PuigFlows

El Patrón Transactional Outbox: Eventos Fiables sin Dual Writes

Premium Guía · Intermedio

The Transactional Outbox Pattern: Reliable Events Without Dual Writes

Autor / Author: Josué García Puig
Publicado / Published: 10 julio 2026 · www.puigflows.com
Lectura / Reading time: ~36 min por idioma / per language

Edición bilingüe — Español (parte 1) · English (part 2)

Español — El Patrón Transactional Outbox

Por qué el dual write pierde eventos

Casi todo servicio acaba necesitando avisar a otros sistemas de que algo pasó: se creó un pedido, se cobró un pago, un usuario cambió su email. La implementación ingenua hace dos escrituras seguidas: primero guarda el dato en la base de datos y después publica el evento en Kafka, RabbitMQ o SQS.

```
# ANTI-PATRÓN: dual write
def create_order(data):
    db.insert("orders", data)              # escritura 1: base de datos
    broker.publish("order.created", data)  # escritura 2: broker
```

El problema es que ninguna transacción cubre ambas escrituras. Hay dos formas de fallar, y las dos duelen:

- **Se guarda el pedido pero el evento nunca sale.** El proceso se cae, el broker no responde o hay un deploy justo entre las dos líneas. El almacén nunca se entera y el pedido queda huérfano.
- **Se publica el evento pero la transacción hace rollback.** Si publicas dentro de la transacción y esta falla después, el resto del sistema reacciona a algo que nunca ocurrió.

Los reintentos no arreglan esto: reintentar la publicación tras confirmar la transacción sigue dejando una ventana en la que el proceso puede morir. La solución no es reintentar mejor, es eliminar la doble escritura.

Cómo funciona el patrón transactional outbox

La idea es simple: en lugar de escribir en dos sistemas, escribe solo en uno. El evento se guarda en una tabla `outbox` de tu propia base de datos, dentro de la misma transacción que modifica los datos de negocio. Como es una única transacción ACID, o se confirman las dos cosas o ninguna.

1. El servicio abre una transacción, escribe el pedido en `orders` y el evento en `outbox`, y hace commit.
2. Un proceso independiente (el *relay*) lee la tabla outbox y publica los eventos en el broker.
3. Cuando la publicación se confirma, el evento se marca como procesado o se borra.

El broker puede caerse, el relay puede reiniciarse, y no pasa nada: el evento sigue en la base de datos esperando su turno. La garantía resultante es entrega **al menos una vez** (at-least-once), nunca cero veces.

La tabla outbox

```
CREATE TABLE outbox (
  id             BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
```

```
  aggregate_type TEXT        NOT NULL,  -- "order", "payment"
  aggregate_id   TEXT        NOT NULL,  -- id de la entidad
  event_type     TEXT        NOT NULL,  -- "order.created"
  payload        JSONB       NOT NULL,
  created_at     TIMESTAMPTZ NOT NULL DEFAULT now(),
  processed_at   TIMESTAMPTZ            -- NULL = pendiente
);

-- Índice parcial: el relay solo escanea lo pendiente
CREATE INDEX idx_outbox_pending
  ON outbox (id) WHERE processed_at IS NULL;
```

`aggregate_id` es clave: más adelante servirá como clave de partición en Kafka para conservar el orden por entidad. El índice parcial mantiene el escaneo barato aunque la tabla acumule millones de filas ya procesadas.

Escribir el evento en la misma transacción

```
import json
import uuid
import psycopg

def create_order(conn: psycopg.Connection, order: dict) -> str:
    order_id = str(uuid.uuid4())
    with conn.transaction():  # una sola transacción ACID
        conn.execute(
            "INSERT INTO orders (id, customer_id, total) VALUES (%s, %s, %s)",
            (order_id, order["customer_id"], order["total"]),
        )
        conn.execute(
            """
            INSERT INTO outbox (aggregate_type, aggregate_id, event_type, payload)
            VALUES ('order', %s, 'order.created', %s)
            """,
            (order_id, json.dumps({
                "event_id": str(uuid.uuid4()),  # para deduplicar en el consumidor
                "version": 1,
                "order_id": order_id,
                "customer_id": order["customer_id"],
                "total": order["total"],
            })),
        )
    return order_id
```

Fíjate en el `event_id` dentro del payload: los consumidores lo usarán para deduplicar, porque con at-least-once los duplicados no son un caso raro, son parte del contrato.

Opción A: relay con polling

La versión más sencilla del relay es un worker que consulta la tabla cada pocos cientos de milisegundos. La pieza importante es `FOR UPDATE SKIP LOCKED`: permite ejecutar varios workers en paralelo sin que se pisen, porque cada uno bloquea filas distintas.

```
import time
import psycopg
from confluent_kafka import Producer

producer = Producer({
    "bootstrap.servers": "localhost:9092",
    "enable.idempotence": True,
    "acks": "all",
})

def relay_batch(conn: psycopg.Connection) -> int:
    with conn.transaction():
        rows = conn.execute(
            """
            SELECT id, aggregate_id, event_type, payload::text
            FROM outbox
            WHERE processed_at IS NULL
            ORDER BY id
            LIMIT 100
            FOR UPDATE SKIP LOCKED
            """
        ).fetchall()

        for _id, aggregate_id, event_type, payload in rows:
            producer.produce(topic=event_type, key=aggregate_id, value=payload)

        producer.flush(timeout=10)  # espera el ack de Kafka

        if rows:
            conn.execute(
                "UPDATE outbox SET processed_at = now() WHERE id = ANY(%s)",
                ([r[0] for r in rows],),
            )
    return len(rows)

while True:
    if relay_batch(conn) == 0:
        time.sleep(0.2)  # sin trabajo pendiente: espera antes de volver a consultar
```

Si el worker muere después de publicar pero antes del UPDATE, el siguiente ciclo volverá a publicar esas filas. Es el comportamiento correcto: duplicados sí, pérdidas no.

El polling añade una latencia media de la mitad del intervalo y algo de carga a la base de datos, pero no requiere infraestructura nueva. Para la mayoría de sistemas es más que suficiente.

Opción B: CDC con Debezium

La alternativa es capturar los cambios directamente del log de transacciones (WAL en PostgreSQL, binlog en MySQL) con una herramienta de change data capture. Debezium es el estándar de facto: detecta cada INSERT en la outbox en cuanto se confirma y lo publica en Kafka en milisegundos, sin consultas adicionales a la base de datos.

```
{
  "name": "outbox-connector",
  "config": {
```

```
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "db",
    "database.dbname": "shop",
    "table.include.list": "public.outbox",
    "plugin.name": "pgoutput",
    "transforms": "outbox",
    "transforms.outbox.type": "io.debezium.transforms.outbox.EventRouter",
    "transforms.outbox.table.field.event.key": "aggregate_id",
    "transforms.outbox.route.by.field": "event_type"
  }
}
```

El `EventRouter` de Debezium está pensado exactamente para este patrón: extrae el payload, usa `aggregate_id` como clave del mensaje y enruta cada tipo de evento a su topic. Con CDC ni siquiera necesitas la columna `processed_at`: Debezium lee del log, no de la tabla, así que puedes borrar la fila justo después de insertarla.

¿Cuál elegir? Con requisitos de latencia por debajo de ~100 ms y Kafka ya desplegado, CDC. Si tu stack es más simple o la latencia de un polling corto te vale, el polling gana por pura simplicidad operativa: Debezium implica Kafka Connect, replicación lógica, monitorización de lag y evolución de esquemas.

El consumidor: idempotencia obligatoria

Con at-least-once, todo consumidor verá eventos repetidos tarde o temprano. La defensa estándar es registrar los `event_id` ya procesados en la misma transacción que aplica el efecto:

```
def handle_order_created(conn: psycopg.Connection, event: dict) -> None:
    with conn.transaction():
        inserted = conn.execute(
            """
            INSERT INTO processed_events (event_id) VALUES (%s)
            ON CONFLICT (event_id) DO NOTHING
            """,
            (event["event_id"],),
        ).rowcount
        if inserted == 0:
            return  # duplicado: ya lo procesamos
        reserve_stock(conn, event["order_id"])
```

Errores comunes

- **Publicar dentro de la transacción y llamarlo outbox.** Si llamas al broker antes del commit, sigues teniendo un dual write con pasos extra. El relay debe ser un proceso separado que lea de la tabla.
- **Olvidar la limpieza.** Una outbox que solo crece degrada el vacuum y los backups. Borra lo procesado por lotes con un job periódico: `DELETE FROM outbox WHERE id IN (SELECT id FROM outbox WHERE processed_at < now() - interval '7 days' LIMIT 10000)`.
- **Asumir orden global.** Ni Kafka ni el relay garantizan orden entre entidades distintas. Lo que sí puedes garantizar es el orden por entidad usando `aggregate_id` como clave de partición.

- **Payloads que solo llevan un id.** Si el consumidor tiene que consultar la fila, puede leer un estado posterior al del evento. Incluye en el payload todo lo que el consumidor necesita (event-carried state).
- **Consumidores sin deduplicación.** At-least-once significa que los duplicados llegarán. Un consumidor que no deduplica cobrará dos veces el mismo pedido el día menos pensado.

Buenas prácticas

- Genera un `event_id` único al escribir en la outbox y propágalo hasta el consumidor final.
- Versiona los payloads (`"version": 1`) para poder evolucionar el esquema sin romper consumidores.
- Monitoriza dos métricas: filas pendientes (`processed_at IS NULL`) y edad del evento pendiente más antiguo. Son tus alertas clave de que el relay va con retraso.
- Activa `enable.idempotence` en el productor de Kafka para no añadir duplicados extra en la capa de publicación.
- Empieza con polling; migra a CDC cuando la latencia o la carga del polling sean un problema medible, no antes.

Garantías de orden: qué puedes prometer y qué no

Una de las primeras preguntas que aparecen al llevar la outbox a producción es qué orden ven los consumidores. La respuesta corta: puedes garantizar orden **por agregado**, y no deberías intentar garantizar orden global. Entender por qué te ahorra semanas de depuración.

El orden global —que todos los eventos del sistema lleguen en la secuencia exacta en que se escribieron— exigiría un único punto de serialización en cada etapa: un solo relay, una sola partición de Kafka, un solo consumidor. Eso convierte tu pipeline en una cola con throughput de un solo hilo. Casi ningún negocio lo necesita: lo que importa es que los eventos *de un mismo pedido* lleguen en orden, no que un evento del pedido A llegue antes que uno del pedido B.

Para conseguir orden por agregado necesitas tres piezas alineadas:

1. **Una clave de secuencia en la outbox.** El `id BIGINT GENERATED ALWAYS AS IDENTITY` de la tabla ya te lo da: refleja el orden de commit con una salvedad que veremos abajo.
2. **Publicar con clave de partición.** En Kafka, todos los eventos con la misma key van a la misma partición, y dentro de una partición el orden es estable. Usa `aggregate_id` como key.
3. **Un consumidor que procese cada partición secuencialmente.** Si paralelizas el procesamiento dentro de una partición, destruyes el orden que tanto costó conservar.

```
from confluent_kafka import Producer

producer = Producer({
    "bootstrap.servers": "kafka:9092",
    "enable.idempotence": True,   # evita duplicados por reintentos del producer
    "acks": "all",                # espera confirmación de todas las réplicas ISR
})

def publish(event: dict) -> None:
```

```
    producer.produce(
        topic=f"{event['aggregate_type']}.events",
        key=event["aggregate_id"],       # misma entidad -> misma partición
        value=json.dumps(event["payload"]),
        headers={"event_type": event["event_type"], "event_id": str(event["id"])},
    )
```

`enable.idempotence` merece una aclaración: elimina los duplicados que crea *el propio producer* al reintentar un envío que en realidad sí llegó. No convierte el pipeline en exactly-once de extremo a extremo —el relay puede seguir publicando el mismo evento dos veces si muere entre el ack del broker y el UPDATE de `processed_at`—, pero sí cierra una fuente de duplicados gratuita.

La trampa del orden de commit

Hay un detalle sutil que muerde en sistemas con alta concurrencia: el `id` de la outbox se asigna al hacer el INSERT, pero la fila se hace visible al hacer COMMIT, y ambos órdenes pueden no coincidir. La transacción T1 puede insertar el evento 100, tardar en confirmar, y la T2 insertar el 101 y confirmar antes. Un relay que lee "todo lo pendiente con id > último_procesado" puede ver el 101, avanzar su marcador y saltarse el 100 para siempre.

Las soluciones habituales, de más simple a más robusta:

- **No usar marcador de id, usar `processed_at IS NULL`.** El relay de este artículo ya lo hace así: nunca se salta filas porque no asume que los ids llegan ordenados. Es la razón por la que el patrón del índice parcial es el recomendado.
- **`pg_current_snapshot()` como barrera.** Si necesitas marcador por rendimiento, lee solo filas cuyo `id` sea menor que el `xmin` del snapshot actual, garantizando que no hay transacciones en vuelo que puedan insertar ids menores.
- **CDC.** Debezium lee el WAL, y el WAL está ordenado por commit por construcción. El problema desaparece de raíz.

Escalar el relay sin romper nada

La intuición dice "si hay muchos eventos, levanto más relays". La experiencia dice otra cosa: **un solo relay bien hecho aguanta muchísimo más de lo que crees**. Publicar en Kafka por lotes alcanza decenas de miles de eventos por segundo desde un único proceso; la mayoría de sistemas no genera ni una fracción de eso. Antes de escalar horizontalmente, mide.

El primer nivel de escalado no es añadir procesos, es **procesar por lotes**:

```
BATCH = 500  # punto de partida razonable; ajusta midiendo lag y latencia

def relay_tick(conn) -> int:
    with conn.transaction():
        rows = conn.execute(
            """
            SELECT id, aggregate_type, aggregate_id, event_type, payload
            FROM outbox
            WHERE processed_at IS NULL
            ORDER BY id
            LIMIT %s
            FOR UPDATE SKIP LOCKED
```

```
            """,
            (BATCH,),
        ).fetchall()

        if not rows:
            return 0

        for r in rows:
            publish(row_to_event(r))
        producer.flush(timeout=10)  # espera los acks del broker ANTES de marcar

        conn.execute(
            "UPDATE outbox SET processed_at = now() WHERE id = ANY(%s)",
            ([r[0] for r in rows],),
        )
    return len(rows)
```

Dos decisiones de este código importan más de lo que parecen:

- **El `flush()` va antes del UPDATE.** Marcar como procesado antes de confirmar la entrega convierte tu at-least-once en at-most-once: si el broker rechaza el lote después, el evento se pierde. El orden correcto es publicar, esperar acks, marcar.
- **El tamaño del lote es un compromiso.** Lotes pequeños queman CPU en overhead de consultas y commits; lotes enormes alargan la transacción, retienen locks y, si algo falla a mitad, el reintento republica el lote entero (más duplicados). Entre 100 y 1.000 suele funcionar; ajusta observando el lag.

Varios relays: cuándo y cómo

`FOR UPDATE SKIP LOCKED` permite que varios relays trabajen a la vez sin pisarse: cada uno bloquea un lote distinto y salta las filas ya bloqueadas. Es la herramienta correcta, pero tiene un coste silencioso: **rompe el orden global de publicación**. El relay A puede publicar los eventos 200–299 antes de que el relay B termine con los 100–199. Si conservas la key de partición por `aggregate_id`, el desorden entre agregados distintos es inofensivo... salvo que dos eventos del *mismo* agregado caigan en lotes de relays distintos.

Si necesitas varios relays y orden estricto por agregado, particiona el trabajo de forma estable en lugar de competir por filas:

```
-- Relay i de N procesa solo su franja estable de agregados
SELECT id, aggregate_type, aggregate_id, event_type, payload
FROM outbox
WHERE processed_at IS NULL
  AND mod(abs(hashtext(aggregate_id)), %(n_relays)s) = %(relay_index)s
ORDER BY id
LIMIT 500
FOR UPDATE SKIP LOCKED;
```

Cada agregado pertenece siempre al mismo relay, así que sus eventos salen en orden, y aun así repartes la carga. Es el mismo principio que usa Kafka con las particiones, aplicado un paso antes.

Un último consejo operativo: **despliega el relay como unidad independiente del API**. Puede vivir en el mismo repositorio, pero con su propio deployment: escala por su cuenta, se reinicia sin tirar tráfico HTTP y sus métricas no se mezclan con las del servicio web.

Semántica de entrega: por qué exactly-once es un contrato, no una función

Conviene decirlo sin rodeos: **la outbox te da at-least-once**. Cada evento confirmado en la transacción de negocio será publicado una o más veces. "Exactamente una vez" de extremo a extremo —desde tu base de datos hasta el efecto en el consumidor— no lo da ninguna herramienta por sí sola, por mucho marketing que lo prometa: siempre existe una ventana entre "publiqué" y "anoté que publiqué" en la que un crash produce repetición.

Lo que sí puedes construir es **procesamiento efectivamente-una-vez**: el evento puede llegar duplicado, pero su efecto se aplica una sola vez. La responsabilidad se reparte:

- **El producer** (relay) minimiza duplicados: idempotencia del producer de Kafka, flush antes de marcar, lotes razonables.
- **El transporte** conserva el orden por partición y no pierde mensajes confirmados (replicación, `acks=all`).
- **El consumidor** deduplica. Aquí es donde se gana o se pierde la partida, y por eso merece su propio patrón.

El patrón transactional inbox: la otra mitad del contrato

El artículo ya estableció que el consumidor debe ser idempotente. El *transactional inbox* es la versión estructurada de esa idea: el mismo truco de la outbox, aplicado del lado del que recibe. En lugar de "procesar y recordar que procesé" como dos pasos separados (otro dual write, ahora en el consumidor), ambos ocurren en una sola transacción local.

```
CREATE TABLE inbox (
  event_id     TEXT PRIMARY KEY,       -- id del evento, viene en el mensaje
  consumer     TEXT NOT NULL,          -- nombre lógico del consumidor
  received_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);
-- Si varios consumidores comparten base de datos:
-- PRIMARY KEY (event_id, consumer)
```

```
def handle_message(conn, msg) -> None:
    event_id = dict(msg.headers() or {}).get("event_id", b"").decode()
    with conn.transaction():
        inserted = conn.execute(
            """
            INSERT INTO inbox (event_id, consumer)
            VALUES (%s, %s)
            ON CONFLICT DO NOTHING
            """,
            (event_id, "warehouse-service"),
        ).rowcount
```

```
        if inserted == 0:
            return  # duplicado: ya lo procesamos, transacción vacía y fuera

        apply_business_logic(conn, json.loads(msg.value()))  # misma transacción

    consumer.commit(msg)  # el offset de Kafka se confirma DESPUÉS del commit local
```

La elegancia del patrón está en que el INSERT en `inbox` y la lógica de negocio comparten transacción: o se aplican los dos o ninguno. Si el proceso muere después del commit local pero antes de confirmar el offset, Kafka reenviará el mensaje, el INSERT chocará con la clave primaria y el handler saldrá limpiamente sin efectos repetidos. El `ON CONFLICT DO NOTHING` con `rowcount` es la comprobación y el candado en una sola sentencia atómica.

La tabla inbox también necesita retención (un job que borre filas de más de, por ejemplo, 30 días), y ese plazo define tu ventana real de deduplicación: un duplicado que llegue más tarde que la retención se procesará de nuevo. Elige el plazo mirando cuánto puede retrasarse legítimamente un mensaje en tu sistema —reprocesados, replays de particiones, restauraciones— y añade margen.

Debezium en producción: la configuración completa

La sección anterior presentó CDC como alternativa al polling; esta baja al detalle de qué hay que configurar para que funcione de verdad. El stack es PostgreSQL con replicación lógica, Kafka Connect y el conector de Debezium (la serie 3.x es la actual; Debezium 3.4 salió a finales de 2025 y se apoya en Kafka Connect 4). El plugin de salida es `pgoutput`, el nativo de PostgreSQL desde la versión 10: no hay que instalar nada en el servidor de base de datos.

Primero, la base de datos. Necesitas `wal_level = logical` (requiere reinicio), un usuario con permiso de replicación y una *publication* que exponga solo la tabla outbox:

```
-- postgresql.conf: wal_level = logical  (y reiniciar)

CREATE USER debezium WITH REPLICATION PASSWORD '...';
GRANT SELECT ON outbox TO debezium;

-- Publicar SOLO la outbox: el resto del esquema no le incumbe al conector
CREATE PUBLICATION outbox_pub FOR TABLE outbox;
```

Después, el conector. Esta configuración registra un conector que lee la outbox y aplica el **outbox event router**, la SMT (Single Message Transform) que Debezium trae específicamente para este patrón: extrae el payload, enruta cada evento al topic de su `aggregate_type` y usa `aggregate_id` como key del mensaje —el orden por agregado queda resuelto de serie—.

```
{
  "name": "outbox-connector",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "db", "database.port": "5432",
    "database.user": "debezium", "database.password": "...",
    "database.dbname": "orders", "topic.prefix": "orders-svc",

    "plugin.name": "pgoutput",
    "publication.name": "outbox_pub",
```

```
    "slot.name": "outbox_slot",
    "table.include.list": "public.outbox",
    "snapshot.mode": "no_data",

    "transforms": "outbox",
    "transforms.outbox.type": "io.debezium.transforms.outbox.EventRouter",
    "transforms.outbox.table.field.event.key": "aggregate_id",
    "transforms.outbox.route.by.field": "aggregate_type",
    "transforms.outbox.table.expand.json.payload": "true",

    "heartbeat.interval.ms": "10000",
    "tombstones.on.delete": "false"
  }
}
```

Tres opciones de ahí arriba evitan incidentes reales:

- **`snapshot.mode: no_data`.** Sin esto, el conector intenta hacer un snapshot inicial de la tabla al arrancar. Para una outbox no tiene sentido republicar el histórico ya procesado.
- **`heartbeat.interval.ms`.** Protege contra el problema más traicionero de la replicación lógica: el *slot bloat*. Un replication slot obliga a PostgreSQL a retener WAL hasta que el consumidor lo confirme. Si tu outbox tiene poco tráfico pero otras tablas de la base de datos escriben mucho, el slot avanza despacio y el WAL retenido crece hasta llenar el disco —un clásico de las tres de la madrugada—. Los heartbeats fuerzan avances periódicos del slot aunque no haya eventos.
- **`tombstones.on.delete: false`.** Si borras filas de la outbox al procesarlas (o en el job de limpieza), no quieres que cada DELETE genere un tombstone en el topic destino.

Y una regla operativa que no está en la configuración: **monitoriza el slot**. Esta consulta debe estar en tu dashboard, con alerta si crece sin freno:

```
SELECT slot_name, active,
       pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS retained_wal
FROM pg_replication_slots;
```

Si un conector se queda parado días (deploy roto, Kafka Connect caído), el WAL retenido de su slot crece sin límite. PostgreSQL ofrece `max_slot_wal_keep_size` como cinturón de seguridad: limita cuánto WAL puede retener un slot antes de invalidarlo. Configúralo; perder un slot y reconstruir el conector es molesto, pero llenar el disco de la base de datos de producción es mucho peor.

Mantenimiento de la outbox: la tabla que solo crece

La outbox es una tabla de solo-inserción con un patrón de acceso muy particular: se escribe constantemente, se lee solo lo reciente y lo antiguo no le importa a nadie. Sin mantenimiento, acumula millones de filas muertas, el autovacuum se pasa la vida limpiándola y su bloat contamina los backups y los tiempos de restore.

Hay dos estrategias, y la elección depende del volumen:

Borrado por lotes (volumen bajo y medio)

```
-- Ejecutar cada pocos minutos (pg_cron, o un job del propio relay)
WITH victims AS (
  SELECT id FROM outbox
  WHERE processed_at IS NOT NULL
    AND processed_at < now() - interval '7 days'
  ORDER BY id
  LIMIT 10000
)
DELETE FROM outbox WHERE id IN (SELECT id FROM victims);
```

El `LIMIT` es lo importante: un DELETE de millones de filas en una sentencia retiene locks durante minutos, genera un pico de WAL y puede tumbar la réplica. Borrar en lotes de diez mil, en bucle hasta que no quede nada, hace el mismo trabajo sin drama. Los 7 días de gracia dejan margen para depurar incidentes recientes con los eventos aún a mano.

Particionado por rango de tiempo (volumen alto)

A partir de cientos de miles de eventos al día, borrar deja de escalar: cada DELETE genera filas muertas que el autovacuum tiene que recoger. La alternativa es que borrar sea gratis:

```
CREATE TABLE outbox (
  id           BIGINT GENERATED ALWAYS AS IDENTITY,
  aggregate_type TEXT NOT NULL,
  aggregate_id TEXT NOT NULL,
  event_type   TEXT NOT NULL,
  payload      JSONB NOT NULL,
  created_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
  processed_at TIMESTAMPTZ,
  PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);

-- Una partición por día (pg_partman automatiza la creación)
CREATE TABLE outbox_2026_07_10 PARTITION OF outbox
  FOR VALUES FROM ('2026-07-10') TO ('2026-07-11');

-- La limpieza ahora es instantánea y no genera filas muertas:
DROP TABLE outbox_2026_07_03;
```

`DROP TABLE` de una partición vieja es una operación de metadatos: milisegundos, cero bloat, cero trabajo para el autovacuum. Extensiones como `pg_partman` crean y eliminan particiones automáticamente con la retención que definas. El precio es algo más de complejidad en el esquema (la clave primaria debe incluir la columna de particionado) y asegurarte de que el relay o Debezium leen de la tabla padre.

Observabilidad: las tres métricas que importan

Una outbox sana es invisible; una enferma se descubre tarde y mal —normalmente porque un equipo consumidor pregunta por qué no le llegan eventos desde hace una hora—. Tres métricas convierten ese descubrimiento tardío en una alerta temprana:

- **`outbox_pending_events`:** cuántas filas esperan publicación. Su valor absoluto importa menos que su tendencia: creciente y sostenida significa que el relay no da abasto o está muerto.
- **`outbox_oldest_pending_seconds`:** la edad del evento pendiente más antiguo. Es tu lag real de extremo a extremo y la mejor señal de alerta: si supera tu SLO de entrega (por ejemplo, 60 segundos), algo va mal, aunque el contador de pendientes parezca pequeño.
- **`outbox_published_total`:** contador de eventos publicados. Su derivada es el throughput; si cae a cero con tráfico de escritura normal, el relay está caído aunque su proceso siga "vivo".

```
from prometheus_client import Gauge, Counter

PENDING = Gauge("outbox_pending_events", "Filas pendientes de publicar")
OLDEST  = Gauge("outbox_oldest_pending_seconds", "Edad del pendiente más antiguo")
PUBLISHED = Counter("outbox_published_total", "Eventos publicados")

def scrape_outbox_metrics(conn) -> None:
    row = conn.execute(
        """
        SELECT count(*),
               coalesce(extract(epoch FROM now() - min(created_at)), 0)
        FROM outbox WHERE processed_at IS NULL
        """
    ).fetchone()
    PENDING.set(row[0])
    OLDEST.set(row[1])
    # PUBLISHED.inc(n) se llama desde relay_tick() con el tamaño del lote
```

Con eso, dos alertas bastan para cubrir el 95 % de los incidentes: `outbox_oldest_pending_seconds > 60` durante 5 minutos (el pipeline está atascado) y ausencia de scrape (el relay ni siquiera responde). Si usas Debezium, añade la métrica de WAL retenido del slot de la sección anterior.

Trazas que cruzan la outbox

Un efecto secundario molesto del patrón es que rompe el trazado distribuido: la petición HTTP que creó el pedido y el consumidor que reaccionó al evento aparecen como trazas inconexas, y "¿por qué tardó 40 segundos en enviarse el email?" se vuelve difícil de responder. La solución es tratar el contexto de traza como parte del evento: guarda el header `traceparent` (W3C Trace Context) en la fila de la outbox al escribirla, propágalo como header del mensaje en Kafka y reconstrúyelo en el consumidor como *link* de la traza original. OpenTelemetry lo soporta en todos sus SDKs; el coste es una columna de texto más.

Cómo testear una outbox sin engañarte

El error clásico al testear este patrón es mockear la base de datos o el broker: el test pasa siempre y no demuestra nada, porque lo que la outbox garantiza es precisamente el comportamiento

transaccional real. La atomicidad de "pedido + evento" solo se puede verificar contra un PostgreSQL de verdad. Con Testcontainers, levantar uno por sesión de tests cuesta unos segundos:

```
import pytest, psycopg
from testcontainers.postgres import PostgresContainer

@pytest.fixture(scope="session")
def pg():
    with PostgresContainer("postgres:17") as container:
        conn = psycopg.connect(container.get_connection_url())
        conn.execute(SCHEMA_SQL)  # orders + outbox
        yield conn

def test_pedido_y_evento_son_atomicos(pg):
    create_order(pg, {"customer": "ada", "total": 99})
    orders = pg.execute("SELECT count(*) FROM orders").fetchone()[0]
    events = pg.execute("SELECT count(*) FROM outbox WHERE processed_at IS
NULL").fetchone()[0]
    assert (orders, events) == (1, 1)

def test_rollback_no_deja_evento_huerfano(pg):
    with pytest.raises(ValueError):
        create_order(pg, {"customer": "ada", "total": -5})  # validación falla DENTRO de la
tx
    assert pg.execute("SELECT count(*) FROM outbox").fetchone()[0] == 0

def test_relay_no_marca_si_el_broker_falla(pg, broker_caido):
    create_order(pg, {"customer": "ada", "total": 99})
    with pytest.raises(BrokerError):
        relay_tick(pg)
    # El evento DEBE seguir pendiente para el siguiente tick
    assert pg.execute(
        "SELECT count(*) FROM outbox WHERE processed_at IS NULL"
    ).fetchone()[0] == 1
```

El tercer test es el que más bugs caza: verifica que un fallo de publicación deja el evento pendiente en lugar de marcarlo, que es exactamente la ventana donde una implementación descuidada pierde datos. Complétalo con su simétrico —si el broker confirma pero el UPDATE falla, el siguiente tick republica y el consumidor deduplica— y tendrás cubierto el contrato completo del patrón.

Más allá de los tests automatizados, dedica una tarde a **ensayar los fallos en un entorno de staging**: mata el relay con eventos a medio lote (kill -9, no un shutdown limpio), tira Kafka durante diez minutos con tráfico de escritura activo, deja el conector de Debezium parado una hora y mira el WAL retenido. Cada uno de esos ensayos suele descubrir un ajuste que faltaba —un timeout, una alerta, un índice— y es infinitamente más barato descubrirlo en staging que en producción.

Versionado de eventos: el contrato también evoluciona

El payload de un evento es un contrato público: en cuanto el primer consumidor externo lo procesa, ya no puedes cambiarlo alegremente. Y lo cambiarás, porque el negocio cambia. Las outbox

longevas acaban sufriendo más por la evolución del esquema que por el throughput, así que conviene decidir las reglas el primer día:

- **Versiona desde el principio.** Añade `event_version` (un entero) a la fila de la outbox y al mensaje. La versión 1 explícita de hoy evita la arqueología de "¿qué formato tienen estos eventos de 2024?" mañana.
- **Los cambios aditivos son seguros; el resto, no.** Añadir un campo opcional no rompe a nadie que ignore campos desconocidos (haz que tus consumidores los ignoren: es la mitad barata de la compatibilidad). Renombrar, eliminar o cambiar el tipo de un campo existente rompe consumidores, y exige subir la versión.
- **Para cambios que rompen, publica ambas versiones durante la transición.** El productor escribe v1 y v2 (dos filas de outbox o un campo con ambas formas) hasta que el último consumidor migra. Es tedioso y es también la única forma ordenada; la alternativa —coordinar un big bang de deploys— falla siempre.
- **Considera un schema registry cuando haya más de dos equipos consumiendo.** Confluent Schema Registry o Apicurio validan compatibilidad en el momento de publicar, convirtiendo "rompimos a un consumidor" de incidente en error de CI. Con JSON a pequeña escala, un directorio de esquemas versionados en el repositorio y validación en los tests cumple el mismo papel.

Una técnica que escala bien en el lado consumidor es el *upcasting*: el consumidor convierte cualquier versión antigua a la actual nada más recibirla, y el resto de su código solo conoce la última forma. La lógica de compatibilidad queda concentrada en una función pura, trivial de testear:

```
def upcast(event: dict) -> dict:
    v = event.get("event_version", 1)
    if v == 1:
        # v2 separó "name" en "first_name" / "last_name"
        first, _, last = event["payload"].pop("name", "").partition(" ")
        event["payload"]["first_name"] = first
        event["payload"]["last_name"] = last
        event["event_version"] = 2
    return event
```

Alternativas: cuándo NO usar una outbox

El patrón es excelente, no universal. Señales de que hay una opción más simple o más adecuada:

- **El "evento" solo lo consumes tú, en el mismo proceso y la misma base de datos.** Entonces no necesitas un broker: una tabla de jobs con `FOR UPDATE SKIP LOCKED` y un worker es el mismo mecanismo sin Kafka en medio. La outbox brilla cuando hay *otros* sistemas escuchando.
- **Necesitas la respuesta ahora.** Si el llamante requiere el resultado de forma síncrona (una validación, un cálculo de precio), esto es una llamada REST/gRPC con timeout y reintentos, no un evento. Forzar petición-respuesta sobre mensajería añade latencia y complejidad sin ganar nada.
- **Ya haces CDC de las tablas de negocio.** Capturar directamente `orders` con Debezium evita la tabla outbox, pero acopla el contrato público a tu esquema interno: cada migración de

columnas se filtra a los consumidores. La outbox es justamente la capa de desacoplo; prescinde de ella solo si controlas todos los consumidores y asumes ese acople conscientemente.

- **Te tienta LISTEN/NOTIFY como transporte.** El pub/sub nativo de PostgreSQL es útil como *despertador* del relay (menos polling en vacío: NOTIFY en el trigger de inserción, el relay duerme en LISTEN), pero no como transporte fiable: las notificaciones no se persisten y quien no está escuchando en ese momento las pierde. La tabla sigue siendo la fuente de verdad.
- **Tu plataforma ya te da la primitiva equivalente.** DynamoDB Streams, Cosmos DB change feed o EventBridge Pipes ofrecen "cambios confirmados como stream" gestionado. Si vives en uno de esos ecosistemas, una outbox artesanal puede ser reinventar la rueda.

Checklist de producción

Antes de dar por terminada una implementación de outbox, repasa:

- **Atomicidad:** evento y dato de negocio en la misma transacción, verificado con un test de rollback.
- **Relay:** publica por lotes, espera acks antes de marcar, `SKIP LOCKED` si hay concurrencia, particionado estable por agregado si el orden importa con varios relays.
- **Orden:** key de partición por `aggregate_id` de extremo a extremo; sin marcadores de id sin barrera de snapshot.
- **Consumidores:** idempotentes, idealmente con inbox transaccional; offset confirmado después del commit local.
- **Debezium (si CDC):** `snapshot.mode` correcto, heartbeats, `max_slot_wal_keep_size`, alerta de WAL retenido.
- **Retención:** limpieza por lotes o particiones con drop automático; la outbox no es un event store.
- **Observabilidad:** pendientes, edad del más antiguo y throughput expuestos; alerta sobre la edad, no solo sobre el volumen.
- **Contrato:** `event_version` en cada evento, consumidores que ignoran campos desconocidos, plan para cambios que rompen.

Caso práctico: un pipeline completo en local

Para aterrizar todas las piezas, montemos el sistema entero en local: un servicio de pedidos con FastAPI, PostgreSQL con la outbox, y el relay de polling publicando en Kafka. Es el mismo esqueleto que funciona en producción, en miniatura. La infraestructura, con Docker Compose:

```
# docker-compose.yml
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_DB: orders
      POSTGRES_PASSWORD: dev
    ports: ["5432:5432"]

  kafka:
```

```
    image: apache/kafka:4.0.0   # modo KRaft: ya no hace falta ZooKeeper
    ports: ["9092:9092"]

  relay:
    build: .
    command: python relay.py
    environment:
      DATABASE_URL: postgresql://postgres:dev@db:5432/orders
      KAFKA_BOOTSTRAP: kafka:9092
    depends_on: [db, kafka]
    restart: unless-stopped   # el relay DEBE reiniciarse solo
```

El servicio expone un único endpoint que escribe pedido y evento en la misma transacción. Fíjate en que el endpoint no sabe nada de Kafka: su única dependencia es la base de datos, que ya tenía. Ese es el punto del patrón.

```
# app.py
from fastapi import FastAPI
import json, uuid, psycopg_pool

app = FastAPI()
pool = psycopg_pool.ConnectionPool(DATABASE_URL)

@app.post("/orders", status_code=201)
def create_order(order: OrderIn):
    order_id = str(uuid.uuid4())
    with pool.connection() as conn, conn.transaction():
        conn.execute(
            "INSERT INTO orders (id, customer, total) VALUES (%s, %s, %s)",
            (order_id, order.customer, order.total),
        )
        conn.execute(
            """
            INSERT INTO outbox (aggregate_type, aggregate_id, event_type, payload)
            VALUES ('order', %s, 'order.created', %s)
            """,
            (order_id, json.dumps({
                "order_id": order_id,
                "customer": order.customer,
                "total": str(order.total),
                "event_version": 1,
            })),
        )
    return {"id": order_id}
```

Y el relay es el bucle de `relay_tick()` que vimos en la sección de escalado, envuelto en un proceso con espera adaptativa: si el último lote vino lleno, repite inmediatamente; si vino vacío, duerme un intervalo corto. Ese detalle mantiene la latencia baja en horas punta sin machacar la base de datos de madrugada:

```
# relay.py
import time

IDLE_SLEEP = 0.5   # segundos cuando no hay trabajo

def main() -> None:
```

```
    conn = connect(DATABASE_URL)
    while True:
        try:
            n = relay_tick(conn)
            if n < BATCH:          # lote incompleto: no hay presión
                time.sleep(IDLE_SLEEP)
        except Exception:
            log.exception("tick falló; reintentando")
            time.sleep(2)          # backoff simple: el siguiente tick reintenta
            conn = reconnect(conn)

if __name__ == "__main__":
    main()
```

Con los tres contenedores arriba, la prueba de fuego es un `curl` a `/orders` con Kafka apagado: el pedido se crea igualmente (201), el evento queda pendiente en la outbox, y al levantar Kafka el relay lo publica solo, unos segundos después. Ver esa secuencia con tus propios ojos es la mejor forma de interiorizar qué te está comprando el patrón.

Integración con ORMs: el hook correcto

Los ejemplos con SQL directo hacen evidente dónde está la transacción, pero la mayoría de servicios reales usan un ORM, y la pregunta pasa a ser "¿dónde meto el INSERT en la outbox para que caiga en la misma transacción?". La respuesta genérica: en el mismo *unit of work* que la entidad de negocio, nunca en un hook que se ejecute después del commit.

Con SQLAlchemy, lo natural es acumular eventos en la sesión y volcarlos a la outbox justo antes del flush:

```
from sqlalchemy import event
from sqlalchemy.orm import Session

def collect(session: Session, evt: dict) -> None:
    session.info.setdefault("pending_events", []).append(evt)

@event.listens_for(Session, "before_flush")
def flush_outbox(session, ctx, _):
    for evt in session.info.pop("pending_events", []):
        session.add(Outbox(**evt))   # entra en la MISMA transacción

# En el código de negocio:
order = Order(customer="ada", total=99)
db.add(order)
collect(db, order_created_event(order))
db.commit()   # pedido + evento, atómico
```

En Django, el equivalente es escribir el modelo `Outbox` dentro del bloque `transaction.atomic()` del caso de uso. El anti-patrón a evitar en ambos frameworks es el mismo: señales o hooks de tipo `after_commit` / `post_save` que publican directamente al broker. Parecen convenientes y reintroducen exactamente el dual write que la outbox elimina —si el proceso muere tras el commit y antes de la señal, el evento no existe—.

Payloads grandes: el patrón claim check

La columna `payload JSONB` invita a guardar documentos enormes, y PostgreSQL lo permitirá sin quejarse: comprime y trocea valores grandes en almacenamiento TOAST de forma transparente. Pero una outbox de alto tráfico con payloads de cientos de kilobytes multiplica el WAL (y con CDC, todo ese WAL viaja por el slot de replicación), engorda los mensajes de Kafka —cuyo límite por defecto ronda 1 MB— y hace la limpieza más cara.

La solución clásica es el *claim check*: el evento lleva una referencia, no el contenido. Sube el documento a un almacén de objetos (S3, GCS) *antes* de la transacción, y guarda en el payload solo la URL y los metadatos que los consumidores necesitan para decidir si les interesa:

```
{
  "event_type": "invoice.generated",
  "invoice_id": "inv_8843",
  "customer_id": "cus_112",
  "total": "1240.00",
  "document_ref": "s3://invoices/2026/07/inv_8843.pdf",
  "event_version": 1
}
```

La regla práctica: si el payload supera unas decenas de kilobytes de forma habitual, referencia en lugar de incrustar. Y sé disciplinado con el orden de las escrituras: primero el objeto en S3, después la transacción con el evento. Al revés, un consumidor rápido puede recibir el evento antes de que el documento exista.

Outbox, sagas y event sourcing: cómo encajan

Tres términos que aparecen juntos en cualquier búsqueda sobre eventos, y que conviene no confundir:

- **La outbox es un mecanismo de publicación fiable.** No define qué eventos publicas ni cómo reaccionan los consumidores; solo garantiza que lo confirmado se publica.
- **Una saga es una coordinación de negocio.** Un proceso de varios pasos entre servicios —reservar stock, cobrar, programar envío— donde cada paso emite eventos y los fallos disparan compensaciones. Las sagas *necesitan* mensajería fiable para no quedarse a medias, y la outbox es la forma estándar de dársela: cada paso escribe su estado y su evento saliente en la misma transacción local.
- **Event sourcing es un modelo de persistencia.** El estado *es* la secuencia de eventos; no hay tabla de pedidos que actualizar, se reconstruye reproduciendo el log. Si haces event sourcing completo, la outbox como tabla separada sobra: el event store ya es la fuente de verdad y publicar es leer de él. La outbox es, en cierto sentido, la dosis mínima de event sourcing que puedes adoptar sin cambiar tu modelo de datos.

Para la mayoría de equipos, la progresión sana es exactamente esa: modelo relacional clásico, outbox para publicar, sagas cuando aparezcan procesos multi-servicio, y event sourcing solo si el dominio de verdad lo pide (auditoría total, reconstrucción temporal, dominios financieros). Saltar directamente al final es la receta habitual para un sistema que nadie sabe operar.

Preguntas frecuentes

¿La outbox no duplica mis escrituras? Sí: cada operación de negocio escribe una fila extra. En la práctica el coste es marginal —un INSERT pequeño en una transacción que ya existía— y desaparece en el ruido comparado con el coste de los índices de las tablas de negocio. Si tu base de datos va justa de IOPS, el problema no te lo creará la outbox.

¿Puedo usar la outbox con MySQL, SQL Server u Oracle? El patrón es agnóstico: cualquier base con transacciones ACID sirve. Cambian los detalles del relay (`SKIP LOCKED` existe en MySQL 8 y Oracle; SQL Server usa hints `READPAST`) y el conector de Debezium correspondiente si optas por CDC.

¿Qué pasa si la propia outbox falla al insertar? Entonces la transacción entera falla y el llamante recibe un error: exactamente el comportamiento correcto. La outbox nunca puede estar "menos disponible" que tu base de datos, porque es tu base de datos. Compáralo con el dual write, donde el broker caído te deja elegir entre rechazar la operación o perder el evento.

¿Cada servicio necesita su propia outbox? Sí, una por base de datos. La outbox vive junto a los datos cuya consistencia protege; compartir una outbox central entre servicios reintroduciría el acoplamiento (y las transacciones distribuidas) que intentabas evitar.

¿Borro las filas de la outbox al publicarlas, o las marco? Marcar con `processed_at` y limpiar después es el valor por defecto más operable: conserva una ventana forense corta (impagable cuando un consumidor jura que algo nunca llegó), hace barato el UPDATE del relay y delega el borrado en un job que tú controlas. Borrar inmediatamente mantiene la tabla diminuta, pero tira esa ventana de depuración y convierte cada publicación en una operación intensiva en escrituras. Con Debezium hay una tercera opción: hacer DELETE dentro de la propia transacción del relay y dejar que CDC lea igualmente el INSERT del WAL —la tabla queda casi vacía y el WAL es tu registro—. Las tres son válidas; lo importante es elegir deliberadamente y no por accidente.

Polling o CDC: guía de decisión

Las dos variantes del relay aparecieron ya en el artículo; después de operar ambas, estos son los criterios que de verdad inclinan la balanza:

- **Empieza con polling si dudas.** Son cien líneas de código que entiendes de arriba abajo, se depuran con un SELECT y no añaden infraestructura. Para latencias objetivo de segundos y volúmenes de hasta decenas de eventos por segundo, es difícil justificar nada más complejo.
- **CDC gana cuando el polling empieza a doler.** Latencia objetivo por debajo del segundo, volumen alto sostenido (el polling agresivo carga la base de datos), o varias outboxes en varios servicios donde un equipo de plataforma puede amortizar el coste de operar Kafka Connect para todos.
- **CDC exige madurez operativa.** Kafka Connect es otro sistema distribuido: versiones, rebalanceos, slots de replicación que vigilar, y depurar un conector es más opaco que depurar tu propio bucle. Si tu equipo no opera Kafka con soltura todavía, el polling es la opción honesta.
- **El código de negocio no cambia.** La tabla outbox y la escritura transaccional son idénticas en ambos casos, así que migrar de polling a CDC más adelante es un cambio de infraestructura, no de aplicación. Empezar simple no te encierra.

Replays y backfills: cuando un consumidor llega tarde

Tarde o temprano aparece la petición: "somos un servicio nuevo y necesitamos los eventos históricos", o peor, "el consumidor tuvo un bug tres días y procesó basura; hay que reprocesar". Conviene tener respuesta preparada, porque improvisarla con producción ardiendo sale mal.

Lo primero es una advertencia: **la outbox no es un event store**. Con la retención de días que recomendamos, no puede servir replays históricos. Las opciones reales:

- **Retención larga en el broker.** Kafka puede retener topics semanas o meses (o indefinidamente con compactación por key). Un consumidor nuevo simplemente arranca desde el offset más antiguo y se pone al día. Es la solución más limpia si el volumen lo permite: el broker ya es el archivo.
- **Reconstruir desde el estado.** Para poblar un servicio nuevo, a menudo no necesitas los eventos históricos sino el estado actual: un job puntual que recorre la tabla `orders` y emite eventos sintéticos `order.snapshot` hacia el topic (o directamente a la outbox, en lotes controlados). Marca estos eventos como snapshot para que los consumidores puedan distinguirlos de los de negocio.
- **Reprocesado del consumidor.** Si el bug estuvo en el consumidor, no toques el productor: rebobina el offset del grupo de consumo (`kafka-consumer-groups --reset-offsets`) al punto anterior al bug. Aquí la inversión en idempotencia paga dividendos: reprocesar tres días de eventos sobre un consumidor idempotente es aburrido, que es exactamente lo que quieres.

Sea cual sea la vía, hazla *a ritmo controlado*. Un replay masivo a velocidad máxima puede saturar a los consumidores sanos que comparten topic; los replays serios se hacen con throttling y monitorizando el lag de todos los grupos afectados.

Multi-tenancy y datos sensibles en la outbox

Dos consideraciones que suelen llegar tarde al diseño y son baratas de incorporar al principio:

Tenant como ciudadano de primera. En un SaaS multi-tenant, añade `tenant_id` como columna propia de la outbox (no solo dentro del payload). Sirve para tres cosas: enrutar a topics por tenant si algún cliente grande exige aislamiento, aplicar retenciones o cuotas distintas por plan, y —la importante— poder responder a "borra todos los datos del cliente X" también en la outbox y en los topics, no solo en las tablas de negocio.

La outbox también es un dato personal. Los payloads copian datos de negocio: emails, direcciones, importes. Eso significa que las políticas de acceso, cifrado y retención que aplicas a `orders` aplican igual a `outbox` y a los topics que alimenta. Los equipos que cifran columnas sensibles en las tablas de negocio y publican esas mismas columnas en claro en un topic con retención infinita tienen un problema de cumplimiento esperando a ser descubierto. Reglas prácticas: publica el mínimo necesario (los consumidores pueden pedir el resto por API con su propia autorización), aplica la retención del broker con la misma seriedad que la de la base de datos, y si el dominio maneja datos especialmente sensibles, considera payloads cifrados por tenant —el borrado de la clave equivale al borrado de los datos en todos los topics a la vez, el llamado *crypto-shredding*—.

Rendimiento: cuánto cuesta de verdad

Cerremos con números orientativos, porque las decisiones de arquitectura se toman mejor con órdenes de magnitud en la cabeza que con miedos difusos:

- **El INSERT extra en la outbox** añade típicamente décimas de milisegundo a una transacción que ya escribía: es una fila pequeña en una tabla con un único índice parcial. En un endpoint que tarda 20–50 ms, el sobrecoste desaparece en el ruido de la red.
- **Un relay de polling con lotes de 500** y un intervalo de reposo de medio segundo sostiene miles de eventos por segundo en hardware modesto; el cuello de botella suele ser el `flush()` hacia Kafka, no PostgreSQL. Si el lag crece con la CPU del relay aburrida, sube el tamaño de lote antes de añadir procesos.
- **El índice parcial es la diferencia entre O(pendientes) y O(tabla).** Con él, el SELECT del relay toca solo las filas sin procesar aunque la tabla acumule millones; sin él, cada tick escanea historia muerta. Si el `EXPLAIN` del relay muestra un Seq Scan, revisa que la condición del índice y la del WHERE coincidan exactamente.
- **La latencia de extremo a extremo** (commit → consumidor) queda dominada por el intervalo de polling en la variante simple (percentil alto ≈ intervalo + red) y baja a decenas de milisegundos con CDC. Pon un número objetivo delante del equipo antes de elegir: "los emails de confirmación pueden tardar 5 segundos" y "el fraude necesita saberlo en 100 ms" llevan a arquitecturas distintas, y las dos están bien.

Y el consejo transversal: **mide tu caso antes de optimizar el de otro**. La mitad de la complejidad que se añade a las outboxes en producción responde a volúmenes que el sistema jamás alcanzará.

Tres fallos reales y su moraleja

Para terminar la parte nueva, tres escenarios de fallo que se repiten en equipos distintos con una regularidad sospechosa. Los tres son evitables con lo que ya has leído; verlos narrados ayuda a reconocerlos a tiempo.

El relay zombi. Un equipo despliega el relay como sidecar del API. Un despliegue deja dos versiones del API corriendo un rato y, con ellas, dos relays compitiendo sin `SKIP LOCKED`: cada evento se publica dos veces durante veinte minutos. Nadie lo nota hasta que facturación pregunta por qué hay abonos duplicados. Moralejas: el relay es un componente con su propio ciclo de vida, y la idempotencia del consumidor no es opcional ni siquiera "de momento" —el día que la necesitas ya es tarde para añadirla—.

El disco que se llenó solo. Otro equipo adopta CDC, funciona de maravilla y se olvida. Meses después migran Kafka Connect y el conector queda parado en un entorno secundario que nadie mira. El slot de replicación retiene WAL durante nueve días hasta llenar el disco de la base de datos... de producción, porque el entorno "secundario" compartía instancia. Moralejas: todo slot necesita una alerta de WAL retenido y un `max_slot_wal_keep_size`, y los conectores olvidados son fugas de disco con temporizador.

La limpieza que tumbó la réplica. Un tercer equipo deja crecer la outbox dos años hasta los cuatrocientos millones de filas y decide arreglarlo un viernes con `DELETE FROM outbox WHERE processed_at IS NOT NULL`. La sentencia genera tal volumen de WAL que la réplica de lectura se

retrasa cuarenta minutos y el failover automático entra en pánico. Moralejas: la retención se diseña el primer día, cuando es barata; y cualquier operación masiva sobre una tabla caliente se hace por lotes, midiendo el lag de réplica entre lote y lote.

El patrón común de los tres: la outbox funcionó exactamente como promete —ningún evento se perdió—. Lo que falló fue siempre la operación alrededor. Esa es la naturaleza de este patrón: es sencillo de implementar y exige disciplina para operarlo, no al revés.

Conclusión

El transactional outbox convierte "guardar y publicar" en una sola escritura atómica más un relay simple y reintentable. Es uno de los patrones con mejor relación coste/beneficio en arquitecturas orientadas a eventos: una tabla, un worker y una regla de oro (idempotencia en el consumidor) eliminan toda una clase de bugs de consistencia que solo aparecen en producción. Si ya aplicas los patrones de reintentos con backoff e idempotencia en APIs de esta serie, la outbox es la pieza que faltaba del lado del productor.

English — The Transactional Outbox Pattern

Why dual writes lose events

Almost every service eventually needs to tell other systems that something happened: an order was created, a payment was charged, a user changed their email. The naive implementation performs two writes in a row: first it saves the data to the database, then it publishes the event to Kafka, RabbitMQ, or SQS.

```
# ANTI-PATTERN: dual write
def create_order(data):
    db.insert("orders", data)              # write 1: database
    broker.publish("order.created", data)  # write 2: broker
```

The problem is that no transaction covers both writes. There are two ways to fail, and both hurt:

- **The order is saved but the event never goes out.** The process crashes, the broker is unreachable, or a deploy lands right between those two lines. The warehouse never finds out and the order is left orphaned.
- **The event is published but the transaction rolls back.** If you publish inside the transaction and it fails afterwards, the rest of the system reacts to something that never happened.

Retries don't fix this: retrying the publish after committing still leaves a window where the process can die. The solution isn't better retries — it's removing the double write altogether.

How the transactional outbox pattern works

The idea is simple: instead of writing to two systems, write to just one. The event is stored in an `outbox` table in your own database, inside the same transaction that modifies the business data. Since it's a single ACID transaction, either both things commit or neither does.

1. The service opens a transaction, writes the order to `orders` and the event to `outbox`, and commits.
2. A separate process (the *relay*) reads the outbox table and publishes the events to the broker.
3. Once the publish is acknowledged, the event is marked as processed or deleted.

The broker can go down, the relay can restart, and nothing is lost: the event sits in the database waiting its turn. The resulting guarantee is **at-least-once** delivery — never zero times.

The outbox table

```
CREATE TABLE outbox (
  id             BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  aggregate_type TEXT        NOT NULL,  -- "order", "payment"
  aggregate_id   TEXT        NOT NULL,  -- entity id
  event_type     TEXT        NOT NULL,  -- "order.created"
```

```
  payload        JSONB       NOT NULL,
  created_at     TIMESTAMPTZ NOT NULL DEFAULT now(),
  processed_at   TIMESTAMPTZ            -- NULL = pending
);

-- Partial index: the relay only scans pending rows
CREATE INDEX idx_outbox_pending
  ON outbox (id) WHERE processed_at IS NULL;
```

`aggregate_id` matters: later it becomes the Kafka partition key that preserves per-entity ordering. The partial index keeps the scan cheap even when the table accumulates millions of already-processed rows.

Writing the event in the same transaction

```
import json
import uuid
import psycopg

def create_order(conn: psycopg.Connection, order: dict) -> str:
    order_id = str(uuid.uuid4())
    with conn.transaction():  # one single ACID transaction
        conn.execute(
            "INSERT INTO orders (id, customer_id, total) VALUES (%s, %s, %s)",
            (order_id, order["customer_id"], order["total"]),
        )
        conn.execute(
            """
            INSERT INTO outbox (aggregate_type, aggregate_id, event_type, payload)
            VALUES ('order', %s, 'order.created', %s)
            """,
            (order_id, json.dumps({
                "event_id": str(uuid.uuid4()),  # for consumer-side dedup
                "version": 1,
                "order_id": order_id,
                "customer_id": order["customer_id"],
                "total": order["total"],
            })),
        )
    return order_id
```

Note the `event_id` inside the payload: consumers will use it to deduplicate, because with at-least-once delivery duplicates aren't an edge case — they're part of the contract.

Option A: polling relay

The simplest relay is a worker that queries the table every few hundred milliseconds. The key piece is `FOR UPDATE SKIP LOCKED`: it lets you run several workers in parallel without them stepping on each other, because each one locks a different set of rows.

```
import time
import psycopg
```

```
from confluent_kafka import Producer

producer = Producer({
    "bootstrap.servers": "localhost:9092",
    "enable.idempotence": True,
    "acks": "all",
})

def relay_batch(conn: psycopg.Connection) -> int:
    with conn.transaction():
        rows = conn.execute(
            """
            SELECT id, aggregate_id, event_type, payload::text
            FROM outbox
            WHERE processed_at IS NULL
            ORDER BY id
            LIMIT 100
            FOR UPDATE SKIP LOCKED
            """
        ).fetchall()

        for _id, aggregate_id, event_type, payload in rows:
            producer.produce(topic=event_type, key=aggregate_id, value=payload)

        producer.flush(timeout=10)  # wait for Kafka's ack

        if rows:
            conn.execute(
                "UPDATE outbox SET processed_at = now() WHERE id = ANY(%s)",
                ([r[0] for r in rows],),
            )
    return len(rows)

while True:
    if relay_batch(conn) == 0:
        time.sleep(0.2)  # no pending work: wait before polling again
```

If the worker dies after publishing but before the UPDATE, the next cycle will publish those rows again. That's the correct behavior: duplicates yes, losses no.

Polling adds an average latency of half the interval plus some database load, but it requires no new infrastructure. For most systems it's more than enough.

Option B: CDC with Debezium

The alternative is capturing changes straight from the transaction log (WAL in PostgreSQL, binlog in MySQL) with a change data capture tool. Debezium is the de facto standard: it detects each INSERT into the outbox as soon as it commits and publishes it to Kafka within milliseconds, with no extra queries against the database.

```
{
  "name": "outbox-connector",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "db",
```

```
    "database.dbname": "shop",
    "table.include.list": "public.outbox",
    "plugin.name": "pgoutput",
    "transforms": "outbox",
    "transforms.outbox.type": "io.debezium.transforms.outbox.EventRouter",
    "transforms.outbox.table.field.event.key": "aggregate_id",
    "transforms.outbox.route.by.field": "event_type"
  }
}
```

Debezium's `EventRouter` was built precisely for this pattern: it extracts the payload, uses `aggregate_id` as the message key, and routes each event type to its topic. With CDC you don't even need the `processed_at` column: Debezium reads from the log, not the table, so you can delete the row right after inserting it.

Which one should you pick? With latency requirements under ~100 ms and Kafka already deployed, go CDC. If your stack is simpler or a short polling interval is acceptable, polling wins on sheer operational simplicity: Debezium means Kafka Connect, logical replication, lag monitoring, and schema evolution.

The consumer: idempotency is mandatory

With at-least-once delivery, every consumer will see repeated events sooner or later. The standard defense is recording processed `event_id` values in the same transaction that applies the effect:

```
def handle_order_created(conn: psycopg.Connection, event: dict) -> None:
    with conn.transaction():
        inserted = conn.execute(
            """
            INSERT INTO processed_events (event_id) VALUES (%s)
            ON CONFLICT (event_id) DO NOTHING
            """,
            (event["event_id"],),
        ).rowcount
        if inserted == 0:
            return  # duplicate: already processed
        reserve_stock(conn, event["order_id"])
```

Common pitfalls

- **Publishing inside the transaction and calling it an outbox.** If you call the broker before committing, you still have a dual write with extra steps. The relay must be a separate process reading from the table.
- **Forgetting cleanup.** An ever-growing outbox degrades vacuum and backups. Delete processed rows in batches with a periodic job: `DELETE FROM outbox WHERE id IN (SELECT id FROM outbox WHERE processed_at < now() - interval '7 days' LIMIT 10000)`.
- **Assuming global ordering.** Neither Kafka nor the relay guarantees ordering across different entities. What you can guarantee is per-entity ordering by using `aggregate_id` as the partition key.

- **Payloads that only carry an id.** If the consumer has to query the row, it may read a state newer than the event's. Include everything the consumer needs in the payload (event-carried state).
- **Consumers without deduplication.** At-least-once means duplicates will arrive. A consumer that doesn't deduplicate will charge the same order twice when you least expect it.

Best practices

- Generate a unique `event_id` when writing to the outbox and propagate it all the way to the final consumer.
- Version your payloads (`"version": 1`) so you can evolve the schema without breaking consumers.
- Monitor two metrics: pending rows (`processed_at IS NULL`) and the age of the oldest pending event. They're your key alerts that the relay is falling behind.
- Enable `enable.idempotence` on the Kafka producer so the publishing layer doesn't add extra duplicates.
- Start with polling; migrate to CDC when polling latency or load becomes a measurable problem, not before.

Ordering guarantees: what you can promise and what you can't

One of the first questions that comes up when taking the outbox to production is what ordering consumers will see. The short answer: you can guarantee **per-aggregate** ordering, and you shouldn't try to guarantee global ordering. Understanding why will save you weeks of debugging.

Global ordering — every event in the system arriving in the exact sequence it was written — would require a single serialization point at every stage: one relay, one Kafka partition, one consumer. That turns your pipeline into a single-threaded queue. Almost no business actually needs it: what matters is that events *for the same order* arrive in order, not that an event for order A arrives before one for order B.

Per-aggregate ordering takes three pieces working together:

1. **A sequence key in the outbox.** The table's `id BIGINT GENERATED ALWAYS AS IDENTITY` already provides it: it reflects commit order, with one caveat we'll cover below.
2. **Publishing with a partition key.** In Kafka, all events with the same key land on the same partition, and ordering within a partition is stable. Use `aggregate_id` as the key.
3. **A consumer that processes each partition sequentially.** If you parallelize processing within a partition, you destroy the ordering you worked so hard to preserve.

```
from confluent_kafka import Producer

producer = Producer({
    "bootstrap.servers": "kafka:9092",
    "enable.idempotence": True,   # avoids duplicates from producer retries
    "acks": "all",                # wait for confirmation from all ISR replicas
```

```
})

def publish(event: dict) -> None:
    producer.produce(
        topic=f"{event['aggregate_type']}.events",
        key=event["aggregate_id"],       # same entity -> same partition
        value=json.dumps(event["payload"]),
        headers={"event_type": event["event_type"], "event_id": str(event["id"])},
    )
```

`enable.idempotence` deserves a clarification: it eliminates the duplicates created by *the producer itself* when it retries a send that actually made it through. It does not make the pipeline exactly-once end to end — the relay can still publish the same event twice if it dies between the broker's ack and the `processed_at` UPDATE — but it does close off one free source of duplicates.

The commit-order trap

There's a subtle detail that bites in highly concurrent systems: the outbox `id` is assigned at INSERT time, but the row becomes visible at COMMIT time, and the two orders can disagree. Transaction T1 can insert event 100, take a while to commit, while T2 inserts 101 and commits first. A relay that reads "everything pending with id > last_processed" can see 101, advance its watermark, and skip 100 forever.

The usual fixes, from simplest to most robust:

- **Don't use an id watermark; use `processed_at IS NULL`.** The relay in this article already does this: it never skips rows because it doesn't assume ids arrive in order. That's why the partial-index pattern is the recommended one.
- **`pg_current_snapshot()` as a barrier.** If you need a watermark for performance, only read rows whose `id` is below the current snapshot's `xmin`, guaranteeing no in-flight transaction can still insert smaller ids.
- **CDC.** Debezium reads the WAL, and the WAL is commit-ordered by construction. The problem disappears at the root.

Scaling the relay without breaking anything

Intuition says "lots of events, spin up more relays." Experience says otherwise: **a single well-built relay handles far more than you think**. Batched publishing to Kafka reaches tens of thousands of events per second from one process; most systems don't generate a fraction of that. Before scaling horizontally, measure.

The first level of scaling isn't adding processes — it's **batching**:

```
BATCH = 500  # reasonable starting point; tune by watching lag and latency

def relay_tick(conn) -> int:
    with conn.transaction():
        rows = conn.execute(
            """
            SELECT id, aggregate_type, aggregate_id, event_type, payload
            FROM outbox
            WHERE processed_at IS NULL
```

```
            ORDER BY id
            LIMIT %s
            FOR UPDATE SKIP LOCKED
            """,
            (BATCH,),
        ).fetchall()

        if not rows:
            return 0

        for r in rows:
            publish(row_to_event(r))
        producer.flush(timeout=10)  # wait for broker acks BEFORE marking

        conn.execute(
            "UPDATE outbox SET processed_at = now() WHERE id = ANY(%s)",
            ([r[0] for r in rows],),
        )
    return len(rows)
```

Two decisions in this code matter more than they look:

- **The `flush()` comes before the UPDATE.** Marking as processed before delivery is confirmed turns your at-least-once into at-most-once: if the broker rejects the batch afterwards, the event is lost. The correct order is publish, wait for acks, mark.
- **Batch size is a trade-off.** Tiny batches burn CPU on query and commit overhead; huge batches stretch the transaction, hold locks longer and, if something fails midway, the retry republishes the whole batch (more duplicates). Between 100 and 1,000 usually works; tune by watching lag.

Multiple relays: when and how

`FOR UPDATE SKIP LOCKED` lets several relays work simultaneously without stepping on each other: each locks a different batch and skips rows already locked. It's the right tool, but it has a silent cost: **it breaks global publish ordering**. Relay A can publish events 200–299 before relay B finishes 100–199. If you keep the partition key on `aggregate_id`, disorder between different aggregates is harmless... unless two events for the *same* aggregate land in batches held by different relays.

If you need multiple relays and strict per-aggregate ordering, partition the work stably instead of competing for rows:

```
-- Relay i of N processes only its stable slice of aggregates
SELECT id, aggregate_type, aggregate_id, event_type, payload
FROM outbox
WHERE processed_at IS NULL
  AND mod(abs(hashtext(aggregate_id)), %(n_relays)s) = %(relay_index)s
ORDER BY id
LIMIT 500
FOR UPDATE SKIP LOCKED;
```

Each aggregate always belongs to the same relay, so its events go out in order, and you still spread the load. It's the same principle Kafka uses with partitions, applied one step earlier.

One last operational tip: **deploy the relay as a unit independent from the API**. It can live in the same repository, but with its own deployment: it scales on its own, restarts without dropping HTTP traffic, and its metrics don't blend into the web service's.

Delivery semantics: why exactly-once is a contract, not a feature

Let's say it plainly: **the outbox gives you at-least-once**. Every event confirmed in the business transaction will be published one or more times. "Exactly once" end to end — from your database to the effect in the consumer — is not something any single tool provides, no matter what the marketing says: there is always a window between "I published" and "I recorded that I published" where a crash produces a repeat.

What you can build is **effectively-once processing**: the event may arrive duplicated, but its effect is applied once. The responsibility is shared:

- **The producer** (relay) minimizes duplicates: Kafka's idempotent producer, flush before marking, sensible batch sizes.
- **The transport** preserves per-partition order and doesn't lose acknowledged messages (replication, `acks=all`).
- **The consumer** deduplicates. This is where the game is won or lost, which is why it deserves its own pattern.

The transactional inbox pattern: the other half of the contract

The article already established that the consumer must be idempotent. The *transactional inbox* is the structured version of that idea: the outbox trick, applied on the receiving side. Instead of "process, then remember I processed" as two separate steps (another dual write, now in the consumer), both happen in a single local transaction.

```
CREATE TABLE inbox (
  event_id     TEXT PRIMARY KEY,       -- event id, comes in the message
  consumer     TEXT NOT NULL,          -- logical consumer name
  received_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);
-- If several consumers share a database:
-- PRIMARY KEY (event_id, consumer)
```

```
def handle_message(conn, msg) -> None:
    event_id = dict(msg.headers() or {}).get("event_id", b"").decode()
    with conn.transaction():
        inserted = conn.execute(
            """
            INSERT INTO inbox (event_id, consumer)
            VALUES (%s, %s)
            ON CONFLICT DO NOTHING
            """,
            (event_id, "warehouse-service"),
        ).rowcount
```

```
        if inserted == 0:
            return  # duplicate: already processed, empty transaction, exit

        apply_business_logic(conn, json.loads(msg.value()))  # same transaction

    consumer.commit(msg)  # Kafka offset is committed AFTER the local commit
```

The elegance of the pattern is that the `inbox` INSERT and the business logic share a transaction: both apply or neither does. If the process dies after the local commit but before committing the offset, Kafka redelivers the message, the INSERT hits the primary key, and the handler exits cleanly with no repeated effects. `ON CONFLICT DO NOTHING` with `rowcount` is the check and the lock in a single atomic statement.

The inbox table needs retention too (a job deleting rows older than, say, 30 days), and that window defines your real deduplication horizon: a duplicate arriving later than the retention will be processed again. Choose the window by looking at how late a message can legitimately arrive in your system — reprocessing, partition replays, restores — and add margin.

Debezium in production: the complete configuration

The earlier section introduced CDC as an alternative to polling; this one gets into what you actually have to configure to make it work. The stack is PostgreSQL with logical replication, Kafka Connect and the Debezium connector (the 3.x series is current; Debezium 3.4 shipped in late 2025 and builds on Kafka Connect 4). The output plugin is `pgoutput`, native to PostgreSQL since version 10: nothing extra to install on the database server.

First, the database. You need `wal_level = logical` (requires a restart), a user with replication permission, and a *publication* exposing only the outbox table:

```
-- postgresql.conf: wal_level = logical  (then restart)

CREATE USER debezium WITH REPLICATION PASSWORD '...';
GRANT SELECT ON outbox TO debezium;

-- Publish ONLY the outbox: the rest of the schema is none of the connector's business
CREATE PUBLICATION outbox_pub FOR TABLE outbox;
```

Then the connector. This configuration registers a connector that reads the outbox and applies the **outbox event router**, the SMT (Single Message Transform) Debezium ships specifically for this pattern: it extracts the payload, routes each event to its `aggregate_type`'s topic, and uses `aggregate_id` as the message key — per-aggregate ordering is handled out of the box.

```
{
  "name": "outbox-connector",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "db", "database.port": "5432",
    "database.user": "debezium", "database.password": "...",
    "database.dbname": "orders", "topic.prefix": "orders-svc",

    "plugin.name": "pgoutput",
    "publication.name": "outbox_pub",
```

```
    "slot.name": "outbox_slot",
    "table.include.list": "public.outbox",
    "snapshot.mode": "no_data",

    "transforms": "outbox",
    "transforms.outbox.type": "io.debezium.transforms.outbox.EventRouter",
    "transforms.outbox.table.field.event.key": "aggregate_id",
    "transforms.outbox.route.by.field": "aggregate_type",
    "transforms.outbox.table.expand.json.payload": "true",

    "heartbeat.interval.ms": "10000",
    "tombstones.on.delete": "false"
  }
}
```

Three of those options prevent real incidents:

- **`snapshot.mode: no_data`.** Without it, the connector tries to snapshot the table on startup. For an outbox, republishing already-processed history makes no sense.
- **`heartbeat.interval.ms`.** Protects against logical replication's most treacherous failure mode: *slot bloat*. A replication slot forces PostgreSQL to retain WAL until the consumer confirms it. If your outbox has little traffic but other tables in the database write heavily, the slot advances slowly and retained WAL grows until the disk fills — a 3 a.m. classic. Heartbeats force periodic slot advances even when there are no events.
- **`tombstones.on.delete: false`.** If you delete outbox rows after processing (or in the cleanup job), you don't want every DELETE emitting a tombstone into the destination topic.

And one operational rule that isn't in the config: **monitor the slot**. This query belongs on your dashboard, with an alert if it grows unchecked:

```
SELECT slot_name, active,
       pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS retained_wal
FROM pg_replication_slots;
```

If a connector stays down for days (broken deploy, Kafka Connect outage), its slot's retained WAL grows without bound. PostgreSQL offers `max_slot_wal_keep_size` as a seatbelt: it caps how much WAL a slot can retain before being invalidated. Set it; losing a slot and rebuilding the connector is annoying, but filling the production database's disk is far worse.

Maintaining the outbox: the table that only grows

The outbox is an insert-only table with a peculiar access pattern: written constantly, only the recent part is ever read, and nobody cares about the old rows. Without maintenance it accumulates millions of dead rows, autovacuum spends its life cleaning it, and its bloat leaks into your backups and restore times.

There are two strategies, and the choice depends on volume:

Batched deletion (low and medium volume)

```
-- Run every few minutes (pg_cron, or a job inside the relay itself)
WITH victims AS (
  SELECT id FROM outbox
  WHERE processed_at IS NOT NULL
    AND processed_at < now() - interval '7 days'
  ORDER BY id
  LIMIT 10000
)
DELETE FROM outbox WHERE id IN (SELECT id FROM victims);
```

The `LIMIT` is the important part: deleting millions of rows in one statement holds locks for minutes, spikes WAL and can knock over your replica. Deleting in batches of ten thousand, looping until nothing is left, does the same job without drama. The 7-day grace period leaves room to debug recent incidents with the events still at hand.

Time-range partitioning (high volume)

Past hundreds of thousands of events per day, deleting stops scaling: every DELETE creates dead rows autovacuum has to collect. The alternative is making deletion free:

```
CREATE TABLE outbox (
  id           BIGINT GENERATED ALWAYS AS IDENTITY,
  aggregate_type TEXT NOT NULL,
  aggregate_id TEXT NOT NULL,
  event_type   TEXT NOT NULL,
  payload      JSONB NOT NULL,
  created_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
  processed_at TIMESTAMPTZ,
  PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);

-- One partition per day (pg_partman automates creation)
CREATE TABLE outbox_2026_07_10 PARTITION OF outbox
  FOR VALUES FROM ('2026-07-10') TO ('2026-07-11');

-- Cleanup is now instantaneous and creates no dead rows:
DROP TABLE outbox_2026_07_03;
```

`DROP TABLE` on an old partition is a metadata operation: milliseconds, zero bloat, zero autovacuum work. Extensions like `pg_partman` create and drop partitions automatically under whatever retention you define. The price is a bit more schema complexity (the primary key must include the partitioning column) and making sure the relay or Debezium reads from the parent table.

Observability: the three metrics that matter

A healthy outbox is invisible; a sick one gets discovered late and badly — usually because a consuming team asks why they haven't received events for an hour. Three metrics turn that late discovery into an early alert:

- **`outbox_pending_events`:** how many rows await publishing. The absolute value matters less than the trend: sustained growth means the relay can't keep up or is dead.
- **`outbox_oldest_pending_seconds`:** the age of the oldest pending event. This is your real end-to-end lag and your best alerting signal: if it exceeds your delivery SLO (say, 60 seconds), something is wrong even if the pending count looks small.
- **`outbox_published_total`:** a counter of published events. Its derivative is throughput; if it drops to zero under normal write traffic, the relay is down even if its process looks "alive".

```
from prometheus_client import Gauge, Counter

PENDING = Gauge("outbox_pending_events", "Rows awaiting publish")
OLDEST  = Gauge("outbox_oldest_pending_seconds", "Age of oldest pending row")
PUBLISHED = Counter("outbox_published_total", "Events published")

def scrape_outbox_metrics(conn) -> None:
    row = conn.execute(
        """
        SELECT count(*),
               coalesce(extract(epoch FROM now() - min(created_at)), 0)
        FROM outbox WHERE processed_at IS NULL
        """
    ).fetchone()
    PENDING.set(row[0])
    OLDEST.set(row[1])
    # PUBLISHED.inc(n) is called from relay_tick() with the batch size
```

With that, two alerts cover 95% of incidents: `outbox_oldest_pending_seconds > 60` for 5 minutes (the pipeline is stuck) and scrape absence (the relay isn't even responding). If you use Debezium, add the retained-WAL slot metric from the previous section.

Traces that cross the outbox

An annoying side effect of the pattern is that it breaks distributed tracing: the HTTP request that created the order and the consumer that reacted to the event show up as disconnected traces, and "why did the email take 40 seconds?" becomes hard to answer. The fix is treating trace context as part of the event: store the `traceparent` header (W3C Trace Context) in the outbox row when writing it, propagate it as a Kafka message header, and restore it in the consumer as a *link* to the original trace. OpenTelemetry supports this in all its SDKs; the cost is one more text column.

How to test an outbox without fooling yourself

The classic mistake when testing this pattern is mocking the database or the broker: the test always passes and proves nothing, because what the outbox guarantees is precisely real transactional behavior. The atomicity of "order + event" can only be verified against a real PostgreSQL. With Testcontainers, spinning one up per test session costs a few seconds:

```
import pytest, psycopg
from testcontainers.postgres import PostgresContainer

@pytest.fixture(scope="session")
def pg():
    with PostgresContainer("postgres:17") as container:
        conn = psycopg.connect(container.get_connection_url())
        conn.execute(SCHEMA_SQL)  # orders + outbox
        yield conn

def test_order_and_event_are_atomic(pg):
    create_order(pg, {"customer": "ada", "total": 99})
    orders = pg.execute("SELECT count(*) FROM orders").fetchone()[0]
    events = pg.execute("SELECT count(*) FROM outbox WHERE processed_at IS
NULL").fetchone()[0]
    assert (orders, events) == (1, 1)

def test_rollback_leaves_no_orphan_event(pg):
    with pytest.raises(ValueError):
        create_order(pg, {"customer": "ada", "total": -5})  # validation fails INSIDE the
tx
    assert pg.execute("SELECT count(*) FROM outbox").fetchone()[0] == 0

def test_relay_does_not_mark_when_broker_fails(pg, broker_down):
    create_order(pg, {"customer": "ada", "total": 99})
    with pytest.raises(BrokerError):
        relay_tick(pg)
    # The event MUST remain pending for the next tick
    assert pg.execute(
        "SELECT count(*) FROM outbox WHERE processed_at IS NULL"
    ).fetchone()[0] == 1
```

The third test catches the most bugs: it verifies that a publish failure leaves the event pending instead of marking it — exactly the window where a careless implementation loses data. Complete it with its mirror image — if the broker confirms but the UPDATE fails, the next tick republishes and the consumer deduplicates — and you'll have the pattern's full contract covered.

Beyond automated tests, spend an afternoon **rehearsing failures in staging**: kill the relay mid-batch (kill -9, not a clean shutdown), take Kafka down for ten minutes under active write traffic, leave the Debezium connector stopped for an hour and watch the retained WAL. Each of those drills tends to uncover a missing adjustment — a timeout, an alert, an index — and it's infinitely cheaper to find it in staging than in production.

Event versioning: the contract evolves too

An event's payload is a public contract: the moment the first external consumer processes it, you can no longer change it casually. And you will change it, because the business changes. Long-lived outboxes end up suffering more from schema evolution than from throughput, so it pays to set the rules on day one:

- **Version from the start.** Add `event_version` (an integer) to the outbox row and the message. Today's explicit version 1 saves tomorrow's archaeology of "what format are these 2024 events in?"

- **Additive changes are safe; the rest are not.** Adding an optional field breaks nobody who ignores unknown fields (make your consumers ignore them: that's the cheap half of compatibility). Renaming, removing or changing the type of an existing field breaks consumers and requires a version bump.
- **For breaking changes, publish both versions during the transition.** The producer writes v1 and v2 (two outbox rows, or one field carrying both shapes) until the last consumer migrates. It's tedious and it's also the only orderly way; the alternative — coordinating a big-bang deploy — always fails.
- **Consider a schema registry once more than two teams consume.** Confluent Schema Registry or Apicurio validate compatibility at publish time, turning "we broke a consumer" from an incident into a CI failure. With JSON at small scale, a directory of versioned schemas in the repository plus validation in tests plays the same role.

A technique that scales well on the consumer side is *upcasting*: the consumer converts any old version to the current one on receipt, and the rest of its code only knows the latest shape. The compatibility logic stays concentrated in one pure function, trivial to test:

```
def upcast(event: dict) -> dict:
    v = event.get("event_version", 1)
    if v == 1:
        # v2 split "name" into "first_name" / "last_name"
        first, _, last = event["payload"].pop("name", "").partition(" ")
        event["payload"]["first_name"] = first
        event["payload"]["last_name"] = last
        event["event_version"] = 2
    return event
```

Alternatives: when NOT to use an outbox

The pattern is excellent, not universal. Signs there's a simpler or better-suited option:

- **The "event" is only consumed by you, in the same process and the same database.** Then you don't need a broker: a jobs table with `FOR UPDATE SKIP LOCKED` and a worker is the same mechanism without Kafka in the middle. The outbox shines when *other* systems are listening.
- **You need the answer now.** If the caller requires the result synchronously (a validation, a price calculation), that's a REST/gRPC call with a timeout and retries, not an event. Forcing request-response over messaging adds latency and complexity for nothing.
- **You already run CDC on the business tables.** Capturing `orders` directly with Debezium avoids the outbox table, but couples your public contract to your internal schema: every column migration leaks to consumers. The outbox is precisely the decoupling layer; skip it only if you control every consumer and accept that coupling knowingly.
- **You're tempted to use LISTEN/NOTIFY as transport.** PostgreSQL's native pub/sub is useful as the relay's *alarm clock* (less empty polling: NOTIFY in the insert trigger, the relay sleeps in LISTEN), but not as reliable transport: notifications aren't persisted and anyone not listening at that moment loses them. The table remains the source of truth.
- **Your platform already gives you the equivalent primitive.** DynamoDB Streams, Cosmos DB change feed or EventBridge Pipes offer "committed changes as a stream" as a managed

service. If you live in one of those ecosystems, a hand-rolled outbox may be reinventing the wheel.

Production checklist

Before calling an outbox implementation done, review:

- **Atomicity:** event and business data in the same transaction, verified with a rollback test.
- **Relay:** publishes in batches, waits for acks before marking, `SKIP LOCKED` under concurrency, stable per-aggregate partitioning if ordering matters with multiple relays.
- **Ordering:** partition key on `aggregate_id` end to end; no id watermarks without a snapshot barrier.
- **Consumers:** idempotent, ideally with a transactional inbox; offset committed after the local commit.
- **Debezium (if CDC):** correct `snapshot.mode`, heartbeats, `max_slot_wal_keep_size`, retained-WAL alert.
- **Retention:** batched cleanup or partitions with automatic drop; the outbox is not an event store.
- **Observability:** pending count, oldest age and throughput exposed; alert on age, not just volume.
- **Contract:** `event_version` on every event, consumers that ignore unknown fields, a plan for breaking changes.

Case study: a complete pipeline running locally

To make all the pieces concrete, let's assemble the whole system locally: an order service in FastAPI, PostgreSQL with the outbox, and the polling relay publishing to Kafka. It's the same skeleton that works in production, in miniature. The infrastructure, with Docker Compose:

```
# docker-compose.yml
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_DB: orders
      POSTGRES_PASSWORD: dev
    ports: ["5432:5432"]

  kafka:
    image: apache/kafka:4.0.0   # KRaft mode: no ZooKeeper needed anymore
    ports: ["9092:9092"]

  relay:
    build: .
    command: python relay.py
    environment:
      DATABASE_URL: postgresql://postgres:dev@db:5432/orders
      KAFKA_BOOTSTRAP: kafka:9092
```

```
    depends_on: [db, kafka]
    restart: unless-stopped   # the relay MUST restart on its own
```

The service exposes a single endpoint that writes order and event in the same transaction. Notice the endpoint knows nothing about Kafka: its only dependency is the database, which it already had. That's the whole point of the pattern.

```
# app.py
from fastapi import FastAPI
import json, uuid, psycopg_pool

app = FastAPI()
pool = psycopg_pool.ConnectionPool(DATABASE_URL)

@app.post("/orders", status_code=201)
def create_order(order: OrderIn):
    order_id = str(uuid.uuid4())
    with pool.connection() as conn, conn.transaction():
        conn.execute(
            "INSERT INTO orders (id, customer, total) VALUES (%s, %s, %s)",
            (order_id, order.customer, order.total),
        )
        conn.execute(
            """
            INSERT INTO outbox (aggregate_type, aggregate_id, event_type, payload)
            VALUES ('order', %s, 'order.created', %s)
            """,
            (order_id, json.dumps({
                "order_id": order_id,
                "customer": order.customer,
                "total": str(order.total),
                "event_version": 1,
            })),
        )
    return {"id": order_id}
```

And the relay is the `relay_tick()` loop from the scaling section, wrapped in a process with adaptive waiting: if the last batch came back full, repeat immediately; if it came back empty, sleep a short interval. That detail keeps latency low at peak hours without hammering the database at 4 a.m.:

```
# relay.py
import time

IDLE_SLEEP = 0.5   # seconds when there is no work

def main() -> None:
    conn = connect(DATABASE_URL)
    while True:
        try:
            n = relay_tick(conn)
            if n < BATCH:          # incomplete batch: no pressure
                time.sleep(IDLE_SLEEP)
        except Exception:
            log.exception("tick failed; retrying")
            time.sleep(2)          # simple backoff: next tick retries
            conn = reconnect(conn)
```

```
if __name__ == "__main__":
    main()
```

With the three containers up, the acid test is a `curl` to `/orders` with Kafka switched off: the order is still created (201), the event sits pending in the outbox, and when Kafka comes back the relay publishes it on its own, a few seconds later. Watching that sequence with your own eyes is the best way to internalize what the pattern is buying you.

ORM integration: the right hook

The raw-SQL examples make it obvious where the transaction is, but most real services use an ORM, and the question becomes "where do I put the outbox INSERT so it lands in the same transaction?". The generic answer: in the same *unit of work* as the business entity, never in a hook that runs after the commit.

With SQLAlchemy, the natural approach is accumulating events on the session and flushing them to the outbox just before the flush:

```
from sqlalchemy import event
from sqlalchemy.orm import Session

def collect(session: Session, evt: dict) -> None:
    session.info.setdefault("pending_events", []).append(evt)

@event.listens_for(Session, "before_flush")
def flush_outbox(session, ctx, _):
    for evt in session.info.pop("pending_events", []):
        session.add(Outbox(**evt))   # joins the SAME transaction

# In business code:
order = Order(customer="ada", total=99)
db.add(order)
collect(db, order_created_event(order))
db.commit()   # order + event, atomic
```

In Django, the equivalent is writing the `Outbox` model inside the use case's `transaction.atomic()` block. The anti-pattern to avoid is the same in both frameworks: `after_commit` / `post_save` style signals that publish directly to the broker. They look convenient and they reintroduce exactly the dual write the outbox eliminates — if the process dies after the commit and before the signal, the event never existed.

Large payloads: the claim check pattern

The `payload JSONB` column invites you to store huge documents, and PostgreSQL will allow it without complaint: it compresses and chunks large values into TOAST storage transparently. But a high-traffic outbox with payloads of hundreds of kilobytes multiplies WAL (and with CDC, all that WAL travels through the replication slot), fattens Kafka messages — whose default limit is around 1 MB — and makes cleanup more expensive.

The classic solution is the *claim check*: the event carries a reference, not the content. Upload the document to an object store (S3, GCS) *before* the transaction, and keep in the payload only the URL and the metadata consumers need to decide whether they care:

```
{
  "event_type": "invoice.generated",
  "invoice_id": "inv_8843",
  "customer_id": "cus_112",
  "total": "1240.00",
  "document_ref": "s3://invoices/2026/07/inv_8843.pdf",
  "event_version": 1
}
```

The practical rule: if the payload routinely exceeds a few tens of kilobytes, reference instead of embedding. And be disciplined about write order: first the object in S3, then the transaction with the event. The other way round, a fast consumer can receive the event before the document exists.

Outbox, sagas and event sourcing: how they fit together

Three terms that show up together in any search about events, and are worth not confusing:

- **The outbox is a reliable publishing mechanism.** It doesn't define which events you publish or how consumers react; it only guarantees that what was committed gets published.
- **A saga is business coordination.** A multi-step process across services — reserve stock, charge, schedule shipping — where each step emits events and failures trigger compensations. Sagas *need* reliable messaging to avoid stalling halfway, and the outbox is the standard way to provide it: each step writes its state and its outgoing event in the same local transaction.
- **Event sourcing is a persistence model.** The state *is* the sequence of events; there's no orders table to update — you rebuild by replaying the log. If you do full event sourcing, a separate outbox table is redundant: the event store is already the source of truth and publishing means reading from it. The outbox is, in a sense, the minimum dose of event sourcing you can adopt without changing your data model.

For most teams, the healthy progression is exactly that: classic relational model, outbox for publishing, sagas when multi-service processes appear, and event sourcing only if the domain truly demands it (full audit, temporal reconstruction, financial domains). Jumping straight to the end is the usual recipe for a system nobody knows how to operate.

Frequently asked questions

Doesn't the outbox double my writes? Yes: every business operation writes one extra row. In practice the cost is marginal — a small INSERT in a transaction that already existed — and vanishes into the noise compared with the cost of the business tables' indexes. If your database is short on IOPS, the outbox won't be what breaks it.

Can I use the outbox with MySQL, SQL Server or Oracle? The pattern is database-agnostic: anything with ACID transactions works. What changes are the relay details (`SKIP LOCKED` exists in MySQL 8 and Oracle; SQL Server uses `READPAST` hints) and the corresponding Debezium connector if you go with CDC.

What if the outbox insert itself fails? Then the whole transaction fails and the caller gets an error: exactly the right behavior. The outbox can never be "less available" than your database, because it *is* your database. Compare that with the dual write, where a downed broker forces you to choose between rejecting the operation and losing the event.

Does every service need its own outbox? Yes, one per database. The outbox lives next to the data whose consistency it protects; sharing a central outbox across services would reintroduce the coupling (and the distributed transactions) you were trying to avoid.

Should I delete outbox rows after publishing, or mark them? Marking with `processed_at` and cleaning up later is the more operable default: it keeps a short forensic window (invaluable when a consumer claims something never arrived), makes the relay's UPDATE cheap, and delegates deletion to a job you control. Deleting immediately keeps the table tiny but throws away that debugging window and turns every publish into a write-heavy operation. If you use Debezium, there's a third option: DELETE right inside the relay transaction and let CDC read the INSERT from the WAL anyway — the table stays nearly empty and the WAL is your record. All three are valid; what matters is choosing deliberately rather than by accident.

Polling or CDC: a decision guide

Both relay variants appeared earlier in the article; after operating both, these are the criteria that actually tip the scales:

- **Start with polling if in doubt.** It's a hundred lines of code you understand top to bottom, it's debugged with a SELECT, and it adds no infrastructure. For target latencies measured in seconds and volumes up to dozens of events per second, anything more complex is hard to justify.
- **CDC wins when polling starts to hurt.** Sub-second target latency, sustained high volume (aggressive polling loads the database), or multiple outboxes across services where a platform team can amortize the cost of operating Kafka Connect for everyone.
- **CDC demands operational maturity.** Kafka Connect is another distributed system: versions, rebalances, replication slots to watch, and debugging a connector is more opaque than debugging your own loop. If your team doesn't operate Kafka comfortably yet, polling is the honest choice.
- **The business code doesn't change.** The outbox table and the transactional write are identical either way, so migrating from polling to CDC later is an infrastructure change, not an application change. Starting simple doesn't lock you in.

Replays and backfills: when a consumer shows up late

Sooner or later the request arrives: "we're a new service and we need the historical events", or worse, "the consumer had a bug for three days and processed garbage; we need to reprocess." It pays to have an answer ready, because improvising one while production burns goes badly.

First, a warning: **the outbox is not an event store**. With the days-long retention we recommend, it can't serve historical replays. The real options:

- **Long retention in the broker.** Kafka can retain topics for weeks or months (or indefinitely with key compaction). A new consumer simply starts from the earliest offset and catches up. It's the cleanest solution if volume allows: the broker is already the archive.
- **Rebuild from state.** To populate a new service, you often don't need the historical events but the current state: a one-off job that walks the `orders` table and emits synthetic `order.snapshot` events into the topic (or directly into the outbox, in controlled batches). Mark these as snapshot events so consumers can tell them apart from business events.
- **Consumer-side reprocessing.** If the bug was in the consumer, don't touch the producer: rewind the consumer group's offset (`kafka-consumer-groups --reset-offsets`) to a point before the bug. Here the investment in idempotency pays dividends: reprocessing three days of events through an idempotent consumer is boring, which is exactly what you want.

Whichever route you take, do it *at a controlled pace*. A full-speed mass replay can drown the healthy consumers sharing the topic; serious replays are done with throttling while watching the lag of every affected group.

Multi-tenancy and sensitive data in the outbox

Two considerations that usually arrive late in the design and are cheap to include from the start:

Tenant as a first-class citizen. In a multi-tenant SaaS, add `tenant_id` as a proper outbox column (not just inside the payload). It serves three purposes: routing to per-tenant topics if a large customer demands isolation, applying different retention or quotas per plan, and — the important one — being able to answer "delete all of customer X's data" in the outbox and the topics too, not just in the business tables.

The outbox is personal data too. Payloads copy business data: emails, addresses, amounts. That means the access, encryption and retention policies you apply to `orders` apply equally to `outbox` and to the topics it feeds. Teams that encrypt sensitive columns in business tables and then publish those same columns in cleartext to a topic with infinite retention have a compliance problem waiting to be discovered. Practical rules: publish the minimum necessary (consumers can fetch the rest via API under their own authorization), take broker retention as seriously as database retention, and if the domain handles especially sensitive data, consider per-tenant encrypted payloads — deleting the key amounts to deleting the data across every topic at once, so-called *crypto-shredding*.

Performance: what it really costs

Let's close with ballpark numbers, because architecture decisions are made better with orders of magnitude in mind than with vague fears:

- **The extra outbox INSERT** typically adds tenths of a millisecond to a transaction that was already writing: it's a small row in a table with a single partial index. In an endpoint that takes 20–50 ms, the overhead disappears into network noise.
- **A polling relay with batches of 500** and a half-second idle interval sustains thousands of events per second on modest hardware; the bottleneck is usually the `flush()` toward Kafka, not

PostgreSQL. If lag grows while the relay's CPU sits bored, raise the batch size before adding processes.

- **The partial index is the difference between O(pending) and O(table).** With it, the relay's SELECT touches only unprocessed rows even when the table holds millions; without it, every tick scans dead history. If the relay's `EXPLAIN` shows a Seq Scan, check that the index condition and the WHERE clause match exactly.
- **End-to-end latency** (commit → consumer) is dominated by the polling interval in the simple variant (high percentile ≈ interval + network) and drops to tens of milliseconds with CDC. Put a target number in front of the team before choosing: "confirmation emails can take 5 seconds" and "fraud needs to know within 100 ms" lead to different architectures, and both are fine.

And the advice that cuts across everything: **measure your case before optimizing someone else's**. Half the complexity added to production outboxes answers volumes the system will never reach.

Three real failures and their lessons

To close the new material, three failure scenarios that repeat across different teams with suspicious regularity. All three are avoidable with what you've already read; seeing them narrated helps you recognize them in time.

The zombie relay. A team deploys the relay as a sidecar of the API. A rollout leaves two versions of the API running for a while and, with them, two relays competing without `SKIP LOCKED`: every event gets published twice for twenty minutes. Nobody notices until billing asks why there are duplicate refunds. Lessons: the relay is a component with its own lifecycle, and consumer idempotency is not optional even "for now" — the day you need it, it's too late to add.

The disk that filled itself. Another team adopts CDC, it works beautifully, and they forget about it. Months later they migrate Kafka Connect and the connector is left stopped in a secondary environment nobody watches. The replication slot retains WAL for nine days until it fills the database disk... in production, because the "secondary" environment shared the instance. Lessons: every slot needs a retained-WAL alert and a `max_slot_wal_keep_size`, and forgotten connectors are disk leaks on a timer.

The cleanup that knocked over the replica. A third team lets the outbox grow for two years to four hundred million rows and decides to fix it one Friday with `DELETE FROM outbox WHERE processed_at IS NOT NULL`. The statement generates so much WAL that the read replica falls forty minutes behind and automatic failover panics. Lessons: retention is designed on day one, when it's cheap; and any mass operation on a hot table is done in batches, checking replica lag between batches.

The common thread across all three: the outbox worked exactly as promised — no event was lost. What failed was always the operations around it. That's the nature of this pattern: it's simple to implement and demands discipline to operate, not the other way around.

Conclusion

The transactional outbox turns "save and publish" into a single atomic write plus a simple, retryable relay. It's one of the best cost/benefit patterns in event-driven architectures: one table, one worker,

and one golden rule (consumer idempotency) eliminate an entire class of consistency bugs that otherwise only show up in production. If you already apply the retry-with-backoff and API idempotency patterns from this series, the outbox is the missing piece on the producer side.