PUIGFLOWS

Exactly-Once en Kafka: Productor Idempotente, Transacciones, read_committed y la Frontera que Todo el Mundo Cruza

Exactly-Once in Kafka: The Idempotent Producer, Transactions, read_committed, and the Boundary Everyone Crosses

Josué Puig · puigflows.com

Guía · Nivel intermedio · Lectura estimada: 37 min · Edición del 20 de septiembre de 2026

Documento bilingüe: versión en español seguida de la versión en inglés.
Bilingual document: Spanish version followed by the English version.

Tres garantías, y sólo una que la gente quiere

Cuando alguien dice que su pipeline de Kafka es exactly-once, casi siempre quiere decir una de estas dos cosas: que no ha visto duplicados todavía, o que ha puesto processing.guarantee=exactly_once_v2 en un fichero de configuración y ha dado el tema por cerrado. Ninguna de las dos es una garantía.

Las tres opciones reales son estas:

- At-most-once: confirmas el offset antes de procesar. Si el proceso muere en medio, el mensaje se pierde y nadie vuelve a mirarlo. Barato y rápido; inaceptable para un pago.
- At-least-once: procesas y luego confirmas. Si el proceso muere entre ambas cosas, el mensaje se reprocesa. Es el comportamiento por defecto de prácticamente todo, y es correcto siempre que tu procesamiento sea idempotente.
- Exactly-once: cada registro de entrada afecta al resultado exactamente una vez. No significa que el mensaje se entregue físicamente una sola vez —eso es imposible en una red asíncrona—, significa que el efecto se aplica una sola vez.

Esa última distinción es la que decide si tu diseño funciona. Kafka no impide que un registro se entregue dos veces; impide que se contabilice dos veces, y sólo dentro de un perímetro muy concreto: de Kafka a Kafka. En cuanto tu consumidor hace un POST a un proveedor de pagos o un INSERT en PostgreSQL, sales de ese perímetro y la garantía deja de aplicarse. Lo repetiré más abajo con un ejemplo, porque es el error que más dinero cuesta.

Los dos duplicados, que no son el mismo problema

Antes de tocar configuración conviene separar dos fuentes de duplicación que la gente mezcla constantemente.

El duplicado del productor. Envías un ProduceRequest, el broker lo escribe en el log, y la respuesta se pierde por un corte de red o un timeout. El cliente no tiene forma de distinguir «no llegó» de «llegó y no me enteré», así que reintenta. Resultado: el mismo registro dos veces en la partición, con dos offsets distintos. Esto ocurre por debajo de tu código; ningún control de duplicados en la aplicación lo ve venir.

El duplicado del consumidor. Lees un lote, lo procesas, escribes la salida y mueres antes de confirmar el offset. Otro miembro del grupo recoge la partición desde el último offset confirmado y vuelve a procesar lo mismo. La salida se duplica aunque la entrada estuviera perfectamente deduplicada.

Kafka resuelve el primero con el productor idempotente y el segundo con las transacciones. Son dos mecanismos distintos, con costes distintos, y uno de ellos ya está activo en tu clúster aunque no lo sepas.

Capa 1: el productor idempotente

Desde Kafka 3.0, enable.idempotence vale true por defecto. El mecanismo es sencillo y elegante: al arrancar, el productor pide al broker un producer ID (PID). A partir de ahí, cada lote que envía lleva el PID, una época y un número de secuencia monótono por partición. El líder de cada partición recuerda el número de secuencia del último lote aceptado para ese PID y rechaza cualquier lote cuyo número ya haya visto, devolviendo éxito al cliente sin escribir nada.

Es deduplicación en el broker, en el camino de datos, sin que tu aplicación se entere. Activarlo fuerza tres ajustes:

```
acks=all
retries=2147483647
max.in.flight.requests.per.connection=5   # como máximo 5
```

El límite de cinco peticiones en vuelo no es arbitrario: el broker sólo guarda metadatos de los últimos cinco lotes por PID y partición, que es lo que necesita para reordenar y deduplicar sin perder el orden. Si subes ese valor con la idempotencia activada, el cliente falla al arrancar en lugar de darte una garantía a medias, que es el comportamiento correcto.

Hay una excepción que merece la pena conocer: si configuras acks=1 o acks=0 junto con enable.idempotence=true, el productor lanza una ConfigException. Mucha gente arrastra un acks=1 de un tutorial de 2019 y se encuentra con que su aplicación ya no arranca tras una actualización. La respuesta correcta casi nunca es desactivar la idempotencia.

Lo que el productor idempotente no cubre

Tres límites, y son importantes:

- Una sola sesión de productor. Si el proceso se reinicia, obtiene un PID nuevo y los contadores empiezan de cero. Un registro enviado antes del reinicio y reenviado por tu lógica de aplicación después será un duplicado perfectamente legítimo a ojos del broker.
- Una sola partición a la vez. No hay atomicidad entre particiones ni entre topics. Si escribes a tres topics y el proceso muere tras el segundo, tienes dos escrituras y media.
- Retención limitada del estado. Si un productor queda inactivo más tiempo del que el broker mantiene su estado (transactional.id.expiration.ms y la caducidad del PID), puede aparecer una OutOfOrderSequenceException. No es un bug tuyo: es el broker diciendo que ha perdido el hilo y que no puede garantizar nada. Se maneja recreando el productor.

Capa 2: transacciones

Las transacciones, introducidas por KIP-98 en la versión 0.11, añaden lo que falta: atomicidad entre varias particiones y topics, y entre la escritura de resultados y la confirmación de offsets de consumo. Esto último es la pieza clave, porque convierte el clásico consume-transform-produce en una operación de todo o nada.

La mecánica, por encima:

- Configuras un transactional.id. Es el identificador estable de una instancia lógica de tu aplicación, y su estabilidad entre reinicios es lo que hace funcionar todo lo demás.
- initTransactions() registra ese id contra el coordinador de transacciones (el broker líder de la partición correspondiente del topic interno __transaction_state), incrementa la época asociada y aborta cualquier transacción pendiente de una encarnación anterior del proceso. Esto es el zombie fencing: un proceso antiguo que siguiera vivo en otro nodo, con la época vieja, recibe ProducerFencedException en su siguiente escritura y ya no puede corromper nada.
- Entre beginTransaction() y commitTransaction(), todo lo que escribas —en cualquier partición de cualquier topic— pertenece a la misma transacción.
- sendOffsetsToTransaction(...) escribe los offsets de consumo en __consumer_offsets dentro de la misma transacción. Este es el detalle que hace que la garantía sea real.

- Al confirmar, el coordinador escribe un marcador de control (commit o abort) en cada partición implicada. Los consumidores con read_committed usan esos marcadores para decidir qué entregar.

El bucle completo, en Java

```
Properties cp = new Properties();
cp.put("bootstrap.servers", "kafka:9092");
cp.put("group.id", "enriquecedor-pedidos");
cp.put("enable.auto.commit", "false");        // obligatorio
cp.put("isolation.level", "read_committed");  // no leer lo no confirmado
cp.put("key.deserializer", StringDeserializer.class.getName());
cp.put("value.deserializer", StringDeserializer.class.getName());

Properties pp = new Properties();
pp.put("bootstrap.servers", "kafka:9092");
// Estable entre reinicios. Aquí: nombre de la app + índice ordinal del pod.
pp.put("transactional.id", "enriquecedor-pedidos-" + ordinalDelPod());
pp.put("enable.idempotence", "true");
pp.put("transaction.timeout.ms", "60000");
pp.put("key.serializer", StringSerializer.class.getName());
pp.put("value.serializer", StringSerializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(cp);
KafkaProducer<String, String> producer = new KafkaProducer<>(pp);

producer.initTransactions();   // fencing + aborta transacciones huérfanas
consumer.subscribe(List.of("pedidos"));

try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(200));
        if (records.isEmpty()) continue;

        producer.beginTransaction();
        try {
            for (ConsumerRecord<String, String> r : records) {
                String salida = enriquecer(r.value());   // función pura, sin efectos externos
                producer.send(new ProducerRecord<>("pedidos-enriquecidos", r.key(), salida));
            }

            Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();
            for (TopicPartition tp : records.partitions()) {
                List<ConsumerRecord<String, String>> deLaParticion = records.records(tp);
                long ultimo = deLaParticion.get(deLaParticion.size() - 1).offset();
                offsets.put(tp, new OffsetAndMetadata(ultimo + 1));   // +1: el siguiente a leer
            }
            // La sobrecarga con groupMetadata() es la que habilita EOS v2.
            producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata());

            producer.commitTransaction();
        } catch (ProducerFencedException | OutOfOrderSequenceException
                 | AuthorizationException e) {
            // Irrecuperables: otro proceso nos ha reemplazado. Salir.
            producer.close();
            throw e;
        } catch (KafkaException e) {
            producer.abortTransaction();
            // El consumidor debe volver atrás: los offsets no se confirmaron.
            for (TopicPartition tp : records.partitions()) {
                consumer.seek(tp, records.records(tp).get(0).offset());
            }
        }
    }
} finally {
    consumer.close();
    producer.close();
}
```

Tres detalles de ese código que se saltan casi todas las implementaciones que he revisado:

El seek tras abortar. Abortar la transacción descarta las escrituras y los offsets, pero no mueve la posición en memoria del consumidor. Sin ese seek, el siguiente poll() devuelve los registros siguientes y acabas de saltarte un lote entero sin procesarlo. Es una pérdida silenciosa de datos causada por un mecanismo pensado para evitar pérdidas.

El +1 en el offset. Kafka confirma «el siguiente offset que quiero leer», no «el último que procesé». Confundirlo reprocesa un registro por partición y por commit para siempre.

consumer.groupMetadata(). La sobrecarga antigua recibía sólo el consumerGroupId. La que recibe ConsumerGroupMetadata transmite también el id de miembro y la época de generación, lo que permite al coordinador aislar a un miembro zombi del grupo. Es el núcleo de lo que se llamó EOS v2 y lo que hace innecesario tener un transactional.id por partición de entrada.

El mismo patrón en Python

La biblioteca confluent-kafka expone la misma API transaccional sobre librdkafka:

```
from confluent_kafka import Consumer, Producer, KafkaException, TopicPartition

consumer = Consumer({
    "bootstrap.servers": "kafka:9092",
    "group.id": "enriquecedor-pedidos",
    "enable.auto.commit": False,
    "isolation.level": "read_committed",
    "auto.offset.reset": "earliest",
})

producer = Producer({
    "bootstrap.servers": "kafka:9092",
    "transactional.id": f"enriquecedor-pedidos-{ordinal_del_pod()}",
    "enable.idempotence": True,
    "transaction.timeout.ms": 60000,
})

producer.init_transactions()
consumer.subscribe(["pedidos"])

while True:
    msgs = consumer.consume(num_messages=500, timeout=1.0)
    msgs = [m for m in msgs if m is not None and not m.error()]
    if not msgs:
        continue

    producer.begin_transaction()
    try:
        for m in msgs:
            producer.produce("pedidos-enriquecidos", key=m.key(),
                             value=enriquecer(m.value()))

        # Offset del siguiente registro a leer, por partición.
        ultimos = {}
        for m in msgs:
            clave = (m.topic(), m.partition())
            ultimos[clave] = max(ultimos.get(clave, -1), m.offset())
        offsets = [TopicPartition(t, p, o + 1) for (t, p), o in ultimos.items()]

        producer.send_offsets_to_transaction(offsets, consumer.consumer_group_metadata())
        producer.commit_transaction()

    except KafkaException as e:
        err = e.args[0]
        if err.txn_requires_abort():
```

```
            producer.abort_transaction()
            for tp in consumer.assignment():
                consumer.seek(TopicPartition(tp.topic, tp.partition,
                                             primer_offset_del_lote(msgs, tp)))
        elif err.retriable():
            continue          # reintentar la misma operación
        else:
            raise             # fatal: cerrar el proceso
```

Fíjate en err.txn_requires_abort() y err.retriable(). librdkafka clasifica los errores transaccionales en tres categorías —reintentable, requiere abortar, y fatal— y tratarlas igual es la forma más rápida de construir un productor que se queda colgado o que aborta cuando no debería.

read_committed, el LSO y los offsets que «faltan»

Un consumidor con isolation.level=read_committed nunca entrega registros más allá del Last Stable Offset (LSO): el offset de la primera transacción todavía abierta en esa partición. Todo lo que haya después existe en el log, está replicado y está a salvo, pero es invisible hasta que esa transacción se confirme o se aborte.

De aquí salen dos comportamientos que sorprenden:

Los offsets no son contiguos. Los marcadores de commit y abort ocupan un offset en la partición. Si procesas los offsets 100 a 104 en una transacción, el marcador se lleva el 105 y tu consumidor verá el siguiente registro real en el 106. Cualquier monitorización que calcule progreso como «último offset consumido menos primero» o que asuma continuidad se equivocará, siempre por poco, siempre de forma difícil de depurar.

Los mensajes abortados se filtran en el cliente. El broker envía a los consumidores el lote completo junto con un índice de transacciones abortadas; es el cliente quien descarta. Esto significa que el ancho de banda de fetch incluye datos que nunca verás. Si tu tasa de aborts es alta, estás pagando red por nada, y eso aparece en las métricas como un throughput que no cuadra con el número de registros procesados.

Y la consecuencia operativa más importante: una transacción colgada congela el LSO. El lag de tus consumidores read_committed crece sin parar mientras el log-end-offset avanza, y no hay ningún error en los logs de la aplicación. Volveremos a esto en la sección de operación.

Kafka Streams: la versión de una línea

Si tu procesamiento vive en Kafka Streams, todo lo anterior está implementado por ti:

```
processing.guarantee=exactly_once_v2
```

Streams gestiona el transactional.id, incluye en la transacción las escrituras a los changelog topics de sus state stores, y confirma los offsets dentro de ella. Los valores antiguos exactly_once y exactly_once_beta fueron eliminados en Kafka 4.0; si migras desde una versión antigua, ese es el cambio.

El parámetro que importa aquí es commit.interval.ms, que con EOS activado baja por defecto a 100 ms. Cada commit es una transacción, y cada transacción tiene un coste fijo. Subirlo a 1000 ms suele multiplicar el throughput a cambio de añadir latencia de punta a punta: los consumidores read_committed no ven nada hasta que la transacción cierra. Ese es el único mando de verdad relevante y casi nadie lo toca.

Qué cambió con KIP-890 y transaction.version=2

El protocolo transaccional original tenía un agujero conocido. Cuando un productor sufría una pausa larga —un GC, un throttling de cgroups, una partición de red— y el coordinador abortaba su transacción por timeout, ese productor podía despertar y seguir enviando registros con el mismo PID y la misma época. Esos registros aterrizaban después del marcador de abort y quedaban atribuidos a la transacción siguiente. El resultado eran dos patologías: mensajes que se filtraban de una transacción a otra, y transacciones colgadas que bloqueaban el LSO indefinidamente.

KIP-890 lo arregla en la raíz. Con transaction.version=2, disponible desde Kafka 4.0, la época del productor se incrementa en cada commit y en cada abort. El par (PID, época) identifica de forma única una transacción concreta, así que un registro tardío de la transacción anterior llega con una época obsoleta y el broker lo rechaza sin ambigüedad. Además, la verificación se pliega dentro de las peticiones existentes en lugar de necesitar un viaje extra por partición, lo que quita sobrecarga del camino de datos.

KIP-890 también introduce TransactionAbortableException, y es una mejora práctica real: marca explícitamente los errores en los que la respuesta correcta es abortar y reintentar el lote, en lugar de obligarte a deducirlo del tipo de excepción. Si tu manejador de errores es un catch (KafkaException e) genérico, esta es la excusa para reescribirlo.

La versión de transacción es una feature del clúster, no una propiedad del cliente. Se consulta y se activa con la herramienta de features:

```
# Ver las versiones de feature activas
bin/kafka-features.sh --bootstrap-server kafka:9092 describe

# Activar transacciones v2 (requiere clientes actualizados)
bin/kafka-features.sh --bootstrap-server kafka:9092 \
  upgrade --feature transaction.version=2
```

Antes de activarlo, comprueba tus clientes. Los que no hablan el protocolo nuevo siguen funcionando con la semántica antigua, pero no obtienes la protección. En el ecosistema no-Java esto tardó en llegar; si usas Sarama, kafka-go o un binding sobre librdkafka, verifica la versión concreta antes de asumir que estás cubierto.

La frontera que todo el mundo cruza sin darse cuenta

Aquí está el error que justifica el artículo entero. Este código parece exactly-once:

```
producer.beginTransaction();
for (ConsumerRecord<String, String> r : records) {
        // &þ  Esto NO forma parte de la transacción de Kafka.
    pasarelaDePagos.cobrar(r.value());
    producer.send(new ProducerRecord<>("pagos-realizados", r.key(), r.value()));
}
producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata());
producer.commitTransaction();
```

Si el proceso muere justo antes de commitTransaction(), Kafka hace exactamente lo que promete: descarta el evento pagos-realizados y descarta los offsets. El lote se reprocesa. Y el cliente recibe un segundo cargo, porque pasarelaDePagos.cobrar() ocurrió fuera del perímetro transaccional y no hay nada que lo pueda deshacer.

Las transacciones de Kafka son un mecanismo Kafka-a-Kafka. Punto. Para cualquier efecto externo tienes

tres salidas honestas:

- Hacer idempotente la operación externa. Una clave de idempotencia derivada de forma determinista del registro —por ejemplo topic-particion-offset, o el identificador de negocio si existe— y que el sistema remoto usa para deduplicar. Casi todas las pasarelas de pago serias lo soportan.
- Mover la frontera transaccional al sink. Escribe el resultado y el offset consumido en la misma transacción de base de datos, con un UPSERT por clave. Entonces el offset de Kafka es sólo una optimización de arranque y la verdad vive en tu base de datos.
- El patrón transactional outbox, al revés. Si el efecto externo lo produces tú, escríbelo como fila en una tabla dentro de tu transacción de negocio y deja que un relay lo publique en Kafka con at-least-once más deduplicación en el destino.

Ninguna de las tres necesita transacciones de Kafka. Esa es justamente la conclusión que mucha gente no quiere oír: en un pipeline que termina en una base de datos, las transacciones de Kafka suelen ser el mecanismo equivocado, y un UPSERT idempotente resuelve el problema con una fracción del coste operativo.

Ocho errores que verás en producción

- transactional.id aleatorio en cada arranque. UUID.randomUUID() parece inofensivo y destruye la garantía: sin un id estable no hay fencing, y un proceso zombi puede seguir escribiendo. Además dejas atrás un id por reinicio, cada uno con su estado en el coordinador, hasta que caduca por transactional.id.expiration.ms. En Kubernetes, usa el ordinal del StatefulSet; con Deployment, deriva el id de una asignación estable, no del nombre del pod.
- Confirmar offsets con consumer.commitSync(). Rompe la atomicidad: los offsets salen de la transacción y vuelves a tener at-least-once con más complejidad y más latencia. Si hay un commitSync en un bucle transaccional, es un bug.
- Dejar enable.auto.commit=true. El mismo problema, pero peor porque es invisible: un hilo de fondo confirma offsets por su cuenta y puede hacerlo antes de que la transacción cierre.
- No hacer seek tras un abort. Salto silencioso de un lote completo, ya comentado arriba. Es el fallo más difícil de detectar de toda la lista porque no genera ni un error.
- transaction.timeout.ms por encima del máximo del broker. Si supera transaction.max.timeout.ms, el broker rechaza al productor en initTransactions() con InvalidTxnTimeoutException. Y si lo pones demasiado bajo, una pausa de GC hace que el coordinador aborte una transacción que tu proceso cree viva.
- Compartir un productor transaccional entre hilos. Un productor transaccional mantiene una única transacción en curso. Dos hilos llamando a beginTransaction() producen un caos que se manifiesta como IllegalStateException intermitentes en producción y nunca en local.
- Reintentar tras ProducerFencedException. Esa excepción significa que otro proceso ha tomado tu identidad legítimamente. Reintentar es exactamente lo contrario de lo correcto: cierra el productor y termina el proceso para que el orquestador lo reemplace.
- Transacciones de un solo mensaje. El coste de una transacción es fijo: dos escrituras en el coordinador más un marcador por partición. Amortizado sobre quinientos mensajes es ruido; sobre uno, puede multiplicar por diez el coste por registro. Agrupa por lote de poll(), no por mensaje.

Operación: la transacción colgada

El incidente característico de EOS no se parece a un incidente. El síntoma es: el lag de un grupo de consumo crece linealmente, el topic recibe tráfico normal, los productores no dan errores, los consumidores tampoco, y reiniciar no cambia nada. Lo que ocurre es que una transacción quedó abierta y el LSO no avanza.

Kafka trae una herramienta específica para esto desde la 3.0:

```
# Encontrar transacciones colgadas en un topic
bin/kafka-transactions.sh --bootstrap-server kafka:9092 \
  find-hanging --topic pedidos-enriquecidos

# Inspeccionar el estado de un transactional.id concreto
bin/kafka-transactions.sh --bootstrap-server kafka:9092 \
  describe --transactional-id enriquecedor-pedidos-0

# Abortar una transacción colgada (último recurso, con los datos de find-hanging)
bin/kafka-transactions.sh --bootstrap-server kafka:9092 \
  abort --topic pedidos-enriquecidos --partition 3 --start-offset 84211
```

El abort manual descarta los registros de esa transacción de forma definitiva. Ejecútalo sabiendo qué estás tirando, y déjalo documentado en el runbook con esa advertencia en la primera línea.

Para detectarlo antes de que alguien abra un ticket, la alerta útil no es sobre el lag sino sobre la distancia entre el log-end-offset y el LSO. Si esa diferencia crece de forma sostenida durante más tiempo que tu transaction.timeout.ms, tienes una transacción atascada, sea cual sea el lag de los consumidores.

En el cliente, el grupo de métricas producer-metrics expone los tiempos de las operaciones transaccionales —txn-init-time-ns-avg, txn-begin-time-ns-avg, txn-send-offsets-time-ns-avg, txn-commit-time-ns-avg y txn-abort-time-ns-avg—. La que más información da por euro invertido es la de abort: si tu tasa de aborts pasa de anecdótica a constante, algo cambió, y suele ser una dependencia lenta que empuja el procesamiento por encima del timeout.

El coste, dicho con números honestos

La sobrecarga de las transacciones no es una constante que puedas citar; depende casi por completo de cuántos mensajes metes en cada una. Con transacciones grandes y un commit.interval.ms generoso, la penalización de throughput es pequeña —del orden de un puñado de puntos porcentuales— y suele quedar enterrada bajo el ruido del resto del sistema. Con transacciones diminutas y commits cada 100 ms, es perfectamente posible perder la mitad del throughput.

La latencia es otra historia y menos negociable. Con read_committed, un registro no es visible hasta que su transacción se confirma, así que tu latencia de punta a punta tiene un suelo igual al intervalo de commit. Un pipeline de cinco etapas transaccionales con commits de un segundo arrastra cinco segundos antes de procesar nada. Ese cálculo hay que hacerlo antes de elegir la arquitectura, no después de la primera queja.

Checklist de producción

- enable.idempotence=true en todos los productores, incluidos los no transaccionales. Es gratis y elimina una clase entera de duplicados.
- transactional.id derivado de una identidad estable (ordinal del StatefulSet, partición asignada, slot fijo). Nunca aleatorio, nunca el hostname efímero.

- enable.auto.commit=false y cero llamadas a commitSync/commitAsync en el bucle transaccional.
- isolation.level=read_committed en todo consumidor que lea un topic escrito transaccionalmente. Por defecto es read_uncommitted, y ese defecto anula la garantía sin avisar.
- seek al primer offset del lote tras cada abortTransaction().
- Manejo de errores diferenciado: fatal (cerrar), abortable (abortar y reintentar), reintentable (reintentar la misma llamada).
- transaction.timeout.ms por encima del peor caso de procesamiento de un lote, con margen para una pausa de GC, y por debajo de transaction.max.timeout.ms del broker.
- Alerta sobre la brecha entre log-end-offset y LSO, además de la alerta de lag habitual.
- Una prueba de integración que mate el proceso entre send y commit —con Testcontainers va bien— y afirme que el topic de salida no contiene duplicados.
- Cualquier efecto externo dentro del bucle, documentado y protegido con su propia clave de idempotencia. Si no puedes, la transacción de Kafka no te está dando lo que crees.

Preguntas frecuentes

¿Necesito transacciones si sólo produzco, sin consumir? No. El productor idempotente cubre los duplicados por reintento, que es tu único riesgo. Las transacciones sólo aportan algo si necesitas atomicidad entre varias particiones o entre escrituras y offsets.

¿Funciona EOS entre dos clústeres de Kafka? No, y esta es una limitación que sorprende. El coordinador de transacciones es un componente de un clúster. Una transacción no puede abarcar dos. Con MirrorMaker 2 o con replicación entre regiones, la garantía es at-least-once y necesitas deduplicación en el destino.

¿Y si uso read_uncommitted contra un topic transaccional? Verás los registros de transacciones abortadas como si fueran válidos, y los verás antes de que se confirmen. Es un modo perfectamente legítimo para depurar o para consumidores que toleran basura, pero anula la garantía por completo. Compruébalo: es el defecto del cliente.

¿Puedo mezclar escrituras transaccionales y no transaccionales en el mismo topic? Técnicamente sí, y los consumidores read_committed entregan las no transaccionales con normalidad. En la práctica complica el razonamiento sobre el LSO y sobre qué está confirmado. Elige una cosa por topic.

¿El compactado de logs respeta las transacciones? Sí; los marcadores de control se limpian cuando ya no son necesarios y el compactado opera sobre los registros confirmados. No es algo que tengas que configurar, pero sí algo que conviene saber antes de diseñar sobre un topic compactado.

Glosario

- PID (producer ID): identificador que el broker asigna a una sesión de productor idempotente.
- Época (epoch): contador asociado al PID o al transactional.id; incrementarlo invalida a cualquier productor anterior.
- Fencing: mecanismo por el que un productor antiguo queda excluido al aparecer uno nuevo con la misma identidad y época mayor.
- LSO (Last Stable Offset): offset de la primera transacción abierta en una partición; techo de lo que un consumidor read_committed puede leer.
- Marcador de control: registro especial de commit o abort que el coordinador escribe en cada partición de

una transacción. Ocupa un offset.

- Coordinador de transacciones: broker responsable de un transactional.id, que mantiene su estado en el topic interno __transaction_state.
- EOS v2: revisión del protocolo que usa los metadatos del grupo de consumo para el fencing, eliminando la necesidad de un productor por partición de entrada.
- Transacción colgada: transacción que quedó abierta sin commit ni abort y que congela el LSO de sus particiones.

Para llevarse

El productor idempotente es gratis, está activado y deberías dejarlo así. Las transacciones son caras, resuelven un problema concreto —atomicidad entre particiones y entre resultados y offsets— y sólo dentro de Kafka. Antes de activarlas, contesta una pregunta: ¿mi pipeline termina en Kafka, o termina en una base de datos o en una API? Si es lo segundo, la respuesta correcta casi siempre es at-least-once con un sink idempotente, y te ahorras un protocolo entero de coordinación distribuida que además tienes que operar.

Ampliación del 20 de septiembre de 2026: las tripas, los bordes y el resto del ecosistema

Lo anterior cubre el modelo mental y el bucle de consumo-proceso-producción. Lo que sigue es la parte que se necesita cuando el sistema lleva meses en producción: cómo funciona el mecanismo por dentro, por qué un transactional.id mal elegido convierte una garantía en una falsa sensación de seguridad, qué hace read_committed a tu latencia, cómo se comportan Kafka Connect y Flink, y qué tests demuestran que todo esto funciona antes de que lo demuestre un incidente.

Por dentro: PID, epoch y números de secuencia

La idempotencia del productor no es magia ni deduplicación por contenido. Cuando arranca, el productor pide al broker un producer ID (PID). A partir de ahí, cada lote que envía lleva tres cosas: el PID, un epoch y un número de secuencia que se incrementa por partición. El broker guarda, para cada par (PID, partición), el último número de secuencia que escribió. Si llega un lote con el número que ya tiene, lo descarta y responde éxito. Si llega uno con un número mayor del esperado, devuelve OutOfOrderSequenceException, porque eso significa que se perdió un lote intermedio y escribirlo rompería el orden.

De ahí salen dos consecuencias prácticas que explican configuraciones que la gente copia sin entender.

La primera: max.in.flight.requests.per.connection puede llegar hasta 5 con idempotencia activada y seguir garantizando el orden. El número no es arbitrario; es cuántos lotes por partición guarda el broker en memoria para poder reordenar los que le llegan desviados. Por encima de 5, el broker no puede reordenar y el productor rechaza la configuración al arrancar. Antes de la idempotencia, cualquier valor por encima de 1 con reintentos activados podía reordenar mensajes en silencio; ese es el bug histórico que esto resuelve.

La segunda: el PID caduca. Si un productor idempotente no escribe en una partición durante producer.id.expiration.ms —por defecto una hora— el broker olvida su estado de secuencia. Si después vuelve a escribir, se encuentra con UNKNOWN_PRODUCER_ID. Los clientes modernos lo resuelven pidiendo un PID nuevo, pero el síntoma que llega a los logs asusta lo suficiente como para que merezca estar escrito aquí: en un productor de bajo tráfico y muchas particiones, verlo es normal y no significa pérdida

de datos.

El epoch es el mecanismo de exclusión. Cada vez que un productor con el mismo transactional.id llama a initTransactions(), el coordinador incrementa el epoch y rechaza con ProducerFencedException cualquier escritura que llegue con uno anterior. Eso es lo que mata a los zombis: si tu proceso se congeló por una pausa de GC de cuarenta segundos, otro ocupó su sitio, y el primero vuelve a la vida creyendo que sigue siendo dueño de la transacción, sus escrituras se rechazan. La excepción no se captura para reintentar: se captura para cerrar el productor y morir.

El coordinador de transacciones y el log de estado

Una transacción no vive en los brokers que tienen tus particiones de datos. Vive en un coordinador de transacciones, que es el líder de una partición del topic interno __transaction_state, elegida haciendo hash de tu transactional.id. Ese topic es el que hace que la transacción sobreviva a la caída del broker: es un log replicado como cualquier otro.

La secuencia completa de un commitTransaction() tiene más pasos de los que sugiere el nombre del método:

- El productor llama a initTransactions() y el coordinador le asigna PID y epoch, y aborta cualquier transacción que el transactional.id anterior hubiera dejado abierta.
- En la primera escritura a cada partición, el productor avisa al coordinador con AddPartitionsToTxn. El coordinador anota en su log qué particiones forman parte de esta transacción; sin eso no sabría dónde escribir los marcadores al final.
- Los mensajes se escriben en las particiones de datos como mensajes normales, visibles de inmediato para un consumidor read_uncommitted.
- Al hacer commit, el coordinador escribe primero PrepareCommit en su propio log —el punto de no retorno— y sólo después manda a cada partición implicada un marcador de transacción, un registro de control que dice «todo lo anterior de este PID está confirmado».
- Cuando todos los marcadores están escritos, el coordinador anota CompleteCommit. Si el coordinador se cae entre el prepare y los marcadores, el nuevo coordinador lee su log, ve el prepare y termina el trabajo. Esto es un two-phase commit de verdad, con el log del coordinador como registro de decisión.

Entender esto explica por qué una transacción «colgada» bloquea a los consumidores de una partición que quizá ni siquiera te interesa: el LSO de esa partición no puede avanzar más allá del primer mensaje de una transacción sin marcador, y eso afecta a todo el que lea con read_committed, no sólo a tu aplicación.

El transactional.id: la decisión que casi nadie toma bien

Un transactional.id aleatorio por arranque —un UUID, por ejemplo— parece inofensivo y anula casi toda la garantía. El fencing funciona comparando epochs del mismo transactional.id; si cada reinicio genera uno nuevo, el proceso antiguo nunca se considera zombi y una instancia colgada puede seguir escribiendo mientras la nueva ya está trabajando. Además, el coordinador no tiene forma de saber que había una transacción abierta que había que abortar.

La regla es que el transactional.id debe ser estable entre reinicios y único por unidad de trabajo. En Kubernetes, un StatefulSet lo da gratis: el nombre del pod (procesador-0, procesador-1) es estable y único, mientras que en un Deployment los nombres cambian en cada despliegue y hay que derivarlo de otra cosa —por ejemplo del índice de partición asignado, si el reparto es estático—.

```
# StatefulSet: el nombre del pod es estable y sirve de transactional.id
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: procesador-pagos
spec:
  serviceName: procesador-pagos
  replicas: 4
  template:
    spec:
      containers:
        - name: app
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef: { fieldPath: metadata.name }
            # -> transactional.id = "pagos-procesador-pagos-0"
            - name: KAFKA_TRANSACTIONAL_ID
              value: "pagos-$(POD_NAME)"
```

Hay una pregunta histórica que conviene cerrar, porque en foros antiguos se lee lo contrario. Antes de KIP-447, cada transactional.id tenía que corresponder a un conjunto fijo de particiones de entrada, lo que en la práctica obligaba a crear un productor por partición de entrada: con doscientas particiones tenías doscientos productores por instancia, cada uno con sus buffers y sus conexiones. KIP-447 eliminó esa restricción pasando la metadata del grupo de consumidores junto con los offsets, de modo que el fencing puede hacerse con el epoch del miembro del grupo en lugar de con el reparto estático de particiones. Desde entonces basta un productor por instancia, y el transactional.id sigue siendo necesario para abortar transacciones huérfanas cuando una transacción no incluye ningún sendOffsets.

```
// El detalle que activa el fencing por grupo (KIP-447):
// pasar groupMetadata(), no el groupId como cadena.
producer.sendOffsetsToTransaction(
    offsets,
    consumer.groupMetadata()   // <- no consumer.groupId()
);
producer.commitTransaction();
```

La sobrecarga de sendOffsetsToTransaction(offsets, "mi-grupo"), con el identificador como cadena, está marcada como obsoleta precisamente porque no lleva el epoch del miembro y deja abierta la ventana del zombi durante un rebalanceo.

Lo que read_committed le hace a tu latencia

Un consumidor con isolation.level=read_committed no lee más allá del Last Stable Offset: el offset del primer mensaje de la transacción abierta más antigua de esa partición. Mientras haya una transacción sin cerrar, todo lo posterior es invisible, aunque pertenezca a otros productores que ya confirmaron.

Esto tiene una consecuencia que sorprende a quien mide por primera vez: la latencia de punta a punta de una cadena transaccional no es la latencia de tu transacción, es la de la más lenta de las transacciones concurrentes sobre esa partición. Si tienes un productor que agrupa mil mensajes por transacción cada dos segundos y otro que confirma cada cincuenta milisegundos, el segundo hereda la latencia del primero.

La palanca directa es el tamaño de la transacción. Confirmar cada mensaje da la mínima latencia y el máximo coste: dos escrituras en el log del coordinador y un marcador por partición por mensaje. Agrupar cien o mil mensajes amortiza ese coste fijo y sube la latencia en la misma proporción. El punto razonable para la mayoría de los pipelines está en confirmar por lote de consumo, que es exactamente lo que hace el bucle de

la sección anterior.

La segunda palanca es transaction.timeout.ms, que por defecto es un minuto y está acotado por transaction.max.timeout.ms en el broker —quince minutos por defecto—. Es el tiempo que el coordinador espera antes de abortar por su cuenta una transacción que nadie cerró. Subirlo protege a los procesos lentos y alarga el bloqueo del LSO cuando algo se cuelga; bajarlo hace lo contrario. La regla: debe ser mayor que el peor caso de tu procesamiento por transacción, con margen, y ni un minuto más.

Y una nota sobre memoria del lado del consumidor: para filtrar los mensajes de transacciones abortadas, el consumidor los recibe y los descarta usando un índice de transacciones abortadas que el broker mantiene junto al segmento. Si tu aplicación aborta mucho, pagas ancho de banda por mensajes que nunca verás. Un ratio de abortos alto no es sólo una señal de errores: es también una factura.

Kafka Connect: exactly-once sin escribir un productor

Desde KIP-618, los conectores de origen de Kafka Connect pueden dar exactly-once sin que tú toques la API de transacciones. El worker envuelve la escritura de registros y la de offsets en la misma transacción, con lo que desaparece el hueco clásico en el que un conector reiniciado reenvía lo que ya había publicado.

```
# worker.properties  (modo distribuido)
exactly.once.source.support=enabled
# El worker necesita un transactional.id derivado y estable:
transaction.boundary=poll        # o 'interval' / 'connector'
offset.flush.interval.ms=10000
```

Tres advertencias que la documentación menciona de pasada y que cuestan una tarde. La activación es un cambio de protocolo entre workers: hay que hacer un rolling restart en dos fases, primero con exactly.once.source.support=preparing en todo el cluster y sólo después con enabled; hacerlo de golpe deja el cluster sin poder elegir líder de forma consistente. El principal administrativo del conector necesita permisos adicionales sobre el topic de offsets, porque el worker tiene que leer hasta el final incluso cuando hay transacciones abiertas. Y transaction.boundary decide el compromiso: poll confirma en cada tanda que devuelve el conector —latencia baja, más marcadores—, mientras que interval agrupa por tiempo.

En el lado de los conectores de destino la historia es distinta y conviene decirla sin adornos: no existe un interruptor equivalente. Un conector de destino consume con read_committed y, a partir de ahí, la garantía depende de si el sistema de destino acepta escrituras idempotentes. Un sink a PostgreSQL con INSERT ... ON CONFLICT DO UPDATE sobre una clave derivada del mensaje es exactly-once en la práctica; el mismo sink con INSERT a secas es at-least-once con duplicados.

Flink y el commit en dos fases

El conector de Kafka de Flink implementa exactly-once atando las transacciones de Kafka a los checkpoints: abre una transacción por checkpoint, escribe, y la confirma cuando el checkpoint se completa. Es elegante y tiene una trampa que produce pérdida de datos silenciosa.

```
KafkaSink<Evento> sink = KafkaSink.<Evento>builder()
    .setBootstrapServers(brokers)
    .setDeliveryGuarantee(DeliveryGuarantee.EXACTLY_ONCE)
    // Único por job: dos jobs con el mismo prefijo se hacen fencing
    // el uno al otro y uno de los dos deja de escribir.
    .setTransactionalIdPrefix("pagos-enriquecidos-v3")
    .setKafkaProducerConfig(props)
```

```
    .setRecordSerializer(serializer)
    .build();
```

La trampa es el tiempo. Si transaction.timeout.ms es menor que la duración máxima de un checkpoint más el tiempo de reinicio del job, Kafka aborta la transacción por su cuenta; cuando Flink recibe la notificación de checkpoint completado e intenta confirmar, la transacción ya no existe y esos registros se pierden sin ningún error visible. Flink pone por defecto una hora, y el broker rechaza cualquier valor por encima de su transaction.max.timeout.ms, así que hay que subir también el del broker. La otra trampa es el prefijo: el identificador de transacción es prefijo más identificador de checkpoint, de modo que dos jobs distintos con el mismo prefijo generan colisiones y se expulsan mutuamente.

Cuando la garantía sale de Kafka

El punto más importante de todo este tema es también el más incómodo: exactly-once de Kafka es exactly-once entre topics de Kafka. En cuanto una de las dos puntas es otra cosa —una base de datos, una API de terceros, un correo, un pago— la transacción de Kafka no puede incluirla, y la única salida es que la operación externa sea idempotente.

Los tres patrones que resuelven esto en la práctica:

- Tabla inbox. El consumidor escribe, en la misma transacción de base de datos que el efecto de negocio, una fila con una clave derivada del mensaje —topic, partición y offset, o un identificador de negocio— con restricción de unicidad. Si el mensaje se reprocesa, la inserción choca y el trabajo se salta. Barato, robusto y aburrido, que es lo que se busca.
- Outbox transaccional. Para el camino inverso: en lugar de escribir en la base de datos y publicar en Kafka —dos sistemas, ninguna atomicidad—, escribes el evento en una tabla de salida dentro de la misma transacción y un proceso aparte lo publica. Con CDC, ese proceso es Debezium.
- Escrituras naturalmente idempotentes. Un UPSERT por clave de negocio, un SET saldo = 500 en lugar de un saldo = saldo + 50, una llamada externa con cabecera Idempotency-Key. Cuando se puede diseñar así, no hace falta ninguna tabla auxiliar.

```
-- Inbox: la restricción de unicidad hace todo el trabajo
CREATE TABLE eventos_procesados (
    topic        text    NOT NULL,
    particion    int     NOT NULL,
    "offset"     bigint  NOT NULL,
    procesado_en timestamptz NOT NULL DEFAULT now(),
    PRIMARY KEY (topic, particion, "offset")
);

-- En la misma transacción que el efecto de negocio:
BEGIN;
  INSERT INTO eventos_procesados (topic, particion, "offset")
  VALUES ('pagos', 3, 918273)
  ON CONFLICT DO NOTHING;
  -- Si no insertó ninguna fila, este mensaje ya se procesó: ROLLBACK y seguir.
  UPDATE cuentas SET saldo = saldo - 4200 WHERE id = 'c-991';
COMMIT;
```

Una nota de operación sobre esta tabla: crece para siempre si nadie la poda. Particionarla por día y desprender particiones más viejas que tu ventana de retención de Kafka es la forma sana; borrar filas con un DELETE masivo cada noche genera exactamente el bloat que otro artículo de este sitio trata con detalle.

Multi-cluster: la garantía no cruza la frontera

MirrorMaker 2 replica entre clusters y no propaga transacciones. Los marcadores de transacción no se replican como tales, y el cluster de destino no tiene el estado del coordinador de origen, así que un consumidor read_committed en destino no está leyendo las mismas fronteras transaccionales que en origen. En la práctica, la replicación entre clusters es at-least-once con offsets traducidos por el RemoteClusterUtils, y cualquier consumidor del destino debe ser idempotente por su cuenta.

Esto es importante porque rompe una suposición muy común en arquitecturas activo-activo: si tu plan de recuperación ante desastres consiste en que los consumidores continúen en el cluster secundario, el exactly-once que tanto cuidas en el primario no viaja con ellos. La respuesta razonable no es intentar arreglarlo dentro de Kafka, es diseñar el consumidor de destino con una tabla inbox y dejar de depender de la garantía del transporte.

Observabilidad: las cinco métricas que importan

Las transacciones fallan de formas que no producen excepciones en tu código, así que hay que mirarlas desde fuera.

- Ratio de abortos. Del productor, txn-abort-rate frente a txn-commit-rate. Una subida sostenida casi siempre es un timeout de transacción por procesamiento lento, no un bug de negocio.
- Distancia entre LSO y final del log. La métrica de veras diagnóstica. Si crece de forma monótona en una partición, tienes una transacción colgada; el consumidor de esa partición se está quedando atrás sin que su propio lag lo explique.
- Duración de la transacción. Percentil 99 del tiempo entre beginTransaction y commitTransaction. Es el número que hay que comparar con transaction.timeout.ms; si el p99 está por encima de la mitad del timeout, vas a tener abortos en la próxima punta de carga.
- ProducerFencedException por minuto. Cero es lo normal. Un goteo constante significa que hay dos procesos peleando por el mismo transactional.id, lo que casi siempre es un despliegue que no termina de drenar el anterior.
- Tamaño de __transaction_state. Crece con el número de transactional.id distintos que has usado, y sus entradas caducan según transactional.id.expiration.ms —siete días por defecto—. Si generas identificadores aleatorios, este topic es el que te lo va a decir.

```
groups:
  - name: kafka-eos
    rules:
      - alert: TransaccionColgada
        # El LSO no avanza pero el log sí: hay una transacción sin cerrar
        expr: |
          (kafka_log_log_end_offset - on(topic, partition)
           kafka_log_last_stable_offset) > 0
          and
          delta(kafka_log_last_stable_offset[10m]) == 0
        for: 10m
        labels: { severity: critical }
        annotations:
          summary: "LSO bloqueado en {{ $labels.topic }}-{{ $labels.partition }}"
          runbook: "https://runbooks.internal/kafka/transaccion-colgada"

      - alert: AbortosDeTransaccionAltos
        expr: |
          rate(kafka_producer_txn_abort_total[5m])
          / clamp_min(rate(kafka_producer_txn_commit_total[5m]), 0.001) > 0.05
```

```
        for: 15m
        labels: { severity: warning }
```

Cómo se prueba esto de verdad

Un test que publica tres mensajes y comprueba que llegan tres no demuestra exactly-once: demuestra que el camino feliz funciona. La garantía sólo se demuestra rompiendo cosas a propósito, y con Testcontainers se hace en un test de integración normal.

```
import pytest, uuid
from testcontainers.kafka import KafkaContainer

def test_no_duplica_al_morir_entre_escritura_y_commit(kafka, spawn_worker):
    """El worker escribe, y lo matamos ANTES de commitTransaction.
    Al reiniciar debe reprocesar desde el offset anterior y no duplicar."""
    publicar(kafka, "entrada", [f"pago-{i}" for i in range(500)])

    w = spawn_worker(transactional_id="pagos-0", kill_before_commit_at=250)
    w.wait_until_dead()                     # muere a mitad del lote

    w2 = spawn_worker(transactional_id="pagos-0")   # mismo id: hace fencing
    w2.wait_until_idle()

    salida = leer_todo(kafka, "salida", isolation_level="read_committed")
    assert len(salida) == 500
    assert len(set(salida)) == 500          # ni uno repetido

def test_el_zombi_no_puede_escribir(kafka, spawn_worker):
    """Dos procesos con el mismo transactional.id: el viejo debe morir."""
    viejo = spawn_worker(transactional_id="pagos-0", pause_after_begin=True)
    nuevo = spawn_worker(transactional_id="pagos-0")
    nuevo.wait_until_idle()

    with pytest.raises(ProducerFenced):
        viejo.resume_and_commit()
```

Hay tres fallos que sólo aparecen con inyección de fallos y que conviene tener en la batería: matar el proceso entre la escritura y el commit —el test de arriba—, matar al coordinador de transacciones a mitad de vuelo para comprobar que el nuevo termina el two-phase commit, y provocar una pausa artificial más larga que transaction.timeout.ms para verificar que tu código trata ProducerFencedException cerrando el productor en lugar de reintentando en bucle.

Sobre ese último punto, la distinción que más código malo produce: ProducerFencedException, OutOfOrderSequenceException y UnsupportedVersionException son fatales —hay que cerrar el productor y terminar el proceso—, mientras que KafkaException genérica en un send dentro de la transacción se resuelve abortando y reintentando la transacción entera. Tratar un error fatal como reintentable es la receta exacta para duplicar.

Un caso completo, con cifras

Un pipeline de conciliación de pagos: lee de un topic con veinticuatro particiones, enriquece contra una base de datos y escribe en dos topics de salida, uno de movimientos conciliados y otro de excepciones. Antes: at-least-once con deduplicación por clave en el consumidor final. El problema no era la corrección, era que la deduplicación vivía en el último eslabón y tres equipos distintos consumían el topic intermedio, cada uno con su propia idea de qué era un duplicado.

La migración a transacciones tomó dos semanas y el trabajo real no estuvo en el código del productor. Estuvo en tres sitios. El primero, el transactional.id: la aplicación corría como Deployment con nombres de pod aleatorios, así que hubo que pasarla a StatefulSet para tener identificadores estables. El segundo, el timeout: el enriquecimiento contra la base de datos tenía un p99 de once segundos en hora punta, muy cerca del minuto por defecto una vez sumado el lote completo, y se subió a tres minutos tras subir también transaction.max.timeout.ms en los brokers. El tercero, los consumidores: los tres equipos tuvieron que pasar a read_committed, y uno de ellos descubrió que su cliente de Go usaba una versión de la librería anterior al soporte de transacciones v2.

Los números después de tres meses: la latencia mediana de punta a punta subió de 240 ms a 310 ms, y el p99 de 1,9 s a 2,6 s, todo ello por el tamaño de lote de la transacción y el bloqueo del LSO. El caudal bajó un 9% con el mismo hardware. A cambio, los tres consumidores borraron su lógica de deduplicación —unas cuatrocientas líneas entre los tres, con dos tablas de Redis que ya nadie mantiene— y el número de incidentes de «pago aplicado dos veces» pasó de una media de 1,3 al mes a cero en noventa días.

Hubo un incidente durante el periodo, y merece contarse porque no fue de Kafka. Un despliegue con terminationGracePeriodSeconds demasiado corto mató pods a mitad de transacción; las transacciones quedaron abiertas hasta que el coordinador las abortó por timeout, y durante esos tres minutos el LSO de varias particiones no avanzó. Los consumidores no perdieron nada, pero se retrasaron lo suficiente como para disparar las alertas de lag y para que alguien pensara, durante diez minutos, que había pérdida de datos. La corrección fue un apagado limpio que cierra la transacción en curso antes de salir.

Cuándo no usar exactly-once

Conviene cerrar con el caso contrario, porque el coste es real y no siempre compensa.

- Cuando el efecto ya es idempotente. Si escribes en un almacén clave-valor con SET por clave de negocio, los duplicados no hacen daño. Pagar latencia y marcadores por una garantía que no necesitas es puro gasto.
- Cuando la latencia es el producto. En una cadena donde cada milisegundo cuenta, el bloqueo del LSO y el coste del commit son difíciles de justificar frente a una deduplicación barata en el destino.
- Cuando el destino no es Kafka. Ya está dicho arriba, pero merece repetirse: si el final de la cadena es una base de datos o una API, la transacción de Kafka no te salva y vas a acabar escribiendo la tabla inbox igualmente. En ese caso, plantéate si la complejidad de las transacciones te aporta algo por encima de at-least-once más idempotencia.
- Cuando no puedes garantizar identificadores estables. Si tu plataforma no te deja tener transactional.id estables entre reinicios, no tienes fencing, y sin fencing las transacciones dan una sensación de seguridad sin la seguridad. Es peor que no usarlas.

La pregunta correcta nunca es «¿quiero exactly-once?» —todo el mundo lo quiere—. Es «¿dónde está el último punto de mi cadena donde un duplicado hace daño, y cuál es la forma más barata de hacerlo inofensivo justo ahí?». Muchas veces la respuesta es una restricción de unicidad en una tabla, y esa respuesta es perfectamente respetable.

Activar transaction.version=2 sin sorpresas

El protocolo endurecido de KIP-890 no se activa solo al actualizar los brokers: es una feature del cluster que hay que subir explícitamente, y el orden importa.

```
# 1. Ver en qué versión está el cluster
kafka-features.sh --bootstrap-server broker:9092 describe

# 2. Subirla sólo cuando TODOS los brokers están ya en la versión nueva
kafka-features.sh --bootstrap-server broker:9092 \
  upgrade --feature transaction.version=2
```

Tres cosas que conviene saber antes de ejecutar el segundo comando. La primera: los clientes antiguos siguen funcionando; el protocolo negocia versión por conexión, así que un productor viejo hablará el protocolo viejo y seguirá expuesto al problema de las transacciones colgadas hasta que se actualice. La segunda: el beneficio real es que el epoch del productor se incrementa en cada transacción, no sólo al inicializar, y eso cierra la ventana en la que un paquete retrasado de una transacción ya abortada aterrizaba dentro de la siguiente y la contaminaba. La tercera, y la que más sorprende: no todos los clientes de terceros implementan la versión 2. Antes de subir la feature conviene inventariar qué librerías hay en el parque —los clientes de Go y de Rust han ido por detrás del cliente de Java— porque un cliente que no la soporta no falla al arrancar, simplemente sigue con el protocolo antiguo y no obtienes la protección para el que más la necesita.

Una vez activada, la utilidad kafka-transactions.sh deja de ser un último recurso incómodo y pasa a ser una herramienta de diagnóstico normal: describe-producers sobre la partición bloqueada te da el PID y el epoch del culpable, y abort con esos datos desbloquea el LSO sin tocar los datos confirmados.

Cuatro preguntas que siempre salen después

¿Las transacciones funcionan entre topics de clusters distintos? No. Una transacción vive en un coordinador de un cluster concreto y sus marcadores se escriben en particiones de ese mismo cluster. Escribir atómicamente en dos clusters no es algo que Kafka ofrezca; el patrón habitual es confirmar en el cluster de origen y replicar después, asumiendo at-least-once en el salto.

¿Puedo mezclar productores transaccionales y no transaccionales en el mismo topic? Sí, y funciona. Los mensajes no transaccionales son visibles de inmediato para cualquier consumidor. Lo que hay que tener presente es que un consumidor read_committed de ese topic seguirá bloqueándose detrás del LSO cuando haya una transacción abierta, aunque los mensajes que le interesan sean los no transaccionales que están más adelante en el log.

¿La compactación de logs y las transacciones se llevan bien? Con matices. Los marcadores de transacción se conservan durante transactional.id.expiration.ms y después son candidatos a limpieza, pero un consumidor que arranque desde el principio de un topic compactado muy antiguo puede encontrarse con mensajes cuyos marcadores ya no están. En la práctica no rompe nada porque el mensaje ya está confirmado, pero explica por qué las cuentas de mensajes no cuadran al recontar un topic compactado.

¿Qué pasa si subo transaction.timeout.ms muy alto «por si acaso»? Que conviertes una transacción colgada en una parada larga. Un timeout de quince minutos significa que, si un proceso muere justo después de escribir y antes de confirmar, los consumidores de esas particiones se quedan quince minutos sin ver nada nuevo. El timeout no es un margen de seguridad gratuito: es el tiempo máximo que aceptas tener la partición bloqueada.

Exactly-Once in Kafka: The Idempotent Producer, Transactions, read_committed, and the Boundary Everyone Crosses

Three guarantees, and only one people actually want

When someone says their Kafka pipeline is exactly-once, they usually mean one of two things: they haven't seen duplicates yet, or they set processing.guarantee=exactly_once_v2 in a config file and considered the matter closed. Neither is a guarantee.

The three real options are:

- At-most-once: commit the offset before processing. If the process dies in between, the message is gone and nobody ever looks at it again. Cheap and fast; unacceptable for a payment.
- At-least-once: process, then commit. If the process dies in between, the message is reprocessed. This is the default behaviour of nearly everything, and it is perfectly correct as long as your processing is idempotent.
- Exactly-once: each input record affects the result exactly once. It does not mean the message is physically delivered once — that is impossible over an asynchronous network — it means the effect is applied once.

That last distinction decides whether your design works. Kafka does not stop a record from being delivered twice; it stops it from being counted twice, and only inside a very specific perimeter: Kafka to Kafka. The moment your consumer issues a POST to a payment provider or an INSERT into PostgreSQL, you have left that perimeter and the guarantee no longer applies. I'll repeat this later with an example, because it's the mistake that costs the most money.

Two kinds of duplicate, and they are not the same problem

Before touching any configuration, it's worth separating two sources of duplication that people constantly conflate.

The producer-side duplicate. You send a ProduceRequest, the broker writes it to the log, and the response is lost to a network blip or a timeout. The client has no way to tell "it never arrived" from "it arrived and I never heard back", so it retries. Result: the same record twice in the partition, at two different offsets. This happens below your code; no application-level duplicate check ever sees it coming.

The consumer-side duplicate. You read a batch, process it, write the output, and die before committing the offset. Another group member picks up the partition from the last committed offset and processes the same records again. The output is duplicated even though the input was perfectly deduplicated.

Kafka solves the first with the idempotent producer and the second with transactions. Two distinct mechanisms with very different costs — and one of them is already switched on in your cluster whether you know it or not.

Layer 1: the idempotent producer

Since Kafka 3.0, enable.idempotence defaults to true. The mechanism is simple and elegant: on startup the producer asks the broker for a producer ID (PID). From then on, every batch it sends carries the PID, an epoch, and a monotonic sequence number per partition. Each partition leader remembers the sequence number of the last batch it accepted for that PID and rejects any batch whose number it has already seen, returning success to the client without writing anything.

It's broker-side deduplication, on the data path, invisible to your application. Enabling it forces three settings:

```
acks=all
retries=2147483647
max.in.flight.requests.per.connection=5   # at most 5
```

The five-in-flight limit is not arbitrary: the broker only keeps metadata for the last five batches per PID and partition, which is exactly what it needs to reorder and deduplicate without losing ordering. If you raise that value with idempotence enabled, the client fails at startup rather than handing you a half-guarantee — which is the right behaviour.

One exception worth knowing: setting acks=1 or acks=0 together with enable.idempotence=true makes the producer throw a ConfigException. Plenty of teams carry an acks=1 forward from a 2019 tutorial and discover their application no longer starts after an upgrade. The right fix is almost never to disable idempotence.

What the idempotent producer does not cover

Three limits, and they matter:

- A single producer session. If the process restarts it gets a fresh PID and the counters reset to zero. A record sent before the restart and re-sent by your application logic afterwards is a perfectly legitimate duplicate as far as the broker is concerned.
- One partition at a time. There is no atomicity across partitions or topics. Write to three topics, die after the second, and you have two and a half writes.
- Limited state retention. If a producer goes idle for longer than the broker keeps its state (transactional.id.expiration.ms and PID expiry), you can get an OutOfOrderSequenceException. That's not your bug: it's the broker saying it lost the thread and can no longer guarantee anything. You handle it by recreating the producer.

Layer 2: transactions

Transactions, introduced by KIP-98 in 0.11, add what's missing: atomicity across multiple partitions and topics, and between writing results and committing consumer offsets. That second half is the key piece, because it turns the classic consume-transform-produce loop into an all-or-nothing operation.

The mechanics, from above:

- You configure a transactional.id. It is the stable identifier of a logical instance of your application, and its stability across restarts is what makes everything else work.
- initTransactions() registers that id with the transaction coordinator (the broker leading the relevant partition of the internal __transaction_state topic), bumps the associated epoch, and aborts any transaction left pending by a previous incarnation of the process. This is zombie fencing: an old process still alive on another node, holding the old epoch, gets a ProducerFencedException on its next write and can no longer

corrupt anything.

- Between beginTransaction() and commitTransaction(), everything you write — to any partition of any topic — belongs to the same transaction.
- sendOffsetsToTransaction(...) writes consumer offsets into __consumer_offsets inside that same transaction. This is the detail that makes the guarantee real.
- On commit, the coordinator writes a control marker (commit or abort) into every partition involved. Consumers running read_committed use those markers to decide what to deliver.

The full loop, in Java

```
Properties cp = new Properties();
cp.put("bootstrap.servers", "kafka:9092");
cp.put("group.id", "order-enricher");
cp.put("enable.auto.commit", "false");        // mandatory
cp.put("isolation.level", "read_committed");  // don't read uncommitted data
cp.put("key.deserializer", StringDeserializer.class.getName());
cp.put("value.deserializer", StringDeserializer.class.getName());

Properties pp = new Properties();
pp.put("bootstrap.servers", "kafka:9092");
// Stable across restarts. Here: app name + pod ordinal.
pp.put("transactional.id", "order-enricher-" + podOrdinal());
pp.put("enable.idempotence", "true");
pp.put("transaction.timeout.ms", "60000");
pp.put("key.serializer", StringSerializer.class.getName());
pp.put("value.serializer", StringSerializer.class.getName());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(cp);
KafkaProducer<String, String> producer = new KafkaProducer<>(pp);

producer.initTransactions();   // fencing + abort orphaned transactions
consumer.subscribe(List.of("orders"));

try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(200));
        if (records.isEmpty()) continue;

        producer.beginTransaction();
        try {
            for (ConsumerRecord<String, String> r : records) {
                String output = enrich(r.value());   // pure function, no external effects
                producer.send(new ProducerRecord<>("orders-enriched", r.key(), output));
            }

            Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();
            for (TopicPartition tp : records.partitions()) {
                List<ConsumerRecord<String, String>> forPartition = records.records(tp);
                long last = forPartition.get(forPartition.size() - 1).offset();
                offsets.put(tp, new OffsetAndMetadata(last + 1));   // +1: next to read
            }
            // The groupMetadata() overload is what enables EOS v2.
            producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata());

            producer.commitTransaction();
        } catch (ProducerFencedException | OutOfOrderSequenceException
                 | AuthorizationException e) {
            // Unrecoverable: another process has replaced us. Exit.
            producer.close();
            throw e;
        } catch (KafkaException e) {
            producer.abortTransaction();
            // The consumer must rewind: offsets were never committed.
```

```
            for (TopicPartition tp : records.partitions()) {
                consumer.seek(tp, records.records(tp).get(0).offset());
            }
        }
    }
} finally {
    consumer.close();
    producer.close();
}
```

Three details in that code that nearly every implementation I've reviewed gets wrong:

The seek after an abort. Aborting discards the writes and the offsets, but it does not move the consumer's in-memory position. Without that seek, the next poll() returns the following records and you have just skipped an entire batch without processing it. It's silent data loss caused by a mechanism designed to prevent data loss.

The +1 on the offset. Kafka commits "the next offset I want to read", not "the last one I processed". Get it wrong and you reprocess one record per partition per commit, forever.

consumer.groupMetadata(). The old overload took only the consumerGroupId. The one taking ConsumerGroupMetadata also carries the member id and generation epoch, which lets the coordinator fence a zombie group member. That's the core of what was called EOS v2, and it's why you no longer need one transactional.id per input partition.

The same pattern in Python

The confluent-kafka library exposes the same transactional API on top of librdkafka:

```
from confluent_kafka import Consumer, Producer, KafkaException, TopicPartition

consumer = Consumer({
    "bootstrap.servers": "kafka:9092",
    "group.id": "order-enricher",
    "enable.auto.commit": False,
    "isolation.level": "read_committed",
    "auto.offset.reset": "earliest",
})

producer = Producer({
    "bootstrap.servers": "kafka:9092",
    "transactional.id": f"order-enricher-{pod_ordinal()}",
    "enable.idempotence": True,
    "transaction.timeout.ms": 60000,
})

producer.init_transactions()
consumer.subscribe(["orders"])

while True:
    msgs = consumer.consume(num_messages=500, timeout=1.0)
    msgs = [m for m in msgs if m is not None and not m.error()]
    if not msgs:
        continue

    producer.begin_transaction()
    try:
        for m in msgs:
            producer.produce("orders-enriched", key=m.key(),
                             value=enrich(m.value()))

        # Next offset to read, per partition.
        last = {}
```

```
        for m in msgs:
            key = (m.topic(), m.partition())
            last[key] = max(last.get(key, -1), m.offset())
        offsets = [TopicPartition(t, p, o + 1) for (t, p), o in last.items()]

        producer.send_offsets_to_transaction(offsets, consumer.consumer_group_metadata())
        producer.commit_transaction()

    except KafkaException as e:
        err = e.args[0]
        if err.txn_requires_abort():
            producer.abort_transaction()
            for tp in consumer.assignment():
                consumer.seek(TopicPartition(tp.topic, tp.partition,
                                             first_offset_of_batch(msgs, tp)))
        elif err.retriable():
            continue          # retry the same operation
        else:
            raise             # fatal: shut the process down
```

Note err.txn_requires_abort() and err.retriable(). librdkafka sorts transactional errors into three buckets — retriable, requires-abort, and fatal — and treating them identically is the fastest way to build a producer that either hangs or aborts when it shouldn't.

read_committed, the LSO, and the offsets that "go missing"

A consumer with isolation.level=read_committed never delivers records beyond the Last Stable Offset (LSO): the offset of the first still-open transaction in that partition. Everything after it exists in the log, is replicated and is safe, but stays invisible until that transaction commits or aborts.

Two surprising behaviours follow from this:

Offsets are not contiguous. Commit and abort markers occupy an offset in the partition. If you process offsets 100 through 104 in one transaction, the marker takes 105 and your consumer sees the next real record at 106. Any monitoring that computes progress as "last consumed minus first" or assumes contiguity will be wrong — always slightly, always in a way that's painful to debug.

Aborted messages are filtered client-side. The broker sends consumers the full batch along with an index of aborted transactions; the client does the discarding. That means your fetch bandwidth includes data you will never see. If your abort rate is high you're paying for network traffic that goes nowhere, and it shows up as throughput that doesn't reconcile with the number of records processed.

And the most important operational consequence: a hanging transaction freezes the LSO. Lag on your read_committed consumers climbs without limit while the log-end-offset advances, and there isn't a single error in the application logs. More on this in the operations section.

Kafka Streams: the one-line version

If your processing lives in Kafka Streams, all of the above is implemented for you:

```
processing.guarantee=exactly_once_v2
```

Streams manages the transactional.id, includes writes to its state stores' changelog topics in the transaction, and commits offsets inside it. The older exactly_once and exactly_once_beta values were removed in Kafka 4.0; if you're migrating from an old version, that's the change.

The parameter that actually matters here is commit.interval.ms, which drops to a 100 ms default once EOS is on. Every commit is a transaction, and every transaction has a fixed cost. Raising it to 1000 ms often multiplies throughput at the price of end-to-end latency: read_committed consumers see nothing until the transaction closes. It's the one knob that genuinely matters and almost nobody touches it.

What changed with KIP-890 and transaction.version=2

The original transactional protocol had a known hole. When a producer suffered a long pause — a GC, cgroup throttling, a network partition — and the coordinator aborted its transaction on timeout, that producer could wake up and keep sending records with the same PID and the same epoch. Those records landed after the abort marker and got attributed to the next transaction. Two pathologies followed: messages leaking from one transaction into another, and hanging transactions that blocked the LSO indefinitely.

KIP-890 fixes it at the root. With transaction.version=2, available from Kafka 4.0, the producer epoch is bumped on every commit and every abort. The (PID, epoch) pair uniquely identifies one specific transaction, so a late record from the previous one arrives with a stale epoch and the broker rejects it unambiguously. Verification also folds into existing requests instead of requiring an extra per-partition round trip, which takes overhead off the data path.

KIP-890 also introduces TransactionAbortableException, and it's a real practical improvement: it explicitly marks the errors where the correct response is to abort and retry the batch, instead of forcing you to infer it from the exception type. If your error handling is a blanket catch (KafkaException e), this is your excuse to rewrite it.

Transaction version is a cluster feature, not a client property. You inspect and enable it with the features tool:

```
# Show active feature versions
bin/kafka-features.sh --bootstrap-server kafka:9092 describe

# Enable transactions v2 (requires up-to-date clients)
bin/kafka-features.sh --bootstrap-server kafka:9092 \
  upgrade --feature transaction.version=2
```

Check your clients before enabling it. Those that don't speak the new protocol keep working with the old semantics, but you don't get the protection. Support took a while to land outside the Java ecosystem; if you use Sarama, kafka-go, or a librdkafka binding, verify the specific version before assuming you're covered.

The boundary everyone crosses without noticing

Here is the mistake that justifies this whole article. This code looks exactly-once:

```
producer.beginTransaction();
for (ConsumerRecord<String, String> r : records) {
        // &þ  This is NOT part of the Kafka transaction.
    paymentGateway.charge(r.value());
    producer.send(new ProducerRecord<>("payments-made", r.key(), r.value()));
}
producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata());
producer.commitTransaction();
```

If the process dies just before commitTransaction(), Kafka does exactly what it promises: it discards the payments-made event and discards the offsets. The batch is reprocessed. And the customer gets charged a second time, because paymentGateway.charge() happened outside the transactional perimeter and nothing

can undo it.

Kafka transactions are a Kafka-to-Kafka mechanism. Full stop. For any external effect you have three honest options:

- Make the external operation idempotent. Derive an idempotency key deterministically from the record — topic-partition-offset, say, or the business identifier if one exists — and let the remote system deduplicate on it. Every serious payment gateway supports this.
- Move the transactional boundary to the sink. Write the result and the consumed offset in the same database transaction, with an UPSERT keyed appropriately. The Kafka offset then becomes a startup optimisation and the truth lives in your database.
- The transactional outbox, in reverse. If you produce the external effect yourself, write it as a row inside your business transaction and let a relay publish it to Kafka with at-least-once plus deduplication downstream.

None of the three needs Kafka transactions. That's precisely the conclusion many people don't want to hear: in a pipeline that terminates in a database, Kafka transactions are usually the wrong mechanism, and an idempotent UPSERT solves the problem at a fraction of the operational cost.

Eight mistakes you'll see in production

- A random transactional.id on every startup. UUID.randomUUID() looks harmless and destroys the guarantee: without a stable id there is no fencing, and a zombie process can keep writing. You also leave one id behind per restart, each with its own coordinator state, until transactional.id.expiration.ms cleans it up. On Kubernetes use the StatefulSet ordinal; with a Deployment, derive the id from a stable assignment, not from the pod name.
- Committing offsets with consumer.commitSync(). It breaks atomicity: offsets leave the transaction and you're back to at-least-once with more complexity and more latency. A commitSync inside a transactional loop is a bug.
- Leaving enable.auto.commit=true. Same problem, worse because it's invisible: a background thread commits offsets on its own schedule, possibly before the transaction closes.
- Not calling seek after an abort. Silent skip of a whole batch, as described above. It's the hardest failure on this list to detect because it produces no error at all.
- transaction.timeout.ms above the broker maximum. Exceed transaction.max.timeout.ms and the broker rejects the producer at initTransactions() with InvalidTxnTimeoutException. Set it too low and a GC pause gets a transaction aborted that your process still believes is alive.
- Sharing a transactional producer across threads. A transactional producer holds one in-flight transaction. Two threads calling beginTransaction() produce chaos that shows up as intermittent IllegalStateExceptions in production and never locally.
- Retrying after ProducerFencedException. That exception means another process has legitimately taken over your identity. Retrying is exactly the wrong move: close the producer and exit so the orchestrator replaces you.
- Single-message transactions. A transaction has a fixed cost: two coordinator writes plus one marker per partition. Amortised over five hundred messages it's noise; over one, it can multiply per-record cost tenfold. Batch per poll(), not per message.

Operations: the hanging transaction

The signature EOS incident doesn't look like an incident. The symptom: consumer group lag climbs linearly, the topic is receiving normal traffic, producers report no errors, consumers report no errors, and restarting changes nothing. What's happening is that a transaction was left open and the LSO isn't advancing.

Kafka has shipped a dedicated tool for this since 3.0:

```
# Find hanging transactions on a topic
bin/kafka-transactions.sh --bootstrap-server kafka:9092 \
  find-hanging --topic orders-enriched

# Inspect the state of a specific transactional.id
bin/kafka-transactions.sh --bootstrap-server kafka:9092 \
  describe --transactional-id order-enricher-0

# Abort a hanging transaction (last resort, using find-hanging output)
bin/kafka-transactions.sh --bootstrap-server kafka:9092 \
  abort --topic orders-enriched --partition 3 --start-offset 84211
```

A manual abort discards that transaction's records permanently. Run it knowing what you're throwing away, and put that warning on the first line of the runbook entry.

To catch it before someone files a ticket, the useful alert isn't on lag — it's on the gap between log-end-offset and LSO. If that difference grows steadily for longer than your transaction.timeout.ms, you have a stuck transaction, whatever consumer lag happens to say.

On the client, the producer-metrics group exposes timings for the transactional operations — txn-init-time-ns-avg, txn-begin-time-ns-avg, txn-send-offsets-time-ns-avg, txn-commit-time-ns-avg and txn-abort-time-ns-avg. The one with the best information-per-euro ratio is abort: when your abort rate goes from anecdotal to constant, something changed, and it's usually a slow dependency pushing processing past the timeout.

The cost, stated honestly

Transaction overhead isn't a constant you can quote; it depends almost entirely on how many messages you put in each one. With large transactions and a generous commit.interval.ms, the throughput penalty is small — a handful of percentage points — and usually buried under the noise of the rest of the system. With tiny transactions and commits every 100 ms, losing half your throughput is entirely possible.

Latency is a different story and far less negotiable. Under read_committed, a record isn't visible until its transaction commits, so your end-to-end latency has a floor equal to the commit interval. A five-stage transactional pipeline with one-second commits carries five seconds of latency before it processes anything. Do that arithmetic before choosing the architecture, not after the first complaint.

Production checklist

- enable.idempotence=true on every producer, transactional or not. It's free and removes an entire class of duplicates.
- transactional.id derived from a stable identity (StatefulSet ordinal, assigned partition, fixed slot). Never random, never the ephemeral hostname.
- enable.auto.commit=false and zero calls to commitSync/commitAsync inside the transactional loop.

- isolation.level=read_committed on every consumer reading a transactionally-written topic. The default is read_uncommitted, and that default silently voids the guarantee.
- seek back to the first offset of the batch after every abortTransaction().
- Differentiated error handling: fatal (exit), abortable (abort and retry), retriable (retry the same call).
- transaction.timeout.ms above your worst-case batch processing time, with headroom for a GC pause, and below the broker's transaction.max.timeout.ms.
- An alert on the log-end-offset to LSO gap, in addition to the usual lag alert.
- An integration test that kills the process between send and commit — Testcontainers works well — and asserts the output topic contains no duplicates.
- Every external effect inside the loop documented and protected by its own idempotency key. If you can't do that, the Kafka transaction isn't giving you what you think it is.

FAQ

Do I need transactions if I only produce, never consume? No. The idempotent producer covers retry duplicates, which is your only exposure. Transactions only add value when you need atomicity across partitions, or between writes and offsets.

Does EOS work between two Kafka clusters? No, and this one catches people out. The transaction coordinator is a per-cluster component; a transaction cannot span two. With MirrorMaker 2 or cross-region replication the guarantee is at-least-once and you need deduplication at the destination.

What if I use read_uncommitted against a transactional topic? You'll see records from aborted transactions as if they were valid, and you'll see them before they commit. It's a perfectly legitimate mode for debugging or for consumers that tolerate junk, but it voids the guarantee completely. Go and check: it's the client default.

Can I mix transactional and non-transactional writes on the same topic? Technically yes, and read_committed consumers deliver the non-transactional ones normally. In practice it muddies reasoning about the LSO and about what is actually committed. Pick one per topic.

Does log compaction respect transactions? Yes; control markers are cleaned up once they're no longer needed, and compaction operates over committed records. It isn't something you configure, but it is something to know before designing on top of a compacted topic.

Glossary

- PID (producer ID): identifier the broker assigns to an idempotent producer session.
- Epoch: counter attached to the PID or transactional.id; bumping it invalidates any earlier producer.
- Fencing: the mechanism by which an older producer is locked out once a new one appears with the same identity and a higher epoch.
- LSO (Last Stable Offset): offset of the first open transaction in a partition; the ceiling of what a read_committed consumer can read.
- Control marker: special commit or abort record the coordinator writes into every partition of a transaction. It occupies an offset.
- Transaction coordinator: the broker responsible for a transactional.id, holding its state in the internal __transaction_state topic.
- EOS v2: protocol revision that uses consumer group metadata for fencing, removing the need for one

producer per input partition.

- Hanging transaction: a transaction left open with neither commit nor abort, freezing the LSO of its partitions.

Takeaways

The idempotent producer is free, it's already on, and you should leave it that way. Transactions are expensive, they solve one specific problem — atomicity across partitions and between results and offsets — and only inside Kafka. Before enabling them, answer one question: does my pipeline end in Kafka, or does it end in a database or an API? If it's the latter, the right answer is almost always at-least-once with an idempotent sink, and you save yourself an entire distributed coordination protocol that you'd also have to operate.

September 20, 2026 expansion: the internals, the edges, and the rest of the ecosystem

Everything above covers the mental model and the consume-process-produce loop. What follows is what you need once the system has been in production for months: how the mechanism works underneath, why a badly chosen transactional.id turns a guarantee into a false sense of safety, what read_committed does to your latency, how Kafka Connect and Flink behave, and which tests prove all of this works before an incident proves it for you.

Underneath: PID, epoch and sequence numbers

Producer idempotence is neither magic nor content-based deduplication. On startup, the producer asks a broker for a producer ID (PID). From then on, every batch it sends carries three things: the PID, an epoch, and a sequence number that increments per partition. The broker keeps, for each (PID, partition) pair, the last sequence number it wrote. If a batch arrives with a number it already has, it discards it and answers success. If one arrives with a number higher than expected, it returns OutOfOrderSequenceException, because that means an intermediate batch was lost and writing this one would break ordering.

Two practical consequences follow, and they explain configuration that people copy without understanding.

First: max.in.flight.requests.per.connection can go up to 5 with idempotence enabled and still guarantee ordering. The number is not arbitrary; it is how many batches per partition the broker keeps in memory so it can reorder ones that arrive out of sequence. Above 5 the broker cannot reorder and the producer rejects the configuration at startup. Before idempotence, any value above 1 with retries enabled could silently reorder messages; that is the historical bug this fixes.

Second: the PID expires. If an idempotent producer does not write to a partition for producer.id.expiration.ms — one hour by default — the broker forgets its sequence state. If it writes again afterwards, it hits UNKNOWN_PRODUCER_ID. Modern clients handle it by requesting a fresh PID, but the symptom that reaches your logs is alarming enough to be worth writing down: on a low-traffic producer with many partitions, seeing it is normal and does not mean data loss.

The epoch is the exclusion mechanism. Every time a producer with the same transactional.id calls initTransactions(), the coordinator bumps the epoch and rejects with ProducerFencedException any write that arrives carrying an older one. That is what kills zombies: if your process froze for a forty-second GC pause, another one took its place, and the first comes back to life still believing it owns the transaction, its writes are

rejected. You do not catch that exception to retry: you catch it to close the producer and exit.

The transaction coordinator and the state log

A transaction does not live on the brokers holding your data partitions. It lives on a transaction coordinator, which is the leader of a partition of the internal __transaction_state topic, chosen by hashing your transactional.id. That topic is what makes the transaction survive a broker failure: it is a replicated log like any other.

The full sequence behind a commitTransaction() has more steps than the method name suggests:

- The producer calls initTransactions() and the coordinator assigns a PID and an epoch, and aborts any transaction the previous holder of that transactional.id left open.
- On the first write to each partition, the producer tells the coordinator via AddPartitionsToTxn. The coordinator records in its log which partitions are part of this transaction; without that it would not know where to write the markers at the end.
- The messages are written to the data partitions as ordinary messages, immediately visible to a read_uncommitted consumer.
- On commit, the coordinator first writes PrepareCommit to its own log — the point of no return — and only then sends each involved partition a transaction marker, a control record saying "everything before this from this PID is committed".
- Once every marker is written, the coordinator records CompleteCommit. If the coordinator dies between the prepare and the markers, the new coordinator reads its log, sees the prepare, and finishes the job. This is a genuine two-phase commit, with the coordinator's log as the decision record.

Understanding this explains why a "hanging" transaction blocks consumers of a partition you may not even care about: that partition's LSO cannot advance past the first message of a transaction with no marker, and that affects everyone reading with read_committed, not just your application.

The transactional.id: the decision almost nobody gets right

A random transactional.id per start — a UUID, say — looks harmless and cancels almost the entire guarantee. Fencing works by comparing epochs of the same transactional.id; if every restart generates a new one, the old process is never considered a zombie and a hung instance can keep writing while the new one is already working. On top of that, the coordinator has no way to know there was an open transaction that needed aborting.

The rule is that the transactional.id must be stable across restarts and unique per unit of work. On Kubernetes, a StatefulSet gives you this for free: the pod name (processor-0, processor-1) is stable and unique, whereas in a Deployment the names change on every rollout and you have to derive it from something else — from the assigned partition index, if the assignment is static.

```
# StatefulSet: the pod name is stable and works as a transactional.id
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: payments-processor
spec:
  serviceName: payments-processor
  replicas: 4
  template:
```

```
    spec:
      containers:
        - name: app
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef: { fieldPath: metadata.name }
            # -> transactional.id = "payments-payments-processor-0"
            - name: KAFKA_TRANSACTIONAL_ID
              value: "payments-$(POD_NAME)"
```

There is a historical question worth closing, because older forum threads say the opposite. Before KIP-447, each transactional.id had to map to a fixed set of input partitions, which in practice forced you to create one producer per input partition: with two hundred partitions you had two hundred producers per instance, each with its own buffers and connections. KIP-447 removed that constraint by passing the consumer group metadata along with the offsets, so fencing can be done with the group member's epoch instead of with a static partition mapping. Since then one producer per instance is enough, and the transactional.id is still needed to abort orphaned transactions when a transaction includes no sendOffsets call at all.

```
// The detail that enables group-based fencing (KIP-447):
// pass groupMetadata(), not the groupId as a string.
producer.sendOffsetsToTransaction(
    offsets,
    consumer.groupMetadata()   // <- not consumer.groupId()
);
producer.commitTransaction();
```

The sendOffsetsToTransaction(offsets, "my-group") overload taking the identifier as a string is deprecated precisely because it does not carry the member epoch and leaves the zombie window open during a rebalance.

What read_committed does to your latency

A consumer with isolation.level=read_committed does not read past the Last Stable Offset: the offset of the first message of the oldest open transaction on that partition. While a transaction is open, everything after it is invisible, even if it belongs to other producers that already committed.

This has a consequence that surprises people the first time they measure it: the end-to-end latency of a transactional chain is not your transaction's latency, it is that of the slowest of the concurrent transactions on that partition. If you have one producer batching a thousand messages per transaction every two seconds and another committing every fifty milliseconds, the second inherits the first one's latency.

The direct lever is transaction size. Committing per message gives minimum latency and maximum cost: two writes to the coordinator's log and one marker per partition per message. Batching a hundred or a thousand messages amortises that fixed cost and raises latency in the same proportion. The reasonable point for most pipelines is to commit per consumed batch, which is exactly what the loop in the earlier section does.

The second lever is transaction.timeout.ms, one minute by default and capped by the broker's transaction.max.timeout.ms — fifteen minutes by default. It is how long the coordinator waits before aborting on its own a transaction nobody closed. Raising it protects slow processes and lengthens the LSO block when something hangs; lowering it does the opposite. The rule: it should be longer than your worst-case processing time per transaction, with margin, and not a minute more.

And a note on consumer-side memory: to filter out messages from aborted transactions, the consumer receives and discards them using an aborted-transaction index the broker keeps alongside the segment. If

your application aborts a lot, you pay bandwidth for messages you will never see. A high abort ratio is not just a sign of errors: it is also a bill.

Kafka Connect: exactly-once without writing a producer

Since KIP-618, Kafka Connect source connectors can provide exactly-once without you touching the transaction API. The worker wraps the record writes and the offset writes in the same transaction, which closes the classic gap where a restarted connector republishes what it had already sent.

```
# worker.properties  (distributed mode)
exactly.once.source.support=enabled
# The worker needs a derived, stable transactional.id:
transaction.boundary=poll        # or 'interval' / 'connector'
offset.flush.interval.ms=10000
```

Three warnings the documentation mentions in passing and that cost an afternoon. Enabling it is a protocol change between workers: you need a two-phase rolling restart, first with exactly.once.source.support=preparing across the cluster and only then with enabled; doing it in one go leaves the cluster unable to elect a leader consistently. The connector's admin principal needs extra permissions on the offsets topic, because the worker has to read to the end even when there are open transactions. And transaction.boundary decides the trade-off: poll commits on every batch the connector returns — low latency, more markers — while interval groups by time.

On the sink connector side the story is different and worth stating plainly: there is no equivalent switch. A sink connector consumes with read_committed and from there the guarantee depends on whether the destination system accepts idempotent writes. A PostgreSQL sink using INSERT ... ON CONFLICT DO UPDATE on a key derived from the message is exactly-once in practice; the same sink with a plain INSERT is at-least-once with duplicates.

Flink and the two-phase commit

Flink's Kafka connector implements exactly-once by tying Kafka transactions to checkpoints: it opens a transaction per checkpoint, writes, and commits when the checkpoint completes. It is elegant, and it has a trap that causes silent data loss.

```
KafkaSink<Event> sink = KafkaSink.<Event>builder()
    .setBootstrapServers(brokers)
    .setDeliveryGuarantee(DeliveryGuarantee.EXACTLY_ONCE)
    // Unique per job: two jobs sharing a prefix fence each other
    // and one of them stops writing.
    .setTransactionalIdPrefix("enriched-payments-v3")
    .setKafkaProducerConfig(props)
    .setRecordSerializer(serializer)
    .build();
```

The trap is timing. If transaction.timeout.ms is shorter than the maximum checkpoint duration plus the job's restart time, Kafka aborts the transaction on its own; when Flink gets the checkpoint-complete notification and tries to commit, the transaction no longer exists and those records are lost with no visible error. Flink defaults to one hour, and the broker rejects any value above its transaction.max.timeout.ms, so you have to raise the broker's too. The other trap is the prefix: the transactional id is prefix plus checkpoint id, so two different jobs with the same prefix generate collisions and fence each other out.

When the guarantee leaves Kafka

The most important point in this whole topic is also the most uncomfortable: Kafka's exactly-once is exactly-once between Kafka topics. As soon as one of the two ends is something else — a database, a third-party API, an email, a payment — the Kafka transaction cannot include it, and the only way out is for the external operation to be idempotent.

The three patterns that solve this in practice:

- Inbox table. The consumer writes, in the same database transaction as the business effect, a row with a key derived from the message — topic, partition and offset, or a business identifier — under a uniqueness constraint. If the message is reprocessed, the insert collides and the work is skipped. Cheap, robust and boring, which is the point.
- Transactional outbox. For the reverse path: instead of writing to the database and publishing to Kafka — two systems, no atomicity — you write the event to an outbox table inside the same transaction and a separate process publishes it. With CDC, that process is Debezium.
- Naturally idempotent writes. An UPSERT on a business key, a SET balance = 500 instead of balance = balance + 50, an external call carrying an Idempotency-Key header. When you can design it this way, no auxiliary table is needed at all.

```
-- Inbox: the uniqueness constraint does all the work
CREATE TABLE processed_events (
    topic        text    NOT NULL,
    partition    int     NOT NULL,
    "offset"     bigint  NOT NULL,
    processed_at timestamptz NOT NULL DEFAULT now(),
    PRIMARY KEY (topic, partition, "offset")
);

-- In the same transaction as the business effect:
BEGIN;
  INSERT INTO processed_events (topic, partition, "offset")
  VALUES ('payments', 3, 918273)
  ON CONFLICT DO NOTHING;
  -- If no row was inserted, this message was already handled: ROLLBACK and move on.
  UPDATE accounts SET balance = balance - 4200 WHERE id = 'c-991';
COMMIT;
```

An operational note about that table: it grows forever if nobody prunes it. Partitioning it by day and detaching partitions older than your Kafka retention window is the healthy approach; deleting rows with a nightly bulk DELETE produces exactly the bloat another article on this site covers in detail.

Multi-cluster: the guarantee does not cross the border

MirrorMaker 2 replicates between clusters and does not carry transactions across. Transaction markers are not replicated as such, and the destination cluster has no copy of the source coordinator's state, so a read_committed consumer on the destination is not reading the same transactional boundaries it would at the source. In practice, cross-cluster replication is at-least-once with offsets translated by RemoteClusterUtils, and any consumer on the destination has to be idempotent on its own.

This matters because it breaks a very common assumption in active-active architectures: if your disaster recovery plan is for consumers to continue on the secondary cluster, the exactly-once you carefully maintain on the primary does not travel with them. The reasonable answer is not to try to fix it inside Kafka, it is to design the destination consumer with an inbox table and stop depending on the transport's guarantee.

Observability: the five metrics that matter

Transactions fail in ways that produce no exceptions in your code, so you have to watch them from outside.

- Abort ratio. From the producer, txn-abort-rate against txn-commit-rate. A sustained rise is almost always a transaction timeout caused by slow processing, not a business bug.
- Distance between LSO and log end. The genuinely diagnostic metric. If it grows monotonically on a partition, you have a hanging transaction; that partition's consumer is falling behind for reasons its own lag does not explain.
- Transaction duration. The 99th percentile of the time between beginTransaction and commitTransaction. This is the number to compare against transaction.timeout.ms; if p99 is above half the timeout, you will get aborts during the next traffic peak.
- ProducerFencedException per minute. Zero is normal. A steady trickle means two processes are fighting over the same transactional.id, which is almost always a rollout that never finishes draining the previous one.
- Size of __transaction_state. It grows with the number of distinct transactional.id values you have used, and its entries expire according to transactional.id.expiration.ms — seven days by default. If you generate random identifiers, this topic is what will tell you.

```
groups:
  - name: kafka-eos
    rules:
      - alert: HangingTransaction
        # The LSO is not advancing but the log is: an unclosed transaction
        expr: |
          (kafka_log_log_end_offset - on(topic, partition)
           kafka_log_last_stable_offset) > 0
          and
          delta(kafka_log_last_stable_offset[10m]) == 0
        for: 10m
        labels: { severity: critical }
        annotations:
          summary: "LSO stuck on {{ $labels.topic }}-{{ $labels.partition }}"
          runbook: "https://runbooks.internal/kafka/hanging-transaction"

      - alert: HighTransactionAbortRate
        expr: |
          rate(kafka_producer_txn_abort_total[5m])
          / clamp_min(rate(kafka_producer_txn_commit_total[5m]), 0.001) > 0.05
        for: 15m
        labels: { severity: warning }
```

How you actually test this

A test that publishes three messages and checks three arrive does not prove exactly-once: it proves the happy path works. The guarantee is only demonstrated by breaking things on purpose, and with Testcontainers that fits in an ordinary integration test.

```
import pytest, uuid
from testcontainers.kafka import KafkaContainer

def test_no_duplicates_when_killed_between_write_and_commit(kafka, spawn_worker):
    """The worker writes, and we kill it BEFORE commitTransaction.
    On restart it must reprocess from the previous offset without duplicating."""
    publish(kafka, "input", [f"payment-{i}" for i in range(500)])
```

```
    w = spawn_worker(transactional_id="payments-0", kill_before_commit_at=250)
    w.wait_until_dead()                     # dies mid-batch

    w2 = spawn_worker(transactional_id="payments-0")   # same id: fences the old one
    w2.wait_until_idle()

    out = read_all(kafka, "output", isolation_level="read_committed")
    assert len(out) == 500
    assert len(set(out)) == 500             # not a single repeat

def test_the_zombie_cannot_write(kafka, spawn_worker):
    """Two processes with the same transactional.id: the old one must die."""
    old = spawn_worker(transactional_id="payments-0", pause_after_begin=True)
    new = spawn_worker(transactional_id="payments-0")
    new.wait_until_idle()

    with pytest.raises(ProducerFenced):
        old.resume_and_commit()
```

Three failures only show up with fault injection and belong in the suite: killing the process between the write and the commit — the test above — killing the transaction coordinator mid-flight to confirm the new one finishes the two-phase commit, and forcing an artificial pause longer than transaction.timeout.ms to verify your code treats ProducerFencedException by closing the producer rather than retrying in a loop.

On that last point, the distinction that produces the most bad code: ProducerFencedException, OutOfOrderSequenceException and UnsupportedVersionException are fatal — close the producer and end the process — whereas a generic KafkaException from a send inside the transaction is handled by aborting and retrying the whole transaction. Treating a fatal error as retryable is the exact recipe for duplicating.

A complete case, with numbers

A payment reconciliation pipeline: it reads from a topic with twenty-four partitions, enriches against a database, and writes to two output topics, one for reconciled movements and one for exceptions. Before: at-least-once with key-based deduplication in the final consumer. The problem was not correctness, it was that deduplication lived in the last link while three different teams consumed the intermediate topic, each with its own idea of what a duplicate was.

Migrating to transactions took two weeks and the real work was not in the producer code. It was in three places. First, the transactional.id: the application ran as a Deployment with random pod names, so it had to become a StatefulSet to get stable identifiers. Second, the timeout: database enrichment had a p99 of eleven seconds at peak, uncomfortably close to the one-minute default once the full batch was added up, and it was raised to three minutes after also raising transaction.max.timeout.ms on the brokers. Third, the consumers: all three teams had to switch to read_committed, and one of them discovered its Go client was on a library version predating transactions v2 support.

The numbers after three months: median end-to-end latency rose from 240 ms to 310 ms, and p99 from 1.9 s to 2.6 s, all of it from transaction batch size and LSO blocking. Throughput dropped 9% on the same hardware. In exchange, the three consumers deleted their deduplication logic — about four hundred lines between them, plus two Redis tables nobody maintains any more — and "payment applied twice" incidents went from an average of 1.3 a month to zero across ninety days.

There was one incident during the period, and it is worth telling because it was not Kafka's fault. A rollout with too short a terminationGracePeriodSeconds killed pods mid-transaction; the transactions stayed open until the coordinator aborted them on timeout, and for those three minutes the LSO on several partitions did not advance. The consumers lost nothing, but they fell behind far enough to trigger lag alerts and to make

somebody believe, for ten minutes, that data had been lost. The fix was a clean shutdown that closes the in-flight transaction before exiting.

When not to use exactly-once

It is worth closing with the opposite case, because the cost is real and does not always pay off.

- When the effect is already idempotent. If you write to a key-value store with SET on a business key, duplicates do no harm. Paying latency and markers for a guarantee you do not need is pure waste.
- When latency is the product. In a chain where every millisecond counts, LSO blocking and commit cost are hard to justify against cheap deduplication at the destination.
- When the destination is not Kafka. Said above, but worth repeating: if the end of the chain is a database or an API, the Kafka transaction does not save you and you will end up writing the inbox table anyway. In that case, ask whether the complexity of transactions buys you anything over at-least-once plus idempotence.
- When you cannot guarantee stable identifiers. If your platform will not let you keep transactional.id values stable across restarts, you have no fencing, and without fencing transactions give a feeling of safety without the safety. That is worse than not using them.

The right question is never "do I want exactly-once?" — everyone does. It is "where is the last point in my chain at which a duplicate does damage, and what is the cheapest way to make it harmless right there?". Often the answer is a uniqueness constraint on a table, and that answer is perfectly respectable.

Enabling transaction.version=2 without surprises

KIP-890's hardened protocol is not enabled just by upgrading the brokers: it is a cluster feature you have to raise explicitly, and the order matters.

```
# 1. See which version the cluster is on
kafka-features.sh --bootstrap-server broker:9092 describe

# 2. Raise it only once ALL brokers are already on the new version
kafka-features.sh --bootstrap-server broker:9092 \
  upgrade --feature transaction.version=2
```

Three things to know before running the second command. First: older clients keep working; the protocol negotiates per connection, so an old producer will speak the old protocol and stay exposed to the hanging-transaction problem until it is upgraded. Second: the real benefit is that the producer epoch is bumped on every transaction, not only on initialisation, and that closes the window where a delayed packet from an already-aborted transaction landed inside the next one and contaminated it. Third, and the most surprising: not every third-party client implements version 2. Before raising the feature it is worth inventorying which libraries are out there — the Go and Rust clients have trailed the Java client — because a client that does not support it does not fail at startup, it simply keeps using the old protocol and you do not get the protection for the one that needs it most.

Once enabled, kafka-transactions.sh stops being an awkward last resort and becomes an ordinary diagnostic tool: describe-producers on the blocked partition gives you the culprit's PID and epoch, and abort with those values unblocks the LSO without touching committed data.

Four questions that always come up afterwards

Do transactions work across topics on different clusters? No. A transaction lives on a coordinator in one specific cluster and its markers are written to partitions of that same cluster. Writing atomically to two clusters is not something Kafka offers; the usual pattern is to commit on the source cluster and replicate afterwards, accepting at-least-once on the hop.

Can I mix transactional and non-transactional producers on the same topic? Yes, and it works. Non-transactional messages are immediately visible to any consumer. What you have to keep in mind is that a read_committed consumer of that topic will still block behind the LSO whenever a transaction is open, even if the messages it cares about are the non-transactional ones further ahead in the log.

Do log compaction and transactions get along? With caveats. Transaction markers are retained for transactional.id.expiration.ms and become eligible for cleanup afterwards, so a consumer starting from the beginning of a very old compacted topic can encounter messages whose markers are gone. In practice nothing breaks, because the message is already committed, but it explains why message counts do not add up when you recount a compacted topic.

What happens if I raise transaction.timeout.ms very high "just in case"? You turn a hanging transaction into a long outage. A fifteen-minute timeout means that if a process dies right after writing and before committing, consumers of those partitions see nothing new for fifteen minutes. The timeout is not free safety margin: it is the maximum time you accept having the partition blocked.