición en pantalla y el funcionamiento de su sistema de alarmas.

5.7.1 DISEÑO DEL FRONT-END

El front-end (code/frontend/) se implementa como una aplicación React 18 servida mediante Vite, desplegada como un servicio adicional de docker-compose.yml. Se optó por un modelo de sondeo periódico (polling) sobre el endpoint GET /flights/active en lugar de una conexión persistente (WebSocket o Server-Sent Events): dado que el propio backend expone únicamente una API REST y la cadencia de actualización necesaria es de pocos segundos, un polling simple resulta suficiente y evita añadir un canal de comunicación adicional (y su complejidad de gestión de reconexión) al servicio de ingesta.

El código se organiza en un pequeño número de módulos con responsabilidades bien delimitadas: App.jsx orquesta el sondeo periódico, mantiene el estado de los vuelos activos y decide cuándo disparar la alarma; FlightCard.jsx renderiza la tarjeta individual de cada aeronave; FlightMap.jsx dibuja el mapa e integra los iconos de las aeronaves mediante Leaflet y react-leaflet; stateStyles.js centraliza tanto el mapeo de colores por estado como la lógica de qué fases se consideran fiables para confiar en la predicción del modelo (véase la sección 6.6.2); y useAlarmSound.js encapsula el aviso sonoro mediante la Web Audio API. El servicio se empaqueta con una imagen node:20-alpine (frontend/Dockerfile) y expone el servidor de desarrollo de Vite en el puerto 5173, configurado con allowedHosts: true para admitir peticiones tanto desde localhost como desde el nombre del servicio dentro de la red de docker-compose.

5.7.2 PANEL DE VISUALIZACIÓN DE AERONAVES

El componente principal (App.jsx) sondea el endpoint cada dos segundos y, con cada respuesta, actualiza la lista de vuelos activos y recalcula qué aeronaves han entrado de nuevo en un estado preocupante:

```
const currentAnomalous = new Set(
  data.flights.filter(isConcerning).map((f) => f.flight_id)
)
// Only beeps for flights ENTERING a concerning state right now,
// not on every poll while they remain concerning.
const isNewAlarm = [...currentAnomalous].some((id) => !seenAnomalousIds.current.has(id))
if (isNewAlarm && soundOn) beep()
seenAnomalousIds.current = currentAnomalous
```

(frontend/src/App.jsx:30-37, extracto)

Cada aeronave se representa mediante una tarjeta (FlightCard.jsx) con su indicador STATUS (el estado predicho, coloreado según su severidad), el nivel de confianza de la predicción, sus valores cinemáticos actuales (altitud, velocidad, rumbo, posición) y, en el pie de la tarjeta, el campo scenario de verdad de terreno generado por el simulador, que permanece siempre visible con independencia del estado predicho por el modelo. Las tarjetas se ordenan de forma estable por identificador de vuelo, no por su estado de alarma:

```
// Sort by a key that never changes while the flight is active (flight_id),
// not by current status: sorting by "is concerning" made cards jump
// position every time a flight's state changed, which was disorienting.
const sorted = [...flights].sort((a, b) => a.flight_id.localeCompare(b.flight_id))
```

(frontend/src/App.jsx:57-60, extracto)

El mapa (FlightMap.jsx), construido con Leaflet y react-leaflet, sitúa cada aeronave en sus coordenadas reales mediante un icono SVG rotado según su rumbo actual y coloreado con el mismo esquema que las tarjetas:

```
function planeIcon(color, headingDeg, alert) {
  const rotation = headingDeg ?? 0
  const glow = alert ? `filter: drop-shadow(0 0 6px ${color});` : ''
  const html = `
    <div style="transform: rotate(${rotation}deg); ${glow}">
      <svg width="26" height="26" viewBox="0 0 24 24">
        <path d="M12 2 L15 10 L22 13 L15 14.5 L14 21 L12 18 L10 21 L9 14.5 L2 13 L9 10 Z"
              fill="${color}" stroke="#0f1115" stroke-width="0.8"/>
      </svg>
    </div>
  `
  return L.divIcon({ html, className: 'plane-icon', iconSize: [26, 26], iconAnchor: [13,
13] })
}
```

(frontend/src/FlightMap.jsx:12-29, extracto)

El mapa ajusta automáticamente su encuadre a la primera carga de datos (componente AutoFitOnce), para mostrar de inmediato el conjunto completo de vuelos activos sin necesidad de interacción manual, pero sin volver a reajustar el zoom en sondeos posteriores, para no interferir con la navegación manual del usuario sobre el mapa.

5.7.3 VISUALIZACIÓN DE ANOMALÍAS DETECTADAS

La lógica que decide si un vuelo debe considerarse preocupante se centraliza en stateStyles.js, y se apoya en la fiabilidad de la predicción según la fase de vuelo, medida empíricamente en la sección 6.6:

```
const RELIABLE_PREDICTION_PHASES = new Set(['climb', 'cruise', 'emergency'])

export function isReliablePhase(phase) {
  return RELIABLE_PREDICTION_PHASES.has(phase)
}

export function isConcerning(flight) {
  if (!isReliablePhase(flight.flight_phase)) return false
  const state = flight.predicted_state
  return Boolean(state && state !== 'none')
}
```

(frontend/src/stateStyles.js:34-44, extracto)

Cuando al menos un vuelo cumple isConcerning(), la interfaz muestra un banner rojo pulsante en la parte superior del panel con el listado de aeronaves afectadas y su estado, y resalta con un borde rojo tanto la tarjeta como el icono en el mapa de cada vuelo concernido.

De forma adicional y opcional, useAlarmSound.js dispara un aviso sonoro mediante la Web Audio API (un oscilador cuadrado de 880 Hz con una breve rampa de atenuación) únicamente cuando un vuelo entra de nuevo en un estado anómalo, no en cada sondeo mientras el vuelo permanece en dicho estado, evitando así un sonido repetitivo mientras la situación persiste. El AudioContext necesario para reproducir sonido se crea de forma perezosa, en el primer clic del usuario sobre el botón de activar sonido, dado que los navegadores bloquean la reproducción de audio hasta que se produce una interacción explícita con la página.

El esquema de colores por estado (STATE_STYLES en stateStyles.js) es intencionadamente el mismo que el del panel 'Predicted State - LSTM' del dashboard de Grafana descrito en la sección 6.2.3, de modo que ambas vistas del sistema, el panel de operación en React y el dashboard de observabilidad en Grafana, resulten consistentes entre sí para un mismo operador que alterne entre ambas.

5.8 DEVOPS, DESPLIEGUE E INTEGRACIÓN CONTINUA

El sistema se despliega íntegramente mediante contenedores Docker, y su corrección se valida de forma automática en cada cambio mediante un pipeline de integración continua en GitHub Actions.

5.8.1 CONTENERIZACIÓN DE LOS SERVICIOS

Los dos servicios propios del proyecto disponen de su propio Dockerfile. service/Dockerfile construye la imagen del servicio de ingesta a partir de python:3.11-slim, instala service/requirements.txt, copia el código de service/ y arranca uvicorn sobre el puerto 8000:

```
FROM python:3.11-slim AS base
WORKDIR /app
COPY service/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY service /app
EXPOSE 8000
ENV PYTHONPATH=/app
CMD ["uvicorn", "main:app", "--host=0.0.0.0", "--port=8000"]
```

(service/Dockerfile)

synthetic/Dockerfile sigue el mismo patrón para el emisor, aunque instala únicamente la dependencia requests directamente por pip (no dispone de un requirements.txt propio) y arranca por defecto flight_emitter.py. Ambas imágenes se construyen desde infra/docker-compose.yml indicando context: .. (la raíz de code/) y el Dockerfile correspondiente, de forma que ambos servicios comparten el mismo contexto de build. Junto a ellas, se emplean tres imágenes de terceros sin modificar (postgres:15, grafana/grafana:11.1.0, prom/prometheus:v2.52.0), y dos volúmenes nombrados (db_data y grafana_data) para persistir los datos de Postgres y la configuración de Grafana entre reinicios del stack.

5.8.2 PIPELINE DE INTEGRACIÓN Y ENTREGA CONTINUA (CI/CD)

La integración continua se define en .github/workflows/github-actions.yml, con disparo en cada push a master y en cada pull request. El pipeline se organiza en dos jobs encadenados: build-test instala Python 3.11 y las dependencias del servicio junto con pytest y ruff, instala el binario de docker-compose v2.20.0 descargándolo directamente desde las releases de GitHub, ejecuta el linter (ruff check service), y a continuación lanza la suite de pruebas en dos fases: primero las pruebas rápidas (pytest -q -m "not slow"), y después las pruebas de carga marcadas explícitamente como slow (pytest -q -m slow, maxfail=1), deteniéndose en el primer fallo dado su mayor coste. El segundo job, docker, depende del anterior mediante needs: build-test y construye la imagen del servicio (docker build -t tfm-service:ci -f service/Dockerfile .), validando así que el servicio no solo pasa sus pruebas unitarias y de integración, sino que además es empaquetable como contenedor sin errores:

```yaml
name: ci
on:
  push:
    branches: [ master ]
  pull_request:

jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.11'
      - name: Install deps
        run: |
          pip install -r service/requirements.txt pytest ruff
      - name: Install docker-compose
        run: |
          sudo curl -L "https://github.com/docker/compose/releases/download/v2.20.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
          sudo chmod +x /usr/local/bin/docker-compose
      - name: Lint
        run: ruff check service
      - name: Run tests (fast)
        run: |
          pytest -q -m "not slow"
      - name: Run tests (slow load)
        run: |
          pytest -q -m slow --maxfail=1
  docker:
    runs-on: ubuntu-latest
    needs: build-test
    steps:
      - uses: actions/checkout@v4
      - name: Build image
        run: docker build -t tfm-service:ci -f service/Dockerfile .
```

(.github/workflows/github-actions.yml)

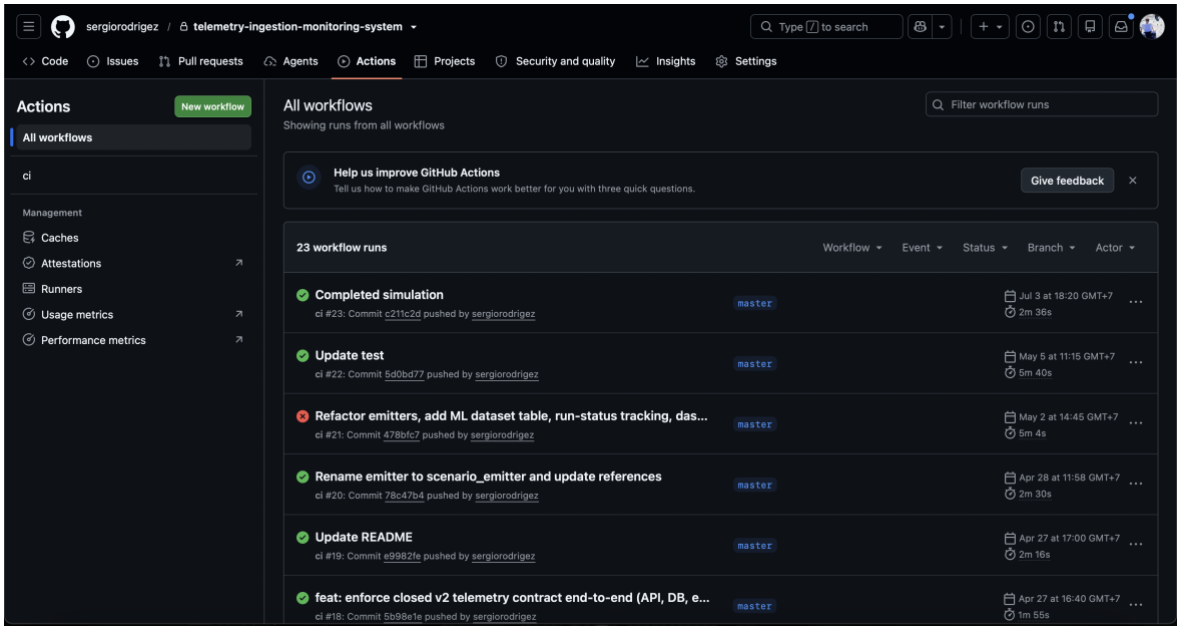

Figura 6. Ejecución en verde de GitHub Actions

CAPÍTULO 6 – ANÁLISIS DE RESULTADOS

6.1 INTRODUCCIÓN Y METODOLOGÍA DE EVALUACIÓN

6.1.1 OBJETIVOS DEL ANÁLISIS

Este capítulo evalúa en qué medida el sistema desarrollado cumple los objetivos definidos en el Capítulo 2, siguiendo el mismo orden en que fue construyéndose el proyecto: en primer lugar, los resultados de la infraestructura base de ingesta, validación y persistencia de telemetría; a continuación, los resultados de la generación de datos sintéticos que alimenta el resto del sistema; después, el proceso de comparación y refinamiento de los modelos de detección de anomalías, desde el primer prototipo estadístico hasta el modelo multiclase finalmente integrado; y, por último, los resultados de la integración en tiempo real y de la interfaz de visualización, validados de forma conjunta mediante una demostración end-to-end del sistema completo.

Este orden progresivo permite mostrar no solo el resultado final del proyecto, sino también las decisiones de diseño y los hallazgos intermedios que lo motivaron, muchos de los cuales condicionaron directamente el estado actual del sistema (por ejemplo, la limitación

de cobertura de determinados tipos de anomalía en el dataset, o la necesidad de ajustar la duración simulada de los vuelos para que el modelo de detección en tiempo real funcionase correctamente).

6.1.2 CRITERIOS Y MÉTRICAS EMPLEADAS

Dado que el sistema combina componentes de naturaleza distinta, se emplea un criterio de evaluación específico para cada uno de ellos. Para la infraestructura de ingesta y persistencia se valora la corrección funcional, verificada mediante la batería de pruebas automatizadas del proyecto (code/tests), la tasa de errores bajo carga y la consistencia del contrato de telemetría entre la capa de aplicación y la base de datos. Para el dataset sintético se valora la cobertura de los distintos tipos de anomalía y el equilibrio de la partición entre entrenamiento, validación y test.

Para los modelos de detección de anomalías se emplean las métricas estándar de clasificación (precisión, recall, F1-score y AUC), calculadas tanto de forma agregada como desglosada por tipo de anomalía, con el fin de evitar que un modelo que solo acierta en los tipos mayoritarios del dataset parezca mejor de lo que realmente es. Para la integración en tiempo real se compara el rendimiento obtenido de forma offline sobre el conjunto de test con el rendimiento observado en producción sobre datos en streaming, dado que ambos escenarios no tienen por qué coincidir. Por último, para la interfaz de visualización se realiza una validación funcional cualitativa del panel de vuelos, el mapa en tiempo real y el sistema de alarmas.

6.2 RESULTADOS DE LA INFRAESTRUCTURA BASE DE INGESTA Y PERSISTENCIA

6.2.1 VALIDACIÓN DEL CONTRATO DE TELEMETRÍA Y CALIDAD DE LOS DATOS INGERIDOS

El contrato de telemetría (TelemetryIn), definido mediante Pydantic v2, se verifica en dos capas independientes: en la capa de aplicación, en el momento de recibir cada punto en el endpoint /ingest, y en la propia base de datos, mediante restricciones CHECK sobre la tabla telemetry_points que reproducen las mismas reglas de consistencia (por ejemplo, que anomaly_label, anomaly_type y anomaly_metadata sean coherentes entre sí). Esta redundancia deliberada, ya descrita en el Capítulo 5 como defensa en profundidad, se ha validado mediante la batería de pruebas test_emitter_contract.py y test_pipeline.py, que comprueban que tanto los puntos válidos como los puntos deliberadamente corruptos (usados para simular fallos de sensor) se comportan según lo esperado en ambas capas.

Durante el desarrollo, esta doble validación ha demostrado su utilidad en la práctica: en varias ocasiones, cambios en el generador sintético que introducían inconsistencias sutiles en los metadatos de anomalía (por ejemplo, al añadir nuevos tipos de escenario) fueron detectados de inmediato por el rechazo de la base de datos, antes de que pudieran contaminar el conjunto de datos utilizado posteriormente para entrenar los modelos.

6.2.2 RENDIMIENTO DE LA INGESTA Y DE LA BASE DE DATOS

El rendimiento de la ingesta se ha evaluado mediante pruebas de carga marcadas como slow en la suite de Pytest (test_pipeline_load.py), que no pretenden ser un perfilado exhaustivo de latencia, sino verificar que el sistema mantiene su corrección funcional bajo tráfico elevado. En concreto, test_high_rate_ingest simula 12 aeronaves emitiendo a una tasa de 40 peticiones por segundo durante 12 segundos, con una probabilidad del 5% de introducir huecos de muestreo y otro 5% de valores atípicos, y comprueba que el número de

filas insertadas se mantiene por encima del mínimo esperado y que la tasa de respuestas HTTP 500 no supera el 5% de las peticiones aceptadas.

Estas pruebas se han ejecutado de forma satisfactoria sobre el stack completo desplegado con Docker Compose (servicio FastAPI, pool de conexiones psycopg_pool y PostgreSQL 15), confirmando que la arquitectura basada en un pool de conexiones reutilizables evita la degradación del servicio bajo ráfagas de tráfico. Queda como trabajo futuro un perfilado más detallado de latencia (percentiles p95/p99) y de rendimiento de escritura en la base de datos a medida que el volumen de datos almacenados crezca, lo que motivaría la migración hacia TimescaleDB descrita en el Capítulo 4.

6.2.3 OBSERVABILIDAD: MÉTRICAS Y PANELES OBTENIDOS EN GRAFANA

El servicio expone métricas Prometheus que cuantifican tanto la actividad de ingesta (ingest_valid_points_total, http_requests_total desglosado por código de estado y ruta) como la actividad del módulo de detección de anomalías en tiempo real (DETECTIONS y ANOMALIES_FLAGGED, introducidos junto con la integración descrita en la sección 6.6). Estas métricas alimentan tres dashboards de Grafana provisionados automáticamente con el despliegue del proyecto.

El dashboard de vuelos (dashboard_flights.json) concentra 17 paneles centrados en una aeronave seleccionada: evolución de altitud, velocidad respecto al suelo, rumbo, velocidad vertical, actitud (pitch/roll/yaw), calidad de señal, confianza del escenario y puntuación de anomalía, junto con paneles específicos del módulo de detección (puntuación del modelo en tiempo real, último veredicto del modelo baseline, estado predicho por el modelo LSTM y su evolución temporal) (Figura 7). El dashboard de estado de simulación (dashboard_simulation_status.json) resume el estado de las ejecuciones del emisor sintético (en curso, completada o fallida) (Figura 8), y el dashboard de tablas (dashboard_tables.json) permite inspeccionar directamente el contenido más reciente de las tablas principales de la base de datos, incluyendo las violaciones de contrato detectadas.

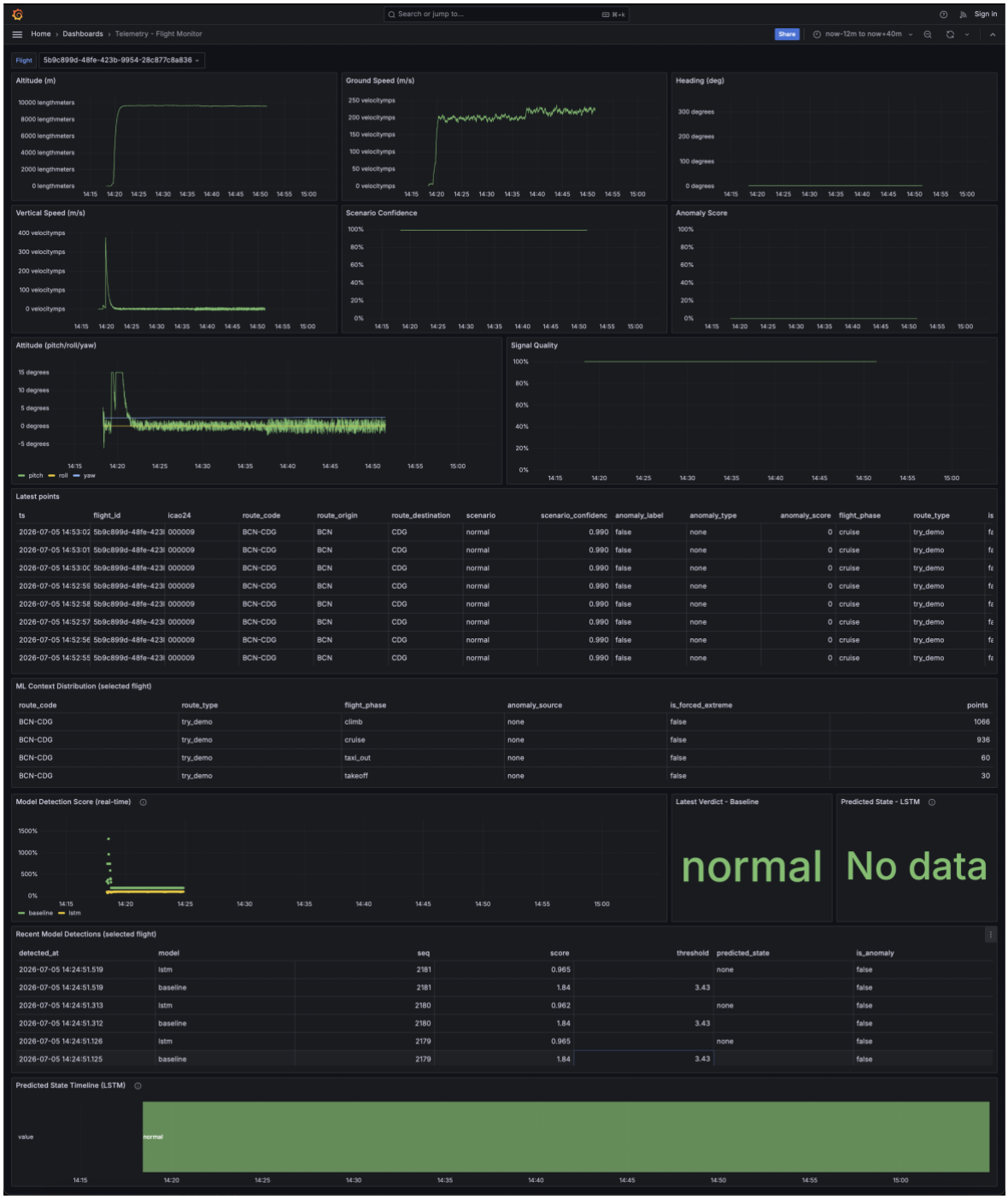

Figura 7. Dashboard flight tracker

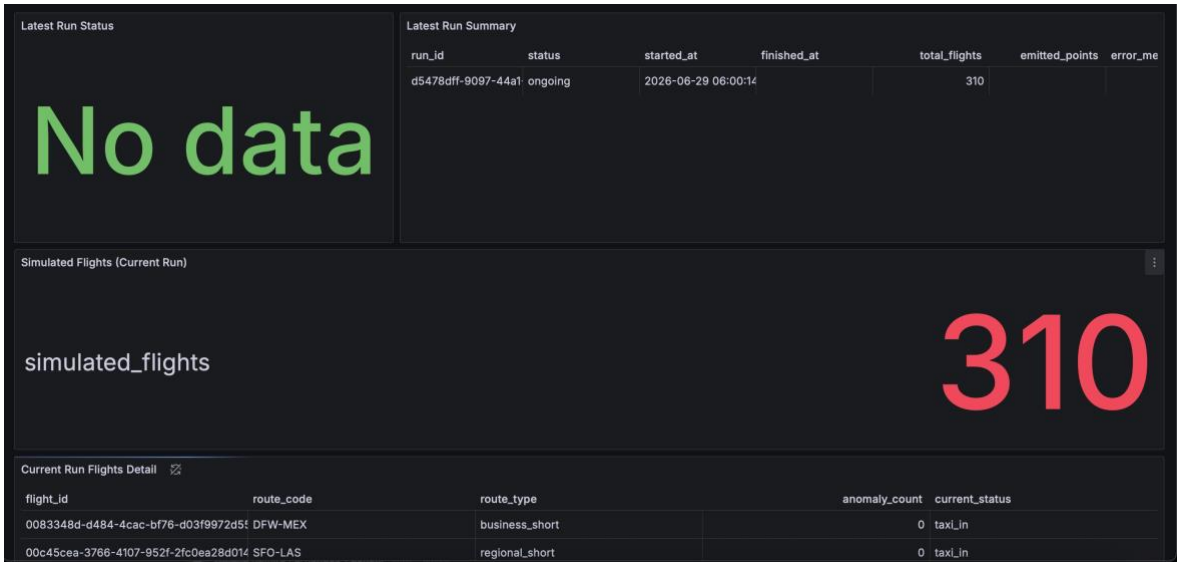

Figura 8. Dashboard simulation status

6.3 RESULTADOS DE LA GENERACIÓN DE DATOS SINTÉTICOS

6.3.1 COBERTURA DE ESCENARIOS Y TIPOS DE ANOMALÍA SIMULADOS

El emisor sintético contempla diez tipos de escenario definidos en scenarios/registry.py, de los cuales ocho son efectivamente alcanzables por el generador de vuelos (normal, engine_failure, gnss_drift, turbulence, wind_shear, sensor_dropout, sensor_spikes y turbulence_spike_dropout); los dos restantes (frozen_sensor e irregular_sampling) están implementados como generadores independientes pero no forman parte de la cadena de pesos ni de encadenamiento de escenarios de build_schedule(), por lo que nunca llegan a activarse en la práctica. Esta limitación, detectada durante la construcción del dataset, se documenta aquí de forma honesta como una cobertura incompleta del catálogo de escenarios definido, y se recoge como trabajo futuro en el Capítulo 7.

Sobre los 300 vuelos finalmente utilizados para entrenar y evaluar los modelos, la cobertura de cada tipo de anomalía (medida en número de vuelos distintos que lo contienen, ya que un mismo vuelo puede combinar varios tipos por el encadenamiento de escenarios) es la siguiente: turbulence (15 vuelos), wind_shear (13), route_deviation (7), gnss_drift (4),

engine_failure (3), y, con una presencia mucho menor, dropout, spike y multi (2 vuelos cada uno). Esta diferencia de un orden de magnitud entre los tipos más frecuentes y los más raros no es casual: se debe a que sensor_dropout y sensor_spikes reciben los pesos de aparición más bajos de todo el generador y, a diferencia de turbulence, wind_shear o engine_failure, no forman parte de ninguna cadena de encadenamiento de escenarios, por lo que solo pueden aparecer mediante un único sorteo de baja probabilidad por vuelo (Figura 9).

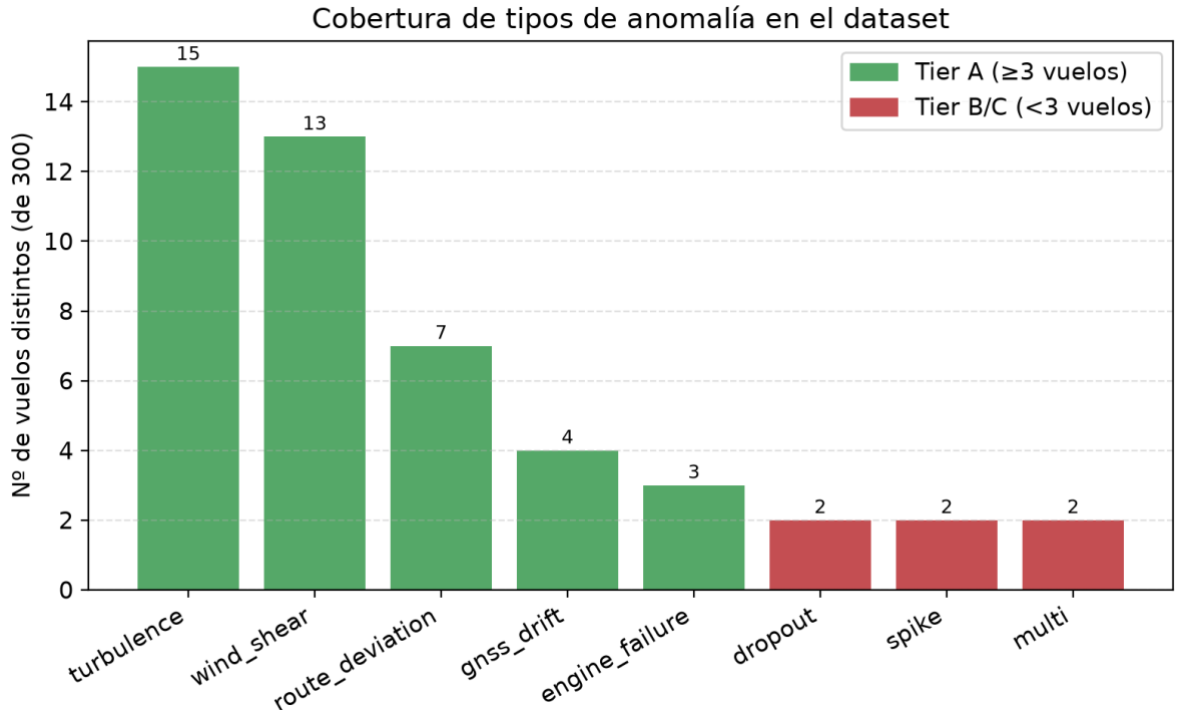

Figura 9. Cobertura de tipos de anomalía en el dataset de entrenamiento

6.3.2 CONSTRUCCIÓN DEL DATASET Y PARTICIÓN TRAIN/VALIDATION/TEST

La partición del dataset se realiza siempre a nivel de vuelo completo, nunca por punto individual, para evitar fuga de información entre los conjuntos de entrenamiento, validación y test (un modelo no debe entrenar y evaluarse sobre puntos del mismo vuelo). Esta restricción, combinada con la baja cobertura de algunos tipos de anomalía descrita en la sección anterior, hace imposible garantizar una partición 70/15/15 uniforme para todos los tipos: un tipo con un único vuelo no puede repartirse simultáneamente entre entrenamiento,

validación y test. Por ello, prepare_split.py clasifica cada tipo de anomalía en niveles (tiers) según su número de vuelos, y aplica una política explícita para cada nivel: los tipos con tres o más vuelos (tier A) garantizan al menos un vuelo en cada uno de los tres conjuntos; los tipos con exactamente dos vuelos (tier B) garantizan un vuelo en entrenamiento y otro en test, renunciando a validación; y los tipos con un único vuelo (tier C) se asignan íntegramente a test, priorizando su visibilidad en la comparativa final de modelos sobre su presencia durante el entrenamiento.

El resultado de esta partición, verificado directamente sobre la tabla ml_training_points, es el siguiente:

Conjunto	Vuelos	Puntos de telemetría
Entrenamiento (train)	207	4.412.320
Validación (val)	45	983.147
Test	48	1.034.292

En total, los 300 vuelos suman 6.429.759 puntos de telemetría repartidos entre los tres conjuntos. Los tipos de anomalía en tier A (turbulence, wind_shear, route_deviation, gnss_drift y engine_failure) cuentan con representación en los tres conjuntos; los de tier B (dropout, spike y multi) solo aparecen en entrenamiento y test, una limitación conocida y documentada que se tiene en cuenta al interpretar los resultados desglosados por tipo de anomalía en la sección 6.4.

6.4 RESULTADOS DE LA COMPARATIVA DE MODELOS DE DETECCIÓN DE ANOMALÍAS

6.4.1 MODELO BASELINE ESTADÍSTICO

El modelo baseline calcula, para cada punto de telemetría, el máximo valor absoluto de z-score entre todas las variables normalizadas del punto (usando media y desviación típica

calculadas exclusivamente sobre el conjunto de entrenamiento), y marca el punto como anómalo si ese máximo supera un umbral calibrado sobre el conjunto de validación. Es, por diseño, el modelo más simple de los cuatro: no requiere entrenamiento propiamente dicho (fit_seconds = 0,00 s, ya que solo se calculan estadísticos descriptivos) y su coste de inferencia es trivial (0,14 s sobre el conjunto de test completo).

Sobre el conjunto de test, con el umbral calibrado en validación (3,425), el modelo obtiene una precisión de 0,527, un recall de 0,421, un F1 de 0,468 y una precisión media (AP) de 0,380. El desglose por tipo de anomalía resulta muy revelador del funcionamiento del modelo: acierta de forma casi perfecta en multi (99,7% de recall) y engine_failure (95,1%), bien en turbulence (83,7%), pero falla casi por completo en gnss_drift (0,1%) y dropout (0,2%), y solo detecta parcialmente route_deviation (26,8%) y wind_shear (14,5%). Este patrón es coherente con su propia definición: al basarse en el valor absoluto de una única variable en un instante concreto, el modelo detecta bien las anomalías que se manifiestan como desviaciones bruscas y de gran magnitud en alguna variable individual (una caída de altitud sostenida, un pico de valor extremo), pero es ciego a las anomalías que consisten en una desviación sutil y mantenida en el tiempo (el pequeño desplazamiento constante de gnss_drift) o que, directamente, no alteran los valores cinemáticos en absoluto (dropout, que solo afecta al espaciado temporal entre mensajes).

6.4.2 ISOLATION FOREST

El segundo candidato es un Isolation Forest de scikit-learn con 200 árboles, entrenado sobre los 4.553.522 puntos de entrenamiento. A diferencia del baseline, este modelo no examina cada variable de forma aislada, sino que considera la posición conjunta del punto en el espacio de características completo, aislando los puntos que requieren menos particiones aleatorias para quedar separados del resto (y que, por tanto, se consideran más anómalos). Su coste de entrenamiento (1,89 s) e inferencia (6,69 s sobre el conjunto de test) es superior al del baseline, pero sigue siendo perfectamente compatible con un uso periódico o por lotes.

Sobre el conjunto de test obtiene una precisión de 0,610 (la más alta de los tres modelos clásicos), un recall de 0,313, un F1 de 0,414 y un AP de 0,397. El desglose por tipo confirma que el modelo sigue detectando bien las anomalías de gran magnitud conjunta (multi: 91,7%, engine_failure: 77,2%, turbulence: 74,1%), pero su rendimiento se desploma en route_deviation (0,09%) y wind_shear (2,6%), muy por debajo incluso del baseline en estos dos tipos. La explicación más plausible es que ambas anomalías se manifiestan como una desviación coherente y mantenida de la trayectoria (un cambio de rumbo sostenido, una racha de viento que empuja la velocidad y el rumbo en una dirección consistente) más que como puntos aislados y extremos en el espacio de características; el Isolation Forest, al tratar cada punto de forma independiente, no está diseñado para premiar la coherencia temporal de una desviación moderada, por lo que estos puntos no resultan especialmente fáciles de aislar frente al resto de la nube de datos normales.

6.4.3 LOCAL OUTLIER FACTOR

El tercer candidato, Local Outlier Factor (LOF) con 35 vecinos, mide la densidad local de cada punto respecto a la de sus vecinos más próximos, lo que en teoría le permite detectar anomalías que son locales a una región del espacio de características aunque no destaquen globalmente. Se trata, sin embargo, del modelo con peores garantías de escalado de los cuatro: al ser un método local que necesita comparar cada punto nuevo con su vecindario en el conjunto de entrenamiento, no resulta viable entrenarlo (ni evaluarlo) sobre los 4.553.522 puntos completos, por lo que se entrena sobre una muestra de 50.000 puntos. Aun así, su coste de inferencia (28,63 s sobre el conjunto de test) es entre 4 y 200 veces superior al de los otros tres modelos, lo que lo descarta de facto para un despliegue en tiempo real punto a punto.

En cuanto a precisión de detección, sobre el conjunto de test obtiene una precisión de 0,236, un recall de 0,435 (el más alto de los tres modelos clásicos) y un F1 de 0,306, el más bajo de los cuatro candidatos. Destaca especialmente en spike (80,5% de recall, el mejor resultado de todo el proyecto para este tipo salvo el LSTM), coherente con su diseño: un

pico puntual y aislado de un valor sobre un entorno estable es, por definición, una anomalía de densidad local. Sin embargo, resulta el peor de los tres modelos clásicos en engine_failure (9,8%), ya que una caída sostenida y progresiva de la altitud no es un punto local aislado, sino un desplazamiento gradual de toda una región de la trayectoria, algo que LOF, centrado en la densidad de vecindarios pequeños, no está bien equipado para capturar. Cabe señalar que el F1 de este modelo mejoró de forma notable (de 0,175 a 0,306) tras corregir un error identificado durante el propio desarrollo del proyecto en el cálculo de delta_t_s (la variable de intervalo temporal entre puntos consecutivos), que inicialmente se rellenaba con 0,0 en el primer punto de cada vuelo en lugar de con la mediana de entrenamiento, generando un valor de z-score extremo y espurio (≈ -146) en cada inicio de vuelo que distorsionaba significativamente los modelos basados en densidad o distancia.

6.4.4 MODELO LSTM

El cuarto candidato es una red LSTM de una capa (hidden_size = 64) entrenada sobre ventanas deslizantes de 50 puntos (stride 25), con un total de 181.827 ventanas de entrenamiento, ejecutada sobre GPU de Apple Silicon (device = mps). El entrenamiento (8 épocas) tarda 35,49 s, sensiblemente más que los otros tres modelos, pero la inferencia sobre el conjunto de test completo (41.733 muestras) se resuelve en 1,06 s, es decir, del orden de 25 microsegundos por punto, un coste perfectamente compatible con el procesamiento en tiempo real descrito en la sección 6.6.

Para poder comparar este modelo con los otros tres candidatos en igualdad de condiciones dentro del mismo arnés de evaluación binario, se deriva una vista binaria a partir de la probabilidad máxima entre todas las clases distintas de 'ninguna anomalía', calibrando un umbral (0,9899) sobre el conjunto de validación. Con este umbral, el modelo obtiene sobre test una precisión de 0,989 (la más alta de los cuatro modelos, con diferencia) y un AP de 0,956; el recall (0,695) es inferior al obtenido en validación (0,884), una brecha de generalización que resulta esperable dado que varios tipos de anomalía cuentan con muy pocos vuelos de test (por ejemplo, gnss_drift o multi).

El desglose por tipo en esta vista binaria muestra un recall muy alto en wind_shear (97,0%), route_deviation (86,2%), engine_failure (85,7%) y turbulence (78,0%), moderado en multi (64,4%) y spike (45,8%), y nulo en dropout (0%) y prácticamente nulo en gnss_drift (3,8%). Los dos primeros casos no son un fallo del modelo, sino una consecuencia directa y esperada de la política de partición descrita en la sección 6.3: dropout y spike son precisamente los dos tipos que, por su escasa cobertura (dos vuelos cada uno), quedan excluidos por completo del conjunto de entrenamiento (missing_in_train), por lo que el modelo nunca ha visto un solo ejemplo de su firma durante el aprendizaje. El caso de gnss_drift es más sutil: como se detalla en la sección 6.5, el modelo sí identifica correctamente esta clase en la vista multiclase (recall del 98,7% al usar directamente el argmax de las 9 clases), pero rara vez alcanza el umbral de confianza tan exigente (0,99) que se calibra para la vista binaria, ya que su firma es sutil y de baja magnitud. Esta discrepancia es precisamente la razón por la que el sistema desplegado en producción (service/detector.py) no utiliza en ningún momento un umbral binario para el LSTM, sino el argmax directo sobre las 9 clases, evitando así descartar detecciones correctas por un criterio de confianza pensado únicamente para la comparación homogénea entre los cuatro modelos.

6.4.5 COMPARATIVA GLOBAL Y JUSTIFICACIÓN DEL MODELO SELECCIONADO

La siguiente tabla resume los resultados de los cuatro modelos sobre el conjunto de test, con el umbral de cada uno calibrado sobre el conjunto de validación:

Modelo	Precisión	Recall	F1	AP	Entrenamiento	Inferencia
Baseline estadístico	0,527	0,421	0,468	0,380	0,00	0,14
Isolation Forest	0,610	0,313	0,414	0,397	1,89	6,69
Local Outlier Factor	0,236	0,435	0,306	0,302	0,79	28,63
LSTM (vista binaria)	0,989	0,695	0,816	0,956	35,49	1,06

El propio arnés de comparación (code/ml/compare.py) señala automáticamente al LSTM como mejor modelo por F1 (best_by_f1). Más allá del F1, el LSTM domina claramente al resto en precisión y en AP, y su coste de inferencia, aunque no es el más bajo del proyecto (el baseline es más barato en términos absolutos), resulta órdenes de magnitud inferior al de LOF y perfectamente asumible para el procesamiento en tiempo real. Pero el argumento decisivo para seleccionarlo no es únicamente su rendimiento cuantitativo: es el único de los cuatro modelos capaz de producir de forma nativa un estado concreto (mediante su salida softmax sobre 9 clases) en lugar de una simple puntuación de anomalía binaria. Este es exactamente el requisito funcional definido en el Capítulo 2, no solo detectar que un vuelo se comporta de forma anómala, sino identificar en qué estado concreto se encuentra, por lo que los otros tres modelos, aun siendo útiles como referencia de comparación offline, quedan descartados para el despliegue en producción.

6.5 RESULTADOS DE LA EVOLUCIÓN HACIA LA CLASIFICACIÓN MULTICLASE

6.5.1 DE LA DETECCIÓN BINARIA A LA IDENTIFICACIÓN DEL ESTADO CONCRETO

La primera versión del modelo LSTM, descrita en el Capítulo 5, era un clasificador binario: una capa LSTM seguida de una cabeza lineal con salida sigmoide, entrenada con entropía cruzada binaria para distinguir únicamente entre 'normal' y 'anómalo'. Este diseño cumplía el objetivo de detección, pero no el de identificación: el propio planteamiento del proyecto (Capítulo 2) exige que el sistema sea capaz de indicar en qué estado concreto se encuentra una aeronave, no solo si su comportamiento es o no anómalo, ya que la respuesta operativa ante una turbulencia, una desviación de ruta o un fallo de motor es radicalmente distinta.

Por ello, el modelo se reformuló como un clasificador multiclase: se sustituyó la cabeza lineal de una única salida por una cabeza lineal de 9 salidas (una por cada clase: none, dropout, engine_failure, gnss_drift, multi, route_deviation, spike, turbulence y wind_shear), y la función de pérdida pasó a ser una entropía cruzada categórica ponderada por clase, necesaria para compensar el fuerte desequilibrio del dataset (la clase none representa por sí sola en torno al 96% de los puntos de test). El estado predicho se obtiene directamente como el argmax de las 9 probabilidades de salida, sin necesidad de calibrar ningún umbral adicional; este es el valor que se expone como predicted_state tanto en el módulo de detección en tiempo real (sección 6.6) como en el campo STATUS del panel de visualización (sección 6.7).

6.5.2 RENDIMIENTO POR TIPO DE ANOMALÍA

Evaluado como clasificador multiclase puro (sin ningún umbral binario intermedio) sobre el conjunto de test, el modelo alcanza una exactitud (accuracy) global del 94,79%. Sin

embargo, esta cifra agregada oculta diferencias muy relevantes entre clases, como muestra la media macro de F1 (0,372, calculada dando el mismo peso a cada clase) frente a la media ponderada por soporte (0,956, dominada por la enorme clase none). El informe de clasificación completo por tipo es el siguiente:

Estado	Precisión	Recall	F1	Nº puntos (test)
none	0,999	0,959	0,979	39.994
dropout	0,000	0,000	0,000	59
engine_failure	0,160	0,929	0,274	14
gnss_drift	0,192	0,987	0,322	158
multi	0,214	0,043	0,072	208
route_deviation	0,213	0,948	0,348	174
spike	0,000	0,000	0,000	203
turbulence	0,384	0,817	0,522	431
wind_shear	0,713	0,994	0,830	492

La matriz de confusión completa permite entender el origen de estas cifras. Las filas representan la clase real y las columnas la clase predicha; se omiten por claridad las columnas con recuento nulo en todas las filas:

Real \ Predicho	none	engine_f.	gnss_drift	multi	route_dev.	spike	turbulence	wind_shear
none	38.374	68	548	3	570	0	359	72
dropout	1	0	58	0	0	0	0	0
engine_failure	0	13	1	0	0	0	0	0
gnss_drift	0	0	156	0	1	0	1	0
multi	2	0	0	9	0	0	195	2
route_deviation	3	0	5	0	165	0	1	0
spike	1	0	43	20	27	0	9	103
turbulence	37	0	0	10	12	0	352	20
wind_shear	3	0	0	0	0	0	0	489

Tres patrones destacan sobre el resto. Primero, wind_shear es, con diferencia, la clase minoritaria mejor aprendida (494 de 492 ejemplos correctamente identificados salvo 3 confundidos con none), gracias a contar con 13 vuelos de entrenamiento y a que su firma cinemática (variaciones bruscas y simultáneas de velocidad y rumbo) es suficientemente distintiva. Segundo, engine_failure, gnss_drift y route_deviation presentan un patrón de recall muy alto pero precisión muy baja: el modelo casi nunca deja pasar un caso real (recall superior al 92% en los tres), pero también tiende a usar estas tres etiquetas en exceso para puntos que en realidad son normales (548 puntos none confundidos con gnss_drift y 570 con route_deviation), un comportamiento conservador que resulta preferible en un sistema de alerta, prefiere avisar de más antes que dejar pasar una anomalía crítica, aunque penaliza la precisión agregada. Tercero, multi es la clase peor aprendida a pesar de contar con dos vuelos de entrenamiento: 195 de sus 208 puntos de test se confunden con turbulence, ya que turbulence_spike_dropout comparte con turbulence el mismo ruido gaussiano de base, y los componentes de pico y corte que la distinguen son comparativamente escasos dentro de la

ventana de 50 puntos que ve el modelo. Finalmente, dropout y spike no obtienen ningún acierto, consecuencia directa y esperada de su ausencia total en el conjunto de entrenamiento, ya documentada en la sección 6.3 y en el desglose binario de la sección 6.4.4 (Figura 10).

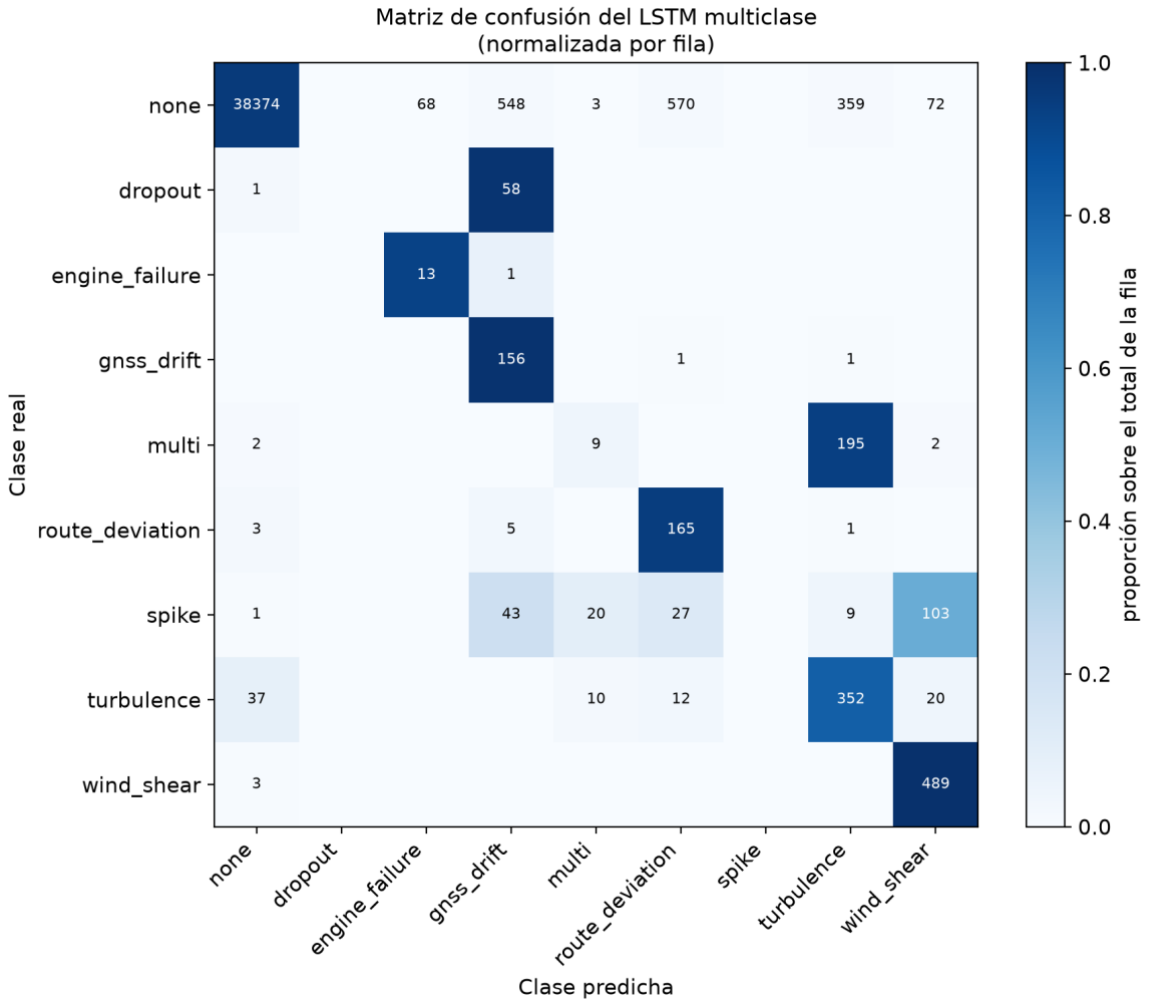

Figura 10. Matriz de confusión del modelo LSTM multiclase

6.6 RESULTADOS DE LA INTEGRACIÓN EN TIEMPO REAL

6.6.1 PRECISIÓN DE LA DETECCIÓN EN PRODUCCIÓN FRENTE A LOS RESULTADOS OFFLINE

Los resultados de la sección 6.5 se obtienen evaluando el modelo sobre ventanas extraídas del propio conjunto de test, con la misma distribución que el conjunto de entrenamiento. Nada garantiza que ese rendimiento se mantenga cuando el modelo se integra en el flujo de ingesta real (service/detector.py) y recibe datos punto a punto según van llegando, en lugar de ventanas ya construidas y normalizadas de forma homogénea. De hecho, la validación de la integración en tiempo real sobre el escenario de demostración de 10 vuelos (descrito en la sección 6.7.3) puso de manifiesto tres discrepancias serias entre el comportamiento offline y el comportamiento real en producción, cada una de ellas achacable a una diferencia concreta entre cómo se generan los datos en directo y las asunciones estadísticas implícitas en el entrenamiento.

La primera discrepancia, ya mencionada en la sección 6.4.3, afectaba a todos los modelos por igual: el primer punto de cada vuelo rellenaba la variable delta_t_s (intervalo entre mensajes consecutivos) con 0,0 en lugar de con la mediana de entrenamiento, generando un z-score espurio de aproximadamente -146 en el arranque de cada vuelo. Se corrigió de forma simétrica en ml/features.py (entrenamiento) y en service/detector.py (inferencia en producción), dejando el primer delta como valor ausente e imputándolo con la mediana de entrenamiento, tal y como se hace con el resto de valores ausentes.

La segunda discrepancia era específica del generador sintético usado para la demostración final (synthetic/flight_emitter_try.py): su primera versión comprimía cada vuelo completo en apenas 480-520 segundos simulados para que la demostración durase pocos minutos. Sin embargo, el modelo se entrenó sobre vuelos reales cuya fase de crucero dura, como mínimo, 600 segundos simulados y habitualmente varios miles, tiempo suficiente para que la aeronave alcance un estado verdaderamente estable (deltas cercanos a

cero) tras un breve transitorio inicial de en torno a 100 pasos. Al comprimir el vuelo, el crucero apenas superaba la duración de ese transitorio, por lo que la aeronave nunca llegaba a estabilizarse del todo, y sus deltas, aun en ausencia de cualquier anomalía inyectada, se parecían estadísticamente a los de una anomalía real. El resultado práctico era una alarma en falso en torno al 28% de las veces sobre el propio vuelo de control 'siempre normal' de la demostración. La solución no consistió en suavizar la señal ni en ocultar el problema, sino en reproducir exactamente las mismas fórmulas de duración de fase que usa el generador real de entrenamiento (climb_sec y descent_sec en función de la altitud, cruise_sec en función de la distancia real entre aeropuertos), dejando que el tiempo de captura en pantalla se ajustase de forma independiente mediante el parámetro, rate del emisor. Tras el ajuste, la precisión medida sobre datos en directo alcanzó el 100% durante el ascenso y el 73,7% durante el crucero para los puntos realmente normales, frente al comportamiento prácticamente aleatorio observado antes de la corrección.

La tercera discrepancia afectaba específicamente a la detección en directo de engine_failure. La física simulada durante la maniobra de emergencia en la demostración hacía descender la aeronave a un ritmo de en torno a 388 m por paso, mientras que el generador real usado para entrenar el modelo (scenarios/anomalies.py) simula un descenso mucho más suave y constante, de entre 15 y 40 m por paso, entre 10 y 25 veces más lento. Esta diferencia empujaba los valores de delta_alt_m muy por fuera de cualquier rango que el modelo hubiera visto durante el entrenamiento para esta clase, y el resultado era que, durante la caída real de la aeronave, el modelo predecía engine_failure en apenas un 3,2% de los puntos (mayoritariamente predecía none o turbulence). Al sustituir la física de la maniobra de emergencia por la misma fórmula de descenso que usa el generador de entrenamiento, la precisión de detección durante la caída pasó del 3,2% al 94,8%, confirmando que el problema no era una limitación del modelo, sino una diferencia de distribución entre los datos de la demostración y los datos de entrenamiento.

6.6.2 CASOS LÍMITE IDENTIFICADOS Y AJUSTES APLICADOS

No todas las discrepancias observadas entre el comportamiento offline y el comportamiento en producción resultaron corregibles ajustando el generador de datos. En concreto, se identificó un sesgo sistemático y persistente del modelo durante las fases naturales de descenso, aproximación, aterrizaje y rodaje de entrada (taxi_in): en estas fases, un vuelo genuinamente normal (anomaly_type = none) es identificado por el modelo como wind_shear en el 100% de los puntos observados durante la validación en directo, independientemente del vuelo o de la ruta. La hipótesis más plausible, coherente con la matriz de confusión de la sección 6.5.2 (donde 570 puntos none se confunden con route_deviation y 359 con turbulence en el propio conjunto de test offline), es que la pérdida de altitud real durante un descenso normal produce deltas de altitud y velocidad de magnitud similar a los que provoca la perturbación de wind_shear inyectada por el generador, resultando estadísticamente difíciles de distinguir para el modelo.

Dado que este comportamiento no depende de ningún parámetro del emisor sintético y se reproduce de forma consistente en todos los vuelos, se ha optado por no enmascararlo artificialmente, sino por gestionarlo de forma honesta en la capa de presentación: el panel de visualización (sección 6.7) solo confía en predicted_state durante las fases en las que se ha comprobado empíricamente que el modelo es fiable (ascenso, crucero y, tras la corrección descrita en 6.6.1, la maniobra de emergencia de engine_failure), y muestra un indicador explícito de 'monitorización pausada' durante el resto de fases, en lugar de una etiqueta de estado que se sabe de antemano que probablemente sea incorrecta. En todos los casos, el campo de verdad de terreno (scenario) permanece siempre visible en el panel, con independencia de la fiabilidad de la predicción del modelo en ese instante.

6.7 RESULTADOS DE LA VISUALIZACIÓN Y VALIDACIÓN FUNCIONAL DEL SISTEMA COMPLETO

6.7.1 PANEL DE MONITORIZACIÓN Y MAPA EN TIEMPO REAL

El panel de visualización, desarrollado en React, consulta cada 2 segundos el endpoint público GET /flights/active del servicio, que devuelve la última posición y el último veredicto de los modelos de detección para cada vuelo activo en los últimos 15 segundos. Cada aeronave se representa mediante una tarjeta con su indicador STATUS (el estado predicho por el LSTM, coloreado según su severidad), el nivel de confianza de la predicción, sus valores cinemáticos actuales (altitud, velocidad, rumbo, posición) y, en el pie de la tarjeta, el campo scenario de verdad de terreno generado por el simulador, que permanece siempre visible con independencia del estado predicho. Las tarjetas se ordenan de forma estable por identificador de vuelo, no por su estado de alarma, precisamente para evitar que cambien de posición en la pantalla cada vez que un vuelo entra o sale de un estado anómalo, lo que resultaba visualmente confuso en las primeras versiones del panel.

El mapa, construido con Leaflet y react-leaflet, sitúa cada aeronave en sus coordenadas reales, con un icono orientado según su rumbo actual y coloreado según el mismo esquema que las tarjetas, ajustando automáticamente el encuadre a la primera carga de datos para mostrar el conjunto completo de vuelos activos sin necesidad de interacción manual (Figura 11).

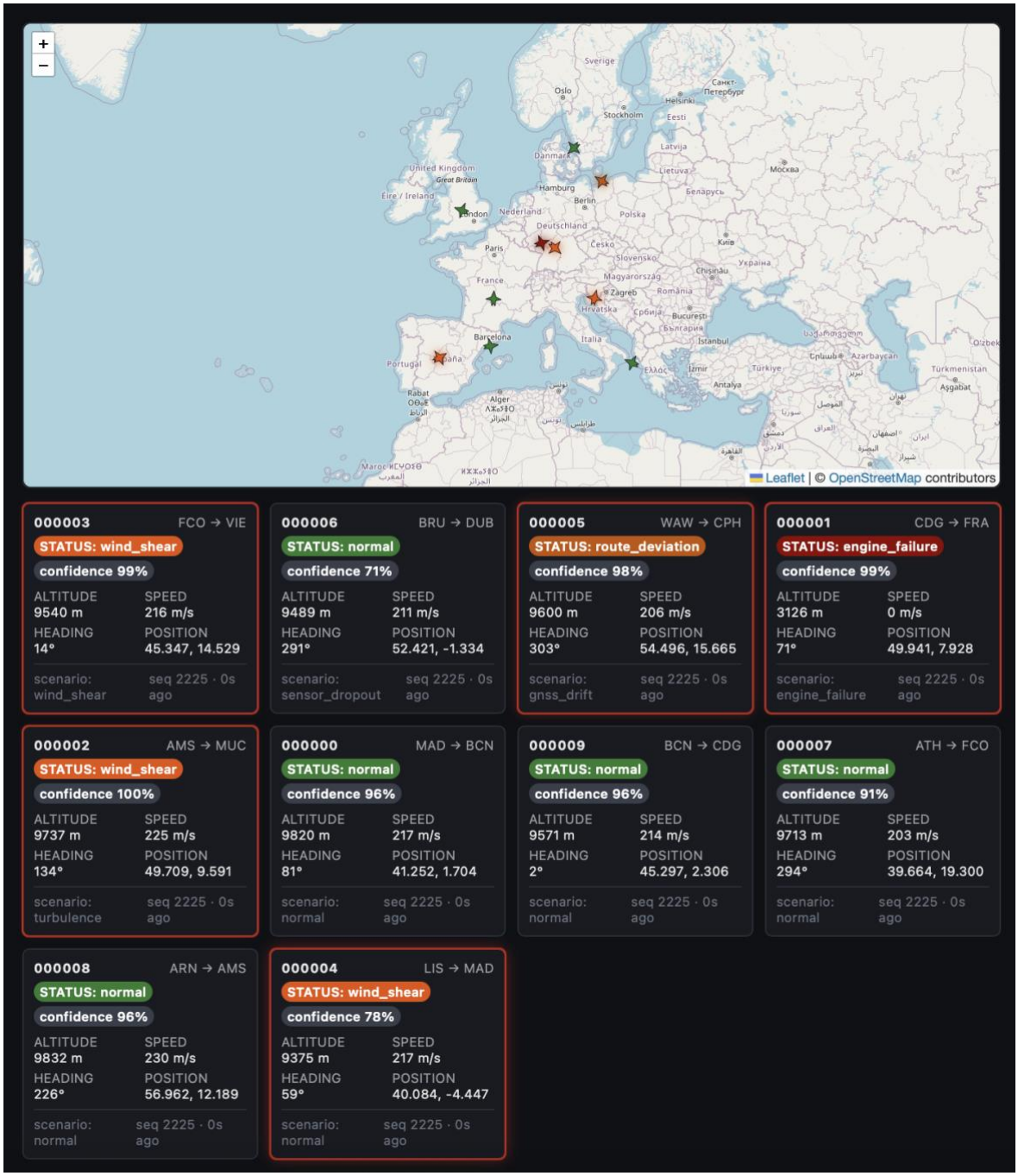

Figura 11. Panel de vuelos y mapa en tiempo real durante la demostración

6.7.2 VALIDACIÓN DE LA ALARMA ANTE ESTADOS ANÓMALOS

El sistema de alarma se activa cuando al menos un vuelo presenta un predicted_state distinto de 'none' durante una fase en la que la predicción se considera fiable (sección 6.6.2), mostrando un banner rojo pulsante en la parte superior del panel con el listado de aeronaves afectadas y su estado, y resaltando con un borde rojo la tarjeta y el icono en el mapa de cada vuelo concernido. De forma adicional, y de manera opcional para el usuario, un aviso sonoro (implementado mediante la Web Audio API) se dispara únicamente cuando un vuelo entra de nuevo en un estado anómalo, no en cada sondeo mientras el vuelo permanece en dicho estado, evitando un sonido repetitivo mientras la situación persiste (Figura 12).

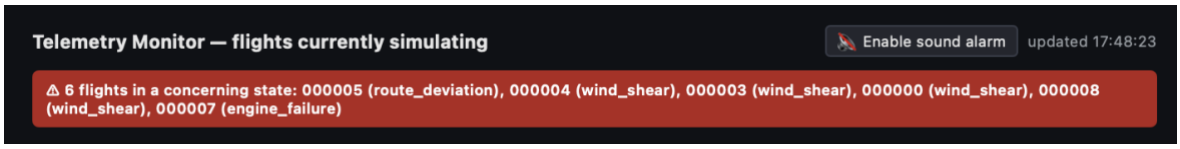

Figura 12. Banner de alarma activo con varias aeronaves en estado anómalo

6.7.3 DEMOSTRACIÓN END-TO-END DEL SISTEMA COMPLETO

La validación funcional definitiva del sistema completo se realiza mediante un escenario de demostración determinista de diez vuelos simultáneos (synthetic/flight_emitter_try.py), diseñado específicamente para mostrar de forma clara y reproducible el funcionamiento de principio a fin de la plataforma: diez rutas reales dentro de la Unión Europea (con coordenadas reales de quince aeropuertos), semilla aleatoria fija (seed = 42) para que la ejecución sea exactamente reproducible, y una asignación explícita de escenario por vuelo que garantiza la aparición de los ocho tipos de anomalía alcanzables por el generador (turbulence, wind_shear, route_deviation, gnss_drift, sensor_dropout, sensor_spikes y turbulence_spike_dropout), además de un vuelo que permanece siempre en estado normal como control y un vuelo que sufre un fallo de motor (engine_failure) que desemboca siempre en un aterrizaje forzoso.

Verificado directamente contra la base de datos, cada uno de los ocho tipos de anomalía se activa correctamente en su vuelo correspondiente durante la fase de crucero (por

ejemplo, gnss_drift se activa en el vuelo WAW→CPH entre los pasos 145 y 360 de su trayectoria simulada), y el vuelo con engine_failure (CDG→FRA) completa su descenso de emergencia hasta una altitud de 0 metros, confirmando que la simulación de aterrizaje forzoso finaliza correctamente. Con las correcciones descritas en la sección 6.6.1 ya aplicadas, la detección en tiempo real reproduce sobre este escenario las cifras de fiabilidad ya mencionadas: 100% de aciertos durante el ascenso, 73,7% durante el crucero y 94,8% durante la maniobra de emergencia, con el panel de visualización mostrando en todo momento el estado STATUS: engine_failure durante la caída real de la aeronave (Figura 13).

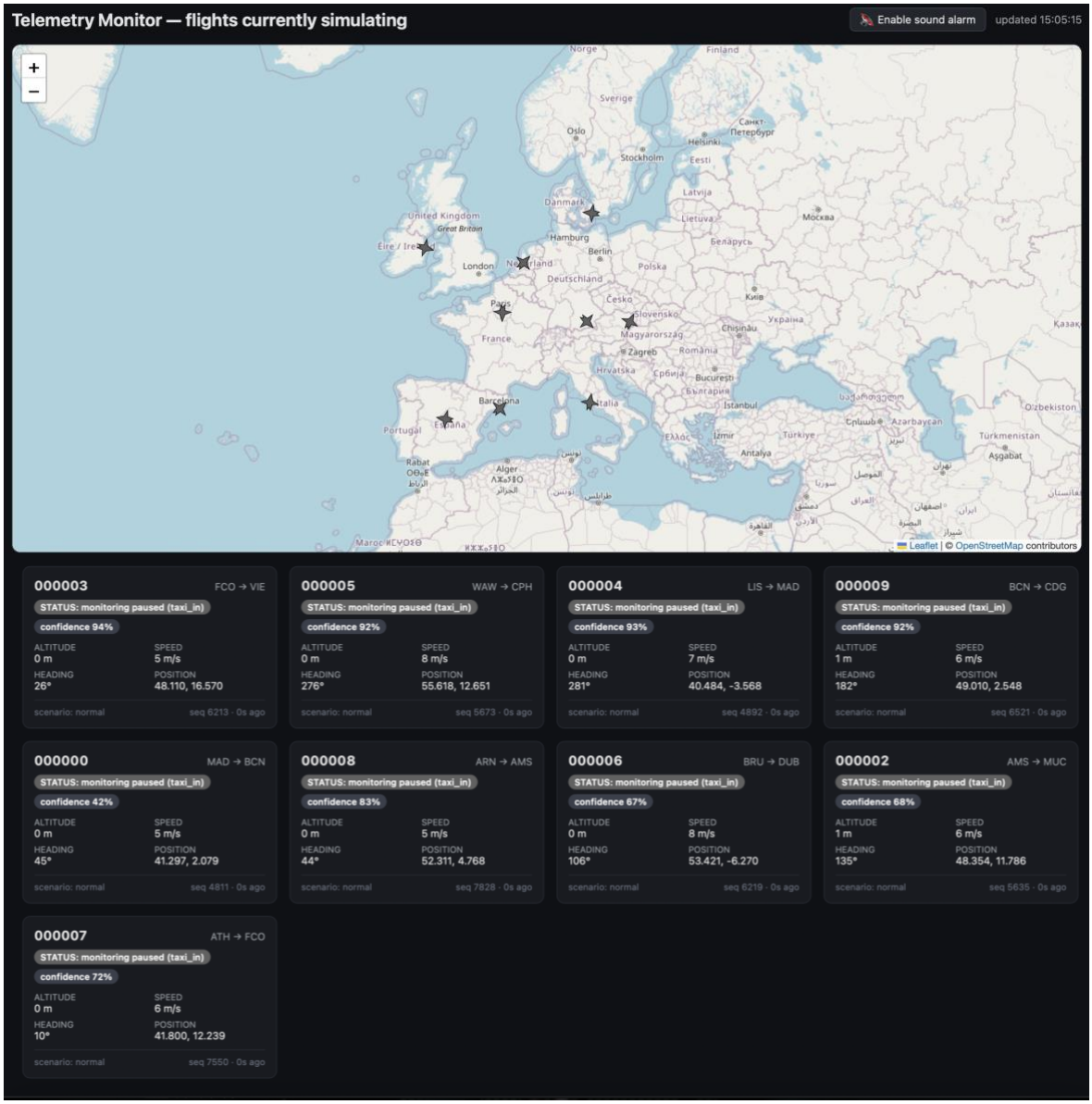

Figura 13. Mapa final de la demostración con los 10 vuelos

6.8 SÍNTESIS DE RESULTADOS

Los resultados presentados en este capítulo muestran una progresión coherente con el desarrollo real del proyecto: una infraestructura de ingesta y persistencia validada funcionalmente y bajo carga (6.2), un dataset sintético construido con una cobertura de

anomalías conocida y documentada, incluidas sus limitaciones (6.3), una comparativa rigurosa de cuatro modelos de detección de anomalías de la que el LSTM emerge como la única alternativa capaz de identificar el estado concreto de una aeronave con una precisión y una latencia compatibles con el tiempo real (6.4), su evolución de clasificador binario a multiclase (6.5), y la validación de su comportamiento en producción, incluyendo tres discrepancias reales entre el entorno offline y el entorno en directo que fueron identificadas y corregidas de forma explícita (6.6).

El resultado final del proyecto, validado de forma conjunta en la demostración end-to-end de la sección 6.7.3, es un sistema que ingiere telemetría en tiempo real, la valida y almacena de forma consistente, y es capaz de identificar en directo, con una precisión medida y no simplemente asumida, en qué estado concreto se encuentra una aeronave, mostrándolo de forma inmediata y comprensible en un panel de visualización con alarma. Las limitaciones que persisten, la falta de cobertura de dropout y spike en el conjunto de entrenamiento, y la confusión sistemática entre descensos normales y wind_shear, están identificadas, cuantificadas y documentadas de forma transparente, y se retoman como líneas de trabajo futuro concretas en el Capítulo 7 (en particular, el reequilibrado de los pesos de escenario en el emisor sintético para aumentar la cobertura de los tipos minoritarios, y la incorporación de características adicionales que permitan distinguir un descenso planificado de una perturbación real de viento).

CAPÍTULO 7 – CONCLUSIONES Y TRABAJOS FUTUROS

7.1 COMPLICACIONES DURANTE EL DESARROLLO

El desarrollo del proyecto no fue una progresión lineal desde el diseño hasta la validación final descrita en el Capítulo 6: varias decisiones de diseño solo se revelaron necesarias tras encontrarse de frente con un problema concreto durante la implementación o la simulación. Esta sección recoge las complicaciones más relevantes, agrupadas en tres frentes: la generación masiva de datos sintéticos, el análisis y la integración del modelo de detección en tiempo real, y las limitaciones de cobertura de determinados escenarios.

7.1.1 GENERACIÓN MASIVA DE DATOS SINTÉTICOS

La primera complicación seria surgió al intentar generar de una sola vez el conjunto completo de vuelos sintéticos necesario para entrenar los modelos de detección de anomalías. Una ejecución continua y prolongada del emisor (flight_emitter.py) generando decenas de vuelos en paralelo resultó, en la práctica, mucho más frágil de lo previsto: por un lado, el propio motor de Docker Desktop fue agotando progresivamente el espacio en disco asignado a medida que se acumulaban capas de imágenes, volúmenes de la base de datos y

logs de contenedores, hasta el punto de que, en más de una ocasión, la simulación se detuvo abruptamente a mitad de una tanda de vuelos, dejando datos parciales e inconsistentes que hubo que descartar por completo. Por otro lado, se detectó un error concreto en la física del emisor por el cual, bajo determinadas combinaciones de rumbo y velocidad, la latitud de una aeronave podía divergir sin control cerca de los polos, provocando que el proceso quedase generando puntos sin sentido físico de forma indefinida en lugar de terminar el vuelo con normalidad. Ante estos dos problemas, generar el dataset completo en una única ejecución masiva se reveló inviable: hubo que dividir la generación en tandas más pequeñas (de un número acotado de vuelos cada una), validar cada tanda de forma independiente antes de incorporarla al conjunto de datos compartido, y purgar el espacio en disco de Docker entre tandas para evitar repetir el fallo.

7.1.2 ANÁLISIS DEL MODELO DE DETECCIÓN EN TIEMPO REAL

La segunda complicación relevante fue el propio análisis del modelo de detección en tiempo real, que resultó mucho más costoso de depurar que la evaluación offline descrita en el Capítulo 6. A diferencia de una prueba unitaria, cuyo resultado se conoce de inmediato, un fallo en el comportamiento del modelo en producción solo se manifiesta a medida que un vuelo avanza por sus distintas fases, lo que exige observar el sistema en directo durante varios minutos por cada hipótesis a descartar, en lugar de ejecutar una prueba y leer su resultado. Este proceso de observación prolongada fue el que permitió aislar los tres problemas descritos en la sección 6.6.1 (el relleno erróneo del primer intervalo temporal de cada vuelo, la compresión excesiva de la duración simulada de los vuelos de demostración, y el desajuste de la velocidad de descenso durante la maniobra de emergencia), ninguno de los cuales era evidente a partir de las métricas offline del Capítulo 6, y todos los cuales requirieron reproducir el problema en directo, formular una hipótesis sobre su causa, corregirla y esperar de nuevo a que la simulación avanzase lo suficiente para confirmar o descartar la corrección.

7.1.3 COBERTURA DE ESCENARIOS EN EL GENERADOR

La tercera complicación, ya adelantada en la sección 6.3, tiene su origen en el propio diseño del generador de escenarios: los pesos de aparición asignados a sensor_dropout y sensor_spikes son deliberadamente bajos frente al resto de escenarios, y ninguno de los dos forma parte de las cadenas de encadenamiento de escenarios (por ejemplo, turbulence → wind_shear → route_deviation → engine_failure) que sí enlazan al resto de anomalías entre sí. El resultado práctico es que, del conjunto de vuelos generado, apenas dos vuelos distintos llegaron a contener cada uno de estos dos tipos, frente a los trece o quince vuelos disponibles para wind_shear o turbulence. Esta limitación no se detectó durante el diseño del generador, sino más adelante, al construir la partición train/validation/test (Capítulo 6): con solo dos vuelos disponibles para un tipo de anomalía, resulta matemáticamente imposible repartirlo entre los tres conjuntos a la vez, lo que obligó a diseñar la política de niveles (tiers) descrita en la sección 6.3.2 como solución de compromiso, en lugar de resolver el problema de raíz aumentando la cobertura real de estos escenarios en el propio generador.

7.2 TRABAJOS FUTUROS

De entre las líneas de trabajo futuro identificadas a lo largo del proyecto, dos se consideran prioritarias por su impacto directo en la calidad del modelo de detección de anomalías, y ambas quedan fuera del alcance del presente trabajo por restricciones prácticas de tiempo más que por limitaciones técnicas de fondo.

7.2.1 CONEXIÓN CON UNA FUENTE DE DATOS DE VUELO REALES

La primera es la conexión del sistema a una fuente de datos de vuelo reales, sustituyendo o complementando el emisor sintético que alimenta actualmente toda la plataforma. Como se describe en el Capítulo 3, existen plataformas ya consolidadas que ofrecen acceso a datos ADS-B reales y a gran escala, como OpenSky Network [36], FlightAware [47], Flightradar24 [46] o ADS-B Exchange [48]. La integración con cualquiera de estas fuentes permitiría validar el contrato de telemetría del proyecto y, sobre

todo, entrenar y evaluar los modelos de detección de anomalías sobre tráfico aéreo genuino en lugar de sobre trayectorias generadas artificialmente, cerrando la brecha entre las condiciones controladas de la simulación y la variabilidad real del tráfico aéreo. No se ha abordado en el presente trabajo por el alcance y la complejidad adicional que supondría gestionar el volumen, la autenticación y las limitaciones de tasa de estas APIs dentro del tiempo disponible para el proyecto, pero la arquitectura del sistema (en particular, el contrato TelemetryIn desacoplado del origen de los datos) se diseñó precisamente para no requerir cambios estructurales el día que esta integración se aborde.

7.2.2 REENTRENAMIENTO CON UN CONJUNTO DE DATOS SINTÉTICOS MAYOR

La segunda línea prioritaria es el reentrenamiento del modelo de detección sobre un conjunto de datos sintéticos sensiblemente mayor, y con una cobertura de escenarios más equilibrada que la descrita en la sección 6.3. Como se ha visto en el Capítulo 6, tipos de anomalía como dropout o spike obtienen un recall nulo simplemente por no estar presentes en el conjunto de entrenamiento, una limitación que no se debe al modelo en sí, sino al número de vuelos disponibles para esos tipos. Ampliar de forma sustancial el número de vuelos simulados, junto con un reequilibrado de los pesos de aparición y de las cadenas de encadenamiento de escenarios en el generador (para que sensor_dropout y sensor_spikes dejen de depender de un único sorteo de baja probabilidad por vuelo), permitiría entrenar de nuevo los cuatro modelos sobre una base de datos con cobertura suficiente en todas las clases. Esta tarea no se ha realizado en el presente trabajo por una cuestión estrictamente práctica de tiempo de simulación: generar, validar e ingerir un conjunto de vuelos varias veces mayor que el actual (del orden de miles de vuelos en lugar de trescientos) habría requerido varios días de ejecución continuada del emisor sintético, incompatibles con los plazos del proyecto, más aún teniendo en cuenta las dificultades de estabilidad de la simulación masiva descritas en la sección 7.1.

7.2.3 MEJORAS EN EL GENERADOR DE ESCENARIOS Y EL MODELO

Se identifican también otras vías de mejora de menor alcance individual relacionadas directamente con el generador de escenarios y con el propio modelo de detección: extender la cobertura del generador a los dos tipos de anomalía actualmente inalcanzables (frozen_sensor e irregular_sampling); incorporar un mecanismo de suavizado temporal (por ejemplo, una decisión por mayoría sobre las últimas predicciones) tanto en el módulo de detección en tiempo real como en el panel de visualización, para reducir el parpadeo entre estados observado en fases de transición; y explorar arquitecturas alternativas a la LSTM para la clasificación de secuencias, como modelos basados en mecanismos de atención, que podrían mejorar la precisión sobre las clases minoritarias sin necesidad de aumentar el tamaño de la ventana temporal.

7.2.4 MEJORAS EN LA INFRAESTRUCTURA Y LA OBSERVABILIDAD

En el plano de la infraestructura, quedan pendientes completar la migración de telemetry_points hacia TimescaleDB conforme crezca el volumen de datos almacenados, tal y como ya se prevé en el Capítulo 4, y realizar un perfilado formal de latencia (percentiles p95/p99) de la ingesta bajo distintos niveles de carga sostenida, más allá de las pruebas de corrección funcional descritas en el Capítulo 6.

7.2.5 MEJORAS EN LA INTERFAZ Y LOS CANALES DE ALERTA

Por último, se propone ampliar los canales de alerta del sistema más allá del panel web (por ejemplo, mediante notificaciones push o por correo electrónico) para escenarios de despliegue en los que no se pueda asumir una supervisión constante del panel de visualización.

7.3 CONCLUSIONES GENERALES

El presente trabajo ha diseñado, implementado y validado de forma completa una plataforma de ingesta, almacenamiento, monitorización y detección de anomalías en tiempo

real para datos de telemetría de vuelo, cubriendo todo el ciclo descrito en los objetivos del Capítulo 2: desde la recepción y validación de cada punto de telemetría bajo un contrato estricto, pasando por su almacenamiento estructurado y su observabilidad mediante métricas y paneles, hasta la identificación en directo del estado concreto de una aeronave y su comunicación inmediata a través de un panel de visualización con alarma.

El resultado más destacable del proyecto no es únicamente que el sistema detecte anomalías, sino que sea capaz de identificar de forma específica en qué estado se encuentra una aeronave, turbulencia, desviación de ruta, fallo de motor, entre otros, en lugar de limitarse a una señal binaria de 'algo va mal'. Alcanzar este objetivo exigió recorrer un camino más largo de lo previsto inicialmente: comparar cuatro modelos de naturaleza distinta, reformular el modelo seleccionado de clasificador binario a multiclase, y, sobre todo, validar su comportamiento en un entorno de producción en tiempo real que expuso limitaciones que ninguna evaluación offline había revelado. Ese proceso de validación, con sus complicaciones y sus correcciones documentadas en los Capítulos 6 y 7, constituye en sí mismo una de las aportaciones más valiosas del trabajo: no basta con entrenar un buen modelo sobre un conjunto de test; es necesario comprobar, de forma explícita y medida, que ese rendimiento se mantiene cuando el modelo procesa datos reales que llegan uno a uno.

Las limitaciones que persisten al cierre de este trabajo, la cobertura incompleta de determinados tipos de anomalía, la dependencia de datos sintéticos en lugar de tráfico aéreo real, y la confusión residual entre ciertos estados de vuelo, están identificadas, cuantificadas y documentadas de forma transparente a lo largo de toda la memoria, y responden en su mayoría a restricciones de tiempo y de escala más que a problemas de fondo en la arquitectura o el enfoque adoptado. En conjunto, el proyecto demuestra que es posible construir, con herramientas de código abierto y en un plazo acotado, una plataforma completa de monitorización inteligente de telemetría que cubre desde la ingesta de datos en bruto hasta la explicación en lenguaje operativo de lo que le está ocurriendo a una aeronave en cada instante, sentando una base sólida sobre la que abordar las líneas de trabajo futuro descritas en la sección 7.2.

ANEXO I – BIBLIOGRAFÍA

[1] M. Kleppmann, Designing Data-Intensive Applications, O'Reilly Media, 2017.

[2] F. McCabe, J. Li, y L. Nguyen, "Event-driven architectures for real-time data ingestion," IEEE Software, vol. 36, no. 4, pp. 56–63, 2019.

[3] R. Buyya, C. Vecchiola, and S. Selvi, Mastering Cloud Computing, Morgan Kaufmann, 2013.

[4] M. Satyanarayanan, "The Emergence of Edge Computing," IEEE Computer, vol. 50, no. 1, pp. 30–39, 2017.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Communications of the ACM, 2008.

[6] B. Kreps, "Questioning the Lambda Architecture," O'Reilly Radar, 2014.

[7] T. Akidau et al., "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale Data Processing," Proc. VLDB Endowment, 2015.

[8] P. Bartholomew, "Observability in Distributed Systems," IEEE Cloud Computing, vol. 7, no. 2, pp. 10–17, 2020.

[9] N. Kreps, J. Rao, y N. Narkhede, "Kafka: A Distributed Messaging System for Log Processing," Proc. NetDB Workshop, ACM, 2011.

[10] M. Zaharia et al., "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," Proc. ACM SOSP, 2013.

[11] P. Carbone et al., "Apache Flink: Stream and Batch Processing in a Single Engine," IEEE Data Engineering Bulletin, 2015.

[12] Apache Software Foundation, "Apache Storm," Apache Software Foundation, 2023. Disponible: https://storm.apache.org

[13] Amazon Web Services, "Amazon Kinesis Data Streams," AWS Documentation, 2023. Disponible: https://aws.amazon.com/kinesis

[14] Apache Software Foundation, "Apache IoTDB: Time Series Database for IoT," 2023.

[15] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch," Journal of Algorithms, 2005.

[16] B. Sigelman et al., "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Research, 2010.

[17] Prometheus Authors, "Prometheus: Monitoring System and Time Series Database," Cloud Native Computing Foundation, 2023.

[18] Grafana Labs, "Grafana: Analytics and Monitoring Solution," Grafana Documentation, 2023.

[19] T. Goldschmidt et al., "The Netflix Atlas Telemetry System," Proc. ACM Symposium on Cloud Computing, 2015.

[20] A. Verbitski et al., "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases," Proc. ACM SIGMOD, 2017.

[21] A. Chandola, A. Banerjee, y V. Kumar, "Anomaly Detection: A Survey," ACM Computing Surveys, vol. 41, no. 3, pp. 1–58, 2009.

[22] H. Gupta et al., "Outlier Detection for Temporal Data: A Survey," IEEE Transactions on Knowledge and Data Engineering, 2014.

[23] D. Montgomery, Introduction to Statistical Quality Control, 7th ed., Wiley, 2012.

[24] F. T. Liu, K. M. Ting, y Z.-H. Zhou, "Isolation Forest," Proc. IEEE International Conference on Data Mining, 2008.

[25] S. Hochreiter y J. Schmidhuber, "Long Short-Term Memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[26] J. Dean, "Large-Scale Deep Learning for Intelligent Computer Systems," Keynote, ACM, 2016.

[27] Y. Chen et al., "LSTM-Based Anomaly Detection for Time Series," IEEE Access, 2020.

[28] I. Laptev et al., "Generic and Scalable Framework for Automated Time-Series Anomaly Detection," Proc. ACM SIGKDD, 2015.

[29] S. Vallis, J. Hochenbaum, and A. Kejariwal, "A Novel Technique for Long-Term Anomaly Detection in the Cloud," Proc. USENIX LISA, 2014.

[30] J. Brownlee, "Time Series Anomaly Detection," Machine Learning Mastery, 2020.

[31] C. Chen et al., "Time-Series Anomaly Detection Service at Microsoft," Proc. ACM SIGKDD, 2019.

[32] C. Aibar Álvarez, "Diseño de una arquitectura de procesamiento de datos en tiempo real para sistemas de monitorización," Trabajo Fin de Máster, Universidad Europea, 2021.

[33] ICAO, Automatic Dependent Surveillance–Broadcast (ADS-B) Manual, International Civil Aviation Organization, 2012.

[34] M. Schäfer, M. Strohmeier, V. Lenders, I. Martinovic, and M. Wilhelm, "Bringing Up OpenSky: A Large-scale ADS-B Sensor Network for Research," Proc. IPSN, 2014.

[35] M. Strohmeier, V. Lenders, and I. Martinovic, "On the Security of the Automatic Dependent Surveillance–Broadcast Protocol," IEEE Communications Surveys & Tutorials, 2015.

[36] OpenSky Network, "The OpenSky Network: A Large-Scale ADS-B Sensor Network," Disponible: https://opensky-network.org

[37] A. Cook and M. Tanner, European Air Traffic Management: Principles, Practice and Research, Routledge, 2015.

[38] M. Olive, M. Strohmeier, and V. Lenders, "Trajectory-based Anomaly Detection for ADS-B Flight Data," IEEE/AIAA Digital Avionics Systems Conference (DASC), 2018.

[39] M. Sun, M. Strohmeier, V. Lenders, and I. Martinovic, “Machine Learning-Based Anomaly Detection in ADS-B Flight Data,” IEEE Transactions on Intelligent Transportation Systems, 2020.

[40] OSIsoft (AVEVA), "PI System: Real-Time Data Infrastructure for Industrial Operations," AVEVA, 2023. Disponible: https://www.aveva.com/en/products/pi-system/

[41] Siemens AG, "Siemens Insights Hub (anteriormente MindSphere): Industrial IoT as a Service," Siemens Digital Industries Software, 2023. Disponible: https://siemens.com/insights-hub

[42] PTC Inc., "ThingWorx Industrial IoT Platform," PTC Documentation, 2023. Disponible: https://www.ptc.com/en/products/thingworx

[43] Amazon Web Services, "AWS IoT SiteWise," AWS Documentation, 2023. Disponible: https://aws.amazon.com/iot-sitewise

[44] Microsoft, "Azure Digital Twins," Microsoft Learn, 2023. Disponible: https://learn.microsoft.com/azure/digital-twins

[45] Datadog Inc., "Datadog: Cloud Monitoring as a Service," Datadog Documentation, 2023. Disponible: https://www.datadoghq.com

[46] Flightradar24 AB, "Flightradar24: Live Flight Tracker," 2023. Disponible: https://www.flightradar24.com

[47] FlightAware LLC, "FlightAware: Flight Tracking and Aviation Data Platform," 2023. Disponible: https://www.flightaware.com

[48] ADS-B Exchange, "ADS-B Exchange: Unfiltered, Uncensored Flight Data," 2023. Disponible: https://www.adsbexchange.com

[49] Airbus, "Skywise: The Open Aviation Data Platform," Airbus, 2023. Disponible: https://skywise.com

[50] Boeing, "AnalytX and Airplane Health Management (AHM)," Boeing Digital Aviation Solutions, 2023. Disponible: https://www.boeing.com/services/digital

[51] Honeywell International, "Honeywell Forge for Flight Efficiency," Honeywell Aerospace, 2023. Disponible: https://aerospace.honeywell.com

[52] Teledyne Controls, "Aircraft Condition Monitoring System (ACMS)," Teledyne Technologies, 2023. Disponible: https://www.teledynecontrols.com

[53] Python Software Foundation, "Python 3.11 Documentation," 2023. Disponible: https://docs.python.org/3.11/

[54] FastAPI, "FastAPI: High Performance Web Framework for Building APIs with Python," 2024. Disponible: https://fastapi.tiangolo.com

[55] Uvicorn, "Uvicorn: An ASGI Web Server for Python," 2024. Disponible: https://www.uvicorn.org

[56] Python Software Foundation, "Requests: HTTP for Humans," Requests Documentation, 2023. Disponible: https://requests.readthedocs.io

[57] Pydantic Services Inc., "Pydantic: Data Validation Using Python Type Hints," Pydantic Documentation, 2024. Disponible: https://docs.pydantic.dev

[58] S. Kumar, "python-dotenv: Read Key-Value Pairs from a .env File," 2023. Disponible: https://github.com/theskumar/python-dotenv

[59] PostgreSQL Global Development Group, "PostgreSQL 15 Documentation," 2023. Disponible: https://www.postgresql.org/docs/15/

[60] Psycopg Team, "Psycopg 3: PostgreSQL Database Adapter for Python," 2023. Disponible: https://www.psycopg.org/psycopg3/

[61] Timescale Inc., "TimescaleDB: Time-Series and Analytics Database Built on PostgreSQL," 2023. Disponible: https://www.timescale.com

[62] Docker Inc., "Docker: Accelerated Container Application Development," 2023. Disponible: https://www.docker.com

[63] Docker Inc., "Docker Compose Documentation," 2023. Disponible: https://docs.docker.com/compose/

[64] Pytest Development Team, "pytest: Helps You Write Better Programs," 2023. Disponible: https://docs.pytest.org

[65] GitHub Inc., "GitHub Actions Documentation," 2023. Disponible: https://docs.github.com/actions

[66] Meta Open Source, "React: The Library for Web and Native User Interfaces," 2024. Disponible: https://react.dev

ANEXO II – ALINEACIÓN DEL PROYECTO CON LAS ODS

ODS 9 – INDUSTRIA, INNOVACIÓN E INFRAESTRUCTURAS

Es el objetivo con el que el proyecto guarda una relación más directa. El sistema desarrollado constituye, en sí mismo, una infraestructura de innovación tecnológica: propone una arquitectura moderna y modular de ingesta, almacenamiento y análisis de telemetría en tiempo real, e incorpora técnicas de aprendizaje automático para la detección de anomalías en sistemas críticos. La meta 9.5 de este ODS, centrada en mejorar la capacidad tecnológica e impulsar la innovación, se ve reflejada directamente en la comparativa sistemática de modelos de detección de anomalías (Capítulo 6) y en el diseño de una plataforma abierta y extensible que puede adaptarse a distintos dominios de aplicación más allá del aeronáutico.

ODS 11 – CIUDADES Y COMUNIDADES SOSTENIBLES

El dominio de aplicación elegido para validar el sistema, la telemetría de vuelo, conecta el proyecto con la meta 11.2 de este ODS, orientada a proporcionar sistemas de transporte seguros, asequibles y sostenibles. Un sistema capaz de detectar en tiempo real comportamientos anómalos en una aeronave (fallos de motor, desviaciones de ruta, condiciones de vuelo adversas) contribuye, a nivel conceptual, a la seguridad y fiabilidad de

las infraestructuras de transporte aéreo, uno de los pilares de la movilidad sostenible en entornos urbanos y regionales.

ODS 4 – EDUCACIÓN DE CALIDAD

Como trabajo fin de máster, el proyecto es en sí mismo un ejercicio de formación técnica avanzada, alineado con la meta 4.4 del ODS 4, que busca aumentar el número de personas con competencias técnicas y profesionales relevantes. El desarrollo del proyecto ha exigido adquirir y aplicar de forma práctica conocimientos de arquitectura de software, bases de datos, observabilidad y aprendizaje automático, y el carácter abierto del repositorio del proyecto (Anexo III) permite que el trabajo realizado sirva como material de referencia para otros estudiantes interesados en sistemas de ingesta de datos en tiempo real o en detección de anomalías.

ODS 3 – SALUD Y BIENESTAR

De forma más indirecta, el proyecto conecta con la meta 3.6 del ODS 3, relativa a la reducción de muertes y lesiones causadas por accidentes de tráfico, incluyendo el transporte aéreo. Un sistema de detección de anomalías capaz de identificar en tiempo real el tipo concreto de incidencia que afecta a una aeronave (y no solo si existe o no una anomalía) proporciona información más accionable para una eventual respuesta operativa, lo que, escalado a un contexto de producción real, podría contribuir a anticipar y mitigar situaciones de riesgo para la seguridad de los pasajeros.