Universidades de Burgos, León y Valladolid

Máster universitario en Inteligencia de Negocio y Big Data en Entornos Seguros

TRABAJO FIN DE MÁSTER

14 DE JULIO DE 2025

Validación, almacenamiento y verificación de integridad de datos IoT con tecnologías Big Data y Blockchain

Autor:
Javier Alonso Núñez
Tutores:

Diego R. Llanos Ferraris
Carlos E. Vivaracho Pascual

Resumen

El presente Trabajo Fin de Máster propone una arquitectura distribuida para la gestión segura de datos generados por dispositivos IoT. El sistema permite recibir datos mediante el protocolo MQTT, validarlos estructuralmente con esquemas JSON, almacenarlos de forma escalable y eficiente mediante Delta Lake, y registrar huellas digitales en una red blockchain para garantizar su integridad y trazabilidad. La solución ha sido diseñada e implementada utilizando tecnologías como PySpark, AWS S3 y Smart Contracts, y permite analizar el ciclo completo de vida de los datos en entornos distribuidos. Este trabajo surge a partir de una contribución previa presentada en las jornadas SARTECO 2025, y representa una evolución del mismo hacia una arquitectura funcional y evaluable en escenarios reales.

Descriptores

Internet de las Cosas (IoT), validación de datos, Delta Lake, Blockchain, Trazabilidad, Smart Contracts, MQTT, JSON Schema

Abstract

This Master's Final Project proposes a distributed architecture for the secure management of data generated by IoT devices. The system is capable of receiving data via the MQTT protocol, validating its structure using JSON Schemas, storing it efficiently and scalably through Delta Lake, and recording digital fingerprints on a blockchain to ensure data integrity and traceability. The solution was designed and implemented using technologies such as PySpark, AWS S3, and Smart Contracts, and it enables the analysis of the complete data lifecycle in distributed environments. This work builds upon a previous contribution presented at the SARTECO 2025 conference and extends it into a functional architecture suitable for real-world scenarios.

Keywords

Internet of Things (IoT), Data validation, Delta Lake, Blockchain, Traceability, Smart Contracts, MQTT, JSON Schema

Agradecimientos

Este trabajo no habría sido posible sin todas las personas que, de una forma u otra, me han acompañado en el camino.

A mi pareja, por el cariño incondicional, los ánimos cuando más los necesitaba y por estar siempre al otro lado del teclado, aunque fueran las tantas.

Gracias a mi familia, por estar siempre ahí, por enseñarme con el ejemplo y por hacerme sentir que todo esfuerzo tiene sentido cuando se hace desde el corazón.

A mis amigos, por su paciencia, por las risas y por esos ratos de desconexión tan necesarios. También por saber escuchar, incluso cuando yo solo hablaba de código, entregas y fechas límite.

A mis profesores, por enseñarme mucho más que teoría: por su pasión, su tiempo y su capacidad de motivar. Y sobre todo, a los tutores de este trabajo, que me apoyaron, ayudaron y aconsejaron en todo momento.

Y a todas esas personas que han pasado por mi vida, dejando huella de una forma u otra. Porque de cada una he aprendido algo, y porque este trabajo también es, en parte, gracias a ellas.

Índice general

Índice de figuras

Índice de tablas

Introducción

1.1. Contexto

En los últimos años, el crecimiento exponencial de los dispositivos conectados a Internet, especialmente en el ámbito del Internet de las Cosas (IoT), ha generado una enorme cantidad de datos que requieren ser procesados, validados y almacenados de manera segura y eficiente [1] [2]. Esta proliferación de datos plantea importantes desafíos en cuanto a su integridad, trazabilidad y fiabilidad, especialmente cuando se utilizan en contextos críticos como la automatización industrial, la monitorización ambiental o la toma de decisiones en tiempo real [3].

En este contexto, surge la necesidad de desarrollar sistemas capaces de validar la estructura y consistencia de los datos en tiempo real, garantizar su almacenamiento inmutable y proporcionar mecanismos para verificar su autenticidad a lo largo del tiempo. Tecnologías como los esquemas JSON, los data lakes y las blockchains permiten construir soluciones robustas a estos desafíos [4] [5] [6], aunque su integración efectiva presenta dificultades técnicas y arquitectónicas no triviales.

El presente Trabajo Fin de Máster se enmarca dentro de esta problemática y tiene su origen en una contribución previa realizada a las jornadas SARTECO 2025 [7] con el título "Arquitectura segura para la trazabilidad basada en IoT y blockchain", donde se propuso un sistema para la validación y almacenamiento seguro de datos provenientes de dispositivos IoT, registrando además su integridad mediante blockchain. A partir de esta base, el objetivo principal de este TFM es desarrollar, implementar y evaluar una arquitectura completa que combine recepción de datos por MQTT [8], validación mediante JSON Schema

[9], persistencia con Delta Lake [10] y registro de huellas digitales en una red blockchain usando Smart Contracts [11].

1.2. Motivación

En la actualidad, el crecimiento exponencial de dispositivos conectados mediante tecnologías IoT ha generado un volumen masivo de datos que se utilizan en ámbitos críticos como la automatización industrial, la monitorización ambiental o la gestión de infraestructuras inteligentes [12] [13]. Sin embargo, la utilidad de estos datos depende directamente de su validez, trazabilidad e integridad.

Muchas soluciones actuales priorizan la captura o visualización de datos, pero descuidan aspectos fundamentales como la validación estructural, la persistencia fiable a gran escala o la verificación de su autenticidad a lo largo del tiempo. Esto representa un riesgo significativo cuando los datos se utilizan para tomar decisiones automatizadas o se exigen como evidencia en auditorías o entornos reglamentados.

En este contexto, surge la necesidad de diseñar una arquitectura capaz de:

- Validar automáticamente la estructura y consistencia de los datos desde el origen.
- Almacenarlos eficientemente en un sistema escalable que permita su explotación futura.
- Garantizar su integridad mediante mecanismos verificables e inmutables como blockchain.

Este Trabajo Fin de Máster se motiva por esta problemática real, y busca dar una respuesta técnica y viable mediante la integración de tecnologías Big Data [14] (como Delta Lake [10]) y blockchain [15] (mediante contratos inteligentes), combinadas con protocolos ligeros como MQTT y estándares de validación como JSON Schema.

1.3. Objetivos

El objetivo general de este Trabajo Fin de Máster es diseñar e implementar una arquitectura que permita la validación, almacenamiento y verificación de

integridad de datos generados por dispositivos IoT, garantizando su trazabilidad a lo largo del tiempo mediante tecnologías Big Data y blockchain.

Para alcanzar este objetivo general, se definen a continuación una serie de objetivos específicos, organizados en dos bloques: los objetivos funcionales, relacionados con los requisitos del sistema a construir; y los objetivos técnicos, centrados en los retos de diseño e implementación del proyecto.

1.3.1. Objetivos funcionales

- Recepción de datos IoT a través del protocolo MQTT, de forma asíncrona y tolerante a fallos.
- Validación estructural de los datos mediante esquemas JSON (JSON Schema), asegurando la conformidad con un modelo predefinido.
- Almacenamiento eficiente y escalable de los datos validados utilizando Delta Lake sobre un sistema de ficheros distribuido (AWS S3 [16]).
- Registro de huellas digitales (*hashes*) de los datos validados en una red blockchain para garantizar su integridad y trazabilidad.
- Persistencia de metadatos y resultados de validación, permitiendo su posterior análisis o auditoría.
- Evaluación del sistema mediante pruebas funcionales y de rendimiento que permitan medir su eficacia y eficiencia.

1.3.2. Objetivos técnicos

- Diseñar una arquitectura modular y desacoplada, compuesta por servicios independientes e interconectados.
- Aplicar buenas prácticas de ingeniería de datos: formatos eficientes (Parquet), almacenamiento en capas (bronze, silver, gold) y validación temprana.
- Integrar tecnologías heterogéneas como PySpark, Web3.py, AWS S3, MQTT y JSON Schema de forma coordinada.
- Asegurar la robustez y tolerancia a fallos ante caídas de servicios o errores en los datos.

- Optimizar el rendimiento del sistema: latencia de validación, throughput de datos y tiempo de escritura en blockchain.
- Fomentar la reproducibilidad del despliegue mediante herramientas de automatización (Docker, scripts) y documentación técnica clara.
- Implementar un entorno local que permita el despliegue y prueba de contratos inteligentes sobre una blockchain simulada, facilitando la validación del sistema sin costes asociados.

1.3.3. Resumen de objetivos

En la siguiente tabla 1.1 se muestra una correspondencia entre los objetivos definidos y los capítulos donde se desarrollan principalmente:

Objetivo	**Capítulo**
Recepción de datos vía MQTT	Capítulo 6.1
Validación con JSON Schema	Capítulo 6.1
Almacenamiento en Delta Lake sobre AWS S3	Capítulo 6.1
Registro en blockchain con Web3.py	Capítulo 6.1
Evaluación del sistema	Capítulo 6.2
Diseño modular y buenas prácticas de arquitectura	Capítulo 5.1
Integración tecnológica (PySpark, Web3.py, etc.)	Capítulo 3.3
Tolerancia a fallos y robustez	Capítulo 6.2
Optimización y rendimiento	Capítulo 6.2
Reproducibilidad del entorno	Anexos

Tabla 1.1: Relación entre objetivos y capítulos del documento

1.4. Organización de la memoria

Este documento se estructura de la siguiente manera: en el capítulo 1 se introduce el trabajo, incluyendo su motivación y los objetivos del proyecto; en el capítulo 2 se presentan los conceptos teóricos fundamentales relacionados con el ámbito del proyecto; en el capítulo 3 se describen las técnicas y herramientas utilizadas durante el desarrollo; en el capítulo 4 se detalla el análisis realizado y la planificación del proyecto; en el capítulo 5 se recoge el diseño del sistema, justificando las decisiones arquitectónicas adoptadas; en el capítulo 6 se describe la implementación técnica y las pruebas realizadas; finalmente, en el capítulo

7 se exponen las conclusiones obtenidas y las posibles líneas de trabajo futuro. Además, se incluyen dos apéndices con el manual del programador y el manual de usuario, respectivamente.

En este capítulo se ha contextualizado el problema que aborda el proyecto, motivando la necesidad de una arquitectura segura y escalable para el tratamiento de datos IoT. Además, se han definido los objetivos del trabajo y su alcance. A continuación, en el siguiente capítulo, se presentan los fundamentos teóricos y conceptuales que sustentan las decisiones tecnológicas adoptadas.

Conceptos teóricos

En este capítulo se presentan los conceptos fundamentales necesarios para la comprensión y desarrollo del proyecto. Se abordan las tecnologías clave involucradas en la arquitectura propuesta, así como los mecanismos utilizados para garantizar la integridad y trazabilidad de los datos.

2.1. Internet de las Cosas (IoT)

El Internet de las Cosas (IoT, por sus siglas en inglés) se ha consolidado como una de las tecnologías más relevantes en el ámbito de los sistemas distribuidos. Su capacidad para interconectar dispositivos físicos que recopilan, procesan y transmiten datos ha permitido el desarrollo de soluciones inteligentes en sectores tan diversos como la industria, la salud, la logística o la gestión medioambiental.

2.1.1. Definición y características principales

El IoT puede definirse como una red de objetos físicos equipados con sensores, software y conectividad, que les permite recopilar e intercambiar datos a través de Internet u otras redes. Estos objetos, también conocidos como *nodos IoT*, pueden actuar de forma autónoma o cooperativa, y están diseñados para monitorear su entorno, tomar decisiones o activar procesos en función de los datos capturados.

Las principales características del IoT son:

- **Conectividad ubicua:** los dispositivos están permanentemente conectados a redes de comunicación, ya sea WiFi, 4G/5G, LoRa, o redes de corto alcance como Zigbee o Bluetooth.
- **Sensores y actuadores:** la combinación de sensores y actuadores permite captar parámetros del entorno físico (temperatura, ubicación, humedad, etc.) y responder con acciones físicas.
- **Procesamiento local o distribuido:** muchos nodos incorporan microcontroladores que permiten realizar un preprocesamiento de los datos antes de transmitirlos.
- **Escalabilidad:** la arquitectura del IoT está diseñada para crecer con facilidad, integrando nuevos dispositivos sin comprometer la funcionalidad del sistema.

2.1.2. Aplicaciones en trazabilidad y gestión de residuos

En el contexto de este proyecto, el IoT se aplica al seguimiento y trazabilidad de residuos, mediante la instalación de dispositivos sensores en contenedores que permiten capturar datos en tiempo real sobre su estado y localización. Estas aplicaciones ofrecen múltiples beneficios:

- **Optimización de rutas de recogida:** gracias a la monitorización del nivel de llenado de los contenedores.
- **Prevención de riesgos:** mediante sensores que miden temperatura o gases peligrosos.
- **Auditoría de operaciones:** al registrar automáticamente eventos como el vaciado o traslado del contenedor.

Estos datos constituyen una fuente valiosa de información para mejorar los procesos logísticos, garantizar el cumplimiento normativo y reducir costes operativos.

2.1.3. Limitaciones y desafíos actuales

Pese a sus múltiples ventajas, la adopción del IoT plantea diversos retos técnicos y operativos que deben ser cuidadosamente abordados en entornos reales. Uno de los principales desafíos es la conectividad intermitente, ya que en

muchos contextos urbanos o industriales no se dispone de una cobertura de red estable o continua. Esta limitación obliga a diseñar estrategias de almacenamiento local de los datos o mecanismos de retransmisión diferida que garanticen la persistencia de la información. Por otro lado, el consumo energético representa una preocupación crítica, especialmente en dispositivos alimentados por batería. Es necesario optimizar su funcionamiento mediante técnicas de bajo consumo como el modo *deep sleep* y limitar la frecuencia de transmisión a lo estrictamente necesario. Además, la seguridad de los datos se convierte en un factor clave, dado que estos dispositivos operan en entornos abiertos susceptibles a ataques como la suplantación de identidad, la interceptación de datos o su alteración maliciosa. Finalmente, el crecimiento exponencial del número de dispositivos plantea importantes retos en cuanto a gestión y escalabilidad, ya que se requiere una infraestructura capaz de administrar de forma remota su configuración, credenciales y actualizaciones seguras.

Estos desafíos justifican la integración de tecnologías complementarias como protocolos de comunicación eficientes (por ejemplo, MQTT), mecanismos de validación estructural (como JSON Schema) y sistemas de trazabilidad inmutable basados en blockchain, que serán analizados en los apartados siguientes.

2.2. Protocolo MQTT

MQTT (Message Queuing Telemetry Transport) es un protocolo de mensajería ligero diseñado para comunicaciones máquina a máquina (M2M) y sistemas IoT. Su eficiencia en el uso del ancho de banda, simplicidad y bajo consumo energético lo han convertido en uno de los protocolos más adoptados para la transmisión de datos desde sensores y dispositivos conectados.

2.2.1. Modelo publicador/suscriptor

A diferencia del modelo cliente-servidor tradicional, el protocolo MQTT se basa en una arquitectura de *publicador-suscriptor*, donde los dispositivos IoT, actuando como publicadores, envían mensajes a uno o varios *topics*, mientras que otros dispositivos o servicios se suscriben a esos mismos topics para recibir únicamente la información relevante. El núcleo de esta arquitectura es el *broker*, que actúa como intermediario y gestor de los mensajes, desacoplando a los emisores de los receptores y simplificando las comunicaciones.

Este enfoque aporta varias ventajas significativas en entornos distribuidos. En primer lugar, ofrece independencia temporal entre los publicadores y los

suscriptores, que no necesitan estar conectados al mismo tiempo para que la comunicación tenga lugar. En segundo lugar, permite una alta escalabilidad, ya que múltiples clientes pueden suscribirse simultáneamente a los mismos tópicos sin que ello incremente la carga sobre los publicadores. Por último, proporciona una gestión simplificada de la distribución de mensajes, ya que es el bróker quien se encarga de enrutar los mensajes a los destinatarios adecuados según las reglas de suscripción definidas.

2.2.2. Ventajas en entornos IoT

MQTT ha sido diseñado teniendo en cuenta las restricciones inherentes a los entornos IoT, en los que la eficiencia en el uso de recursos es una prioridad. Una de sus principales ventajas es su bajo consumo de ancho de banda, ya que los mensajes transmitidos tienen un tamaño reducido y la cabecera del protocolo puede ocupar tan solo 2 bytes, lo que resulta especialmente útil en redes limitadas o de baja capacidad. Además, el protocolo está preparado para funcionar de forma eficiente en entornos con conectividad inestable, siendo capaz de tolerar fallos temporales y reconectarse automáticamente cuando la red lo permite, sin pérdida de información.

Otro aspecto destacable es la disponibilidad de niveles de calidad de servicio (QoS) configurables, que permiten adaptar el nivel de fiabilidad de la entrega de mensajes según los requerimientos del sistema, desde una entrega mínima garantizada hasta una entrega exactamente una vez, lo cual proporciona flexibilidad en el diseño del sistema. Asimismo, MQTT ofrece un amplio soporte en múltiples plataformas, con implementaciones en diversos lenguajes de programación y entornos embebidos, lo que facilita su integración en una gran variedad de dispositivos y arquitecturas.

Gracias a estas características, MQTT se presenta como una solución especialmente adecuada para escenarios en los que los dispositivos presentan conectividad limitada, baja capacidad de procesamiento o restricciones energéticas, como es el caso de los nodos IoT empleados en este proyecto.

2.2.3. Limitaciones de seguridad y escalabilidad

A pesar de sus múltiples ventajas, el uso de MQTT también conlleva ciertas limitaciones que deben considerarse cuidadosamente al diseñar sistemas seguros y escalables. Una de las principales debilidades es su seguridad limitada por defecto, ya que el protocolo no incorpora mecanismos nativos de cifrado ni autenticación robusta. Esto implica que, para garantizar la confidencialidad e

integridad de los datos transmitidos, es necesario complementarlo con protocolos adicionales como TLS o implementar soluciones de seguridad personalizadas en las capas superiores del sistema.

Otra consideración importante es el riesgo asociado al broker centralizado, que actúa como punto único de intermediación en la comunicación. Si este componente no se despliega con redundancia o técnicas de balanceo de carga adecuadas, puede convertirse en un cuello de botella o, peor aún, en un punto único de fallo o ataque que comprometa la disponibilidad del sistema completo.

Finalmente, la gestión de la autenticación y la autorización de los dispositivos y usuarios en arquitecturas a gran escala introduce una complejidad adicional. Es necesario establecer mecanismos para controlar el acceso a los distintos topics y garantizar que únicamente entidades autorizadas puedan publicar o recibir mensajes, lo cual requiere una configuración cuidadosa y una infraestructura de gestión sólida.

En este proyecto, se ha optado por MQTT como protocolo de transporte principal debido a su eficiencia y compatibilidad con dispositivos embebidos. Sus limitaciones en materia de seguridad se abordan mediante mecanismos adicionales, incluyendo la validación estructural de datos, el almacenamiento confiable en Data Lakes y el uso de blockchain para garantizar la integridad.

2.3. Validación de datos con JSON Schema

La validación de los datos en sistemas distribuidos es un paso crítico para garantizar su calidad, coherencia y adecuación al modelo esperado. En entornos IoT, donde los dispositivos pueden generar datos con estructuras variables o erróneas, establecer mecanismos automáticos de validación estructural es esencial para evitar errores posteriores en el procesamiento, almacenamiento o análisis de los datos. En este contexto, el estándar *JSON Schema* ofrece una solución robusta y ampliamente adoptada.

2.3.1. Introducción al formato JSON

El formato *JavaScript Object Notation* (JSON) se ha convertido en uno de los estándares más utilizados para el intercambio de datos entre sistemas. Su estructura ligera, basada en pares clave-valor y listas, facilita la interoperabilidad entre distintos lenguajes de programación y plataformas.

Un ejemplo sencillo de objeto JSON puede ser:

```
{
  "deviceId": "abc123",
  "timestamp": 1684700000,
  "data": {
    "temperature": 22.4,
    "humidity": 45
  }
}
```

Este formato es fácil de leer, manipular y transformar, lo que lo hace ideal para sistemas IoT que requieren eficiencia en el envío y recepción de datos.

2.3.2. JSON Schema como mecanismo de validación

JSON Schema es una especificación que permite definir de forma estructurada las reglas que debe cumplir un documento JSON. A través de este esquema es posible:

- Especificar los tipos de datos esperados (números, cadenas, objetos, etc.).
- Definir campos obligatorios (*required*) y opcionales.
- Establecer rangos válidos, patrones de texto o formatos específicos.
- Anidar validaciones en objetos complejos y listas de elementos.

Un esquema JSON para validar el ejemplo anterior podría ser:

```
{
  "type": "object",
  "required": ["deviceId", "timestamp", "data"],
  "properties": {
    "deviceId": { "type": "string" },
    "timestamp": { "type": "integer" },
```

```
    "data": {
      "type": "object",
      "required": ["temperature", "humidity"],
      "properties": {
        "temperature": { "type": "number" },
        "humidity": { "type": "integer" }
      }
    }
  }
}
```

Este tipo de validación es especialmente útil en flujos de datos continuos, permitiendo detectar errores tempranamente y rechazar mensajes malformados.

2.3.3. Ejemplos de uso y ventajas

La validación estructural mediante JSON Schema aporta una serie de ventajas clave en arquitecturas como la desarrollada en este trabajo. En primer lugar, permite la detección temprana de errores, ya que los mensajes que no cumplen con la estructura esperada pueden ser descartados antes de llegar a las etapas de almacenamiento o procesamiento, evitando así la propagación de datos inconsistentes en el sistema.

Además, esta validación garantiza la homogeneidad de los datos, al imponer una estructura unificada sobre todos los mensajes recibidos, lo que facilita su posterior análisis y tratamiento. Otra ventaja importante es la facilidad de mantenimiento, ya que los esquemas definidos en JSON Schema son fácilmente legibles y modificables, lo que permite adaptarse con rapidez a cambios en los dispositivos o a nuevos requisitos del sistema sin necesidad de reescribir grandes porciones del código.

Por último, la validación estructural permite la automatización del control de calidad, al integrarse de forma transparente en los flujos de ingesta y persistencia de datos. Esto elimina la necesidad de intervención manual en la verificación de formatos, haciendo el sistema más eficiente, fiable y escalable.

En este proyecto, JSON Schema se utiliza como primer mecanismo de control tras la recepción de datos por MQTT. Solo aquellos mensajes que superan la validación son almacenados en el Data Lake y registrados en blockchain, asegurando así una trazabilidad basada en datos fiables y estructurados.

2.4. Almacenamiento distribuido con Delta Lake

En arquitecturas que manejan grandes volúmenes de datos generados por dispositivos IoT, es fundamental contar con sistemas de almacenamiento que permitan escalar horizontalmente, mantener integridad transaccional y ofrecer flexibilidad en las consultas. Delta Lake es una solución que extiende las capacidades de los data lakes tradicionales, añadiendo características propias de las bases de datos relacionales, como transacciones ACID, manejo de versiones y consistencia de los datos.

2.4.1. Data Lake frente a base de datos tradicional

Un *Data Lake* es un repositorio centralizado que permite almacenar grandes cantidades de datos, estructurados o no estructurados, en su formato original. A diferencia de las bases de datos tradicionales, no impone un esquema rígido al momento de la escritura, lo que lo convierte en una solución ideal para flujos IoT con formatos de datos variables o en evolución.

Sin embargo, los data lakes convencionales presentan ciertas limitaciones [17]:

- No garantizan integridad transaccional.
- No permiten versiones nativas de los datos.
- Pueden presentar problemas de consistencia en operaciones concurrentes.

Para superar estas limitaciones sin perder las ventajas del almacenamiento distribuido, surge Delta Lake como una capa de abstracción que aporta funcionalidades adicionales.

2.4.2. Formato Parquet y almacenamiento en AWS S3

En este proyecto se ha optado por el uso del formato *Apache Parquet* [18], un estándar de almacenamiento columnar diseñado específicamente para opti-

mizar el análisis de grandes volúmenes de datos. Entre sus principales ventajas se encuentra su capacidad para realizar un almacenamiento eficiente mediante técnicas de compresión y codificación a nivel de columnas, lo que reduce significativamente el espacio ocupado en disco sin sacrificar rendimiento.

Además, Parquet permite un acceso rápido a subconjuntos de datos, ya que su estructura columnar facilita la lectura selectiva de campos, evitando la necesidad de cargar la totalidad del conjunto de registros en memoria. Esta característica es especialmente útil en operaciones analíticas intensivas o en entornos distribuidos donde el rendimiento de lectura es crítico.

Finalmente, su amplia compatibilidad con herramientas del ecosistema Big Data, como Apache Spark, Trino o Hive, lo convierte en una opción ideal para arquitecturas que requieren integración flexible con motores de procesamiento de datos a gran escala.

Parquet se integra perfectamente con soluciones de almacenamiento escalable como AWS S3, que ofrece alta disponibilidad, durabilidad y soporte para entornos de producción en la nube. Para pruebas locales o despliegues de bajo coste, se puede utilizar MinIO como alternativa compatible con el API de S3.

2.4.3. Delta Lake: transacciones ACID sobre Data Lakes

Delta Lake actúa como una capa de almacenamiento sobre ficheros Parquet que añade soporte para operaciones transaccionales en arquitecturas de tipo data lake. Su principal aportación es la capacidad de realizar transacciones ACID, lo que garantiza que las operaciones de escritura se ejecuten de forma atómica y consistente, evitando así lecturas intermedias, estados corruptos o pérdidas de integridad en entornos concurrentes.

Otra funcionalidad destacada es el control de versiones, que permite mantener un historial completo de los cambios realizados sobre los datos y consultar su estado en cualquier punto anterior en el tiempo, una característica conocida como *time travel*. Este mecanismo resulta especialmente útil para auditorías, análisis retrospectivos o recuperación ante errores.

Delta Lake también proporciona un manejo eficiente de archivos, mediante la compactación automática de pequeños ficheros generados por escrituras sucesivas y la optimización estructural de los datos, lo que mejora considerablemente el rendimiento de las consultas. Además, ofrece una integración nativa con Apache Spark, lo que posibilita ejecutar operaciones analíticas complejas de forma eficiente y escalable, sin necesidad de mover los datos fuera del entorno distribuido de procesamiento.

En el contexto de este proyecto, Delta Lake actúa como la capa de persistencia principal, almacenando los datos validados provenientes del sistema IoT. La integridad y trazabilidad de los datos se refuerzan posteriormente con el registro de sus huellas digitales en una red blockchain, estableciendo un vínculo verificable entre el contenido del Data Lake y su representación inmutable en la cadena de bloques.

2.5. Blockchain para trazabilidad e integridad

La tecnología *blockchain* ha transformado el modo en que se garantiza la integridad, trazabilidad y transparencia en entornos distribuidos. Originalmente concebida como soporte para criptomonedas, su uso se ha extendido a sectores como la logística, la gestión documental, la sanidad y, como en este caso, el IoT. Su capacidad para almacenar registros inmutables de forma descentralizada la convierte en una aliada clave para sistemas que requieren confiabilidad en los datos.

2.5.1. Principios básicos de blockchain

Una blockchain puede definirse como un libro de registros distribuido y vinculado criptográficamente, cuya estructura se organiza en bloques que almacenan transacciones o datos de forma secuencial e inmutable. Cada uno de estos bloques contiene un conjunto de transacciones o datos validados, lo que garantiza que la información registrada ha sido previamente verificada según las reglas de consenso de la red.

Además, cada bloque incluye un hash del bloque anterior, lo que permite encadenarlos de forma segura y resistente a manipulaciones: cualquier alteración en un bloque modificaría su hash, invalidando la integridad de toda la cadena posterior. Finalmente, cada bloque incorpora una marca de tiempo y metadatos asociados, que permiten rastrear cuándo y cómo se registró la información, contribuyendo a la trazabilidad y auditabilidad del sistema.

Las blockchains funcionan en redes de nodos donde se alcanza consenso mediante algoritmos específicos (Proof of Work, Proof of Stake, etc.) [19]. Una vez que un bloque es validado y añadido a la cadena, no puede modificarse sin alterar todos los bloques posteriores, lo que garantiza la inmutabilidad de la información.

2.5.2. Comparación con TSA (Timestamping Authority)

Una de las decisiones clave en el diseño de sistemas orientados a la integridad y trazabilidad de datos es la elección del mecanismo de verificación temporal. Tradicionalmente, se ha recurrido a Autoridades de Sellado de Tiempo (TSA)[20], que actúan como terceros de confianza, generando sellos de tiempo firmados digitalmente. Si bien estas autoridades cumplen su función en entornos centralizados, presentan limitaciones en cuanto a transparencia, resistencia a fallos y confianza descentralizada. Sin embargo, la tecnología blockchain ofrece una alternativa descentralizada y más robusta. La Tabla 2.2 presenta una comparativa entre ambos enfoques, destacando las ventajas de blockchain en términos de inmutabilidad, resistencia a fallos, transparencia y automatización mediante contratos inteligentes. Esta comparación justifica la elección de blockchain en este proyecto como tecnología base para el registro de huellas digitales de datos IoT.

Característica	**TSA (Centralizado)**	**Blockchain (Descentralizado)**
Modelo de confianza	Basado en entidad de confianza	Basado en consenso distribuido
Inmutabilidad	Depende del proveedor	Garantizada criptográficamente
Transparencia	Limitada, acceso controlado	Pública y auditable
Resistencia a fallos	Vulnerable a errores en el servidor central	Alta disponibilidad por replicación
Automatización	Requiere servicios externos	Posible mediante contratos inteligentes

Tabla 2.2: Comparativa entre TSA y blockchain para la verificación de integridad

2.5.3. Características clave: inmutabilidad, descentralización y trazabilidad

En el contexto de este proyecto, blockchain aporta un conjunto de propiedades fundamentales que refuerzan la integridad y la fiabilidad del sistema propuesto. Una de sus características más destacadas es la inmutabilidad, ya que los datos registrados en la cadena no pueden ser modificados sin invalidar toda la estructura criptográfica posterior. Esto impide cualquier tipo de altera-

ción maliciosa o no autorizada, garantizando que la información almacenada permanezca intacta desde el momento de su registro.

Otra propiedad esencial es la descentralización. Al no depender de una única entidad o nodo, la red blockchain incrementa su resiliencia frente a ataques o fallos del sistema, distribuyendo la responsabilidad de validación entre múltiples participantes. Esta característica elimina los puntos únicos de fallo y proporciona un modelo de confianza más robusto.

La trazabilidad también es un valor clave: cada dato registrado puede ser rastreado hasta su origen gracias al historial encadenado de transacciones. Esto permite auditar el ciclo de vida de la información y comprobar su legitimidad en cualquier momento. Finalmente, la verificabilidad garantiza que cualquier parte interesada pueda comprobar de forma autónoma que un dato no ha sido alterado, utilizando funciones hash y pruebas criptográficas como las pruebas de inclusión basadas en árboles de Merkle.

Estas características hacen que blockchain sea una solución idónea para reforzar la seguridad e integridad de los datos en sistemas IoT. En esta arquitectura, la blockchain actúa como un registro de huellas digitales de los datos almacenados, que pueden ser validadas posteriormente frente al contenido en el Data Lake, asegurando así la fiabilidad del sistema frente a auditorías o disputas.

2.6. Árboles de Merkle

El uso de árboles de Merkle (*Merkle Trees*)[21] es una estrategia ampliamente utilizada en sistemas distribuidos y blockchains para verificar la integridad de grandes volúmenes de datos sin necesidad de almacenarlos todos en la cadena. Esta estructura permite comprobar de forma eficiente si un determinado conjunto de datos pertenece a un bloque previamente registrado, mediante una prueba criptográfica conocida como *Merkle proof*.

2.6.1. Definición y funcionamiento

Un árbol de Merkle es una estructura de datos jerárquica en la que cada hoja representa el hash de un dato individual, y cada nodo intermedio representa el hash de la concatenación de sus nodos hijos. La raíz del árbol, conocida como *Merkle root*, resume de forma única todo el contenido del conjunto de datos.

El proceso de construcción es el siguiente:

1. Se calcula el hash de cada dato individual y se colocan como hojas del árbol.
2. Se agrupan los hashes de dos en dos y se calcula su hash combinado.
3. Se repite el proceso hasta llegar a un único nodo raíz.

Este diseño permite que cualquier modificación en un dato cambie su hash, lo que se propaga hasta la raíz, facilitando así la detección de alteraciones.

2.6.2. Ventajas en optimización de almacenamiento en blockchain

Registrar cada dato individual en una red blockchain pública puede resultar ineficiente, tanto por el espacio que ocupa como por el coste económico asociado al consumo Gas [22]. Para mitigar este problema, el uso de árboles de Merkle se presenta como una solución altamente eficiente. Gracias a su estructura jerárquica de hashes, es posible representar múltiples datos mediante un único valor: la raíz del árbol. Esto permite una notable reducción del espacio requerido, ya que únicamente se necesita almacenar esta raíz en la blockchain, en lugar de registrar cada uno de los elementos de forma individual [23].

Además, los árboles de Merkle permiten una agrupación eficiente de los datos, generando una representación criptográfica compacta que resume todo el conjunto. De esta forma, el sistema puede escalar sin comprometer la verificabilidad de los datos almacenados. La escalabilidad es otra ventaja clave, ya que permite validar la integridad de un dato concreto sin necesidad de consultar o registrar todos los hashes intermedios en la red. Este enfoque mantiene la fiabilidad del sistema y reduce significativamente los costes asociados a la persistencia en blockchain, especialmente en escenarios donde se generan grandes volúmenes de datos de forma continua.

Por ejemplo, si un mensaje contiene 10 mediciones, en lugar de registrar 10 hashes en la blockchain, se calcula la raíz del árbol Merkle correspondiente y se registra solo ese valor. Las pruebas de inclusión permiten después validar cualquier dato individual frente a la raíz almacenada.

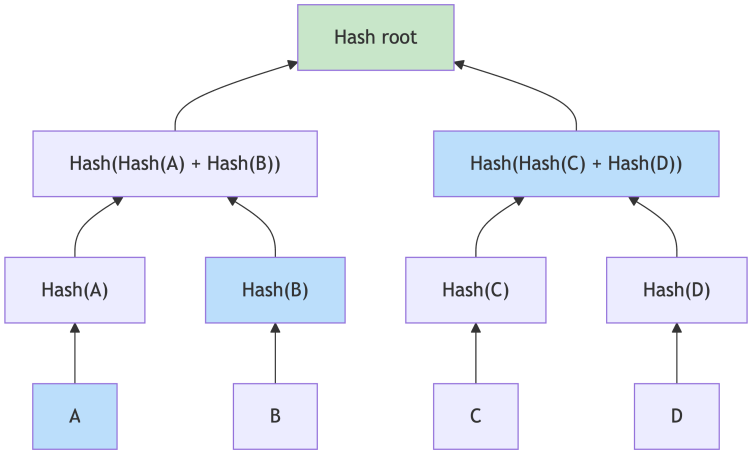

Figura 2.1: Diagrama de un arbol de Merkle

El árbol de Merkle representado en la Figura 2.1 muestra cómo es posible verificar la integridad de un dato individual, como el dato `A`, sin necesidad de recorrer o almacenar todo el conjunto de datos. Esta estructura, ampliamente utilizada en sistemas distribuidos y tecnologías blockchain, permite generar una raíz de hash que resume criptográficamente todo el contenido del árbol.

Cada nodo hoja del árbol representa el *hash* de un dato original (`A`, `B`, `C`, `D`), mientras que los nodos intermedios almacenan el *hash* de la concatenación de los hashes de sus nodos hijos. El nodo superior, denominado *hash raíz*, constituye un resumen criptográfico único de toda la estructura.

Para comprobar si el dato `A` ha sido modificado, no es necesario tener acceso a todos los datos ni recorrer el árbol completo. Basta con conocer el hash raíz y un conjunto mínimo de nodos intermedios, marcados en color azul en la figura, que actúan como *prueba de inclusión* (*Merkle proof*). Este proceso consiste en:

1. Calcular localmente el `Hash(A)` a partir del dato original.
2. Combinar este hash con el `Hash(B)` (parte de la prueba), generando el hash del nodo superior: `Hash(Hash(A) + Hash(B))`.
3. Combinar este resultado con el valor `Hash(Hash(C) + Hash(D))`, también incluido en la prueba, para obtener finalmente el `hash raíz`.

Si el valor calculado coincide con el hash raíz almacenado en la blockchain, se puede garantizar que el dato `A` no ha sido alterado.

Este mecanismo permite validar datos de forma eficiente y segura, utilizando una cantidad mínima de información, y resulta especialmente útil en entornos donde se registran grandes volúmenes de datos, pero se requiere verificar únicamente partes específicas del conjunto.

2.6.3. Pruebas de inclusión y verificación eficiente

Una *Merkle proof* es un conjunto de hashes hermanos necesarios para reconstruir la raíz del árbol partiendo del dato a verificar. Para un árbol binario con N hojas, el tamaño de la prueba es proporcional a $\lceil \log_2(N) \rceil$, lo que la hace extremadamente eficiente incluso en conjuntos grandes.

- Cada prueba contiene solo los hashes necesarios para reconstruir la cadena de combinaciones desde la hoja hasta la raíz.
- La verificación consiste en aplicar recursivamente la función de hash a los valores proporcionados y comprobar que el resultado coincide con la raíz registrada en la blockchain.

En el sistema desarrollado, las pruebas de inclusión se almacenan en el Data Lake junto con los datos originales. Así, al recalcular el hash de un dato y su Merkle proof, es posible comprobar su validez sin necesidad de acceder al resto del conjunto, reduciendo el uso de la blockchain y facilitando auditorías rápidas y precisas [23].

2.7. Contratos inteligentes y Web3

Los contratos inteligentes (*smart contracts*) son piezas de código que se ejecutan de forma automática en una red blockchain cuando se cumplen ciertas condiciones. Esta funcionalidad permite automatizar procesos sin necesidad de intermediarios, garantizando transparencia, inmutabilidad y confianza entre las partes. En este proyecto, se utilizan contratos inteligentes para registrar la integridad de los datos y permitir su verificación posterior.

2.7.1. Qué son los contratos inteligentes

Un contrato inteligente es un programa almacenado en la blockchain que define reglas y condiciones lógicas para ejecutar operaciones de forma autónoma. Estos contratos se despliegan en redes como Ethereum o Hyperledger y responden a eventos generados por usuarios o sistemas externos.

Los contratos inteligentes presentan una serie de propiedades clave que los convierten en una herramienta fundamental para la automatización segura de procesos en entornos distribuidos. En primer lugar, destacan por su carácter

descentralizado, ya que su ejecución se lleva a cabo directamente en la red blockchain, sin depender de servidores centrales ni intermediarios, lo que incrementa la resiliencia y la confiabilidad del sistema.

Otra propiedad esencial es su inmutabilidad. Una vez que un contrato ha sido desplegado en la red, su código no puede ser modificado, salvo que se hayan diseñado mecanismos explícitos de actualización. Esto garantiza que las reglas del contrato permanecen constantes, evitando manipulaciones o alteraciones posteriores.

Además, los contratos inteligentes proporcionan un alto grado de transparencia, dado que su código es público y auditable, y todas sus ejecuciones quedan registradas de forma permanente en la blockchain. Esta trazabilidad facilita la verificación independiente y la confianza entre partes que no necesariamente comparten una relación previa.

Finalmente, una de sus características más destacadas es la automatización: los contratos se ejecutan de manera automática cuando se cumplen las condiciones previamente definidas, sin necesidad de intervención humana, lo que permite construir flujos de trabajo autónomos, fiables y verificables.

En este trabajo, los contratos inteligentes se encargan de almacenar el hash raíz del árbol de Merkle generado a partir de los datos IoT, estableciendo así un vínculo entre los registros almacenados en el Data Lake y su representación verificable en la blockchain. Además, su desarrollo se ha realizado en un entorno controlado que permite la ejecución local del código y la validación automatizada mediante una suite de pruebas, lo que ha facilitado la verificación del correcto funcionamiento del contrato antes de su despliegue en producción.

2.7.2. Interacción desde Python con Web3.py

Para interactuar con contratos inteligentes desde aplicaciones externas se emplea *Web3.py*, una biblioteca de Python que permite conectarse a nodos de Ethereum, enviar transacciones, leer datos de contratos y desplegar nuevos contratos.

La biblioteca *Web3.py*, utilizada en este proyecto para interactuar con contratos inteligentes desde Python, ofrece una amplia gama de funcionalidades que facilitan la integración de aplicaciones externas con redes blockchain. Entre sus capacidades más destacadas se encuentra la posibilidad de establecer conexión con diferentes tipos de redes Ethereum, ya sean entornos locales de desarrollo, redes de pruebas (*testnet*) o la red principal (*mainnet*).

Asimismo, permite realizar tanto la lectura como la escritura de variables de contrato, facilitando el acceso a datos persistentes en la blockchain. Además, proporciona soporte para la ejecución de funciones de solo lectura (sin coste de gas) y para el envío de transacciones firmadas que modifican el estado del contrato. Otra funcionalidad relevante es la posibilidad de obtener el identificador de la transacción (*transaction hash*) y los eventos emitidos por el contrato, lo que resulta fundamental para rastrear operaciones y vincular datos persistentes con registros blockchain verificables.

Web3.py permite integrar fácilmente la lógica blockchain dentro de los flujos de procesamiento de datos en Python, lo que ha sido clave en la implementación del módulo *Blockchain Controller* del sistema.

2.7.3. Consideraciones de seguridad y gas

El diseño de contratos inteligentes requiere tener en cuenta ciertos aspectos técnicos y económicos:

- **Coste de ejecución (gas):** cada operación tiene un coste medido en gas, que debe ser pagado en la moneda nativa de la red (por ejemplo, Ether en Ethereum).
- **Eficiencia del código:** el uso de estructuras como árboles de Merkle reduce el número de transacciones y el consumo de gas, optimizando el rendimiento.
- **Errores irreversibles:** un contrato mal diseñado o con fallos puede generar pérdidas o comportamientos no deseados, por lo que es fundamental su validación exhaustiva antes del despliegue.
- **Privacidad:** la información registrada en blockchains públicas es accesible a cualquiera, por lo que debe evitarse almacenar datos sensibles directamente.

En este proyecto se ha optado por almacenar únicamente hashes, garantizando así la integridad sin comprometer la privacidad ni generar costes excesivos. Además, se han realizado pruebas en entornos de red de pruebas locales para validar el correcto funcionamiento antes de desplegar en entornos reales.

Este capítulo ha introducido los conceptos clave relacionados con IoT, blockchain, data lakes y validación de datos, proporcionando el marco teórico necesario para entender el sistema propuesto. Sobre esta base, el siguiente capítulo describe las herramientas tecnológicas y frameworks utilizados para su implementación.

Técnicas y herramientas

Este capítulo describe las técnicas metodológicas y herramientas utilizadas durante el desarrollo del proyecto. Se aborda, en primer lugar, la estrategia de trabajo adoptada y, posteriormente, las tecnologías, frameworks y librerías empleadas para implementar los diferentes componentes del sistema. También se justifica la elección de estas herramientas frente a posibles alternativas consideradas.

3.1. Metodología de desarrollo

El desarrollo del proyecto ha seguido un enfoque ágil, centrado en la entrega incremental de valor. En lugar de construir una solución monolítica en una única fase, se optó por dividir el proyecto en unidades funcionales que pudieran ser desarrolladas, probadas y evaluadas de forma independiente. Este enfoque permitió validar las decisiones arquitectónicas, detectar problemas técnicos en fases tempranas y adaptar la evolución del sistema a medida que se comprendían mejor sus necesidades operativas y de integración.

La planificación del trabajo se organizó utilizando un sistema de tareas de estilo *Kanban*, que permitía visualizar el flujo de trabajo y priorizar los objetivos de cada iteración. Cada bloque funcional, por ejemplo, la validación de datos, el almacenamiento en el data lake o la interacción con la blockchain, se trató como una unidad entregable completa, compuesta por código funcional acompañado de pruebas asociadas.

La validación de cada módulo se realizó a través de un conjunto de pruebas automatizadas y revisión manual, aplicando criterios de aceptación definidos previamente. Aunque el proyecto fue desarrollado de forma individual, hubo un componente colaborativo inicial para consensuar el esquema de datos JSON con la persona encargada del desarrollo del dispositivo IoT, garantizando así la interoperabilidad del sistema.

El entorno de trabajo se mantuvo completamente local durante todo el desarrollo, empleando contenedores para reproducir los distintos servicios. Esta estrategia permitió realizar pruebas realistas sin depender de entornos externos ni servicios en la nube, asegurando control total sobre las configuraciones y facilitando la depuración de errores en cada fase del proyecto.

3.2. Entorno de desarrollo

El entorno de desarrollo ha sido diseñado para fomentar la reproducibilidad, la modularidad y la automatización, elementos esenciales en un sistema compuesto por múltiples servicios interconectados. Desde las etapas iniciales, se priorizó la construcción de un entorno portable, que permitiera replicar la ejecución del sistema completo en diferentes máquinas sin necesidad de realizar configuraciones manuales o ajustes dependientes del sistema operativo.

Para los componentes desarrollados en Python, se utilizó Poetry como herramienta de gestión de dependencias y entornos virtuales. Esta elección permitió definir explícitamente las versiones de cada librería utilizada en el proyecto mediante los ficheros `pyproject.toml` y `poetry.lock`, facilitando tanto la instalación reproducible del entorno como su mantenimiento a lo largo del tiempo.

Todos los servicios principales, incluyendo el backend, el broker MQTT, el servicio de almacenamiento, el cliente de Web3 y la red blockchain local, han sido contenedorizados mediante Docker y orquestados con Docker Compose. Esta infraestructura permitió levantar el sistema completo de forma automatizada y coherente, simulando un entorno de producción sin necesidad de configurar manualmente cada componente. Además, la contenedorización facilitó la integración de librerías específicas, como aquellas necesarias para la interacción con MinIO, PySpark o Web3.py, reduciendo problemas de compatibilidad o configuración entre sistemas.

El control de versiones se ha gestionado mediante *Git*, con un repositorio público en *GitHub* que contiene tanto el código fuente como la documentación técnica mínima necesaria para su instalación y ejecución. En particular, el fichero `README.md` incluye instrucciones para levantar el entorno con Docker, ejecutar

las pruebas automatizadas y verificar la interacción entre componentes. Esta estrategia no solo facilita la colaboración futura y la extensión del sistema, sino que también mejora la trazabilidad de los cambios y el mantenimiento del código.

Además, el uso de herramientas como `.env` para la configuración de variables sensibles y la organización del código por servicios (según patrones de diseño modulares) contribuyó a mantener un entorno limpio, estructurado y fácilmente escalable. Esta aproximación también facilita su posterior despliegue en entornos más complejos, como servidores de integración continua o plataformas cloud.

3.3. Tecnologías utilizadas

3.3.1. Procesamiento y validación de datos

Una de las prioridades del sistema propuesto ha sido garantizar que los datos recibidos desde los dispositivos IoT tengan una estructura coherente, predecible y válida antes de ser almacenados o procesados. Para ello, se ha empleado JSON Schema como mecanismo de validación estructural. Esta tecnología permite definir con precisión el formato esperado de cada mensaje, incluyendo tipos de datos, campos obligatorios, formatos específicos y estructuras anidadas, y aplicar automáticamente reglas de validación sobre cada mensaje recibido. Los mensajes que no cumplen con el esquema definido son descartados, evitando así la propagación de datos erróneos o incompletos al sistema de persistencia y análisis.

La validación mediante JSON Schema se integra en la etapa de ingesta, justo después de recibir el mensaje a través de MQTT, y actúa como un primer filtro de control de calidad de los datos. Esta validación temprana reduce la complejidad de las etapas posteriores del sistema y mejora su robustez frente a entradas inesperadas o corruptas.

Para el procesamiento y transformación de los datos validados, se ha utilizado PySpark, la interfaz de Python para Apache Spark. Esta herramienta proporciona un motor de análisis distribuido capaz de manejar grandes volúmenes de datos de forma paralela, lo cual resulta especialmente útil cuando se trabaja con historiales IoT extensos o se requieren aplicar operaciones de agregación, filtrado o análisis temporal a gran escala.

Sobre el sistema de ficheros distribuido se ha empleado Delta Lake como capa de almacenamiento transaccional. Delta Lake extiende las capacidades del formato Parquet con funcionalidades adicionales como control de versiones,

manejo eficiente de actualizaciones e inserciones, y soporte para transacciones ACID. Esta capa permite mantener la consistencia del sistema incluso en escenarios de escritura concurrente, y facilita el desarrollo de pipelines de datos escalables que pueden evolucionar sin perder control sobre el linaje de los datos.

La combinación de PySpark con Delta Lake proporciona un entorno de procesamiento robusto, flexible y orientado a la analítica de datos históricos, alineado con las necesidades de un sistema IoT orientado a la trazabilidad e integridad de los datos. Además, esta arquitectura deja abierta la posibilidad de incorporar futuras extensiones basadas en machine learning o inteligencia artificial, aprovechando el mismo ecosistema de herramientas.

3.3.2. Comunicación IoT

La comunicación entre los dispositivos IoT y el sistema backend se ha implementado utilizando el protocolo MQTT (*Message Queuing Telemetry Transport*), ampliamente reconocido por su ligereza, eficiencia y orientación a entornos con restricciones de ancho de banda o conectividad intermitente. Su modelo de comunicación basado en el patrón *publicador-suscriptor*, junto con una arquitectura centrada en un *broker*, lo convierten en una solución ideal para sistemas distribuidos donde múltiples sensores deben enviar datos de forma periódica a un sistema centralizado sin acoplamiento directo entre emisores y receptores.

En este proyecto, se ha adoptado Eclipse Mosquitto como broker MQTT por tratarse de una solución ligera, de código abierto y de fácil integración, tanto en entornos de desarrollo como en despliegues de producción a pequeña escala. Mosquitto ofrece un rendimiento estable con bajo consumo de recursos, lo que facilita su ejecución en contenedores dentro del entorno Docker orquestado del sistema.

La recepción de los mensajes en el backend desarrollado en Python se ha implementado mediante la librería paho-mqtt, mantenida por la Eclipse Foundation. Esta librería proporciona una interfaz sencilla y eficiente para conectarse al broker, suscribirse a *topics*, recibir mensajes en tiempo real y gestionarlos de manera asíncrona. Su compatibilidad con múltiples versiones del protocolo y su documentación consolidada han sido factores clave en su elección.

Aunque existen otras alternativas comerciales y open source como HiveMQ o EMQX, se optó por Mosquitto por su facilidad de configuración, su uso extendido en proyectos IoT reales y su integración directa con bibliotecas de cliente ampliamente soportadas. Esta elección ha permitido centrar el esfuerzo de desarrollo en la lógica de validación, procesamiento y trazabilidad, minimizando

las fricciones técnicas en la capa de comunicación. En futuras versiones del proyecto, Mosquitto podría ser reemplazado por brokers más robustos o escalables como EMQX, especialmente si se requiere autenticación avanzada, clustering o métricas en tiempo real.

3.3.3. Persistencia de datos

Una vez validados, los datos IoT deben ser almacenados de forma segura, estructurada y con soporte para consultas analíticas y trazabilidad histórica. Para cumplir con estos requisitos, se ha implementado un data lake utilizando Delta Lake como capa de control transaccional sobre un sistema de almacenamiento basado en MinIO, una solución compatible con la API de *Amazon S3*. Cabe destacar que, aunque en este proyecto se ha utilizado MinIO como sistema de almacenamiento local compatible con la API de Amazon S3, Delta Lake ofrece soporte para conectarse de forma transparente a otros servicios de almacenamiento en la nube como Amazon S3, Azure Data Lake Storage o Google Cloud Storage. Esto facilita la portabilidad de la solución y su posible escalado a entornos productivos en la nube, sin requerir cambios significativos en la lógica de acceso a datos.

El uso de Delta Lake permite organizar el almacenamiento en diferentes capas, comúnmente conocidas como *bronze*, *silver* y *gold*, lo que facilita la evolución progresiva de los datos desde su estado original hasta versiones limpias y enriquecidas, listas para su análisis. Esta separación no solo mejora la organización lógica de la información, sino que permite definir políticas de retención, control de calidad y versionado sobre cada nivel del ciclo de vida de los datos.

Una de las principales ventajas de Delta Lake es su soporte para transacciones ACID, incluso en entornos de ficheros distribuidos como S3 o MinIO. Gracias a este mecanismo, se garantiza que las operaciones de inserción, actualización o eliminación se realizan de manera consistente y segura, evitando estados intermedios o duplicidades que puedan comprometer la integridad del sistema. Además, Delta Lake permite realizar *time travel*, es decir, consultar el estado histórico de los datos en versiones anteriores, lo que resulta de gran utilidad en procesos de auditoría o en la reconstrucción de eventos.

Para el almacenamiento físico, se ha utilizado MinIO por tratarse de una solución ligera, autoalojada y completamente compatible con la interfaz de AWS S3. Esta elección ha permitido simular un entorno de almacenamiento en la nube dentro del entorno de desarrollo local mediante contenedores Docker, facilitando la portabilidad del sistema sin dependencia directa de servicios externos.

Si bien inicialmente se valoró el uso de bases de datos relacionales convencionales para la persistencia, estas fueron descartadas por su menor flexibilidad en el manejo de datos semiestructurados y por sus limitaciones en escenarios de escalabilidad horizontal y análisis histórico. La arquitectura basada en Delta Lake no solo ofrece un rendimiento adecuado para flujos de datos IoT, sino que también deja la puerta abierta a futuras integraciones con herramientas del ecosistema Big Data como Apache Spark, Trino o Apache Flink, lo que refuerza su elección como componente clave del sistema.

3.3.4. Blockchain y contratos inteligentes

La verificación de la integridad de los datos IoT se ha implementado mediante el uso de contratos inteligentes desarrollados en Solidity y desplegados sobre una red Ethereum local, simulada y gestionada con Hardhat. Estos contratos actúan como anclas criptográficas que registran las huellas digitales de los datos validados, en forma de hashes raíz de árboles de Merkle, proporcionando un mecanismo descentralizado, inmutable y verificable para garantizar que la información almacenada no ha sido manipulada tras su recepción.

La elección de Ethereum como plataforma blockchain se fundamentó en su amplio soporte para contratos inteligentes, su madurez tecnológica y la disponibilidad de un conjunto consolidado de herramientas de desarrollo. Aunque se valoró el uso de alternativas como Hyperledger Fabric, esta fue descartada por la mayor complejidad que implica su despliegue y configuración, así como por su menor grado de integración con herramientas comunes del ecosistema Python. En un entorno de trabajo individual, y con necesidad de reproducibilidad local, Ethereum y sus herramientas asociadas ofrecieron una solución más directa, eficiente y alineada con los objetivos del proyecto.

Para facilitar la interacción entre el backend desarrollado en Python y la red Ethereum, se ha utilizado la biblioteca Web3.py. Esta librería proporciona una interfaz completa para conectarse a nodos Ethereum, desplegar contratos, enviar transacciones, leer eventos emitidos por los contratos y consultar datos almacenados en la blockchain. Su integración directa con el lenguaje Python fue decisiva para su elección frente a alternativas como Web3.js, permitiendo mantener coherencia en el stack tecnológico y evitando la necesidad de introducir un componente adicional basado en JavaScript en un entorno centrado en Python.

El entorno de pruebas se construyó utilizando la funcionalidad de redes locales de Hardhat, lo que permitió simular el comportamiento de una red Ethereum sin necesidad de acceder a testnets públicas. Esta configuración facilitó

una ejecución rápida y controlada de pruebas sobre los contratos inteligentes, incluyendo la validación de funcionalidades como el almacenamiento de hashes, la recuperación de identificadores de transacción y la verificación de la integridad de datos mediante pruebas de inclusión. Adicionalmente, se desarrollaron pruebas automatizadas para los contratos utilizando la infraestructura de testing integrada en Hardhat, garantizando la robustez del código antes de integrarlo con el resto del sistema. No obstante, el uso de Hardhat como entorno de desarrollo no permite persistir el estado de la blockchain entre ejecuciones, por lo que al reiniciar el contenedor se pierde toda la información almacenada. Para solventar esta limitación en la fase final del proyecto, se sustituyó Hardhat por Ganache como nodo de blockchain, exclusivamente en esta parte del sistema, permitiendo así conservar el estado de la cadena entre sesiones.

Esta integración entre contratos inteligentes y backend permite construir un sistema auditable, en el que cada dato IoT almacenado en el Data Lake puede ser verificado a posteriori mediante su hash raíz registrado en la blockchain, asegurando así la trazabilidad y fiabilidad del sistema en todo momento.

3.3.5. Pruebas y despliegue

La fiabilidad del sistema ha sido garantizada mediante una combinación de pruebas unitarias, centradas en módulos individuales, y pruebas de integración, orientadas a validar el correcto funcionamiento entre componentes. Este enfoque ha permitido asegurar tanto el comportamiento aislado de cada parte como su interacción en un entorno completo.

En el caso de los contratos inteligentes, se ha utilizado la suite de pruebas incluida en Hardhat, que ofrece un entorno robusto para simular llamadas, eventos y transacciones en una red Ethereum local. Las pruebas han cubierto funcionalidades críticas como el almacenamiento de hashes, la recuperación de identificadores de transacción y la verificación de datos mediante estructuras de Merkle. Este entorno también ha permitido ejecutar escenarios de error y comprobar el manejo de excepciones dentro del contrato.

En el backend desarrollado en Python, se han implementado pruebas automatizadas utilizando *pytest*, con el objetivo de validar la lógica asociada a la ingesta de mensajes, la validación mediante JSON Schema y la persistencia en el Data Lake. Estas pruebas han sido especialmente importantes para garantizar la robustez del sistema frente a entradas malformadas, errores en el procesamiento de datos o pérdidas de conectividad con los servicios asociados.

Para garantizar coherencia entre los entornos de desarrollo, prueba y ejecución, se ha adoptado una estrategia de despliegue local basado en contenedores, utilizando Docker para empaquetar todos los servicios necesarios: el broker MQTT, el backend de procesamiento, el servicio de almacenamiento basado en MinIO y la red blockchain local gestionada con Hardhat. La orquestación de estos servicios se realiza mediante Docker Compose, lo que facilita la configuración inicial, el arranque simultáneo y la monitorización de todos los componentes del sistema.

Este enfoque permite replicar de forma exacta el entorno de desarrollo en cualquier máquina, evitando problemas derivados de diferencias de configuración, versiones de librerías o dependencias del sistema operativo. Además, deja abierta la posibilidad de extender el sistema a entornos de integración continua o despliegue en la nube, manteniendo una base sólida y automatizada para futuras evoluciones del proyecto.

3.4. Justificación de las elecciones tecnológicas

Durante el desarrollo del proyecto se analizaron distintas alternativas tecnológicas para cada uno de los componentes clave del sistema. La elección final de herramientas y plataformas respondió tanto a criterios técnicos como a la coherencia con el stack general de desarrollo, priorizando la integración fluida entre módulos, la facilidad de uso en entornos locales y la proyección a futuro del sistema.

En lo referente a la interacción con la red blockchain, se consideró inicialmente el uso de Web3.js, una librería ampliamente utilizada en el ecosistema Ethereum para desarrollos en JavaScript. Sin embargo, se optó por Web3.py debido a su integración natural con el backend implementado en Python. Esta elección evitó la incorporación de dependencias innecesarias en otros lenguajes y permitió mantener un stack tecnológico homogéneo, reduciendo la complejidad del desarrollo y favoreciendo la mantenibilidad del código.

En cuanto al almacenamiento persistente de los datos IoT, se evaluó el uso de bases de datos relacionales tradicionales como alternativa al enfoque basado en data lakes. No obstante, estas fueron descartadas por su menor capacidad de adaptación a flujos de datos semiestructurados, su rigidez en esquemas y su menor eficiencia en escenarios de análisis de grandes volúmenes históricos. Por el contrario, la combinación de Delta Lake con almacenamiento S3-compatible (MinIO) ofrecía una solución más adecuada a los requerimientos del sistema, pro-

porcionando flexibilidad, control de versiones y compatibilidad con herramientas del ecosistema Big Data.

Asimismo, se valoró la posibilidad de utilizar Hyperledger como plataforma blockchain. A pesar de sus ventajas en entornos corporativos privados, su complejidad de despliegue y configuración, unida a su menor integración con herramientas como Web3.py o Hardhat, hicieron que Ethereum resultara más conveniente para los fines del proyecto. La existencia de entornos de desarrollo maduros, documentación abundante y herramientas de testing como Hardhat fueron factores clave para esta decisión.

En conjunto, las elecciones tecnológicas adoptadas han permitido construir un sistema modular, reproducible y fácilmente ampliable. La selección de herramientas ampliamente adoptadas y bien integradas entre sí ha facilitado el desarrollo y ha reducido las barreras técnicas en fases críticas del proyecto, asegurando una base sólida sobre la que se podrían realizar futuras evoluciones o despliegues en producción.

Resumen del capítulo

A lo largo de este capítulo se han descrito las principales decisiones metodológicas y técnicas adoptadas durante el desarrollo del proyecto. Desde el uso de un enfoque ágil basado en entregas incrementales, hasta la integración de tecnologías especializadas en validación, almacenamiento y trazabilidad de datos, cada componente ha sido seleccionado y configurado en función de su idoneidad para los objetivos definidos.

La elección de herramientas como JSON Schema, PySpark, Delta Lake, Web3.py y Hardhat ha permitido construir un sistema modular, reproducible y orientado a la escalabilidad. Asimismo, el uso de Docker como plataforma de despliegue ha facilitado la coherencia entre entornos y ha contribuido a la mantenibilidad del sistema a largo plazo. En conjunto, estas técnicas y herramientas han conformado una arquitectura robusta y versátil, adecuada para gestionar datos IoT con garantías de integridad y trazabilidad.

Por lo tanto, las herramientas empleadas a lo largo del proyecto han sido seleccionadas cuidadosamente en función de su propósito específico y su compatibilidad con el resto del sistema, como se muestra en la Tabla 3.3.

Herramienta	**Finalidad**	**Justificación**
Poetry	Gestión de dependencias en Python	Aislamiento y reproducibilidad del entorno
Docker / Docker Compose	Contenerización y orquestación de servicios	Despliegue local reproducible y modular
PySpark	Procesamiento distribuido de datos IoT	Escalabilidad y compatibilidad con Big Data
Delta Lake	Almacenamiento con control ACID	Integridad y versionado de datos
MinIO	Almacenamiento S3-compatible local	Simulación de almacenamiento cloud
JSON Schema	Validación estructural de mensajes	Prevención de errores en la ingesta de datos
MQTT + Mosquitto	Comunicación IoT eficiente	Protocolos ligeros y fiables para IoT
Web3.py	Interacción con blockchain desde Python	Integración directa con el stack del backend
Solidity + Hardhat	Contratos inteligentes y pruebas	Trazabilidad y verificación de integridad en blockchain
pytest	Pruebas automatizadas en Python	Verificación funcional de los módulos
Git + GitHub	Control de versiones y documentación	Trazabilidad del código y colaboración futura

Tabla 3.3: Resumen de herramientas utilizadas y su función en el sistema

Tras revisar las tecnologías utilizadas como MQTT, Delta Lake, Ethereum, etc, este capítulo ha sentado las bases prácticas del desarrollo. En el capítulo siguiente se explican los detalles de análisis y planificación que permitieron estructurar la solución propuesta.

Análisis y Plan de Proyecto

4.1. Análisis de requisitos

Antes del diseño e implementación del sistema, resulta fundamental identificar y analizar los requisitos que deben cumplirse para garantizar que la solución propuesta sea funcional, escalable y alineada con los objetivos del proyecto. Esta sección presenta un análisis detallado de los requisitos tanto funcionales como no funcionales, teniendo en cuenta las necesidades del sistema desde el punto de vista del usuario final, los componentes tecnológicos involucrados y las restricciones propias del contexto IoT y blockchain.

El análisis se ha estructurado en torno a los siguientes aspectos: funcionalidades clave esperadas del sistema, rendimiento, seguridad, persistencia de datos, validación, trazabilidad y compatibilidad con tecnologías de despliegue y desarrollo.

4.1.1. Requisitos

El sistema propuesto debe cumplir una serie de requisitos para garantizar su funcionalidad, rendimiento y viabilidad técnica. A continuación, se detallan los requisitos funcionales y no funcionales identificados durante el desarrollo del proyecto.

4.1.2. Requisitos funcionales

Los requisitos funcionales describen las capacidades y servicios específicos que debe ofrecer el sistema. Entre ellos se encuentran:

- **RF1: Recolección de datos IoT.** El sistema debe ser capaz de recibir datos en tiempo real procedentes de dispositivos IoT mediante el protocolo MQTT.
- **RF2: Validación de datos.** Cada mensaje recibido debe ser validado conforme a un esquema definido en JSON Schema antes de su procesamiento.
- **RF3: Almacenamiento de datos.** Los datos validados deben almacenarse en un sistema de ficheros tipo data lake basado en Delta Lake sobre almacenamiento AWS S3 o compatible.
- **RF4: Registro de integridad en blockchain.** El sistema debe calcular un hash del conjunto de los datos a almacenar de un mensaje y registrar dicho hash en una red blockchain.
- **RF5: Verificación de integridad.** Debe existir una funcionalidad para comprobar la integridad de los datos almacenados a partir del hash registrado en blockchain.
- **RF6: Interfaz de usuario.** Se debe proporcionar una interfaz web sencilla para mostrar los datos almacenados y facilitar la verificación de su integridad.

4.1.3. Requisitos no funcionales

Los requisitos no funcionales definen características de calidad que debe cumplir el sistema, aunque no estén directamente relacionadas con funcionalidades específicas:

- **RNF1: Escalabilidad.** El sistema debe ser capaz de escalar horizontalmente para gestionar un aumento en el número de dispositivos IoT o en el volumen de datos.
- **RNF2: Fiabilidad.** El sistema debe garantizar la disponibilidad y consistencia de los datos, incluso en caso de pérdida de conectividad temporal.

- **RNF3: Seguridad.** La arquitectura debe asegurar la confidencialidad, integridad y autenticidad de los datos durante su transmisión, almacenamiento y registro.

- **RNF4: Trazabilidad.** Debe ser posible rastrear el origen y evolución de cada dato, incluyendo su hash y el momento de registro en la blockchain.

- **RNF5: Compatibilidad.** El sistema debe estar diseñado para integrarse fácilmente con otras herramientas del ecosistema Big Data o con soluciones blockchain híbridas.

- **RNF6: Usabilidad.** La interfaz de usuario debe ser intuitiva y accesible para usuarios técnicos y no técnicos.

4.1.4. Casos de uso

El análisis de casos de uso permite identificar los principales actores que interactúan con el sistema y las funcionalidades clave que deben estar disponibles. A partir de estos casos, se definen los requisitos funcionales y se orienta el diseño técnico del sistema.

En el contexto de este proyecto, los principales actores son:

- **Usuario final:** Consulta datos, verifica su integridad y visualiza el estado del sistema desde la interfaz web.

- **Dispositivo IoT:** Encargado de enviar datos sensorizados al sistema mediante MQTT.

A continuación, en la Figura 4.2, se presenta el diagrama de casos de uso que resume las principales interacciones entre los actores y el sistema.

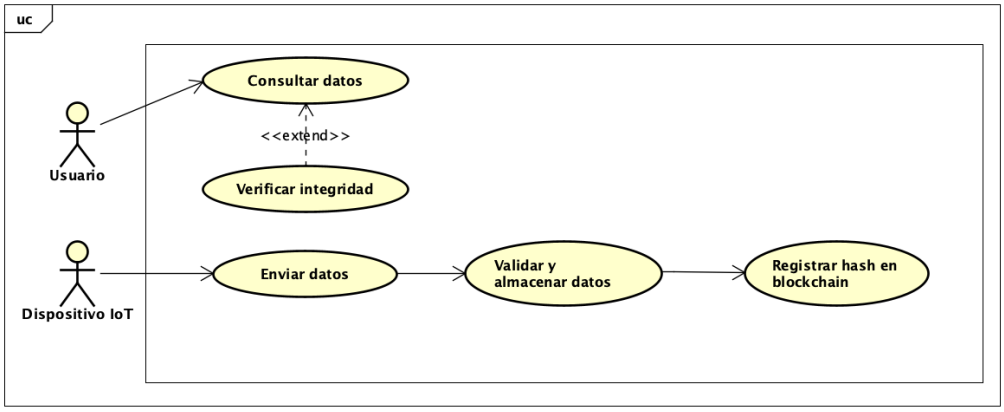

Figura 4.2: Diagrama de casos de uso

Asimismo, para detallar el flujo de operaciones internas de los diferentes casos de uso, se incluyen una serie de diagramas de actividades que representan el proceso completo del caso de uso.

4.1.4.1. UC1 - Consultar datos

La funcionalidad de consulta de datos permite al usuario acceder a la información sensorizada previamente registrada por los dispositivos IoT asociados a su cuenta. Este caso de uso refleja una de las interacciones principales con el sistema, ya que proporciona visibilidad sobre los datos almacenados en el data lake, así como la base para otras operaciones como la verificación de integridad.

En la Figura 4.3 se presenta el diagrama de casos de uso correspondiente, que ilustra la interacción entre el usuario y el sistema. A continuación, la Tabla 4.4 detalla la descripción completa del caso de uso UC1, incluyendo su secuencia normal, flujos alternativos y postcondiciones.

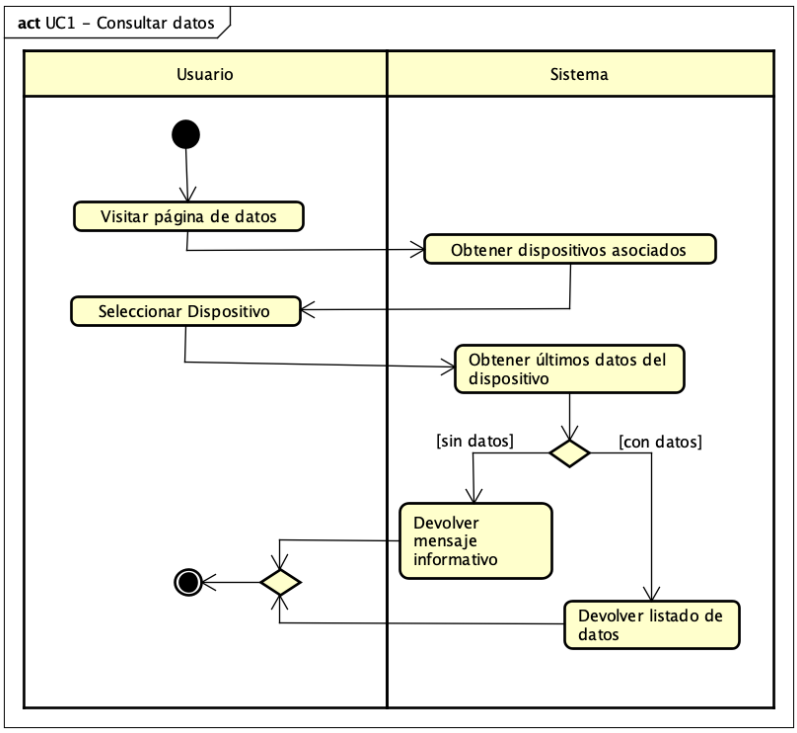

Figura 4.3: Diagrama del caso de uso de consultar datos

UC1	**Consultar datos**
Descripción	El usuario consulta los datos almacenados de un dispositivo IoT desde la interfaz del sistema.
Secuencia normal	1. El usuario accede a la página de visualización de datos. 2. El sistema obtiene los dispositivos asociados al usuario. 3. El usuario selecciona un dispositivo de la lista. 4. El sistema recupera los últimos datos disponibles del dispositivo. 5. El sistema muestra el listado de datos al usuario.
Flujos alternativos	4a-1. No existen datos registrados para el dispositivo. 4a-2. El sistema informa al usuario de que no hay datos disponibles. 4a-3. El caso de uso finaliza sin mostrar datos.
Postcondición	El usuario ha visualizado los datos disponibles o ha sido informado de su ausencia.

Tabla 4.4: Caso de uso UC1: Consultar datos

4.1.4.2. UC2 - Verificar integridad

La funcionalidad de verificación de integridad permite al usuario comprobar si los datos almacenados de un dispositivo IoT coinciden con el valor hash previamente registrado en la blockchain. Este caso de uso es fundamental para garantizar la trazabilidad y confiabilidad de la información registrada en el sistema, detectando posibles alteraciones o manipulaciones.

La Figura 4.4 muestra el diagrama de actividades asociado al caso de uso, donde se detallan las acciones tanto del usuario como del sistema. Posteriormente, la Tabla 4.5 recoge la descripción formal del caso de uso UC2, incluyendo su flujo principal, flujos alternativos y la postcondición esperada.

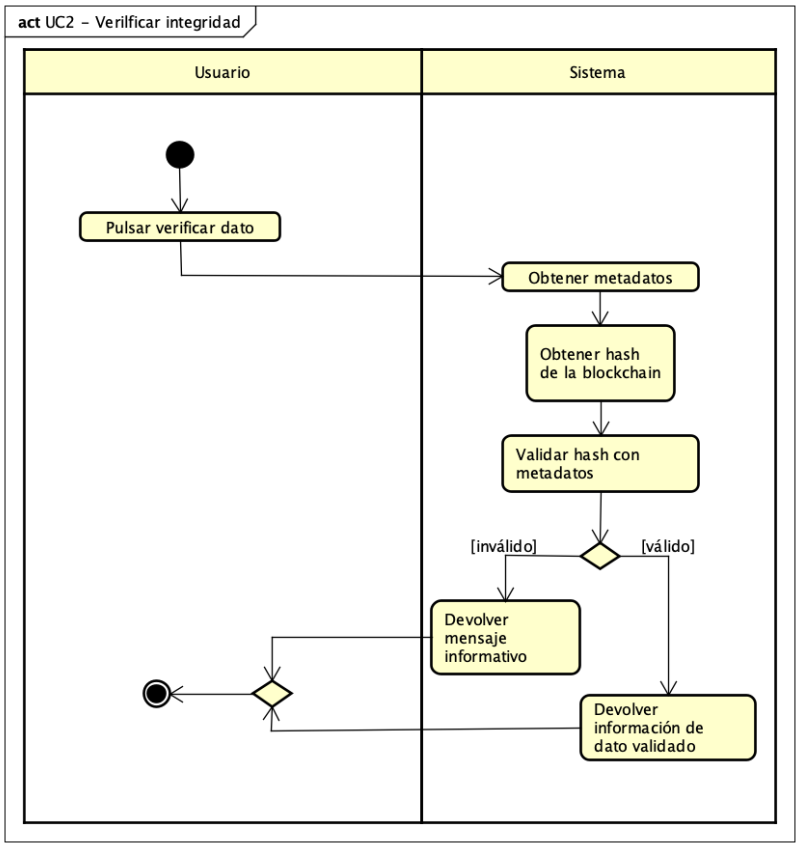

Figura 4.4: Diagrama del caso de uso de verificar integridad

UC2	**Verificar integridad**
Descripción	El usuario solicita comprobar si los datos almacenados de un dispositivo coinciden con el hash registrado en la blockchain, validando así su integridad.
Secuencia normal	1. El usuario pulsa la opción para verificar un dato desde la interfaz. 2. El sistema obtiene los metadatos asociados al dato. 3. El sistema recupera el hash previamente registrado en la blockchain. 4. El sistema compara el hash calculado con los metadatos y el registrado en la blockchain. 5. El sistema devuelve al usuario la información de verificación si la integridad es válida.
Flujos alternativos	4a-1. El hash calculado no coincide con el registrado en la blockchain. 4a-2. El sistema informa al usuario de que la integridad del dato no ha podido ser verificada. 4a-3. El caso de uso finaliza sin mostrar la información validada.
Postcondición	El usuario ha sido informado de si el dato consultado mantiene su integridad respecto al valor registrado en la blockchain.

Tabla 4.5: Caso de uso UC2: Verificar integridad

4.1.4.3. UC3 - Envío de datos

El envío de datos constituye el punto de entrada del sistema, en el que los dispositivos IoT transmiten información sensorizada en formato JSON. Esta funcionalidad permite al sistema recibir, validar y almacenar dichos datos de

forma segura, además de registrar un identificador de integridad (hash) en la blockchain como mecanismo de trazabilidad y verificación futura.

La Figura 4.5 muestra el diagrama de actividades asociado al caso de uso UC3, en el que se representan los pasos realizados tanto por el dispositivo como por el sistema, incluyendo el tratamiento de errores en caso de que el mensaje recibido no cumpla con el formato esperado. A continuación, en la Tabla 4.6 se recoge la especificación completa del caso de uso.

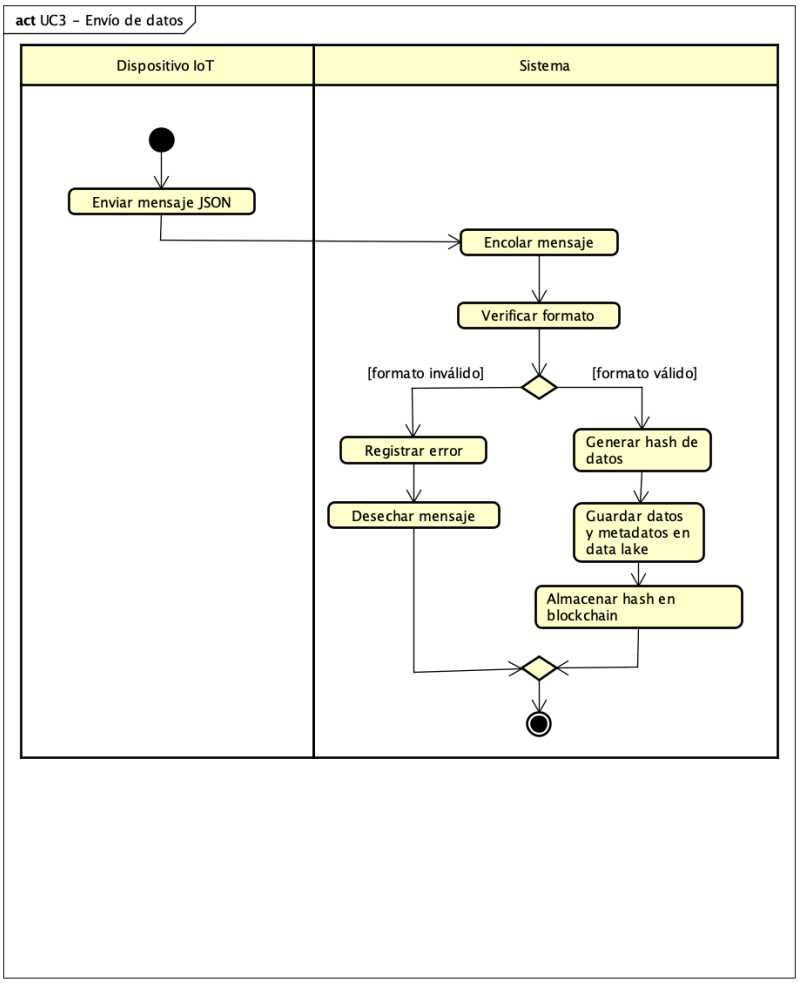

Figura 4.5: Diagrama del caso de uso de envío de datos

UC3	**Envío de datos**
Descripción	Un dispositivo IoT envía un mensaje JSON con datos sensorizados al sistema. Este los procesa, valida y registra si cumplen el formato establecido. En caso contrario, se registra el error y se descarta el mensaje.
Secuencia normal	1. El dispositivo IoT envía un mensaje en formato JSON. 2. El sistema encola el mensaje recibido. 3. El sistema verifica que el formato del mensaje sea válido. 4. Si el mensaje es válido: *a*) Se genera un hash a partir de los datos. *b*) Se almacenan los datos y metadatos en el data lake. *c*) Se registra el hash en la blockchain.
Flujos alternativos	3a-1. El mensaje no cumple con el formato esperado. 3a-2. El sistema registra un error asociado al mensaje. 3a-3. El mensaje es descartado y no se almacena.
Postcondición	El mensaje ha sido procesado y almacenado correctamente con su hash registrado en blockchain, o ha sido descartado tras su validación fallida.

Tabla 4.6: Caso de uso UC3: Envío de datos

4.2. Plan de proyecto

El objetivo de esta sección es definir el plan de proyecto seguido durante el desarrollo del Trabajo Fin de Máster. Se describe la planificación temporal, así como un estudio básico de viabilidad en sus dimensiones económica y legal. Dado el carácter académico del proyecto, la planificación se ha ajustado al calendario del curso y a los hitos intermedios de entrega.

4.2.1. Planificación temporal

El proyecto ha sido desarrollado a lo largo de varios meses siguiendo un enfoque iterativo. Cada bloque funcional se ha abordado como una unidad

independiente de trabajo, permitiendo realizar entregas parciales que aportaban valor y facilitaban la integración progresiva de los componentes.

La tabla 4.7 presenta una estimación de la duración y dedicación horaria aproximada de cada fase del proyecto, considerando una media de 20 horas semanales de dedicación:

Fase	**Duración**	**Horas estimadas**
Análisis y diseño preliminar	3 semanas	60 h
Validación de datos con JSON Schema	2 semanas	40 h
Persistencia en Delta Lake y pruebas locales con MinIO	3 semanas	60 h
Desarrollo del contrato inteligente y pruebas con Hardhat	2 semanas	40 h
Integración del backend con Web3.py	2 semanas	40 h
Pruebas funcionales e integración total con Docker Compose	3 semanas	60 h
Documentación, redacción de memoria y presentación	4 semanas	80 h
Total estimado	19 semanas	**380 h**

Tabla 4.7: Planificación temporal y estimación horaria del proyecto

La planificación temporal detallada puede visualizarse en el diagrama de Gantt de la figura 4.6, donde se muestra la distribución estimada de las tareas a lo largo del calendario de desarrollo.

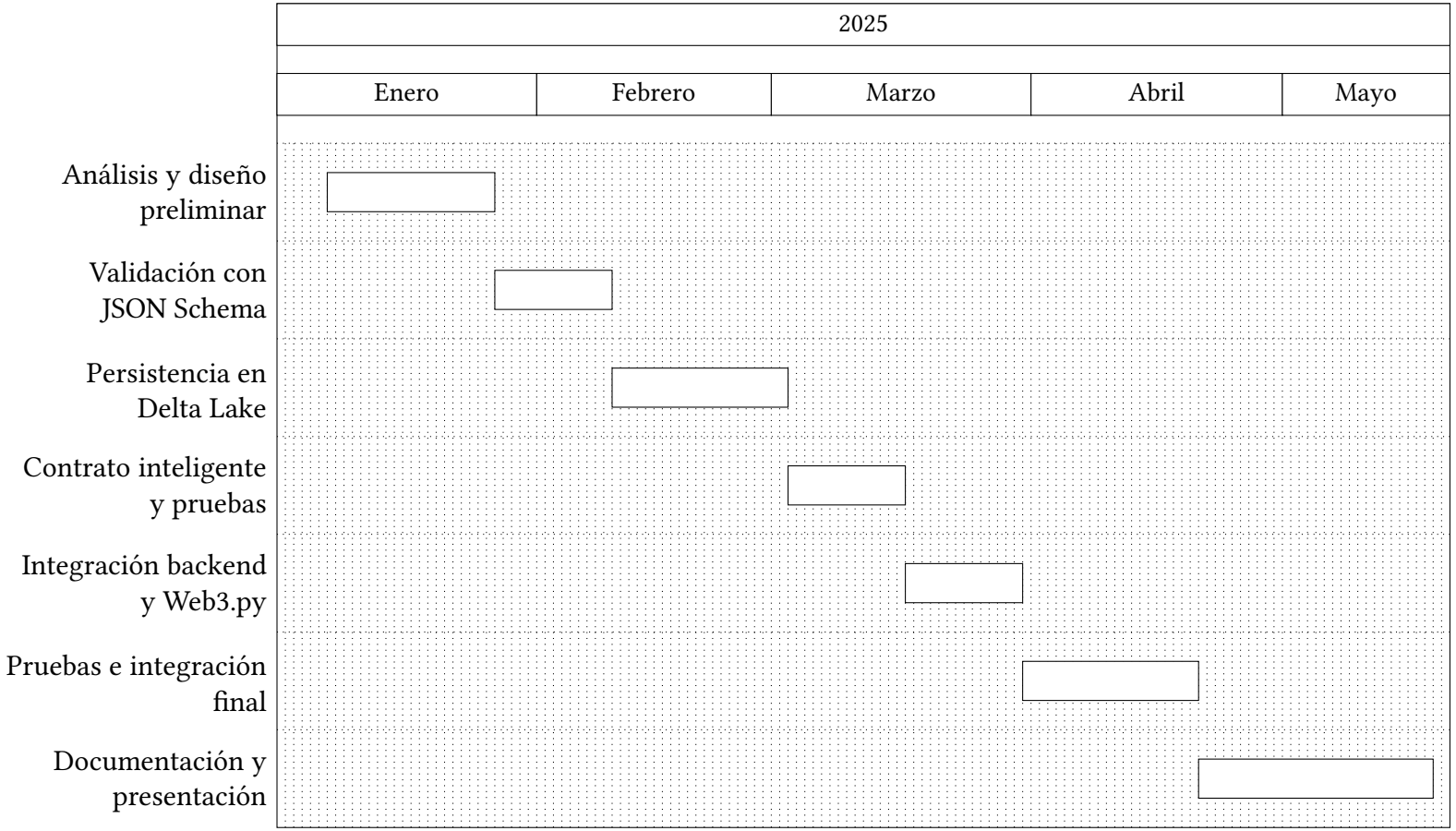

Figura 4.6: Diagrama de Gantt del proyecto

4.2.2. Estudio de viabilidad

4.2.2.1. Presupuesto

A partir de la planificación estimada y los recursos empleados, se ha calculado un presupuesto orientativo del proyecto. Se parte de un coste salarial anual total de 50.000 euros (incluyendo costes salariales y sociales), lo que, considerando una dedicación anual de 1750 horas, supone un coste horario de aproximadamente 28,57 €/hora.c

El desarrollo del proyecto, sin la documentación y la presentación, ha requerido una dedicación aproximada de 300 horas, lo que se traduce en un coste laboral directo de:

$$300\,\text{h} \times 28{,}57\,€/\text{h} = 8.571\,€$$

A ello se suma el coste proporcional del equipo informático utilizado, valorado en 1200 euros y amortizado en tres años. Dado que el periodo de trabajo ha sido de unos seis meses, se estima un coste de 200 euros:

$$(1200\,€ \div 3\,\text{años}) \times 0{,}5\,\text{años} = 200\,€$$

El coste total sin beneficio asciende, por tanto, a 8.771 €. Considerando un beneficio industrial del 7 %, se obtiene:

$$8.771\,€ \times 1{,}07 = 9.384{,}97\,€$$

Por tanto, el precio estimado de venta del trabajo, incluyendo margen empresarial, asciende a 9.384,97 euros. Desde un punto de vista más comercial, se podría redondear el precio de venta a 10.000 euros y jugar con márgenes para realizar descuentos a clientes.

En la tabla 4.8 se presenta un desglose del presupuesto estimado del proyecto como resumen de los cálculos realizados anteriormente.

Concepto	**Coste (€)**
Coste horario estimado	28,57
Horas de trabajo estimadas	300
Coste laboral (28,57 € × 300 h)	**8.571**
Coste de uso del equipo (6 meses)	200,00
Subtotal (sin beneficio)	**8.771**
Beneficio industrial (7 %)	614,00
Precio final del proyecto	**9.384,97**

Tabla 4.8: Presupuesto estimado del proyecto

4.2.2.2. Viabilidad económica

El proyecto se ha desarrollado íntegramente utilizando herramientas de código abierto y gratuitas, lo que reduce prácticamente a cero los costes económicos en tecnología. Las tecnologías empleadas han sido:

- Python, PySpark, Poetry
- Delta Lake y Parquet
- Docker, Docker Compose, MinIO
- Solidity, Hardhat, Ganache, Web3.py
- GitHub, Visual Studio Code

Los únicos recursos utilizados han sido un equipo personal con entorno MacOS y una conexión a Internet. Esto hace que el proyecto sea económicamente viable tanto en un contexto académico como para posibles desarrollos futuros en pequeña escala.

El proyecto se ha podido realizar con una blockchain local y, por lo tanto, no se ha realizado ningún gasto asociado a la compra de criptomonedas para poner en marcha el proyecto en una red blockchain Ethereum de producción.

4.2.2.3. Viabilidad legal

El sistema desarrollado no gestiona datos personales ni sensibles, por lo que no está sujeto a restricciones legales directas en materia de protección de datos (como el RGPD). En caso de extender el sistema a un entorno real con datos de usuarios, sería necesario llevar a cabo un análisis de impacto conforme a la normativa vigente.

Las tecnologías empleadas (software libre y de código abierto) permiten su uso, modificación y distribución en entornos académicos y comerciales, conforme a sus respectivas licencias. Además, el uso de blockchain y almacenamiento local evita el envío de información a terceros, reduciendo riesgos de cumplimiento normativo.

4.2.3. Análisis DAFO

En la Tabla 4.9 se presenta un análisis DAFO del proyecto, con el objetivo de evaluar sus puntos fuertes y débiles, así como los factores externos que podrían representar una oportunidad o una amenaza en su evolución hacia un entorno real.

Fortalezas (F)	**Debilidades (D)**
– Arquitectura modular y escalable – Tecnologías open source – Registro inmutable con blockchain – Validación automática de datos IoT – Reproducibilidad con Docker y Makefile	– Alta curva de aprendizaje en blockchain – Sin pruebas reales con dispositivos IoT – Falta de interfaz de usuario – Documentación limitada para no técnicos
Oportunidades (O)	**Amenazas (A)**
– Aplicación a trazabilidad real (residuos, alimentos...) – Integración con plataformas cloud – Análisis predictivo futuro – Potencial para investigación académica	– Complejidad técnica para adopción – Obsolescencia tecnológica – Desconfianza hacia blockchain – Riesgos legales con datos personales

Tabla 4.9: Análisis DAFO del proyecto

Como se puede observar en la tabla 4.9, entre las fortalezas, destaca el diseño modular y escalable de la arquitectura, el uso de tecnologías ampliamente

adoptadas en la industria (como Docker, PySpark o Ethereum), y la incorporación de mecanismos avanzados como la validación estructural automática de datos IoT y la verificación de integridad basada en blockchain. Estas características, junto con la reproducibilidad del entorno gracias a herramientas como Docker y Makefile, refuerzan la robustez técnica del sistema.

No obstante, también se reconocen debilidades importantes, como la ausencia de pruebas reales con dispositivos IoT, una interfaz de usuario limitada, y la elevada complejidad técnica asociada a conceptos como los árboles de Merkle o el despliegue de contratos inteligentes. En cuanto al entorno externo, el proyecto presenta oportunidades claras de aplicación en trazabilidad de datos en sectores como la gestión de residuos, la industria alimentaria o el transporte. También se abre la posibilidad de extender la solución con módulos de análisis predictivo o integraciones con plataformas cloud. Sin embargo, existen amenazas que deben tenerse en cuenta, como la rápida evolución tecnológica, la posible resistencia a soluciones blockchain por parte de ciertos sectores, y las implicaciones legales si se incorporan datos personales en escenarios reales.

Este capítulo ha detallado los requisitos funcionales y no funcionales del sistema, así como los principales casos de uso y la planificación temporal del proyecto. Con esta información como base, el siguiente capítulo aborda el diseño de la arquitectura y las decisiones estructurales adoptadas.

Diseño

5.1. Diseño de la Arquitectura del Sistema

La arquitectura del sistema propuesta ha sido diseñada con el objetivo de garantizar la trazabilidad, integridad y disponibilidad de los datos generados por dispositivos IoT en entornos distribuidos. Para ello, se ha optado por una arquitectura modular en la que cada componente cumple una función claramente definida, lo que facilita su desarrollo, prueba y despliegue.

El sistema se estructura en cuatro grandes bloques: captura y validación de datos, almacenamiento en Data Lake, registro de huellas digitales en blockchain y visualización de datos y validación en blockchain. La elección de una arquitectura actual está desacoplada, usando para ello llamadas HTTP entre los servicios. Como trabajo futuro de mejora, esta arquitectura puede pasar a ser basada en eventos, permitiendo una mayor escalabilidad y flexibilidad ante cambios o ampliaciones futuras.

5.1.1. Componentes principales

- **Dispositivos IoT**: sensores físicos que recolectan datos (como temperatura, posición GPS, humedad, etc.) y los publican mediante el protocolo MQTT.
- **Broker MQTT**: punto de entrada de los datos, encargado de recibir los mensajes publicados por los dispositivos y redirigirlos al sistema.
- **Servicio de procesamiento**: desarrollado en Python, mediante una suscripción al broker MQTT, se encarga de:

  - Validar los mensajes conforme a un esquema JSON.
  - Calcular una huella digital criptográfica de los datos (hash SHA-256).
  - Enviar los datos válidos al sistema de almacenamiento (Delta Lake).
  - Registrar el hash y metadatos en un contrato inteligente desplegado sobre la blockchain.

- **Sistema de almacenamiento**: se utiliza Delta Lake sobre MinIO (S3-compatible) para almacenar los datos de forma estructurada y versionada, facilitando su análisis futuro y garantizando su inmutabilidad.

- **Blockchain**: En este proyecto, Ethereum se emplea como sistema de registro descentralizado, donde se almacenan los hashes de los datos, el identificador del dispositivo y la marca temporal, garantizando así la trazabilidad y la no repudiación.

- **Frontend**: Interfaz de usuario desarrollada para visualizar el estado de los datos recibidos, consultar la trazabilidad y lanzar acciones de verificación. Sirve como punto de entrada para la interacción con el sistema de forma accesible y visual.

- **Backend API**: Servicio responsable de gestionar la lógica de negocio y exponer los endpoints necesarios para la comunicación entre el frontend, la base de datos y la blockchain. Esta se ha desarrollado en Python creando una API GraphQL.

5.1.2. Justificación de la arquitectura

La decisión de emplear una arquitectura basada en microcomponentes comunicados por HTTP o suscripción y con almacenamiento desacoplado responde a varias motivaciones:

- **Escalabilidad horizontal**: los distintos servicios pueden replicarse y escalar de forma independiente si aumenta la carga de trabajo.

- **Trazabilidad robusta**: separar el almacenamiento del contenido completo (en Delta Lake) del almacenamiento del resumen (en blockchain) permite verificar la integridad sin congestionar la red.

- **Facilidad de auditoría**: al estar cada componente claramente definido y al registrar huellas digitales en blockchain, es sencillo reconstruir el flujo de datos en caso de auditoría.

- **Adaptabilidad tecnológica**: el uso de estándares abiertos como MQTT, JSON y HTTP facilita la interoperabilidad con otros sistemas o dispositivos.

5.1.3. Diagrama de arquitectura

A continuación se muestra el diagrama general del sistema, donde se visualiza los diferentes componentes y sus conexiones:

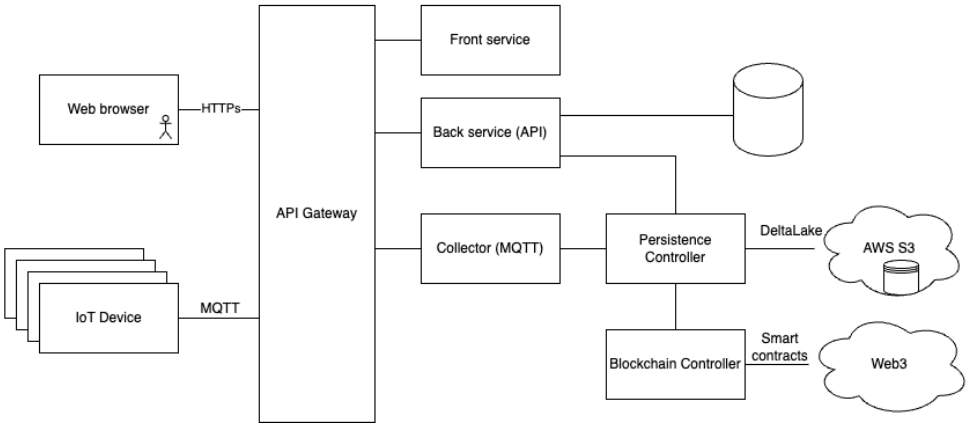

Figura 5.7: Arquitectura general del sistema propuesto

5.2. Diseño de la Aplicación Backend

El backend del sistema ha sido diseñado con el objetivo de gestionar de forma eficiente el flujo de datos provenientes de los dispositivos IoT, garantizando su validación, trazabilidad y almacenamiento seguro. La aplicación se ha desarrollado en `Python`, siguiendo principios de modularidad, responsabilidad única y separación de capas.

5.2.1. Estructura general

La aplicación se organiza en módulos independientes, cada uno de los cuales encapsula una funcionalidad concreta del sistema. Esta separación permite facilitar tanto las pruebas unitarias como el mantenimiento evolutivo del software. Los principales módulos son los siguientes:

- **Recepción de datos**: encargado de suscribirse a un *broker* MQTT y recibir los mensajes publicados por los dispositivos. Se utiliza la librería `paho-mqtt` para gestionar las conexiones y el flujo de mensajes.
- **Validación**: cada mensaje recibido se valida mediante un esquema `JSON Schema`, asegurando que la estructura y los tipos de datos sean correctos antes de ser procesados.
- **Hashing**: se calcula una huella digital criptográfica (`SHA-256`) sobre el contenido del mensaje validado, que se usará como identificador de integridad.
- **Almacenamiento**: los datos validados se almacenan en un Data Lake basado en `Delta Lake`, alojado sobre un sistema compatible con S3 (MinIO), permitiendo almacenamiento distribuido, versionado y consultas eficientes.
- **Registro en Blockchain**: se utiliza la librería `web3.py` para interactuar con un contrato inteligente en Ethereum. Se registra un identificador único (`UUID`), el hash de los datos, el identificador del dispositivo y una marca temporal.

5.2.2. Flujo de ejecución

El flujo básico seguido por la aplicación puede resumirse en los siguientes pasos:

1. Suscripción al *topic* correspondiente del *broker* MQTT.
2. Recepción de un mensaje con datos IoT.
3. Validación del mensaje contra el esquema definido.
4. Generación del hash criptográfico mediante la generación de un árbol de Merkle.
5. Persistencia del mensaje en el Data Lake junto a los metadatos.
6. Registro del hash y metadatos en el contrato inteligente.

Este flujo garantiza que sólo los datos válidos y correctamente estructurados se almacenen y se tracen en blockchain, proporcionando integridad desde la entrada hasta el registro descentralizado.

5.2.3. Consideraciones adicionales

Se han implementado mecanismos de gestión de errores y reintentos automáticos para operaciones críticas, como la conexión al *broker* MQTT o el envío de

transacciones a la blockchain. Además, se ha integrado un sistema de `logging` estructurado para facilitar el monitoreo y la trazabilidad de eventos.

La arquitectura modular también permite procesar tanto mensajes individuales como lotes de datos, facilitando futuras integraciones con sistemas de procesamiento distribuido (*batch* o *streaming*).

5.3. Diseño de la Interfaz de Usuario (Frontend)

La interfaz de usuario ha sido desarrollada utilizando el framework `Angular 2+`, con el objetivo de ofrecer una visualización clara, estructurada y usable de los datos procesados por el sistema. Se ha optado por una aplicación de tipo *Single Page Application* (SPA), que permite una experiencia fluida y sin recargas completas del navegador, mejorando la interactividad y reduciendo los tiempos de espera del usuario.

5.3.1. Objetivos del diseño

Los objetivos principales de la interfaz de usuario son:

- Facilitar la consulta de los datos recopilados por los dispositivos IoT, ya validados y almacenados.
- Permitir la verificación de la integridad de los datos mediante la comparación de la huella digital almacenada en la blockchain.
- Ofrecer una navegación sencilla, centrada en los flujos principales de uso, con un diseño responsive compatible con distintos tamaños de pantalla.

5.3.2. Estructura de la aplicación

La aplicación Angular se ha estructurado siguiendo las buenas prácticas del framework, separando componentes, servicios y modelos. Los principales elementos de la interfaz son:

- **Vista principal**: muestra un mensaje de bienvenida e indica que se navegue al apartado Datos en el menú.
- **Vista de datos**: página en la que se puede seleccionar el dispositivo y el rango de fechas de los datos a visualizar.

- **Verificación de integridad**: funcionalidad que permite consultar la blockchain para verificar si el hash de un conjunto de datos está registrado, garantizando así su autenticidad.
- **Servicios**: se han desarrollado servicios en Angular para consumir las APIs GraphQL expuestas por el backend y para interactuar con el nodo Ethereum a través de `web3.js`, adaptado al navegador.

5.3.3. Diseño visual y usabilidad

Para el diseño visual se ha utilizado `Angular Material`, que proporciona componentes accesibles y con un estilo moderno y uniforme. Se ha priorizado la simplicidad, eliminando elementos visuales innecesarios y centrando la atención del usuario en la información relevante.

El diseño responsive permite utilizar la interfaz desde dispositivos móviles y tablets, facilitando su acceso en entornos industriales o de campo. Asimismo, se ha validado la accesibilidad básica del sistema, siguiendo pautas como el contraste de colores y la navegación por teclado.

5.3.4. Wireframes

A continuación se presentan algunos bocetos de las principales pantallas de la interfaz, que sirvieron de guía durante el desarrollo:

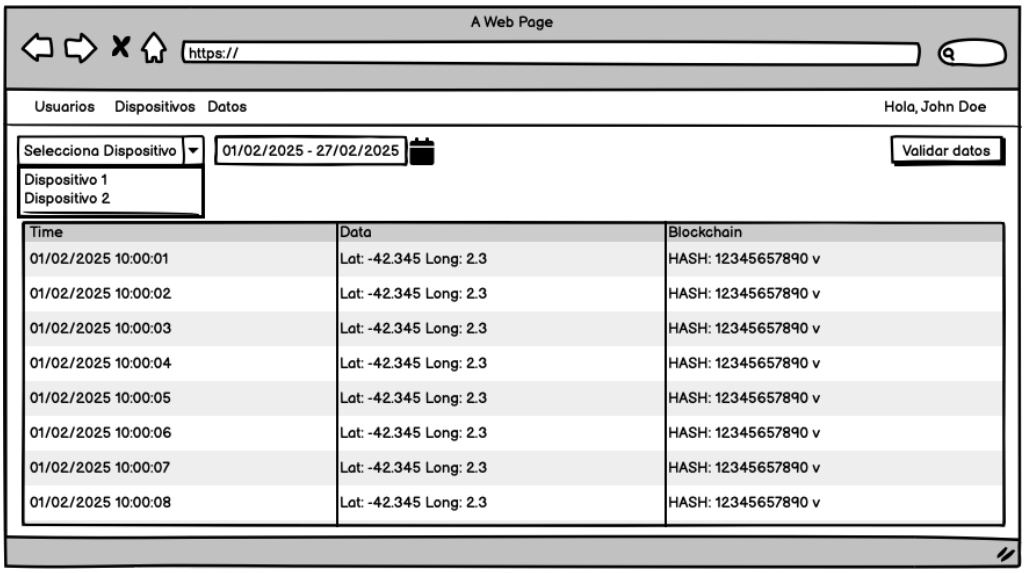

Figura 5.8: Wireframe: visualización de datos

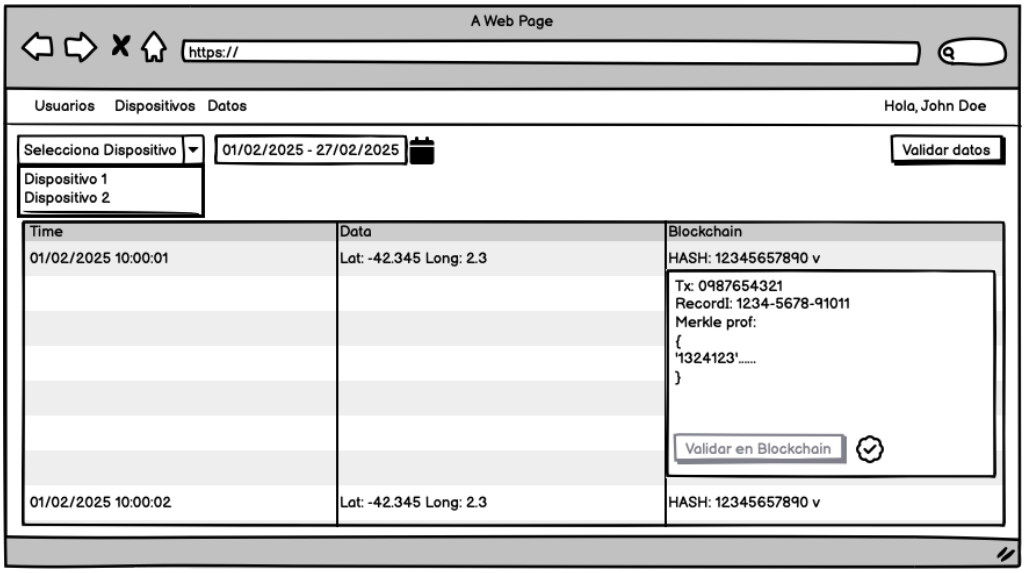

Figura 5.9: Wireframe: detalle de validación del dato en blockchain

Aquí se ha descrito la arquitectura modular del sistema, así como las relaciones entre sus componentes. Este diseño es la base sobre la que se ha construido la implementación del sistema, la cual se detalla en el próximo capítulo.

Implementación

6.1. Implementación Técnica

6.1.1. Implementación del procesamiento de datos

El procesamiento de los datos recibidos desde los dispositivos IoT constituye una de las piezas fundamentales del sistema. Esta funcionalidad ha sido desarrollada en Python, haciendo uso de librerías especializadas para la gestión de flujos de datos, validación estructural, generación de hashes criptográficos y escritura en sistemas de almacenamiento distribuidos.

Recepción de mensajes MQTT

La aplicación se conecta a un *broker* MQTT, al que los dispositivos publican mensajes con los datos recogidos. Para gestionar esta comunicación, se ha utilizado la librería `paho-mqtt`, que permite suscribirse a uno o varios *topics* y definir funciones de *callback* para cada mensaje recibido.

El flujo básico consiste en:

1. Suscripción al *topic* configurado.
2. Ejecución automática de la función de procesamiento al recibir un nuevo mensaje.
3. Decodificación del mensaje en formato JSON.

Validación estructural de los datos

Una vez recibido y decodificado el mensaje, se lleva a cabo una validación de su estructura utilizando esquemas `JSON Schema`. Esto permite asegurar que el mensaje cumple con el formato esperado antes de continuar con su procesamiento. Entre las validaciones realizadas se incluyen:

- Presencia obligatoria de campos como `timestamp`, `lat`, `long`, `temp`, etc.
- Tipos correctos para cada valor (números, cadenas, arreglos).
- Formatos y rangos válidos (por ejemplo, rangos de temperatura razonables).

En caso de que el mensaje no supere la validación, se descarta y se registra el error en el sistema de `logging` para su posterior análisis.

Generación de hash criptográfico

Una vez validado, se calcula una huella digital (*hash*) del contenido del mensaje. Para ello se utiliza el algoritmo `SHA-256`, que genera un resumen único e inmutable. Esta huella servirá como verificación de que el dato enviado por el dispositivo IoT no ha sido alterado.

Una vez varificado y validado el mensaje se procesan los datos de tal forma que se genera un arbol de Merkle con ellos. Esto generará un hash raiz y un conjunto de hashes llamados pruebas de Merkle que servirán para la validación del dato de forma individual. Las pruebas de Merkle se almacenan junto a cada dato a almacenar como un metadato y a un UUID. Por otro lado, El hash raíz se envía a la blockchain para su almacenamiento junto al UUID. Esta forma nos permite en el futuro validar que el dato forma parte de una hoja del arbol del Merkle cuya raíz corresponde al UUID almacecnado.

El proceso incluye:

- Serialización determinista del mensaje (orden de claves consistente).
- Codificación en UTF-8 y aplicación de `hashlib.sha256`.

Almacenamiento en Delta Lake

El mensaje original validado se almacena en formato Parquet utilizando Delta Lake como sistema de gestión de datos. Para ello, se emplea `PySpark`, permitiendo:

- Escribir en un Data Lake basado en MinIO (compatible con S3).
- Añadir particiones por fecha u otros criterios relevantes, como el identificador del dispositivo.
- Garantizar versionado, transacciones ACID y consultas eficientes.

Cada lote de datos procesados se escribe como una nueva versión del conjunto de datos, lo que permite mantener un histórico completo sin sobrescrituras.

Registro en la blockchain

Finalmente, se realiza una llamada al contrato inteligente desplegado en Ethereum mediante la librería `web3.py`. En esta transacción se incluyen:

- El identificador del dispositivo.
- El hash generado, es decir, la raíz del árbol de Merkle.
- Un identificador único de registro (`recordId`).
- El `timestamp`.

El identificador generado se recupera desde el recibo de la transacción, permitiendo su almacenamiento externo para futuras consultas. Este proceso asegura la trazabilidad y la inmutabilidad del dato, sin almacenar información confidencial directamente en la blockchain. Esto se debe a que, aun teniendo el identificador de la transacción, blockchain no permite acceder directamente al dato.

6.1.2. Implementación del contrato inteligente

Para garantizar la integridad y trazabilidad de los datos recolectados por los dispositivos IoT, se ha desarrollado un contrato inteligente (*smart contract*) en el lenguaje `Solidity`, desplegado sobre una red Ethereum compatible. Este

contrato actúa como un registro inmutable que almacena referencias a los datos procesados, en forma de huellas digitales (hashes), junto con información contextual relevante.

Estructura del contrato

El contrato, denominado `IoTDataRegistry`, permite almacenar y consultar los registros asociados a cada dispositivo. La estructura principal es la siguiente:

- **Estructura `DataRecord`**: contiene el hash del dato (`dataHash`) y la marca temporal (`timestamp`).
- **Mapa `records`**: relaciona cada identificador de dispositivo (`deviceId`) con una lista de registros de tipo `DataRecord`.
- **Mapa `recordIds`**: permite asociar un identificador único (`recordId`) con un registro específico de un dispositivo.

La lógica de almacenamiento se ha optimizado para garantizar eficiencia en el uso de gas y facilitar la recuperación de datos por parte del cliente.

Funciones principales

Las funciones clave implementadas en el contrato son:

- `storeData(string memory deviceId, string memory dataHash, uint256 timestamp)`: almacena un nuevo registro para un dispositivo, genera internamente un `recordId` único y lo asocia al dispositivo. Esta función emite un evento con el identificador generado.
- `getData(string memory deviceId)`: devuelve todos los registros asociados a un `deviceId`, permitiendo consultar su historial completo de huellas.
- `getDataWithRecordId(string memory deviceId, string memory recordId)`: permite recuperar únicamente el registro que coincide con el `recordId` indicado, útil para trazabilidad precisa desde el exterior del sistema.

Despliegue y pruebas

El contrato ha sido desarrollado y probado utilizando el entorno `Hardhat`, lo que permite compilar, desplegar y testear el contrato localmente o en redes públicas de pruebas como Sepolia. Durante el proceso se han realizado pruebas unitarias para verificar el correcto comportamiento de cada función, asegurando:

- El correcto almacenamiento de registros.
- La recuperación precisa de datos mediante `deviceId` o `recordId`.
- La generación y persistencia del identificador único (`recordId`) dentro de la blockchain.

Además, se ha integrado `web3.py` en el backend del sistema, lo que permite invocar las funciones del contrato desde Python de forma programática, firmando las transacciones con una clave privada local o de entorno seguro. El identificador de registro (`recordId`) se recupera desde el recibo de transacción y se conserva junto con el dato completo para futuras comprobaciones.

Optimización de costes

Se han aplicado varias técnicas para minimizar el coste de gas de las operaciones:

- Uso de tipos de datos compactos y estructuras planas.
- Separación de datos completos (almacenados en el Data Lake) y metadatos (registrados en la blockchain).
- Uso de eventos para obtener información desde el exterior sin necesidad de llamadas adicionales a la cadena.
- Uso de árboles de Merkle para compactar las huellas digitales de los datos y facilitar la verificación de integridad ante posibles alteraciones, minimizando el espacio requerido en blockchain.

Esta aproximación permite escalar el sistema sin incurrir en costes elevados por el uso de la red Ethereum, al mismo tiempo que se garantiza la trazabilidad e integridad de los datos.

6.1.3. Implementación del frontend

La interfaz de usuario del sistema ha sido desarrollada utilizando `Angular 2+`, un framework de desarrollo frontend basado en TypeScript que permite crear aplicaciones web reactivas, modulares y de fácil mantenimiento. La aplicación tiene como objetivo principal facilitar el acceso a los datos procesados y validados, así como permitir la verificación de su integridad mediante la interacción con la blockchain.

Estructura de la aplicación

El frontend se ha organizado en base a componentes reutilizables y servicios centralizados. La estructura sigue las convenciones de Angular y está compuesta principalmente por:

- **Componentes**: encargados de representar visualmente las vistas de la aplicación. Entre los principales se encuentran:
  - `DeviceListComponent`: muestra un listado de dispositivos IoT registrados.
  - `DeviceDetailComponent`: presenta los datos individuales enviados por un dispositivo concreto, junto con sus hashes y timestamps.
  - `VerificationComponent`: permite al usuario verificar si un hash está registrado en la blockchain y visualizar los detalles asociados.
- **Servicios**: gestionan la lógica de negocio y la comunicación con APIs externas. Los más relevantes son:
  - `DataService`: se comunica con el backend para obtener los datos almacenados en el Data Lake.
  - `BlockchainService`: interactúa con la red Ethereum (a través de `web3.js`) para consultar el contrato inteligente.

Integración con el backend y la blockchain

El frontend consume dos tipos de fuentes de datos:

1. API GraphQL desarrollada en Python, que expone los datos recolectados y validados, permitiendo su consulta mediante llamadas HTTP.
2. Contrato inteligente desplegado en Ethereum, al que se accede desde el navegador mediante `web3.js` y el proveedor inyectado por `MetaMask`.

Esta doble fuente permite comparar los datos almacenados con los registros en blockchain, validando así su integridad de forma transparente para el usuario.

Diseño visual

Se ha utilizado `Angular Material` como biblioteca de componentes UI para garantizar una apariencia moderna, accesible y coherente. Además, se ha aplicado un diseño *responsive*, lo que permite utilizar la aplicación desde diferentes dispositivos (PC, tablet o móvil) sin pérdida de funcionalidad.

Las vistas se han diseñado priorizando la simplicidad y claridad, mostrando la información relevante en primer plano y reduciendo al mínimo las acciones necesarias para acceder a los datos o verificar su validez.

Navegación y flujo de uso

La navegación se estructura en rutas claramente diferenciadas:

- `/devices`: listado de dispositivos registrados.
- `/data`: para la visualización de los datos.
- `/users`: para la visualización de los usuarios.

El flujo de uso está pensado para que un usuario pueda, en pocos pasos, consultar los datos de un dispositivo, visualizarlos en detalle y verificar su autenticidad, todo ello sin necesidad de conocimientos técnicos avanzados.

Este capítulo ha detallado el proceso de implementación técnica del sistema, abordando de forma modular la lógica de procesamiento de datos, el contrato inteligente desarrollado en Solidity y la interfaz web construida con Angular. Se ha documentado cómo cada componente interactúa dentro de la arquitectura propuesta, asegurando la integración entre tecnologías IoT, Big Data y blockchain. En el siguiente capítulo se describe la estrategia de validación seguida y las pruebas realizadas para verificar el correcto funcionamiento del sistema.

Pruebas

7.1. Pruebas Realizadas

Durante el desarrollo del sistema se han llevado a cabo diversas pruebas con el fin de validar tanto su correcto funcionamiento como el cumplimiento de los objetivos planteados. Estas pruebas permiten verificar la robustez de los distintos módulos, detectar posibles errores en fases tempranas y asegurar la integridad de los datos desde su recepción hasta su verificación en blockchain.

El sistema, al estar compuesto por múltiples componentes distribuidos (servicios de backend, frontend, worker de procesamiento, almacenamiento en Data Lake y contratos inteligentes), requiere una estrategia de pruebas completa que abarque las distintas capas de la arquitectura. En esta sección se detallan los tipos de pruebas realizados, la metodología seguida y los resultados obtenidos.

7.1.1. Estrategia de pruebas

La estrategia de pruebas adoptada tiene como objetivo garantizar la correcta funcionalidad, fiabilidad e integridad del sistema desarrollado. Para ello, se ha seguido un enfoque progresivo que incluye pruebas unitarias, pruebas de integración y pruebas funcionales, abarcando tanto el backend como el frontend, así como la interacción con la blockchain.

Tipos de pruebas

Se han definido los siguientes tipos de pruebas, cada uno orientado a validar distintos niveles del sistema:

- **Pruebas unitarias**: verifican el comportamiento de funciones y módulos individuales, como la validación de mensajes, el cálculo de hashes o la lógica del contrato inteligente.
- **Pruebas de integración**: aseguran que los distintos componentes del sistema (por ejemplo, recepción de datos, almacenamiento en Delta Lake y registro en blockchain) funcionan correctamente en conjunto.
- **Pruebas funcionales**: simulan casos de uso reales para validar que el sistema cumple con los requisitos funcionales definidos.
- **Pruebas end-to-end**: ejecutadas desde el frontend, verifican que los datos se visualizan correctamente, que las verificaciones contra la blockchain funcionan como se espera y que el usuario puede completar flujos completos sin errores.

Herramientas utilizadas

Para la ejecución y automatización de las pruebas se han utilizado las siguientes herramientas:

- `pytest`: para las pruebas unitarias e integración del backend Python.
- `Hardhat`: para pruebas automatizadas del contrato inteligente en entornos de test.
- `web3.py` y `web3.js`: para invocar y verificar funciones del contrato desde backend y frontend, respectivamente.
- `Jasmine` y `Karma`: para pruebas unitarias en el frontend Angular.
- `MetaMask`: como proveedor de Web3 para pruebas desde navegador con interacción directa con la blockchain.

Entorno de pruebas

Durante el desarrollo se ha utilizado un entorno local compuesto por contenedores Docker, que permite simular todos los componentes necesarios del sistema:

- **MinIO**: como sistema de almacenamiento compatible con la API de S3.

- **Backend y frontend**: desplegados en contenedores separados para facilitar la modularidad y las pruebas independientes.
- **Red Ethereum local**: utilizando Hardhat o Ganache como nodos de desarrollo para el despliegue y prueba de contratos inteligentes.

Este entorno ha permitido realizar pruebas rápidas, reproducibles y sin costes, facilitando el desarrollo iterativo y la validación del sistema de forma completa en local.

Cobertura y enfoque incremental

La estrategia de pruebas ha sido incremental, validando progresivamente cada componente conforme se completaba su desarrollo. Se ha buscado alcanzar una alta cobertura de código en el backend, así como asegurar la robustez del contrato inteligente ante entradas maliciosas o inesperadas. Las pruebas han sido documentadas para permitir su reproducción, y se han automatizado aquellas susceptibles de ser ejecutadas en CI/CD en el futuro.

7.1.2. Pruebas del backend

El backend del sistema, desarrollado en Python, ha sido sometido a un conjunto de pruebas orientadas a validar tanto la lógica de procesamiento de datos como la interacción con los sistemas de almacenamiento y con la blockchain. Estas pruebas se han dividido en pruebas unitarias y pruebas de integración, empleando datos simulados representativos de los dispositivos IoT.

Validación de mensajes

Se han definido múltiples casos de prueba para verificar que el módulo de validación JSON Schema rechaza correctamente los mensajes mal formados. Se han comprobado:

- Rechazo de mensajes con campos ausentes o vacíos.
- Rechazo de tipos de datos incorrectos (por ejemplo, texto en lugar de números).
- Aceptación de mensajes válidos con valores dentro de los rangos esperados.

Cada caso ha sido probado mediante funciones de test utilizando `pytest`, garantizando que sólo los mensajes bien estructurados avanzan en el flujo del sistema.

Cálculo de hash

Se ha verificado que el algoritmo de hashing genera resultados consistentes y deterministas para entradas idénticas. También se ha comprobado que cualquier modificación, por mínima que sea, en los datos de entrada, produce un hash completamente distinto, cumpliendo así con las propiedades deseadas del algoritmo SHA-256.

Almacenamiento en Delta Lake

Para validar el almacenamiento, se han ejecutado pruebas que:

- Comprueban que los datos se escriben correctamente en formato Parquet.
- Verifican la creación de particiones por fecha y su correcta resolución mediante consultas.
- Evalúan la existencia de versiones anteriores de los datos mediante la funcionalidad de `time travel`.

Se ha empleado `PySpark` con consultas directas sobre los datos almacenados en MinIO para validar el contenido de los ficheros generados.

Manejo de errores y reintentos

Se han simulado fallos comunes como:

- Desconexión del broker MQTT.
- Fallo de escritura en el Data Lake.
- Error de conexión con el nodo blockchain.

El sistema responde ante estos fallos mediante mecanismos de reconexión, reintentos controlados y registro estructurado de errores. Estas pruebas aseguran que el backend puede recuperarse de errores transitorios sin pérdida de datos ni necesidad de intervención manual inmediata.

Invocación del contrato inteligente

Se han probado las funciones de interacción con el contrato mediante `web3.py`. En concreto, se ha verificado:

- El envío correcto de transacciones con datos válidos.
- La recepción del *receipt* y extracción del `recordId`.
- El comportamiento del sistema ante transacciones fallidas (por ejemplo, por falta de gas).

Todas las pruebas se han realizado inicialmente en una red local con HardHat o Ganache, utilizando ETH de prueba para evaluar el consumo real de gas en condiciones cercanas al entorno de producción.

Estas pruebas permiten verificar el cumplimiento de los requisitos RF1 (recolección de datos IoT), RF2 (validación de datos) y RF3 (almacenamiento de datos), al garantizar que los mensajes recibidos se validan estructuralmente mediante JSON Schema y se almacenan de forma persistente. También contribuyen al cumplimiento de los requisitos no funcionales RNF2 (fiabilidad del sistema frente a entradas inválidas) y RNF5 (compatibilidad con formatos estandarizados como JSON).

7.1.3. Pruebas del contrato inteligente

El contrato inteligente desarrollado en `Solidity` ha sido sometido a pruebas exhaustivas para validar su correcto funcionamiento, garantizar la integridad de los datos almacenados y asegurar un uso eficiente del gas. Las pruebas se han llevado a cabo utilizando el entorno de desarrollo `Hardhat`, que permite ejecutar test automatizados sobre una red Ethereum local simulada.

Pruebas unitarias

Se han diseñado pruebas unitarias para cada una de las funciones principales del contrato:

- `storeData`: se ha comprobado que permite almacenar registros correctamente, que emite el evento correspondiente con el `recordId` generado y que se puede invocar múltiples veces con distintos dispositivos.

- `getData`: se ha verificado que devuelve la lista de registros completa para un `deviceId` determinado, en el mismo orden en que fueron almacenados.
- `getDataWithRecordId`: se han probado casos positivos (el `recordId` existe y coincide) y casos negativos (no existe el registro o no pertenece al dispositivo consultado), evaluando que el contrato responde correctamente.

Estas pruebas se han implementado en JavaScript utilizando la API de `ethers.js`, incluida en Hardhat, permitiendo verificar tanto el estado interno del contrato como los eventos emitidos.

Pruebas de consistencia y validación

Además de las pruebas funcionales, se han llevado a cabo pruebas orientadas a asegurar la consistencia de los datos y la protección frente a entradas maliciosas. Entre ellas:

- Almacenamiento de registros con el mismo hash para dispositivos distintos: el contrato debe permitirlo, ya que el hash representa datos equivalentes generados en distintos contextos.
- Intentos de almacenar registros con campos vacíos o inválidos: se ha comprobado que la validación se realiza en la capa del backend, y que el contrato asume que los datos ya han sido preprocesados y validados.
- Detección de `recordId` duplicado: se ha verificado que el contrato genera internamente un identificador único para cada registro mediante una combinación de `keccak256`, lo que garantiza la unicidad sin intervención externa.

Medición de gas y eficiencia

Se ha prestado especial atención al consumo de gas de las funciones del contrato. Se han medido los costes de las operaciones de escritura y lectura con distintos tamaños de entrada y número de registros por dispositivo. Las pruebas han demostrado que:

- El coste de la función `storeData` se mantiene dentro de límites razonables, incluso con múltiples registros.

- Las funciones de consulta no generan consumo de gas al ser llamadas como `view`, permitiendo su uso libre desde el frontend o scripts de backend.
- La emisión de eventos con el `recordId` permite recuperar la información necesaria sin almacenar datos redundantes.

Las pruebas realizadas sobre el contrato inteligente permiten verificar los requisitos RF4 (registro de integridad en blockchain) y RF5 (verificación de integridad), ya que validan el correcto almacenamiento del hash raíz del árbol de Merkle y su recuperación posterior. Asimismo, satisfacen los requisitos no funcionales RNF3 (seguridad e inmutabilidad de los registros) y RNF4 (trazabilidad completa del ciclo de vida de los datos).

7.1.4. Pruebas de la interfaz de usuario

La interfaz de usuario desarrollada con `Angular 2+` ha sido sometida a pruebas orientadas a verificar la correcta visualización de los datos, la fluidez de la navegación y la interacción con el backend y la blockchain. Se han llevado a cabo pruebas tanto manuales como automatizadas, utilizando datos simulados y reales procedentes de la red de pruebas.

Pruebas funcionales

Se han ejecutado pruebas funcionales para validar los flujos principales del sistema desde el punto de vista del usuario final:

- Carga del listado de dispositivos registrados.
- Acceso al detalle de un dispositivo y visualización de los datos recogidos.
- Visualización de los hashes y timestamps asociados a cada entrada.
- Consulta de la existencia de un hash en la blockchain a través de la interfaz de verificación.
- Manejo de errores de red o fallos en la conexión con el backend o el nodo Web3.

Estas pruebas han permitido identificar y corregir errores de integración y validaciones incompletas en etapas tempranas del desarrollo.

Pruebas de interacción con la blockchain

Dado que la aplicación permite al usuario verificar la integridad de los datos consultando directamente a la blockchain, se han realizado pruebas específicas para este módulo. Se ha verificado que:

- La conexión con `MetaMask` se establece correctamente al cargar la página.
- El contrato inteligente se consulta adecuadamente mediante `web3.js`, y los datos devueltos coinciden con los registrados.
- Los errores de red o fallos en la carga de la blockchain son detectados y mostrados al usuario de forma clara.

Estas pruebas se han realizado exclusivamente en un entorno local, lo que ha permitido validar la interoperabilidad de la interfaz con una red Ethereum simulada, reproduciendo condiciones similares a un entorno real sin incurrir en costes ni depender de infraestructura externa. Este enfoque facilita una futura migración a una red pública o privada de Ethereum, ya que los contratos y la lógica de interacción están diseñados para ser compatibles con redes EVM sin requerir modificaciones sustanciales.

Pruebas de usabilidad y diseño responsive

Se ha evaluado la experiencia de usuario en distintos dispositivos y tamaños de pantalla. Gracias al uso de `Angular Material` y diseño adaptativo (*responsive*), la aplicación se adapta correctamente a:

- Escritorios con resoluciones altas y bajas.
- Tablets en orientación horizontal y vertical.
- Teléfonos móviles de diferentes tamaños.

Además, se han realizado pruebas con usuarios ajenos al desarrollo para evaluar la claridad de las vistas, la facilidad de navegación y la comprensibilidad de los datos presentados.

Pruebas unitarias en Angular

Se han desarrollado pruebas unitarias para los componentes y servicios de Angular mediante `Jasmine` y `Karma`. Estas pruebas aseguran:

- La correcta inicialización de los componentes principales.
- El funcionamiento esperado de los servicios HTTP al consumir la API REST.
- El correcto tratamiento de errores y estados de carga.

Estas pruebas se integran con el sistema de desarrollo continuo, permitiendo validar rápidamente cambios antes de realizar despliegues.

La interfaz ha sido probada en distintos escenarios de visualización y verificación de datos, satisfaciendo los requisitos RF6 (interfaz de usuario funcional y accesible) y RF5 (verificación de integridad desde el cliente). Estas pruebas también contribuyen al cumplimiento de los requisitos no funcionales RNF6 (usabilidad de la aplicación por usuarios sin conocimientos técnicos) y RNF1 (escalabilidad, al permitir la visualización de múltiples dispositivos).

7.1.5. Ejecución de las pruebas en CI/CD

Con el objetivo de garantizar la calidad y estabilidad del sistema, se ha configurado un flujo de integración continua (CI) mediante *GitHub Actions*, una plataforma que permite definir y automatizar tareas como la ejecución de pruebas, compilación o despliegue en respuesta a eventos sobre el repositorio.

En este proyecto, se ha definido un flujo de trabajo (*workflow*) que se activa automáticamente en los siguientes casos:

- Cuando se realiza un *push* sobre la rama principal o de desarrollo.
- Cuando se crea un *pull request*, permitiendo validar los cambios antes de su fusión.

Por ejemplo, el `workflow` para el código en Python, ejecuta los siguientes pasos:

1. Se instala el entorno de Python y las dependencias del proyecto definidas en `pyproject.toml`.

2. Se lanza la ejecución de las pruebas unitarias mediante `pytest`.
3. Se muestra el resultado del conjunto de pruebas, indicando si todas han sido superadas correctamente o si existen errores.

Esta automatización permite detectar errores de forma temprana, prevenir regresiones y facilitar la colaboración segura en el código fuente. Además, el resultado de las pruebas queda visible directamente en la interfaz de GitHub, tanto en la vista de confirmaciones (*commits*) como en las solicitudes de incorporación de cambios (*pull requests*).

En la Figura 7.10 se muestra un ejemplo del resultado de una ejecución correcta del flujo de pruebas sobre una *pull request* en Github.

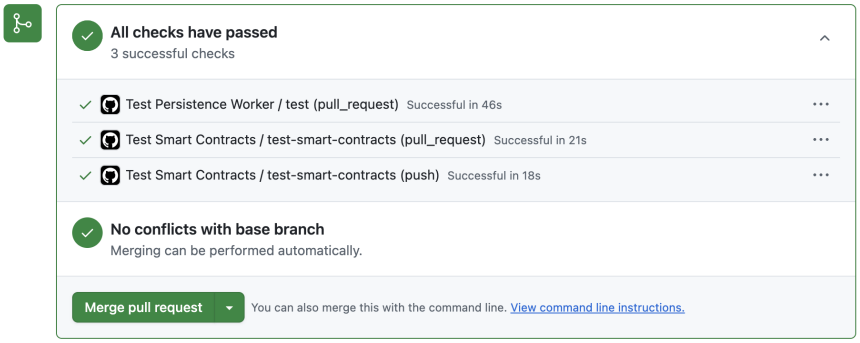

Figura 7.10: Resultado de la ejecución de pruebas en GitHub Actions

7.1.6. Resultados de las pruebas

Tras la ejecución de las distintas pruebas descritas en los apartados anteriores, se ha podido comprobar que el sistema desarrollado cumple satisfactoriamente con los objetivos funcionales y no funcionales definidos al inicio del proyecto. Las pruebas han permitido validar tanto la correcta integración entre componentes como la robustez del sistema frente a errores esperados.

Validación de los objetivos funcionales

Los resultados obtenidos permiten afirmar que los requisitos principales han sido alcanzados:

- Los datos enviados por los dispositivos IoT se reciben correctamente a través del broker MQTT, se validan y almacenan en el Data Lake de forma estructurada.

- Se calcula una huella digital de los datos y se registra en la blockchain pública, asegurando su trazabilidad e integridad.
- La interfaz de usuario permite consultar los datos recolectados y verificar su existencia y consistencia en la blockchain de forma intuitiva.
- El contrato inteligente permite almacenar y recuperar datos de forma eficiente, incluyendo la búsqueda por identificador único de registro (`recordId`).

Cobertura de pruebas y estabilidad del sistema

Las pruebas unitarias alcanzan una alta cobertura de código en el backend, validando los módulos de recepción, validación, hash y escritura en el Data Lake. Asimismo, las pruebas automatizadas del contrato inteligente y las pruebas funcionales del frontend confirman la correcta operatividad de las interfaces y la lógica del sistema.

Durante las pruebas de integración realizadas en el entorno local, el sistema ha demostrado una estabilidad adecuada ante condiciones adversas, como la pérdida de conexión con servicios externos o la introducción de datos inválidos, recuperándose de forma automática sin necesidad de intervención manual.

Aspectos detectados y posibles mejoras

Durante la fase de pruebas también se han identificado algunos aspectos susceptibles de mejora en futuras iteraciones del sistema:

- **Optimización del consumo de gas**: aunque el contrato funciona correctamente, su eficiencia podría mejorarse mediante el uso de estructuras más compactas o almacenamiento más directo.
- **Gestión de errores más informativa en frontend**: en algunos casos de fallo en la conexión blockchain, los mensajes al usuario podrían ser más específicos y orientativos.
- **Automatización del despliegue**: la integración de herramientas de CI/CD permitiría acelerar los ciclos de prueba y despliegue tanto para el backend como para el frontend y el contrato.
- **Pruebas de rendimiento a gran escala**: sería conveniente evaluar el sistema con un volumen elevado de dispositivos y datos para validar su comportamiento en escenarios de producción real.

Conclusión

En conjunto, los resultados obtenidos confirman que la solución propuesta es técnicamente viable, funcional y robusta. Las pruebas realizadas han permitido no sólo validar los objetivos alcanzados, sino también sentar las bases para futuras mejoras que refuercen la escalabilidad, eficiencia y experiencia de usuario del sistema.

A la vista de los resultados obtenidos, se puede afirmar que el sistema cumple con todos los requisitos funcionales definidos (RF1–RF6), cubriendo desde la captura y validación de datos IoT hasta su trazabilidad en blockchain y consulta desde la interfaz. Del mismo modo, se satisfacen los principales requisitos no funcionales (RNF1–RNF6), relacionados con la fiabilidad, seguridad, trazabilidad, compatibilidad y usabilidad del sistema.

Para finalizar, en la tabla 7.10 se presenta un resumen de la relación entre las pruebas realizadas y los requisitos funcionales y no funcionales validados. Esta trazabilidad permite comprobar de forma clara que todos los objetivos planteados en el análisis han sido correctamente cubiertos mediante el sistema desarrollado.

Prueba realizada	**Requisitos funcionales validados**	**Requisitos no funcionales validados**
Validación de datos IoT en el backend	RF1, RF2, RF3	RNF2, RNF5
Registro y verificación en blockchain	RF4, RF5	RNF3, RNF4
Consulta y verificación desde la interfaz web	RF5, RF6	RNF1, RNF6
Pruebas de integración y flujo completo de datos	RF1–RF6	RNF1, RNF2, RNF4

Tabla 7.10: Relación entre pruebas realizadas y requisitos validados

A lo largo de este capítulo se ha presentado la estrategia de pruebas utilizada para verificar la funcionalidad y robustez del sistema. Se han descrito las pruebas unitarias, funcionales e integradas aplicadas a los distintos módulos, así como la automatización de pruebas mediante GitHub Actions. Los resultados obtenidos confirman el cumplimiento de los requisitos definidos. En el próximo capítulo

se extraen las conclusiones del trabajo realizado y se plantean posibles líneas de mejora y desarrollo futuro.

Conclusiones y Líneas de trabajo futuras

8.1. Conclusiones

El trabajo realizado ha abordado con éxito el diseño e implementación de una arquitectura segura y trazable para el almacenamiento de datos IoT. Se han cumplido los objetivos establecidos, desarrollando una solución técnicamente robusta que integra tecnologías modernas como Delta Lake, MQTT y Ethereum blockchain.

A modo de resumen, los principales logros alcanzados durante el desarrollo del proyecto son los siguientes:

- Se ha diseñado e implementado una arquitectura modular, escalable y desacoplada.
- Se ha desarrollado un sistema de recolección de datos IoT basado en MQTT y validación con JSON Schema.
- Se ha integrado un sistema de almacenamiento escalable usando Delta Lake sobre MinIO.
- Se ha implementado un contrato inteligente en Ethereum para registrar hashes de datos y garantizar su integridad.
- Se ha desarrollado una interfaz web funcional para consultar y verificar la consistencia de los datos almacenados.

- Se ha evaluado la solución en un entorno local mediante pruebas unitarias, de integración y funcionales.
- Se ha automatizado la ejecución de pruebas en flujos CI/CD con GitHub Actions.
- Se ha documentado todo el sistema, incluyendo aspectos técnicos y de usuario.

La arquitectura propuesta demuestra ser viable desde el punto de vista técnico, combinando fiabilidad, trazabilidad y descentralización. Además, el sistema está preparado para escalar horizontalmente y adaptarse a futuros requisitos sin modificaciones estructurales profundas.

El sistema resultante ha demostrado ser funcional, fiable y fácilmente extensible. Además, permite una aplicación realista de blockchain en contextos no financieros y sirve como base para líneas de investigación académica o industrial.

8.2. Trabajo Futuro

Durante el desarrollo del proyecto han surgido oportunidades de mejora y ampliación que podrían ser abordadas en futuras fases o por otros investigadores interesados en la temática. Entre ellas destacan:

- **Firma digital de los datos**: integrar mecanismos de firma electrónica desde el propio dispositivo para asegurar también el origen de los datos, no solo su integridad.
- **Auditoría automatizada de integridad**: desarrollar un servicio que periódicamente verifique que los datos almacenados en el Data Lake siguen coincidiendo con los hashes registrados en blockchain, detectando posibles manipulaciones.
- **Visualización avanzada**: incorporar herramientas de visualización gráfica (por ejemplo, dashboards en tiempo real o mapas interactivos) para facilitar la explotación visual de los datos recogidos.
- **Despliegue sobre infraestructura en la nube**: migrar la solución a un entorno cloud completo (por ejemplo, AWS o GCP), aprovechando servicios gestionados para escalar de forma automática y segura.

- **Evaluación con dispositivos reales en escenarios reales**: desplegar el sistema en un entorno industrial o de monitorización ambiental con múltiples sensores distribuidos y analizar su comportamiento bajo carga.
- **Interoperabilidad con otras blockchains o sistemas externos**: estudiar la posibilidad de usar otras redes (como Hyperledger o Polygon), así como integraciones con sistemas ERP, bases de datos tradicionales o herramientas de análisis Big Data.

Estas líneas abren un camino claro para continuar explorando soluciones seguras y trazables en entornos IoT, ampliando tanto el alcance del sistema como su robustez frente a escenarios más exigentes.

Este capítulo ha sintetizado las principales conclusiones del trabajo, evaluando el grado de cumplimiento de los objetivos planteados y destacando las aportaciones técnicas del sistema propuesto en el contexto de la validación, almacenamiento y verificación de integridad de datos IoT. Asimismo, se han identificado varias líneas de trabajo futuro orientadas a la mejora de la solución, incluyendo pruebas con dispositivos reales, despliegue en entornos productivos o ampliación de funcionalidades mediante analítica avanzada. Con ello, se da por finalizado el presente Trabajo Fin de Máster.

Apéndices

Apéndice A

Documentación técnica de programación

A.1. Introducción

Este apéndice recoge la documentación técnica del proyecto desarrollado como parte del Trabajo Fin de Máster. Incluye detalles sobre la estructura del repositorio, instrucciones de instalación, ejecución, pruebas y el uso de los principales servicios que componen la arquitectura.

El código fuente completo está disponible en el siguiente repositorio público:

- **Repositorio GitHub:** `https://github.com/javalon/iot-trace-chain`

La documentación adicional del proyecto puede consultarse en:

- **Deepwiki:** `https://deepwiki.com/javalon/iot-trace-chain`

El proyecto permite procesar datos IoT en tiempo real, validarlos, almacenarlos en un Data Lake y registrar su integridad en una red blockchain simulada.

También incluye una interfaz web para la visualización y verificación de dichos datos.

A.2. Estructura de directorios

La estructura general del repositorio es la siguiente:

```
persistence-worker/ ... Lógica de procesamiento, validación y hash
blockchain/ ..... Contratos inteligentes y scripts de Hardhat
mosquitto/ ..... Configuración del broker MQTT
minio-mirror/ ..... Herramientas para consulta local (DuckDB)
back-api/ ..... Backend API con FastAPI + GraphQL
delta-reader/ ..... Servicio de lectura desde Delta Lake
iot-chain-front/ ..... Aplicación web en Angular
docker-compose.yaml ..... Orquestación de servicios
README.md ..... Documentación
```

A.3. Manual del programador

El código está modularizado en servicios Docker, cada uno con un propósito concreto. La lógica del sistema sigue una arquitectura basada en microservicios. Los puntos clave para desarrolladores son:

- **persistence-worker**: contiene la lógica de ingestión MQTT, validación con JSON Schema, cálculo de hashes y escritura en Delta Lake.
- **blockchain**: define y despliega el contrato inteligente en una red local (Hardhat o Ganache).
- **back-api** y **delta-reader**: exponen servicios GraphQL/REST para acceder a los datos.
- **iot-chain-front**: permite consultar y verificar los datos vía interfaz gráfica.

Las dependencias de Python se gestionan con `Poetry`. El sistema puede iniciarse completamente mediante Docker Compose.

A.4. Compilación, instalación y ejecución del proyecto

Para iniciar el entorno completo en desarrollo, se requiere tener instalado Docker y Docker Compose.

Inicio completo del sistema

Para facilitar el despliegue del sistema en entornos de desarrollo y pruebas, se ha definido una configuración basada en `Docker Compose` que permite levantar todos los servicios principales de forma automatizada. Esta aproximación garantiza la reproducibilidad del entorno, simplifica la instalación de dependencias y facilita la ejecución coordinada de los diferentes módulos del sistema.

La instrucción que se muestra a continuación construye las imágenes necesarias (en caso de que no existan localmente) y lanza todos los contenedores definidos en el archivo `docker-compose.yaml`.

```
docker compose up --build
```

Al ejecutar este comando, se pondrán en marcha los siguientes componentes clave de la arquitectura:

- **Broker MQTT (Mosquitto)**: encargado de recibir los mensajes de los dispositivos IoT.
- **Almacenamiento (MinIO + Delta Lake)**: infraestructura de almacenamiento escalable y compatible con S3.
- **Contrato blockchain (Ganache)**: red Ethereum local para registrar huellas digitales (hashes).
- **Backend y frontend**: servicios de API (FastAPI + GraphQL) y aplicación web (Angular) para visualización e interacción.

Ejecución de servicios auxiliares (perfil `manual`)

Además de los servicios principales, el sistema cuenta con herramientas opcionales que pueden facilitar el desarrollo, la depuración y la exploración de datos. Estos servicios no se lanzan por defecto, pero pueden activarse de forma individual mediante el perfil `manual` definido en `docker-compose.yaml`. Las variables de entorno utilizadas en dicho archivo disponen de un valor por defecto, pero pueden ser fácilmente sobrescritas si existen variables de entorno definidas en el sistema en el momento de ejecutar el comando, lo que permite adaptar el entorno sin necesidad de modificar los ficheros de configuración.

Por ejemplo, si se desea cambiar el valor por defecto de la variable `BLOCKCHAIN_RPC_URL`, se puede ejecutar el siguiente comando:

```
BLOCKCHAIN_RPC_URL=http://localhost:8545 \\
docker compose --profile manual up expedition
```

En este caso, el valor proporcionado sobrescribirá el definido por defecto en el archivo `docker-compose.yaml`, sin necesidad de modificarlo manualmente.

De forma alternativa, es posible definir variables de entorno de manera persistente mediante un archivo `.env` en la raíz del proyecto. Docker Compose las detectará automáticamente al iniciar los servicios. Por ejemplo, el siguiente contenido en un archivo `.env` establecerá la URL del nodo blockchain:

```
BLOCKCHAIN_RPC_URL=http://localhost:8545
DATA_LAKE_BUCKET=trace-data
...
```

Este enfoque permite personalizar el entorno de ejecución sin modificar directamente el archivo `docker-compose.yaml`, facilitando la portabilidad y el versionado del proyecto.

Para ejecutar uno de estos servicios, se debe emplear el siguiente comando, sustituyendo el nombre por el del servicio deseado:

```
docker compose --profile manual up mqtt-explorer
```

Entre los servicios auxiliares más destacados se encuentran:

- **MQTT Explorer**: una herramienta con interfaz gráfica para inspeccionar temas MQTT y los mensajes recibidos en tiempo real.
- **Expedition**: explorador web para redes Ethereum locales, útil para visualizar transacciones, bloques y contratos desplegados.
- **minio-mirror**: servicio para crear una copia local de los datos almacenados en MinIO, útil para análisis offline.
- **duckdb**: entorno SQL interactivo que permite consultar datos en formato Parquet o Delta directamente desde el sistema de archivos.

Estos servicios se inician de forma aislada, lo que permite activarlos únicamente cuando se necesitan, sin afectar al resto de la infraestructura.

Ejecución manual del worker

En caso de querer ejecutar el módulo principal manualmente:

```
cd persistence-worker
poetry install
poetry run poe start-local
```

Opcionalmente, se puede usar:

```
poetry run poe publish-message
```

para enviar un mensaje de prueba MQTT.

A.5. Pruebas del sistema

El proyecto incorpora pruebas en varias capas:

- **Backend**: pruebas unitarias con `pytest` para validación, hash y escritura.

- **Contratos inteligentes**: pruebas con Hardhat (JavaScript) sobre red local.
- **Frontend**: pruebas unitarias con Jasmine/Karma en Angular.
- **CI/CD**: ejecución automática de tests mediante GitHub Actions en cada `push` o `pull request`.

Apéndice B

Documentación de usuario

B.1. Introducción

Este apéndice está dirigido a los usuarios finales del sistema, en especial a aquellos que necesitan consultar los datos generados por dispositivos IoT, verificar su integridad o interactuar con el sistema a través de su interfaz web. Se describe el proceso de instalación, los requisitos necesarios y un manual básico de uso orientado a usuarios no técnicos.

B.2. Requisitos de usuarios

Para utilizar la plataforma se requiere acceso a un navegador moderno y conexión a la red donde esté desplegado el sistema. Las funcionalidades principales disponibles para el usuario son:

- Visualización de dispositivos IoT registrados.
- Consulta de datos sensorizados almacenados.
- Verificación de la integridad de los datos mediante blockchain.
- Interacción con la interfaz web desarrollada en Angular.

No se requieren conocimientos técnicos sobre blockchain, IoT o big data para la utilización básica del sistema.

B.3. Instalación

En entornos de producción, la instalación del sistema será realizada por personal técnico. No obstante, para entornos de prueba o demostración, el usuario puede clonar el repositorio y lanzar el sistema con Docker:

```
git clone https://github.com/javalon/iot-trace-chain.git
cd iot-trace-chain
docker compose up --build
```

Una vez desplegado, se puede acceder a la interfaz web a través de la URL:

```
http://localhost:4200
```

B.4. Manual del usuario

La interacción con el sistema se realiza a través de una interfaz web accesible desde el navegador. A continuación, se describen los pasos principales para comenzar a utilizarla, así como las funcionalidades disponibles.

Inicio de sesión

Para acceder al sistema, el usuario debe autenticarse con unas credenciales predefinidas. Por defecto, se puede utilizar el siguiente usuario de prueba con rol de administrador:

- **Usuario:** `john@doe.es`
- **Contraseña:** `securepassword123`

En la Figura B.1 se muestra la pantalla de inicio de sesión.

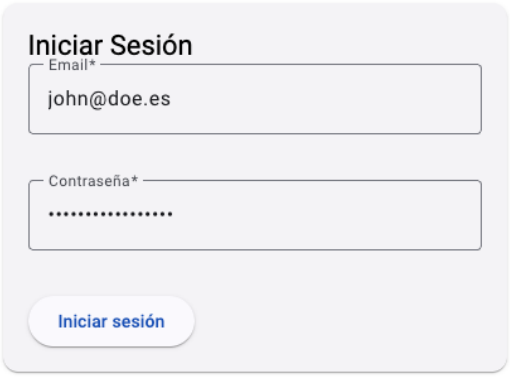

Figura B.1: Pantalla de inicio de sesión

Pantalla principal

Una vez autenticado, el usuario accede a la pantalla principal del sistema, donde se resumen las opciones disponibles. Desde aquí puede navegar al listado de usuarios, listado de dispositivos o consultar datos registrados.

Figura B.2: Interfaz principal tras el inicio de sesión

Visualización de dispositivos y datos

En la sección **Datos**, el usuario puede visualizar los dispositivos disponibles. Al seleccionar uno, se accede a la vista de datos sensorizados almacenados. Estos datos pueden incluir temperatura, posición geográfica u otras variables.

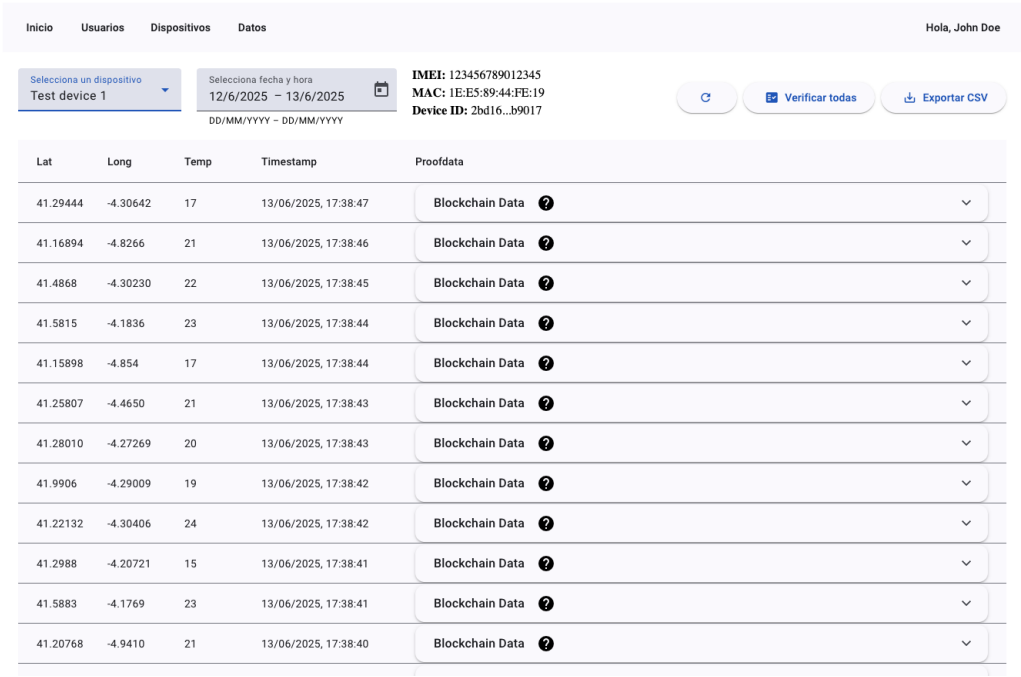

Figura B.3: Visualización de los datos de un dispositivo IoT

Verificación de integridad

Mediante los botones de verificación, el usuario puede iniciar la verificación de los datos frente a la blockchain y comprobar si los datos almacenados han sido alterados. El sistema realiza la verificación de integridad comparando los datos y metadatos almacenados localmente con el *hash raíz* del árbol de Merkle que fue previamente registrado en la blockchain. Para ello, se emplean pruebas de Merkle (*Merkle proofs*) que permiten reconstruir el camino desde el dato hasta la raíz, garantizando así que el dato no ha sido alterado desde su inserción original. La Figura B.4 muestra la pantalla con los datos validados.

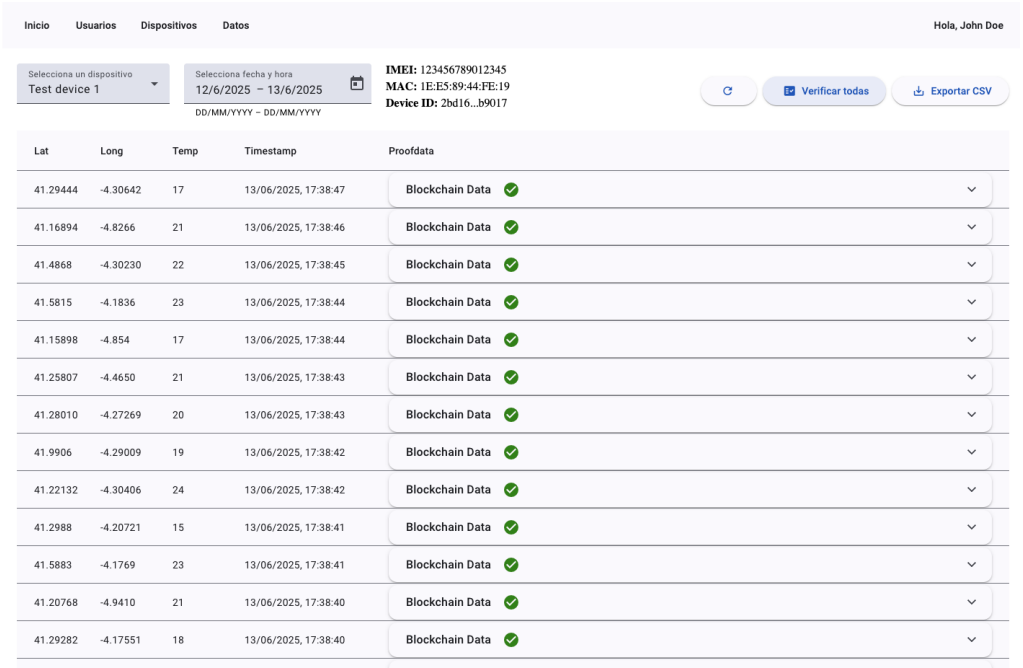

Figura B.4: Comprobación de integridad de los datos mediante blockchain

Esta verificación proporciona garantías de que los datos no han sido manipulados desde su recepción, aportando transparencia y trazabilidad al sistema.

En la sección Blockchain Data se muestra la información relacionada con la verificación de integridad del dato seleccionado, tal y como se puede observar en la Figura B.5. Se incluyen los identificadores clave del registro, como el `Tx` (hash de la transacción en blockchain), el `Hash` del dato concreto, el `RecordId` asociado y el `Merkle root`, que representa el hash raíz del árbol de Merkle en el que se agrupan los datos validados. Justo debajo, se presenta la *Merkle proof* en formato JSON, que contiene los metadatos del algoritmo de hash utilizado (por ejemplo, `sha256`), el tamaño del árbol y la secuencia de nodos (`path`) necesarios para verificar criptográficamente que el dato pertenece al árbol cuya raíz está registrada en la blockchain. Esta información permite validar la integridad del dato sin necesidad de acceder al resto del conjunto de datos, garantizando trazabilidad y seguridad.

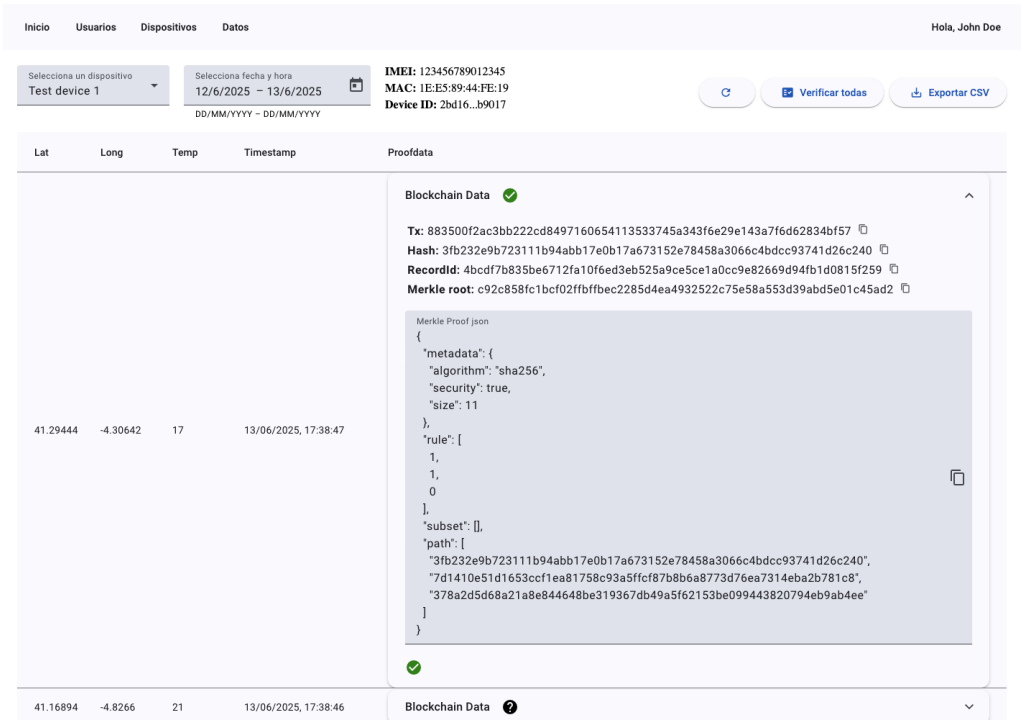

Figura B.5: Detalle de los metadatos del dato y verificación en blockchain

Bibliografía

[1] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[2] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[3] Ana Reyna, Cristian Martín, Jaime Chen, Enrique Soler, and Manuel Díaz. On blockchain and its integration with iot. challenges and opportunities. *Future Generation Computer Systems*, 88:173–190, 2018.

[4] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoc. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273, 2016.

[5] Pwint Phyu Khine and Zaw Wang. Data lake: A new ideology in big data era. In *2018 6th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, volume 2, pages 311–315. IEEE, 2018.

[6] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.

[7] SARTECO. Jornadas sarteco – sociedad de arquitectura y tecnología de computadores. `https://jornadassarteco.org/`, 2025. Consultado en julio de 2025.

[8] MQTT Organization. Mqtt - the standard for iot messaging. `https://mqtt.org/`, 2025. Consultado en julio de 2025.

[9] JSON Schema. Json schema - a vocabulary that allows you to annotate and validate json documents. https://json-schema.org/, 2025. Consultado en julio de 2025.

[10] Delta Lake Project. Delta lake: Open-source storage framework for reliable data lakes. https://delta.io/, 2025. Consultado en julio de 2025.

[11] Wikipedia. Contrato inteligente. https://es.wikipedia.org/wiki/Contrato_inteligente, 2025. Consultado en julio de 2025.

[12] IoT Analytics. State of iot summer 2024 – number of connected iot devices. Press release, 2024. 16.6 B devices in 2023; 18.8 B forecast for 2024; 40 B by 2030; accessed July 2025.

[13] Estuary. 72+ eye-opening iot statistics, facts, & trends for 2024. Online article, 2024. Accessed July 2025.

[14] PowerData. ¿qué es big data? https://www.powerdata.es/big-data, 2025. Consultado en julio de 2025.

[15] IBM. ¿qué es blockchain? https://www.ibm.com/es-es/topics/blockchain, 2025. Consultado en julio de 2025.

[16] Amazon Web Services. Amazon simple storage service (amazon s3). https://aws.amazon.com/es/s3/, 2025. Consultado en julio de 2025.

[17] Michael Armbrust, Tathagata Das, Shixuan Zhu, Reynold Xin Hernandez, et al. Delta lake: High-performance acid table storage over cloud object stores. In *Proceedings of the VLDB Endowment*, volume 13, pages 3411–3424, 2020.

[18] Apache Software Foundation. Apache parquet. https://parquet.apache.org/, 2025. Consultado en julio de 2025.

[19] KeepCoding. ¿qué son los algoritmos de consenso en blockchain? https://keepcoding.io/blog/que-son-algoritmos-de-consenso-blockchain/, 2025. Consultado en julio de 2025.

[20] Wikipedia. Sellado de tiempo confiable. https://es.wikipedia.org/wiki/Sellado_de_tiempo_confiable, 2025. Consultado en julio de 2025.

[21] Wikipedia. Árbol de merkle. https://es.wikipedia.org/wiki/%C3%81rbol_de_Merkle, 2025. Consultado en julio de 2025.

[22] Kraken Learn team. What is a blockchain gas fee? `https://www.kraken.com/learn/what-is-a-blockchain-gas-fee`, 2023. Consultado en julio de 2025.

[23] Javier Alonso-Núñez, Daniel López-Martínez, and Diego R. Llanos. Arquitectura segura para la trazabilidad basada en iot y blockchain. In *XXXV Jornadas de Paralelismo (JP2025)*, Sevilla, Spain, 2025. Universidad de Sevilla.