

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA INFORMÁTICA EN
TECNOLOGÍAS DE LA INFORMACIÓN



Estudio Comparativo de Herramientas de
Procesamiento de Datos en Streaming

TRABAJO FIN DE GRADO

Junio - 2026

AUTOR: Miguel Aparicio Ferrando
DIRECTOR: Jesús Javier Rodríguez Sala

RESUMEN

Este proyecto tiene como objetivo el estudio y comparación de Apache Flink, Apache Kafka Streams y Apache Spark Streaming, tres herramientas destinadas al procesamiento de datos en streaming. Este tipo de procesamiento consiste en procesar los datos de forma continua a medida que estos son generados sin necesidad de almacenarlos previamente para su posterior análisis. A diferencia del procesamiento por lotes, en el que los datos se acumulan durante un período de tiempo y se procesan de manera conjunta, el procesamiento en streaming permite obtener resultados de forma inmediata, lo que resulta útil en escenarios donde la latencia es un factor crítico.

Para realizar esta comparación se desarrollará un sistema con múltiples servicios para generar datos personalizados, recoger, almacenar y visualizar métricas. Dicho sistema se desplegará sobre Kubernetes en Google Cloud Platform. Cada motor procesará los mismos datos bajo los mismos escenarios de prueba. Los resultados se evaluarán en base a la latencia y uso de recursos, con la intención de identificar fortalezas y limitaciones de cada herramienta y determinar cuál resulta más adecuada según el caso de uso.



AGRADECIMIENTOS

Gracias a todas las personas que me han apoyado durante estos años.



ÍNDICE GENERAL

1. INTRODUCCIÓN	10
1.1.- CONTEXTO DEL PROCESAMIENTO EN STREAMING	10
1.2.- JUSTIFICACIÓN DEL PROYECTO	12
1.2.1.- Relevancia actual	12
1.2.2.- Problemática en la selección de herramientas	12
1.2.3.- Impacto en la eficiencia y costes operativos	13
1.3.- OBJETIVOS	13
1.3.1.- Objetivos principales	13
1.3.2.- Objetivos secundarios	14
1.3.3.- Objetivos personales	14
1.4.- LÍMITES DEL PROYECTO	14
2. ANTECEDENTES Y ESTADO DE LA CUESTIÓN	16
2.1.- MODELO CONCEPTUAL DEL PROCESAMIENTO EN STREAMING	16
2.1.1.- Eventos y Data Streams	16
2.1.2.- Operaciones con Data Streams	18
2.1.3.- Gestión del tiempo	20
2.2.- MODELO DE EJECUCIÓN DEL PROCESAMIENTO EN STREAMING	22
2.2.1.- Particionamiento y Paralelización	22
2.2.2.- Gestión del estado y tolerancia a fallos	23
2.3.- TECNOLOGÍAS EMPLEADAS PARA EL PIPELINE COMPLETO	25
2.3.1.- Fuentes de eventos	25
2.3.1.1.- Clasificación de fuentes de eventos	25
2.3.1.2.- Mecanismos de ingestión de eventos	26
2.3.2.- Sistemas de mensajería	26
2.3.2.1.- Mensajería basadas en colas vs. en logs	27
2.3.2.2.- Apache Kafka	28
2.3.2.3.- Amazon Kinesis	29
2.3.3.- Motores de procesamiento de streams	29
2.3.3.1.- Apache FLink	29
2.3.3.2.- Kafka Streams	30
2.3.3.3.- Apache Spark	31
2.3.4.- Destinos de los eventos procesados	32
3. HIPÓTESIS DE TRABAJO	33
3.1.- ARQUITECTURA DE LOS MOTORES DE PROCESAMIENTO	33
3.1.1.- Apache Flink	34
3.1.2.- Apache Spark Streaming	35

3.1.3.- Kafka Streams	36
3.2.- INFRAESTRUCTURA DE MENSAJERÍA Y TRANSPORTE DE DATOS	37
3.2.1.- Apache Kafka	37
3.2.2.- Protocol Buffers	38
3.3.- Lenguajes de programación	38
3.3.1.- Go	38
3.3.2.- Python	39
3.3.3.- Java	39
3.4.- HERRAMIENTAS DE MONITORIZACIÓN	39
3.4.1.- Prometheus	39
3.4.2.- JMX	40
3.4.3.- cAdvisor	40
3.4.3.- Grafana	41
3.4.5.- Redpanda Console	41
3.5.- LIBRERÍAS	41
3.5.1.- franz-go	41
3.5.2.- Seaborn	42
3.5.3.- Pandas	42
3.6.- HERRAMIENTAS DE DESPLIEGUE	42
3.6.1.- Kubernetes	42
3.6.2.- Google Cloud Platform	43
3.7.- HERRAMIENTAS DE DESARROLLO	44
3.7.1.- Docker	44
3.7.2.- Kind	45
3.7.3.- Maven	45
3.8.- OTRAS HERRAMIENTAS	46
3.8.1.- Visual Studio Code	46
3.8.2.- Claude Code	46
4. METODOLOGÍA Y RESULTADOS	47
4.1.- Tareas y planificación	47
4.2.- Diseño de las pruebas	48
4.2.1.- Datos de entrada	48
4.2.2.- Job de filtro sin estado	50
4.2.3.- Job de agregado por clave	51
4.2.4.- Job de agregado por ventana	52
4.2.5.- Métricas utilizadas	52
4.2.6.- Escenarios	53
4.3.- Diseño del sistema	55
4.3.1.- Motores de procesamiento	55
4.3.2.- Productor	56
4.3.3.- Consumidor	57

4.3.4.- Coordinación entre componentes	58
4.3.5.- Recolección de métricas	59
4.4.- Infraestructura	59
4.4.1.- Construcción de imágenes	59
4.4.2.- Plataforma sobre Kubernetes	60
4.4.3.- Infraestructura local	61
4.4.4.- Infraestructura en Google Cloud Platform (GCP)	61
4.5.- Resultados	63
4.5.1.- Variación del número de eventos	63
4.5.1.1.- Filtro sin estado	63
4.5.1.2.- Agregado por clave	67
4.5.1.3.- Agregado por ventana	73
4.5.2.- Variación del tamaño de evento	75
4.5.3.- Variación del número de claves	78
 5. CONCLUSIONES Y TRABAJO FUTURO	 82
5.1.- CONCLUSIONES	82
5.2.- POSIBLES DESARROLLOS FUTUROS	83
 Bibliografía	 85
 Anexo. INSTALACIÓN Y CONFIGURACIÓN	 95
A.1.- SERVICIOS EN GO	95
A.1.1.- Instalación de Go	95
A.1.2.- Instalación de protoc y protoc-gen-go	96
A.1.3.- Compilación	96
A.2.- JOBS EN JAVA	96
A.2.1.- Instalación de Java	96
A.2.2.- Instalación de Maven	97
A.2.3.- Compilación de Jobs	97
A.3.- CONTENEDORIZACIÓN	97
A.3.1.- Instalación de Docker	98
A.3.2.- Construcción de imágenes	98
A.4.- ENTORNO LOCAL CON KIND	101
A.4.1.- Instalación de Kind y Kubectl	101
A.4.2.- Creación del clúster	101
A.4.3.- Despliegue de imágenes	102
A.4.4.- Instalación de Helm	102
A.4.5.- Instalación de operadores	103
A.5.- DESPLIEGUE EN GOOGLE CLOUD PLATFORM	104
A.5.1.- Creación del proyecto	104
A.5.2.- Instalación de gcloud	105

A.5.3.- Creación de registro y subida de imágenes	106
A.5.4.- Creación y configuración de Google Kubernetes Engines	107
A.6.- OPERATIVIDAD DEL CLÚSTER	108
A.6.1.- Despliegue de Kafka	108
A.6.2.- Despliegue de la observabilidad	108
A.6.3.- Ejecución de un experimento	108
A.6.4.- Recolección y extracción de resultados	109
A.6.5.- Monitorización	110



ÍNDICE DE TABLAS

Tabla 2.1 Comparación entre el modelo push y pull	26
Tabla 2.1 Comparación entre los distintos motores de procesamiento en Streaming	31
Tabla 4.1 Variación del throughput	54
Tabla 4.2 Variación del tamaño de evento	54
Tabla 4.3 Variación del número de claves distintas	54
Tabla 4.4 Infraestructura de Google Cloud Platform	62
Tabla 4.5 Latencia extremo a extremo del job filtro sin estado	64
Tabla 4.6 Latencia extremo a extremo del job de agregado por clave	68
Tabla 4.7 Latencia extremo a extremo ante la variación del tamaño de evento	76
Tabla 4.8 Latencia extremo a extremo ante la variación del número de claves distintas	79
Tabla A.1 Parámetros de ejecución de pruebas	109



ÍNDICE DE FIGURAS

Figura 1.1 Esquema genérico de procesamiento de datos en streaming	11
Figura 2.1 Ejemplo de evento de dispositivo IoT en formato JSON	17
Figura 2.2 Flujo continuo de eventos sobre el tiempo	17
Figura 2.3 Componentes principales de un grafo de Dataflow	18
Figura 2.4 Diagrama de operación stateless de filtro con valores mayor a 100	18
Figura 2.5 Diagrama ejemplificativo de los tres tipos de ventana	19
Figura 2.6 Comparación del tiempo de evento y el tiempo de procesamiento	20
Figura 2.7 Cálculo de Watermark mediante el modelo de desfase máximo acotado	21
Figura 2.8 Representación del watermark y el desfase en el tiempo	21
Figura 2.9 Modelo lógico vs. Modelo físico	22
Figura 2.10 Estrategías de particionamiento	23
Figura 2.11 Mensajería basada en colas	27
Figura 2.12 Mensajería basada en el Log	27
Figura 2.13 Arquitectura de Apache Kafka	28
Figura 3.1 Arquitectura interna de Apache Flink	34
Figura 3.2 Arquitectura interna de Apache Spark	35
Figura 3.3 Arquitectura interna de Kafka Streams	37
Figura 3.4 Arquitectura de clúster de Kubernetes	43
Figura 3.5 Servicios ofrecidos por GCP	44
Figura 4.1 Diagrama de Gantt de la planificación del proyecto	48
Figura 4.2 Definición del evento InputEvent	49
Figura 4.3 Definición del evento FilteredEvent	51
Figura 4.4 Definición del evento KeyedAggregate	51
Figura 4.5 Definición del evento WindowedAggregate	52
Figura 4.6 Diagrama de componentes del sistema	55
Figura 4.7 Configuración del productor de Kafka	56
Figura 4.8 Lógica de generación de eventos en base a un throughput determinado	57
Figura 4.9 Coordinación del productor y consumidor mediante tópicos	58
Figura 4.10 Dockerfile de Flink	60
Figura 4.11 Configuración del clúster local de Kind	61
Figura 4.12 Uso de CPU en el filtro sin estado frente al throughput	66
Figura 4.13 Uso de memoria en el filtro sin estado frente al throughput	66
Figura 4.14 Uso de CPU en el agregado por clave frente al throughput	69
Figura 4.15 Tráfico de red de Flink en el agregado por clave a 200.000 eventos/s	70
Figura 4.16 Evolución del lag del consumidor de Flink con 100.000 eventos/s	70
Figura 4.17 Evolución del lag del consumidor de Flink a 200.000 eventos/s	71
Figura 4.18 Uso de memoria en el job de agregado por clave frente al throughput	71
Figura 4.19 Escritura en disco en el job de agregado por clave frente al throughput	72

Figura 4.20 Uso de CPU en el job de agregado por ventanas frente al throughput	73
Figura 4.21 Uso de memoria en el job de agregado por ventanas frente al throughput	74
Figura 4.22 Escritura en disco en el job de agregado por ventanas frente al throughput	75
Figura 4.23 Uso de CPU en el job de agregado por clave frente al tamaño del evento	76
Figura 4.24 Uso de memoria en job de agregado por clave frente al tamaño de evento	77
Figura 4.25 Escritura en disco en job de agregado por clave frente al tamaño de evento	78
Figura 4.26 Uso de CPU en job de agregado por clave frente a número claves distintas	80
Figura 4.27 Uso de memoria vs. claves distintas en job de agregado por clave	80
Figura 4.28 Escritura en disco vs. claves distintas en job de agregado por clave	81
Figura A.1 Instalación de Golang	95
Figura A.2 Instalación de herramientas de Protobuf	96
Figura A.3 Instalación de Java	97
Figura A.4 Instalación de Docker	98
Figura A.5 Dockerfile del productor	98
Figura A.6 Dockerfile de Flink	99
Figura A.7 Dockerfile de Kafka Streams	100
Figura A.8 Listado de imágenes de Docker	100
Figura A.9 Instalación de kubectf	101
Figura A.10 Listado de clúster y nodos de Kind	101
Figura A.11 Inyección de imágenes en Kind	102
Figura A.12 Instalación de Strimzi	103
Figura A.13 Instalación del operador de Flink	103
Figura A.14 Instalación del operador de Spark	104
Figura A.15 Instalación del operador kube-prometheus-spark	104
Figura A.16 Creación de proyecto en Google Cloud Platform	105
Figura A.17 Instalación de gcloud	105
Figura A.18 Autenticación de gcloud	106
Figura A.19 Creación del repositorio de imágenes	106
Figura A.20 Imágenes en el repositorio de Artifact Registry	106
Figura A.21 Aprovisionamiento del clúster en GCP	107
Figura A.22 Obtención de claves del clúster	107
Figura A.23 Despliegue de herramientas de observabilidad	108
Figura A.24 Panel de Kafka en Grafana	110

Capítulo 1

Introducción

1.1.- CONTEXTO DEL PROCESAMIENTO EN STREAMING

Históricamente, la arquitectura de los entornos de tratamiento de información se ha basado en el modelo de procesamiento por lotes. Bajo este enfoque, los datos primero se almacenan para ser procesados posteriormente [1]. El sistema no trabaja en tiempo real, sino que espera a acumular un volumen determinado de información o cumplir un intervalo de tiempo para ejecutar las tareas de computación [2].

A pesar de su eficiencia en el tratamiento de volúmenes masivos, esta metodología impone un tiempo de espera que hace imposible la inmediatez, obligando a las organizaciones a basar sus decisiones en datos históricos, sin disponer de la capacidad técnica para intervenir en los eventos mientras ocurren [3].

Como respuesta a estas limitaciones surge el procesamiento en streaming. A diferencia del modelo por lotes, este enfoque rompe la secuencia de ‘almacenar y procesar’, ya que trata la información como un flujo continuo e ininterrumpido donde se procesa conforme va llegando, permitiendo una respuesta inmediata ante cada evento individual [4].

Este cambio de paradigma responde a la naturaleza de las fuentes de información modernas. Actualmente, la generación de datos no ocurre en bloques aislados ni en intervalos definidos, sino que se manifiesta como una sucesión infinita de eventos provenientes de sensores industriales, mercados financieros, transacciones, dispositivos móviles o interacciones en plataformas digitales. Intentar adaptar estos flujos ininterrumpidos al modelo tradicional resulta ineficiente [5], ya que introduce una latencia innecesaria que deprecia el valor de la información.

Para soportar este ritmo de procesamiento, es fundamental contar con una infraestructura diseñada específicamente para el flujo continuo. Resulta vital asegurar la alta disponibilidad y la resiliencia ante errores, ya que el sistema debe ser capaz de permanecer operativo frente a posibles fallos para evitar la pérdida de datos en tránsito. En este contexto, la paralelización no solo facilita la gestión de cargas masivas, sino que proporciona la escalabilidad necesaria para adaptar el procesamiento a las demandas actuales de información.

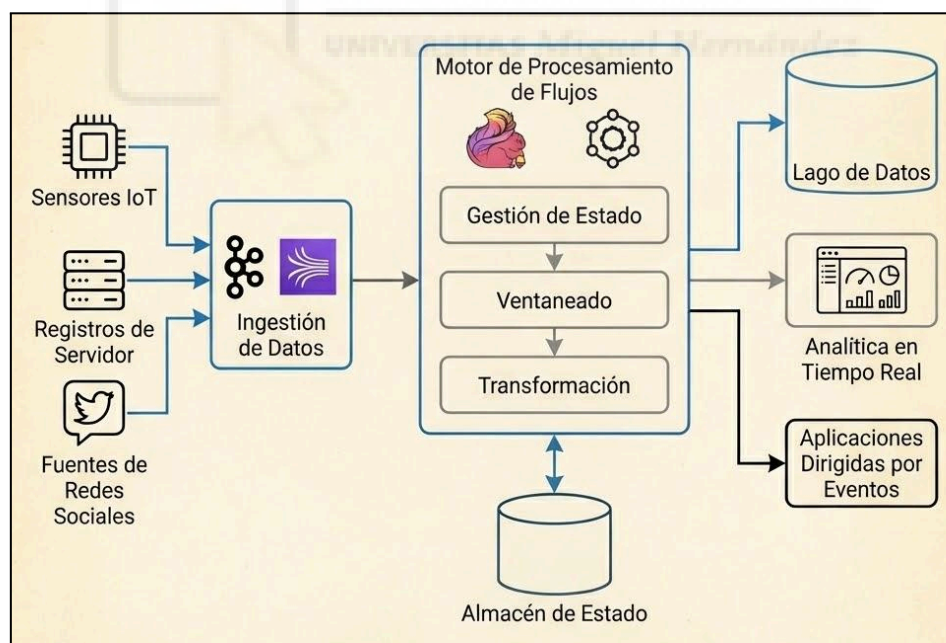


Figura 1.1: Esquema genérico de procesamiento de datos en streaming

Esta evolución técnica ha derivado en un enfoque que busca eliminar la barrera histórica entre el tratamiento de datos en streaming y por lotes. Bajo esta visión, el procesamiento en lotes deja de ser un paradigma independiente para entenderse simplemente como un caso particular del streaming, donde el flujo de datos tiene un inicio y un fin definidos [6]. Esta

arquitectura unificada permite a las organizaciones simplificar su arquitectura tecnológica, utilizando el mismo motor de ejecución y una lógica de programación única para analizar tanto eventos históricos como datos en vivo.

Resulta fundamental distinguir entre el procesamiento en streaming y el procesamiento en tiempo real. Mientras que el primero define una arquitectura para tratar flujos continuos de datos, el segundo se refiere estrictamente al cumplimiento de plazos de respuesta garantizados. Así, el streaming es el medio técnico que permite alcanzar bajas latencias, pero no todo sistema de streaming es necesariamente un sistema de tiempo real, ya que este último depende de los requisitos de inmediatez del caso de uso [7].

1.2.- JUSTIFICACIÓN DEL PROYECTO

1.2.1.- Relevancia actual

La importancia de este proyecto radica en el cambio de paradigma que vive actualmente la ingeniería de datos, donde el modelo de procesamiento por lotes ya no es suficiente. En un mercado globalizado y digitalizado, el valor de la información disminuye con el paso del tiempo. Por ello, la capacidad de procesar datos en el instante en que se generan se ha convertido en una necesidad estratégica. Este proyecto analiza las tecnologías que permiten transformar los flujos de información en respuestas de valor.

Más allá de la estrategia de negocio, esta necesidad de inmediatez resulta vital en sectores donde el tiempo de reacción determina el éxito o fracaso de una operación. En el ámbito de las finanzas y la ciberseguridad, por ejemplo, la detección de un fraude bancario o de un ciberataque debe ocurrir de manera instantánea para ser efectiva. Del mismo modo, en el Internet de las Cosas (IoT) y la industria 4.0, la monitorización continua de sensores permite realizar mantenimientos preventivos en infraestructuras críticas, evitando paradas de servicio que podrían suponer costes elevados.

1.2.2.- Problemática en la selección de herramientas

El desafío actual no es la falta de opciones, sino las grandes diferencias en cómo están construidas internamente. Bajo la etiqueta común de procesamiento en streaming, conviven herramientas con diseños técnicos muy distintos. Esta diversidad crea una complicación importante: si no se comprende cómo trabaja cada herramienta internamente, es muy difícil acertar a la hora de diseñar sistemas de procesamiento en streaming.

La ausencia de un análisis comparativo podría derivar en una selección de tecnología basada en tendencias de mercado en lugar de requisitos técnicos objetivos. Esto conlleva el riesgo de implementar infraestructuras ineficientes, con costes operativos imprevistos o una incapacidad de resolver la problemática inicial, especialmente en escenarios con requisitos técnicos de alta exigencia.

1.2.3.- Impacto en la eficiencia y costes operativos

La elección de una plataforma de procesamiento no solo es una decisión funcional, sino también económica. Cada herramienta gestiona el cómputo y la memoria de manera distinta, mientras algunas priorizan el rendimiento bruto y la baja latencia, otras están optimizadas para minimizar la sobrecarga operativa. Un diseño ineficiente puede derivar en un consumo excesivo de recursos en la nube o en el centro de datos, elevando los costes de infraestructura de forma innecesaria.

Asimismo, resulta fundamental evaluar la escalabilidad y mantenibilidad. Implementar una solución excesivamente compleja para un caso de uso sencillo aumenta la “deuda técnica” y el tiempo que el equipo debe dedicar a tareas de mantenimiento y resolución de errores. Por el contrario, una herramienta que no escale correctamente ante incrementos en el volumen de datos obligará a realizar migraciones costosas en el futuro.

Finalmente, es crucial considerar el impacto del tiempo de desarrollo. La facilidad de uso, documentación y madurez de las herramientas influyen directamente en la productividad. Analizar estos factores permite a las organizaciones no solo ahorrar en costes directos de hardware, sino también el esfuerzo humano necesario para desplegar y operar estos sistemas.

1.3.- OBJETIVOS

El propósito de este proyecto es presentar un análisis y comparación técnica de las principales tecnologías para el procesamiento de datos en streaming con la finalidad de establecer un marco de referencia que permita identificar los escenarios de uso óptimos para cada herramienta.

1.3.1.- Objetivos principales

El objetivo central consiste en realizar una comparativa técnica entre Apache Flink, Kafka Streams y Spark Streaming, evaluando sus arquitecturas bajo criterios de paralelización,

tolerancia a fallos, latencia y uso de recursos. Dicho estudio se validará mediante un caso práctico experimental.

1.3.2.- Objetivos secundarios

- Investigar las arquitecturas internas de Apache Flink, Kafka Streams y Spark Streaming.
- Desarrollar una aplicación de procesamiento en streaming común que sirva como base para el caso práctico comparativo.
- Establecer métricas de evaluación técnica para cuantificar el desempeño de cada herramienta.
- Someter a los sistemas a escenarios de estrés y evaluar la escalabilidad.
- Evaluar la curva de aprendizaje y la complejidad operativa de cada herramienta.
- Desarrollar un sistema capaz de evaluar las tres herramientas
- Clasificar las herramientas en función de su caso de uso.

1.3.3.- Objetivos personales

A nivel personal, este proyecto surge por el interés en la rama del procesamiento en streaming, una disciplina que considero fundamental ante el crecimiento exponencial del volumen de datos global. Mi objetivo es profundizar en esta rama de conocimiento, motivado por la convicción de que la capacidad de obtener información instantánea será un pilar indispensable para la competitividad y la toma de decisiones en las organizaciones del futuro.

1.4.- LÍMITES DEL PROYECTO

El proyecto se centra exclusivamente en el análisis de tres tecnologías líderes en el mercado: Apache Flink, Kafka Streams y Spark Streaming. No se incluirán en la comparativa servicios propietarios de proveedores de nube (como Amazon Kinesis o Google Cloud Dataflow) para asegurar que el estudio se base en soluciones de código abierto.

El desarrollo se limitará a la implementación de una lógica de negocio común que incluya operaciones esenciales del streaming, tales como agregaciones, ventanas temporales y gestión de estado. Quedan fuera del alcance del proyecto:

- La implementación de medidas de seguridad avanzadas.
- La optimización específica a cada hardware.
- El alcance se limita al procesamiento de datos, por lo que la capa de presentación se restringirá a dashboards funcionales para la visualización de métricas.



Capítulo 2

Antecedentes y estado de la cuestión

2.1.- MODELO CONCEPTUAL DEL PROCESAMIENTO EN STREAMING

2.1.1.- Eventos y Data Streams

En el modelo de streaming, la información se representa mediante eventos, donde cada evento representa la ocurrencia de un suceso en un instante determinado con unos datos asociados [8]. Por ejemplo, una interacción de un usuario con una aplicación web, una transacción, los logs de una aplicación o una medida generada por un sensor se pueden representar como eventos.

Los eventos se estructuran en tres componentes principales [9]:

- **Timestamp**, que indica cuando el evento ha ocurrido.
- **Clave**, que representa un identificador para asociar el evento a una identidad concreta, como por ejemplo a un usuario o dispositivo.
- **Payload**, que contiene los datos del evento.

Un ejemplo de evento en formato JSON se puede ver representado en la Figura 2.1.

```
{
  "key": "sensor_01",
  "timestamp": "2026-01-15T10:32:06Z",
  "payload": {
    "temperatura": 22.5,
    "humedad": 60,
    "estado": "OK"
  }
}
```

Figura 2.1: Ejemplo de evento de dispositivo IoT en formato JSON

Es importante tener en cuenta que los eventos son inmutables, es decir, una vez que son registrados ya no pueden ser modificados ni borrados [10]. Esto significa que, si queremos actualizar el estado de un elemento, la acción que hay que realizar será enviar otro evento que represente el nuevo estado. Esto asegura que cada cambio de estado esté registrado, haciendo que el conjunto de todos los eventos represente un historial completo de los datos, siendo este la denominada “fuente de la verdad” [11].

Los eventos generados por las fuentes forman flujos de eventos llamados data streams, que se consideran datasets no acotados debido a que no tienen un fin definido pudiendo ser potencialmente infinitos [7]. La Figura 2.2 muestra la naturaleza de estos, donde cada nuevo evento se integra en el stream, formando un conjunto en constante expansión a medida que transcurre el tiempo.

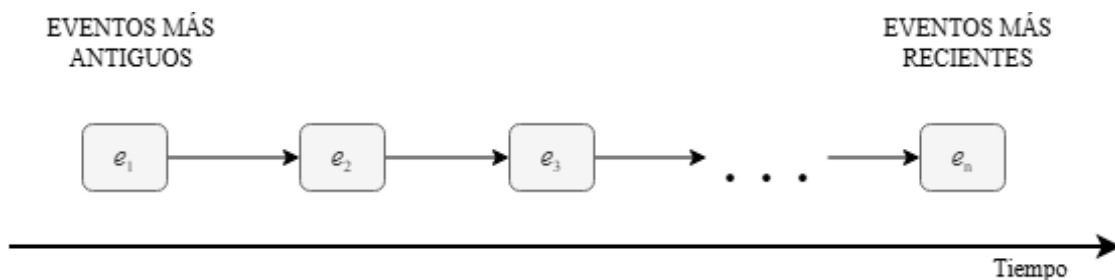


Figura 2.2: Flujo continuo de eventos sobre el tiempo

2.1.2.- Operaciones con Data Streams

Para poder analizar y procesar estos data streams haremos uso del modelo de Dataflow [3], donde los programas se representan con grafos donde las aristas representan data streams y los vértices pueden tener una de estas tres funciones [12]:

- **Origen**, es el punto de entrada del sistema, su función es conectar una fuente externa y enviar el data stream al siguiente operador.
- **Operador**, su función es transformar el data stream mediante funciones como filtrado, agregación o unión, las cuales producen otro data stream. Se pueden encadenar múltiples operadores.
- **Sumidero**, es el destino final del data stream para ser enviado en un almacenamiento externo, base de datos, sistema de archivos o ser consumido por otras aplicaciones.

En la Figura 2.3 se observa un ejemplo de un grafo que representa el modelo de Dataflow para el procesamiento en Streaming.



Figura 2.3: Componentes principales de un grafo de Dataflow

Dependiendo de cómo los operadores procesan los eventos se pueden clasificar en dos tipos: *stateless* o *stateful*.

Los operadores *stateless* procesan cada evento de manera independiente, sin almacenar información de los eventos procesados anteriormente, por lo que el resultado solo depende del evento que se está procesando [13]. En la Figura 2.4 se ilustra un ejemplo de la transformación *filter* donde el nuevo stream solo tendrá elementos mayor a 100.



Figura 2.4: Diagrama de operación *stateless* de filtro de elementos con valor mayor a 100

Por otro lado, en las operaciones stateful, las transformaciones requieren mantener el estado de eventos anteriores para obtener los resultados [12]. Las transformaciones comunes son las agregaciones, como calcular la suma de todos los eventos que han llegado o la media.

Los operadores stateful nos permiten trabajar con el tiempo mediante la técnica de *windowing* que consiste en dividir el data stream en lotes temporales [14]. Esto permite hacer consultas del tipo ¿Cuántos usuarios ha habido en los últimos diez minutos?, ¿Cuántas veces se ha vendido una acción determinada en las tres últimas horas? Para resolver los distintos casos de uso existen tres tipos de ventanas [7]:

- **Ventanas fijas**, que segmentan los eventos en bloques exactos donde cada evento solo pertenece a un bloque.
- **Ventanas deslizantes**, que agrupa los eventos en intervalos de tiempo de tamaño fijo que se desplazan periódicamente, provocando que los eventos puedan pertenecer a varias ventanas.
- **Ventanas de sesión**, que agrupa los eventos que ocurren de forma consecutiva y finalizan cuando existe un periodo de inactividad mayor a un umbral definido. De esta manera, cada ventana representa una sesión de actividad.

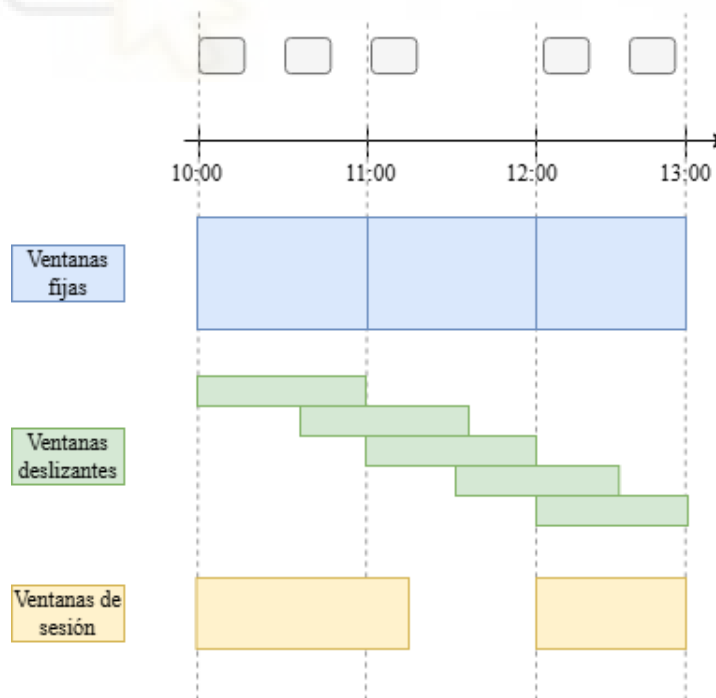


Figura 2.5: Diagrama ejemplificativo de los tres tipos de ventanas

En la Figura 2.5 se aprecian los tres tipos de ventanas. Primero, las ventanas fijas que se calculan cada hora. Por otra parte, las ventanas deslizantes donde cada media hora se hace una ventana de la última hora. Por último, están las ventanas de sesión que se cierran cuando hay un periodo de inactividad de 45 minutos.

2.1.3.- Gestión del tiempo

A la hora de trabajar con ventanas hay que tener en cuenta dos nociones del tiempo [7], el tiempo de evento, que representa en qué instante originalmente ha ocurrido, y el tiempo de proceso, que es el instante en el que el evento ha sido observado por el sistema.

En la Figura 2.6 se puede observar una comparación entre estos dos conceptos. La línea discontinua representa la situación ideal (ambos tiempos son iguales), pero en la realidad esto no se produce debido a retrasos en la red, cargas altas en los sistemas y eventos que llegan tarde, provocando un desfase temporal entre el instante en el que el evento se produce y el instante en el que se procesa.

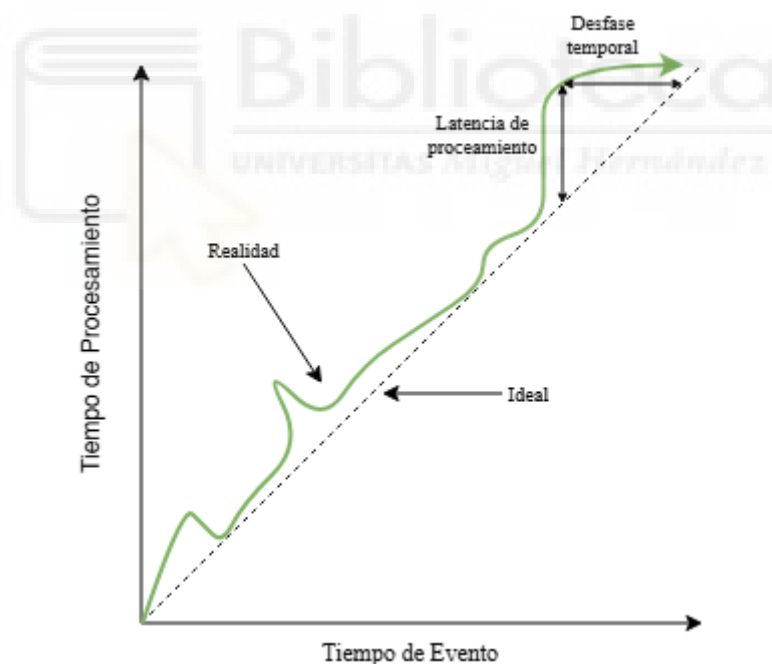


Figura 2.6: Comparación del tiempo de evento y el tiempo de procesamiento [7].

Tener estas dos nociones de tiempo significa que si utilizamos ventanas basadas en el tiempo de procesamiento, no hay eventos que lleguen fuera de orden, ya que el orden viene determinado por el momento en el que se produce. Por el contrario, si se usan ventanas basadas en el tiempo de evento, estos podrían llegar tarde. Por ello, surge la pregunta: ¿Cómo se sabe que todos los eventos se han recibido para una determinada ventana de tiempo?

Para resolver esto se utiliza el mecanismo de *watermarks*, que consiste en enviar indicadores con un instante de tiempo t al pipeline que aseguran que han llegado todos los eventos más antiguos a este instante t . Hay múltiples estrategias para generar *watermarks*. Por ejemplo, cada cierto intervalo de tiempo definido se podría generar una *watermark* determinando un desfase permitido según la fórmula de la Figura 2.7 (ilustrada visualmente en la Figura 2.8).

$$\text{Watermark} = \text{Max}(\text{Tiempo de Evento}) - \text{Desfase}$$

Figura 2.7: Cálculo de Watermark mediante el modelo de desfase máximo acotado

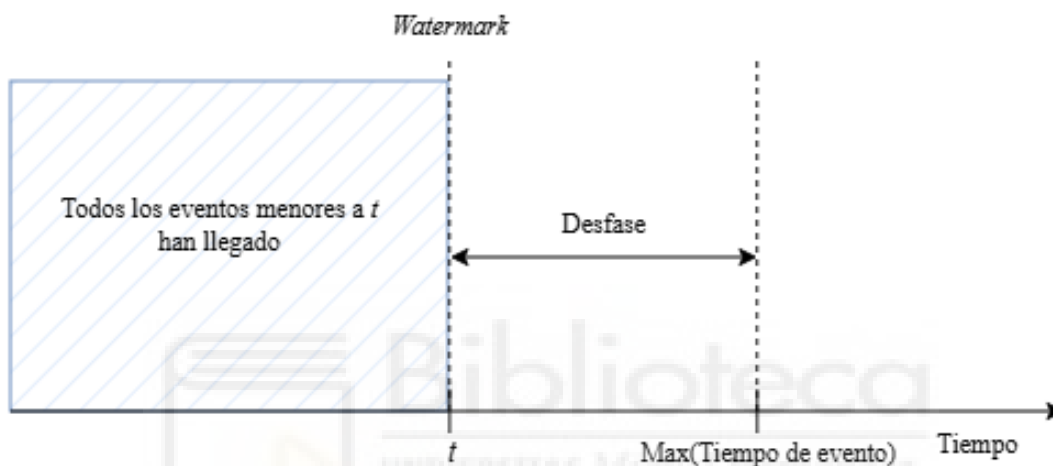


Figura 2.8: Representación del watermark y el desfase en el tiempo

De esta manera, si hay una ventana que cierra en el instante T , el sistema procederá a materializar la ventana tan pronto como se emita un *watermark* que iguale o supere el instante T . Sin embargo, los *watermarks* no garantizan que no haya eventos más antiguos al tiempo que marca el *watermark*. Por este motivo, debe entenderse como una estimación heurística sobre el progreso del pipeline.

Por otra parte, la elección del método de generación de *watermarks* introduce un compromiso entre latencia y completitud de los datos. Si los *watermarks* se generan con un desfase pequeño, las ventanas se materializarían con una baja latencia, pero aumentando la posibilidad de que haya eventos que lleguen tarde y se queden fuera. Por el contrario, si se utilizan *watermarks* con desfases mayores, el sistema puede esperar más tiempo aumentando la probabilidad de incluir todos los eventos, lo que implica tener los resultados con una latencia mayor.

2.2.- MODELO DE EJECUCIÓN DEL PROCESAMIENTO EN STREAMING

2.2.1.- Particionamiento y Paralelización

En los sistemas de procesamiento en streaming, para poder manejar grandes volúmenes de eventos de forma eficiente, los data stream suelen dividirse en múltiples particiones. Estas particiones permiten que diferentes instancias de un mismo operador procesen subconjuntos del stream de manera independiente, lo que facilita el paralelismo y la escalabilidad horizontal del sistema. Por esta razón, se distingue entre el modelo lógico y el físico, donde ambos son representados como un grafo acíclico dirigido. El modelo lógico, representa las transformaciones a aplicar al data stream sin considerar el paralelismo. Por otra parte, el modelo físico representa cómo se ejecutaría este grafo de manera distribuida teniendo en cuenta el paralelismo [12]. La Figura 2.9 representa esta situación.

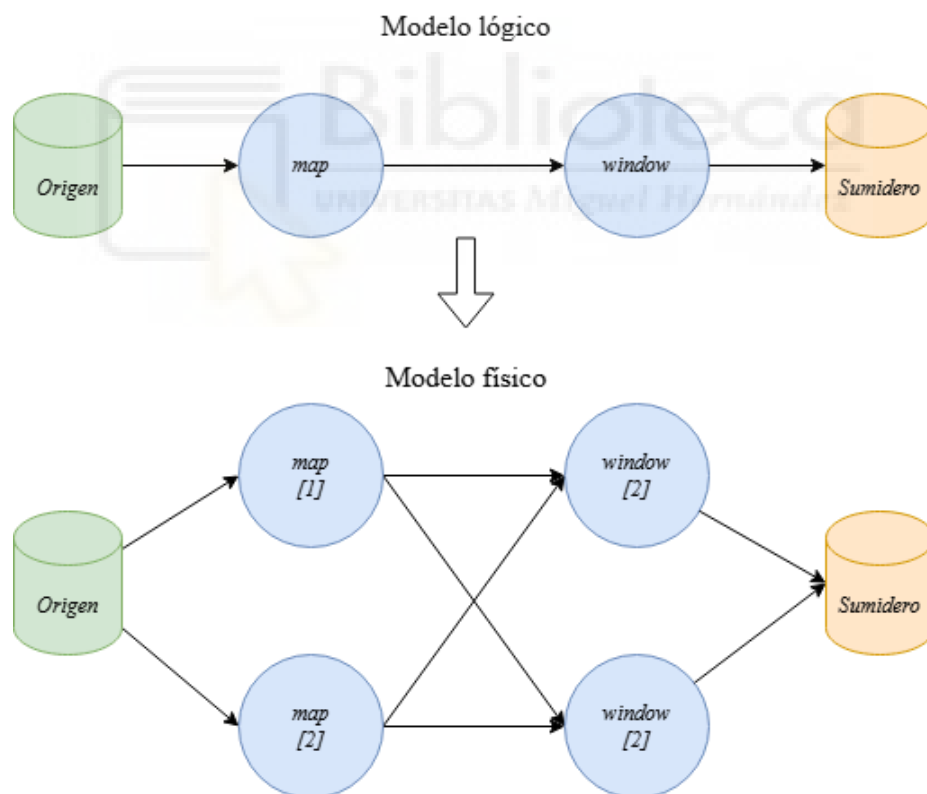


Figura 2.9: Modelo lógico vs. Modelo físico

Existen diversas estrategias de particionamiento entre operadores. Una de las más comunes es el particionamiento por clave, que garantiza que todos los eventos que comparten el mismo identificador se envíen a la misma instancia del operador. Esto resulta

especialmente importante en transformaciones con estado, como agregaciones o ventanas, donde es necesario mantener el estado asociado a cada clave de manera consistente.

Además del particionamiento por clave, existen otras estrategias como la distribución round-robin, que reparte los eventos de forma secuencial entre las distintas instancias, la distribución aleatoria uniforme, que asigna cada evento a una partición al azar, el broadcast, donde cada evento se envía a todas las instancias del operador, o incluso esquemas de particionamiento definidos mediante lógica personalizada [15]. En la Figura 2.10 se aprecian los distintos tipos de estrategias.

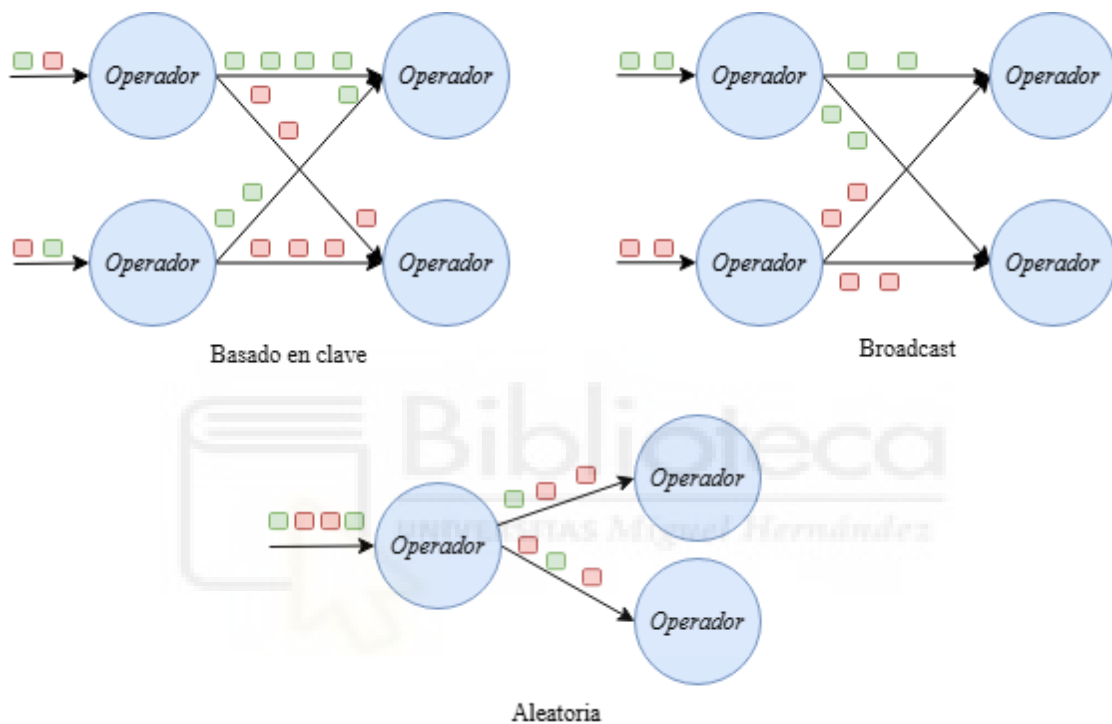


Figura 2.10: Estrategias de particionamiento

2.2.2.- Gestión del estado y tolerancia a fallos

Anteriormente, se introdujo que los operadores mantienen un estado interno para las transformaciones como las agregaciones o ventanas. En el modelo físico, cuando los operadores se paralelizan y hay varias instancias, cada una de estas instancias mantiene su propio estado local interno [18].

Para gestionar este estado, los motores de streaming utilizan *state backends*, que determinan cómo el estado es almacenado, organizado y accedido [12]. Se puede implementar mantenimiento el estado directamente en memoria, permitiendo accesos muy rápidos. Sin embargo, debido a que el tamaño del estado puede crecer considerablemente,

es preferible implementar almacenamientos basados en disco o bases de datos embebidas como por ejemplo RocksDB [16].

El uso del estado implica que si un nodo falla, se pierde el estado, lo que provoca que se obtengan resultados incorrectos. Por esta razón, los sistemas de procesamiento en streaming han de incorporar mecanismos de tolerancia a fallos, es decir, permitir que el sistema continúe funcionando sin interrupción en presencia de fallos de red, hardware o software.

Para solucionar esto, se utilizan mecanismos de *snapshots* del estado, donde periódicamente se hace una copia del estado y se almacena de manera persistente. De esta forma, en caso de error, se puede reiniciar la ejecución desde el último *snapshot*. Estos snapshots no se almacenan únicamente en el nodo de procesamiento, sino que se almacenan en sistemas de almacenamiento distribuidos y duraderos como Amazon S3 o *Hadoop Distributed File System* (HDFS), que permiten almacenar grandes volúmenes de datos con redundancia y accesibles de manera sencilla [2]. De esta manera, en caso de fallo de un nodo o del sistema entero, el sistema se recuperaría utilizando los snapshots que residen en el almacenamiento externo.

Otras de las medidas que ofrecen robustez al sistema son las semánticas de entrega, que definen la fiabilidad de que los eventos del data stream sean procesados en presencia de fallos. Estas garantías determinan si los eventos pueden perderse, procesarse múltiples veces o que el evento solo se procese una vez para el resultado final. Se distinguen tres tipos de garantías [17]:

- ***At-least-once***, donde un evento será procesado una o más veces, esto provoca que haya eventos duplicados.
- ***At-most-once***, donde un evento se procesará ninguna vez o una.
- ***Exactly-once***, donde ningún evento se perderá y tampoco será duplicado, solo se procesará una vez.

Hay que destacar que las semánticas de entrega no dependen únicamente del motor de procesamiento, sino también de las fuentes de donde se obtienen los datos y los sumideros donde se guardan [12]. Por otra parte, elegir una garantía de entrega con mayor fiabilidad como *exactly-once* supone comprometer la latencia y la complejidad debido a que se tienen que añadir mecanismos extras.

2.3.- TECNOLOGÍAS EMPLEADAS PARA EL PIPELINE COMPLETO

2.3.1.- Fuentes de eventos

Las fuentes de eventos son el punto de partida del pipeline del procesamiento en streaming, ya que son las responsables de aportar la información que posteriormente será analizada por el sistema de datastream. Para diseñar adecuadamente el pipeline, es fundamental una correcta identificación del tipo de fuente a que nos enfrentamos y la técnica concreta de propagación de los datos.

2.3.1.1.- Clasificación de fuentes de eventos

Aunque no existe una clasificación aceptada para los eventos, es posible identificar distintas categorías en función del origen de los eventos. En muchos casos, los eventos se originan como consecuencia del funcionamiento de aplicaciones. Donde, los eventos representan acciones o transiciones de estado, como las interacciones de los usuarios en aplicaciones web o móviles siendo estas conocidas como *clickstreams*, la ejecución de pedidos, generación de logs o métricas. Estas fuentes son especialmente relevantes dado el volumen y la frecuencia con la que se generan.

Adicionalmente, pueden ser fuentes externas que generan información procedente del entorno como por ejemplo los dispositivos IoT que producen lecturas de sensores, sistemas de telemetría en infraestructuras distribuidas o proveedores externos que ofrecen datos en tiempo real, como servicios financieros o de geolocalización. A menudo, estas fuentes pueden estar fuera del control directo de la organización.

Por último, no todos los sistemas producen eventos de forma nativa. En particular, muchos sistemas mantienen únicamente el estado actual de la información, como es el caso de las bases de datos relacionales. En estos casos, los eventos deben obtenerse de manera indirecta a partir de los cambios que ocurren en dicho estado. Una estrategia habitual consiste en capturar las modificaciones realizadas sobre los datos y transformarla en eventos. Esto puede ser abordado mediante la técnica de *Change Data Capture* (CDC), que permite detectar y extraer de forma continua los cambios producidos en una base de datos [18]. De este modo, CDC permite convertir sistemas que no están planteados inicialmente para funcionar con eventos, poder ser integrado en arquitecturas de streaming sin necesidad de modificar la lógica ya existente.

2.3.1.2.- Mecanismos de ingestión de eventos

En el modelo *pull*, el sistema de streaming constantemente consulta a la fuente en intervalos de tiempo regulares para obtener nuevos datos [19]. Este enfoque, se implementa utilizando peticiones REST o consultas a bases de datos. Como parte negativa, la latencia se ve afectada por el intervalo de consultas, además que resulta ineficiente debido a que se pueden consultar cuando no hay datos disponibles.

Por otro lado, en el modelo *push*, el sistema fuente envía los eventos al sistema de streaming nada más ocurren, permitiendo una mayor latencia. Algunos ejemplos comunes son webhooks o los productores de eventos. Este enfoque es típico en los sistemas diseñados con una arquitectura orientada a eventos. La Tabla 2.1 muestra una comparativa entre ambos modelos.

Características	Modelo Push	Modelo Pull
Concepto	La fuente envía los datos al sistema cuando se generan	El sistema solicita los datos a la fuente periódicamente
Control de flujo	Reside en la fuente, pudiendo saturar al receptor	Reside en el receptor. El sistema pide solo lo que puede procesar
Latencia	Mínima	Variable, dependiendo de la frecuencia de muestreo
Estado	La fuente no necesita saber qué se ha enviado antes	El receptor debe llevar un control de qué datos ha extraído
Ejemplos	Websockets, Webhooks	REST Polling, CRON jobs

Tabla 2.1: Comparación entre el modelo *push* y *pull*

2.3.2.- Sistemas de mensajería

En las arquitecturas de procesamiento en streaming, los sistemas de mensajería actúan como la capa intermediaria entre las fuentes de eventos y los motores de procesamiento. Su función principal es permitir un bajo acoplamiento entre los productores de eventos y los sistemas que los consumen mediante una comunicación asíncrona, permitiendo que ambos operen de manera independiente y sin necesidad de estar activos simultáneamente ni sincronizados entre sí. De esta manera la responsabilidad de transportar los datos es del sistema de mensajería, y los productores y consumidores pueden centrarse en la generación y uso de eventos respectivamente, y así no tener que preocuparse de los detalles de la comunicación [20].

Los sistemas de mensajería no son exclusivos para el procesamiento en streaming, sino que son una pieza fundamental en las arquitecturas orientadas a eventos. Por esta razón, existe

una gran cantidad de tecnologías a utilizar las cuales se dividen en si están basadas en una cola o en un log [21]. A continuación, se analizarán ambos modelos y se mostrará cual es más adecuado para el procesamiento en streaming.

2.3.2.1.- Mensajería basada en colas vs. en logs

Los sistemas de mensajería basados en colas se basan en la estructura de datos de la cola. En el caso más simple, hay un productor que envía mensajes a la cola y un consumidor que los lee desde la cola. Cada vez que leemos un mensaje y se confirma la lectura, este será eliminado, por lo que solo un consumidor podrá leer el mensaje [22]. Si se requiere que múltiples consumidores lean el mismo mensaje sería necesario utilizar el modelo Pub/Sub, donde los productores envían los mensajes a un canal lógico común que copiara el mensaje a la cola interna de cada consumidor [23]. Esto se ve ilustrado en la Figura 2.11.

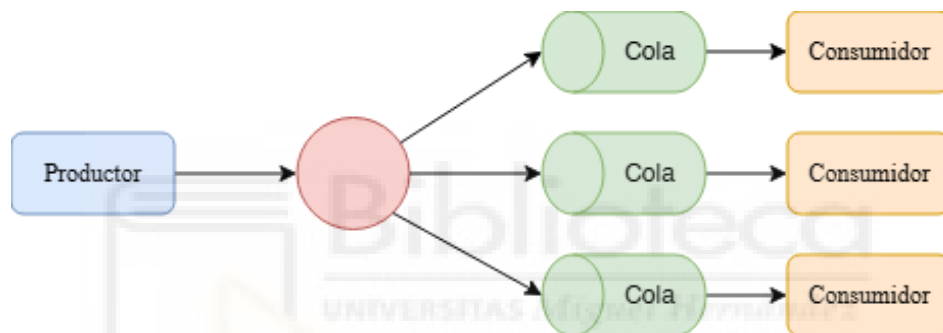


Figura 2.11: Mensajería basada en colas

Por otra parte, los sistemas de mensajería basados en logs, cuando un mensaje se escribe en el log es almacenado, permitiendo que múltiples consumidores lean de este [24]. Además, cada consumidor puede empezar a leer desde donde lo requiera (ver Figura 2.12).

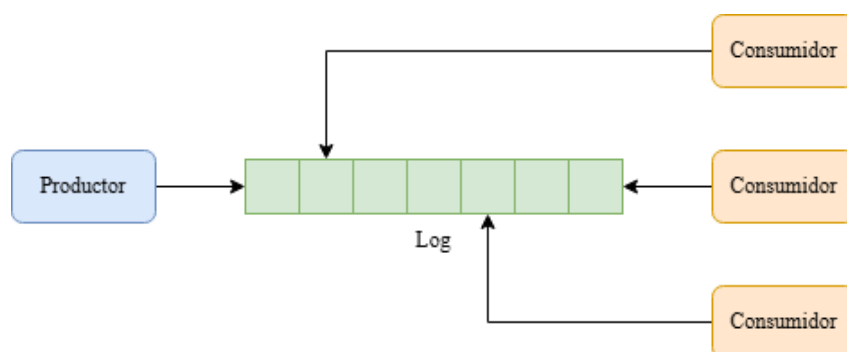


Figura 2.12: Mensajería basada en el Log

En el procesamiento en streaming, anteriormente se introdujeron los mecanismos de tolerancia a fallos, donde se indicaba que en caso de fallo, se recalcularían los eventos desde el momento anterior al fallo utilizando el estado correspondiente. Los sistemas de

mensajería basados en logs son ideales para esto, ya que los eventos son almacenados y permiten hacer lecturas desde un punto determinado del pasado. Además, en caso de aumentos de throughput la escalabilidad de los sistemas de mensajería basados en logs es más simple ya que consistiría en añadir particiones y escalar horizontalmente .

2.3.2.2.- Apache Kafka

Apache Kafka es un sistema de mensajería distribuido basado en logs, altamente escalable y tolerante a fallos, desarrollado por la compañía LinkedIn en 2010 [25]. Es una herramienta de código abierto pero también se ofrece como servicio mediante plataformas como Confluent. La arquitectura de kafka consiste en un clúster de múltiples nodos denominados brokers, que son las unidades de computación.

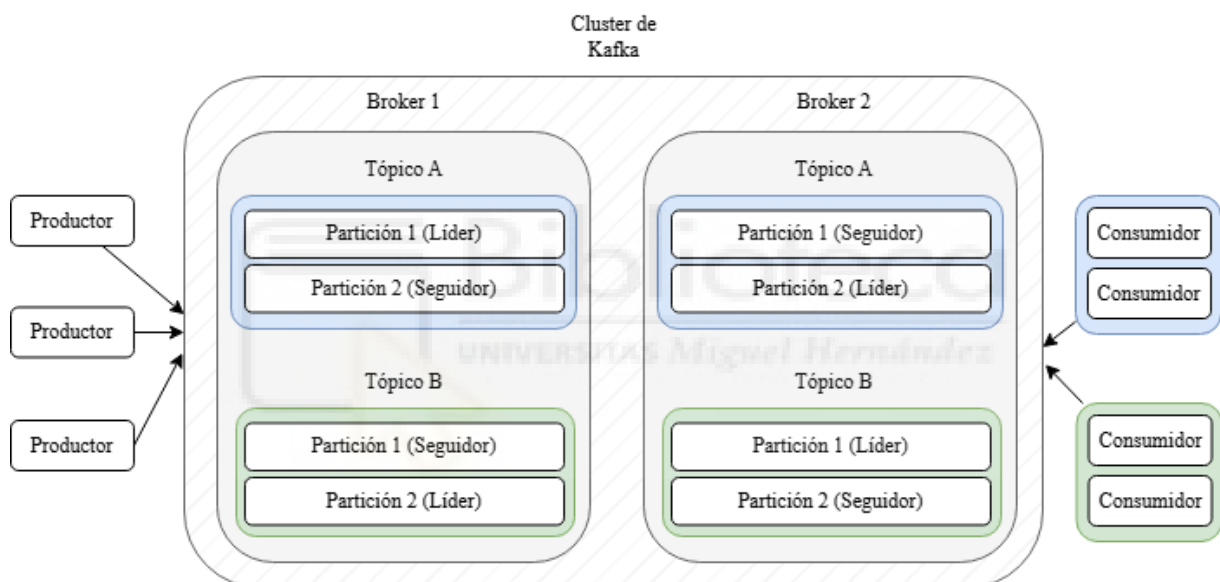


Figura 2.13: Arquitectura de Apache Kafka

La unidad fundamental de organización son los tópicos cuyo objetivo es permitir que haya una agrupación lógica de los mensajes [26]. Estos tópicos residen en particiones, la unidad fundamental de almacenamiento, pudiendo dividir el tópico en tantas particiones como se quiera para permitir la escalabilidad y paralelismo. Los eventos con una misma clave se almacenan en la misma partición para asegurar un orden global por tópico.

Cada evento dentro de una partición se identifica mediante un offset, que actúa como índice secuencial único dentro de dicha partición. Estos offsets permiten a los consumidores llevar un control de la posición en la cual están leyendo.

Para garantizar la tolerancia a fallos, Kafka replica las particiones a través de múltiples brokers. Cada partición tiene una réplica líder y varias réplicas seguidoras. El líder

gestiona todas las operaciones de lectura y escritura, mientras que los seguidores replican los datos. Este mecanismo de replicación permite que el sistema se mantenga disponible ante la caída de nodos individuales.

En caso de que los productores escriban mensajes a una velocidad las cuales los consumidores no pueden procesar a una velocidad suficiente, es necesario introducir consumidores adicionales. Esto se hace mediante los grupos de consumidores, donde cada grupo de consumidores se asocia a un tópic y cada consumidor leerá de una partición.

2.3.2.3.- Amazon Kinesis

Amazon Kinesis es el servicio gestionado de Amazon que implementa un log distribuido para la ingesta y almacenamiento de eventos en streaming [27]. Su modelo es conceptualmente similar al de Apache Kafka, ya que ambos sistemas se basan en el log distribuido. No obstante, la terminología de sus componentes difiere: los topics en Kafka corresponden a los streams en Kinesis, y las particiones a los shards.

En el caso de los shards, la diferencia también es funcional, ya que cada shard tiene asociado un límite de throughput tanto de escritura como de lectura, medido en MB/s o número de eventos por segundo [28].

Una diferencia fundamental es que, al ser un servicio gestionado, Kinesis abstrae completamente la gestión de clústeres y brokers. Esto implica un cambio en el modelo de escalabilidad respecto a Kafka. Mientras que en Kafka el escalado se realiza mediante la adición de brokers y el incremento del número de particiones, en Kinesis se basa en la modificación del número de shards. Por ejemplo, operaciones como el shard split permiten dividir el rango de claves de un shard, incrementando el throughput disponible y el paralelismo potencial del sistema.

2.3.3.- Motores de procesamiento de streams

A continuación se describen distintas herramientas encargadas de procesar estos eventos. A pesar de que el objetivo sea el mismo, internamente cada una tiene un enfoque de procesamiento e infraestructura distintas.

2.3.3.1.- Apache Flink

Apache Flink es un motor de procesamiento distribuido que funciona tanto para procesar en streaming como en batch. Su modelo de procesamiento es el streaming nativo, donde

todos los datos son tratados como flujos continuos, incluso cuando se trabaja con datasets acotados. Está implementado principalmente en Java y Scala, aunque también ofrece APIs para Python mediante PyFlink.

Flink proporciona diversas APIs que permiten distintos niveles de abstracción. Por un lado, la DataStream API permite el procesamiento de data streams no estructurados con control detallado sobre el flujo de ejecución, siendo adecuada para aplicaciones complejas de streaming. Por otro lado, la Table API y SQL ofrecen un nivel más declarativo, permitiendo expresar transformaciones mediante consultas similares a SQL sobre streams y datos en modo batch.

En cuanto a su arquitectura, Flink se ejecuta sobre un clúster distribuido que permite la ejecución paralela de tareas y puede desplegarse sobre gestores de recursos como Kubernetes o YARN

2.3.3.2.- Kafka Streams

Kafka Streams es una librería de código abierto que permite construir aplicaciones que procesan datos en streaming. Adopta un modelo de procesamiento de eventos nativo al igual que Flink. Está diseñada para integrarse de forma transparente con Apache Kafka, actuando tanto consumidor como productor. Kafka Streams está implementado en Java y está orientado principalmente a su uso en Java y Scala.

Proporciona dos APIs principales. Por un lado, Kafka Streams DSL, una API de alto nivel, que ofrece una interfaz declarativa que permite definir las operaciones sobre el data stream sin especificar los detalles de ejecución. Este DSL se basa en dos abstracciones fundamentales: KStream, que modela el data stream en eventos independientes, y KTable que representa el estado de los eventos con una misma clave, sobrescribiendo los valores previos. Por otro lado, la Processor API, de bajo nivel, que proporciona un control exhaustivo utilizada para casos en los que se requiere más control o se necesita lógica que no está soportada por el DSL.

Respecto a su arquitectura, Kafka Streams no se ejecuta como un clúster independiente, sino como una librería importada dentro de las propias aplicaciones. Cada instancia de la aplicación actúa como un nodo de procesamiento, y la escalabilidad se logra mediante la ejecución de múltiples instancias en paralelo. La coordinación, particionado de datos y balanceo de carga se delega en el propio sistema de Kafka, lo que simplifica considerablemente la gestión operativa al eliminar la necesidad de un gestor de recursos externo.

2.3.3.3.- Apache Spark

Apache Spark es un motor de procesamiento distribuido de código abierto utilizado para el procesamiento de datos a gran escala. Surgió como una respuesta a las limitaciones del modelo MapReduce de Apache Hadoop para el procesamiento por lotes, el cual depende de constantes lecturas y escrituras en almacenamiento secundario, Spark optimiza el rendimiento mediante la computación en memoria.

En cuanto su arquitectura, Spark se ejecuta sobre un clúster distribuido, donde un nodo maestro coordina la ejecución de tareas en múltiples nodos trabajadores. De esta manera, puede integrarse con diferentes gestores de recursos como YARN o Kubernetes.

Spark dispone de un ecosistema unificado de librerías que permiten ir más allá del procesamiento por lotes. Entre ellas se encuentran Spark SQL, para consultas estructuradas, MLlib para Machine Learning, GraphX para el procesamiento de grafos y por último permite el procesamiento el streaming mediante Spark Streaming o la actual versión Spark Structured Streaming.

Para permitir el procesamiento en streaming, Spark utiliza un modelo basado en micro lotes. En lugar de procesar cada evento de manera individual, Spark los agrupa durante todos los eventos que lleguen durante un intervalo de tiempo muy corto, y los procesa como si fuesen lotes. Este enfoque permite reutilizar el mismo motor utilizado para el procesamiento por lotes. Sin embargo, implica un aumento de latencia debido a que los eventos para ser procesados han de esperar a que se complete el intervalo de agregación.

Característica	Apache Flink	Kafka Streams	Apache Spark
Modelo de procesamiento	Streaming nativo	Streaming nativo	Microlotes
Latencia	Muy baja	Baja	Media-Baja
Throughput	Muy alto	Moderado	Muy alto
Despliegue	Cluster, YARN, Kubernetes	Librería de Java	YARN, Kubernetes, Databricks
Complejidad operativa	Alta	Baja	Media
Soporte a procesamiento en batch	Si	No	Si
Soporte a SQL	Flink SQL	KSQL	Spark SQL
Lenguajes principales	Java/Scala	Java/Scala	Scala/Java/Python/R

Tabla 2.2: Comparación entre los distintos motores de procesamiento en Streaming

2.3.4.- Destinos de los eventos procesados

Una vez los eventos han sido procesados por los motores de procesamiento, los resultados no se utilizan en un único sistema, sino que se distribuyen hacia varios destinos según el tipo de consulta, latencia requerida y el caso de uso [12]. Este enfoque permite que un mismo data stream alimenta tanto sistemas analíticos como aplicaciones operacionales, utilizando en cada caso la tecnología más adecuada.

Para el análisis de grandes volúmenes de datos, los eventos suelen almacenarse en bases de datos analíticas OLAP como ClickHouse o Amazon Redshift. Estos sistemas utilizan almacenamiento columnar, lo que facilita la ejecución eficiente de agregaciones y consultas complejas. Al mismo tiempo, en arquitecturas orientadas a eventos, los resultados procesados pueden enviarse a sistemas de mensajería, permitiendo que otros componentes reaccionen a los resultados.

En el contexto del análisis en tiempo real, existen bases de datos en streaming, como Materialize o RisingWave, que mantienen vistas materializadas actualizadas continuamente. Estas permiten obtener resultados inmediatos conforme llegan nuevos eventos. Por otro lado, cuando se requieren capacidades de búsqueda avanzada, se utilizan motores como Elasticsearch, que permiten realizar consultas de texto completo y detectar patrones de interés de forma eficiente.

En aplicaciones que requieren latencias muy bajas, como sistemas de recomendación o detección de fraude, los datos procesados se almacenan en sistemas clave-valor como Redis, que ofrecen tiempos de acceso extremadamente reducidos. Para almacenamiento a largo plazo, es habitual utilizar arquitecturas de tipo lakehouse basadas en formatos como Apache Iceberg, que combinan flexibilidad y consistencia. Finalmente, cuando los datos tienen una naturaleza temporal, como en el caso de sensores IoT, se emplean bases de datos de series temporales como InfluxDB, diseñadas para gestionar grandes volúmenes de datos ordenados en el tiempo de forma eficiente.

Capítulo 3

Hipótesis de trabajo

3.1.- ARQUITECTURA DE LOS MOTORES DE PROCESAMIENTO

La arquitectura de cualquier motor de procesamiento de datos en tiempo real (streaming) determina su rendimiento y comportamiento frente a los requisitos críticos de un sistema de este tipo: la latencia (tiempo que tarda un evento en ser procesado), la consistencia (garantía sobre la integridad y veracidad de los datos procesados) y la escalabilidad (capacidad de manejar volúmenes crecientes de datos de manera eficiente). Los frameworks Apache Flink, Apache Spark Streaming y Kafka Streams comparten la misión fundamental de gestionar y procesar flujos de datos continuos (data streams), pero cada uno de ellos implementa diferentes diseños arquitectónicos para abordar dichos requisitos. Con el fin de establecer una base técnica para la comparativa del siguiente capítulo, esta sección examina en detalle la arquitectura interna de cada uno de estos motores.

3.1.1.- Apache Flink

Apache Flink se ejecuta sobre un cluster distribuido cuya arquitectura se organiza en torno a dos componentes principales: el *JobManager* y los *TaskManagers*. El *JobManager* actúa como nodo maestro y asume las responsabilidades de coordinación de la ejecución distribuida, incluyendo la planificación de tareas, la coordinación de la recuperación ante fallos y la gestión de los checkpoints. Internamente, el *JobManager* se compone de tres elementos [29]:

- El *ResourceManager* es responsable de la asignación y liberación de recursos en el clúster, gestionando los tasks slots, la unidad básica de planificación
- El Dispatcher expone una interfaz REST para la enviar jobs, donde cada vez que se envía uno, se inicia un JobMaster
- El *JobMaster* gestiona la ejecución de un *JobGraph*, el grafo acíclico dirigido que representa el job. Pueden coexistir múltiples *JobMasters* simultáneamente en el cluster, uno por cada trabajo.

Los *TaskManagers*, por su parte, son los nodos trabajadores, son los responsables de ejecutar las tareas, los vértices de la representación física del grafo. Cada *TaskManager* es un proceso JVM que puede ejecutar una o más subtareas en hilos separados, donde cada subtarea es una instancia en paralelo de la tarea encargada de procesar una partición del data stream. La gestión del número de tareas de cada *TaskManager* se realiza mediante los slots de tareas, los cuales representan una división fija de los recursos del *TaskManager*.

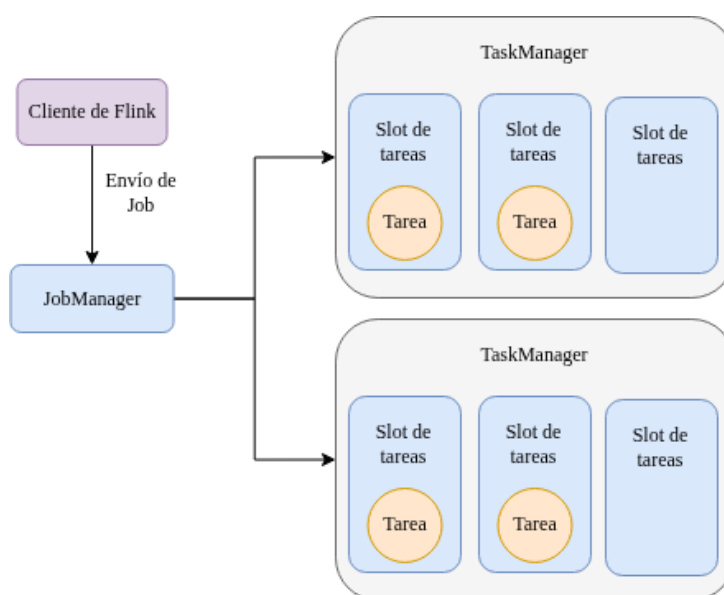


Figura 3.1: Arquitectura interna de Apache Flink

Flink ofrece dos backends de estado principales. El *HashMapStateBackend* almacena el estado como objetos Java en el heap de la JVM, representando los estados clave-valor y los operadores de ventana mediante tablas hash en memoria. El *EmbeddedRocksDBStateBackend*, por su parte, persiste el estado en una base de datos RocksDB local, serializando los datos como arrays de bytes. La elección entre ambos implica un compromiso entre rendimiento y escalabilidad: *HashMapStateBackend* ofrece un acceso al estado más rápido al operar directamente sobre el heap de la JVM, pero queda limitado por la memoria disponible en el clúster; *EmbeddedRocksDBStateBackend* permite escalar en función del espacio en disco, aunque cada acceso al estado requiere serialización y lectura a disco, lo que puede suponer un rendimiento inferior [30].

3.1.2.- Apache Spark Streaming

Apache Spark Streaming se construye sobre el motor de procesamiento distribuido de Spark, cuya arquitectura sigue un modelo maestro-esclavo. El nodo maestro, denominado Driver, es el responsable de coordinar la ejecución de la aplicación. Internamente, el Driver alberga el *SparkContext*, punto de entrada de la aplicación que establece la conexión con el gestor de recursos del clúster, y el *DAGScheduler*, encargado de traducir el plan lógico de transformaciones en un grafo acíclico dirigido de etapas y tareas que se distribuyen entre los nodos trabajadores. Entre el Driver y los nodos trabajadores actúa el cluster manager, el componente responsable de la gestión y asignación de recursos del clúster, para el cual Spark soporta distintas implementaciones como YARN, Kubernetes o su propio gestor standalone. Los nodos trabajadores (workers) son las máquinas físicas del clúster sobre las que se despliegan los Executors, procesos JVM que ejecutan las tareas asignadas por el Driver y almacenan en memoria o disco los datos que necesiten persistirse. Un nodo trabajador puede alojar múltiples *Executors* de forma simultánea, dependiendo de los recursos disponibles y de la configuración del clúster [31].

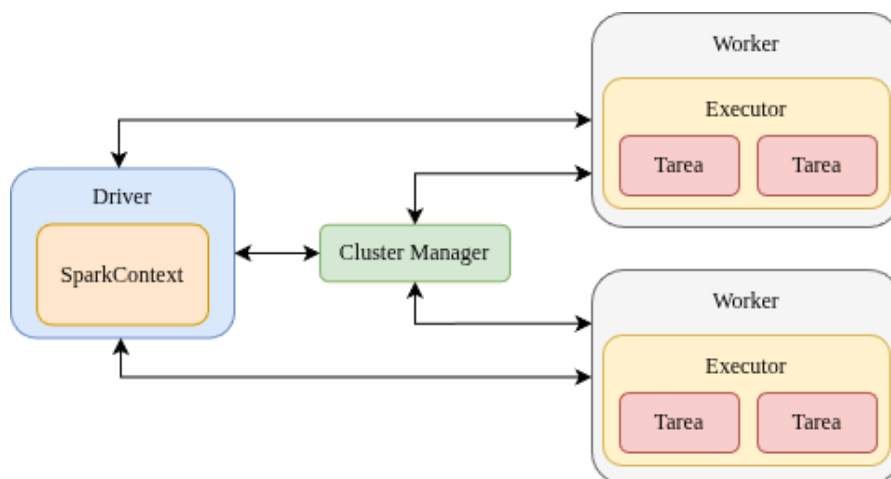


Figura 3.2: Arquitectura interna de Apache Spark

Para el procesamiento en streaming, Spark emplea un modelo de micro-batch. En la interfaz de bajo nivel, basada en *DStreams* (Discretized Streams) [32], el flujo de datos entrante se divide en pequeños lotes de registros representados como RDDs (*Resilient Distributed Datasets*), la estructura de datos fundamental de Spark. Un RDD es una colección distribuida e inmutable de elementos particionada a lo largo de los nodos del clúster, sobre la que se aplican transformaciones de forma paralela [33]. Cada micro-batch genera un nuevo RDD que es procesado de forma independiente aplicando la cadena de transformaciones definida por el desarrollador.

La interfaz de alto nivel, Structured Streaming, supone una evolución sobre *DStreams* que abstrae el modelo de micro-batch mediante el concepto de tabla sin límite (unbounded table). El flujo de datos entrante se modela como una tabla a la que se añaden filas de forma continua, y las transformaciones se expresan mediante el API de Spark SQL o DataFrames [34]. Estas transformaciones son optimizadas por el Catalyst, el optimizador de consultas de Spark, que genera un plan de ejecución físico eficiente aplicando reglas de optimización lógica y física sobre el plan de consulta. El resultado se escribe en el sumidero configurado según el modo de salida definido: append, que escribe únicamente las filas nuevas; update, que escribe las filas modificadas; o complete, que reescribe la tabla de resultados completa en cada micro-batch.

3.1.3.- Kafka Streams

Kafka Streams es una abstracción de alto nivel construida sobre las APIs nativas de productor y consumidor de Kafka, encapsulando su complejidad y proporcionando un modelo de programación orientado al procesamiento de streams [35]. Toda la lógica de procesamiento se ejecuta dentro de la propia aplicación cliente, sin requerir un clúster de procesamiento dedicado.

El paralelismo en Kafka Streams se articula en torno al concepto de tarea. Kafka Streams crea una tarea por cada partición de los topics de entrada, de modo que el grado de paralelismo máximo queda determinado por el número de particiones configuradas [36]. Cada tarea es independiente y procesa exclusivamente los registros de su partición asignada. La ejecución de estas tareas se distribuye entre uno o varios stream threads, configurables por instancia de la aplicación, donde cada hilo puede gestionar múltiples tareas de forma concurrente. Para escalar horizontalmente, basta con ejecutar múltiples instancias de la aplicación, y Kafka Streams redistribuye automáticamente las tareas entre los hilos e instancias disponibles mediante el protocolo de coordinación de grupos de consumidores de Kafka.

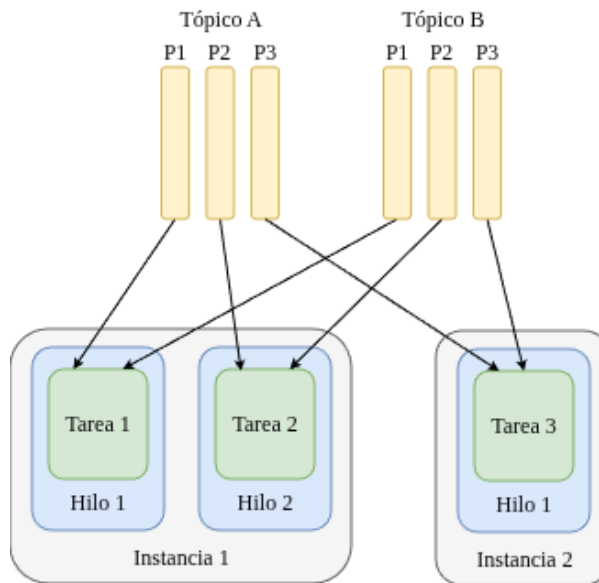


Figura 3.3: Arquitectura interna de Kafka Streams

Para las operaciones con estado, como agregaciones, joins o ventanas temporales, Kafka Streams utiliza almacenes de estado locales, implementados por defecto mediante RocksDB [38]. Cada tarea gestiona su propio almacén de estado de forma independiente, lo que evita la necesidad de coordinación entre tareas y contribuye al rendimiento del sistema. Para garantizar la tolerancia a fallos, cada almacén de estado se respalda automáticamente mediante un changelog topic en Kafka: cada modificación del estado local se publica como un registro en dicho topic, de modo que ante un fallo, la tarea puede reconstruir su estado reproduciendo los registros del changelog topic desde el inicio, garantizando así la recuperación completa sin pérdida de información.

3.2.- INFRAESTRUCTURA DE MENSAJERÍA Y TRANSPORTE DE DATOS

3.2.1.- Apache Kafka

Apache Kafka actúa como la infraestructura central de transporte de datos, desempeñando el rol de intermediario entre el generador de eventos y los tres motores de procesamiento evaluados. Los eventos generados son publicados en topics de Kafka desde los que cada motor los consume de forma independiente.

Su capacidad de reposicionamiento en el stream es un requisito indispensable para que los motores puedan implementar la semántica de entrega *exactly-once* [39], al permitir reproducir los registros desde el offset registrado en el último checkpoint ante un fallo.

Asimismo, su modelo de replicación de particiones garantiza la disponibilidad de los datos ante la caída de nodos individuales, evitando que la infraestructura de mensajería se convierta en un punto de fallo durante la ejecución de los experimentos.

3.2.2.- Protocol Buffers

Protocol Buffers, también conocido como Protobuf, es un mecanismo de serialización de datos binario desarrollado por Google [40]. A diferencia de formatos basados en texto como JSON o XML, Protobuf serializa los datos en un formato binario compacto a partir de un esquema definido en un fichero .proto, lo que resulta en mensajes de menor tamaño y un proceso de serialización y deserialización significativamente más eficiente. El esquema define de forma explícita la estructura de los mensajes, los tipos de cada campo y sus identificadores numéricos, a partir del cual se genera automáticamente el código de serialización para el lenguaje de destino [41].

Protobuf se utiliza ampliamente en sistemas distribuidos y pipelines de datos de alto rendimiento donde el tamaño de los mensajes y la latencia de serialización son factores críticos. Su tipado estricto y la definición explícita del esquema proporcionan además una mayor robustez frente a cambios en la estructura de los mensajes.

3.3.- LENGUAJES DE PROGRAMACIÓN

3.3.1.- Go

Go es un lenguaje de programación compilado y de tipado estático desarrollado por Google en 2009 [42]. Está diseñado para la construcción de sistemas distribuidos y servicios de red de alto rendimiento, siendo ampliamente utilizado en el desarrollo de microservicios, herramientas de línea de comandos, APIs REST y componentes de infraestructura [43]. Entre sus características más destacadas se encuentran su modelo de concurrencia basado en goroutines, su compilación a binarios nativos sin dependencias externas y su reducido consumo de memoria en tiempo de ejecución.

En este trabajo, Go se ha utilizado para el desarrollo del generador de eventos debido a su ligereza y su capacidad para gestionar conexiones concurrentes con un overhead mínimo. Su rápido ciclo de desarrollo, junto con la simplicidad de su modelo de concurrencia, lo hace una opción adecuada para la construcción de un productor de eventos que requiere alta frecuencia de emisión y baja latencia.

3.3.2.- Python

Python es un lenguaje de programación interpretado y de tipado dinámico ampliamente utilizado en el ámbito del análisis de datos, la visualización y el machine learning [44]. Su extenso ecosistema de librerías lo convierten en una de las opciones más populares para el tratamiento y la representación gráfica de datos [45].

Para este trabajo, Python se emplea para la visualización de los resultados y métricas producidas por los motores de procesamiento.

3.3.3.- Java

Java es un lenguaje de programación compilado a bytecode y ejecutado sobre la máquina virtual de Java (JVM), ampliamente adoptado en el desarrollo de aplicaciones empresariales de gran escala [46]. Su modelo de tipado estático, su robustez y su maduro ecosistema de herramientas se ha consolidado como uno de los lenguajes de referencia en el ámbito.

El uso que se le da en este trabajo es para el desarrollo de los jobs para los motores de procesamiento. Su elección viene motivada por el hecho de que los tres frameworks están implementados de forma nativa sobre la JVM y ofrecen sus APIs principales en Java, lo que garantiza compatibilidad completa y acceso a la totalidad de las funcionalidades de cada motor sin las limitaciones que pueden presentarse en otras APIs como PyFlink o PySpark.

3.4.- HERRAMIENTAS DE MONITORIZACIÓN

3.4.1.- Prometheus

Prometheus es un sistema de monitorización y alerta de código abierto desarrollado originalmente por SoundCloud y actualmente mantenido por la Cloud Native Computing Foundation (CNCF) [47]. Su modelo de recolección de métricas se basa en el scraping periódico de endpoints HTTP expuestos por los componentes monitorizados, almacenando los datos en una base de datos de series temporales optimizada para consultas sobre métricas numéricas [48].

El modelo de datos de Prometheus se organiza en torno a métricas identificadas por un nombre y un conjunto de etiquetas clave-valor, lo que permite un alto grado de dimensionalidad y flexibilidad en la definición y consulta de métricas. Las consultas se realizan mediante PromQL, un lenguaje de consulta propio que permite agregar, filtrar y operar sobre series temporales de forma expresiva [49].

Prometheus se va a utilizar como el componente central de recolección de métricas, recogiendo datos de rendimiento de los tres motores de procesamiento evaluados, así como de la infraestructura subyacente.

3.4.2.- JMX

JMX (Java Management Extensions) es una tecnología estándar de la plataforma Java que proporciona una arquitectura para la monitorización y gestión de aplicaciones y servicios en tiempo de ejecución [50]. Mediante la exposición de beans gestionados (MBeans), las aplicaciones Java pueden publicar métricas internas relativas a parámetros como el uso de memoria, la actividad del recolector de basura, el estado de los hilos y otros indicadores específicos de la aplicación.

Dado que Apache Flink, Apache Spark y Kafka Streams se ejecutan sobre la JVM, todos ellos exponen métricas internas a través de JMX. Sin embargo, JMX por sí solo no es directamente compatible con Prometheus, por lo que su integración requiere el uso de un exportador intermedio que traduzca las métricas JMX al formato de exposición de Prometheus [51].

3.4.3.- cAdvisor

cAdvisor (Container Advisor) es una herramienta de monitorización de código abierto desarrollada por Google, diseñada específicamente para la recolección de métricas de rendimiento de contenedores [52]. Analiza y expone datos relativos al uso de CPU, memoria, red y sistema de ficheros de cada contenedor en ejecución, sin requerir modificaciones en las aplicaciones monitorizadas.

cAdvisor se integra de forma nativa con entornos Docker y Kubernetes, siendo capaz de descubrir automáticamente los contenedores en ejecución y recopilar sus métricas sin necesidad de configuración adicional por contenedor. Las métricas recolectadas son expuestas en formato compatible con Prometheus, lo que facilita su integración en la pila de monitorización.

3.4.4.- Grafana

Grafana es una plataforma de visualización y análisis de métricas de código abierto. Su arquitectura se basa en un modelo de fuentes intercambiables, permitiendo conectarse a múltiples backends de almacenamiento de métricas, logs y trazas mediante plugins [53], sin imponer restricciones sobre el sistema de almacenamiento subyacente.

El modelo de trabajo de Grafana se organiza en torno a dashboards compuestos por paneles configurables, cada uno de los cuales ejecuta consultas sobre una fuente de datos y representa los resultados mediante distintos tipos de visualización, como gráficas de series temporales, tablas, mapas de calor o indicadores numéricos [54]. Las consultas se definen en el lenguaje nativo de cada fuente de datos conectada, de modo que cuando se integra con Prometheus, las consultas se expresan directamente en PromQL.

Grafana se va a utilizar como el componente central de visualización del sistema de monitorización, consumiendo las métricas recolectadas por Prometheus y permitiendo inspeccionar de forma unificada el rendimiento de los tres motores de procesamiento evaluados, así como el estado de la infraestructura.

3.4.5.- Redpanda Console

Redpanda Console es una interfaz web de código abierto para la administración y monitorización de clústeres Apache Kafka. Proporciona una vista unificada de los topics, particiones, grupos de consumidores y mensajes del clúster [55], permitiendo inspeccionar visualmente el estado del sistema de mensajería en tiempo real.

Entre sus funcionalidades más relevantes se encuentran la visualización del lag de los grupos de consumidores, la exploración interactiva de los mensajes almacenados en los topics y la monitorización del estado de los brokers y las particiones. Estas capacidades resultan especialmente útiles durante el desarrollo y la depuración de pipelines de procesamiento en streaming.

3.5.- LIBRERÍAS

3.5.1.- franz-go

franz-go es una librería cliente de Apache Kafka para Go, de código abierto, que proporciona una implementación completa del protocolo Kafka [56]. A diferencia de otras

librerías cliente para Go, `franz-go` implementa el protocolo de Kafka de forma nativa, sin depender de la librería C `librdkafka`, lo que simplifica la compilación y el despliegue al generar binarios autocontenidos.

3.5.2.- Seaborn

Seaborn es una librería de visualización estadística para Python construida sobre Matplotlib [57]. Proporciona una API de alto nivel que simplifica la generación de gráficos estadísticos, como diagramas de caja, mapas de calor o gráficas de distribución, con un estilo visual más cuidado que el ofrecido por Matplotlib de forma predeterminada.

3.5.3.- Pandas

Pandas es la librería de referencia para el análisis y la manipulación de datos tabulares en Python. Proporciona la estructura de datos *DataFrame*, que permite representar, transformar y analizar conjuntos de datos de forma eficiente mediante una API expresiva y bien documentada [58], con soporte para la lectura y escritura de múltiples formatos de fichero como CSV, JSON o Parquet.

Para este trabajo, Pandas se utiliza para la carga, limpieza y transformación de las métricas exportadas por Prometheus antes de su visualización.

3.6.- HERRAMIENTAS DE DESPLIEGUE

3.6.1.- Kubernetes

Kubernetes es una plataforma de orquestación de contenedores de código abierto que automatiza los procesos de despliegue, gestión y escalado de aplicaciones en contenedores [59]. Está basado en Borg, la plataforma interna de orquestación de Google, posteriormente Kubernetes se presentó al público y en 2015 fue donado a la Cloud Native Computing Foundation (CNCF). Actualmente se encuentra entre los treinta proyectos más importantes de código abierto [60].

Proporciona un sistema declarativo para la gestión del ciclo de vida de las aplicaciones en contenedores, abstrayendo la infraestructura mediante una capa de automatización que gestiona la planificación de cargas de trabajo, asignación y seguimiento de recursos, la tolerancia a fallos y el descubrimiento de servicios. La unidad mínima de despliegue en

Kubernetes es el Pod, que encapsula uno o más contenedores que comparten almacenamiento y red [61]. Sobre esta abstracción se construyen recursos de nivel superior como por ejemplo los Deployments, que gestionan el ciclo de vida de conjuntos de Pods, garantizando el número deseado de réplicas en ejecución.

Un clúster de Kubernetes se compone de un conjunto de nodos, que pueden ser máquinas físicas o virtuales, agrupados en dos planos: el plano de control, responsable de gestionar el clúster, que expone una API REST como punto de entrada a través de la cual se gestionan todos los recursos, y el plano de trabajo, formado por los nodos trabajadores donde se despliegan y ejecutan los Pods. La figura 3.4 ilustra los componentes internos de un clúster de Kubernetes.

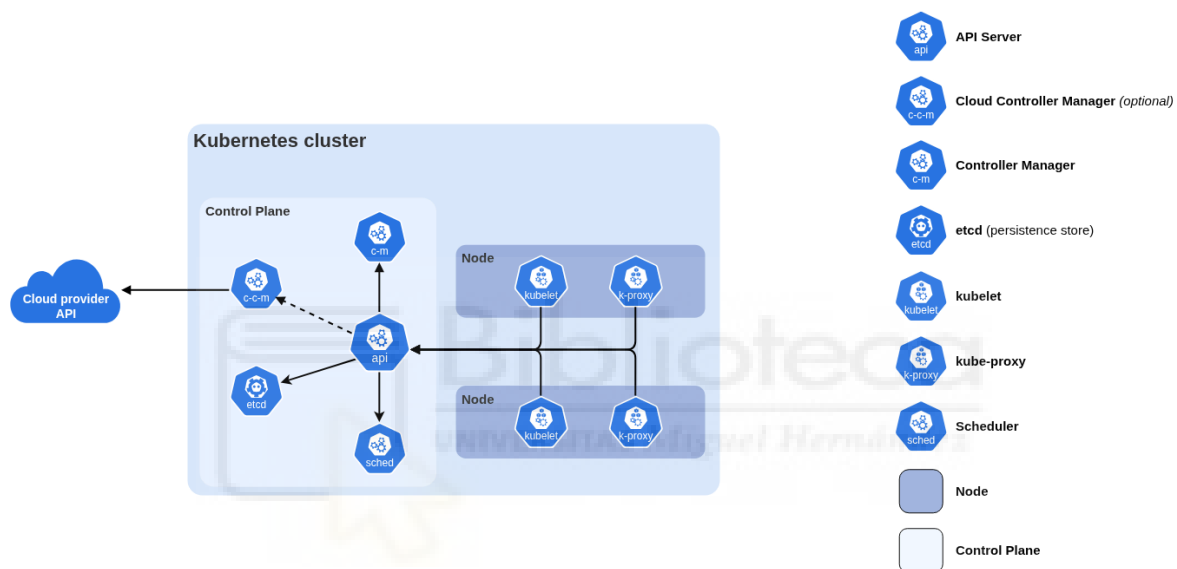


Figura 3.4: Arquitectura de clúster de Kubernetes

Kubernetes se utiliza en el presente trabajo para orquestar los diferentes contenedores, incluyendo los motores de procesamiento, el clúster de kafka, las aplicaciones de monitorización y servicios encargados de enviar y consumir los eventos.

3.6.2.- Google Cloud Platform

Google Cloud Platform (GCP) es el conjunto de servicios de computación en la nube ofrecido por Google. Proporciona infraestructura y servicios gestionados bajo demanda que abarcan áreas como cómputo, almacenamiento, bases de datos o redes. La Figura 3.5 muestra algunos de los servicios ofrecidos. De esta manera se permite desplegar y operar aplicaciones sin gestionar la infraestructura física.

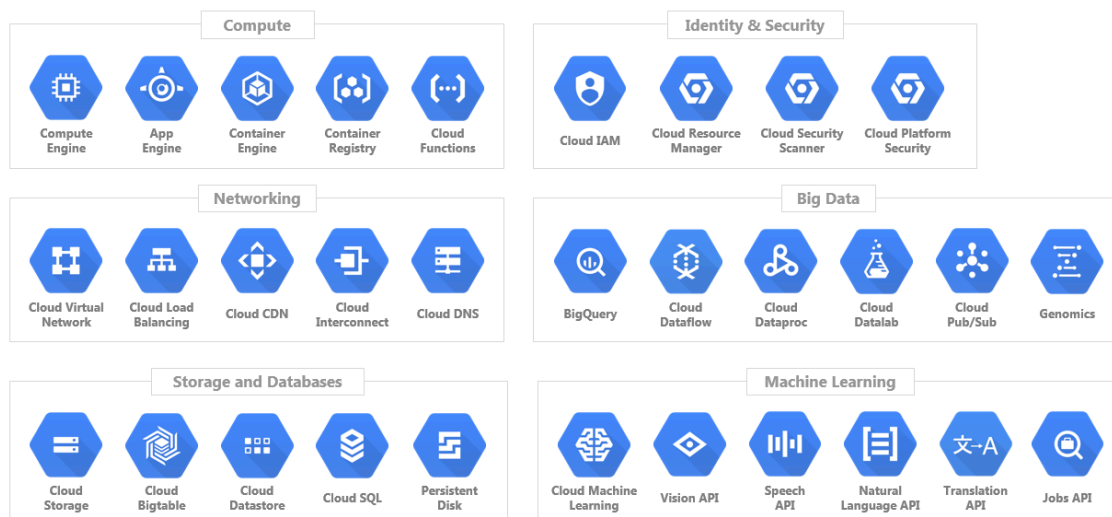


Figura 3.5: Servicios ofrecidos por GCP

Su modelo de facturación se basa en el consumo real de recursos, cobrando únicamente por lo que se utiliza sin requerir compromisos de capacidad previos. Los precios varían en función de la región geográfica donde se despliegan los recursos, dado que los costes operativos de los centros de datos difieren según su ubicación.

Entre sus servicios se encuentra Google Kubernetes Engine (GKE), un servicio gestionado de Kubernetes, el cual delega en GCP la gestión del plano de control. Esto permite que el usuario se centre en las cargas de trabajo desplegadas [62]. GKE será la infraestructura donde se ejecutará el clúster de Kubernetes del trabajo, proporcionando un escenario similar al que se usaría en un proyecto en un entorno de producción.

3.7.- HERRAMIENTAS DE DESARROLLO

3.7.1.- Docker

Docker es una plataforma de código abierto para la creación, distribución y ejecución de aplicaciones en contenedores [63]. Un contenedor es una unidad de software que empaqueta el código de una aplicación junto con todas sus dependencias, librerías y configuración, garantizando que la aplicación se ejecute de forma idéntica independientemente del entorno donde se creó. A diferencia de las máquinas virtuales, los contenedores comparten el kernel del sistema operativo anfitrión, lo que los hace significativamente más ligeros y eficientes en términos de recursos [64].

Docker se ha consolidado como el estándar de facto [65] para el despliegue de aplicaciones en entornos de desarrollo, pruebas y producción, siendo ampliamente adoptado en arquitecturas de microservicios y pipelines de integración y entrega continua. Su ecosistema incluye Docker Hub, un registro público de imágenes que permite reutilizar y distribuir imágenes preconfiguradas de una gran variedad de servicios y herramientas.

Docker se utilizará para contenerizar todos los componentes de la arquitectura experimental, incluyendo los motores de procesamiento, el clúster Kafka y las herramientas de monitorización. Este enfoque garantiza la reproducibilidad del entorno experimental y elimina las dependencias con la configuración específica del sistema.

3.7.2.- Kind

Kind (Kubernetes in Docker) es una herramienta que permite ejecutar clústeres de Kubernetes locales utilizando contenedores de Docker como nodos [66]. Fue desarrollada originalmente para la ejecución de pruebas de integración del propio Kubernetes [67], aunque su simplicidad y bajo coste de aprovisionamiento la han convertido en una opción ampliamente adoptada para el desarrollo y la validación local de cargas de trabajo destinadas a entornos Kubernetes.

Kind se utilizará como entorno de desarrollo y validación local previo al despliegue en GCP, permitiendo iterar sobre los distintos componentes y la configuración de Kubernetes, verificando el comportamiento sin incurrir en costes de infraestructura en la nube.

3.7.3.- Maven

Maven es una herramienta de gestión y construcción de proyectos para el ecosistema Java, mantenida por la Apache Software Foundation. Se basa en el concepto de gestión declarativa de proyectos: mediante un fichero de configuración en formato XML denominado pom.xml (Project Object Model), el desarrollador describe las dependencias del proyecto, los plugins necesarios y las fases del ciclo de construcción, delegando en Maven la resolución automática de dependencias y la ejecución de las tareas de compilación, empaquetado y pruebas [68].

Maven organiza el ciclo de vida de construcción en fases bien definidas, como la compilación, la ejecución de pruebas unitarias, el empaquetado en un fichero JAR o WAR y la instalación en el repositorio local [69]. Su repositorio central, Maven Central, alberga la práctica totalidad de las librerías del ecosistema Java, facilitando la gestión de dependencias transitivas de forma automática.

Maven se utiliza como herramienta de construcción de los proyectos Java que implementan los jobs. Su uso garantiza una gestión de las dependencias de cada framework y simplifica el empaquetado de los jobs en ficheros JAR autocontenidos listos para su despliegue en los respectivos motores de procesamiento.

3.8.- OTRAS HERRAMIENTAS

3.8.1.- Visual Studio Code

Visual Studio Code es un editor de código fuente de código abierto desarrollado por Microsoft [70], ampliamente adoptado en el desarrollo de software por su equilibrio entre simplicidad y funcionalidad. Dispone de un extenso ecosistema de extensiones que permiten añadir soporte para lenguajes de programación, herramientas de depuración, integración con sistemas de control de versiones y asistentes de código, entre otras funcionalidades.

3.8.2.- Claude Code

Claude Code es una herramienta de desarrollo asistido por inteligencia artificial desarrollada por Anthropic, diseñada para integrarse en el flujo de trabajo del desarrollador mediante una interfaz de línea de comandos [71]. A diferencia de los asistentes de código tradicionales, Claude Code opera de forma agéntica, siendo capaz de leer y modificar ficheros del proyecto, ejecutar comandos y razonar sobre la estructura completa de una base de código para llevar a cabo tareas de desarrollo de mayor complejidad.

Capítulo 4

Metodología y resultados

4.1.- TAREAS Y PLANIFICACIÓN

El desarrollo del proyecto comprende múltiples tareas, abarcando la investigación, el diseño del sistema, la implementación y el análisis de los resultados. La planificación temporal de estas tareas se muestra en la Figura 4.1. En primer lugar, se llevará a cabo una fase de investigación con el objetivo de determinar qué métricas resultan más adecuadas para analizar los motores, así como los distintos parámetros que permiten garantizar una comparación justa entre ellos. Posteriormente, se procederá al diseño de las pruebas que se van a realizar, definiendo qué datos utilizar, la metodología de medición y los jobs que se utilizarán.

Una vez completa esta fase, se desarrollarán dichos jobs junto con la infraestructura local necesaria para su ejecución y luego se desplegará la infraestructura correspondiente en Google Cloud Platform donde se llevarán a cabo todas las pruebas y se recogerán los datos.

TAREAS	SEMANAS													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Investigación de los motores de procesamiento	■	■	■											
Investigación de métricas	■	■	■											
Investigación de infraestructura	■	■	■											
Diseño de las pruebas				■										
Diseño del sistema					■									
Creación del clúster local con Kafka y herramientas de observabilidad						■	■							
Implementación del productor						■	■							
Implementación de los jobs en los tres motores								■	■					
Implementación del consumidor										■				
Implementación del recolector de métricas y de dashboards en Grafana										■				
Configuración de los motores de procesamiento mediante operadores de Kubernetes											■	■		
Configuración de los manifiestos para GCP											■	■		
Ejecución de las pruebas													■	
Análisis de resultados														■

Figura 4.1: Diagrama de Gantt de la planificación del proyecto

Por último, se realizará un análisis de los resultados obtenidos, con el objetivo de extraer conclusiones sobre el rendimiento, eficacia, comportamiento, así como identificar ventajas, limitaciones y casos de uso más adecuados para cada uno de ellos.

4.2.- DISEÑO DE LAS PRUEBAS

4.2.1.- Datos de entrada

El flujo de entrada simula la actividad de usuarios interactuando en tiempo real con una plataforma web. Todos los eventos se estructuran utilizando Protobuf.

El evento de entrada es común para los distintos jobs, denominado InputEvent. Este evento contiene campos que permiten realizar análisis del comportamiento. La Figura 4.2 muestra la definición del evento en formato Protobuf.

```

message InputEvent {
    string event_id = 1;
    string user_id = 2;
    int64 event_time_ns = 3;
    uint64 sequence_number = 4;
    string session_id = 5;
    string page_id = 6;
    string device_type = 7;
    string country = 8;
    string region = 9;
    UserAction action = 10;
    double value = 11;
    int32 quantity = 12;
    repeated string tags = 13;
    bytes payload = 14;
}

enum UserAction {
    USER_ACTION_UNKNOWN = 0;
    USER_ACTION_VIEW = 1;
    USER_ACTION_CLICK = 2;
    USER_ACTION_ADD_TO_CART = 3;
    USER_ACTION_PURCHASE = 4;
    USER_ACTION_UPDATE = 5;
}

```

Figura 4.2: Definición del evento InputEvent

Estos datos son generados mediante un productor hecho en Go. Estos datos son generados de la siguiente manera con los siguientes posibles valores:

- *user_id*: Se selecciona aleatoriamente un número entero dentro de un rango uniforme definido, donde el formato es user-<ID>. La distribución uniforme asegura que dentro de Kafka todas las particiones reciban el mismo número de eventos.
- *event_id*: Se genera una cadena hexadecimal aleatoria de 16 caracteres mediante el cálculo del hash de un entero aleatorio de 64 bits.
- *event_time_ns*: Representa la marca temporal del sistema con resolución de nanosegundos utilizando la Unix epoch. Se utilizará para realizar las agregaciones basadas en ventanas temporales y también para medir la latencia.
- *sequence_number*: Es un contador que crece de manera monótona, su uso es para asegurar que no se han perdido eventos.

- *session_id*: Una cadena de 16 caracteres hexadecimales pseudoaleatorios.
- *page_id*: Un identificador de página web con formato “page-[0-999]” generado de manera uniforme.
- *device_type*: Un elemento aleatorio de la lista estática “mobile”, “desktop”, “tablet”
- *country*: Un país aleatorio extraído uniformemente de la lista “US”, “DE”, “FR”, “ES”, “GB”.
- *region*: Un identificador regional de tipo “region-[0-49]”.
- *action*: Enumeración de la acción del usuario, seleccionado aleatoriamente un valor número entre 1 y 5.
- *value*: Un número real en un rango de [0.0, 500.0).
- *quantity*: Un entero uniforme que oscila entre 1 y 10.
- *tags*: Un array estático compuesto por las etiquetas “tag-a”, “tag-b” y “tag-c”.
- *payload*: Es un array de bytes que se dimensiona de manera dinámica para rellenar el evento con el objetivo de poder asignar el tamaño definido sin alterar los campos.

4.2.2.- Job de filtro sin estado

El primer job está diseñado para medir la capacidad de rendimiento bruto de los motores en operaciones simples de filtrado y transformación. El procesamiento carece de estado, por lo que la carga se concentra en la entrada/salida de red de Kafka, la deserialización del evento de entrada, la ejecución de la condición y la serialización del evento resultante.

El job consiste en consumir eventos del tópico de entrada y descartar aquellos cuya acción sea *USER_ACTION_VIEW*. Los eventos que superan el filtro son transformados a *FilteredEvent*, cuya definición se muestra en la Figura 4.3, propagando los campos del evento original junto con el campo *event_time_ns* que corresponde al del evento original. Adicionalmente, se calcula un checksum y se envía al tópico de salida.

```

message FilteredEvent {
    string event_id = 1;
    string user_id = 2;
    int64 event_time_ns = 3;
    uint64 sequence_number = 4;
    string action = 5;
    string country = 6;
    string device_type = 7;
    double value = 8;
    int32 quantity = 9;
    uint64 checksum = 10;
}

```

Figura 4.3: Definición del evento FilteredEvent

4.2.3.- Job de agregado por clave

El segundo job introduce el procesamiento con estado, midiendo el rendimiento de los motores cuando cada evento debe actualizar un agregado acumulativo por clave. A diferencia del job anterior, la carga se extiende más allá de la entrada/salida de red, abarcando el acceso y persistencia del estado por clave y la serialización del agregado.

En este job cada evento recibido, actualiza el agregado acumulativo asociado a esa clave. El resultado emitido al tópico de salida es el de *KeyedAggregate*, cuya definición se muestra en la Figura 4.4, que representa el estado acumulado de todos los eventos procesados para esa clave desde el inicio. Dicho evento mantiene el número total de eventos y de compras, la suma mínimo y máximo del campo value, la suma del campo quantity, y el *event_time_ns* del evento que ha provocado la actualización del agregado. Tras la actualización, este evento se envía al tópico de salida.

```

message KeyedAggregate {
    string user_id = 1;
    uint64 event_count = 2;
    uint64 purchase_count = 3;
    double value_sum = 4;
    double value_min = 5;
    double value_max = 6;
    int64 quantity_sum = 7;
    int64 event_time_ns = 8;
    uint64 checksum = 9;
}

```

Figura 4.4: Definición del evento KeyedAggregate

4.2.4.- Job de agregado por ventana

El tercer job extiende el anterior introduciendo la dimensión temporal, agrupando los eventos en ventanas de tiempo fijas, y emitiendo un resultado por ventana al cierre de la misma. La carga abarca la gestión del evento, la asignación de eventos a ventanas, el mantenimiento del estado acumulativo por clave y ventana, y la serialización del resultado. El formato del evento resultante se aprecia en la Figura 4.5.

```
message WindowedAggregate {  
    string user_id = 1;  
    int64 window_start_ns = 2;  
    int64 window_end_ns = 3;  
    uint64 event_count = 4;  
    uint64 purchase_count = 5;  
    double value_sum = 6;  
    double value_min = 7;  
    double value_max = 8;  
    double value_avg = 9;  
    int64 quantity_sum = 10;  
    uint64 first_sequence_number = 11;  
    uint64 last_sequence_number = 12;  
    int64 max_event_time_ns = 13;  
    uint64 checksum = 14;  
}
```

Figura 4.5: Definición del evento WindowedAggregate

4.2.5.- Métricas utilizadas

Para llevar a cabo una evaluación comparativa justa de motores con diseños y principios distintos, se han seleccionado cuatro métricas que permiten caracterizar tanto la velocidad como el coste en recursos, estas métricas son: latencia, uso de CPU, uso de memoria y escritura en disco.

La latencia se mide de extremo a extremo, desde que el evento es generado hasta que es consumido por el tópico de salida. Esta métrica permite evaluar en qué medida cada uno introduce overhead adicional por su modelo de ejecución (gestión de checkpoints, comunicación entre operadores, etc.). En el caso de Spark, dado que su modelo de ejecución se basa en microlotes, la latencia incluye el tiempo de espera asociado al trigger del batch, por lo que los valores obtenidos permiten enseñar el impacto del modelo de procesamiento de streaming nativo frente al de microlotes. Para el job de ventana temporal, la latencia no se considera una métrica significativa, dado que una ventana retiene los

eventos hasta que se expira el límite de tiempo o el watermark cierra la ventana, por lo que el tiempo de procesamiento queda determinado por la propia semántica del watermark y no por el motor.

El uso de CPU se expresa en núcleos consumidos por los procesos asociados a cada job, capturando tanto el consumo del proceso principal como el de los componentes auxiliares de cada motor. El consumo de Flink y Kafka Streams refleja de forma directa la eficiencia de su bucle de procesamiento por evento. En Spark, en cambio, el uso de CPU permite analizar el coste asociado a la planificación y ejecución periódica de cada microlote.

El uso de memoria se mide a través del working set, que incluye la memoria que usa directamente el proceso (heap y stack) más la caché de ficheros que se ha usado recientemente. No es memoria 100% imprescindible: parte de esa caché se puede liberar si el sistema necesita espacio [72]. Por eso, en jobs con estado, conviene tener en cuenta que el working set puede crecer no solo porque aumenten las claves o el tamaño de los eventos, sino también por el uso de caché de disco.

Por último la escritura en disco es relevante para los jobs con estado, donde la presión sobre el almacenamiento local puede convertirse en un cuello de botella. En los tres motores se emplea RocksDB como backend de estado, por lo que esta métrica permite observar como las operaciones de volcados del estado a disco genera escrituras adicionales que crecen con el volumen de claves y eventos procesados, así como el impacto adicional que puede introducir los checkpoints periódicos bajo la garantía *exactly-once*.

4.2.6.- Escenarios

Todas las pruebas siguen el mismo procedimiento para ser ejecutadas. En primer lugar, se establece un período de calentamiento de 3 minutos, cuyas métricas no se tienen en cuenta en el análisis posterior. Este intervalo inicial es necesario debido a que las optimizaciones que la JVM aplica en tiempo de ejecución provocan que los valores obtenidos durante los primeros minutos no sean representativos del rendimiento.

Una vez transcurrido dicho período, se produce la recogida de métricas durante diez minutos, intervalo sobre el que se realiza el análisis. Finalmente, se añade un minuto adicional al final de cada ejecución que se utilizará para incluir los eventos que pueden emitirse después de los diez minutos en el job de agregado por ventana.

A continuación se presentan los tres escenarios de evaluación diseñados, cada uno de los cuales tiene como objetivo analizar el impacto de la variación de un único parámetro sobre el rendimiento del sistema, manteniendo el resto de parámetros constantes.

Parámetro	Valor
Eventos/s	10.000 - 200.000
Tamaño de evento	1 KB
Número de claves distintas	5.000
Jobs	Filtro sin estado, agregado por clave, agregado por ventanas
Garantías de entrega	<i>At-least-once</i> y <i>exactly-once</i> en Flink y Kafka streams, en Spark únicamente <i>at-least-once</i>

Tabla 4.1: Variación del throughput

Parámetro	Valor
Eventos/s	100.000
Tamaño de evento	128 B, 1 KB, 5 KB
Número de claves distintas	5.000
Jobs	Agregado por clave
Garantías de entrega	<i>At-least-once</i>

Tabla 4.2: Variación del tamaño de evento

Parámetro	Valor
Eventos/s	100.000
Tamaño de evento	1 KB
Número de claves distintas	1.000, 10.000, 100.000, 200.000
Jobs	Agregado por clave
Garantías de entrega	<i>At-least-once</i>

Tabla 4.3: Variación del número de claves distintas

4.3.- DISEÑO DEL SISTEMA

El sistema desarrollado está formado por los motores de procesamiento, que son los sistemas bajo prueba y ejecutan los jobs descritos, y tres herramientas de medición propias escritas en Go: el productor, el consumidor y el recolector de métricas. Todo ello se apoya en Kafka para la mensajería, Prometheus para la recogida de métricas y Grafana para su visualización. La Figura 4.6 muestra el diagrama del sistema.

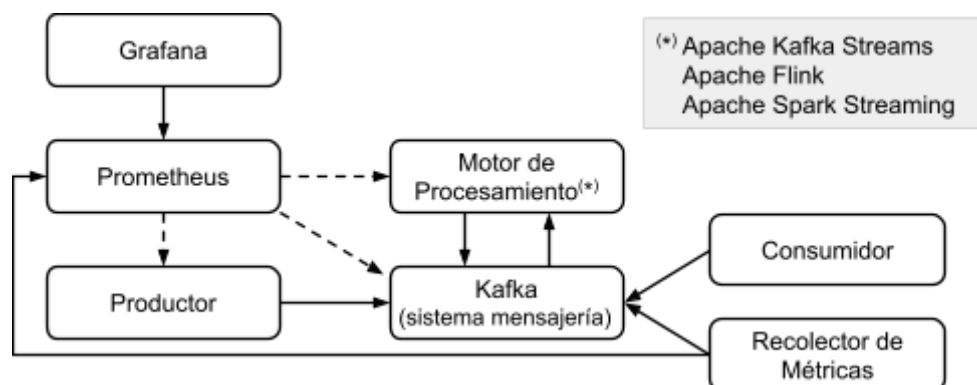


Figura 4.6: Diagrama de componentes del sistema

4.3.1.- Motores de procesamiento

Los tres motores no se evalúan con implementaciones arbitrarias, sino ejecutando los mismos tres jobs, implementados de forma equivalente sobre cada motor. Para que las diferencias observadas reflejen el modelo de ejecución de cada motor y no decisiones de configuración dispares, se fija un conjunto de parámetros comunes a los tres.

Los motores se configuran con un intervalo de checkpoint, commit o trigger de 1000 ms. Cada motor da un nombre distinto a este intervalo según el modelo de ejecución: en Flink, el checkpoint es la operación periódica que vuelca el estado de forma consistente para poder recuperarse ante fallos, en Kafka Streams, el commit marca cada cuánto se confirman los offsets procesados y se persiste el estado, y en Spark el trigger fija con qué frecuencia se lanza cada microlote. Igualar este intervalo en los tres garantiza que la frecuencia con la que cada motor persiste su estado no distorsiona la comparación.

Asimismo, los tres emplean RocksDB como backend de estado en los jobs con estado, de modo que el coste de acceso y persistencia del estado parte de una base común.

Entre ejecución y la siguiente se borra la información de los tópicos, de modo que cada job parte de cero y procesa únicamente la carga nueva generada durante su ejecución, sin arrastrar eventos residuales de pruebas anteriores. Por último, se desactivan las cachés de

escritura que algunos motores aplican por defecto sobre el estado, las cuales agruparían varias actualizaciones de una misma clave en una única emisión [73]. De este modo, el volumen de salida es equivalente entre los tres motores.

4.3.2.- Productor

El productor es el componente encargado de generar la carga y enviarla a Kafka, donde posteriormente será consumida por los motores de procesamiento. Se permite ajustar los parámetros como el throughput objetivo, tamaño del evento, número de claves distintas y la duración tanto de la ejecución como de la fase de calentamiento.

El cliente de Kafka del productor se configura con idempotencia y reconocimiento total de réplicas. En Kafka, cuando un productor envía un evento y no recibe confirmación del broker lo reintenta. Sin protección adicional, este reintento puede generar un duplicado. Un productor idempotente elimina este riesgo asignando a cada sesión de productor un identificador único, numerando secuencialmente cada evento enviado a cada partición. El broker almacena el último número de secuencia aceptado por partición y descarta silenciosamente cualquier evento que llegue con un número ya visto. De este modo se garantiza semántica *exactly-once* a nivel de productor.

Además, este productor configura un valor de *linger.ms* de 5 ms, indicando que se espere 5 ms antes de enviar un lote de eventos a Kafka, permitiendo que se acumulen más eventos en el mismo lote. Con un valor más bajo, habría más lotes, provocando más peticiones al broker por unidad de tiempo, pudiendo sobrecargar la red. Esta configuración se define en el archivo `runner.go` del productor y podría ser modificada. En la Figura 4.7 se aprecia la configuración utilizada.

```
func newKafkaClient(cfg config) (*kgo.Client, error) {
    return kgo.NewClient(
        kgo.SeedBrokers(cfg.brokers...),
        kgo.ClientID(clientID),
        kgo.RequiredAcks(kgo.AllISRacks()),
        kgo.ProducerLinger(5*time.Millisecond),
    )
}
```

Figura 4.7: Configuración del productor de Kafka

Para generar un throughput constante y configurable se utiliza un token bucket, un mecanismo que regula cuántos eventos se pueden enviar por unidad de tiempo y bloquea al productor cuando se supera ese límite [74]. Si se configuran 100.000 eventos por segundo y el bucle itera cada 100 ms, en cada iteración se envían 10.000 eventos, si el bucle avanza

demasiado rápido, el limitador lo detiene hasta que el intervalo se cumple, manteniendo así el throughput buscado de manera estable. En la Figura 4.8 se aprecia la implementación utilizada.

```
func runProducer(ctx context.Context, client *kgo.Client, cfg
config, msgen MessageGenerator) {
    batch := batchSizeForRate(int(cfg.targetRate), cfg.interval)
    limiter := rate.NewLimiter(rate.Limit(cfg.targetRate), batch)

    for {
        if err := limiter.WaitN(ctx, batch); err != nil {
            return
        }
        produceBatch(ctx, client, cfg.topic, batch, msgen)
    }
}
```

Figura 4.8: Lógica de generación de eventos en base a un throughput determinado

La decisión de agrupar los eventos en lotes, en lugar de producirlos de uno en uno, es con la intención de reducir el número de peticiones y permitir un alto throughput sin que el propio productor se convierta en el cuello de botella.

Respecto a la generación de la clave, el `user_id` se obtiene seleccionando de forma uniforme un entero dentro del rango de claves distintas configurado. Dado que Kafka asigna las particiones a partir del hash de la clave, una distribución uniforme reparte la carga de manera homogénea entre las particiones, evitando que aparezcan particiones calientes que sesguen las mediciones.

4.3.3.- Consumidor

El consumidor es el componente encargado de la medición de la latencia del sistema. Se suscribe al tópico de salida del job y, por cada evento recibido, deserializa el evento, calcula la latencia de extremo a extremo y la acumula en un histograma.

La latencia se calcula como la diferencia entre el instante en el que el consumidor lee el evento y el instante en el que se generó. Las muestras se acumulan en un histograma de alta resolución dinámica (HDR) [75] configurado para capturar valores entre 1 ns y 60 segundos con tres cifras significativas, lo que permite obtener percentiles precisos sin almacenar las muestras individuales.

Por último, cuando se evalúan las garantías *exactly-once*, el consumidor lee con nivel de aislamiento *read_committed*. Este nivel hace que el consumidor solo entregue los eventos

pertenecientes a transacciones que el motor ha confirmado, ignorando las escrituras que aún forman parte de una transacción no confirmada. De esta forma, el consumidor nunca contabiliza como salida un evento que el motor podría descartar, lo cual es imprescindible para medir correctamente bajo *exactly-once*. Cuando los motores operan en *at-least-once* no producen escrituras transaccionales, por lo que todos los eventos del tópico están siempre disponibles, este nivel de aislamiento no altera el comportamiento ni penaliza la medición.

4.3.4.- Coordinación entre componentes

Para coordinar el productor y el consumidor se hace uso de un tópico de control además del de los datos, como se aprecia en la Figura 4.9. El productor publica dos eventos en este tópico a lo largo de la ejecución: uno al inicio y otro al final. El marcador de inicio indica los parámetros del experimento: throughput, tamaño de payload, número de claves, duración de la fase de calentamiento e identificador de ejecución. El consumidor, al recibirlo, sabe exactamente con qué configuración fue lanzado el experimento, sin necesidad de coordinación externa. El marcador de finalización dispara la fase de drenaje, además de ser utilizado posteriormente para recolectar las métricas.

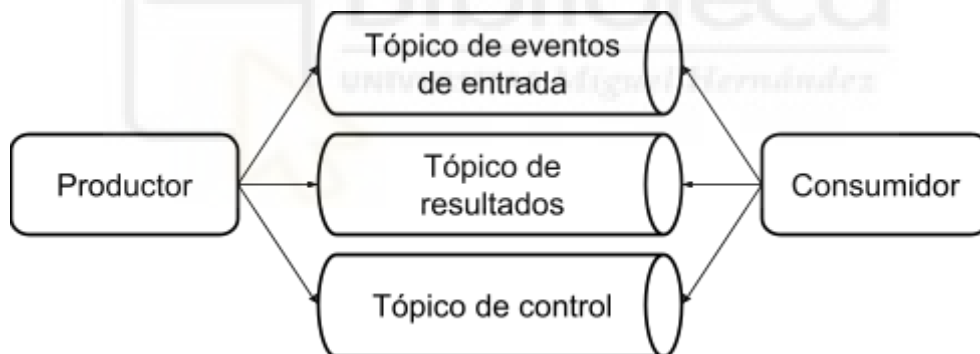


Figura 4.9: Coordinación del productor y consumidor mediante tópicos

El consumidor filtra los marcadores por marca temporal e identificador de ejecución, de modo que ignora marcadores antiguos que pudieran quedar en el tópico de control procedentes de ejecuciones anteriores.

Una vez finalizada la ejecución, el consumidor escribe un fichero de metadatos (meta.json) que actúa como puente entre la medición de latencia y la recolección de métricas de recursos. Este fichero contiene el identificador de ejecución, el motor y job evaluados, el modo de garantía, el tópico de salida, los parámetros del experimento y las marcas temporales de pared de inicio y fin obtenidas de los marcadores de control. Estos timestamps son los que delimitan con precisión la ventana de tiempo de cada ejecución concreta.

4.3.5.- Recolección de métricas

Durante la ejecución de las pruebas, Prometheus recoge continuamente métricas de todos los componentes. El servicio del recolector se encarga de extraer, una vez terminada la prueba, únicamente los datos correspondientes a cada ejecución concreta.

Para ello, el recolector lee el fichero meta.json generado por el consumidor y, a partir de las marcas temporales de inicio y fin que contiene, construye la ventana de consulta. A esta ventana se le descuenta la duración de calentamiento desde el inicio.

Sobre ese intervalo, el recolector recoge las métricas mediante expresiones PromQL [76]. El resultado de cada métrica se almacena en un fichero CSV independiente por ejecución, listo para el análisis posterior.

4.4.- INFRAESTRUCTURA

Para el desarrollo y posterior ejecución de las pruebas de rendimiento de este proyecto, se han construido dos entornos de ejecución sobre Kubernetes. El primero consiste en un entorno local simplificado enfocado para desarrollar los distintos componentes y validar sin coste. El segundo es un clúster desplegado sobre GCP (Google Cloud Platform), específicamente para aislar los recursos durante las mediciones además de disponer de mayores recursos.

4.4.1.- Construcción de imágenes

Todos los componentes del sistema se distribuyen como imágenes de Docker, lo que garantiza que se ejecuta de manera idéntica tanto en local como en GCP. Tanto las herramientas de medición (productor, consumidor y recolector) como cada motor de procesamiento disponen de su propia imagen.

En el caso de los motores, se construye una única imagen por motor que contiene los tres jobs, y se podrá seleccionar cada uno mediante variables de entorno. Cada imagen parte de la imagen base oficial del motor correspondiente y le incorpora los tres jobs ya compilados. La Figura 4.10 muestra el Dockerfile de Flink, donde se copian los tres jobs compilados.

```
FROM apache/flink:2.2.1-scala_2.12-java21

RUN curl -fsSL \

https://repo1.maven.org/maven2/org/apache/flink/flink-statebacke
nd-rocksdb/2.2.1/flink-statebackend-rocksdb-2.2.1.jar \
    -o /opt/flink/lib/flink-statebackend-rocksdb-2.2.1.jar

COPY stateless-filter/target/stateless-filter-1.0.jar
/opt/flink/usrlib/stateless-filter.jar

COPY keyed-aggregator/target/keyed-aggregator-1.0.jar
/opt/flink/usrlib/keyed-aggregator.jar

COPY window-aggregator/target/window-aggregator-1.0.jar
/opt/flink/usrlib/window-aggregator.jar
```

Figura 4.10: Dockerfile de Flink

4.4.2.- Plataforma sobre Kubernetes

El proyecto se despliega sobre Kubernetes, que facilita el cambio entre la infraestructura local y la de la nube.

El despliegue se apoya en el patrón operador. Un operador es un componente que extiende Kubernetes con conocimiento específico de una aplicación, de modo que esta se gestiona de forma declarativa: en lugar de escribir manualmente los recursos de bajo nivel, basta con declarar el estado deseado y el operador se encarga de crearlo, mantenerlo y recuperarlo ante fallos. Se emplean tres operadores. Strimzi despliega y administra el clúster de Kafka, incluyendo el broker, la configuración del protocolo KRaft y la creación de los tópicos [77]. El operador de Flink gestiona el ciclo de vida de los jobs de Flink, levantando el JobManager y los TaskManagers. Por último, el operador de Spark traduce la definición de cada aplicación en el driver y los executors correspondientes.

La monitorización se basa en el stack kube-prometheus-stack [78], que instala Prometheus mediante su propio operador. El descubrimiento de los objetivos a monitorizar se realiza de forma declarativa a través de objetos PodMonitor, uno por cada tipo de componente, que indican a Prometheus de qué pods debe extraer métricas. Sobre los datos recogidos por Prometheus, Grafana proporciona la capa de visualización, con paneles específicos para cada motor de procesamiento, para Kafka y para el productor, lo que permite seguir en tiempo real el comportamiento del sistema durante las pruebas.

Por último, los manifiestos de despliegue se organizan con Kustomize [79], estructurándolos en una capa base común y en overlays específicos para cada entorno. De esta forma, las diferencias entre el entorno local y el de GCP, fundamentalmente de dimensionamiento, se expresan como variaciones sobre una misma base compartida, sin duplicar la definición del sistema.

4.4.3.- Infraestructura local

Para el proceso de desarrollo, depuración y validación previa de los manifiestos de despliegue, se ha diseñado un entorno de ejecución en local basado en Kind. En este entorno local, los recursos físicos son compartidos en un único equipo, por lo que no es posible garantizar el aislamiento físico ni emular las pruebas de rendimiento debido a los bajos recursos. Sin embargo, el valor de este entorno es asegurar la reproducibilidad del sistema y validar la lógica.

La arquitectura del clúster de este entorno son dos nodos: un nodo de control y un nodo de trabajo. Esta configuración está definida en el manifiesto kind-config.yaml, donde la configuración utilizada se observa en la Figura 4.11. En caso de querer aumentar el número, este archivo tendría que ser modificado.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
```

Figura 4.11: Configuración de clúster local de Kind

En este entorno todos los servicios (Kafka, Prometheus, Grafana, los operadores y los motores de procesamiento) se ubican en el mismo nodo para ahorrar recursos, lo que se expresa mediante el overlay de Kustomize correspondiente a este entorno.

4.4.4.- Infraestructura en Google Cloud Platform (GCP)

El entorno de GCP se ejecuta sobre Google Kubernetes Engine y se ha diseñado aislando los recursos físicos. Para lograrlo se ha hecho uso del modo de operación GKE standard en lugar del modo GKE Autopilot. Aunque Autopilot simplifica la administración, limita el control del usuario sobre el hardware físico. Al optar por GKE Standard, se obtiene un control total y de bajo nivel de la infraestructura del clúster, con la intención de configurar los tipos de máquinas, almacenamiento y políticas de planificación específicas.

Toda la infraestructura de GKE ha sido desplegada en la región geográfica us-central1 situada en Iowa, Estados Unidos, concretamente en la zona de disponibilidad us-central1-a. La elección de esta localización responde a criterios de coste y cantidad de hardware, la región de Iowa presenta uno de los mejores ratios de precio y cantidad de instancias disponibles a fecha de Mayo 2026 [80].

El clúster se segmenta en tres grupos de nodos, implementados sobre instancias de tipo Spot para optimizar costes. Las instancias de tipo Spot, son máquinas virtuales que aprovechan el exceso de capacidad de cómputo disponible en los centros de datos de Google, ofreciéndose con un descuento que puede llegar hasta el 90% en comparación con las tarifas estándar [81]. A pesar de que el proveedor puede reclamar estas instancias en cualquier momento si la demanda global del sistema aumenta, no resulta un problema para el proyecto, ya que podemos volver a ejecutar la prueba en caso de fallo. La Tabla 4.4 muestra un resumen de los nodos y hardware utilizado.

Grupo de nodos	Número de nodos	Tipo de máquina	Disco
Operaciones	1	e2-standard-8 (8 vCPUs, 32 GB RAM)	Disco persistente equilibrado de 100 GB
Kafka	1	n2-standard-4 (4 vCPUs, 16 GB RAM)	Disco persistente de 50 GB + 2 SSDs locales efímeros
Worker	3	n2-standard-4 (4 vCPUs, 16 GB RAM)	Disco persistente de 50 GB + 1 SSD local efímero por nodo

Tabla 4.4: Infraestructura de Google Cloud Platform

El primer grupo de nodos, consiste en una única instancia e2-standard-8 asociada a un almacenamiento de disco equilibrado de 100 GB. La finalidad de este nodo es centralizar todos los servicios de observabilidad, envío y recepción de eventos para evitar que afecte a los nodos responsables de ejecutar los motores de procesamiento. En él se despliegan los operadores, el stack de monitorización con Prometheus y Grafana. Por último también se añaden el JobManager de Flink y el Spark Driver, encargados de la coordinación de los respectivos workers.

El segundo grupo de nodos, el pool de Kafka, consta de una única instancia n2-standard-4 con dos unidades de estado sólido de conexión local. Esta máquina aloja únicamente el broker de Kafka, que integra los roles de controller y broker mediante el protocolo KRaft. A diferencia de los discos persistentes estándar de GCP, que operan mediante red de almacenamiento externo, los discos locales SSD son dispositivos físicos integrados directamente en el servidor que aloja la máquina. Esta proximidad física elimina la latencia de red de las escrituras y lecturas de disco. Dado que la persistencia de los datos en estas

unidades está ligada a la vida del servidor físico, perdiéndose la información si el nodo se apaga o recrea, Kubernetes expone y administra este recurso bajo el concepto de almacenamiento efímero, el cual representa un espacio de disco de carácter temporal reservado únicamente para el ciclo de vida del contenedor.

Por último, el pool de workers, formado por tres nodos físicos independientes del tipo n2-standard-4, Este conjunto de máquinas está destinado exclusivamente al procesamiento de los datos. Se despliega exactamente una instancia de ejecución por nodo físico.

En entornos virtualizados y de contenedores, la precisión de las métricas de rendimiento puede verse severamente alterada por el fenómeno conocido como noisy neighbours [82]. Este problema ocurre cuando múltiples procesos compiten por el tiempo de CPU, ancho de banda del bus de memoria o las operaciones de disco en un mismo servidor físico. Para mitigarlo, el clúster se apoya en tres mecanismos de Kubernetes: taints, tolerations y reglas de antiafinidad.

Los taints son marcas de exclusión aplicadas directamente sobre los nodos físicos durante la creación del clúster en GKE. Los nodos dedicados a Kafka y a los motores de procesamiento reciben la *NoSchedule*, lo que le indica al planificador que rechace cualquier pod que no esté explícitamente autorizado a ejecutarse en ellos. La autorización se concede mediante tolerations, definidas en la especificación de cada pod. Sin embargo, una toleration únicamente elimina el rechazo, no garantiza que el pod se ubique en el nodo deseado. Para forzar esa colocación, los pods de los motores emplean un nodeSelector que los vincula a los nodos.

Finalmente, para garantizar una distribución homogénea entre los nodos de procesamiento, cada motor define una regla de antiafinidad de carácter obligatorio. Esta regla impide que el planificador coloque dos instancias del mismo tipo de carga en el mismo servidor físico.

4.5.- RESULTADOS

4.5.1.- Variación del número de eventos

4.5.1.1.- Filtro sin estado

El job de filtro sin estado se utiliza como escenario base para evaluar el comportamiento de los motores cuando el procesamiento no requiere mantener estado ni realizar agregaciones. En esta prueba, cada evento puede procesarse de forma independiente: el motor consume el

evento de entrada, aplica una condición de filtrado, transforma el resultado y lo escribe en el t pico de salida. Por tanto, la carga principal procede de la lectura y escritura en Kafka, la serializaci n y deserializaci n de los eventos, y el coste propio del modelo de ejecuci n de cada motor.

Motor	Modo	Throughput	Media (ms)	Desviaci�n est�ndar (ms)	p50 (ms)	p99 (ms)
Flink	<i>At-least-once</i>	10.000	11,94	2.20	11,87	15,82
		25.000	13,51	17,38	12,75	19,42
		50.000	11,23	8,60	10,78	17,37
		100.000	12,09	9,60	11,47	20,05
		200.000	12,48	7,74	11,54	36,34
	<i>Exactly-once</i>	10.000	578,50	286,21	549,45	1.033,57
		25.000	528,11	284,29	555,22	983,56
		50.000	535,19	282,35	552,60	994,05
		100.000	574,03	281,56	583,01	1.041,76
		200.000	680,84	246,84	672,14	1.138,75
Kafka Streams	<i>At-least-once</i>	10.000	69,55	53,28	108,53	115,02
		25.000	46,49	44,83	16,54	115,54
		50.000	27,16	31,69	14,42	104,20
		100.000	19,15	20,44	13,57	86,25
		200.000	14,72	11,31	13,40	39,06
	<i>Exactly-once</i>	10.000	560,43	318,64	519,57	1.114,64
		25.000	549,25	310,71	524,55	1.115,68
		50.000	528,71	299	523,24	1.027,08
		100.000	547,30	294,81	543,69	1.054,34
		200.000	643,24	264,01	647,50	1.141,90
Spark	<i>At-least-once</i>	10.000	608,18	275,90	637,53	1.052,25
		25.000	591,97	275,16	623,90	1.030,75
		50.000	652,91	267,99	641,20	1.102,05
		100.000	694,45	239,45	687,34	1.095,76
		200.000	755,71	199,58	759,69	1.096,81

Tabla 4.5: Latencia extremo a extremo del job de filtro sin estado

En la Tabla 4.5 se observa que, en el job de filtro sin estado, la latencia no viene determinada  nicamente por el motor utilizado, sino principalmente por el modelo de emisi n de cada configuraci n. Flink en modo *at-least-once* mantiene una latencia pr cticamente plana, en torno a 12 ms, para todo el rango de throughput evaluado. Esto

sugiere que, al no existir estado ni redistribuciones intermedias, el coste por evento permanece estable y el motor es capaz de procesar cada evento poco después de recibirlo.

Kafka Streams en modo *at-least-once* presenta un comportamiento diferente: la latencia media disminuye al aumentar el throughput, pasando de 69,55 ms a 10.000 eventos/s a 14,72 ms a 200.000 eventos/s. La causa probable es su modelo de lectura por lotes desde Kafka. Cuando el throughput de entrada es baja, algunos eventos esperan más tiempo hasta ser leídos por el siguiente ciclo de consumo; en cambio, al aumentar el throughput, los lotes se llenan antes y el tiempo medio de espera disminuye. Además, el percentil 50 también se reduce hasta quedar muy próximo al de Flink en las tasas más altas. Por tanto, en este job sin estado, Flink ofrece una latencia baja y estable, mientras que Kafka Streams converge hacia valores similares conforme aumenta la carga.

En las configuraciones *exactly-once* de Flink y Kafka Streams se observa una penalización clara: la latencia media pasa a situarse aproximadamente entre 530 y 680 ms, con percentiles 99 cercanos o superiores a 1.000 ms. En esta configuración, el coste observado está dominado por la visibilidad transaccional de Kafka y por el intervalo de commit/checkpoint de 1.000 ms, por lo que las diferencias entre motores quedan parcialmente ocultas. El evento puede haber sido procesado internamente antes, pero no es visible para el consumidor hasta que la transacción se confirma. Por tanto, la métrica de latencia extremo a extremo refleja no solo tiempo de cómputo, sino también tiempo de espera hasta el commit. Como conclusión, *exactly-once* introduce en este escenario una barrera de visibilidad que desplaza la latencia hacia la escala del segundo, incluso en un job sin estado.

Spark, aunque se evalúa únicamente en modo *at-least-once*, presenta latencias medias entre 608 y 756 ms, próximas a las observadas en Flink y Kafka Streams bajo *exactly-once*. La causa es su modelo de ejecución mediante microlotes con un trigger de 1.000 ms. Los eventos se emiten al cierre del microlote, por lo que la latencia incluye el tiempo de espera hasta la siguiente ejecución del batch y el tiempo necesario para procesarlo. Por otra parte, la latencia aumenta progresivamente con el throughput, lo que indica que al tiempo medio de espera se le añade un coste de cómputo creciente. En consecuencia, Spark muestra una latencia estructuralmente superior en este escenario, no por un mayor coste de transformación por evento, sino por el modelo de emisión por microlotes.

La Figura 4.12 ilustra el consumo de CPU, a throughput bajos los tres motores se mantienen próximos, entre 0,2 y 0,5 núcleos, con Spark partiendo algo por encima por el coste fijo de planificar un microlote aun con poca carga. Al aumentar el throughput las pendientes divergen y las curvas se cruzan en torno a 50.000-100.000 eventos/s: Spark, que partía como el más caro, deja de serlo, y Flink pasa a ser el de mayor consumo. A 200.000 eventos/s, Flink es el más costoso, alrededor de 2,8 núcleos con las curvas *exactly-once* y *at-least-once* cerca de estar solapadas. Ese cruce ilustra dos decisiones de diseños

opuestos: Spark presenta un coste fijo elevado pero una pendiente suave, amortiza el procesamiento por lote, mientras que Flink parte de un coste fijo bajo pero resulta más caro por evento procesado a medida que crece el throughput. Que en Flink las curvas de ambos modos estén casi solapadas confirma que, en ausencia de estado, el sobre coste de las transacciones sobre el CPU es despreciable.

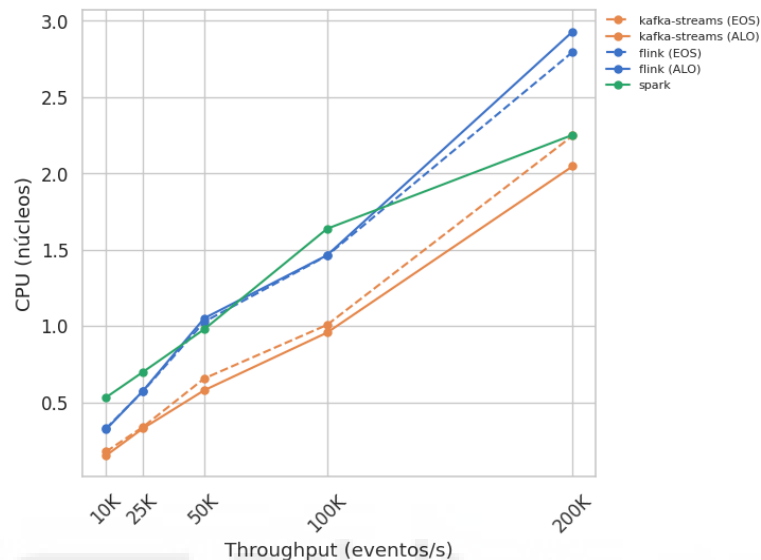


Figura 4.12: Uso de CPU en el filtro sin estado frente al throughput

El uso de memoria se ve representado en la Figura 4.13, que es completamente estable frente al throughput y queda separado por motor: Spark en torno a los 14 GB, Flink a los 11 GB y Kafka Streams a los 4,5 GB. Al no haber estado, la memoria la determina el dimensionamiento del heap de la JVM y no la carga procesada. Spark es el que reserva memoria de forma más agresiva, mientras que Kafka Streams resulta unas tres veces más ligero que los otros dos.

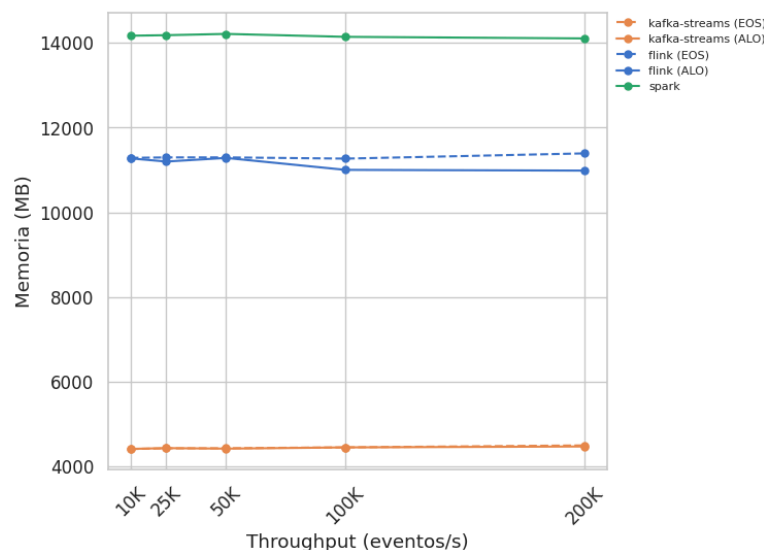


Figura 4.13: Uso de memoria en el filtro sin estado frente al throughput

El job de filtro sin estado muestra que, cuando no existe estado ni redistribución intermedia, Flink y Kafka Streams pueden alcanzar latencias muy bajas en modo *at-least-once*. Flink mantiene una latencia prácticamente constante en todo el rango de throughput, mientras que Kafka Streams mejora conforme aumenta el throughput de entrada y los ciclos de consumo se aprovechan de forma más eficiente. Spark, por el contrario, presenta una latencia estructuralmente superior debido a su modelo de microlotes, aunque su consumo de CPU crece con una pendiente más suave a cargas altas.

La comparación entre *at-least-once* y *exactly-once* en Flink y Kafka Streams muestra que la garantía de entrega tiene un impacto mucho mayor en la latencia que en el consumo de CPU para este job. En esta configuración, la latencia bajo *exactly-once* queda dominada por el intervalo de commit/checkpoint de 1.000 ms y por la visibilidad transaccional de Kafka, mientras que el coste computacional adicional se reduce al no existir estado complejo que persistir.

4.5.1.2.- Agregado por clave

El job de agregado por clave introduce una carga más representativa de los motores de procesamiento en streaming, ya que cada evento no solo debe transformarse y emitirse, sino también actualizar un estado asociado a una clave. En este caso, el motor mantiene un agregado acumulativo por *user_id*, de modo que cada nuevo evento modifica los contadores, sumas y valores agregados correspondientes a esa clave antes de producir un nuevo resultado.

Este escenario permite evaluar el impacto de añadir estado al procesamiento. A diferencia del filtro sin estado, donde los eventos son independientes entre sí, en el agregado por clave el rendimiento depende también del acceso al estado, de su actualización, de su persistencia y de la forma en que cada motor distribuye las claves entre sus tareas.

En la Tabla 4.6 se muestra la latencia extremo a extremo del job de agregado por clave. Hasta 100.000 eventos/s, los resultados mantienen una tendencia similar a la observada en el filtro sin estado: Flink y Kafka Streams en modo *at-least-once* se sitúan en valores de pocas decenas de milisegundos, mientras que las configuraciones *exactly-once* y Spark permanecen en una escala cercana al segundo.

Sin embargo, a 200.000 eventos/s aparece una diferencia significativa. Flink deja de operar en régimen estable: su latencia media aumenta hasta 23.320,98 ms en *at-least-once* y hasta 39.231,16 ms en *exactly-once*, con percentiles 99 superiores a 67 segundos. Kafka Streams, en cambio, mantiene una latencia baja incluso en el punto de mayor carga, y Spark incrementa su latencia de forma más moderada, aunque sigue condicionado por su

modelo de microlotes. Por tanto, la latencia de Flink a 200.000 eventos/s no debe interpretarse como la latencia normal del motor bajo carga estable, sino como la consecuencia de una cola creciente de eventos pendientes de procesar.

Motor	Modo	Throughput	Media (ms)	Desviación estándar (ms)	p50 (ms)	p99 (ms)
Flink	<i>At-least-once</i>	10.000	51,07	36,27	48,30	109,31
		25.000	37,54	29,01	20,25	107,94
		50.000	29,32	22,26	21,02	99,88
		100.000	27,03	14,77	23,22	82,38
		200.000	23.320,98	23.107,98	11953,77	67.746,40
	<i>Exactly-once</i>	10.000	559,06	288,18	595,59	1.011,88
		25.000	551,89	285,22	576,19	1.009,25
		50.000	584,02	284,33	597,16	1.045,95
		100.000	660,39	284,10	659,03	1.147,14
		200.000	39.231,16	16.530,85	38.151,39	68.081,94
Kafka Streams	<i>At-least-once</i>	10.000	67,42	50,25	109,18	115,02
		25.000	50,89	47,23	17,92	114,95
		50.000	31,73	32,99	16,96	106,04
		100.000	22,56	21,25	16,35	88,28
		200.000	18,59	12,26	16,26	58,92
	<i>Exactly-once</i>	10.000	559,61	320,38	519,83	1.116,73
		25.000	543,54	308,75	522,19	1.115,68
		50.000	530,86	299,71	524,55	1.029,18
		100.000	540,91	294,83	536,87	1.043,86
		200.000	600,56	272,78	594,54	1.095,76
Spark	<i>At-least-once</i>	10.000	729,72	287,86	762,84	1.189,09
		25.000	761,39	288,78	773,85	1.243,61
		50.000	750,69	288,60	757,07	1.234,17
		100.000	856,58	292,54	857,21	1.382,02
		200.000	1.044,16	299,54	1.042,81	1.608,52

Tabla 4.6: Latencia extremo a extremo del job de agregado por clave

La causa de este comportamiento es la saturación de CPU provocada por la redistribución del flujo tras la operación keyBy. Aunque los eventos llegan a Kafka particionados por user_id, Flink vuelve a repartir el flujo entre sus subtareas para materializar la agrupación por clave. Esta redistribución implica serialización, deserialización y transferencia de datos entre subtareas, añadiendo un coste considerable por evento. Kafka Streams, por el contrario, se beneficia de que la clave lógica de agregación coincide con la clave de particionado de Kafka. Cada instancia procesa las claves de las particiones que tiene asignadas, por lo que puede actualizar el estado localmente sin realizar una fase adicional de repartición.

La Figura 4.14 confirma esta hipótesis desde el punto de vista del consumo de CPU. Flink pasa de aproximadamente 1,2 núcleos a 10.000 eventos/s a unos 6 núcleos a 100.000 eventos/s, alcanzando alrededor de 8,4 núcleos a 200.000 eventos/s. Dado que el despliegue dispone de nueve núcleos para el procesamiento, este último punto se encuentra muy próximo a la saturación. Kafka Streams, en cambio, se mantiene en torno a 4,5-4,9 núcleos en el punto de mayor carga, con un consumo ligeramente superior en *exactly-once* por el coste de las transacciones. Spark alcanza aproximadamente 3,5 núcleos a 200.000 eventos/s, mostrando la pendiente de crecimiento más suave. En este escenario concreto y bajo esta configuración, Spark amortiza mejor el coste de CPU al trabajar por microlotes, aunque esto no implica una menor latencia extremo a extremo.

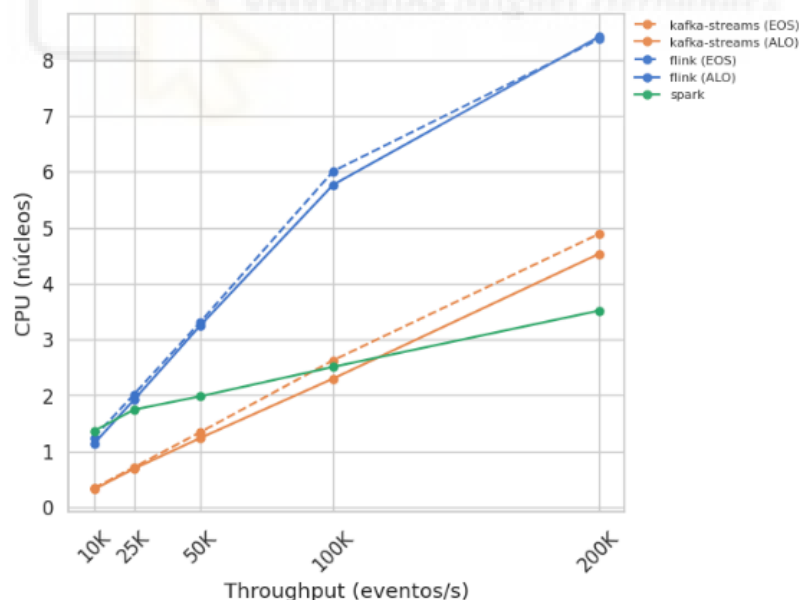


Figura 4.14: Uso de CPU en el agregado por clave frente al throughput

La diferencia entre Flink y Kafka Streams se aprecia con mayor claridad en la Figura 4.15, donde se muestra el tráfico de red redistribuido por Flink en el agregado por clave a 200.000 eventos/s. En este punto, Flink mueve internamente alrededor de 131 MB/s, lo que indica que una parte muy elevada del flujo está siendo reenviada entre subtareas. Esta

redistribución explica que el consumo de CPU crezca con tanta rapidez y que el límite de rendimiento aparezca antes que en Kafka Streams. La comparación no implica que Flink sea siempre menos eficiente en agregaciones por clave, sino que, en esta implementación concreta, Kafka Streams aprovecha mejor el particionado previo de Kafka, mientras que Flink introduce una redistribución adicional.

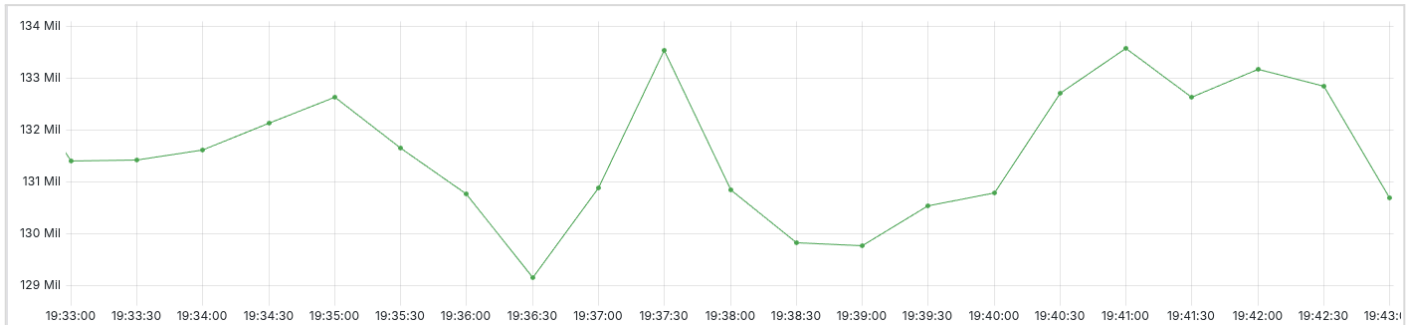


Figura 4.15: Tráfico de red de Flink en el agregado por clave a 200.000 eventos/s

La saturación se confirma al observar la evolución temporal del lag del consumidor de Flink. En la Figura 4.16, con 100.000 eventos/s, el lag oscila de forma acotada alrededor de 30.000 eventos y no muestra una tendencia creciente. Esto indica que el motor consume la entrada al mismo ritmo al que se produce. En cambio, en la Figura 4.17, con 200.000 eventos/s, el lag crece de forma lineal durante toda la ejecución, pasando aproximadamente de 8 millones a 26 millones de eventos en diez minutos. La pendiente, cercana a 30.000 eventos por segundo, indica que Flink solo consigue procesar alrededor de 170.000 de los 200.000 eventos/s entrantes. Por tanto, el sistema no está degradándose únicamente en latencia, sino que está acumulando backlog de forma sostenida.

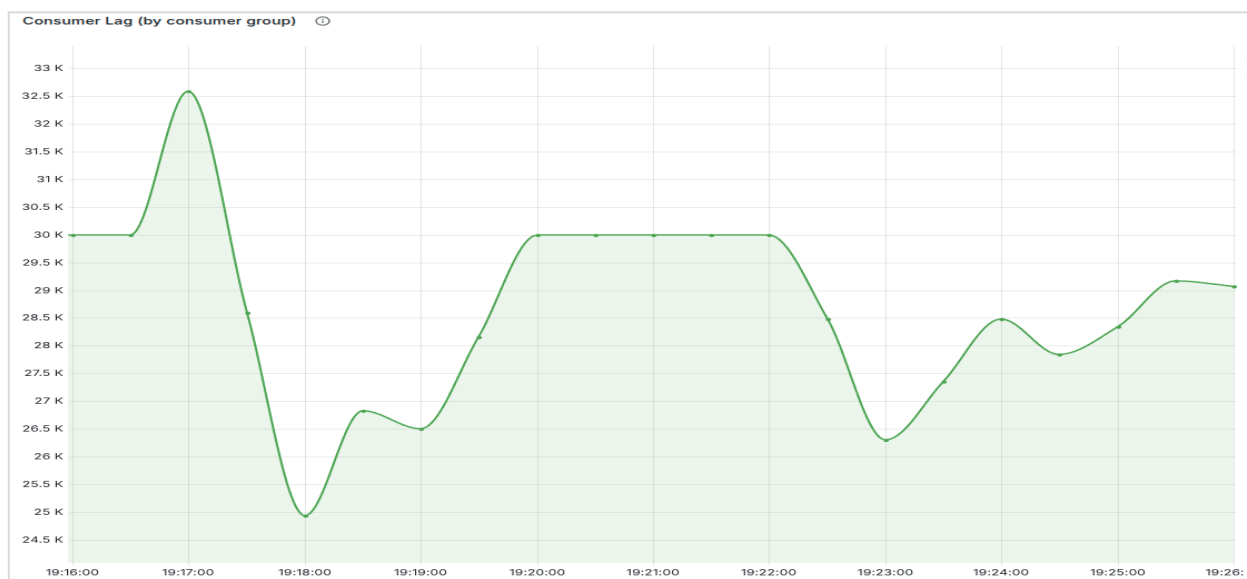


Figura 4.16: Evolución del lag del consumidor de Flink con 100.000 eventos/s

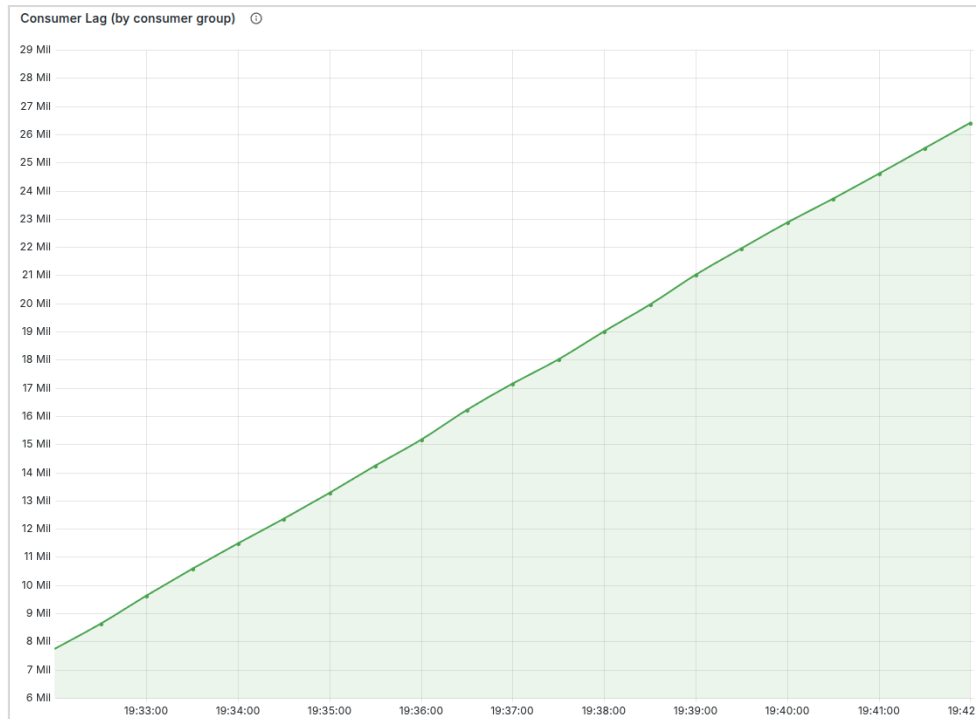


Figura 4.17: Evolución del lag del consumidor de Flink a 200.000 eventos/s

El uso de memoria, representado en la Figura 4.18, vuelve a ser prácticamente plano frente al throughput. Spark se mantiene en torno a los 15-16 GB, Flink alrededor de los 12 GB y Kafka Streams cerca de los 4,5-5 GB. Esta estabilidad indica que, en este escenario, el aumento de throughput no incrementa de forma significativa el tamaño del estado mantenido, ya que el número de claves distintas permanece fijo en 5.000. Por ello, la memoria no aparece como el recurso limitante principal. El cuello de botella se manifiesta antes en CPU y en la redistribución interna de datos.

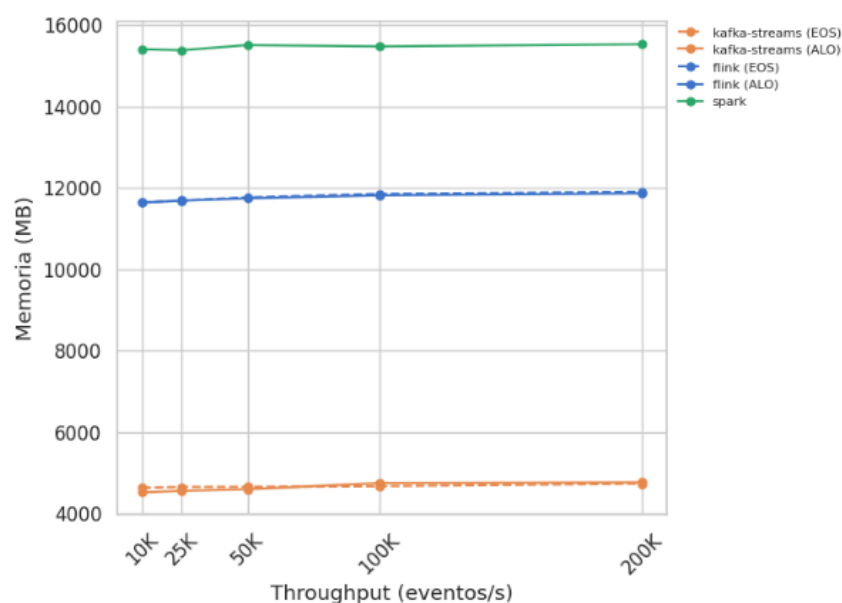


Figura 4.18: Uso de memoria en el job de agregado por clave frente al throughput

La Figura 4.19 muestra la escritura en disco del job de agregado por clave. Flink escribe de forma casi constante, en torno a 2,5-2,65 MB/s, independientemente del throughput. Este comportamiento sugiere que la escritura está dominada por la cadencia de checkpoints y por el tamaño del estado, más que por el número de eventos procesados por segundo. Spark, en cambio, incrementa su escritura con la carga, pasando de aproximadamente 0,57 MB/s a 2,5 MB/s. La causa probable es que, en cada microlote, Spark materializa en disco parte del reparto por clave y de las actualizaciones de estado. Como el volumen de datos por lote crece con el throughput, también lo hace la escritura. Kafka Streams presenta una separación clara entre modos: en *exactly-once* la escritura aumenta con la carga, mientras que en *at-least-once* se mantiene prácticamente nula en las métricas locales observadas.

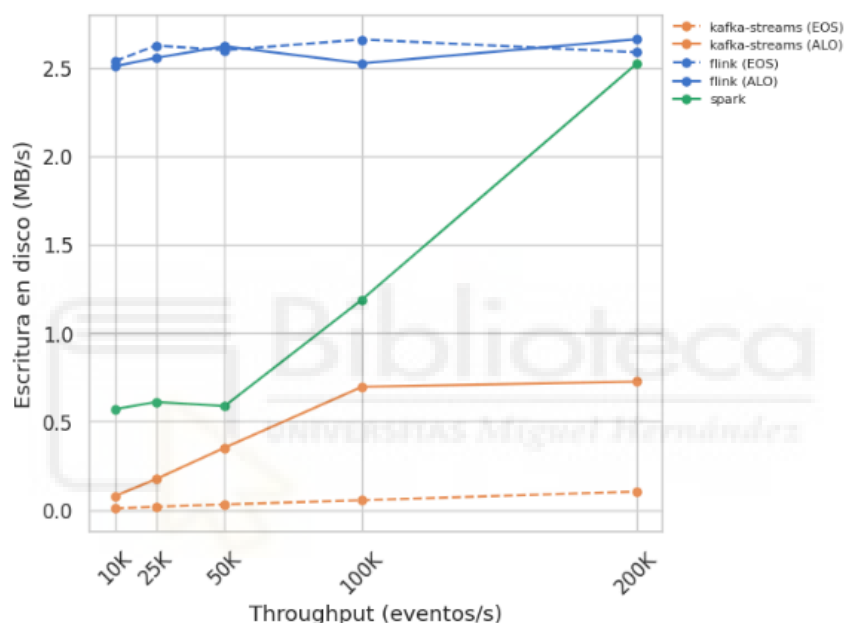


Figura 4.19: Escritura en disco en el job de agregado por clave frente al throughput

No obstante, el bajo consumo de disco local de Kafka Streams debe interpretarse con cautela. Kafka Streams mantiene la durabilidad del estado mediante tópicos de changelog en Kafka. Con la caché de escritura desactivada, cada actualización del estado puede generar una escritura adicional en el changelog, además del evento emitido al tópico de salida. Por tanto, parte del coste de durabilidad no aparece en las métricas de disco del pod del motor, sino que se desplaza al broker de Kafka. Flink y Spark reflejan más directamente este coste en sus checkpoints y escrituras locales, mientras que Kafka Streams lo externaliza hacia Kafka.

En conjunto, el job de agregado por clave muestra que la introducción de estado modifica de forma clara el comportamiento respecto al filtro sin estado. Kafka Streams obtiene el resultado más estable en este escenario porque la clave de agregación coincide con el particionado de Kafka y evita redistribuciones adicionales. Spark mantiene una latencia

más alta debido a su modelo de microlotes, pero presenta una pendiente de CPU reducida a cargas elevadas. Flink ofrece latencias bajas mientras el sistema permanece en régimen estable, pero a 200.000 eventos/s queda limitado por la redistribución del flujo y la saturación de CPU, lo que provoca crecimiento de lag y una latencia dominada por backlog acumulado.

4.5.1.3.- Agregado por ventana

En el job de agregado por ventanas no se analiza la latencia extremo a extremo como métrica principal, ya que la emisión de resultados no depende únicamente del tiempo de procesamiento del motor, sino también de la semántica temporal de la ventana y del cierre mediante watermark. A diferencia del agregado por clave, donde se emite un resultado por cada evento procesado, en este caso cada motor acumula eventos por clave y ventana, y solo emite el resultado cuando la ventana se considera completa.

La Figura 4.20 muestra que el uso de CPU mantiene una tendencia similar a la observada en el agregado por clave, aunque con una presión algo menor en el punto de mayor carga. Flink vuelve a presentar la pendiente de crecimiento más pronunciada, pasando de aproximadamente 0,8 núcleos a 10.000 eventos/s a unos 7,6 núcleos a 200.000 eventos/s. Kafka Streams crece de forma más moderada, hasta situarse alrededor de 4,5 núcleos, mientras que Spark alcanza aproximadamente 4,25 núcleos.

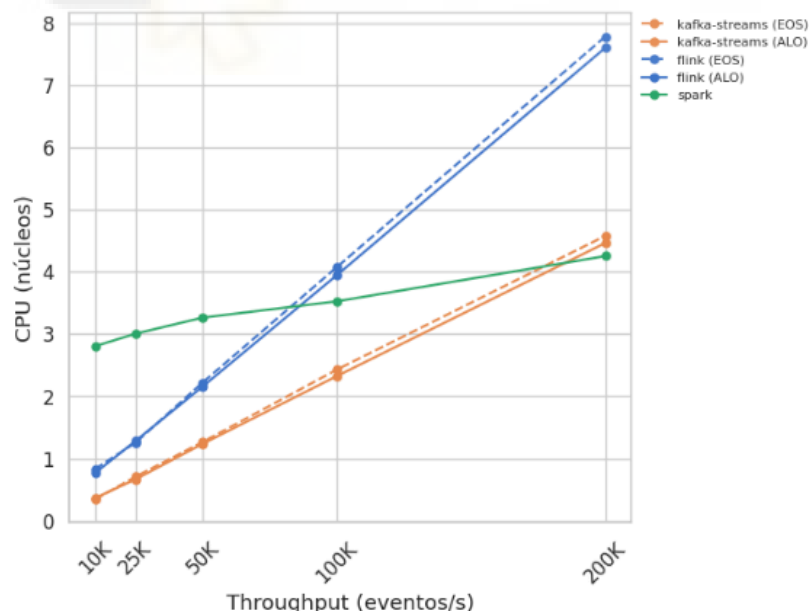


Figura 4.20: Uso de CPU en el job de agregado por ventanas frente al throughput

Este resultado es coherente con el análisis anterior: Flink sigue penalizado por la redistribución interna del flujo, mientras que Kafka Streams aprovecha la localidad del particionado de Kafka. Sin embargo, la diferencia específica de este job está en la salida.

El agregado por ventanas no emite un resultado por cada evento de entrada, sino un único resultado por clave y ventana al cierre de la misma. Por esta razón, aunque el coste de asignar eventos a claves y ventanas sigue creciendo con el throughput, se reduce la carga asociada a serializar y escribir resultados en el tópic de salida. Esto explica que Flink alcance 7,6 núcleos a 200.000 eventos/s, frente a los 8,4 núcleos observados en el agregado por clave.

El uso de memoria, representado en la Figura 4.21, mantiene el mismo orden que en los escenarios anteriores: Kafka Streams presenta el menor working set, Flink se sitúa en una posición intermedia y Spark mantiene el consumo más elevado. En este job, el estado depende no solo del número de claves, sino también de las ventanas que permanecen abiertas hasta que el watermark permite cerrarlas. Aun así, la memoria se mantiene prácticamente estable frente al throughput, lo que indica que, con una cardinalidad de claves y una configuración temporal constantes, aumentar los eventos por segundo no incrementa de forma proporcional la cantidad de estado vivo.

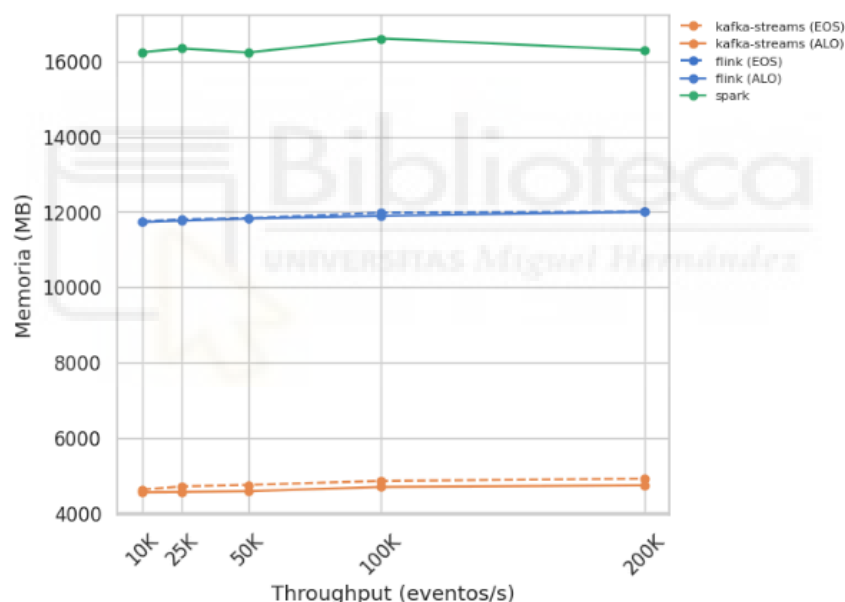


Figura 4.21: Uso de memoria en el job de agregado por ventanas frente al throughput

La Figura 4.22 muestra la escritura en disco del job de agregado por ventanas. Flink presenta un nivel alto y casi plano, en torno a 3,0 MB/s, lo que sugiere que la escritura está dominada por la cadencia de checkpoint y por el tamaño del estado mantenido. Spark se mantiene en una franja inferior a la observada en el agregado por clave, aproximadamente 1 MB/s menos, debido a que la agregación por ventanas reduce el número de actualizaciones emitidas y materializadas. Kafka Streams reproduce la separación entre modos de garantía: en *exactly-once* aparece una mayor escritura asociada a los commits transaccionales, mientras que en *at-least-once* el coste local observado es mucho menor.

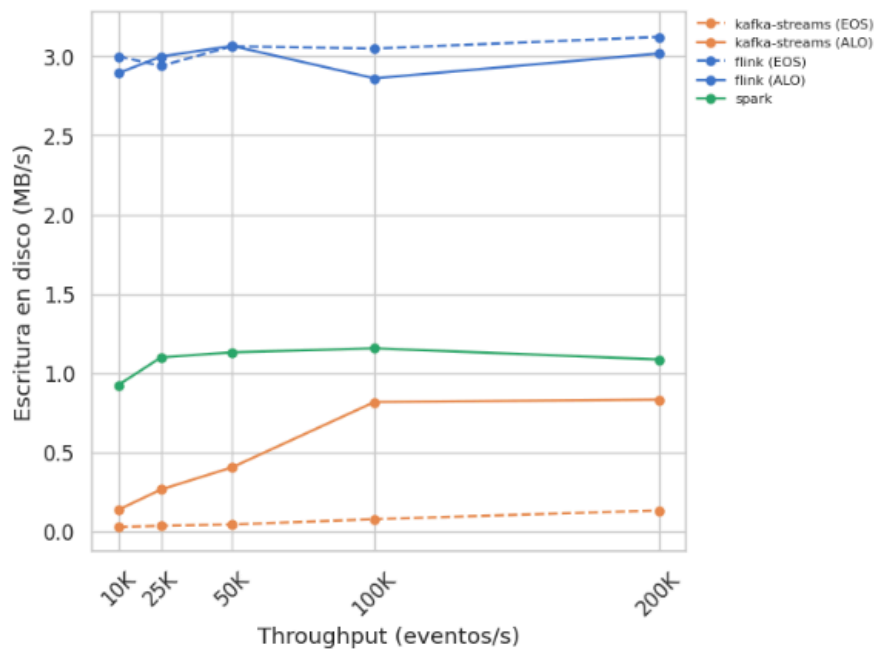


Figura 4.22: Escritura en disco en el job de agregado por ventanas frente al throughput

Este escenario confirma que las ventanas reducen la presión sobre la salida, pero no eliminan el coste de mantener el estado. La carga principal se desplaza hacia la asignación de eventos a claves y ventanas, el mantenimiento del estado temporal y la persistencia periódica de dicho estado. Por ello, Flink sigue siendo el motor más sensible al aumento de throughput, Kafka Streams mantiene un comportamiento estable al aprovechar el particionado previo de Kafka y Spark vuelve a mostrar un consumo de CPU competitivo a tasas altas, aunque condicionado por su modelo de ejecución por microlotes.

4.5.2.- Variación del tamaño de evento

Este escenario evalúa el impacto del tamaño del evento manteniendo constantes el throughput, el número de claves distintas y el job ejecutado. A diferencia del escenario anterior, el número de actualizaciones de estado por segundo no cambia, todos los motores siguen procesando 100.000 eventos/s y 5.000 claves distintas. Lo que aumenta es la cantidad de bytes que deben deserializarse, transportarse, almacenarse temporalmente y, en algunos casos, repartirse entre tareas. Por tanto, esta prueba permite distinguir qué motores son más sensibles al volumen de datos por evento y no únicamente al número de eventos procesados.

La Tabla 4.7 muestra que Kafka Streams es el motor menos afectado por el aumento del tamaño del evento. Su latencia media pasa de 20,38 ms con eventos de 128 B a 26,78 ms con eventos de 5.000 B, mientras que el percentil 99 aumenta de 83,62 ms a 174,33 ms. Aunque existe cierta degradación, esta es limitada en comparación con el incremento del tamaño del mensaje. La razón principal es que el aumento del payload no introduce una

nueva fase de repartición: el coste adicional se concentra en mover más bytes entre Kafka y la instancia de procesamiento, y en serializar y deserializar eventos de mayor tamaño.

Motor	Tamaño de evento (Bytes)	Media (ms)	Desviación estándar (ms)	p50 (ms)	p99 (ms)
Flink	128	42,01	25,64	92,01	104,14
	1.024	26,36	15,87	58,23	80,22
	5.000	3.292,73	5.285,68	15.275,66	18.136,17
Kafka Streams	128	20,38	19,95	79,5	83,62
	1.024	21,66	19,99	79,17	84,41
	5.000	26,78	32,66	62,91	174,33
Spark	128	848,07	291,59	1.304,43	1.356,86
	1.024	843,34	293,66	1.305,48	1.376,78
	5.000	964,86	295,82	1.426,06	1.501,56

Tabla 4.7: Latencia extremo a extremo ante la variación del tamaño de evento

Flink mantiene latencias bajas con eventos de 128 B y 1.024 B, con medias de 42,01 ms y 26,36 ms respectivamente. Sin embargo, con eventos de 5.000 B la latencia media aumenta hasta 3.292,73 ms y el percentil 99 alcanza 18.136,17 ms. Este salto debe interpretarse junto con el consumo de CPU de la Figura 4.23. En ese punto, Flink alcanza aproximadamente 8,9 núcleos, muy cerca de los nueve núcleos disponibles para el procesamiento. Por tanto, la degradación no refleja únicamente que cada evento sea más grande, sino que el sistema se aproxima a su límite de CPU al tener que mover y procesar muchos más bytes por evento.

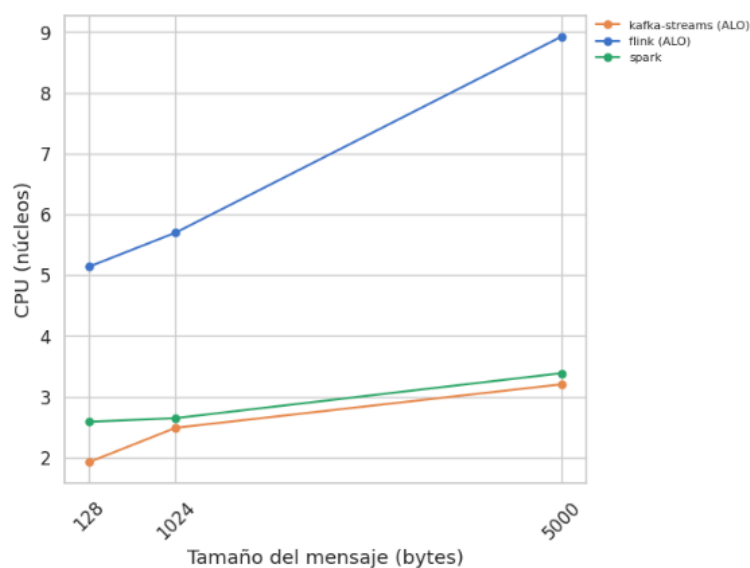


Figura 4.23: Uso de CPU en el job de agregado por clave frente al tamaño del evento

La diferencia respecto al escenario de throughput es importante. Allí el problema aparecía al aumentar el número de eventos por segundo; aquí aparece al mantener la misma tasa de eventos, pero multiplicar el tamaño de cada uno. En ambos casos, el efecto final es similar: aumenta el coste de serialización, transferencia y procesamiento interno antes de actualizar el estado. Esto explica que Flink sea especialmente sensible a eventos grandes en este job.

Spark mantiene una latencia elevada pero relativamente estable. Su media se sitúa en torno a 848 ms con eventos de 128 B, 843 ms con eventos de 1.024 B y 964,86 ms con eventos de 5.000 B. El aumento del tamaño del evento encarece el procesamiento, pero su efecto queda parcialmente oculto por la latencia base del microlote. La subida en el caso de 5.000 B indica que los lotes más pesados requieren más tiempo de procesamiento, aunque sin llegar a provocar una degradación comparable a la de Flink.

El uso de memoria, mostrado en la Figura 4.24, permanece en general estable, con un ligero incremento en Flink y Spark. Este comportamiento es coherente con la presencia de buffers y datos en tránsito de mayor tamaño. No obstante, la memoria no aparece como el recurso limitante principal: el número de claves se mantiene fijo y el estado agregado no crece con el payload. Lo que aumenta es el volumen de datos que atraviesa el sistema antes de ser reducido al agregado.

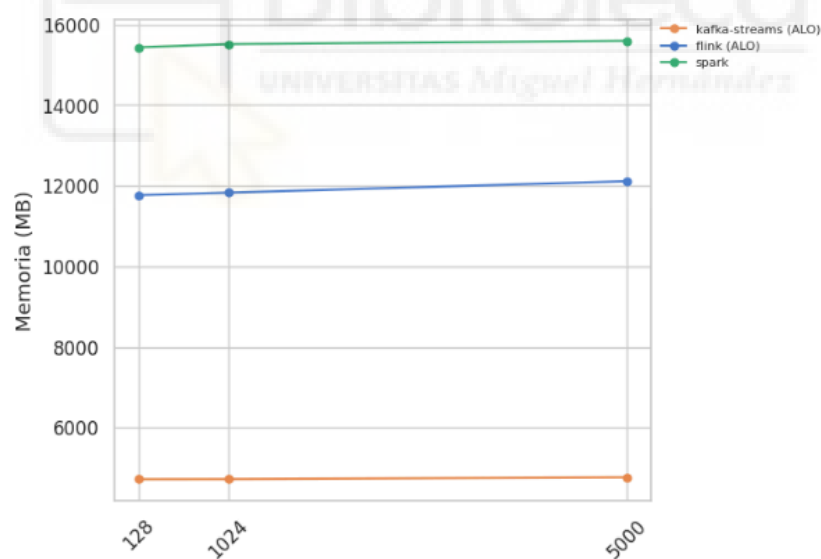


Figura 4.24: Uso de memoria en job de agregado por clave frente al tamaño de evento

La Figura 4.25 muestra que la escritura en disco tampoco responde igual en todos los motores. Flink se mantiene prácticamente plano, en torno a 2,6 MB/s, porque el estado persistido en los checkpoints corresponde al agregado por clave y no al evento completo. Kafka Streams muestra también una escritura local estable, alrededor de 0,7 MB/s. Spark, en cambio, incrementa su escritura de aproximadamente 0,57 MB/s a 1,28 MB/s, lo que indica que su shuffle y la materialización intermedia de los microlotes sí son más sensibles al tamaño de los datos de entrada.

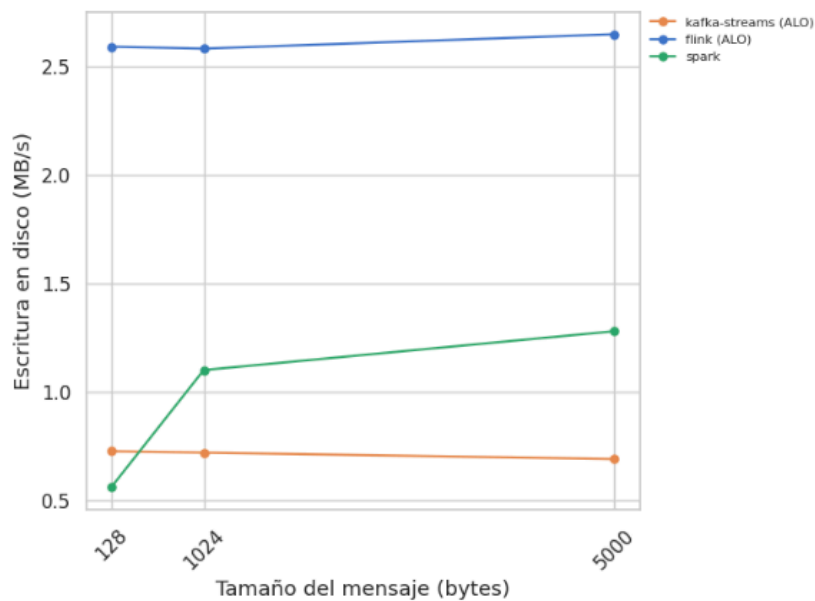


Figura 4.25: Escritura en disco en job de agregado por clave frente al tamaño de evento

Los resultados muestran que el tamaño del evento afecta principalmente al coste de transportar y procesar cada registro antes de la agregación, no al tamaño del estado final. Kafka Streams presenta el comportamiento más robusto, Flink conserva buen rendimiento con eventos pequeños y medianos pero se degrada con eventos grandes al aproximarse a la saturación de CPU, y Spark tolera razonablemente bien el aumento del tamaño de mensaje, aunque trasladando parte del coste adicional a disco.

4.5.3.- Variación del número de claves

Este escenario evalúa el impacto de aumentar el número de claves manteniendo constantes el throughput y el tamaño de los eventos. A diferencia de la prueba anterior, aquí no aumenta el tamaño de cada registro, sino la cardinalidad del estado que debe mantener el motor. Cada evento sigue teniendo el mismo tamaño y la tasa de entrada permanece fija, pero el sistema debe gestionar un número mayor de agregados independientes. Por tanto, esta prueba permite observar cómo responden los motores cuando crece el espacio de estado.

La Tabla 4.8 muestra que Kafka Streams mantiene una latencia prácticamente constante al aumentar el número de claves distintas. Su media permanece alrededor de 21 ms y el percentil 99 se mantiene próximo a 79 ms tanto con 1.000 como con 200.000 claves. Este comportamiento indica que, dentro del rango evaluado, el crecimiento de la cardinalidad no introduce una penalización significativa en la latencia observable. Aunque el estado es mayor, el acceso a las claves sigue siendo local a las particiones asignadas y no aparece una fase adicional de redistribución.

Motor	Número de claves distintas	Media (ms)	Desviación estándar (ms)	p50 (ms)	p99 (ms)
Flink	1.000	24,81	17,43	20,40	80,28
	10.000	23,89	14,43	20,48	76,94
	100.000	31,01	23,36	24,07	111,80
	200.000	36,77	31,66	25,97	173,80
Kafka Streams	1.000	20,82	28,54	14,79	79,36
	10.000	20,921	30,91	14,70	79,43
	100.000	21,10	19,18	15,62	79,97
	200.000	21,34	22,49	15,70	78,97
Spark	1.000	852,40	294,08	849,87	1.380,97
	10.000	851,50	294,13	851,97	1.387,27
	100.000	880,23	296,42	882,38	1.428,16
	200.000	945,14	300,53	943,72	1.519,39

Tabla 4.8: Latencia extremo a extremo ante la variación del número de claves distintas

Flink también mantiene latencias bajas en términos absolutos, pero muestra una degradación gradual conforme aumenta el número de claves. La media pasa de 24,81 ms con 1.000 claves a 36,77 ms con 200.000, mientras que el percentil 99 aumenta de 80,28 ms a 173,80 ms. Este incremento no representa un colapso como el observado en el escenario de 200.000 eventos/s, ya que no hay evidencia de backlog creciente, pero sí indica que el aumento del estado introduce más variabilidad y mayor coste por actualización. En este caso, el efecto no procede de procesar más eventos ni de mover eventos más grandes, sino de gestionar un espacio de claves más amplio en RocksDB.

Spark mantiene el patrón general observado en escenarios anteriores: su latencia está dominada por el modelo de microlotes. Con 1.000 y 10.000 claves, la media se sitúa en torno a 852 ms, mientras que con 200.000 claves aumenta hasta 945,14 ms. El incremento existe, pero queda parcialmente oculto por la latencia base del trigger. Por ello, el efecto del aumento de cardinalidad se observa con más claridad en CPU y disco que en la latencia extremo a extremo.

La Figura 4.26 confirma que el aumento del número de claves encarece la gestión del estado en los tres motores. Flink parte del consumo más alto y alcanza aproximadamente 7 núcleos con 200.000 claves. Kafka Streams se mantiene como el motor con menor consumo, creciendo de unos 2,3 a 3,05 núcleos. Spark muestra el crecimiento relativo más pronunciado, pasando aproximadamente de 2,4 a 4,85 núcleos. Esto sugiere que, aunque Spark amortiza bien el procesamiento cuando aumenta el throughput, el aumento de grupos

independientes dentro de cada microlote incrementa de forma clara el trabajo de agregación y mantenimiento de estado.

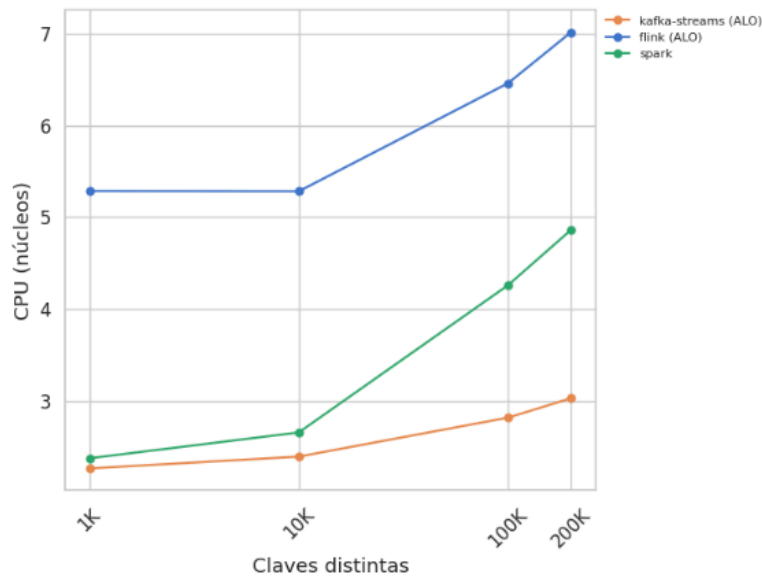


Figura 4.26: Uso de CPU en job de agregado por clave frente a número claves distintas

La Figura 4.27 muestra una diferencia relevante en memoria. En Flink, el working set crece de aproximadamente 11,6 GB con 1.000 claves a 16,8 GB con 200.000 claves. Este resultado es coherente con un mayor uso de memoria por parte del backend de estado y de las estructuras asociadas a RocksDB. En Kafka Streams y Spark, en cambio, el working set se mantiene mucho más estable, alrededor de 5 GB y 15,5 GB respectivamente. Esta estabilidad no implica que el estado no crezca, sino que el crecimiento no se refleja de forma proporcional en la memoria observada.

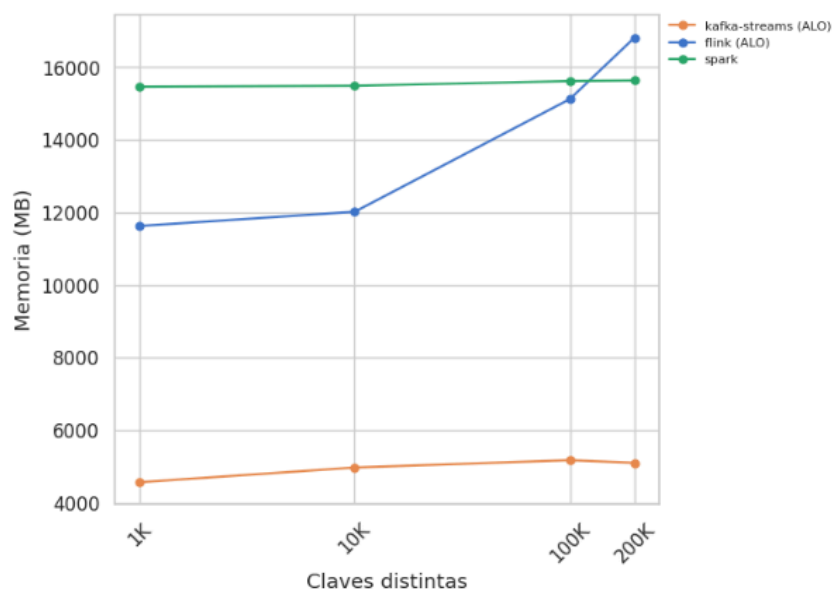


Figura 4.27: Uso de memoria vs. claves distintas en job de agregado por clave

La escritura en disco, representada en la Figura 4.28, completa la interpretación. En los tres motores la escritura aumenta con el número de claves, lo que confirma que el crecimiento del estado tiene impacto directo sobre la persistencia. Flink es el motor que más escribe, pasando aproximadamente de 1,8 MB/s a 18 MB/s. Kafka Streams queda en una posición intermedia, de 0,3 MB/s a 12,6 MB/s, mientras que Spark aumenta de 1,1 MB/s a 7,5 MB/s. En este escenario, el coste principal deja de estar únicamente en procesar eventos y pasa a estar también en mantener, volcar y compactar un estado más grande.

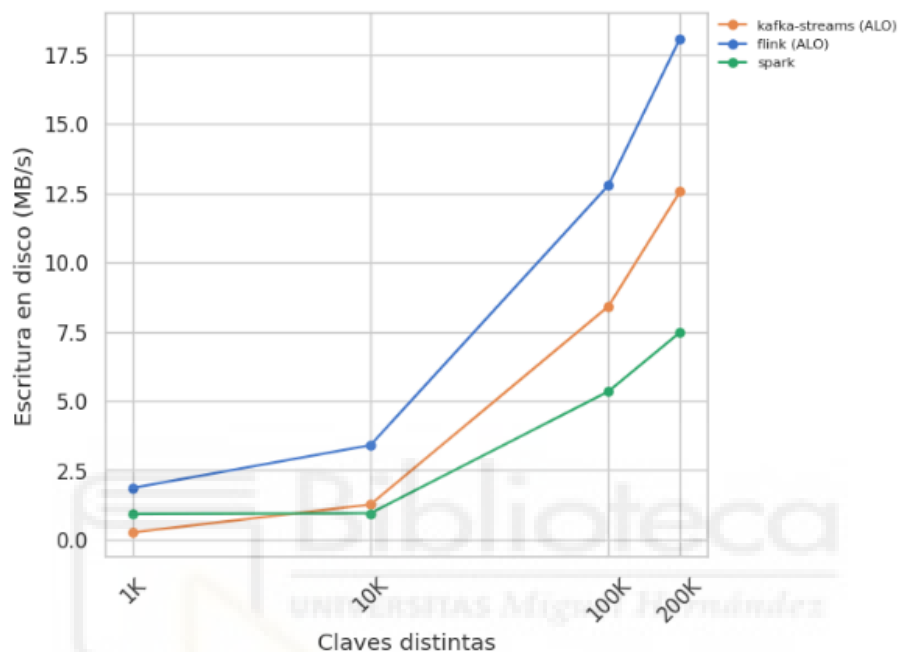


Figura 4.28: Escritura en disco vs. claves distintas en job de agregado por clave

La lectura conjunta de memoria y disco permite distinguir cómo absorbe cada motor el crecimiento del estado. Flink refleja el aumento de cardinalidad tanto en memoria como en escritura a disco, lo que explica el incremento gradual de latencia y CPU. Kafka Streams mantiene una memoria más acotada y una latencia casi plana, pero desplaza parte del coste hacia la escritura asociada al estado y al changelog en Kafka. Spark mantiene un working set estable, pero aumenta su CPU y su escritura a disco conforme crece el número de claves procesadas dentro de cada microlote.

En definitiva, el número de claves distintas afecta a los tres motores. Kafka Streams es el más estable en latencia y CPU dentro del rango evaluado. Flink mantiene latencias bajas, pero es el más sensible al crecimiento del estado en términos de memoria y escritura en disco. Spark sigue condicionado por su modelo de microlotes: su latencia base es alta, pero el aumento de claves se aprecia sobre todo en CPU y en escritura. Este escenario muestra que, cuando la presión principal procede de la cardinalidad del estado, las métricas de memoria y disco pasan a ser tan importantes como la latencia.

Capítulo 5

Conclusiones y trabajo futuro

5.1.- CONCLUSIONES

Este trabajo ha presentado una comparativa técnica entre Apache Flink, Kafka Streams y Spark Structured Streaming desde una doble perspectiva: el estudio de sus modelos de ejecución y la validación práctica mediante un sistema experimental común. A lo largo del proyecto se han analizado sus arquitecturas internas, sus mecanismos de paralelización, sus garantías de entrega, la gestión del estado y las implicaciones que todo ello tiene sobre la latencia y el consumo de recursos.

Para realizar la comparación se ha desarrollado un sistema de pruebas desplegado sobre Kubernetes, capaz de ejecutar las tres tecnologías y recoger métricas comparables. Los experimentos se han diseñado variando parámetros como el throughput, el tamaño de los eventos y el número de claves distintas, lo que ha permitido estudiar el comportamiento de

los motores ante distintos tipos de presión y relacionar los resultados obtenidos con sus modelos de ejecución.

Los resultados obtenidos muestran que no existe una herramienta superior en todos los casos. La elección del motor depende del tipo de procesamiento, de los requisitos de latencia, de la semántica de entrega, del volumen de datos, del número de claves distintas y de la complejidad operativa que se esté dispuesto a asumir.

Apache Flink destaca por su baja latencia y por su modelo de procesamiento continuo por evento. Es una herramienta adecuada para aplicaciones de streaming avanzado, especialmente cuando se requiere estado complejo, ventanas temporales y control preciso del procesamiento. A cambio, exige una configuración y operación más cuidadosas.

Kafka Streams presenta un comportamiento muy estable en aplicaciones centradas en Kafka. Su principal ventaja es su integración directa con este ecosistema, lo que simplifica el despliegue y facilita el desarrollo de transformaciones y agregaciones por clave. Es una opción especialmente interesante cuando se busca una solución ligera y cercana a los tópicos Kafka.

Spark Streaming queda condicionado por su modelo de microlotes, que introduce una latencia base superior. Sin embargo, se integra de forma natural con el ecosistema Spark y resulta adecuado cuando el procesamiento en streaming se combina con tareas por lotes, analítica de datos o pipelines ya contruidos sobre Spark.

Como resultado final, puede establecerse una clasificación práctica por caso de uso. Kafka Streams encaja especialmente bien en aplicaciones integradas con Kafka y agregaciones por clave. Flink es más adecuado para procesamiento de streaming avanzado, baja latencia, estado complejo y semántica temporal sofisticada. Spark Structured Streaming resulta conveniente en entornos donde ya se utiliza Spark y donde la integración con procesos batch o analíticos compensa una mayor latencia base.

5.2.- POSIBLES DESARROLLOS FUTUROS

Aunque el proyecto ha permitido comparar Apache Flink, Kafka Streams y Spark Streaming bajo varios escenarios representativos, existen distintos desarrollos futuros que permitirían ampliar el alcance del estudio y obtener una visión todavía más completa de estas tecnologías.

Una primera mejora sería incorporar nuevos tipos de job, especialmente joins entre data streams. Este tipo de procesamiento permitiría evaluar escenarios con mayor complejidad de estado y sincronización temporal.

Otra posible opción sería evaluar la tolerancia a fallos de forma experimental. Para ello, podrían introducirse fallos controlados, como reinicios de pods, caídas de workers, interrupciones durante checkpoints o reinicios del broker de Kafka, midiendo el tiempo de recuperación y el impacto sobre la latencia.

Además, se podrían estudiar configuraciones optimizadas para cada motor. En este proyecto se ha priorizado una configuración común y comparable, pero un trabajo posterior podría ajustar parámetros específicos de Flink, Kafka Streams y Spark para analizar el rendimiento máximo alcanzable por cada herramienta.

Otra comparativa sería probar distribuciones de claves no uniformes. Escenarios con hot keys, distribuciones Zipf o cargas sesgadas permitirían analizar cómo responde cada motor ante desequilibrios entre particiones y tareas.

Por último, podría añadirse un análisis económico del rendimiento, relacionando los resultados obtenidos con el coste de infraestructura en la nube. Esto permitiría comparar no solo el rendimiento técnico, sino también el coste por volumen de eventos procesados.



Bibliografía

- [1] Big Data: Principles and best practices of scalable realtime data systems
J. Warren, N. Marz
Mayo 2015

- [2] Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems
M. Kleppmann
Marzo 2017

- [3] The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing
T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle
Agosto 2015

- [4] Models and Issues in Data Stream Systems
B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom
Junio 2002
- [5] The 8 Requirements of Real-Time Stream Processing
M. Stonebraker, U. Çetintemel, S. Zdonik
2005
- [6] Apache Flink: Stream and Batch Processing in a Single Engine
P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas
2015
- [7] Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing
T. Akidau, S. Chernyak, R. Lax
Agosto 2018
- [8] Event Streams in Action: Real-time event systems with Kafka and Kinesis
A. Dean, V. Crettaz
Mayo 2019
- [9] Kafka Streams in Action: Real-time apps and microservices with the Kafka Streams API
B. Bejeck
Septiembre 2018
- [10] Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale
G. Shapira, T. Palino, R. Sivaram, K. Petty
Diciembre 2021
- [11] Making Sense of Stream Processing
M. Kleppmann
Mayo 2016
- [12] Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications
F. Hueske, V. Kalavri
Abril 2019
- [13] Mastering Kafka Streams and ksqlDB: Building real-time data systems by example
M. Seymour
Marzo 2021

- [14] Windows, Documentación de Apache Flink
<https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream/operators/windows/>
Marzo 2026
- [15] Building Blocks, Documentación de Apache Flink
https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/datastream-v2/building_blocks/
Marzo 2026
- [16] RocksDB
<https://rocksdb.org/>
Marzo 2026
- [16] Designing Event-Driven Systems
Ben Stopford
Abril 2018
- [17] Kafka Message Delivery Guarantees
<https://docs.confluent.io/kafka/design/delivery-semantics.html>
Marzo 2026
- [18] What is Change Data Capture?
<https://www.confluent.io/learn/change-data-capture/#why-change-data-capture>
Marzo 2026
- [19] What is Data Ingestion?
<https://aws.amazon.com/what-is/data-ingestion/>
Marzo 2026
- [20] Intro to Messaging Systems
<https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingComponentsIntro.html>
Marzo 2026
- [21] Event-Driven Architectures - The Queue vs The Log
<https://jack-vanlightly.com/blog/2018/5/20/event-driven-architectures-the-queue-vs-the-log>
Marzo 2026
- [22] What is a Message Queue?
<https://aws.amazon.com/message-queue/>
Abril 2026

- [23] What is a Message Queue?
<https://www.ibm.com/think/topics/message-queues>
Abril 2026
- [24] I Heart Logs: Event Data, Stream Processing, and Data Integration
J. Kreps
Septiembre 2014
- [25] What is Apache Kafka?
<https://www.confluent.io/what-is-apache-kafka/>
Abril 2026
- [26] Kafka Topics - A Deep Dive
<https://www.redpanda.com/guides/kafka-architecture-kafka-topics>
Abril 2026
- [27] Amazon Kinesis Documentation
<https://docs.aws.amazon.com/kinesis/>
Abril 2026
- [28] Amazon Kinesis Quotas and Limits
<https://docs.aws.amazon.com/streams/latest/dev/service-sizes-and-limits.html>
Abril 2026
- [29] Flink Architecture
<https://nightlies.apache.org/flink/flink-docs-stable/docs/concepts/flink-architecture/>
Mayo 2026
- [30] State Backends
https://nightlies.apache.org/flink/flink-docs-release-2.2/docs/ops/state/state_backends/
Mayo 2026
- [31] Cluster Mode Overview
<https://spark.apache.org/docs/latest/cluster-overview.html>
Mayo 2026
- [32] Spark Streaming Programming Guide
<https://spark.apache.org/docs/latest/streaming-programming-guide.html>
Mayo 2026

- [33] RDD Programming Guide
<https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds>
Mayo 2026
- [34] Structured Streaming Programming Guide
<https://spark.apache.org/docs/latest/streaming/index.html>
Mayo 2026
- [35] A Quick Introduction to Kafka Streams
<https://bigdata.2minutestreaming.com/p/what-is-kafka-streams-api-guide>
Mayo 2026
- [36] Kafka Streams Architecture for Confluent Platform
<https://docs.confluent.io/platform/current/streams/architecture.html>
Mayo 2026
- [37] Kafka Streams Internals
<https://developer.confluent.io/courses/kafka-streams/internals/>
Mayo 2026
- [38] Configuring a Streams Application
<https://kafka.apache.org/42/streams/developer-guide/config-streams/>
Mayo 2026
- [39] Fault Tolerance Guarantees of Data Sources and Sinks
<https://nightlies.apache.org/flink/flink-docs-release-2.2/docs/connectors/datastream/guarantees/>
Mayo 2026
- [40] Protocol Buffers Overview
<https://protobuf.dev/overview/>
Mayo 2026
- [41] Protocol Buffers Language Guide
<https://protobuf.dev/programming-guides/editions/>
Mayo 2026
- [42] The Go Programming Language Specification
<https://go.dev/ref/spec>
Mayo 2026

- [43] Go Use Cases
<https://go.dev/solutions/use-cases>
Mayo 2026
- [43] Go Use Cases
<https://go.dev/solutions/use-cases>
Mayo 2026
- [44] Applications for Python
<https://www.python.org/about/apps/>
Mayo 2026
- [45] Python for Data Analysis: Key Stats and Trends
<https://www.deeplearning.ai/blog/python-for-data-analysis-key-stats-and-trends>
Mayo 2026
- [46] What is Java?
<https://www.ibm.com/think/topics/java>
Mayo 2026
- [47] Cloud Native Foundation Projects - Prometheus
<https://www.cncf.io/projects/prometheus/>
Mayo 2026
- [48] What is Prometheus?
<https://prometheus.io/docs/introduction/overview/>
Mayo 2026
- [49] Prometheus Querying basics
<https://prometheus.io/docs/prometheus/latest/querying/basics/>
Mayo 2026
- [50] Java Management Extensions (JMX)
<https://docs.oracle.com/javase/tutorial/jmx/index.html>
Mayo 2026
- [51] JMX Exporter
https://prometheus.github.io/jmx_exporter/
Mayo 2026

- [52] cAdvisor
<https://github.com/google/cadvisor>
Mayo 2026
- [53] About Grafana
<https://grafana.com/docs/grafana/latest/introduction/>
Mayo 2026
- [54] Grafana Visualize Data
<https://grafana.com/docs/grafana/latest/visualizations/>
Mayo 2026
- [55] A Kafka web UI for developers
<https://www.redpanda.com/data-streaming/redpanda-console-kafka-ui>
Mayo 2026
- [56] franz-go
<https://github.com/twmb/franz-go>
Mayo 2026
- [57] Seaborn: statistical data visualization
<https://seaborn.pydata.org/>
Mayo 2026
- [58] Pandas: Getting started
https://pandas.pydata.org/docs/getting_started/index.html
Mayo 2026
- [59] What is Kubernetes?
<https://www.redhat.com/en/topics/containers/what-is-kubernetes>
Mayo 2026
- [60] Top 30 Open Source Projects
<https://insights.linuxfoundation.org/collection/details/top-open-source-projects>
Mayo 2026
- [61] Pods
<https://kubernetes.io/docs/concepts/workloads/pods/>
Mayo 2026
- [62] GKE overview

<https://docs.cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview>
Mayo 2026

[63] What is Docker?
<https://aws.amazon.com/docker/>
Mayo 2026

[64] What is a Container?
<https://www.docker.com/resources/what-container/>
Mayo 2026

[65] Why Docker?
<https://www.docker.com/why-docker/>
Mayo 2026

[66] kind
<https://github.com/kubernetes-sigs/kind>
Mayo 2026

[67] Documentación de kind
<https://kind.sigs.k8s.io/>
Mayo 2026

[68] Apache Maven Project
<https://maven.apache.org/>
Mayo 2026

[69] Introduction to the Build Lifecycle
<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>
Mayo 2026

[70] Visual Studio Code
<https://code.visualstudio.com/>
Mayo 2026

[71] Claude Code Overview
<https://code.claude.com/docs/en/overview>
Mayo 2026

[72] Beyond working set memory: understanding the cAdvisor memory metrics

<https://sourcegraph.com/blog/beyond-working-set-memory-understanding-the-cadvisor-memory-metrics>

Junio 2026

- [73] How to emit only the final result of a windowed aggregation in Kafka Streams

<https://developer.confluent.io/confluent-tutorials/window-final-result/kstreams/>

Junio 2026

- [74] Token Bucket Algorithm

<https://www.geeksforgeeks.org/computer-networks/token-bucket-algorithm/>

Junio 2026

- [75] HdrHistogram: A High Dynamic Range Histogram

<https://hdrhistogram.github.io/HdrHistogram/>

Junio 2026

- [76] Introduction to PromQL, the Prometheus query language

<https://grafana.com/blog/introduction-to-promql-the-prometheus-query-language/>

Junio 2026

- [77] Strimzi Overview

<https://strimzi.io/docs/operators/latest/overview>

Junio 2026

- [78] kube-prometheus-stack

<https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack>

Junio 2026

- [79] Kustomize

<https://kustomize.io/>

Junio 2026

- [80] GCP Region Pricing

<https://cloudprice.net/gcp/regions>

Junio 2026

- [81] Spot VMs

<https://cloud.google.com/solutions/spot-vms>

Junio 2026

- [82] Noisy Neighbor - SaaS Lens

<https://docs.aws.amazon.com/wellarchitected/latest/saas-lens/noisy-neighbor.html>
Junio 2026



Anexo

Instalación y Configuración

A.1.- SERVICIOS EN GO

A.1.1.- Instalación de Go

La versión utilizada es Go 1.26 y los comandos para la instalación son los indicados en la Figura A.1.

```
curl -fsSL0 https://go.dev/dl/go1.26.2.linux-amd64.tar.gz  
sudo rm -rf /usr/local/go  
sudo tar -C /usr/local -xzf go1.26.2.linux-amd64.tar.gz
```

Figura A.1: Instalación de Golang

A.1.2.- Instalación de protoc y protoc-gen-go

Tanto productor y consumidor intercambian eventos serializados con Protocol Buffers. Los esquemas se definen una sola vez en el directorio `proto/` de la raíz del proyecto (`InputEvent.proto`, `FilteredEvent.proto`, `KeyedAggregate.proto`) y, a partir de ellos, se genera el código Go correspondiente. Esta generación hace uso de dos herramientas: `protoc`, el compilador de Protocol Buffers, que analiza los ficheros `.proto` y `protoc-gen-go`, el complemento que dota a `protoc` de la capacidad de emitir el código en Go. Los comandos para la instalación se describen en la Figura A.2.

```
sudo apt update
sudo apt install -y protobuf-compiler
go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.36.11
```

Figura A.2: Instalación de herramientas de Protobuf

A.1.3.- Compilación

Para generar el código de Go a partir del esquema del archivo `.proto` se ejecuta el siguiente comando: `protoc --go_out=proto --go_opt=paths=source_relative -I ../proto ../proto/InputEvent.proto`. El resultado será un fichero Go con la estructura correspondiente al evento.

Para compilar el proyecto de Go, se ejecuta el comando `go build -o <servicio> .`, que generará el archivo binario necesario.

A.2.- JOBS EN JAVA

A.2.1.- Instalación de Java

La versión requerida del lenguaje viene determinada por el propio proyecto: todos los módulos fijan tanto el origen como el destino de compilación en la versión 21 (`maven.compiler.source` y `maven.compiler.target` con valor 21), por lo que se requiere un JDK 21 o superior. Es preciso instalar el JDK (Java Development Kit) completo, y no únicamente el entorno de ejecución (JRE), puesto que la compilación necesita el compilador `javac` que solo el primero incluye. Los comandos para la instalación se describen en la Figura A.3.

```
sudo apt update  
sudo apt install -y openjdk-21-jdk
```

Figura A.3: Instalación de Java

A.2.2.- Instalación de Maven

El proyecto emplea Apache Maven como herramienta de construcción y gestión de dependencias. Maven se encarga de descargar las dependencias declaradas en cada proyecto, compilar el código, ejecutar las pruebas unitarias y empaquetar los fat-JAR. Se instala igualmente desde el gestor de paquetes mediante el comando *sudo apt install -y maven*.

A.2.3.- Compilación de Jobs

Cada job es un proyecto Maven que, en la fase de empaquetado, produce un fat-JAR: un único archivo que contiene tanto el código del job como todas sus dependencias. Esta consolidación se realiza mediante el complemento *maven-shade-plugin*, que además fija en el manifiesto del archivo la clase principal del job (por ejemplo, *dev.miguel.benchmark.KeyedAggregator*) y excluye metadatos de firma y descriptores de módulo que provocarían conflictos al fusionar dependencias. El resultado es un artefacto autoejecutable, requisito necesario para que la imagen de contenedor pueda lanzarlo sin resolver dependencias en tiempo de ejecución.

Las dependencias propias de cada motor se declaran con alcance *provided*, ya que las aporta el entorno de ejecución del motor y, por tanto, no deben incluirse en el fat-JAR. En el caso de Kafka Streams, al tratarse de una librería embebida y no de un motor externo, sus dependencias sí se empaquetan dentro del artefacto.

Los jobs comparten los mismos esquemas .proto al igual que los servicios en Go. La generación del código en Java correspondiente se integra en la propia construcción de Maven mediante el plugin *protobuf-maven-plugin*. La compilación se realiza con Maven mediante el comando *mvn clean package* para cada proyecto.

A.3.- CONTENEDORIZACIÓN

Todos los componentes del sistema se ejecutan en el clúster como contenedores. La contenedorización garantiza que cada componente se despliega con un entorno de ejecución idéntico e inmutable, con independencia de la máquina anfitriona, y permitir que

el mismo contenedor se utilice sin modificaciones tanto en el clúster de desarrollo local como en el clúster gestionado en la nube.

A.3.1.- Instalación de Docker

La construcción y ejecución de imágenes se apoya en Docker. La Figura A.4 describe los comandos necesarios para instalarlo en Ubuntu.

```
sudo apt update
sudo apt install -y ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
    sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo "deb [arch=$(dpkg --print-architecture)
signed-by=/etc/apt/keyrings/docker.gpg] \
    https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo
$VERSION_CODENAME) stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io
docker-buildx-plugin
```

Figura A.4: Instalación de Docker

Por defecto, la interacción con Docker requiere de privilegios de administrador. Para permitir su uso sin sudo, se añade la cuenta de usuario al grupo docker mediante el comando *sudo usermod -aG docker \$USER*.

A.3.2.- Construcción de imágenes

Las imágenes de los servicios en Go se construyen mediante una definición multietapa. El Dockerfile representado en la Figura A.5, correspondiente al productor, ilustra el patrón común a los tres servicios.

```
FROM golang:1.26-alpine AS builder
WORKDIR /src
COPY go.mod go.sum ./
COPY . .
RUN CGO_ENABLED=0 go build -o producer .

FROM alpine:3.21
COPY --from=builder /src/producer /producer
ENTRYPOINT ["/producer"]
```

Figura A.5: Dockerfile del productor

La primera etapa (builder) parte de la imagen oficial de Go y compila el código fuente. La directiva `CGO_ENABLED=0` produce un binario enlazado estáticamente, sin dependencias de bibliotecas del sistema, lo que permite que la segunda etapa parte de una imagen base mínima (alpine:3.2.1) en la que únicamente se copia el ejecutable resultante. El empleo de esta técnica reduce el tamaño de la imagen final.

Por otro lado, las imágenes de los motores de procesamiento incorporan los archivos `.jar` previamente compilados con Maven. La construcción de estas imágenes se divide en dos casos dependiendo de si el motor ya tiene una imagen base, el caso de Spark y Flink, y el caso de Kafka Streams en el que es una librería.

En el caso de que el motor disponga de una imagen, se utiliza esta y se incorporan los `.jar` de los jobs en el directorio que cada motor reserva para el código del usuario. La Figura A.6 muestra el Dockerfile de Flink.

```
FROM apache/flink:2.2.1-scala_2.12-java21

RUN curl -fsSL \
https://repo1.maven.org/maven2/org/apache/flink/flink-statebacke
nd-rocksdb/2.2.1/flink-statebackend-rocksdb-2.2.1.jar \
-o /opt/flink/lib/flink-statebackend-rocksdb-2.2.1.jar

COPY stateless-filter/target/stateless-filter-1.0.jar
/opt/flink/usrlib/stateless-filter.jar

COPY keyed-aggregator/target/keyed-aggregator-1.0.jar
/opt/flink/usrlib/keyed-aggregator.jar

COPY window-aggregator/target/window-aggregator-1.0.jar
/opt/flink/usrlib/window-aggregator.jar
```

Figura A.6: Dockerfile de Flink

La imagen no define un punto de entrada propio: el control de la ejecución corresponde al motor, que localiza el trabajo en el directorio y lo lanza.

En el caso de Kafka Streams, al ser una librería de Java que se enlaza dentro de la propia aplicación no existe ningún servicio externo al que enviar el job ni una imagen oficial. La aplicación es un proceso de Java autónomo que incorpora la librería, por lo que su imagen parte de una distribución genérica del entorno de ejecución de Java y ejecuta directamente el fat-JAR. La selección del job se realiza mediante variables de entorno. Este dockerfile se observa en la Figura A.7.

```

FROM eclipse-temurin:21-jre-jammy

WORKDIR /app
COPY stateless-filter/target/stateless-filter-1.0.jar
/app/lib/stateless-filter.jar
COPY keyed-aggregator/target/keyed-aggregator-1.0.jar
/app/lib/keyed-aggregator.jar
COPY window-aggregator/target/window-aggregator-1.0.jar
/app/lib/window-aggregator.jar
COPY jmx_config.yml jmx_config.yml
RUN mkdir -p /opt/jmx && curl -fsSL \

https://github.com/prometheus/jmx_exporter/releases/download/1.5
.0/jmx_prometheus_javaagent-1.5.0.jar \
    -o /opt/jmx/jmx_prometheus_javaagent-1.5.0.jar

ENV JOB=stateless-filter
ENTRYPOINT ["sh", "-c", "exec java -Xms2g -Xmx6g
-javaagent:/opt/jmx/jmx_prometheus_javaagent-1.5.0.jar=9249:jmx_
config.yml -jar /app/lib/${JOB}.jar"]

```

Figura A.7: Dockerfile de Kafka Streams

Para construir las imágenes se hace uso del comando *docker build*. Donde se indica el tag de la imagen para identificarlo y versionar y la ruta del directorio donde se encuentra el Dockerfile de la imagen a construir. Un ejemplo para construir la imagen de Spark sería así: *docker build -t spark:local ./frameworks/spark*

Una vez construidas todas las imágenes, se puede comprobar que existen mediante el comando *docker image ls*, el resultado se muestra en la Figura A.8.

```
→ tfg git:(development) X docker image ls
```

IMAGE	ID	DISK USAGE	CONTENT SIZE	EXTRA
benchmark-consumer:local	f67c33fcb979	40.8MB	13.2MB	
benchmark-producer:local	2ea50f577012	45.7MB	14.9MB	
benchmark-scrape-metrics:local	efb55246336b	26.4MB	8.74MB	
flink:local	9c9d374ff641	1.74GB	741MB	
kafka-streams:local	eb084afe237d	1.01GB	401MB	
spark:local	d861b1d8b65b	2.14GB	888MB	

Figura A.8: Listado de imágenes de Docker

A.4.- ENTORNO LOCAL CON KIND

A.4.1.- Instalación de Kind y Kubectl

La interacción con el clúster requiere dos herramientas: Kind, para crear y destruir el clúster, y kubectl, el cliente de línea de comandos de Kubernetes empleado para desplegar y consultar los recursos.

Kind se distribuye como un binario. Su instalación consiste en ejecutar el siguiente comando: `go install sigs.k8s.io/kind@v0.31.0`.

Para instalar kubectl tendremos que descargar el ejecutable, otorgarle permisos de ejecución e instalarlo en una ruta incluida en el PATH, tal y como se muestra en la Figura A.9.

```
curl -fsSL "https://dl.k8s.io/release/$(curl -fsSL
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
chmod +x kubectl
sudo mv kubectl /usr/local/bin/kubectl
```

Figura A.9: Instalación de kubectl

A.4.2.- Creación del clúster

Para crear el clúster utilizaremos el fichero `kind-config.yaml`, que especifica un nodo para el plano de control y un nodo de trabajo. Ejecutaremos el comando `kind create cluster -config kind-config.yaml -name bench`.

Al crear el clúster, Kind configura automáticamente un contexto de kubectl denominado `kind-bench` y lo establece como contexto activo, de manera que las órdenes posteriores se dirigen a este clúster. Una vez creado, se declara el espacio de nombres `bench`, en el que se desplegarán todos los componentes del banco de pruebas: `kubectl apply -f namespaces.yaml`. En la Figura A.10 se ve el listado de cluster y nodos una vez creados.

```
→ tfg git:(development) ✕ kind get clusters
bench
→ tfg git:(development) ✕ kind get nodes -n bench
bench-control-plane
bench-worker
```

Figura A.10: Listado de clúster y nodos de Kind

La eliminación del clúster una vez concluidas las pruebas se realiza con: *kind delete cluster -name bench*.

A.4.3.- Despliegue de imágenes

Existe una diferencia esencial entre un clúster de Kind y un clúster convencional en lo relativo al origen de las imágenes. En un despliegue habitual, los nodos extraen las imágenes de un registro remoto. Los nodos de Kind, en cambio, son contenedores aislados que no tienen acceso al almacén local de Docker de la máquina anfitriona donde se han construido las imágenes.

La solución que adopta Kind es inyectar las imágenes directamente en el sistema de archivos de los nodos, sin intermediación de ningún registro. Esta operación se realiza con el comando *kind load docker-image*, que copia una imagen del almacén local de Docker a los nodos del clúster indicado. La figura A.11 indica los comandos para inyectar las imágenes.

```
kind load docker-image benchmark-producer:local --name bench
kind load docker-image benchmark-consumer:local --name bench
kind load docker-image benchmark-scrape-metrics:local --name
bench
kind load docker-image kafka-streams:local --name bench
kind load docker-image flink:local --name bench
kind load docker-image spark:local --name bench
```

Figura A.11: Inyección de imágenes en Kind

Esta particularidad condiciona la configuración de los manifiestos de despliegue. Puesto que las imágenes ya residen en los nodos y no deben buscarse en ningún registro externo, la política de extracción de imágenes se fija en Never (*imagePullPolicy: Never*) para el entorno local.

A.4.4.- Instalación de Helm

Helm es el gestor de paquetes de Kubernetes. Permite instalar aplicaciones complejas, denominadas chart, como una unidad parametrizable, encargándose de generar y aplicar el conjunto de Kubernetes que las componen a partir de un único fichero de valores de configuración. La instalación de Helm se realiza mediante el siguiente comando: *curl -fsSL https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash*.

A.4.5.- Instalación de operadores

Tanto Kafka, como Flink y Spark se gestionan mediante el patrón operador de Kubernetes: en lugar de desplegar Kafka, Flink o Spark con manifiestos manuales, se instala un operador que extiende la API de Kubernetes con recursos personalizados (CRDs) propios y se encarga de traducir esos recursos de alto nivel en los pods, servicios y configuraciones necesarios, además de gestionar su ciclo de vida.

Strimzi es el operador que despliega y gestiona Kafka. Se instala en su propio namespace (strimzi-system) y se le indica, mediante *watchNamespaces*, el namespace en el que vigilará los recursos de Kafka (bench). La Figura A.12 indica los comandos para la instalación.

```
helm repo add strimzi https://strimzi.io/charts/  
helm repo update  
  
helm upgrade --install strimzi-kafka-operator  
strimzi/strimzi-kafka-operator \  
-n strimzi-system --create-namespace \  
--set "watchNamespaces={bench}" \  
--set "resources.limits.memory=768Mi" \  
--set "resources.requests.memory=768Mi" \  
--wait
```

Figura A.12: Instalación de Strimzi

El operador de Flink se distribuye desde el repositorio de charts del propio Apache, cuya URL incluye el número de versión del operador (en este proyecto, la 1.14.0). Se instala en el namespace flink-system y se parametriza con un fichero de valores como se ve en la Figura A.13, además de aplicarle los permisos (RBAC) que autorizan a los jobs de Flink a operar en el namespace correspondiente.

```
helm repo add flink-operator-repo \  
https://archive.apache.org/dist/flink/flink-kubernetes-operator-1.14.0/  
helm repo update  
helm upgrade --install flink-kubernetes-operator  
flink-operator-repo/flink-kubernetes-operator \  
-n flink-system --create-namespace \  
-f infra/k8s/platform/flink/operator-values.yaml --wait --timeout=10m  
kubectl apply -f infra/k8s/platform/flink/rbac.yaml
```

Figura A.13: Instalación del operador de Flink

Por otro lado, el operador de Spark, mediante *spark.jobNamespaces* se le indica en qué namespace atenderá los recursos *SparkApplication* y se le aplicarán los permisos tal y como se indica en la Figura A.14.

```
helm repo add spark-operator
https://kubeflow.github.io/spark-operator
helm repo update

helm upgrade --install spark-operator
spark-operator/spark-operator \
  -n spark-system --create-namespace \
  --set "spark.jobNamespaces={bench}" --wait --timeout=10m

kubectl apply -f infra/k8s/platform/spark/rbac.yaml
```

Figura A.14: Instalación del operador de Spark

Por último, la recogida de métricas se apoya en el chart kube-prometheus-stack, que despliega Prometheus (con su operador), Grafana y los exportadores de métricas de infraestructura. La Figura A.15 indica los comandos para la instalación.

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts && helm repo
update
helm upgrade --install kube-prometheus-stack
prometheus-community/kube-prometheus-stack \
  -n monitoring --create-namespace \
  -f infra/k8s/monitoring/values.yaml --wait --timeout=10m
```

Figura A.15: Instalación del operador kube-prometheus-stack

A.5.- DESPLIEGUE EN GOOGLE CLOUD PLATFORM

A.5.1.- Creación del proyecto

Primero crearemos un proyecto de GCP mediante la interfaz de console.cloud.google.com. La Figura A.16 muestra el formulario para la creación del proyecto.

Proyecto nuevo

⚠ Tienes 23 projects restantes en tu cuota. Solicita un incremento o borra algunos proyectos. [Más información](#)

[Manage Quotas](#)

Nombre del proyecto *

ID del proyecto: tfg-001-500014. No se puede cambiar más adelante. [Editar](#)

Organización *

Selecciona una organización para vincularla a un proyecto. No podrás cambiar esta selección más adelante.

Recurso superior * [Explorar](#)

Selecciona la organización o carpeta en la que se creará este recurso

[Crear](#) [Cancelar](#)

Figura A.16: Creación de proyecto en Google Cloud Platform

A.5.2.- Instalación de gcloud

La interacción con GCP se realiza mediante gcloud, la interfaz por línea de comandos que permite gestionar tanto los registros de artefactos como los clústers de GKE. Los comandos para la instalación se aprecian en la Figura A.17.

```
sudo apt update
sudo apt install -y apt-transport-https ca-certificates gnupg curl
curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg | \
sudo gpg --dearmor -o /usr/share/keyrings/cloud.google.gpg

echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] \
https://packages.cloud.google.com/apt cloud-sdk main" | \
sudo tee /etc/apt/sources.list.d/google-cloud-sdk.list

sudo apt update
sudo apt install -y google-cloud-cli
```

Figura A.17: Instalación de gcloud

Una vez instalada, la herramienta debe autenticarse con la cuenta de usuario y asociarse al proyecto de GCP en el que se desplegará el clúster. Los comandos de autenticación se aprecian en la Figura A.18.

```
gcloud auth login
gcloud config set project <ID_PROYECTO>
```

Figura A.18: Autenticación de gcloud

A.5.3.- Creación de registro y subida de imágenes

A diferencia del entorno local, los nodos de GKE obtienen las imágenes extrayéndolas de un registro remoto. Para ello se emplea Artifact Registry, el servicio de registro de imágenes de contenedor de GCP. Se utiliza un repositorio de tipo Docker, denominado bench, ubicado en la región us-central1. Su dirección completa sigue el formato: *us-central1-docker.pkg.dev/<ID_PROYECTO>/bench*.

La creación del repositorio se realiza mediante el comando mostrado en la Figura A.19.

```
gcloud artifacts repositories create bench \
  --repository-format=docker \
  --location=us-central1 \
  --description="Imágenes del proyecto"
```

Figura A.19: Creación del repositorio de imágenes

Para subir las imágenes se les añade la etiqueta con la dirección del repositorio y se ejecuta el comando docker push. En la Figura A.20 se aprecia las imágenes en el Artifact Registry.

Detalles del repositorio

Formato

Docker

Tipo

Estándar

▼

Mostrar más

≡ Filtro

Escribir el nombre o valor de la propiedad

☐	Nombre ↑	Conexión	Fecha de creación	Actualizado
☐	 benchmark-consumer	—	hace 1 minuto	Hace unos instantes
☐	 benchmark-producer	—	hace 1 minuto	hace 1 minuto
☐	 benchmark-scrape-metrics	—	Hace unos instantes	Hace unos instantes
☐	 flink	—	hace 23 horas	hace 1 minuto
☐	 kafka-streams	—	hace 1 minuto	hace 1 minuto
☐	 spark	—	hace 1 minuto	hace 1 minuto

Figura A.20: Imágenes en el repositorio de Artifact Registry

A.5.4.- Creación y configuración de Google Kubernetes Engine

El aprovisionamiento del clúster descrito en el Capítulo 4 se aprovisiona con los comandos descritos en la Figura A.21.

```
gcloud container clusters create bench \
  --project=<ID_PROYECTO> --zone=us-central1-a \
  --num-nodes=1 --machine-type=e2-standard-8 --spot \
  --node-labels=bench-role=ops \
  --disk-type=pd-balanced \
  --disk-size=100

gcloud container node-pools create kafka \
  --cluster=bench --project=<ID_PROYECTO> --zone=us-central1-a \
  --num-nodes=1 --machine-type=n2-standard-4 --spot \
  --node-labels=bench-role=kafka \
  --node-taints=bench-role=kafka:NoSchedule \
  --disk-type=pd-balanced --disk-size=50 \
  --ephemeral-storage-local-ssd count=2

gcloud container node-pools create worker \
  --cluster=bench --project=<ID_PROYECTO> --zone=us-central1-a \
  --num-nodes=3 --machine-type=n2-standard-4 --spot \
  --node-labels=bench-role=worker \
  --node-taints=bench-role=worker:NoSchedule \
  --disk-type=pd-balanced --disk-size=50 \
  --ephemeral-storage-local-ssd count=1
```

Figura A.21: Aprovisionamiento del clúster en GCP

Una vez concluido el aprovisionamiento, es necesario obtener las credenciales del clúster para que kubectl pueda operar sobre él. El comando de la Figura A.22 configura un contexto de kubectl apuntado al clúster de GKE y lo establece como activo.

```
gcloud container clusters get-credentials bench \
  --project=<ID_PROYECTO> --zone=us-central1-a
```

Figura A.22: Obtención de claves del clúster

Por último se aplica el espacio de nombres mediante el comando *kubectl apply -f namespace.yaml*.

A.6.- OPERATIVIDAD DEL CLÚSTER

A.6.1.- Despliegue de Kafka

Con el operador de Strimzi, el clúster de Kafka se despliega de forma declarativa. La definición del recurso Kafka se gestiona con Kustomize, lo que permite seleccionar la variante correspondiente al entorno, local o en GKE. El comando para desplegarlo en local es el siguiente: `kubectl apply -k infra/k8s/kafka/overlays/kind`.

A continuación se crean los tópicos de Kafka, igualmente como recursos gestionados por Strimzi (KafkaTopic). El comando es `kubectl apply -f infra/k8s/kafka/topics.yaml`

A.6.2.- Despliegue de la observabilidad

Mediante el operador kube-prometheus-stack se aplican los recursos *PodMonitor*, que indican a Prometheus que pods debe monitorizar. También se cargan los dashboards de Grafana, que se publican como *ConfigMap* etiquetados como *grafana_dashboard=1*, etiqueta por la que el sidecar de Grafana los descubre y carga automáticamente. La Figura A.23 muestra los comandos necesarios.

```
kubectl apply -f infra/k8s/monitoring/podmonitors.yaml

# Por cada panel en infra/grafana/dashboards/:
kubectl create configmap grafana-dashboard-producer -n
monitoring \

--from-file=producer-dashboard.json=infra/grafana/dashboards/pro
ducer-dashboard.json

kubectl label configmap grafana-dashboard-producer -n monitoring
grafana_dashboard=1
```

Figura A.23: Despliegue de herramientas de observabilidad

A.6.3.- Ejecución de un experimento

La ejecución sigue los siguientes pasos:

1. **Despliegue del job.** Se genera el manifiesto del trabajo mediante Kustomize, seleccionando la combinación de marco, carga, modo de garantía y entorno, y se

aplica al clúster. Se espera a que alcance el estado de ejecución (RUNNING para Flink y Spark, Available para el despliegue de Kafka Streams).

2. **Arranque del consumidor.** Se lanza el consumidor sobre el tópico de salida y se espera a que esté listo antes de generar carga, para no perder los primeros resultados.
3. **Arranque del productor.** Se lanza el productor sobre el tópico de entrada con los parámetros del experimento.
4. **Espera y medición.** Se aguarda a la finalización del productor y del consumidor. La ventana de medición se acota descartando el periodo de calentamiento inicial.
5. **Extracción de métricas.** Se lanza el job de extracción, que consulta Prometheus sobre la ventana delimitada.

El comportamiento de cada ejecución se controla mediante un conjunto de parámetros descritos en la Tabla A.1.

Parámetro	Descripción
FRAMEWORK	Motor de procesamiento: kafka-streams, flink o spark.
JOB	Job a utilizar: stateless-filter, keyed-aggregator o window-aggregator.
MODE	Modo de garantía de entrega: exactly-once (eos) o at-least-once (alo).
ENV	Entorno de ejecución: kind o gke.
RATE	Throughput objetivo en eventos/s.
DURATION	Duración de la fase de envío de evento, en segundos.
SIZE_BYTES	Tamaño de cada evento, en bytes.
NUM_KEYS	Número de claves distintas
WARMUP	Periodo inicial de calentamiento, en segundos
DRAIN	Margen final, en segundos
BENCHMARK_ID	Identificador para agrupar pruebas

Tabla A.1: Parámetros de ejecución de pruebas

A.6.4.- Recolección y extracción de resultados

Los resultados de un experimento tienen dos orígenes. Por un lado, el consumidor calcula directamente las métricas de latencia y rendimiento de extremo a extremo, que registra en su salida. Por otro, el servicio de recolección de métricas consulta la API de Prometheus

sobre la ventana temporal del experimento y vuelca, en un fichero CSV por métrica, las series temporales de las métricas de los componentes (uso de CPU y memoria, tasas de proceso, retraso del consumidor, etc.).

Estos ficheros se generan en un volumen del clúster y deben transferirse a la máquina local para su análisis posterior. Dicha transferencia se realiza mediante un pod auxiliar que monta el volumen de métricas, desde el cual se copian los ficheros al directorio de resultados del proyecto: **`kubectll cp bench/benchctl-metrics-copier:/metrics/./analysis/results/ -c copier`**.

Los resultados quedan así disponibles en *analysis/results/* listos para su procesamiento en el cuaderno de análisis (*analysis/analysis.ipynb*).

A.6.5.- Monitorización

Durante la ejecución de un experimento, o tras ella, puede consultarse el estado del sistema en tiempo real a través de Grafana. Al estar el servicio expuesto internamente (*ClusterIP*), el acceso desde la máquina local se habilita mediante una redirección de puertos: **`kubectll port-forward -n monitoring svc/kube-prometheus-stack-grafana 3000:80`**.

La interfaz queda accesible en <http://localhost:3000>, con las credenciales definidas en el fichero de valores (usuario admin). Desde ella se dispone de los paneles cargados, que cubren múltiples servicios. La Figura A.24 muestra un ejemplo del panel de Kafka.

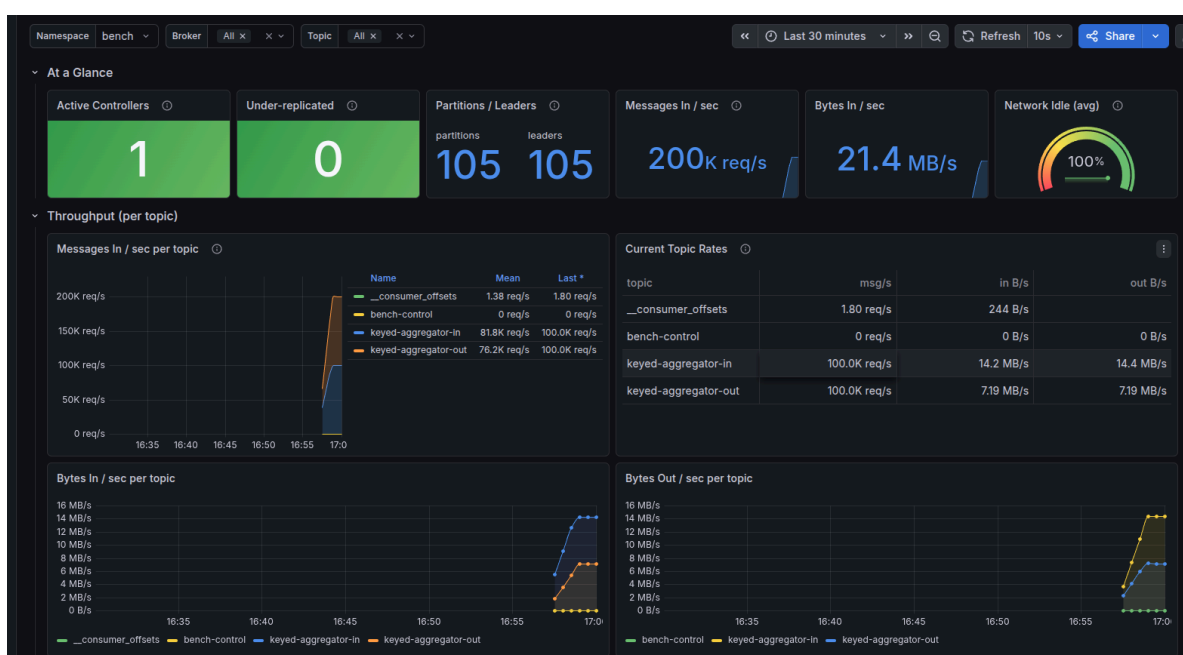


Figura A.24: Panel de Kafka en Grafana