

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE ELECTRÓNICA

SANTIAGO - CHILE



**UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA**

“OPTIDATALAKE: PLATAFORMA PARA LA GESTIÓN DE MARKET DATA EN TIEMPO REAL”

Informe Individual de Memoria

ARIEL BRAVO PONCE

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL
TELEMÁTICO

Profesor Guía:

Nicolás Torres

Profesor Correferente:

Patricio Olivares

Octubre - 2025



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: OptiDataLake: PLATAFORMA PARA LA GESTIÓN DE MARKET DATA EN TIEMPO REAL

Nombre del candidato(a): Ariel Alonso Bravo Ponce

Carrera / Grado: Ingeniería Civil Telemática

Campus: San Joaquín

Departamento: Electrónica

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Nicolás Torres Rudloff, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (**marcar una opción**):

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 13/10/2025

Firma:

Estudiante o Candidato(a):

Fecha: 13/10/2025

Firma:

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

Agradecimientos

Quiero expresar mi más profundo agradecimiento, en primer lugar, a mi profesor guía por todo el apoyo y el seguimiento que me brindó a lo largo de este proyecto, así como por las cátedras y los consejos que siempre nos entregó. Extiendo este agradecimiento a todos los docentes que me acompañaron durante la carrera, ya sea por sus clases o por sus orientaciones. Su dedicación, paciencia y disposición para compartir conocimientos y experiencias de vida me ayudaron a seguir adelante pese a las dificultades de la carrera; muchos de esos consejos fueron clave para superar los desafíos de este proyecto y para mi crecimiento profesional y personal.

Agradezco también a mi familia por su apoyo, paciencia y consejos. En especial, a mis papás, cuyos consejos me ayudaron a tranquilizarme y a sobrellevar los tropiezos y obstáculos que se presentaron. Su comprensión y el espacio que me dieron para concentrarme en los momentos difíciles fueron fundamentales para que este trabajo llegara a buen término.

Quiero destacar el apoyo incondicional de mis amigos y compañeros de estudio: gracias por el respaldo constante, por las conversaciones hasta tarde buscando resolver problemas y dudas, y por estar presentes tanto en los buenos momentos como, sobre todo, en los más difíciles. Ustedes conocían mejor que nadie las dificultades que atravesamos; gracias a su apoyo pude avanzar hasta llegar a este punto.

Finalmente, agradezco a mi equipo de proyecto por el compromiso, la colaboración y las noches de trabajo que dedicaron para llevar a buen puerto este proyecto. Cada aporte, conversación, revisión y sugerencia contribuyó más de lo que imaginan a mejorar la calidad de esta tesis.

*Esta tesis está dedicada a **mi familia**. Por su paciencia, consejos y sobre todo el apoyo que me dieron para poder llevar a cabo este proyecto de buena manera, todos sus consejos fueron clave para poder alcanzar mi meta propuesta.*

Resumen

Este trabajo presenta el diseño e implementación de una plataforma para la gestión y análisis de *market data* en tiempo real, integrada con **Apache Kafka**, **Apache Spark** y **MongoDB** bajo un enfoque de *Data Lake* (colecciones *Raw*, *Structured* y *Analytics*). El objetivo es habilitar **ingestión continua**, **procesamiento confiable** y **recuperación eficiente** con baja latencia para analítica operativa y estudios históricos.

La solución captura datos de alta frecuencia desde la API de Binance, los valida y enriquece, y los persiste para su consulta. La arquitectura modular y de bajo acoplamiento incorpora prácticas de operación confiable (validadores de esquema, índices por símbolo/tiempo, manejo de idempotencia y control de duplicados), favoreciendo robustez y escalabilidad.

Se desarrolló una interfaz de consulta para pruebas que permite filtrar por símbolo y rango de precios y que reporta tamaño recuperado, tiempo de respuesta y *ancho de banda* efectivo (MB/s). Las pruebas de rendimiento muestran recuperaciones consistentemente por sobre **20 MB/s**, incluso con grandes volúmenes de datos, mientras que la tasa de ingestión por minuto se mantiene estable en *Raw_data* y *Structured*. Los resultados confirman continuidad del flujo, ausencia de cuellos de botella en escritura y preparación de datos para *dashboards*, KPIs y análisis históricos.

En conjunto, la plataforma cumple los objetivos propuestos y constituye una base **escalable** para futuras extensiones y despliegues productivos, habilitando **analítica en tiempo real** sobre datos de mercado de alta frecuencia.

Palabras clave: Big Data, *Streaming*, Apache Kafka, Apache Spark, MongoDB, Analítica en Tiempo Real.

Glosario

Market Data:

Información de mercado usada en finanzas (precios, volúmenes, libro de órdenes y eventos asociados).

Tick by Tick:

Registro de cada cambio individual del mercado (precio, volumen o estado del libro) en el instante en que ocurre.

Análisis en Tiempo Real:

Interpretación y procesamiento de datos a medida que se generan o arriban, con el fin de entregar información útil o acciones en el menor tiempo posible. Es esencial cuando la inmediatez es crítica, como en el análisis de mercados financieros o en la detección de eventos en sistemas de monitoreo.

Procesamiento en Streaming:

Técnica que procesa eventos continuamente en tiempo real a medida que se van generando.

Procesamiento en Batch:

Técnica que agrupa datos y los procesa por lotes en momentos definidos.

Fuente de Datos:

Origen desde el cual se obtiene la información que alimenta el sistema (por ejemplo, una API o un flujo de eventos). Para este proyecto se utilizó la API de Binance.

Formato de Datos:

Estructura en que se representan los datos para su interpretación y almacenamiento por el sistema. En el proyecto se utiliza JSON por su legibilidad y compatibilidad con APIs.

JSON (JavaScript Object Notation):

Formato de intercambio de datos ligero y legible, común en APIs y servicios web.

BSON (Binary JSON):

Formato binario usado por MongoDB para almacenar documentos de manera eficiente.

Data Lake:

Repositorio central que guarda datos en distintos estados (crudos, estructurados y analíticos) para su posterior uso.

NoSQL:

Familia de bases de datos diseñada para manejar grandes volúmenes y esquemas flexibles, más allá del modelo relacional tradicional.

MongoDB:

Base de datos NoSQL orientada a documentos (BSON/JSON) que facilita consultas rápidas y escalado horizontal. Se utiliza en el proyecto como almacenamiento principal para todos los datos procesados.

Índice (BD):

Estructura que acelera las consultas sobre uno o más campos (p. ej., símbolo y tiempo).

Sharding / Replicación:

Partición horizontal de los datos para escalar / mantenimiento de copias para alta disponibilidad.

Apache Kafka:

Plataforma distribuida de mensajería para consumo, ingesta y transporte de grandes volúmenes de datos en tiempo real con tolerancia a fallos.

Topic (Kafka):

Canal lógico donde los productores publican mensajes y los consumidores los leen.

Partición (Kafka):

Segmento de un *topic* que permite paralelismo y escalado.

Producer / Consumer:

Proceso que publica / lee mensajes en Kafka.

Offset: Posición de lectura dentro de una partición de Kafka.

Apache Spark:

Framework de procesamiento de datos en clústeres. En este proyecto se utiliza para **streaming y batch**, facilitando el análisis de datos en tiempo real.

microbatches (Spark):

Lote pequeño de eventos procesados en intervalos breves dentro de *Structured Streaming*.

Ventana de Tiempo:

Intervalo temporal sobre el que se calculan agregaciones (p. ej., 1 minuto).

Checkpoint:

Estado almacenado de un flujo para reanudar el procesamiento sin perder ni duplicar eventos.

Idempotencia:

Propiedad por la cual procesar un mismo evento una o más veces produce el mismo resultado (evita duplicados).

Latencia / Throughput:

Tiempo de respuesta / volumen procesado o transferido por unidad de tiempo.

Binance API:

Interfaz de programación de aplicaciones que entrega *market data* y operaciones del exchange en tiempo real.

Ingesta de Datos:

Conjunto de procesos que reciben flujos o lotes de múltiples orígenes y los almacenan, de forma ordenada y segura, al repositorio del sistema para su análisis y procesamiento.

Big Data:

Conjunto de datos de gran volumen, variedad y velocidad de generación que excede las capacidades de las herramientas tradicionales para capturarlos, almacenarlos y analizarlos de manera eficiente.

Acrónimos

API	Application Programming Interface.
CLI	Command Line Interface.
GUI	Graphical User Interface.
I/O	Input/Output.
JVM	Java Virtual Machine.
KPI	Key Performance Indicator.
SDK	Software Development Kit.
SQL	Structured Query Language.
ER	Entidad–Relación.
ETL / ELT	Extract, Transform, Load / Extract, Load, Transform.
HFT	High-Frequency Trading.
MB/s	Megabytes por segundo.
UTC	Coordinated Universal Time.
QoS / SLA	Quality of Service / Service Level Agreement.
RTO / RPO	Recovery Time Objective / Recovery Point Objective.
TTL	Time To Live.

Tabla de Contenido

Agradecimientos	I
Resumen	III
Glosario	IV
Acrónimos	VII
1 Introducción	1
2 Marco General	3
2.1 Definición del Problema	3
2.2 Acercamiento de la Solución	3
2.3 Descripción del Sistema	4
2.4 Tecnologías Utilizadas	5
2.4.1 MongoDB	5
2.4.2 Apache Kafka	6
2.4.3 Apache Spark	7
2.4.4 Resumen comparativo	8
2.5 Estado del Arte y Técnica	8
3 Objetivos	10
3.1 Objetivo general	10
3.2 Objetivos específicos	10
4 Diseño de la Solución	11
4.1 Requisitos del Sistema	11
4.1.1 Requisitos funcionales	11
4.1.2 Requisitos no funcionales	11
4.1.3 Requisitos de interfaces	12
4.1.4 Requisitos de ambiente	12
4.1.5 Perfiles de usuario	14
4.2 Arquitectura del Sistema	14
4.2.1 Diagrama de contexto	15
4.2.2 Esquema general del sistema	15
4.2.3 Diagrama de arquitectura	16
4.2.4 Definición y Diseño de Módulos	17

4.2.5	Matriz de requisitos funcionales y componentes	23
4.3	Estructura de Datos en el Data Lake	24
4.3.1	Datos Crudos (Raw Data)	24
4.3.2	Estructurados	25
4.3.3	Analytics (KPIs y Métricas Derivadas)	26
4.3.4	Esquema UML del Modelo de Datos	27
4.4	Diseño de Interfaces	28
4.4.1	Diagrama Entidad–Relación	28
4.4.2	Diagrama de Flujo del Sistema	30
4.5	Síntesis del Diseño del Sistema	32
4.6	Caso de uso	33
4.6.1	Monitoreo de Datos en Tiempo Real a través de un Dashboard	33
5	Desarrollo de la Aplicación	36
5.1	Revisión del Sistema	36
5.1.1	Flujo de Trabajo para el Desarrollo del Sistema	37
5.1.2	Diagrama de Componentes del Sistema	39
5.2	Implementación de los Componentes	41
5.2.1	Ingestión de Datos	41
5.2.2	Contribución en el módulo de ingestión de datos.	43
5.2.3	Procesamiento de Datos	44
5.2.4	Flujo del procesamiento de los datos	46
5.2.5	Almacenamiento de Datos	48
5.3	Validación y Pruebas del Sistema	52
5.3.1	Ingesta de datos desde la API de Binance: Producer y Consumer . . .	52
5.3.2	Procesamiento de datos con Spark	53
5.3.3	Almacenamiento: MongoDB (Raw y Analytics)	53
5.3.4	Almacenamiento: Structured	54
5.3.5	Contribuciones en las pruebas para la validación del Sistema	55
5.4	Pruebas de Rendimiento	56
5.4.1	Objetivo	56
5.4.2	Entorno y datos	56
5.4.3	Procedimiento	57
5.4.4	Métricas	58
5.4.5	Resultados	58
5.4.6	Análisis e interpretación	60
5.4.7	Evaluación del sistema completo	60
5.4.8	Contribuciones en las Pruebas de Rendimiento	60
5.5	Conclusión	61
5.5.1	Trabajo Futuro	62

Bibliografía	63
Anexos	64

Índice de Figuras

4.1 Diagrama de contexto del sistema.	15
4.2 Esquema general del sistema y flujo de información.	16
4.3 Diagrama de arquitectura detallada del sistema.	17
4.4 Esquema UML del modelo de datos implementado en el Data Lake.	28
4.5 Diagrama Entidad–Relación del sistema.	29
4.6 Diagrama de flujo del sistema.	30
4.7 Dashboard de monitoreo con selección de símbolo, métrica y rango de fechas.	34
5.1 Visión general de los componentes y su interacción en el flujo de datos.	40
5.2 Esquema simplificado del flujo de ingesta con Apache Kafka.	41
5.3 Secuencia de publicación (Producer) y recepción (Consumer) durante la ingesta.	43
5.4 Arquitectura de procesamiento con Apache Spark (Driver, Cluster Manager y Workers).	45
5.5 Log del <i>Producer</i> : evento enviado a <code>odl-ticks</code>	52
5.6 Log del <i>Consumer</i> : evento recibido desde <code>odl-ticks</code> y confirmado para persistencia.	52
5.7 Ejecución en consola: KPIs calculadas.	53
5.8 Colección <i>Raw_data</i> : eventos crudos con campos de Binance.	53
5.9 Colección <i>Analytics</i> : KPIs y métricas por ventana de tiempo.	54
5.10 Ejemplos de organización en <i>Structured</i> para consultas por producto y por mercado.	55
5.11 Interfaz de consulta utilizada para medir tiempo de respuesta, filas recuperadas y tamaño total.	57
5.12 Ancho de banda efectivo (MB/s) versus rango de datos recuperados. La línea punteada marca 20 MB/s.	58
5.13 Datos ingresados por minuto en <i>Raw_data</i>	59
5.14 Datos ingresados por minuto en <i>Structured</i>	59

Índice de Tablas

2.1	Resumen de ventajas y desventajas de las tecnologías utilizadas	8
4.1	Requisitos de interfaces	12
4.2	Perfiles de Usuario	14
4.3	Descripción del módulo Data Ingestion (DI)	18
4.4	Descripción del módulo Raw / Landing (RD)	19
4.5	Descripción del módulo Transform (T)	20
4.6	Descripción del módulo Structured Data (SD)	21
4.7	Descripción del módulo Analytics (A)	22
4.8	Descripción del módulo Data Lake (DL)	23
4.9	Matriz de requisitos funcionales y módulos	23
5.1	Campos relevantes en los logs de Binance.	43

1 Introducción

En los últimos años, el crecimiento del *Big Data* y de los sistemas de análisis en tiempo real ha cambiado la forma en que las instituciones financieras recolectan y usan la información. En este escenario, la *market data tick by tick*, que registra cada variación de precio, volumen y estado del libro de órdenes, se ha vuelto un factor crítico para la toma de decisiones y la gestión del riesgo. No obstante, la alta velocidad en que se generan los datos y el gran volumen de estos hacen insuficientes muchas soluciones tradicionales de almacenamiento y procesamiento, que no logran responder con baja latencia ni mantener la trazabilidad completa de los datos.

Este informe aborda ese desafío mediante el diseño e implementación de una plataforma de datos basada en un *Data Lake*, cuyo objetivo es habilitar la ingesta, el procesamiento, el almacenamiento y las consultas de *market data* en tiempo real de forma confiable y escalable. La arquitectura propuesta organiza la información en tres zonas complementarias: **Raw** (dato crudo preservado tal como llega), **Structured** (dato limpio y consistente para consulta) y **Analytics** (métricas y resultados derivados). Esta separación permite conservar el detalle original de la data (data histórica), mejorar el rendimiento de las consultas y asegurar trazabilidad de extremo a extremo.

Para implementar el flujo se integran tecnologías de uso extendido en la industria. **Apache Kafka** se utiliza para recibir y distribuir datos de manera continua y tolerante a fallos; **Apache Spark** realiza la validación, transformación y cálculo de analíticas tanto en *streaming* como en *batch*; y **MongoDB** ofrece un almacenamiento flexible orientado a documentos, adecuado para grandes volúmenes y consultas rápidas. Como fuente de datos se considera, a modo de referencia, la API de Binance, sin perder generalidad para incorporar otras fuentes en el futuro que estén enfocadas en otros mercados o áreas diferentes al de una corredora de bolsa.

El desarrollo de los módulos de ingesta y procesamiento permitió implementar una arquitectura modular, con bajo acoplamiento entre etapas y preparada para escalar por componentes. Las pruebas efectuadas con datos reales validan la continuidad del flujo, la consistencia temporal de las ventanas y los tiempos de respuesta esperados, evidenciando que la solución puede gestionar grandes volúmenes de información de manera eficiente, confiable y rápida. En conjunto, el sistema establecido sirve como base técnica y referencia para proyectos posteriores de análisis en tiempo real, dejando abierta la posibilidad de

integrar nuevas fuentes, ampliar el catálogo de métricas y evolucionar hacia capacidades de predicción y alertas.

2 Marco General

2.1 Definición del Problema

El mercado financiero actual genera enormes volúmenes de información en tiempo real. Cada variación en los precios, el volumen de transacciones y las órdenes de compra o venta produce un flujo de datos continuo conocido como *market data tick by tick*. La alta frecuencia y el nivel de detalle de estos eventos convierten su gestión en un desafío técnico considerable.

Las soluciones tradicionales de almacenamiento y procesamiento no están diseñadas para manejar estas tasas de generación de datos, lo que puede producir pérdidas de información, demoras en el acceso y dificultades para analizar de manera oportuna la información crítica. La capacidad de una corredora de bolsa para reaccionar a condiciones de mercado depende directamente de contar con datos confiables, completos y disponibles en milisegundos, por lo que la ineficiencia en la gestión de este flujo puede impactar de forma negativa en la toma de decisiones y en la competitividad.

En este contexto, el problema a resolver consiste en diseñar e implementar una plataforma robusta y escalable que permita ingestar, procesar, almacenar y poner a disposición grandes volúmenes de datos de alta frecuencia en tiempo real. La solución debe garantizar continuidad en la recepción de la información, integridad en el almacenamiento y flexibilidad para integrar nuevas fuentes, todo bajo un enfoque que soporte la escalabilidad futura y mantenga baja latencia.

2.2 Acercamiento de la Solución

La problemática identificada requiere una solución que permita gestionar de forma integral el flujo de *market data* en tiempo real, asegurando su disponibilidad y fiabilidad para el análisis y la toma de decisiones. Para ello, se propone el diseño e implementación de una **plataforma basada en un Data Lake**, capaz de centralizar el ciclo completo de la información: ingestión, procesamiento, almacenamiento y consulta.

El Data Lake se selecciona como eje de la solución debido a su capacidad para **almacenar datos en diferentes niveles de tratamiento**: desde registros crudos (*raw data*) capturados directamente de la fuente, hasta estructuras transformadas y métricas analí-

tivas derivadas. Este enfoque garantiza la preservación del detalle original de los eventos y, al mismo tiempo, habilita su explotación en distintos niveles de análisis.

La arquitectura propuesta se articula sobre tres componentes fundamentales:

1. **Ingesta de datos en tiempo real:** un sistema de mensajería distribuido que permite recibir y transmitir los eventos financieros de manera continua, tolerante a fallos y con baja latencia.
2. **Procesamiento distribuido:** un motor de análisis en *streaming* y *batch*, capaz de transformar los datos crudos, aplicar reglas de limpieza y cálculo, y generar métricas relevantes para el negocio.
3. **Almacenamiento flexible y escalable:** una base de datos NoSQL orientada a documentos, que ofrece consultas rápidas, esquemas dinámicos y soporte para la persistencia tanto de datos históricos como de resultados analíticos.

Este enfoque modular asegura que cada componente pueda evolucionar o escalarse de manera independiente, manteniendo una integración coherente entre las distintas etapas. Asimismo, se establecen principios de **escalabilidad, disponibilidad y resiliencia**, de manera que la plataforma no solo cubra los requerimientos actuales de ingestión de datos financieros, sino que también se proyecte como una solución adaptable a futuros escenarios con mayores volúmenes de información o con nuevas fuentes de datos.

En síntesis, el acercamiento planteado consiste en construir una **infraestructura robusta, flexible y adaptable**, que permita administrar flujos de datos de alta frecuencia con confiabilidad y eficiencia, constituyendo una base tecnológica sólida para el análisis en tiempo real en el ámbito financiero.

2.3 Descripción del Sistema

La solución propuesta se organiza como una **plataforma modular** cuyo objetivo principal es habilitar el flujo completo de los datos financieros, desde su captura en origen hasta su disposición para análisis en distintos niveles.

El sistema se estructura en tres capas fundamentales:

1. **Capa de Ingesta:** recibe los flujos de *market data* en tiempo real y asegura su llegada continua mediante un sistema de mensajería distribuido. Su función es garantizar que no existan pérdidas de información y que los datos estén disponibles inmediatamente para su procesamiento.
2. **Capa de Procesamiento:** transforma los datos crudos en estructuras consistentes, aplicando procesos de limpieza, enriquecimiento y cálculo de métricas. Esta capa opera tanto en modalidad *streaming* (para análisis en tiempo real) como en modalidad *batch* (para análisis históricos y consolidación).

3. **Capa de Almacenamiento y Explotación:** organiza la información en distintas zonas dentro del Data Lake:

- **Zona Raw:** preserva los registros tal como se reciben.
- **Zona Estructurada:** normaliza los datos y los adapta a modelos consistentes.
- **Zona Analítica:** integra métricas derivadas y KPIs listos para consultas y visualización.

Cada una de estas capas interactúa de manera orquestada, permitiendo un flujo de información continuo y trazable. La arquitectura busca mantener **bajo acoplamiento**, de modo que cada componente pueda evolucionar o escalar de forma independiente sin comprometer la estabilidad global del sistema.

De esta forma, la plataforma resultante no solo asegura la ingesta confiable y el almacenamiento eficiente de datos de alta frecuencia, sino que también ofrece las herramientas necesarias para su análisis y explotación en escenarios de negocio críticos.

2.4 Tecnologías Utilizadas

El desarrollo de la plataforma requirió seleccionar tecnologías capaces de soportar ingesta continua de datos en tiempo real y también que contara con almacenamiento flexible. Cada componente fue seleccionado en función de sus características técnicas y de su capacidad para integrarse en una arquitectura modular y escalable. A continuación se describe cada tecnología utilizada.

2.4.1 MongoDB

Descripción y funcionamiento básico: Es el componente que se encargará de almacenar las grandes cantidades de datos (Market data) que se manejarán. MongoDB es una base de datos NoSQL o no relacional orientada a documentos. A diferencia de los sistemas relacionales, no utiliza tablas estructuradas, sino colecciones de documentos en formato BSON (similar a JSON). Esto permite almacenar tanto datos crudos (raw data) como datos ya procesados que cuentan con estructuras dinámicas y diferentes entre sí, lo que lo convierte en una herramienta adecuada para datos con diferentes formatos. Sus operaciones de lectura y escritura están optimizadas para manejar grandes volúmenes de datos en entornos distribuidos lo que nos permite realizar consultas más rápidas y eficientes.

Ventajas:

- Flexibilidad de esquema, ideal para datos con estructuras cambiantes (datos estructurados y no estructurados).

- Buen rendimiento y rapidez en consultas para obtener los datos.
- Escalabilidad horizontal para grandes volúmenes de datos (*sharding*).
- Replicación nativa para asegurar alta disponibilidad.

Desventajas:

- Mayor consumo de memoria si no se gestionan adecuadamente los índices.
- Requiere ciertos ajustes y ayuda de otras tecnologías para poder optimizar el rendimiento para grandes volúmenes de información.

Justificación de uso: Se eligió MongoDB porque el proyecto debía almacenar tanto datos crudos como estructurados y métricas derivadas, todos con estructuras cambiantes. Su flexibilidad de esquema y facilidad para escalar lo convierten en la mejor alternativa frente a bases de datos relacionales tradicionales.

2.4.2 Apache Kafka

Descripción y funcionamiento básico: Es el componente encargado de la ingestión de los datos, permite una transmisión de datos que sea escalable y también confiable, sin pérdida de integridad de la información transmitida desde la API.

Apache Kafka es una plataforma distribuida de mensajería diseñada para manejar flujos de datos en tiempo real. Funciona con un modelo *publicador-suscriptor* (*Pub/Sub*), en el que los productores envían mensajes a *topics* y los consumidores los leen de manera secuencial. Kafka utiliza *brokers* para gestionar la persistencia de los mensajes y replicación, asegurando tolerancia a fallos.

Ventajas:

- Capacidad para manejar grandes volúmenes de datos en tiempo real.
- Tolerancia a fallos mediante replicación de particiones.
- Baja latencia en la transmisión de mensajes.
- Permite la integración de diferentes tecnologías y sistemas como por ejemplo de análisis y almacenamiento de datos.
- Alta tolerancia a fallas y escalabilidad.

Desventajas:

- Curva de aprendizaje muy alta para su configuración, operación y mantenimiento.

- Complejidad en la administración de clústeres grandes.
- Complejidad para aprender a configurar los tópicos y también para poder integrarla con otras tecnologías.

Justificación de uso: Kafka fue seleccionado porque asegura orden, durabilidad y transmisión de mensajes en tiempo real, características que son críticas en un entorno financiero donde no puede haber pérdidas de datos.

2.4.3 Apache Spark

Descripción y funcionamiento básico: Es el componente encargado del procesamiento de los datos, se encarga de transformar, parsear y analizar los datos que fueron recibidos en tiempo real, ayudando a generar métricas y gráficos para una futura interpretación.

Apache Spark es un motor de procesamiento distribuido para grandes volúmenes de datos. Permite realizar operaciones tanto en modo *batch* (lotes de datos) como en *streaming* (procesamiento en tiempo real). Su arquitectura se basa en clústeres, donde un *driver* coordina la ejecución y múltiples *workers* ejecutan las tareas en paralelo.

Ventajas:

- Procesamiento distribuido en memoria, lo que permite mayor rapidez.
- Soporte para procesamiento por lotes y en tiempo real.
- Ecosistema con múltiples APIs (SQL, MLlib, GraphX, Structured Streaming).
- Escalabilidad en clústeres grandes.
- Arquitectura con alta escalabilidad y tolerancia a fallas.

Desventajas:

- Alto consumo de memoria si no se configura adecuadamente.
- Se necesita un ajuste preciso para evitar cuellos de botella.
- Alta complejidad en la configuración de los clústeres.

Justificación de uso: Spark se seleccionó como motor de procesamiento porque el proyecto requería limpiar, transformar y analizar flujos de datos en tiempo real. Su capacidad para integrar análisis en streaming y batch permite construir métricas y KPIs de manera eficiente y rápida.

2.4.4 Resumen comparativo

La integración de MongoDB, Kafka y Spark responde a la necesidad de cubrir todo el ciclo de vida del dato: ingesta, procesamiento y almacenamiento. Cada tecnología aporta una ventaja competitiva al sistema, aunque también implica desafíos de administración, configuración y recursos.

Tecnología	Ventajas	Desventajas
MongoDB	Flexibilidad de esquema; consultas rápidas; escalabilidad horizontal; replicación	Mayor consumo de memoria; menor desempeño en transacciones complejas
Apache Kafka	Alta capacidad de ingesta; baja latencia; tolerancia a fallos; durabilidad de mensajes	Complejidad de configuración; curva de aprendizaje pronunciada
Apache Spark	Procesamiento distribuido; <i>batch</i> y <i>streaming</i> ; ecosistema amplio; alto rendimiento	Consumo intensivo de memoria; requiere ajuste fino de recursos

Tabla 2.1: Resumen de ventajas y desventajas de las tecnologías utilizadas

2.5 Estado del Arte y Técnica

En finanzas, la disponibilidad de datos de mercado en tiempo real y su correcta gestión son factores clave para ejecutar estrategias de *trading* con baja latencia y buena calidad de información. La literatura subraya desde hace años que la precisión y la velocidad son determinantes en el ciclo captura–proceso–decisión para *market data tick by tick*, especialmente en contextos de *high-frequency trading* [1]. Este foco en tiempos cortos y trazabilidad impulsa arquitecturas que separan con claridad la ingestión, el procesamiento y el almacenamiento histórico.

En ese marco, los *Data Lakes* se consolidaron como una opción flexible para integrar datos crudos, estructurados y analíticos en una misma plataforma. Liu et al. proponen un *Data Lake* multinivel para analítica por ventanas en tiempo real, mostrando cómo organizar capas *raw*, estructuradas y de resultados para equilibrar preservación del detalle, consistencia y rendimiento de consulta [2]. Este enfoque respalda la decisión de este proyecto de mantener tres zonas —**Raw**, **Structured** y **Analytics**— con trazabilidad de extremo a extremo y acceso eficiente para distintos perfiles de consumo.

Respecto del procesamiento en tiempo real, dos motores ampliamente utilizados son Apache Spark y Apache Flink. El estudio comparativo de Jaime revisa diferencias prácticas entre ambos, destacando que Spark ofrece un ecosistema maduro y una curva de adopción más suave (en especial cuando se combinan cargas por lotes y en *streaming*),

mientras que Flink tiende a sobresalir en *streaming* puro con muy baja latencia a costa de mayor complejidad operativa [3]. Considerando los objetivos del proyecto —limpieza, transformación y cálculo de KPIs en tiempo real junto con reprocesos por lotes— se privilegia Spark para reducir complejidad de operación y facilitar el mantenimiento, manteniendo la posibilidad de evaluar Flink si surgen casos futuros con requisitos de latencia más estrictos.

En el plano de servicios gestionados en la nube, Velickovska y Gusev comparan alternativas de *streaming* en AWS a partir de un caso de uso con señales ECG, analizando costos, facilidad de integración y desempeño de servicios nativos (por ejemplo, Amazon Kinesis) [4]. Sus resultados muestran que las opciones administradas pueden acelerar el despliegue y simplificar la operación cuando la organización ya trabaja en el ecosistema AWS, aunque esto introduce dependencia de proveedor. Para este trabajo se optó por componentes *open-source* (Kafka, Spark, MongoDB) por su portabilidad y control de la arquitectura, sin descartar una futura migración híbrida o administrada si se prioriza tiempo de puesta en marcha sobre control y costo total de propiedad.

La revisión de las tecnologías muestra que un Data Lake con zonas Raw, Structured y Analytics es adecuado para mantener trazabilidad y buen rendimiento [2]; Spark resulta práctico para combinar batch y streaming, mientras que Flink queda como alternativa para latencias aún menores [3]; y, aunque los servicios gestionados de AWS agilizan el despliegue, partimos con componentes open-source por control y portabilidad [4]. Estas decisiones responden a la necesidad de precisión y velocidad en entornos de negociación electrónica [1]. A continuación, se presentan los objetivos del proyecto y la integración propuesta.

3 Objetivos

3.1 Objetivo general

Crear una plataforma escalable que permita almacenar una gran cantidad de market data, diseñada para la recuperación rápida de la información.

3.2 Objetivos específicos

- OE1) Research de herramientas:** Buscar las herramientas que sean aplicables a nuestro contexto de desafío.
- OE2) Arquitectura del sistema:** Modelar una arquitectura que se adecue a las necesidades descritas por el cliente.
- OE3) Ingestión de datos:** Desarrollar un sistema capaz de recibir y almacenar datos de mercado, tick by tick, asegurando que no haya pérdida de información durante el proceso.
- OE4) Almacenamiento eficiente y escalable:** Implementar una solución de almacenamiento que sea eficiente y escalable, capaz de manejar grandes volúmenes de datos de alta frecuencia, utilizando tecnologías de almacenamiento en la nube y distribuidas.
- OE5) Procesamiento en tiempo real:** Implementar una solución dentro del sistema que permita procesar la información en tiempo real, garantizando una latencia mínima de respuesta y soporte grandes volúmenes de datos de alta frecuencia.

4 Diseño de la Solución

4.1 Requisitos del Sistema

4.1.1 Requisitos funcionales

RF1 Ingestión de Datos en Tiempo Real: El sistema debe ser capaz de recibir y procesar datos de mercado tick by tick con baja latencia, garantizando ingestión continua de datos sin demoras ni pérdidas significativas.

RF2 Almacenamiento Eficiente: El sistema debe permitir el almacenamiento de grandes volúmenes de datos de manera eficiente, utilizando MongoDB, permitiendo almacenar los datos en su forma cruda para su futura extracción y que puedan ser utilizados para análisis futuros.

RF3 Escalabilidad: El diseño del sistema debe permitir el escalamiento horizontal y vertical para adaptarse al manejo de grandes volúmenes de datos sin comprometer el rendimiento del sistema.

RF4 Disponibilidad y Fiabilidad: El sistema debe estar disponible en todo momento con alta tolerancia a fallos para asegurar la continuidad del servicio.

RF5 Interoperabilidad: El sistema debe ser compatible con otras aplicaciones y sistemas utilizados por la corredora, permitiendo la integración sin problemas con otras herramientas y plataformas.

RF6 Validación y Limpieza de Datos: Implementar un proceso de validación y depuración de datos con Apache Spark que sea previo al almacenamiento. Este procedimiento garantiza la calidad y precisión de la información, de modo que solo datos válidos y limpios queden disponibles para análisis posteriores.

4.1.2 Requisitos no funcionales

RNF1 Rendimiento: El sistema debe ser capaz de procesar y analizar grandes volúmenes de datos de manera rápida y eficiente, garantizando tiempos de respuesta bajos incluso con cargas elevadas de datos.

RNF2 Disponibilidad: El sistema debe estar disponible en todo momento, con una tolerancia a fallos mínima para garantizar la correcta operación de la market data.

RNF3 Mantenibilidad: Con la integración de MongoDB, Apache Spark y Apache Kafka el sistema debe ser modular para que sea fácil de mantener y actualizar, facilitando la gestión de versiones, la depuración de errores y la optimización del sistema sin afectar el rendimiento general.

RNF4 Monitorización en Tiempo Real: El sistema debe contar con un dashboard, aunque no sea una funcionalidad obligatoria, para la posible visualización y monitoreo de métricas en tiempo real, dándole un valor adicional a los usuarios para que tomen decisiones más informadas.

RNF5 Resiliencia y Recuperación: Ante fallas, el sistema debe reanudar automáticamente el procesamiento y envío de datos para evitar la pérdida de información.

4.1.3 Requisitos de interfaces

La tabla 4.1 sintetiza los **requisitos de interfaz** del sistema. En ella se describen los principales *eventos* de interacción con la *market data* (p. ej., recepción y solicitud de datos), el *iniciador* que los dispara (fuente externa o aplicación cliente), los *parámetros* mínimos que deben proporcionar las partes involucradas y la *respuesta* esperada del sistema.

Evento	Descripción	Iniciador	Parámetros	Respuesta
Recepción de datos nuevos	El Data Lake recibe nueva market data para almacenamiento.	Llega un nuevo conjunto de datos al sistema.	Tipo de datos, Fuente de datos, Formato de datos	Datos almacenados y metadatos se actualizan automáticamente.
Solicitud de datos	Una aplicación cliente solicita datos específicos desde el sistema.	Una petición es enviada al sistema.	ID de solicitud, Rango de fechas, Filtros específicos	Solicitud procesada y datos enviados al solicitante.

Tabla 4.1: Requisitos de interfaces

4.1.4 Requisitos de ambiente

Hardware de desarrollo

- Servidor con capacidades de almacenamiento adecuadas para manejar grandes volúmenes de datos.
- Conexión de red estable y rápida para garantizar la transferencia eficiente de datos en el servidor del sistema.

Software de desarrollo

- Herramientas que permitan desarrollar y administrar el sistema de almacenamiento y procesamiento de datos.
- Sistema Operativo: Linux.
- **MongoDB 7.0.15:** Base de datos seleccionada por su flexibilidad en esquemas de datos y su capacidad para manejar grandes volúmenes de datos no estructurados de forma eficiente. Esto es crucial para la ingestión de datos en tiempo real y para manejar la variedad de fuentes de datos que esperamos procesar.
- **Java 22.0.2:** Lenguaje de programación seleccionado por su robustez y portabilidad entre sistemas operativos, lo que facilita la estructuración, el desarrollo y el mantenimiento del backend. Además, cuenta con un amplio ecosistema de librerías y *frameworks*.
- **Apache Spark 4.0.0-preview2:** Motor de procesamiento distribuido en memoria para *batch* y *streaming*. Permite ejecutar transformaciones en tiempo real sobre grandes volúmenes de datos con baja latencia y alta escalabilidad, permitiendo analizar grandes volúmenes de datos con rapidez y eficiencia.
- **Apache Kafka 3.8.0:** Plataforma de mensajería distribuida resistente a fallas y mantiene independientes a los componentes que envían y reciben datos. Permite la transmisión de datos en tiempo real entre los distintos componentes del sistema y asegura que la ingesta sea continua y sin pérdidas.
- **Apache ZooKeeper:** Servicio de coordinación distribuida utilizado para administrar el clúster de Kafka. Proporciona sincronización entre nodos, mantiene la configuración compartida y gestiona la elección de líderes, contribuyendo a la estabilidad y disponibilidad del sistema.
- **Docker 26.1.4:** Plataforma de contenedores para empaquetar y desplegar los servicios del sistema de forma reproducible. En el proyecto se usa en archivo `.yaml` para levantar ZooKeeper y Kafka, definiendo parámetros críticos como puertos, conexiones, volúmenes y parámetros de replicación; esto simplifica la administración de los servicios y la replicabilidad del entorno.
- **Spring Framework 3.3.4:** Framework (marco de desarrollo) para Java que permite gestionar la conexión y poder organizar el backend mediante inyección de dependencias y componentes reutilizables, facilitando una arquitectura modular, mantenible y escalable.

4.1.5 Perfiles de usuario

La Tabla 4.2 presenta los **perfiles de usuario** considerados para el sistema, caracterizados por su contexto socioeconómico y cultural, rol ocupacional, rango etario y rasgos relevantes (cognitivos y conductuales) junto con características claves para cada perfil presentado. Esta segmentación guía el diseño de las interfaces y la priorización de los requisitos.

Perfil	Socioeconómico y cultural	Ocupacional	Etario	Características físicas, fisiológicas, psicológicas
Analistas de Datos	Usuarios con formación en ciencias, matemáticas o economía	Profesionales que analizan grandes volúmenes de datos para identificar tendencias y patrones	25-45 años	Alto nivel de habilidades analíticas, capacidad de concentración, trabajo bajo presión
Traders	Usuarios con formación en finanzas y economía	Profesionales que compran y venden activos financieros	25-50 años	Toma de decisiones rápidas, alta tolerancia al riesgo, alto nivel de estrés
Desarrolladores y Administradores de Sistemas	Usuarios con formación en informática y ciencias de la computación	Profesionales encargados del desarrollo y mantenimiento del Data Lake	22-45 años	Altas habilidades técnicas, capacidad de resolución de problemas

Tabla 4.2: Perfiles de Usuario

4.2 Arquitectura del Sistema

La arquitectura del sistema se organiza en un conjunto de componentes que permiten **capturar, procesar, ingestar y almacenar** grandes volúmenes de datos en **tiempo real**. Para describir su funcionamiento desde distintos niveles, se presentan tres diagramas complementarios: (i) *diagrama de contexto*, que expone los actores y sistemas externos que interactúan con la plataforma; (ii) *esquema general del sistema*, que resume el flujo de información a través de las capas de ingesta, procesamiento y almacenamiento; y (iii) *diagrama de arquitectura*, que detalla los componentes internos principales y sus interconexiones.

4.2.1 Diagrama de contexto

El diagrama de contexto (Figura 4.1) representa al sistema de manera general y como interactúan los principales componentes o entidades en todo el flujo del sistema. Se muestran las diferentes *Fuentes de datos*, como los log files (.log) o los Data stream, de donde se extrae la *market data* que alimenta la plataforma, el Data Lake como núcleo central donde se almacenan los datos en su colección correspondiente, y los consumidores finales (traders, analistas financieros, etc.) que acceden a los datos procesados y organizados. Este nivel de abstracción nos proporciona una vista general del sistema y además permite visualizar los límites del sistema y los actores involucrados en su operación.



Figura 4.1: Diagrama de contexto del sistema.

4.2.2 Esquema general del sistema

Con el fin de clarificar cómo se gestionan los datos en el sistema, el esquema general (Figura 4.2) presenta los bloques funcionales. En él se aprecia que las fuentes de información —tanto archivos históricos como flujos en tiempo real— son recibidas por el módulo de *Data Ingestion*, responsable de extraer, validar y preparar los datos antes de su ingestión.

A continuación, los datos son ingestados en el *Data Lake*, que actúa como repositorio central, permite el almacenamiento y acceso inmediato a los datos. Este orden facilita un acceso eficiente y rápido para distintos perfiles de consumo como analistas, *traders*, científicos de datos, procesos automatizados, algoritmos y mecanismos de control de acceso permiten la disponibilidad de la información para que los consumidores puedan realizar análisis en tiempo real con datos fiables.

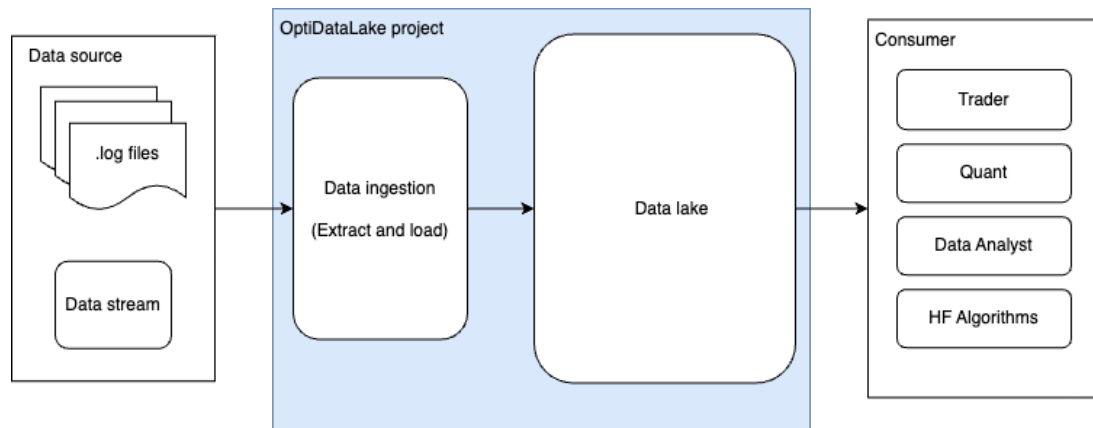


Figura 4.2: Esquema general del sistema y flujo de información.

4.2.3 Diagrama de arquitectura

El mayor nivel de detalle se aprecia en el diagrama de arquitectura (Figura 4.3), el cual representa el recorrido interno de los datos y la interacción entre los módulos principales del sistema. A partir de las diferentes fuentes de información, compuestas por *Data Streams* y archivos de registro (.log files), los eventos ingresan al módulo de *Data Ingestion*. En esta etapa inicial los datos son recibidos, validados y preparados para almacenamiento en el *Data Lake*.

Dentro del *Data Lake*, la información se organiza en diferentes colecciones cada una con un propósito específico:

- **Colección Raw/Landing:** Almacena los datos en su estado original, garantizando que se mantenga una copia fiel de la fuente, ya que no se realiza ninguna modificación a estos.
- **Colección Structured Data:** Transforma y organiza los datos en estructuras y formatos consistentes que facilitan consultas rápidas, eficientes y permiten análisis y consultas recurrentes.
- **Colección Analytics:** Concentra métricas, indicadores, KPIs y resultados derivados de los procesos analíticos aplicados sobre los datos estructurados.

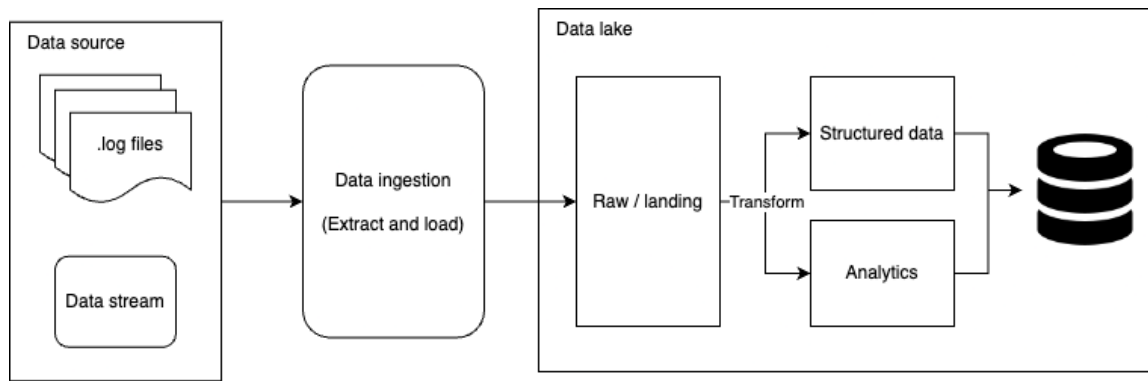


Figura 4.3: Diagrama de arquitectura detallada del sistema.

Este modelo evidencia cómo la arquitectura permite separar responsabilidades y mantener un flujo ordenado desde la ingesta hasta la realización de procesos analíticos. Asimismo, el diseño modular asegura escalabilidad y flexibilidad para incorporar nuevas fuentes de información, tecnologías o ampliar las capacidades de análisis en el futuro.

4.2.4 Definición y Diseño de Módulos

Módulo Data Ingestion (DI)

El Módulo Data Ingestion (DI) constituye la puerta de entrada del sistema: se conecta a las fuentes de *market data* en tiempo real (p. ej., Binance) y estandariza cada dato antes de enviarlo al flujo interno. Su ingesta alimenta directamente al módulo **Raw / Landing (RD)**, donde el dato se almacena tal como llegó, este módulo se encarga de su gestión y distribución.

Propósito	Integrarse con la(s) API/stream de market data (API de Binance), extraer gran volumen de datos en tiempo real y publicar los datos hacia el módulo RD para su posterior gestión y distribución.
Alcance	Se encarga de establecer y mantener la conexión directa hacia la API de Binance que se utiliza como fuente de datos para la market data que se almacenará, recibe los datos sin interrupciones y verifica que tengan el formato esperado antes de enviarlos al módulo RD para que no existan errores al publicarlos.
Dependencias	Depende directamente de la API de Binance (fuente de datos), las credenciales de acceso y el módulo RD que se encarga de publicar los datos para el resto de componentes del sistema.
Supuestos	La API de Binance estará disponible en cada momento y entregará datos válidos para su posterior almacenamiento; el módulo cuenta con la capacidad de procesar al menos 50 mensajes por segundo; los mensajes incluyen identificadores que evitan duplicados.
Restricciones	No guarda datos de forma permanente ni realiza cálculos de negocio; sólo estandariza y reenvía. Si la conexión falla, intenta reconectar; los mensajes que no cumplen el formato mínimo se deben enriquecer para adoptar el formato aceptable (formato JSON).

Tabla 4.3: Descripción del módulo Data Ingestion (DI)

Módulo Raw / Landing (RD)

El módulo **Raw/Landing (RD)** recibe, en tiempo real, los mensajes que publica el **Módulo Data Ingestion (DI)** y los almacena *tal cual llegaron*. Desde aquí el dato puede seguir dos caminos: pasar al **Módulo Transform (T)** para su limpieza y estandarización o permanecer como respaldo histórico dentro del **Módulo Data Lake (DL)**.

Propósito	Operar como intermediario entre DI y el resto del sistema: recibe los datos crudos y los envía tanto al Data Lake (zona Raw/Landing) para su almacenamiento como al módulo Transform (T) para su procesamiento, manteniendo una copia fiel y metadatos básicos de trazabilidad.
Alcance	Recibe los mensajes en tiempo real desde el DI y mantiene una copia fiel de cada mensaje recibido con los metadatos mínimos para rastrear su origen y su momento de ingreso, sirviendo como respaldo histórico y punto de partida para los procesos posteriores (T y DL).
Dependencias	Trabaja directamente con el Módulo DI del cual recibe mensajes en tiempo real y también con el Módulo de Data Lake para el almacenamiento de mensajes crudos y el Módulo Transform para encaminar los mensajes a su procesamiento.
Supuestos	Los datos provenientes de DI llegan en formato <i>JSON</i> válido con la estructura correcta, existe conectividad estable y tasa de escritura acorde al volumen de datos para sostener un flujo continuo.
Restricciones	No almacena datos de forma permanente ni realiza transformaciones o cálculos de negocio; su función es actuar como componente de tránsito y distribución hacia T y DL .

Tabla 4.4: Descripción del módulo Raw / Landing (RD)

Módulo Transform (T)

El módulo **Transform (T)** recibe los datos crudos desde el módulo **RD** y los convierte en datos limpios y estructurados según las reglas del sistema. Para ello valida campos, corrige tipos y elimina duplicados. Los mensajes resultantes se publican hacia el módulo **SD** donde quedan listos para ser consultados, y al módulo **A**, para el cálculo de métricas y KPIs; además, se almacenan en el módulo **DL** como parte de la zona estructurada.

Propósito	Convertir los datos crudos provenientes de RD en información estandarizada y estructurada para su almacenamiento en el módulo SD y además realizar cálculos analíticos que serán almacenados en el módulo A como KPIs, métricas, etc.
Alcance	Este módulo toma los datos crudos y los somete a transformaciones para dejarlos con un formato estructurado listos para ser consultados. Además, puede realizar cálculos analíticos que sean de apoyo como promedios, volúmenes y marcas de apertura/cierre que sirven como recurso para las métricas consolidadas en el módulo A .
Dependencias	Depende directamente del módulo RD como origen de los datos crudos y del módulo del Data Lake (DL) como destino para almacenar tanto las métricas y analíticas calculadas como los mensajes estructurados.
Supuestos	Los datos crudos recibidos desde RD son válidos y contienen los campos obligatorios e información necesaria para su transformación y posterior almacenamiento (datos estructurados en módulo SD y analíticas en módulo A). El módulo debe poder procesar los datos en tiempo real para sostener un flujo continuo, rápido y eficiente.
Restricciones	Sólo se aceptarán datos que luego de la transformación cumplan con el esquema definido para datos estructurados, que fueron definidos para el sistema; cualquier resultado analítico deberá tener el formato establecido para su consulta y correcta visualización.

Tabla 4.5: Descripción del módulo Transform (T)

Módulo Structured Data (SD)

El módulo **Structured Data (SD)** recibe los datos transformados por **T** y los organiza para almacenarlos en un formato estructurado. Su objetivo principal es asegurar su disponibilidad en las colecciones correspondientes, de modo que puedan consultarse y visualizarse, y servir como base para cálculos analíticos, todo conforme a los requerimientos del sistema.

Propósito	Almacenar dentro del Data Lake la información ya transformada en un formato estructurado, asegurando que esté disponible para análisis futuros y consultas.
Alcance	Recibe los datos transformados del módulo Transform (T) y la dispone en la capa de datos estructurados del Data Lake, organizándola en colecciones listas para ser consultadas por usuarios y procesos.
Dependencias	Requiere los datos procesados por el módulo T y del Data Lake para poder almacenar los datos organizados en sus colecciones.
Supuestos	Los datos transformados respetan el esquema definido; al quedar almacenados en el Data Lake su disponibilidad para usos posteriores, como consultas o analíticas, queda garantizada.
Restricciones	Los datos estructurados deben cumplir con el esquema definido y estar organizados; una vez que son almacenados en el Data Lake, no se modifican, sólo se permite la lectura de ellos.

Tabla 4.6: Descripción del módulo Structured Data (SD)

Módulo **Analytics (A)**

El Módulo **Analytics (A)** se encarga de realizar los cálculos a partir de los datos transformados que entrega el módulo **Transform (T)** (y, cuando sea necesario, de mensajes del módulo **SD**). Los resultados se almacenan en la capa *Analytics* del módulo **Data Lake (DL)**, quedando disponibles para consultas, dashboards y análisis posteriores.

Propósito	Producir métricas y KPIs confiables a partir de datos transformados, y dejarlos almacenados en la capa <i>Analytics</i> del Data Lake para su consulta y análisis.
Alcance	Consume los datos provenientes del módulo T para calcular analíticas y KPIs relevantes para el negocio y almacena los resultados en la capa Analytics del Data Lake (DL) . Los resultados se organizan en colecciones y están listos para ser usados en reportes, servicios y consultas de usuarios.
Dependencias	Depende directamente del módulo Transform (T) como fuentes de datos y del Data Lake para almacenar las analíticas generadas.
Supuestos	Los datos de entrada son recibidos con un formato adecuado para el cálculo de métricas y las analíticas que se generan cumplen tanto con el formato y los requisitos definidos por el sistema.
Restricciones	Las métricas deben almacenarse en la capa Analytics respetando la estructura definida y quedar indexadas para consultas eficientes. Una vez publicadas, son de sólo lectura y su acceso debe cumplir con tiempos de respuesta bajos.

Tabla 4.7: Descripción del módulo Analytics (A)

Módulo **Data Lake (DL)**

El módulo **Data Lake (DL)** es el repositorio central del sistema. Integra las tres zonas de datos —**Raw** (desde RD), **Structured** (desde SD) y **Analytics** (desde A)— y está diseñado para almacenar y organizar la información que entregan dichos módulos. Los datos se clasifican en crudos, analíticos y estructurados, lo que garantiza su disponibilidad para consultas posteriores y que también quede un registro histórico de la data.

Propósito	Almacenar y organizar todos los datos del sistema incluyendo datos crudos, estructurados y analíticos, garantizando su disponibilidad para consultas y análisis posteriores.
Alcance	Recibir datos desde los módulos RD , SD y A ; almacenarlos según su nivel de procesamiento: estructurados, analíticos y crudos.
Dependencias	Depende de los datos recibidos desde los módulos: T , RD (datos crudos), SD (datos estructurados) y A (datos analíticos) para poder cumplir con su rol de repositorio principal.
Supuestos	Los datos que ingresan al Data Lake ya cumplen las reglas de los módulos previos (formato, validaciones y estructura) y se garantiza que los datos sean almacenados de manera histórica; el DL mantiene su disponibilidad para consultas y análisis futuros, con esquema conocido.
Restricciones	La información almacenada en el Data Lake es inmutable (sólo lectura); se debe garantizar que tanto el acceso como las consultas a los datos sean eficientes y rápidas aún con grandes volúmenes de datos.

Tabla 4.8: Descripción del módulo Data Lake (DL)

4.2.5 Matriz de requisitos funcionales y componentes

La matriz siguiente cruza los **requisitos funcionales** (RF1–RF6) con los **módulos** de la solución (DI, RD, T, SD, A y DL). Una marca “X” indica que el módulo es *responsable directo* del cumplimiento del requisito; cuando un requisito aparece en varias columnas, la responsabilidad es *compartida*. Esta vista permite verificar cobertura (cada RF tiene al menos un responsable), detectar vacíos o solapamientos, y orientar el plan de pruebas, la asignación de tareas y el seguimiento de hitos.

	DI	RD	T	SD	A	DL
RF1	X					
RF2		X				X
RF3		X		X	X	X
RF4						X
RF5						X
RF6		X	X	X	X	X

Tabla 4.9: Matriz de requisitos funcionales y módulos

4.3 Estructura de Datos en el Data Lake

El **Data Lake (DL)** almacena grandes volúmenes de datos clasificados en tres **tipos de datos**: **Raw Data**, **Structured Data** y **Analytics**. En términos generales, *Raw Data* preserva el mensaje tal como llega; *Structured Data* corresponde a los datos ya transformados y listos para consulta; y *Analytics* reúne resultados derivados como métricas y KPIs. Esta clasificación por tipos asegura trazabilidad de extremo a extremo, mantiene la fidelidad del dato original, garantiza calidad y consistencia para las consultas y favorece el rendimiento y la escalabilidad del sistema. En las subsecciones siguientes se detalla el contenido y uso de cada tipo.

4.3.1 Datos Crudos (Raw Data)

Los **datos crudos** representan exactamente lo recibido, sin transformaciones, desde la fuente de datos (API de Binance). Se conservan de tal manera que no presentan alteraciones, siempre preservando la integridad del dato para que sirva como almacenamiento histórico y pueda ser utilizada para procesos de validación.

Campos obligatorios de la colección *raw_data*:

- **eventType**: tipo de evento registrado (por ejemplo, *trade* para una operación de mercado).
- **eventTime**: instante en que el emisor reporta el evento (marca temporal del suceso).
- **symbol**: identificador del activo financiero negociado (p.ej., BTCUSDT, ETHUSDT, XRPUSDT).
- **tradeId**: identificador único de la transacción, útil para trazabilidad e idempotencia.
- **price**: precio al cual se ejecutó la operación.
- **quantity**: cantidad/volumen de activos financieros transaccionados.
- **tradeTime**: hora exacta de ejecución de la transacción.
- **buyerIsMaker**: indicador booleano que señala si el comprador actuó como *market maker* del libro de órdenes (creador de la orden) esto es relevante para análisis de profundidad y liquidez en el mercado.
- **marketMaker**: indicador que determina si la operación fue iniciada por un *market maker*, permitiendo medir participación de proveedores de liquidez.

Estos campos son imprescindibles para la posterior normalización, detección de duplicados y cálculo de métricas en los procesos posteriores.

Ejemplo de *raw_data*. A continuación, se presenta un ejemplo real de cómo se almacenan los datos crudos en formato JSON con sus respectivos campos:

```
{
  "_id": "6Bf60c252ec4ed44496fa5f1",
  "eventType": "trade",
  "eventTime": 4427400997866,
  "symbol": "BTCUSDT",
  "tradeId": 3678884281,
  "price": 65123.73,
  "quantity": 0.0406,
  "tradeTime": 1727400633897,
  "buyerIsMaker": true,
  "marketMaker": true
}
```

Listing 4.1: Ejemplo de Datos Crudos (Raw Data)

4.3.2 Estructurados

Los datos crudos (raw data) pasan por un proceso de transformación y limpieza para ser almacenados en una estructura más manejable y que facilite la consulta eficiente. Estas estructuras incluyen los datos sobre los productos financieros y los mercados financieros.

- **Productos Financieros:** Información sobre los activos financieros negociados en los mercados.
 - **producto_id:** Identificador único del producto financiero.
 - **nombre_producto:** Nombre del activo, por ejemplo, **Bitcoin / Tether**.
- **Ejemplo de datos estructurados: Productos Financieros:**

```
{
  "producto_id": "BTC/USDT",
  "nombre_producto": "Bitcoin / Tether"
}
```

Listing 4.2: Ejemplo de Datos Estructurados: Productos Financieros

- **Mercados Financieros:** Información sobre los mercados donde se negocian los productos financieros.
 - **mercado_id:** Identificador único del mercado.
 - **nombre_mercado:** Nombre del mercado, por ejemplo, **Binance Exchange**.

- **tipo_mercado**: Tipo de mercado, como **Spot**.
- **Ejemplo de datos estructurados: Mercados Financieros:**

```
{  
  "mercado_id": "Binance",  
  "nombre_mercado": "Binance Exchange",  
  "tipo_mercado": "Spot"  
}
```

Listing 4.3: Ejemplo de Datos Estructurados: Mercados Financieros

4.3.3 Analytics (KPIs y Métricas Derivadas)

En esta categoría se almacenan los **KPIs** derivados de los datos de mercado. Estos KPIs permiten realizar análisis avanzados sobre el comportamiento de los activos financieros en los mercados. Los KPIs se calculan utilizando los datos que son extraídos de Binance y se almacenan para su análisis.

- **KPIs de Market Data**: Indicadores clave calculados que pueden ser útiles para realizar análisis:
- **cryptocurrency**: Identificador del activo financiero, como **BTCUSDT**, **ETHUSDT** o **XRPUSDT**.
 - **time_window**: Intervalo de tiempo durante el cual se recopilan y calculan los KPIs.
 - **volumen_total**: Cantidad total de activos transaccionados en el intervalo de tiempo.
 - **promedio_precios**: Media aritmética de los precios de cierre durante el periodo.
 - **volatilidad**: Medida de la variabilidad del precio en el intervalo de tiempo.
 - **movimiento_precios**: Diferencia entre el precio de apertura y el precio de cierre durante el intervalo de tiempo.

■ Ejemplo de datos analíticos (KPIs):

```
{
  "_id": "670fcb1fa6dcb1b4896ec5c",
  "cryptocurrency": "BTCUSDT",
  "time_window": "2024-10-16T14:18:00.000+00:00",
  "volumen_total": 53.649190000000059,
  "promedio_precios": 68222.0110529753,
  "volatilidad": 20.61907742339122,
  "movimiento_precios": 55.429999999993015
}
```

Listing 4.4: Ejemplo de Datos Analíticos (KPIs)

La separación en *Raw*, *Structured* y *Analytics* ordena el manejo de la información y hace más simple trabajar con ella. Se conserva el dato original para tener registro histórico de la información, se dispone de datos limpios y estructurados para consultas y se guardan métricas listas para análisis. Con esta organización el Data Lake puede asegurar escalabilidad, incorporar nuevas fuentes de datos y adaptarse a cambios del negocio sin romper lo que ya existe, siempre listo para un ambiente que es dinámico.

4.3.4 Esquema UML del Modelo de Datos

El diagrama UML de la Figura 4.4 muestra, de forma general, la relación entre los tres tipos de datos del **Data Lake**. Las colecciones de **Raw Data** actúan como origen común; desde ellas se derivan y enlazan las colecciones de **Structured Data**, que organizan la información para su consulta; y, sobre esa base ya transformada y validada, se calculan y almacenan en **Analytics**, los resultados agregados por ventanas de tiempo. En síntesis: *Raw* alimenta a *Structured* y ambos proveen datos a *Analytics*, manteniendo trazabilidad de extremo a extremo.

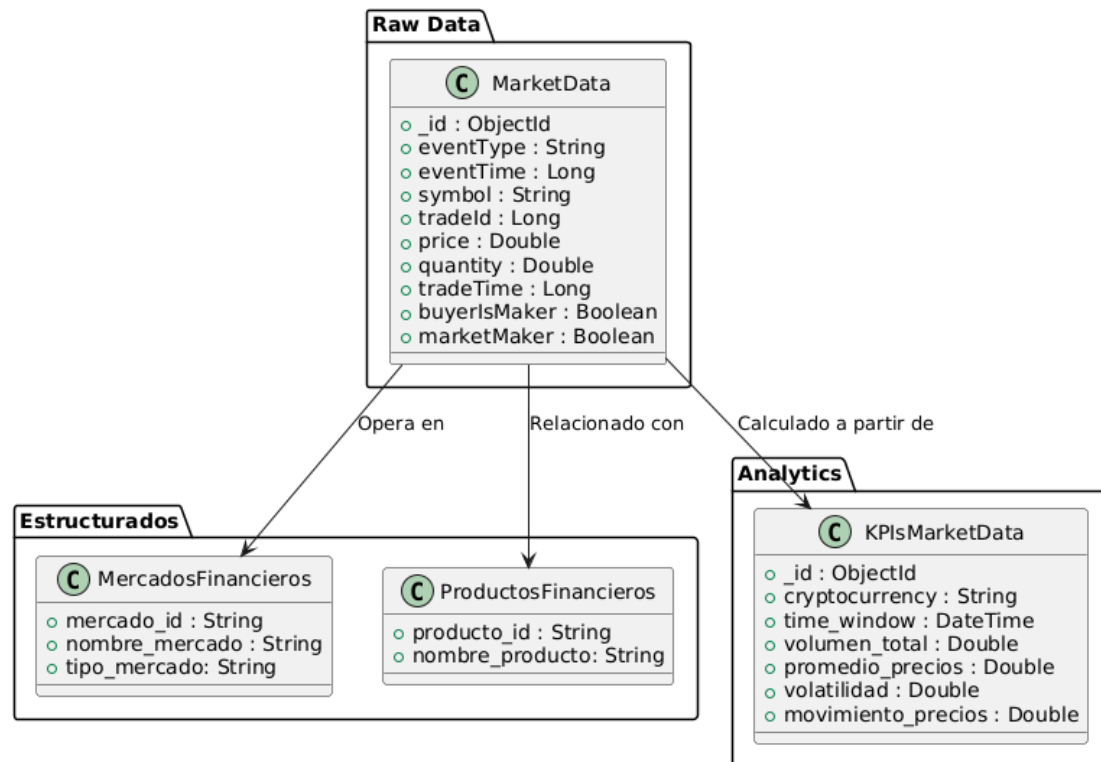


Figura 4.4: Esquema UML del modelo de datos implementado en el Data Lake.

4.4 Diseño de Interfaces

4.4.1 Diagrama Entidad–Relación

El diagrama Entidad–Relación de la Figura 4.5 resume cómo se enlazan los actores y componentes del sistema: quién solicita información, por dónde circulan los datos y en qué forma se entregan los resultados. La lectura general es: *el consumidor formula una consulta, el **Data Lake** la resuelve con la información que recibe de las fuentes de datos, y el resultado se entrega como reporte.*

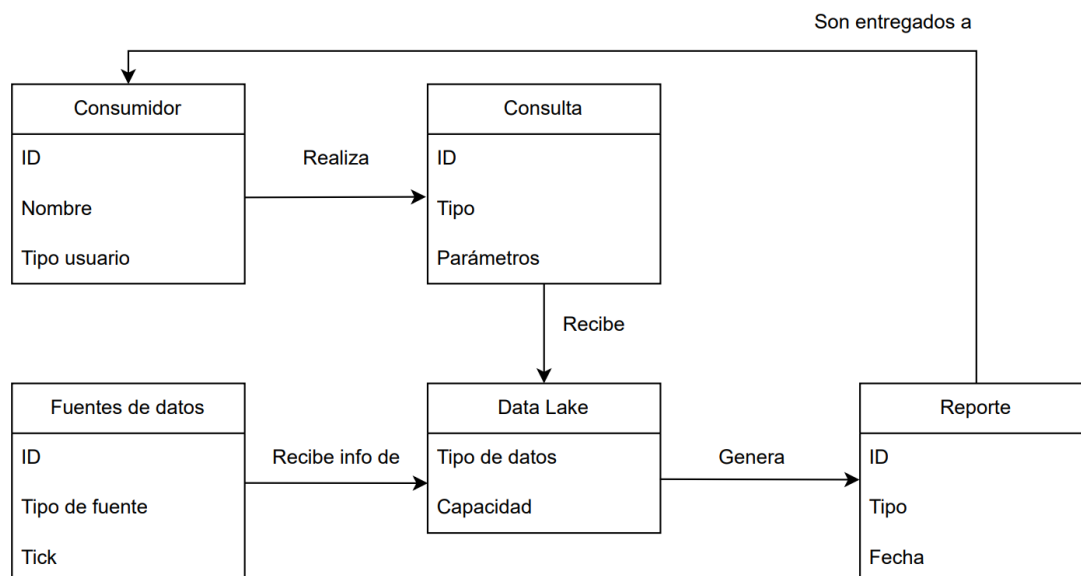


Figura 4.5: Diagrama Entidad-Relación del sistema.

A continuación se sintetizan las entidades y los vínculos principales del diagrama:

- **Consumidor**: usuario o sistema que requiere información. Se identifica por un *ID*, un nombre y un tipo de usuario (define su rol y permisos). *Relación*: **realiza** consultas.
- **Consulta**: Se modela la interacción de los consumidores con el sistema, se realiza una solicitud que describe qué información se necesita. Incluye un *ID*, el *tipo* (p. ej., histórica, en tiempo real) y *parámetros* (rango de fechas, símbolo, etc.). *Relación*: es **recibida** y atendida por el **Data Lake**.
- **Data Lake**: repositorio central donde residen los distintos tipos de datos (raw, estructurados y analíticos), se encarga de recibir y procesar las consultas, devolviendo la información requerida según el tipo de consulta. *Relaciones*: **recibe información** de las **Fuentes de Datos** y **genera** los **Reportes** que responden a cada consulta.
- **Fuentes de Datos**: Hace referencia a los orígenes de la información que alimenta al sistema (p. ej., APIs de mercado). Cada fuente tiene un *ID*, un *tipo de fuente* y un *tick* o marca temporal asociada a la generación de eventos. *Relación*: **envía** datos al **Data Lake**.
- **Reportes**: son la respuesta que produce el **Data Lake** a una consulta. Cada reporte deja trazado qué se pidió y cuándo se generó, incluyendo al menos su *ID*, el *tipo* (analítico o resumen) y la *fecha/hora* de emisión. *Relación*: Una **Consulta** puede *generar* uno o varios **Reportes**, que luego son *entregados* al **Consumidor**.

En conjunto, este modelo muestra un flujo eficiente y escalable: los datos entran por las fuentes, se consolidan en el **Data Lake**, se consultan según la necesidad del usuario y se devuelven como reportes listos para uso operativo o análisis.

4.4.2 Diagrama de Flujo del Sistema

La Figura 4.6 resume el recorrido que siguen los datos: desde que se capturan en la fuente, pasan por ingestión y procesamiento, quedan almacenados en el **Data Lake** y finalmente se ponen a disposición de los usuarios. El diagrama destaca las etapas clave: ingesta, validación, transformación, almacenamiento y consumo.

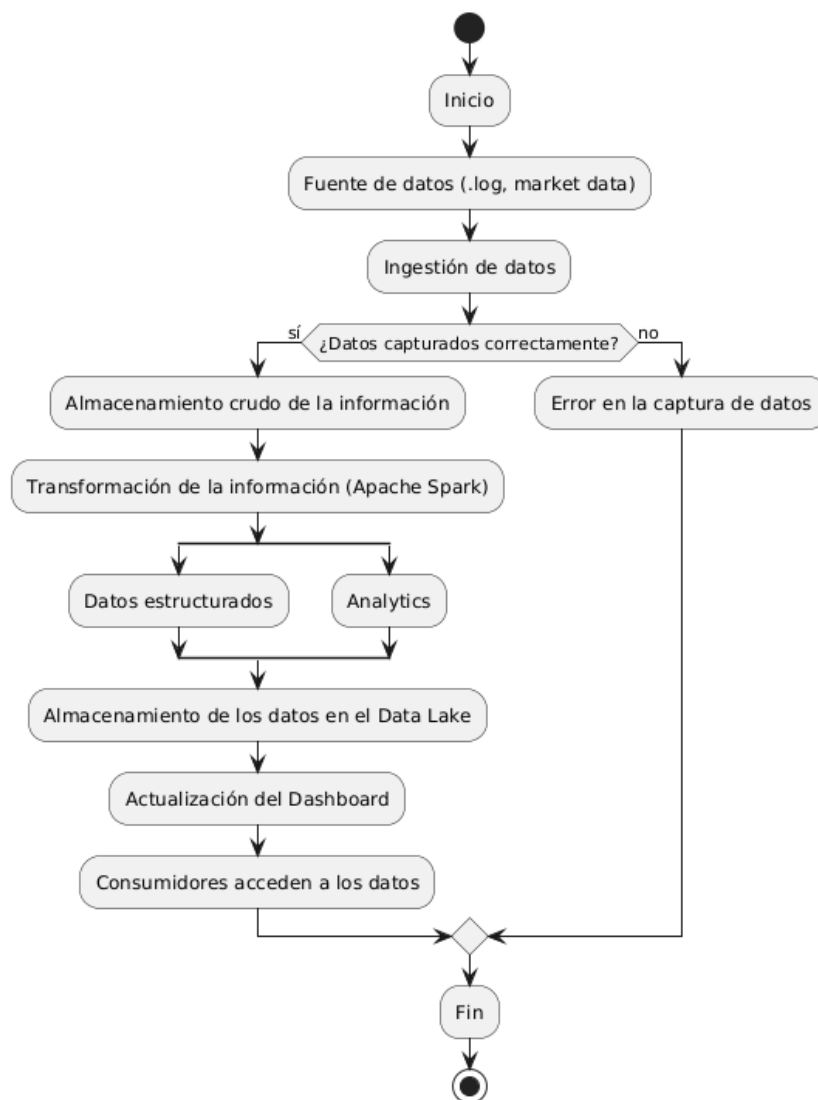


Figura 4.6: Diagrama de flujo del sistema.

A continuación se describen las etapas del diagrama de flujo:

1. **Inicio:** Marca el comienzo del proceso de manejo de datos en el sistema.

2. **Fuentes de datos:** Entrada de eventos desde diferentes fuentes de datos como archivos de *logs* y/o *market data* en tiempo real (p. ej., API de Binance).
3. **Ingestión de datos:** Captura y preparación inicial de los datos para que puedan ser almacenados en el sistema y estén disponibles para posteriores consultas o procesos.
4. **Validación básica:** Se valida si los datos llegan correctos, si es así continúa el flujo; de lo contrario se registra un *error de captura* y se redirigen a manejo de errores.
5. **Almacenamiento crudo:** Almacenamiento del mensaje tal cual llega en su respectiva colección (*Raw/Landing*) dentro del Data Lake, para mantener data histórica y respaldo.
6. **Transformación de los mensajes:** los datos crudos se parsean y se convierten a formatos estructurados definidos para el sistema. En esta etapa también se calculan métricas y analíticas derivadas. De aquí nacen dos salidas:
 - **Datos Estructurados (SD):** Datos o mensajes listos para consultas rápidas y eficientes.
 - **Analytics (A):** cálculos y métricas que se derivan de los datos crudos.
7. **Almacenamiento de datos en el Data Lake:** Repositorio central donde se almacenan las tres clases de datos (raw, estructurados y analíticos), para su uso futuro.
8. **Actualización del dashboard:** Las colecciones de datos estructurados y analíticos alimentan las visualizaciones y KPIs del tablero, permitiendo un monitoreo en tiempo real.
9. **Consumidores:** Usuarios finales y/o servicios acceden a los datos almacenados mediante consultas o reportes generados.
10. **Fin:** Cierre del flujo una vez cumplido el objetivo de disponibilizar los datos.

Esta descripción conecta las etapas del diagrama con los módulos definidos en la solución, ofreciendo una vista coherente del flujo de la información desde su captura hasta su uso por parte de los consumidores.

4.5 Síntesis del Diseño del Sistema

Los diagramas presentados anteriormente —el **Diagrama de Flujo** (Figura 4.6) y el **Diagrama Entidad–Relación** (Figura 4.5)— describen de manera complementaria el funcionamiento del *OptiDataLake*, desde la captura inicial de eventos de *market data* hasta su consumo por parte de los usuarios y servicios.

El **Diagrama de Flujo** resume el recorrido end-to-end de la información. El proceso inicia en el *módulo de Ingesta (DI)*, donde se reciben datos en tiempo real desde fuentes externas. Dichos datos se almacenan inmediatamente en la colección *Raw/Landing* a través del módulo **RD**, manteniendo una copia fiel e histórica de la data. Luego, el **Módulo Transform (T)** aplica transformaciones, estructura los datos y genera analíticas, de ella nacen lo que serían dos salidas: datos *estructurados* para consultas futuras (**SD**) y *analíticas* para KPIs y métricas (**A**). Finalmente, el **Data Lake (DL)** consolida estas tres vistas (raw, estructurada y analítica) y habilita su explotación, incluyendo la actualización de tableros para monitoreo en tiempo real.

Por su parte, el **Diagrama Entidad–Relación** complementa esta visión mostrando cómo interactúan los consumidores con el sistema. Los *consumidores* (p.ej., analistas, traders o servicios) formulan consultas que el **Data Lake** resuelve utilizando las colecciones *raw*, *structured* y *analytics*. Este modelo resalta la trazabilidad: desde un reporte o métrica es posible referenciar el dato estructurado y, si es necesario, retroceder hasta el registro crudo que le dio origen gracias a la data histórica.

En conjunto, ambos diagramas permiten apreciar un diseño **modular y escalable** que separa responsabilidades por etapas y facilita su evolución independiente:

- **Escalabilidad y baja latencia:** la ingesta continua separada del procesamiento asegura que el sistema soporte mayores tasas de datos sin afectar el tiempo de respuesta.
- **Resiliencia y continuidad:** la conservación de *raw data* y los procesos de transformación idempotentes reducen riesgos de pérdida y permiten re-intentos controlados.
- **Trazabilidad de extremo a extremo:** cada dato estructurado o analítico mantiene vínculo con su origen crudo, habilitando el monitoreo y verificación de estos datos.
- **Interoperabilidad:** la organización en **SD** y **A** estandariza el acceso, agiliza consultas y alimenta visualizaciones y servicios futuros.

En síntesis, el *Data Lake* integra ingesta, transformación, almacenamiento y consumo bajo principios de **escalabilidad, disponibilidad y calidad de datos**. Esto permite manejar grandes volúmenes de *market data* en tiempo real, asegurando información confiable y oportuna para la toma de decisiones y el análisis futuro.

4.6 Caso de uso

4.6.1 Monitoreo de Datos en Tiempo Real a través de un Dashboard

El sistema dispone de un *dashboard* web que permite a analistas y *traders* (consumidores) visualizar, en tiempo real, métricas clave derivadas de la colección **Analytics** del *Data Lake*. El panel ofrece selectores de *símbolo*, *métrica* (p.ej., *volumen_total*, *promedio_precios*, *volatilidad*) y *rango de fechas*; al confirmar, se consulta el **DL** y se renderiza un gráfico temporal con los valores agregados. La Figura 4.7 muestra un ejemplo de uso.

Actor Principal: Analista financiero o trader.

Propósito: Obtener una visión general en tiempo real del comportamiento del mercado financiero mediante métricas e indicadores procesados por el sistema.

Flujo de Trabajo:

1. El usuario ingresa al *dashboard* desde cualquier dispositivo con acceso a la plataforma y que tenga conexión, ya sea mediante la red corporativa o a través del **Data Lake**.
2. Configura los filtros de visualización seleccionando símbolo, métrica (por ejemplo, *volatilidad*, *volumen transado*, *variación de precios*) y rango de fechas según sus necesidades.
3. El sistema consulta el **Data Lake** y despliega gráficos que se actualizan conforme llegan nuevas ventanas de datos, lo que permite identificar tendencias o eventos relevantes.
4. A partir de las visualizaciones, el usuario puede ajustar sus estrategias de *trading* o realizar un análisis más profundo del comportamiento del mercado.

Validación del caso de uso: Con el objetivo de comprobar que el *dashboard* refleja fielmente la información procesada por el sistema y que sirve para monitoreo, se ejecutó una serie de pruebas sobre datos reales provenientes del **Data Lake**.

Objetivo de la validación: Evidenciar que las métricas publicadas en **Analytics** (p.ej., *volumen_total*, *promedio_precios*, *volatilidad*) se proyectan en el panel con oportunidad, consistencia temporal y precisión de cálculo.

Alcance: Lectura sobre colecciones **Analytics** y **Structured Data**, actualización visual en el frontend, y verificación de tiempos de respuesta bajo alta carga.

Resultados principales

- **Actualización continua:** los gráficos se refrescan según la ventana de agregación definida (`time_window`); se verificó que nuevas ventanas aparecen en el panel cuando **A** publica datos, manteniendo la secuencia temporal sin huecos ni solapamientos.
- **Precisión de KPIs:** para un conjunto de símbolos y periodos, los valores mostrados por el panel coincidieron con consultas directas a **Analytics** (cálculo de control), validando el pipeline de transformación en **T** y la lectura en el dashboard.
- **Monitoreo de eventos:** el selector de *métrica/símbolo/rango* permitió detectar picos, caídas abruptas y variaciones inusuales en series dentro del día, facilitando una inspección rápida de los movimientos del mercado.
- **Robustez de la interfaz:** ante ausencia de datos en el rango, el panel informa “sin resultados” en lugar de fallar, sin perder los filtros configurados.

Observaciones y mejoras

- Incluir comparación simultánea de múltiples símbolos y exportación de series (CSV/PNG).
- Agregar alertas configurables por umbral de KPI.

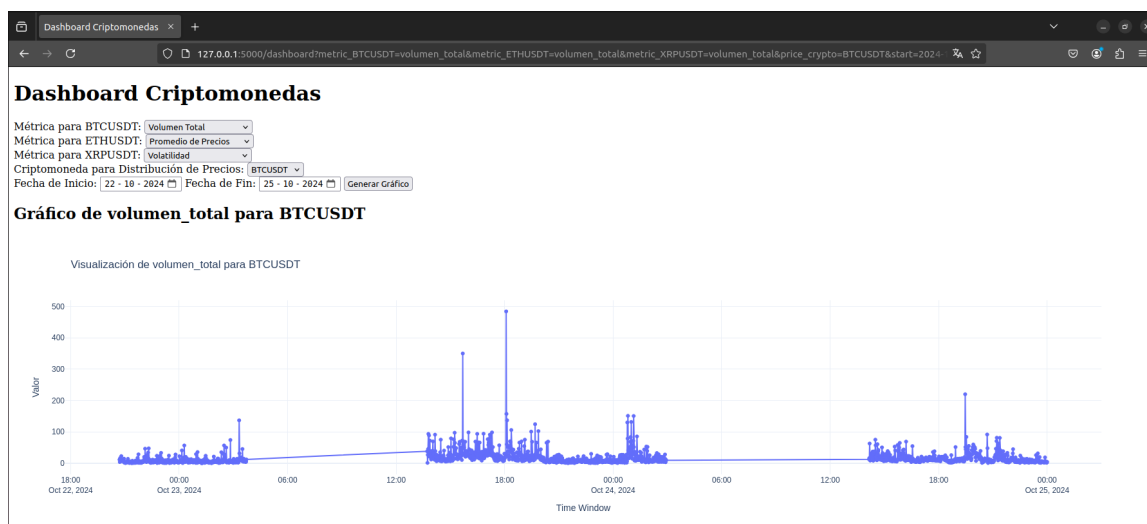


Figura 4.7: Dashboard de monitoreo con selección de símbolo, métrica y rango de fechas.

Este caso de uso evidencia la aplicabilidad práctica del sistema para cubrir necesidades de monitoreo operativo en tiempo real, ofreciendo acceso ágil a métricas y KPIs relevantes. Si bien el *dashboard* valida la utilidad del *Data Lake* en un escenario concreto, constituye sólo una de sus posibles aplicaciones. Gracias a su diseño modular y escalable, la plataforma puede extenderse a otros frentes: integración con modelos de predicción financiera

y detección de anomalías, generación automática de reportes periódicos y definición de alertas por umbrales.

5 Desarrollo de la Aplicación

En esta sección se describe el desarrollo técnico de los módulos del sistema, con énfasis en la **ingesta**, el **procesamiento** y la **integración/optimización del almacenamiento**. El trabajo se ejecutó de forma **iterativa e incremental**: se comienza con un *prototipo* que ingiere archivos `.log` para validar el flujo, luego se integra la API de Binance en tiempo real y, sobre esa base, se fue afinando el rendimiento, la limpieza de datos y las consultas. Cada avance se validó con datos reales para asegurar continuidad del flujo, trazabilidad y tiempos de respuesta acordes a los objetivos del proyecto.

El desarrollo cubre desde pruebas y diseño de componentes hasta su implementación, validación y evaluación. La **interacción entre módulos** (ingesta, raw/landing, transformación, datos estructurados y analítica) y el **flujo end-to-end** —desde el origen hasta su almacenamiento en el *Data Lake*— se presenta en el **Diagrama de Componentes del Sistema** (Figura 5.1). Este diagrama muestra cómo se integran *Kafka* (transporte y envío), *Spark* (limpieza, estructuración y cálculo por ventanas de tiempo) y *MongoDB* (colecciones de Raw, Structured y Analytics) para cumplir requisitos de **disponibilidad**, **rendimiento** y **calidad de datos**.

Como resultados prácticos del desarrollo, se consolidaron: (i) **validadores de esquema** e **índices compuestos** en MongoDB orientados a consultas por símbolo y tiempo; (ii) **optimización de ingesta** bajo *rate-limit* (lotes, concurrencia controlada y reintentos) con manejo de **idempotencia** mediante (`symbol`, `tradeId`); y (iii) un **esquema mínimo** para Structured/Analytics que habilita el cálculo consistente de *KPIs* y su visualización en el *dashboard* en tiempo real.

5.1 Revisión del Sistema

El sistema desarrollado apunta a ofrecer una solución **escalable**, **trazable** y **eficiente** para la gestión de *market data* en tiempo real. La implementación se orienta a eventos *tick-by-tick* provenientes de Binance y habilita un flujo continuo de ingestión, transformación y almacenamiento que soporta tanto consultas operativas de baja latencia como analítica por ventanas temporales. Desde su diseño inicial, el sistema privilegió la separación de responsabilidades y la observabilidad de extremo a extremo, de modo que cada etapa (captura, transformación, almacenamiento y consumo) pueda monitorearse, evaluarse y actualizarse sin interrumpir el servicio ni disminuir la calidad de los datos.

La arquitectura integra **Apache Kafka** como capa de transporte e independencia entre componentes (actuando como *consumer* y garantizando durabilidad y reintentos controlados), **Apache Spark** como motor de cómputo para *streaming* y *batch* (normalización de tipos, validación de campos y generación de métricas) y **MongoDB** como repositorio principal (colecciones para *Raw*, *Structured* y *Analytics*). Este enfoque modular permite **escalar cada componente de forma independiente** —más particiones en Kafka, más ejecutores en Spark, más réplicas o *shards* en MongoDB— y, si hiciera falta, sustituir tecnologías en un punto del flujo sin impactar al resto, porque el esquema y las reglas de datos están definidos de manera explícita.

Aunque hoy se trabaja con una sola fuente de datos para la *market data* (Binance), el sistema se diseñó con **modularidad y escalabilidad** como ejes principales. Esto permite sumar nuevos orígenes o mercados conectando su ingesta y mapeando sus mensajes al **esquema** de **Structured** y a las colecciones de **Analytics**, manteniendo registro de la data histórica en la colección **raw**.

Las contribuciones de este trabajo se enfocan en tres áreas. En **MongoDB** se definieron colecciones y validadores de esquema (campos obligatorios y tipos) e *índices compuestos* para consultas por símbolo y tiempo. En la **ingesta**, se implementó el flujo Binance→Kafka con publicación/consumo continuo, arranque en el *último offset* y lectura por ventanas de *poll* en *lotes*; además, se aplicaron validaciones ligeras de entrada, reintentos y un diseño de *escalado horizontal* por particiones y múltiples consumidores, junto con *logs* refinados (símbolo, precio, cantidad y tiempos) para trazabilidad. La **idempotencia** y el control de duplicados se resuelven en el módulo de **procesamiento** mediante la clave (`symbol`, `tradeId`). En **procesamiento**, se definió un **esquema mínimo** para los mensajes provenientes de **Raw**, se normalizaron los tiempos a formato *UTC* y se seleccionaron campos clave que permiten calcular *KPIs* y métricas que se almacenarán en la colección **Analytics**.

5.1.1 Flujo de Trabajo para el Desarrollo del Sistema

El desarrollo se organizó en cinco hitos que guiaron la construcción e integración del sistema, desde la definición del problema hasta su validación en funcionamiento continuo. A continuación se detalla cada hito y las contribuciones realizadas en cada una de ellas para cumplir con los objetivos propuestos para el correcto funcionamiento del sistema.

1) Definición de objetivos y requisitos: Se estableció el alcance funcional y no funcional del proyecto: ingestión en tiempo real, baja latencia de consulta, almacenamiento histórico, trazabilidad extremo a extremo (end-to-end), operación continua y un diseño modular.

2) Selección y evaluación de tecnologías: Se realizó una investigación comparativa entre diferentes tecnologías y lenguajes de programación considerando rendimiento, facilidad de operación, ecosistema y soporte. De este análisis se definió la combinación tecnológica entre: Apache Kafka para consumo y envío de datos, Apache Spark para la transformación de los datos, también para procesamiento *streaming/batch* y MongoDB como repositorio principal (Data Lake).

3) Diseño de la arquitectura: Se modeló una arquitectura por capas separando explícitamente cada responsabilidad: ingesta, transformación, analítica y almacenamiento. Se definieron los flujos de datos, tópicos y los esquemas mínimos por etapa (para interoperabilidad). Se confeccionaron diagramas (contexto, componentes y flujo) que aseguraran un diseño que sea escalable y eficiente a la vez.

4) Implementación inicial: Se fueron realizando prototipos funcionales que cumplirían con la siguiente configuración: conexión a la fuente de datos (API de Binance), publicación y consumo en Kafka, flujo de procesamiento inicial de Spark para limpieza/estructuración y almacenamiento en MongoDB en colecciones Raw/Structured/Analytics. Se prepararon datos de origen real para validar el ciclo completo, junto con *checkpoints* y configuración básica de reintentos ante fallos.

5) Pruebas y afinamiento: Se ejecutaron pruebas de funcionamiento y rendimiento para la validación completa del sistema incluyendo pruebas de ingestión, transformación, almacenamiento de datos y tiempos de consultas.

Contribuciones del Presente Trabajo

- **Observabilidad de la ingesta.** Se rediseña la salida de los *logs* para que cada mensaje muestre información útil (símbolo, precio, cantidad, tipo de transacción, marcas de tiempo y estado de envío/consumo), facilitando el diagnóstico y la verificación de extremo a extremo sin afectar el rendimiento del flujo.
- **Consumo y validación de datos de mercado.** Se implementa el consumidor que recibe los mensajes desde la fuente de datos y valida su recepción correcta; además, se incorporan consultas REST a la API de Binance (precios, profundidad y flags) como apoyo para pruebas y contrastes de datos. Con ello se habilitó la medición de la latencia del flujo.
- **Procesamiento con Apache Spark (estructuración y limpieza).** En colaboración se realiza la integración de Apache Spark con los demás componentes y también la configuración para transformar los datos crudos en registros estructurados. En particular, se define la normalización de campos, la eliminación de atributos

vacíos o no utilizados y el mapeo hacia el **esquema** de **Structured**, dejando los datos listos para consultas y para el cálculo posterior de métricas.

- **Data Lake en MongoDB (diseño y optimización).** Se configura el repositorio documental para operar como *Data Lake* con tres colecciones: **Raw**, **Structured** y **Analytics**. Se definen las colecciones y su finalidad, los validadores de esquema (campos obligatorios y tipos), y también **índices por símbolo/tiempo**, para poder acelerar las búsquedas y mantener un orden y consistencia aun en un modelo de base de datos no relacional.
- **Herramienta de consulta y evaluación de métricas.** Se desarrolla una interfaz de consulta con filtros, por símbolo y rango de precios, que reporta *tiempo de respuesta*, *cantidad de resultados* y *tamaño de datos recuperados (MB)*. Esta herramienta permitió verificar el objetivo operativo de volumen de consultas (referencia de 20 MB/s como umbral mínimo) y se generaron en base a los datos obtenidos **gráficos de rendimiento** para la verificación del sistema.

En conjunto, estos aportes mejoran la visibilidad y confiabilidad de la ingesta, dejan los datos **estructurados y consultables** en el *Data Lake*, y proporcionan **evidencia empírica** del desempeño del sistema mediante filtros y métricas.

5.1.2 Diagrama de Componentes del Sistema

La Figura 5.1 resume el flujo de la *market data* en cuatro funciones clave: (1) **captura** de eventos *tick-by-tick* desde la fuente de datos; (2) **recepción y distribución** para separar la llegada de datos del cómputo y suavizar picos de carga; (3) **procesamiento** para limpiar, tipar y calcular métricas por ventanas temporales; y (4) **almacenamiento** en un repositorio que conserva el dato crudo, el dato limpio y estructurado los cuales están listos para ser consultados. En la práctica, la API de Binance se conecta a Kafka, que la pone a disposición de Spark; éste aplica las transformaciones y, finalmente, MongoDB organiza el resultado en colecciones que permiten tanto monitoreo como análisis de baja latencia. A diferencia de los diagramas conceptuales, esta vista enfatiza la implementación concreta y cómo cada pieza contribuye a un flujo continuo y escalable.

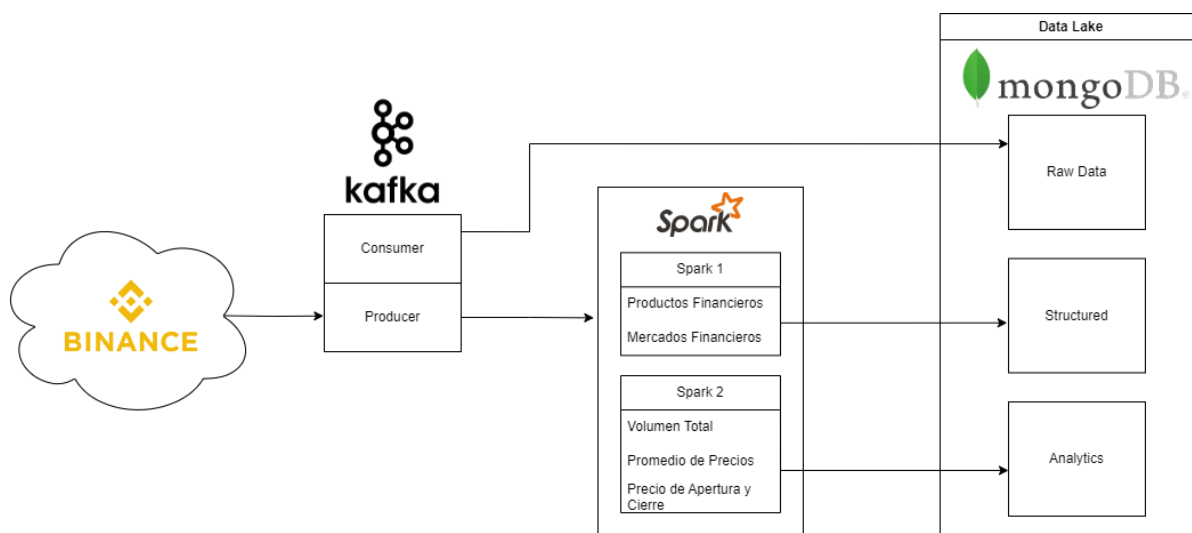


Figura 5.1: Visión general de los componentes y su interacción en el flujo de datos.

Componentes principales

- **API de Binance:** punto de origen de los datos de mercado en tiempo real (*tick-by-tick*), que alimentan el sistema.
- **Apache Kafka:** Gestiona la ingesta y el envío de mensajes mediante tópicos, manteniendo un flujo estable y separando (captura, procesamiento y almacenamiento).
- **Apache Spark:** motor de procesamiento que realiza la limpieza y la estructuración de los registros y calcula métricas por ventanas temporales para su análisis.
- **MongoDB (Data Lake):** repositorio principal organizado en tres colecciones complementarias: *Raw* (copia fiel del mensaje original), *Structured* (dato limpio listo para consulta) y *Analytics* (métricas/KPIs).

Este diagrama se incorpora para mostrar la **implementación concreta** del sistema: qué tecnologías se usan en cada tramo del flujo de los datos (Kafka, Spark y MongoDB), cómo se enlazan entre sí y dónde fluye efectivamente la *market data* desde la captura hasta su almacenamiento y consulta. A diferencia de los esquemas previos —más conceptuales o de alto nivel— aquí se enfatiza el “**cómo opera**” la solución: puntos de integración, responsabilidades técnicas por etapa y componentes desplegados. En resumen, ofrece una vista operativa que complementa las visiones de contexto, arquitectura y flujo, dejando claro **qué corre, dónde corre y cómo se comunica**, con el objetivo de garantizar continuidad end-to-end, escalamiento independiente por componente y trazabilidad desde cualquier métrica hasta su registro original.

5.2 Implementación de los Componentes

5.2.1 Ingestión de Datos

La ingesta de los datos se implementó integrando la API de Binance con **Apache Kafka**, que actúa como canal de entrada y distribución de *market data* en tiempo real. En este flujo mostrado en la figura 5.2, un *producer* publica los mensajes en un *tópico* central y uno o más *consumers* los extraen para enviarlos a las etapas o módulos posteriores.

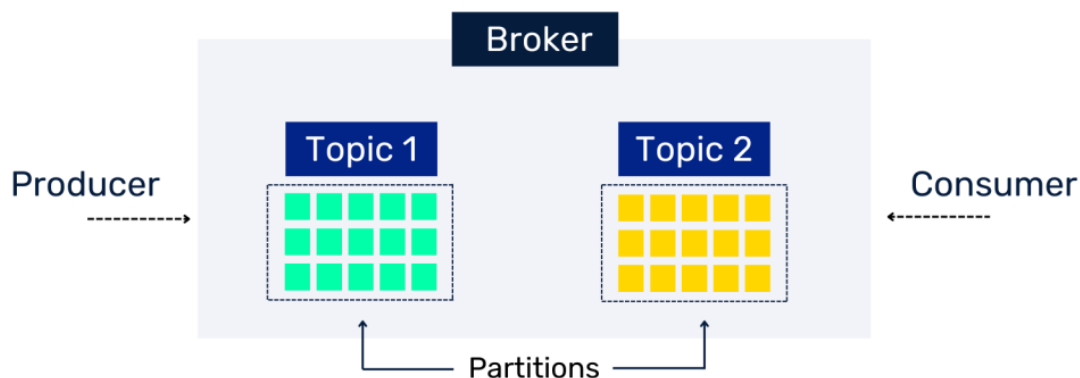


Figura 5.2: Esquema simplificado del flujo de ingesta con Apache Kafka.

A continuación se detallan los componentes principales del proceso mencionado:

Producer: El *Producer* es el componente que **publica** los eventos en el clúster de Kafka. En este proyecto toma los *ticks* de Binance, agrega metadatos mínimos (p.ej., *receivedAt*, *source*) y los envía en formato JSON hacia el tópico de ingesta.

Principales características del producer

- **Conexión continua:** mantiene la sesión con la API y publica eventos en tiempo real según la actividad del mercado.
- **Tolerancia a fallos:** reintentos con *backoff* y confirmaciones *acks* para garantizar escritura.
- **Soporte a múltiples tipos de criptomonedas:** Al inicio el sistema procesaba un solo tipo de criptomoneda (p.ej., BTCUSDT); luego se habilitó el manejo concurrente de varios símbolos (BTCUSDT, ETHUSDT, XRPUSDT, entre otros). Además, se ajustó el ritmo de envío por símbolo para cumplir con el *rate-limits* de Binance sin sacrificar *throughput*.
- **Manejo de ráfagas:** envío en *lotas* para estabilizar el flujo sin perder eventos.

- **Formato del mensaje:** Los mensajes son enviados con un formato JSON con sus respectivos campos únicos (`symbol`, `Id`) que ayudan al manejo de idempotencia y la duplicación de los datos.

Topic: Un *topic* es un **canal nombrado** dentro de Kafka: los *producers* escriben allí sus mensajes y los *consumers* los leen de manera independiente. En este proyecto utilizamos el tópico `odl-ticks` como punto único de entrada para los *ticks* del mercado, lo que simplifica el enrutamiento y la supervisión del flujo.

Principales características del topic

- **Retención y reprocesamiento:** se configuran políticas de retención (por tiempo o tamaño) que permiten *reintentos* controlados para que no existan pérdidas de mensajes cuando un mensaje no alcance a procesarse o exista un error, sin convertir a Kafka en almacenamiento permanente.
- **Escalamiento:** al centralizar los datos en `odl-ticks`, si aumenta el volumen se puede escalar agregando particiones o más instancias consumidoras; además, el sistema admite crear tópicos adicionales para integrar nuevas fuentes de datos sin afectar los flujos existentes.
- **Datos actualizados:** Se implementa un script que se encarga de eliminar los mensajes que ya fueron procesados en ejecuciones previas y quedan registradas en los checkpoints, esto con el fin de que el sistema siempre empiece a consumir datos nuevos.

Consumer: El *Consumer* **extrae** los mensajes almacenados en el tópico definido (`odl-ticks`) y los **entrega** al módulo de procesamiento (Spark) para su transformación. Puede operar en grupo, es decir, ejecutar varias copias del lector en paralelo; cada una toma una parte de los mensajes, de modo que el trabajo se reparte y el procesamiento es más rápido y estable.

Principales características del consumer

- **Transmisión en tiempo real:** Procesa los mensajes en tiempo real y de forma continua lo que mantiene un flujo constante de datos hacia los componentes posteriores.
- **Validación de entrada:** realiza chequeos ligeros (estructura y campos mínimos) antes de entregar a Spark.
- **Escalado horizontal:** varias instancias pueden trabajar en paralelo y repartirse el procesamiento para aumentar el rendimiento cuando crece el volumen de datos.

Relación entre los tres: El **Producer** publica los datos en `odl-ticks`; el **Topic** almacena y distribuye los eventos; los **Consumers** los leen y los envían a Spark para limpieza/estructuración y luego a MongoDB para su almacenamiento en las colecciones *Raw/Structured/Analytics*. Este patrón independiza la captura, procesamiento y almacenamiento, y permite escalar cada parte de forma independiente.

5.2.2 Contribución en el módulo de ingestión de datos.

- **Refinamiento de los Logs:** Se rediseñaron los logs de salida para que cada mensaje muestre símbolo, precio, cantidad, tipo de evento y marcas de tiempo, facilitando el diagnóstico de envío/consumo, la trazabilidad básica y el manejo de errores, esto para poder tener una correcta visión de si los mensajes se están consumiendo/enviando de manera correcta.

En la Figura 5.3 se muestran dos capturas de la ingesta: (i) el *Producer* cuando publica los *ticks* hacia Kafka y (ii) el *Consumer* cuando los extrae del tópico `odl-ticks` para enviarlos a Spark. Ambos logs incluyen los campos clave para trazabilidad y diagnóstico.

```
- [SpringProducer@binance] [ctReadThread-57] c.o.p.config.BinanceWebSocketClient : Mensaje enviado a Kafka: {"e":"trade","E":1726535696623,"s":"BTCUSD","t":3825858070,"p":"57699.99000000","q":"0.00010000"}
- [SpringProducer@binance] [ctReadThread-57] c.o.p.config.BinanceWebSocketClient : Mensaje enviado a Kafka: {"e":"trade","E":1726535696623,"s":"BTCUSD","t":3825858071,"p":"57699.99000000","q":"0.00013000"}
```

(a) Log del *Producer*: mensaje publicado en `odl-ticks`.

```
- [SpringConsumer@binance] [ntainer#0-0-C-1] c.o.consumer.listeners.ConsumerListener : Mensaje recibido: {"e":"trade","E":1726535790965,"s":"BTCUSD","t":3825862512,"p":"57778.72000000","q":"0.00010000","T":1726535790965}
- [SpringConsumer@binance] [ntainer#0-0-C-1] c.o.consumer.listeners.ConsumerListener : Datos guardados en MongoDB: com.odl.consumer.model.TradeData@e40567
```

(b) Log del *Consumer*: mensaje recibido desde `odl-ticks`.

Figura 5.3: Secuencia de publicación (Producer) y recepción (Consumer) durante la ingesta.

Los mensajes de Binance llegan con claves abreviadas que normalizamos de la siguiente manera:

Clave	Significado
e	eventType
E	eventTime (ms)
s	symbol
p	price
q	quantity
t	tradeId
T	tradeTime (ms)
m	buyerIsMaker

Tabla 5.1: Campos relevantes en los logs de Binance.

- **Consumer de Kafka:** Componente encargado de **leer en tiempo real** los mensajes publicados en `odl-ticks` y **entregarlos** de forma segura al módulo de procesamiento (Spark).

Flujo de ejecución (paso a paso)

- Paso 1: Suscripción al tópico:** El consumer se suscribe al tópico `odl-ticks`, quedando habilitado para recibir particiones asignadas.
- Paso 2: Bucle de *poll*:** Inicia un ciclo continuo de lectura (*poll*) con ventana temporal fija, que extrae lotes de mensajes sin bloquear el procesamiento.
- Paso 3: Chequeo de integridad:** Cada mensaje se valida con reglas definidas (estructura JSON y campos mínimos como `symbol`, `tradeId`, `price`, `eventTime`).
- Paso 4: Envío a procesamiento:** Los mensajes válidos se envían a Spark para limpieza y estructuración.
- Paso 5: *Commit* de *offsets*:** Tras procesar el lote con éxito, se confirman (*commit*) los *offsets* del grupo, garantizando una reanudación sin pérdidas.

Comportamiento operativo:

- **Arranque con datos frescos.** En la inicialización, el consumidor se posiciona en el último offset del tópico para comenzar con datos recientes, evitando reprocesar mensajes de ejecuciones anteriores.
- **Flujo constante.** El diseño mantiene un ritmo sostenido de lectura y envío, evitando picos mediante lotes y ventanas de *poll* estables.
- **Escalamiento.** Al crecer el volumen, se puede aumentar el número de instancias del consumer y/o las particiones del tópico.

5.2.3 Procesamiento de Datos

El procesamiento se realiza con **Apache Spark**. A partir de los mensajes publicados en **Kafka**, Spark ejecuta una *limpieza y estructuración* para dar un uso consistente a los datos y, sobre esa base, calcula *métricas analíticas por ventanas de tiempo*. Gracias a su arquitectura distribuida y a **Spark Streaming**, ofrece una solución eficiente y escalable para operar con grandes volúmenes de datos en tiempo real. El objetivo de este módulo es convertir datos crudos en información lista para consulta y análisis, manteniendo baja latencia y la trazabilidad hasta el registro original (*Raw*).

Qué resuelve esta etapa

- **Limpieza y validación:** convierte los mensajes crudos en registros confiables: descarta datos incompletos o vacíos (p. ej., precio/cantidad) y evita duplicados usando la clave (`symbol`, `Id`).

- **Estructuración:** normalización temporal a *UTC*, selección de campos mínimos y cumplimiento de esquema para la vista *Structured*.
- **Cálculo de KPIs:** cálculos por ventana (p. ej., 1 minuto) para volumen, precios de apertura/cierre, promedios y otras métricas que alimentan *Analytics*.

Cómo encaja en el flujo: Kafka entrega los *ticks* a Spark en *streaming/microbatches*. Tras aplicar las transformaciones, Spark almacena dos salidas: (i) *Structured* para consultas y (ii) *Analytics* con KPIs por símbolo/tiempo. En ambos casos se preserva la referencia al registro *Raw*.

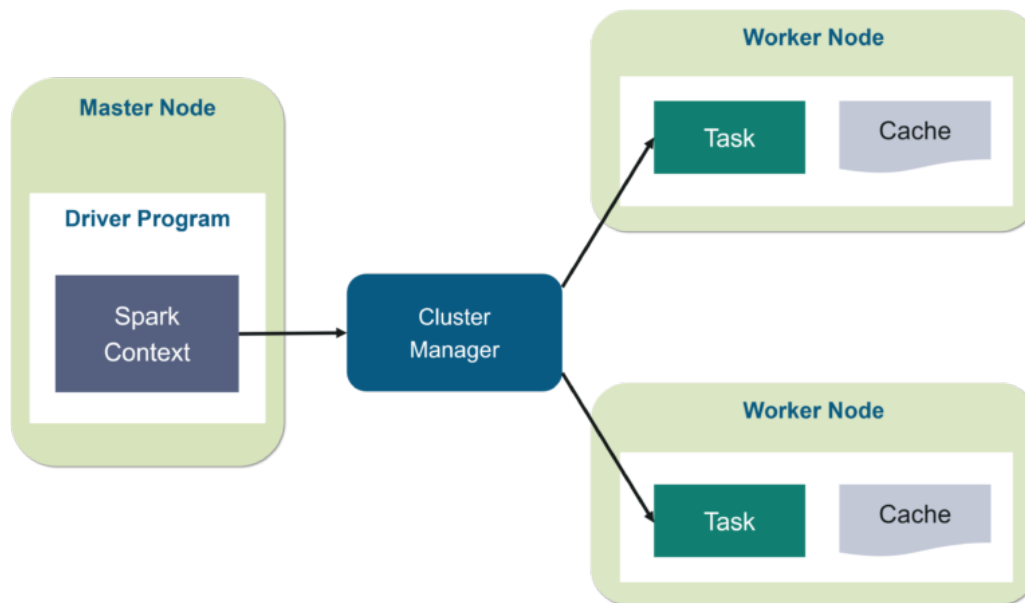


Figura 5.4: Arquitectura de procesamiento con Apache Spark (Driver, Cluster Manager y Workers).

Componentes internos de Spark

- **Master Node (Driver Program / SparkContext):** nodo principal que ejecuta el *Driver Program*, crea el *SparkContext* (contexto de spark) y define el plan de ejecución. Desde este componente se divide el trabajo y lo distribuye a los workers nodes.
- **Cluster Manager:** coordina los recursos del clúster (CPU, memoria) y programa las tareas en los nodos disponibles. En este proyecto se utilizó el *modo local*, que ejecuta todo en un solo equipo y resulta ideal para desarrollo y pruebas. Para producción, el diseño permite escalar a un clúster distribuido.
- **Worker Nodes (Executors y Tasks):** nodos que ejecutan en paralelo las tareas asignadas por el Master como lectura desde Kafka, filtros, entre otros. Cada nodo

puede usar *cache* en memoria para almacenar temporalmente los datos para poder acelerar el acceso y a su vez mejorar el rendimiento, al finalizar, publica los resultados para su almacenamiento en MongoDB.

Ventajas para el proyecto

- **Escalabilidad:** Spark reparte el trabajo tanto por particiones (batches que llegan desde Kafka) como por tareas en los workers. Si el volumen crece, basta con aumentar particiones o añadir ejecutores para procesar más mensajes en paralelo, sin cambiar la lógica de negocio.
- **Tolerancia a fallos:** El motor maneja reintentos automáticos cuando una tarea falla y utiliza checkpointing de los microbatches para que, si hay una interrupción, el procesamiento se reanude exactamente donde quedó, sin duplicar resultados ni perder datos. Esto asegura consistencia en las métricas y en los datos que se almacenan en MongoDB.
- **Compatibilidad:** conectores nativos con Kafka y MongoDB que simplifican la integración extremo a extremo con ellas y otras herramientas.

5.2.4 Flujo del procesamiento de los datos

El procesamiento se realiza con **Apache Spark**. A partir de los mensajes publicados en Kafka, Spark ejecuta una *limpieza y estructuración* para su uso posterior, y calcula *métricas analíticas por ventanas de tiempo*. El objetivo es convertir datos crudos en información lista para consulta y análisis, manteniendo baja latencia, confiabilidad y trazabilidad hasta el registro original (*Raw*).

Explicación detallada del flujo

1. **Lectura desde Kafka:** Spark actúa como consumidor del tópico `odl-ticks` y recibe los mensajes en tiempo real.
2. **Transformaciones y Estructuración en Spark:** los datos crudos se convierten a *DataFrames* y se aplican filtros/validaciones como: conversión de tipos de datos para campos numéricos y temporales, descarte de valores nulos o inválidos y eliminación de duplicados con la clave (`symbol, tradeId`); luego se normalizan las marcas de tiempo a *UTC*, se seleccionan los campos mínimos y se aplica el esquema para el almacenamiento en la colección *Structured*.
3. **Organización de datos:** Spark consolida y ordena la información en catálogos básicos para facilitar las consultas y la reutilización en el resto del sistema, organizándola en torno a:

- *Productos financieros*: catálogo de productos financieros (p.ej., criptomonedas), con identificadores únicos (`symbol`), nombre legible y metadatos mínimos.
 - *Mercados financieros*: catálogo de mercados financieros (p.ej., spot, futuros), con su identificador único, nombre y tipo de mercado.
4. **Cálculo de KPIs**: Se aplican ventanas de tiempo por símbolo (BTCUSDT, ETHUSDT y XRPUSDT) para obtener:
- ***Volumen total***: suma de las cantidades transaccionadas dentro de la ventana de tiempo.
 - ***Precio de apertura***: precio del *primer* trade registrado en la ventana de tiempo.
 - ***Precio de cierre***: precio del *último* trade registrado en la ventana de tiempo.
 - ***Promedio de precios***: promedio de los precios de las transacciones realizadas en la ventana de tiempo.
 - ***Volatilidad***: medida de la dispersión de precios en la ventana de tiempo, por ejemplo desviación estándar, esta refleja la variabilidad del mercado.
 - ***Movimiento de precios***: diferencia entre el precio de cierre y el de apertura calculado en la ventana de tiempo.
5. **Almacenamiento de los datos en MongoDB**. Los resultados ya procesados se almacenan en MongoDB en dos colecciones: *Structured*, la cual contiene el mensaje limpio y estructurado para consultas relacionados con los mercados y productos financieros, y *Analytics*, la cual almacena las métricas y KPIs calculadas.

Contribuciones en el módulo de procesamiento de datos (Spark)

Los aportes en esta capa se centraron en que los datos llegaran *limpios*, con un formato estable para consultas y con salidas analíticas consistentes:

- **Modelo y limpieza básica de los *ticks***: En conjunto con el equipo se define el mapeo de las claves abreviadas de Binance a nombres claros (símbolo, precio, cantidad, tiempos, etc.). Se convierten los tipos de datos (numéricos y temporales) y se eliminan campos vacíos o irrelevantes, dejando un esquema compacto solo con la información necesaria.
- **Estructuración para consultas**: Se define un conjunto mínimo de columnas para dejar la salida con un esquema representativo para la colección *Structured*, de modo que las consultas siempre encuentren el mismo formato.

- **Integración extremo a extremo:** Se conectó el flujo *Kafka* $\rightarrow$ *Spark* $\rightarrow$ *MongoDB* para procesar eventos en microbatches y publicar los resultados. Para prevenir duplicados, se aplica una verificación con la clave (`symbol`, `Id`) antes de almacenar, manteniendo la consistencia, confiabilidad y la trazabilidad del dato.

5.2.5 Almacenamiento de Datos

El almacenamiento se implementó con **MongoDB**, una base de datos NoSQL orientada a documentos la cual permite un manejo más flexible de grandes volúmenes de datos, estos datos son almacenado en diferentes colecciones de acuerdo al procesamiento que tuvieron. En este proyecto, MongoDB centraliza los datos que llegan desde la captura y el procesamiento, separando estos datos por nivel de madurez para facilitar las consultas y el análisis que se realizará posteriormente.

La configuración inicial del repositorio se realizó con **MongoDB Compass**, herramienta que facilitó la creación rápida y visual de las colecciones *Raw_data*, *Structured* y *Analytics*. Desde Compass se definieron los campos básicos y se verificó la recepción de registros durante las primeras pruebas de ingesta, dejando preparado el entorno para que los módulos de Kafka y Spark pudieran escribir/consultar de forma consistente y confiable.

Fuentes de entrada: Los datos ingresan a MongoDB desde dos fuentes de datos diferentes la cuales son:

- **Kafka $\rightarrow$ Raw:** el productor publica los datos tal como llegan desde Binance; esos mensajes se almacenan *sin transformar* para conservar un registro histórico de la data.
- **Spark $\rightarrow$ Structured / Analytics:** el procesador toma los datos, los limpia/estructura y calcula métricas por ventanas de tiempo; esas salidas se guardan en colecciones específicas para consultas y análisis posteriores.

Colecciones y campos:

Raw_data (dato crudo)

Contiene los eventos tal como fueron recibidos. Se usa poder realizar un análisis histórico de la data y también por si necesita algún re procesamiento de algún dato. Los principales campos que contiene este tipo de datos son:

- `eventType`: tipo de evento (p.ej., `trade`).
- `eventTime` (ms): momento en que Binance emitió el evento.
- `symbol`: activo financiero (p.ej., `BTCUSDT`).

- **price, quantity:** precio y cantidad del activo negociado.
- **tradeId, tradeTime (ms):** identificador y hora de la transacción.

Structured (dato limpio y estructurado)

La colección **Structured** almacena los datos ya transformados por Spark con un esquema estable y listo para recibir consultas. En términos lógicos, esta colección reúne dos grupos de información:

- **Productos financieros (lógico):** Identifican el activo negociado. En el proyecto se utilizan los campos `producto_id` y `nombre_producto` para referirse al activo negociado (p. ej., BTCUSDT).
- **Mercados financieros (lógico):** Describen el mercado donde ocurre la transacción. Se emplean `mercado_id`, `nombre_mercado` y `tipo_mercado` (p. ej., *spot*).

En este proyecto no se modelaron *colecciones* separadas llamadas *Productos_Financieros* y *Mercados_Financieros*. En su lugar, la información se conserva dentro de *Structured*. Aun así, la implementación de **índices** agiliza las búsquedas y permite que sean mucho más rápidas.

Con este enfoque, *Structured* mantiene un solo punto de consulta, pero soporta consultas eficientes tanto por activo (*producto financiero*) como por *mercado*, sin necesidad de separar físicamente la información en múltiples colecciones.

Analytics (métricas por ventana)

Almacena **KPIs y métricas** calculadas por Spark mediante streaming, estas se derivan de la market data procesada en tiempo real. Las métricas incluyen los siguientes campos relevantes:

- **volumen_total:** cantidad total del activo negociado
- **promedio_precios:** media de los precios observados en una ventana de tiempo.
- **volatilidad:** variación relativa de precios en el período.
- **movimiento_precios:** diferencia entre el primer y el último precio registrado en la ventana de tiempo.

Optimización de las consultas

Indexación: Para acelerar búsquedas y poder tener un acceso más rápido a los datos se implementaron las siguientes estrategias:

- En *Structured*: `(symbol, eventTime)` y/o `(symbol, time_window)` para lecturas por activo y rango temporal.

- En *Analytics*: (`symbol`, `time_window`) para recuperar rápido métricas por símbolo y un rango temporal.

Estos índices reducen tiempos de respuesta y estabilizan las consultas aun con volúmenes muy altos de datos.

Estructuración por colecciones: Separar *Raw*, *Structured* y *Analytics* evita mezclar usos distintos:

- *Raw_data*: preserva data histórica.
- *Structured*: entrega un esquema controlado para las consultas.
- *Analytics*: concentra sólo métricas/KPIs, aligerando las lecturas analíticas.

Rol de MongoDB en el proyecto: Para los fines de este trabajo, **MongoDB funciona como un *Data Lake***, este centraliza datos crudos, limpios y estructurados en un mismo repositorio, manteniendo trazabilidad de extremo a extremo. Este rol es posible gracias a la integración con **Kafka** (ingesta en tiempo real) y **Spark**. Es decir, **MongoDB por sí solo es una base de datos** que al combinarlo con el flujo de captura y procesamiento, se obtiene el comportamiento de *Data Lake* requerido (histórico, múltiples colecciones para los datos y consultas eficientes).

Contribuciones en el módulo de Almacenamiento de datos

El propósito principal es convertir MongoDB en un *Data Lake* funcional para el proyecto, pasando de una base sin estructura a un repositorio organizado, que permita consultas y sea escalable. Los principales aportes en este módulo incluyeron lo siguiente:

Diseño y creación de colecciones

- **Estructura base:** se define y crea, con *MongoDB Compass*, las colecciones *Raw_data*, *Structured* y *Analytics*, alineadas con las etapas del flujo (ingesta, limpieza/estructuración y métricas).
- **Criterios de diseño:** se separaron los *datos crudos*, *datos listos para consulta y métricas/KPIs* para evitar mezclar usos, reducir la carga de las consultas y facilitar el escalamiento.

Modelado de campos y normalización

- **Raw_data:** se almacena cada mensaje tal como llega (sin modificaciones) con un conjunto mínimo de campos: `eventType`, `eventTime (ms)`, `symbol`, `price`, `quantity`, `Id`, `tradeTime (ms)`, `tradeId` y `buyerIsMaker`. Mantener este registro original permite

verificar el origen de cualquier dato, comparar contra resultados procesados y volver a procesar el histórico cuando cambie la lógica o se necesite reconstruir información.

- **Structured:** se define un esquema limpio y consistente (tipos numéricos/temporales, tiempos en UTC y campos indispensables como `symbol`, `price`, `quantity`, `tradeId`, `eventTime` y `tradeTime`).
- **Dimensiones (productos y mercados):** logicamente se define la organización de atributos en *productos* y *mercados* financieros (p. ej., `producto_id`, `nombre_producto`, `mercado_id`, `nombre_mercado`, `tipo_mercado`). Aunque no se desplegaron como colecciones separadas, la combinación de campos e índices en la colección **Structured** facilitan la vista cuando se hacen consultas.
- **Analytics:** se establece el formato para métricas y KPIs por símbolo y ventana de tiempo: `symbol`, `time_window`, `volumen_total`, `promedio_precios`, `precio_apertura`, `precio_cierre` y `volatilidad`.

Índices y rendimiento de consultas

- **Índices compuestos en Structured:** se configura una indexación con (`symbol`, `eventTime`) y/o (`symbol`, `tradeTime`) para acelerar búsquedas y obtención de datos por activo y rango temporal.
- **Índices en Analytics:** se configura una indexación con (`symbol`, `time_window`) para recuperar rápido métricas por símbolo y ventana de tiempo.
- **Resultado:** menor tiempo de respuesta y consultas más estables bajo mayor volumen, sin cambiar lógica de aplicación.

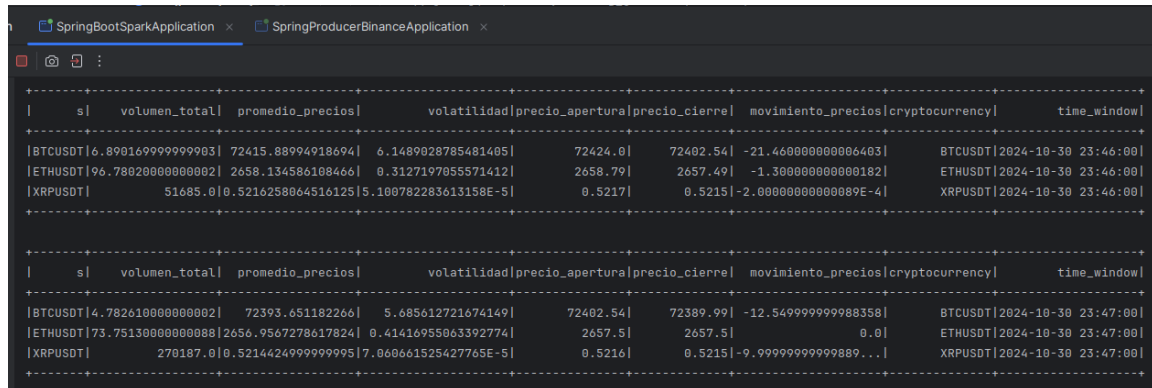
Criterios de calidad y operación

- **Idempotencia y duplicados:** se define un manejo de duplicados e idempotencia mediante el uso de (`symbol`, `Id`) para evitar duplicaciones entre los datos que existen en la colección `Raw_data` y `Structured`.
- **Mantenimiento:** se configura que al reiniciar el sistema, se eliminan solo los registros temporales de la ejecución anterior para volver a empezar desde datos nuevos, evitando re lecturas de mensajes antiguos y sin perder la información ya guardada en MongoDB.

Con estas tareas, **MongoDB pasó de ser sólo una base de datos a operar como *Data Lake* del proyecto:** centralizando los datos de ingesta (Kafka), recibiendo las salidas procesadas (Spark) y ofreciendo colecciones optimizadas para diferentes tipos de

5.3.2 Procesamiento de datos con Spark

Se busca verificar que Apache Spark consuma los datos desde Kafka, que aplique una limpieza y estructuración, y calcule métricas/KPIs por ventanas de tiempo (p.ej., 1 minuto).



s	volumen_total	promedio_precios	volatilidad	precio_apertura	precio_cierre	movimiento_precios	cryptocurrency	time_window
BTCUSD	6.8901699999999998	72415.8894918694	6.1489028785481405	72424.0	72402.54	-21.460000000000004	BTCUSD	2024-10-30 23:46:00
ETHUSD	196.78020000000002	2658.134586108466	0.3127197055571412	2658.79	2657.49	-1.3000000000000182	ETHUSD	2024-10-30 23:46:00
XRPUSD	51685.0	0.5216258064516125	5.100782283613158E-5	0.5217	0.5215	-2.000000000000089E-4	XRPUSD	2024-10-30 23:46:00
BTCUSD	4.7826100000000002	72393.651182266	5.685612721674149	72402.54	72389.99	-12.549999999998358	BTCUSD	2024-10-30 23:47:00
ETHUSD	173.75130000000008	2656.9567278617824	0.41416955063392774	2657.5	2657.5	0.0	ETHUSD	2024-10-30 23:47:00
XRPUSD	270187.0	0.52144249999999995	7.060661525427765E-5	0.5216	0.5215	-9.99999999999889...	XRPUSD	2024-10-30 23:47:00

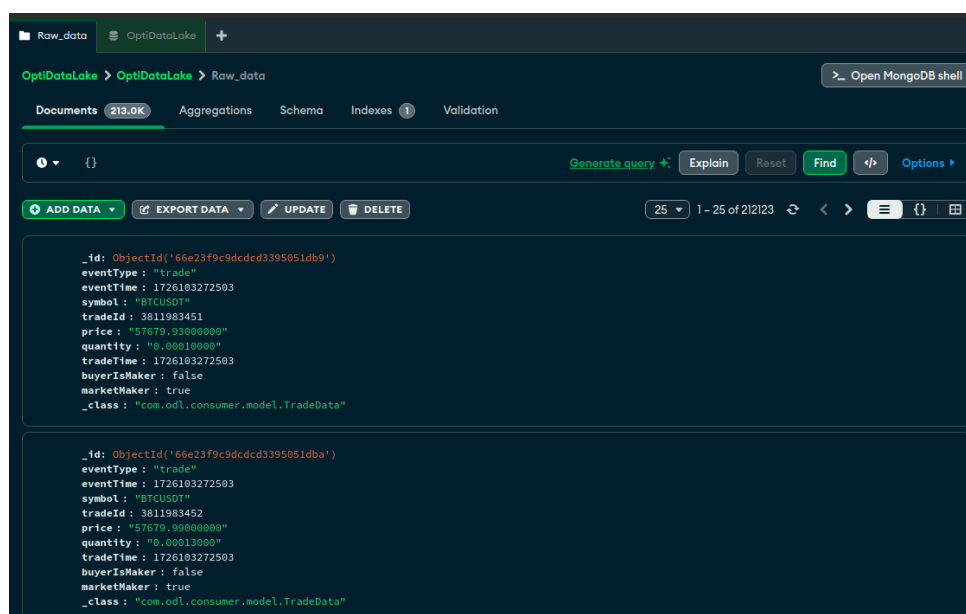
Figura 5.7: Ejecución en consola: KPIs calculadas.

Resultado esperado: Para cada símbolo se observan columnas con valores consistentes (sin *nulls* inesperados, sin duplicados) y con tiempos normalizados. Esto confirma que el módulo de procesamiento está operativo y entrega valores consistentes.

5.3.3 Almacenamiento: MongoDB (Raw y Analytics)

Se busca comprobar que los datos crudos (*Raw_data*) y los resultados analíticos (*Analytics*) se almacenen correctamente en MongoDB en sus respectivas colecciones con el formato y estructura definido.

Evidencia *Raw_data*



Document 1	Document 2
<pre>{ "_id": "66e23f9c9dcd3395051db9", "eventType": "trade", "eventTime": 1726103272503, "symbol": "BTCUSD", "tradeId": 3811983451, "price": "57679.93000000", "quantity": "0.00010000", "tradeTime": 1726103272503, "buyerIsMaker": false, "marketMaker": true, "_class": "com.odl.consumer.model.TradeData" }</pre>	<pre>{ "_id": "66e23f9c9dcd3395051dba", "eventType": "trade", "eventTime": 1726103272503, "symbol": "BTCUSD", "tradeId": 3811983452, "price": "57679.99000000", "quantity": "0.00013000", "tradeTime": 1726103272503, "buyerIsMaker": false, "marketMaker": true, "_class": "com.odl.consumer.model.TradeData" }</pre>

Figura 5.8: Colección *Raw_data*: eventos crudos con campos de Binance.

Evidencia *Analytics*

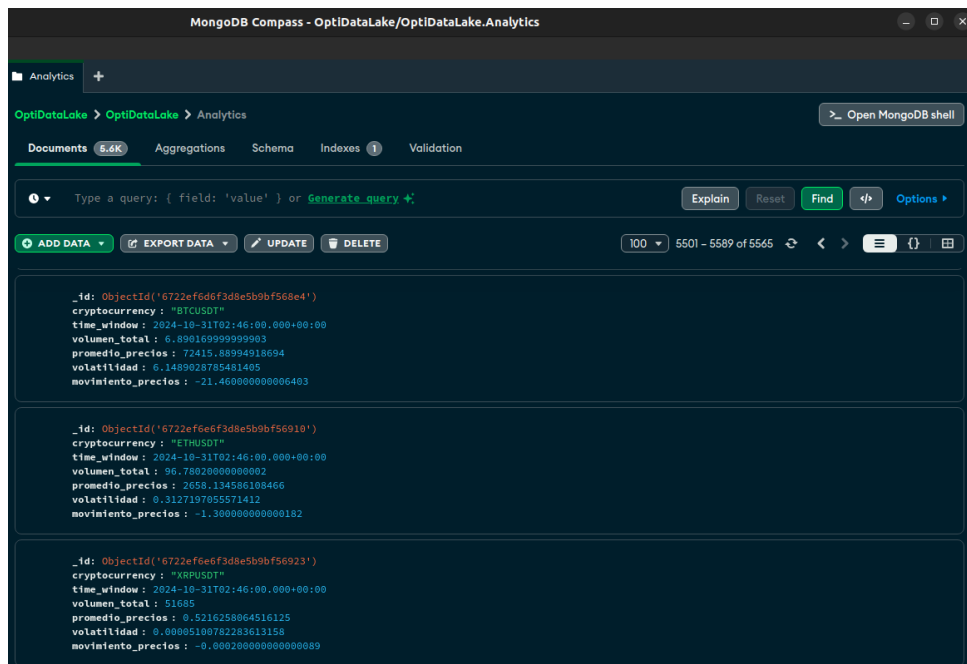
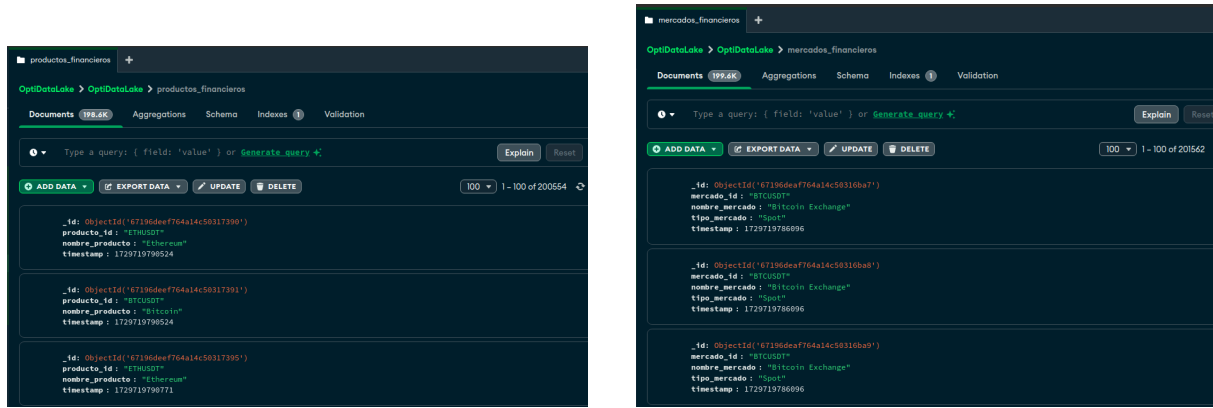


Figura 5.9: Colección *Analytics*: KPIs y métricas por ventana de tiempo.

Resultado esperado: *Raw_data* conserva los mensajes tal como llegan (para historia y trazabilidad). *Analytics* contiene únicamente métricas agregadas por símbolo/ventana, listas para consultas rápidas. Las colecciones muestran, en MongoDB Compass, índices configurados que acelerarán las búsquedas por `symbol` y tiempo.

5.3.4 Almacenamiento: Structured

Se busca verificar que los datos estén organizados y normalizados en la colección *Structured*. En el proyecto no se separaron en colecciones distintas, pero los índices y el esquema reflejan dos ejes de consulta: *Producto financiero* y *Mercado financiero*, esto con el fin de que se observe la diferencia entre ambas.



(a) Vista *producto financiero* con sus respectivos campos.

(b) Vista *mercado financiero* con sus respectivos campos

Figura 5.10: Ejemplos de organización en *Structured* para consultas por producto y por mercado.

Resultado esperado: Las dos vistas permiten observar las dos estructuras de datos que existen en la colección structured. Además de los respectivos índices que agilizan las consultas mejorando los tiempos de respuesta.

5.3.5 Contribuciones en las pruebas para la validación del Sistema

Se ejecutaron pruebas end-to-end para comprobar el correcto funcionamiento de los módulos de ingesta, procesamiento y almacenamiento. En síntesis, se verificó lo siguiente:

- **Ingesta desde Binance a Kafka:** Se comprueba que los eventos llegaran sin interrupciones al *topic odl-ticks*, validando en los *logs* el contenido del mensaje (símbolo, precio, cantidad y marcas de tiempo) y la continuidad del flujo.
- **Procesamiento con Apache Spark:** Se confirma que Spark consumiera los datos en tiempo real, aplicara la limpieza/estructuración y calculara las métricas analíticas (por ventanas de tiempo). Asimismo, se verifica la escritura de resultados en MongoDB.
- **Almacenamiento en *Raw_data*:** Se verifica que los datos crudos quedaran almacenados tal como se recibieron, con los campos mínimos definidos para trazabilidad y posibles reintentos.
- **Almacenamiento en *Structured*:** Se valida que los datos procesados quedaran normalizados, estructurados y listos para ser consultados.

- **Almacenamiento en *Analytics*:** Se comprueba que las métricas calculadas por Spark (volumen total, promedios de precio, volatilidad, etc.) se almacenan correctamente quedando listas para futuros análisis.

Estas validaciones entregan un respaldo técnico del sistema, demostrando que este recibe, procesa y almacena la market data según las especificaciones definidas, y que las colecciones *Raw_data*, *Structured* y *Analytics* cumplen su propósito de consulta y análisis.

5.4 Pruebas de Rendimiento

En esta sección se evalúa el comportamiento del sistema bajo alta carga de datos desde el punto de vista de: (i) **ancho de banda efectivo** durante la recuperación de grandes volúmenes de datos y (ii) **ritmo de consultas por minuto** en las colecciones *Raw_data* y *Structured*. El objetivo es comprobar que la plataforma mantiene continuidad, baja latencia y un caudal de lectura superior al umbral definido.

5.4.1 Objetivo

Validar que el sistema:

- alcanza o supera un umbral de **20 MB/s** en recuperación de datos.
- sostiene un **flujo estable de inserciones** por minuto en *Raw_data*, en *Structured* y en *Analytics*.
- mantiene **consistencia temporal** y ausencia de pérdidas visibles durante picos de actividad.

5.4.2 Entorno y datos

Las pruebas se ejecutaron en el entorno local del proyecto con los servicios de ingesta (*Producer/Consumer* de Kafka), procesamiento (Spark Streaming) y almacenamiento (MongoDB). La *market data* de referencia provino de la API de Binance. Para la medición de lectura se utilizó una **interfaz de consulta**, como la que se observa en la Figura 5.11, que permite filtrar por símbolo y rango de precios, y reporta tiempo total, filas recuperadas y tamaño (MB), lo que nos ayuda a facilitar la validación del rendimiento del sistema bajo diferentes volúmenes de datos.

Consulta de Precios en MongoDB

Símbolo de las monedas (separados por comas, ej: BTCUSDT,ETHUS...): BTCUSDT

Precio mínimo: 60000

Precio máximo: 68000

☒ Mostrar Símbolo
☒ Mostrar Cantidad
☒ Mostrar Tipo de Evento
☒ Mostrar TradeId
☒ Mostrar MarketMaker

☒ Mostrar Precio
☒ Mostrar Tiempo de Trade
☒ Mostrar Tiempo de Evento
☒ Mostrar BuyerIsMaker

Buscar

Symbol	Price	Quantity	TradeTime	EventType	EventTime	TradeId	BuyerIsMaker	MarketMaker
BTCUSDT	60590.14	0.00453	1728570876070	trade	1728570876102	3908567617	true	true
BTCUSDT	60590.18	9.0E-5	1728570876070	trade	1728570876102	3908567616	true	true
BTCUSDT	60590.25	2.8E-4	1728570876070	trade	1728570876102	3908567614	true	true
BTCUSDT	60590.25	1.4E-4	1728570876070	trade	1728570876102	3908567615	true	true
BTCUSDT	60590.36	0.00709	1728570876069	trade	1728570876101	3908567612	true	true
BTCUSDT	60590.36	0.00896	1728570876070	trade	1728570876102	3908567613	true	true
BTCUSDT	60590.37	4.8E-4	1728570876069	trade	1728570876101	3908567611	true	true
BTCUSDT	60590.39	1.8E-4	1728570876069	trade	1728570876101	3908567610	true	true
BTCUSDT	60590.57	9.0E-5	1728570876069	trade	1728570876101	3908567609	true	true
BTCUSDT	60590.62	1.0E-4	1728570876069	trade	1728570876101	3908567608	true	true
BTCUSDT	60590.7	1.1E-4	1728570876069	trade	1728570876101	3908567607	true	true
BTCUSDT	60590.81	1.4E-4	1728570876069	trade	1728570876101	3908567604	true	true
BTCUSDT	60590.81	1.8E-4	1728570876069	trade	1728570876101	3908567605	true	true
BTCUSDT	60590.81	0.0011	1728570876069	trade	1728570876101	3908567606	true	true
BTCUSDT	60590.91	0.00115	1728570876069	trade	1728570876101	3908567602	true	true
BTCUSDT	60590.91	9.0E-5	1728570876069	trade	1728570876101	3908567603	true	true
BTCUSDT	60591.0	0.00251	1728570876069	trade	1728570876101	3908567601	true	true
BTCUSDT	60591.03	9.0E-5	1728570876069	trade	1728570876101	3908567600	true	true
BTCUSDT	60591.16	2.9E-4	1728570876069	trade	1728570876101	3908567599	true	true
BTCUSDT	60591.25	1.7E-4	1728570876069	trade	1728570876101	3908567598	true	true
BTCUSDT	60591.27	1.3E-4	1728570876069	trade	1728570876101	3908567597	true	true
BTCUSDT	60591.28	0.01403	1728570876069	trade	1728570876101	3908567596	true	true

Tiempo de consulta: 30182 ms
 Total de resultados: 2844836
 Tamaño de datos recuperados: 610,44 MB (20,23 MB/s)

Figura 5.11: Interfaz de consulta utilizada para medir tiempo de respuesta, filas recuperadas y tamaño total.

5.4.3 Procedimiento

1. **Carga de trabajo (ingesta):** se ejecutó la ingesta en tiempo real desde Binance hacia Kafka; Spark consumió los eventos, realizó limpieza/estructuración y escribió en MongoDB (*Raw_data*, *Structured* y *Analytics*).
2. **Escenarios de lectura:** se realizaron consultas con distintos rangos de precio y símbolos para obtener conjuntos de tamaños variables. Para cada ejecución se registró *tamaño total recuperado (MB)* y *tiempo (s)*.
3. **Cómputo de ancho de banda:** se utilizó la siguiente expresión para obtener la tasa de consumo efectivo:

$$\text{Ancho de Banda (MB/s)} = \frac{\text{Tamaño Total de Datos Recuperados (MB)}}{\text{Tiempo Total de Consulta (s)}} \quad (5.1)$$

El valor resultante se comparó con el umbral objetivo de 20 MB/s.

4. **Monitoreo de ingesta:** en paralelo, se midió el *número de inserciones por minuto* en *Raw_data* y *Structured* para verificar la continuidad del *flujo de datos*.

5.4.4 Métricas

- Ancho de banda efectivo (MB/s) en lectura desde MongoDB.
- Datos ingresados por minuto en *Raw_data*.
- Datos ingresados por minuto en *Structured*.

5.4.5 Resultados

Recuperación de datos: La Figura 5.12 relaciona el tamaño total de las respuestas con el ancho de banda efectivo. En todos los casos analizados, el sistema se mantuvo sobre el umbral de 20 MB/s. Esto valida un buen rendimiento del sistema al poder procesar grandes volúmenes de datos sin comprometer su rendimiento, cumpliendo con las expectativas del proyecto.

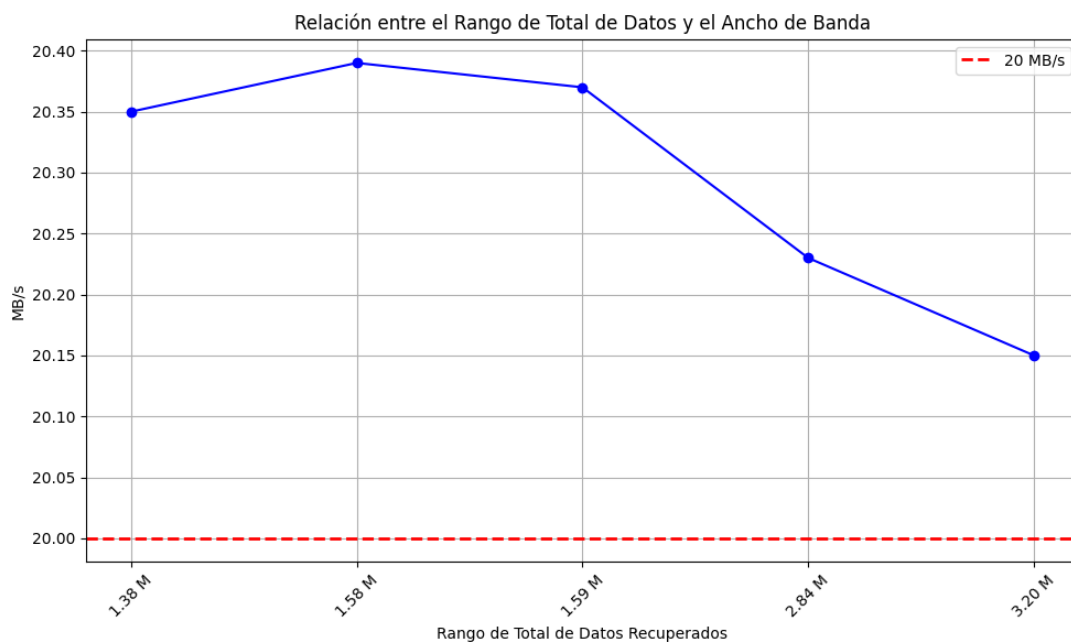


Figura 5.12: Ancho de banda efectivo (MB/s) versus rango de datos recuperados. La línea punteada marca 20 MB/s.

Ingesta por minuto (*Raw_data*): En la Figura 5.13 se observa como se contabilizan las inserciones por minuto en la colección raw para evaluar la continuidad de llegada.

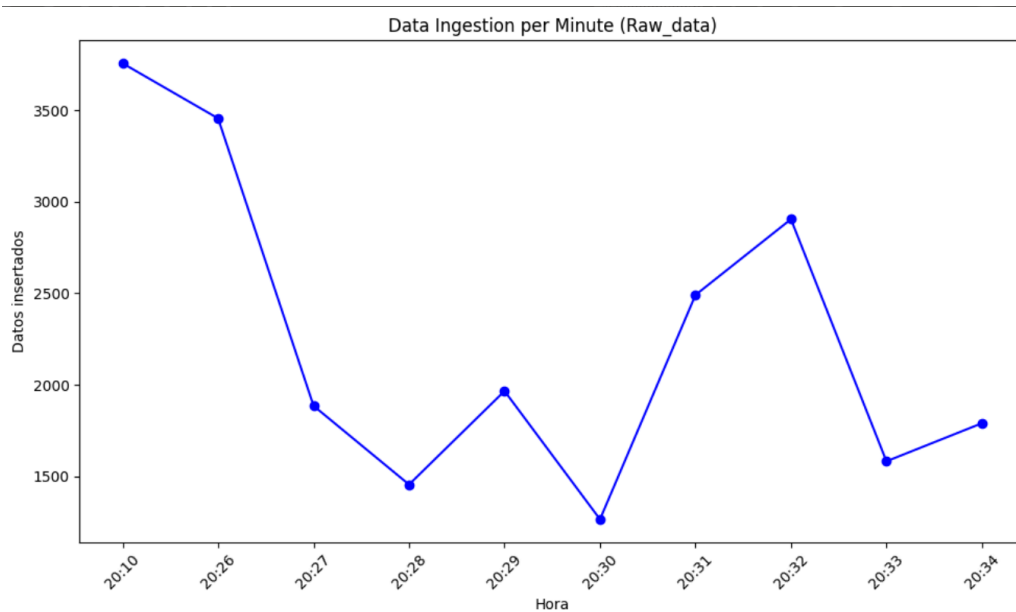


Figura 5.13: Datos ingresados por minuto en *Raw_data*.

Se evidencia en el gráfico un flujo sostenido con variaciones coherentes con la dinámica del mercado. La etapa de captura y publicación se mantiene estable, sin pérdidas observables, y absorbe adecuadamente picos de datos.

Ingesta por minuto (*Structured*): En la Figura 5.14 se muestran las inserciones por minuto en la colección de datos estructurados, con el fin de evaluar el desempeño de la etapa de procesamiento y escritura.

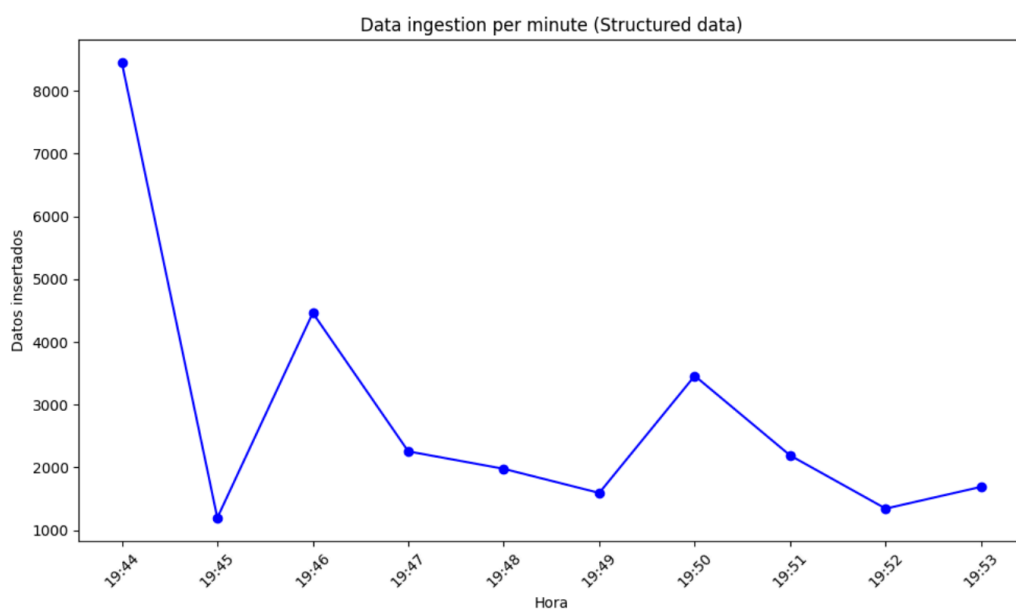


Figura 5.14: Datos ingresados por minuto en *Structured*.

Se evidencia un patrón con picos y valles propio del vaciado de *microbatches*: algunos minutos concentran más registros y otros menos. Aun así, el caudal promedio se mantiene estable, sin minutos en cero ni señales de acumulación, lo que indica que la fase de limpieza y estructuración no introduce cuellos de botella y la escritura se mantiene sin saturación.

5.4.6 Análisis e interpretación

Los resultados confirman que:

- el sistema **supera** el umbral de **20 MB/s** en lectura, incluso con respuestas de gran tamaño.
- la ingesta se mantiene **estable y continua** tanto en *Raw_data* como en *Structured*.
- el comportamiento por *microbatches* de Spark explica la ligera variabilidad minuto a minuto, sin afectar la continuidad.

5.4.7 Evaluación del sistema completo

En conjunto, la plataforma mostró un desempeño consistente tanto en la ingesta en tiempo real como en la recuperación de datos. La integración entre **Apache Kafka**, **Apache Spark** y **MongoDB** sostuvo un flujo continuo sin pérdidas observables, superando el umbral de **20 MB/s** en las pruebas de lectura y manteniendo estabilidad en las inserciones de *Raw_data* y *Structured*. La variabilidad minuto a minuto responde al vaciado de *microbatches* de Spark, sin generar atrasos ni cuellos de botella en escritura. Estos resultados validan que el sistema cumple con los objetivos de rendimiento y disponibilidad definidos para el proyecto.

5.4.8 Contribuciones en las Pruebas de Rendimiento

En esta sección se resumen los aportes personales y en colaboración realizados para medir y validar el rendimiento del sistema, entre ellos se encuentran:

- **Interfaz de consultas para pruebas:** se desarrolla una interfaz que permite seleccionar símbolos y un rango de precios como filtro, evitando traer datos innecesarios y mostrando:
 - cantidad de documentos recuperados,
 - tamaño total de datos recuperados (MB),
 - tiempo total de consulta (s),
 - **ancho de banda efectivo (MB/s)** calculado como:

$$\text{MB/s} = \frac{\text{MB recuperados}}{\text{Tiempo de consulta (s)}}$$

- **Realización de pruebas y análisis de recuperación de datos:** se ejecutaron consultas específicas para validar el *throughput* y comprobar el cumplimiento del objetivo de superar los **20 MB/s**. Se utilizaron datos almacenados previamente, partiendo de un rango inicial de precios de BTC de **60000** y ampliándolo gradualmente en pasos de **2000** hasta llegar a los **70000**. Este esquema permitió evaluar el desempeño del sistema con distintos volúmenes de respuesta.
- **Elaboración y análisis de reportes de resultados:** se generaron visualizaciones que relacionan el volumen recuperado con la tasa (MB/s) e incluyen una línea objetivo de 20 MB/s, además de series de *ingesta por minuto* en *Raw_data* y *Structured*. Estos reportes facilitaron verificar la continuidad del flujo y descartar cuellos de botella en lectura y escritura.

Estas contribuciones permitieron la verificación del rendimiento, asegurando que el sistema cumpliera con el objetivo de 20 MB/s en lectura y mantuviera una ingesta estable, sin pérdidas ni cuellos de botella.

5.5 Conclusión

Este proyecto presentó y validó una plataforma integral para la gestión de *market data* en tiempo real basada en la integración de **Apache Kafka**, **Apache Spark** y **MongoDB** organizada en un *Data Lake* con colecciones *Raw*, *Structured* y *Analytics*. La arquitectura, implementada de forma modular y con bajo acoplamiento, demostró ser capaz de capturar, transformar y almacenar datos de alta frecuencia con continuidad y baja latencia.

Desde el punto de vista de desempeño, las **pruebas de rendimiento** confirmaron dos hitos clave: (i) la **recuperación de datos** supera de manera consistente el umbral objetivo de **20 MB/s**, incluso ante respuestas de gran tamaño, y (ii) la **ingesta por minuto** se mantiene estable tanto en *Raw_data* como en *Structured*. La variabilidad observada minuto a minuto responde al vaciado natural de *microbatches* en Spark, sin evidenciar cuellos de botella ni atrasos en escritura.

Durante el desarrollo se consolidaron prácticas que habilitan la operación y la trazabilidad de extremo a extremo: validadores de esquema e *índices* en MongoDB orientados a consultas por símbolo/tiempo; manejo de idempotencia y control de duplicados; y una **interfaz de consulta** para pruebas que reporta tamaño recuperado, tiempo de respuesta y *ancho de banda efectivo* (MB/s), lo que permite contrastar los resultados con el umbral de 20 MB/s y generar evidencia visual (gráficos de rendimiento e ingesta).

En conjunto, estos resultados muestran que la solución cumple los objetivos propuestos: **ingestión continua**, **procesamiento confiable** y **recuperación eficiente** (≥ 20 MB/s) para soportar analítica en tiempo real, quedando además una base limpia y estructurada para explotación posterior (KPIs, dashboards y análisis históricos).

5.5.1 Trabajo Futuro

Aunque el sistema alcanzó los objetivos planteados, se identifican varias líneas de mejora y extensión:

- **Ampliación de funcionalidades:** incorporar nuevas fuentes de datos y métricas personalizadas en el *dashboard*, adaptándose a distintos perfiles de usuario y escenarios.
- **Optimización del sistema:** aplicar estrategias avanzadas de afinamiento en **Apache Spark** y **MongoDB** (particionamiento, *caching*, compresión, *time-series collections*, índices compuestos) para mejorar el rendimiento bajo mayor concurrencia y volumen.
- **Pruebas en entornos reales:** validar el sistema bajo condiciones de mercado financiero en producción para evaluar desempeño y robustez frente a picos y fallas.
- **Escalabilidad y disponibilidad:** migrar la solución a infraestructuras en la nube con auto escalado y alta disponibilidad (por ejemplo, **AWS** o **Google Cloud**), incorporando balanceo de carga, *checkpointing* distribuido y monitoreo centralizado.
- **Análisis predictivo:** integrar componentes de aprendizaje automático para predicción, detección de anomalías y generación de alertas en tiempo real.
- **Ampliación a otros mercados:** extender el funcionamiento del sistema hacia otros mercados como de bienes raíces, supermercados, entre otros.

Esta migración permitiría mayor elasticidad, seguridad administrada y reducción de tareas operativas, manteniendo o mejorando el costo total de propiedad.

En conjunto, estas líneas de trabajo futuro no solo fortalecerán la capacidad, resiliencia y escalabilidad del sistema, sino que también ampliarán su aplicabilidad a nuevos mercados y casos de uso, reforzando su valor como plataforma para la gestión y analítica de datos en tiempo real.

Bibliografía

- [1] Aldridge, I. (2013). *High-Frequency Trading: A Practical Guide to Algorithmic Strategies and Trading Systems*. Wiley Trading.
- [2] Liu, R., Isah, H., & Zulkernine, F. (2020). *A Big Data Lake for Multilevel Streaming Analytics*. IEEE Access, 8, 167183–167198.
- [3] Jaime, R. A. (2021). *Estudio comparativo entre Apache Flink y Apache Spark*. Repositorio SEDICI, Universidad Nacional de La Plata. Disponible en <https://sedici.unlp.edu.ar/handle/10915/126780>. Fecha de acceso: 29 de septiembre de 2025.
- [4] Velickovska, M., & Gusev, M. (2022). *Comparing AWS Streaming Services: A Use Case on ECG Data Streams*. En *Proc. 45th Int. Conv. on Information, Communication and Electronic Technology (MIPRO 2022)*, Opatija, Croacia, pp. 1387–1392. IEEE. Disponible en: <https://ieeexplore.ieee.org/document/9803359>.

Anexos

Repositorio del Proyecto

El código fuente del sistema desarrollado, junto con los scripts y configuraciones utilizadas, se encuentra disponible en el siguiente repositorio de GitHub:

- <https://github.com/ArielBravoP/OptiDataLake-PMM>

Si bien el repositorio no contiene una guía detallada para la replicación completa del sistema, incluye los elementos principales que pueden ser utilizados como base para su estudio o ampliación.