PuigFlows

El Patrón Saga: Transacciones Distribuidas entre Microservicios sin 2PC

Guía | Premium | Intermedio | Proyectos Reales

Lectura: ~37 min · Julio 2026

> Aprende a mantener la consistencia de datos entre microservicios sin two-phase commit usando el patrón saga: coreografía frente a orquestación, transacciones de compensación bien diseñadas, transacciones pivote, el problema del aislamiento y sus contramedidas, y cómo combinar sagas con el transactional outbox para no perder eventos.

El problema: una operación de negocio repartida en varios servicios

En un monolito, crear un pedido, cobrar al cliente y reservar stock cabe en una sola transacción ACID: o se hace todo, o no se hace nada. Al dividir el sistema en microservicios esa garantía desaparece, porque cada servicio tiene su propia base de datos. El pedido vive en un servicio, el pago en otro y el inventario en un tercero. Si el cobro falla después de haber creado el pedido, nadie hace rollback por ti.

La respuesta clásica, two-phase commit (2PC), coordina un commit atómico entre varias bases de datos, pero encaja mal con los microservicios: el coordinador es un punto único de fallo, los locks se mantienen durante toda la conversación (hundiendo el throughput) y muchas piezas modernas —Kafka, la mayoría de bases NoSQL, cualquier API de terceros— sencillamente no lo soportan. Necesitas otra estrategia, y esa estrategia es la saga.

Qué es una saga

Una saga descompone la transacción distribuida en una secuencia de transacciones locales. Cada paso hace commit en la base de datos de su propio servicio y a continuación publica un evento o notifica a un coordinador; el siguiente paso arranca a partir de ahí. Si un paso falla, la saga ejecuta transacciones de compensación que deshacen los pasos ya confirmados, en orden inverso.

La palabra clave es «deshacen semánticamente». Una compensación no es un ROLLBACK: los pasos anteriores ya hicieron commit y otros procesos pueden haber leído esos datos. Compensar significa ejecutar una transacción nueva que revierte el efecto de negocio: si cobraste, reembolsas; si reservaste stock, lo liberas; si creaste el pedido, lo marcas como cancelado (no lo borras).

Dentro de una saga conviene distinguir tres tipos de paso:

- **Transacciones compensables:** pasos que pueden deshacerse si algo posterior falla (reservar stock, crear el pedido en estado pendiente).
- **Transacción pivote:** el punto de no retorno. Una vez confirmada (por ejemplo, el cobro), la saga ya no retrocede: debe completarse hacia delante.
- **Transacciones reintentables:** pasos posteriores al pivote que no pueden fallar de forma permanente; se reintentan con backoff hasta que terminan (enviar el email de confirmación, actualizar el estado final).

Coreografía: la saga como cadena de eventos

En una saga coreografiada no hay coordinador. Cada servicio escucha los eventos que le interesan y reacciona publicando los suyos. Para un flujo de pedido: el servicio de pedidos publica order.created, el de inventario reserva stock y publica stock.reserved, el de pagos cobra y publica payment.completed o payment.failed, y el de pedidos escucha el resultado para confirmar o cancelar.

```
# Servicio de pagos: reacciona al evento del servicio de inventario
@consumer.on("stock.reserved")
def handle_stock_reserved(event):
    order = event["payload"]
    try:
        charge = payment_gateway.charge(
            customer=order["customer_id"],
            amount_cents=order["total_cents"],
            # Clave de idempotencia: una reentrega del broker no cobra dos veces
            idempotency_key=f"order-{order['order_id']}-charge",
        )
        publish("payment.completed", {"order_id": order["order_id"], "charge_id": charge.id})
    except CardDeclinedError as e:
        # No lances: publica el fallo para disparar las compensaciones
        publish("payment.failed", {"order_id": order["order_id"], "reason": str(e)})
```

Y en el servicio de inventario, la compensación es simplemente otro handler:

```
# Servicio de inventario: compensa cuando el pago falla
@consumer.on("payment.failed")
def handle_payment_failed(event):
    release_reservation(order_id=event["payload"]["order_id"])  # idempotente
    publish("stock.released", {"order_id": event["payload"]["order_id"]})
```

La coreografía brilla por su bajo acoplamiento: no hay un servicio central que conozca a todos los demás, y añadir un consumidor nuevo no toca el flujo existente. Su gran defecto aparece con el tamaño: el flujo completo no está escrito en ningún sitio. Para saber qué hace la saga hay que leer los handlers de cinco servicios, y una dependencia cíclica entre eventos puede pasar desapercibida hasta que llega a producción.

Orquestación: un coordinador explícito

En una saga orquestada, una pieza central define los pasos, invoca a cada servicio y decide qué compensar cuando algo falla. El flujo completo se lee en un solo archivo:

```
from dataclasses import dataclass
from typing import Awaitable, Callable, Optional

@dataclass
class SagaStep:
    name: str
    action: Callable[[dict], Awaitable[dict]]                # transacción local
    compensation: Optional[Callable[[dict], Awaitable[None]]] # cómo deshacerla

class CreateOrderSaga:
    def __init__(self, orders, inventory, payments, repo):
        self.repo = repo   # estado durable de la saga
        self.steps = [
            SagaStep("create_order", orders.create_pending, orders.cancel),
            SagaStep("reserve_stock", inventory.reserve, inventory.release),
            SagaStep("charge", payments.charge, None),        # pivote
            SagaStep("confirm_order", orders.confirm, None),  # reintentable
        ]

    async def execute(self, ctx: dict) -> dict:
        saga_id = await self.repo.start("create_order", ctx)
        completed: list[SagaStep] = []
        for step in self.steps:
            try:
                ctx = await step.action(ctx)
                completed.append(step)
                # El estado durable permite reanudar tras un reinicio
                await self.repo.mark_step_done(saga_id, step.name, ctx)
            except Exception:
                await self._compensate(saga_id, completed, ctx)
                raise
        await self.repo.mark_completed(saga_id)
        return ctx

    async def _compensate(self, saga_id, completed, ctx):
        for step in reversed(completed):
            if step.compensation:
                # Las compensaciones se reintentan: no pueden fallar sin plan B
                await retry_with_backoff(step.compensation, ctx, max_attempts=10)
            await self.repo.mark_compensated(saga_id, step.name)
```

Fíjate en self.repo: el estado de la saga debe sobrevivir a un reinicio del orquestador. Si el proceso muere entre el cobro y la confirmación, al arrancar tiene que saber exactamente dónde se quedó y continuar (o compensar). Escribir esa maquinaria a mano es viable, pero en producción la mayoría de los equipos delega en un motor de workflows durable como Temporal o AWS Step Functions, que persiste cada paso y reanuda la ejecución automáticamente. La tendencia en 2026 es clara: en cuanto una saga supera los tres o cuatro pasos, la orquestación gana por visibilidad, depuración y control.

El problema del aislamiento

Las sagas sacrifican la I de ACID. Entre paso y paso, los cambios intermedios ya confirmados son visibles para el resto del sistema, lo que abre la puerta a anomalías: otro proceso puede leer un

pedido que después se cancela (lectura sucia a nivel de negocio) o dos sagas concurrentes pueden pisarse una actualización. Las contramedidas clásicas:

- **Bloqueo semántico:** el registro lleva un estado que avisa de que hay una saga en curso. Un pedido en PENDING_PAYMENT no se muestra como confirmado ni puede modificarse por otros flujos.
- **Actualizaciones conmutativas:** diseña operaciones cuyo orden no importe (sumar y restar saldo) para que dos sagas concurrentes no se corrompan entre sí.
- **Vista pesimista:** reordena los pasos para que lo más arriesgado ocurra lo más tarde posible; así minimizas la ventana en la que una compensación deja datos inconsistentes a la vista.
- **Releer antes de escribir:** antes de un paso crítico, verifica que el dato no cambió desde que la saga lo leyó; si cambió, aborta o recalcula.

```
-- Bloqueo semántico: el estado marca el registro como "en vuelo"
UPDATE orders SET status = 'PENDING_PAYMENT'
WHERE id = :order_id AND status = 'CREATED';
-- Todos los demás flujos saben que un pedido PENDING_* puede revertirse
```

Sagas y el transactional outbox

Hay un fallo que rompe cualquier saga basada en eventos: el servicio confirma su transacción local pero muere antes de publicar el evento. Resultado: la saga se queda colgada para siempre. La solución es el patrón transactional outbox, que ya cubrimos en detalle en otra guía de esta serie: el evento se escribe en una tabla outbox dentro de la misma transacción local, y un relay lo publica después. Así cada paso de la saga —cambio de estado más evento— es atómico de verdad. Sagas y outbox no compiten: se complementan, y en producción casi siempre viajan juntos.

Construye tu propio orquestador: un modelo de datos que sobrevive a los reinicios

El orquestador esbozado más arriba esconde su requisito más duro en una línea de aspecto inocente: `self.repo`. El estado durable de la saga es lo que separa una demo de un sistema de producción, así que merece la pena hacer ese repositorio concreto. El esquema mínimo necesita dos tablas: una para la instancia de la saga y otra para su registro de pasos.

```
CREATE TABLE saga_instances (
  id            UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  saga_type     TEXT        NOT NULL,          -- 'create_order'
  status        TEXT        NOT NULL,          -- RUNNING | COMPENSATING | COMPLETED | FAILED
  context       JSONB       NOT NULL,          -- salidas acumuladas de los pasos
  current_step  TEXT,
  created_at    TIMESTAMPTZ NOT NULL DEFAULT now(),
  updated_at    TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE saga_steps (
  saga_id     UUID  NOT NULL REFERENCES saga_instances(id),
  step_name   TEXT  NOT NULL,
  status      TEXT  NOT NULL,        -- DONE | COMPENSATED | FAILED
  output      JSONB,
```

```
  finished_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  PRIMARY KEY (saga_id, step_name)
);

-- El reaper y los dashboards viven sobre este índice:
CREATE INDEX idx_sagas_running
  ON saga_instances (updated_at)
  WHERE status IN ('RUNNING', 'COMPENSATING');
```

Aquí importan tres decisiones de diseño. Primera: `context` es una instantánea JSONB de todo lo que los pasos han producido hasta el momento (el id del pedido, el id del cargo, el id de la reserva); un paso que haya que compensar meses después debe encontrar cada identificador que necesite dentro de esa columna, nunca en memoria. Segunda: el registro de pasos es un apéndice por paso con clave primaria compuesta, lo que hace que marcar un paso como completado sea idempotente por naturaleza: al reanudar una saga que murió tras confirmar un paso pero antes de registrarlo, basta un `INSERT ... ON CONFLICT DO NOTHING` para seguir adelante. Tercera: el índice parcial hace que la pregunta que todo operador se hace — "¿qué hay en vuelo y desde hace cuánto?" — nunca recorra el histórico completado.

La recuperación al arrancar es entonces un bucle, no un arte:

```
async def resume_pending_sagas(repo, registry):
    """Se llama una vez al arrancar el servicio, antes de aceptar trabajo nuevo."""
    for instance in await repo.find_unfinished():        # usa idx_sagas_running
        saga = registry.build(instance.saga_type)
        done = await repo.step_names_done(instance.id)
        if instance.status == "COMPENSATING":
            # Murió a mitad de deshacer: sigue compensando donde se quedó
            await saga.compensate_remaining(instance, done)
        elif saga.passed_pivot(done):
            # Pasado el punto de no retorno: la única dirección es hacia delante
            await saga.run_remaining(instance, done)
        else:
            # Antes del pivote, lo seguro por defecto es deshacer
            await saga.compensate_remaining(instance, done)
```

Fíjate en la asimetría alrededor del pivote: una saga interrumpida que ya cobró al cliente se reanuda hacia delante, mientras que una que solo reservó stock se compensa. Codificar esa decisión en la rutina de recuperación — en lugar de dejársela a quien esté de guardia a las 3 de la madrugada — es exactamente la razón por la que identificaste la transacción pivote desde el principio.

Timeouts y el reaper: ninguna saga espera para siempre

Cada paso que espera a otro servicio necesita dos temporizadores, no uno. El timeout de la petición (segundos) acota una llamada individual; el timeout de la saga (minutos u horas, según el flujo) acota la operación de negocio completa. El segundo es fácil de olvidar porque nada explota cuando falta: los pedidos simplemente se acumulan en silencio en `PENDING_PAYMENT` hasta que un cliente escribe preguntando.

La contramedida estándar es un *reaper*: un pequeño trabajo periódico que encuentra sagas cuyo `updated_at` supera el plazo por tipo y las empuja a un estado terminal.

```
REAPER_DEADLINES = {"create_order": timedelta(minutes=30)}
```

```
async def reap_stuck_sagas(repo):
    for saga_type, deadline in REAPER_DEADLINES.items():
        stuck = await repo.find_older_than(saga_type, deadline)
        for instance in stuck:
            if instance.passed_pivot():
                # El dinero ya se movió: escala a un humano, nunca canceles en automático
                await alerts.page(
                    "saga-stuck-past-pivot",
                    saga_id=instance.id, minutes=instance.age_minutes(),
                )
            else:
                await repo.mark(instance.id, "COMPENSATING")
                await queue.enqueue("compensate_saga", saga_id=instance.id)
```

La rama sobre el pivote vuelve a ser la parte interesante. Antes del pivote, expirar hacia la compensación es seguro y automático. Después, una cancelación automática reembolsaría a un cliente que quizá ya recibió su email de confirmación, así que el movimiento correcto es un aviso, no un cambio de estado. Una regla útil: **el reaper puede compensar, pero solo un humano puede abandonar una saga pasado su pivote.**

Dos detalles operativos ahorran incidentes reales. Haz que el intervalo del reaper sea mucho más corto que el plazo más corto (un plazo de 30 minutos comprobado cada hora es, en la práctica, un plazo de 90 minutos). Y emite una métrica por saga segada: un pico repentino de sagas `create_order` segadas suele ser el primer síntoma visible de una dependencia caída aguas abajo.

Idempotencia en la práctica: la tabla inbox

El artículo ha repetido "haz cada paso idempotente" como un mantra; esto es lo que significa mecánicamente. Los brokers entregan al menos una vez, así que cada consumidor de una saga coreografiada — y cada handler de compensación de una orquestada — acabará viendo el mismo mensaje dos veces. La defensa de propósito general es el *inbox transaccional*: registrar el id del mensaje en la misma transacción local que aplica su efecto.

```
CREATE TABLE processed_messages (
  consumer   TEXT NOT NULL,   -- 'inventory-service'
  message_id TEXT NOT NULL,   -- del sobre del evento
  processed_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  PRIMARY KEY (consumer, message_id)
);
```

```
async def handle_stock_reserved(event, db):
    async with db.transaction():
        inserted = await db.execute(
            """INSERT INTO processed_messages (consumer, message_id)
               VALUES ('payment-service', :mid)
               ON CONFLICT DO NOTHING""",
            mid=event["message_id"],
        )
        if inserted == 0:
            return                      # entrega duplicada: ack y seguimos
        await apply_payment_effects(event, db)   # misma transacción
```

Como el insert de deduplicación y el efecto de negocio se confirman atómicamente, un crash entre ambos es imposible por construcción: el fallo clásico de comprobar en una caché "¿he visto esto ya?" antes de hacer el trabajo, y morir antes de registrarlo, sencillamente no puede ocurrir. Para pasos que llaman a proveedores externos, combina el inbox con el mecanismo de

idempotencia del propio proveedor (como la `Idempotency-Key` que ya usa el ejemplo del cargo): el inbox protege tu base de datos, la clave protege al tercero.

Una sutileza: la tabla inbox crece para siempre si no la podas. Una ventana de retención algo mayor que el horizonte máximo de reentrega de tu broker (una semana es una elección habitual) la mantiene pequeña; borrar filas más antiguas es seguro porque el broker ya no puede reentregar esos mensajes.

Sagas sobre un motor de workflows durable: Temporal

Todo lo construido a mano hasta aquí — las tablas de estado, el bucle de recuperación, el reaper — es exactamente lo que un motor de ejecución durable te regala. En Temporal, el propio estado de ejecución del workflow *es* el estado de la saga: el motor registra cada paso en un historial de eventos y, si el proceso worker muere, otro worker reproduce el historial y continúa desde la última actividad completada como si nada hubiera pasado. Sin tabla `saga_instances`, sin reaper, sin recuperación escrita a mano.

El patrón saga en Temporal suele expresarse como una lista creciente de compensaciones y un único `try/except`:

```
from datetime import timedelta
from temporalio import workflow
from temporalio.common import RetryPolicy

@workflow.defn
class CreateOrderSaga:
    @workflow.run
    async def run(self, order: OrderInput) -> str:
        compensations = []          # crece según se confirman pasos
        opts = dict(
            start_to_close_timeout=timedelta(seconds=30),
            retry_policy=RetryPolicy(maximum_attempts=5),
        )
        try:
            order_id = await workflow.execute_activity(
                create_pending_order, order, **opts)
            compensations.append((cancel_order, order_id))

            resv_id = await workflow.execute_activity(
                reserve_stock, order, **opts)
            compensations.append((release_stock, resv_id))

            # Pivote: después de esto, ya no se compensa
            await workflow.execute_activity(charge_customer, order, **opts)

            await workflow.execute_activity(confirm_order, order_id, **opts)
            return order_id
        except Exception:
            for comp, arg in reversed(compensations):
                await workflow.execute_activity(
                    comp, arg,
                    start_to_close_timeout=timedelta(seconds=30),
                    # Las compensaciones reintentan, en la práctica, para siempre
                    retry_policy=RetryPolicy(maximum_attempts=0),
                )
            raise
```

Cada llamada a `execute_activity` es un paso de la saga: corre en un proceso Python normal, puede hablar con cualquier base de datos o API, y su resultado queda registrado de forma durable antes de que el workflow avance. El bucle `reversed(compensations)` es el mismo deshacer LIFO del orquestador artesanal, pero la lista sobrevive a los crashes porque forma parte del estado reproducible del workflow. Fíjate en las dos políticas de reintento: los pasos hacia delante reintentan un número acotado de veces y luego disparan la ruta de fallo de la saga, mientras que las compensaciones reintentan indefinidamente — una compensación que se rinde es exactamente el error "sin plan B" de la lista de errores comunes.

Los trade-offs, eso sí, son reales. El código del workflow debe ser determinista (nada de I/O directa, números aleatorios ni lecturas de reloj fuera de las actividades), que es un modelo de programación genuinamente distinto; el motor es otra pieza de infraestructura con estado que operar (o una factura de proveedor si usas la versión gestionada); y el testing se desplaza hacia el framework de tests basado en replay de Temporal. Para un equipo con dos sagas simples, ese sobrecoste puede no compensar. Para un equipo a punto de construir su tercer bucle de recuperación artesanal, casi seguro que sí.

Sagas como máquinas de estados: AWS Step Functions

La misma lógica de orquestación puede vivir en una máquina de estados gestionada en lugar de en código. En AWS Step Functions, cada paso de la saga es un estado (normalmente invocando una función Lambda), `Retry` maneja los fallos transitorios con backoff exponencial, y `Catch` enruta los fallos definitivos hacia la cadena de compensación:

```
{
  "StartAt": "ReserveStock",
  "States": {
    "ReserveStock": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:reserve-stock",
      "Retry": [{
        "ErrorEquals": ["States.Timeout", "Lambda.ServiceException"],
        "MaxAttempts": 3, "IntervalSeconds": 2, "BackoffRate": 2.0
      }],
      "Catch": [{
        "ErrorEquals": ["States.ALL"],
        "ResultPath": "$.error",
        "Next": "CancelOrder"
      }],
      "Next": "ChargeCustomer"
    },
    "ChargeCustomer": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:charge-customer",
      "Catch": [{
        "ErrorEquals": ["States.ALL"],
        "ResultPath": "$.error",
        "Next": "ReleaseStock"
      }],
      "Next": "ConfirmOrder"
    },
    "ReleaseStock": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:release-stock",
      "Retry": [{ "ErrorEquals": ["States.ALL"], "MaxAttempts": 10, "BackoffRate": 1.5 }],
      "Next": "CancelOrder"
```

```
    },
    "CancelOrder": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:cancel-order",
      "Retry": [{ "ErrorEquals": ["States.ALL"], "MaxAttempts": 10, "BackoffRate": 1.5 }],
      "End": true
    },
    "ConfirmOrder": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:confirm-order",
      "End": true
    }
  }
}
```

Lee con atención los destinos de cada `Catch` y verás la forma de la saga codificada en el cableado: un fallo en `ReserveStock` solo necesita cancelar el pedido, mientras que un fallo en `ChargeCustomer` debe primero liberar el stock y luego cancelar — el catch de cada estado apunta a la entrada correcta de la cadena de compensación, y las propias compensaciones enlazan hacia delante con los pasos de deshacer restantes. La consola pinta todo esto como un diagrama, que es el argumento de visibilidad de la orquestación hecho literal: el flujo que nadie podía dibujar de memoria en una coreografía de ocho pasos es ahora una imagen que todo el equipo puede señalar.

Los límites son la imagen especular de los de Temporal: el lenguaje de definición es JSON en lugar de un lenguaje de propósito general, así que las bifurcaciones complejas y el remodelado de datos se vuelven incómodos; y el precio por transición en los workflows Standard hace que una saga parlanchina tenga un coste unitario real. Encaja mejor cuando tus pasos ya son funciones Lambda y tu equipo prefiere servicios gestionados antes que operar un motor.

Ramas paralelas: cuando la saga es un DAG, no una lista

Los flujos reales rara vez son puramente secuenciales. Reservar stock y ejecutar una comprobación antifraude no dependen entre sí; forzarlos a una secuencia suma sus latencias. Una saga puede ejecutar pasos independientes en concurrencia — solo cambia lo que significa compensar, porque ahora un fallo en una rama debe deshacer *ramas hermanas que quizá siguen en vuelo*.

```
async def execute_parallel_phase(self, ctx):
    results = await asyncio.gather(
        self.inventory.reserve(ctx),
        self.fraud.check(ctx),
        return_exceptions=True,      # nunca pierdas el desenlace de una hermana
    )
    failures = [r for r in results if isinstance(r, Exception)]
    if failures:
        # Compensa cada rama que SÍ tuvo éxito
        if not isinstance(results[0], Exception):
            await retry_with_backoff(self.inventory.release, ctx)
        raise SagaStepFailed(failures[0])
    ctx.update(results[0]); ctx.update(results[1])
    return ctx
```

Las reglas que mantienen esto manejable: paraleliza solo pasos que sean todos compensables (un pivote dentro de una rama paralela significa que el fallo de una hermana puede dejarte

varado pasado el punto de no retorno con otra rama a medias); haz siempre el gather con `return_exceptions=True` para que la excepción de una rama no pueda dejar huérfano el resultado exitoso — y por tanto compensable — de otra; y trata el bloque paralelo completo como una sola unidad en el registro de pasos, anotando qué ramas terminaron. Si te descubres dibujando un grafo de dependencias en rombo con pivotes en medio, esa es la señal para pasarte a un motor: Temporal expresa esto con un `asyncio.gather` normal dentro de un workflow, y Step Functions tiene un estado `Parallel` nativo con catches por rama.

Sagas, TCC y 2PC: elegir una estrategia de consistencia

La saga no es la única alternativa al 2PC, y conviene saber cuándo encajan mejor sus primos. **Try-Confirm-Cancel (TCC)** traslada la idea de la reserva al propio protocolo: cada participante expone tres operaciones, y el coordinador primero llama a `try` en todos (reservando recursos en estado tentativo), luego a `confirm` en todos si todos los tries triunfaron, o a `cancel` en todos en caso contrario.

La diferencia práctica con una saga: en TCC nada visible para el negocio ocurre hasta la fase de confirmación, así que el problema del aislamiento desaparece en gran parte — un bloqueo tentativo de hotel es invisible para otros clientes de una forma en que una reserva confirmada-y-luego-compensada no lo es. El coste es la intrusividad: cada servicio participante debe implementar el contrato de tres operaciones y gestionar reservas que caducan, y por eso TCC tiende a aparecer en dominios que ya piensan en bloqueos y capturas — pagos (authorize/capture/void es TCC disfrazado), reservas de viajes, ticketing — y rara vez en otro sitio.

Una tabla de decisión rápida, contada con honestidad:

- **2PC/XA**: sigue estando bien dentro de una única frontera de confianza donde cada participante lo soporta de verdad (dos bases de datos relacionales tuyas, un broker con XA) y el coste en latencia del bloqueo síncrono es aceptable. Muere en el momento en que un participante es Kafka, un almacén NoSQL o la API HTTP de un tercero.
- **TCC**: necesidades fuertes de aislamiento, participantes que controlas, un dominio con semántica natural de reserva. Paga el precio de implementar try/confirm/cancel en cada servicio.
- **Saga**: la respuesta de propósito general para flujos entre servicios con participantes heterogéneos. El aislamiento más débil, la aplicabilidad más fuerte; las contramedidas de la sección de aislamiento son el impuesto.
- **Ningún patrón**: la opción más infravalorada. Si dos "servicios" siempre cambian juntos transaccionalmente, puede que sean un solo servicio con dos unidades de despliegue — fusionarlos recompra ACID gratis. Una saga que no necesitabas es deuda operativa pura.

Observabilidad: ver una saga como una sola traza

Una saga dispersa una operación de negocio entre servicios, colas y minutos de reloj, que es exactamente la forma de problema para la que existe el trazado distribuido. El objetivo: pegar un id de pedido en tu UI de trazas y ver cada paso, cada reintento y cada compensación como una única historia conectada.

La mecánica difiere según el estilo. En una saga orquestada, haz del `execute` del orquestador el span raíz y de cada paso un hijo — trivial, porque es un proceso llamando a otros. En una coreografiada no hay proceso raíz, así que propaga el contexto de traza *a través del sobre del evento*: el servicio que inicia el flujo inyecta `traceparent` en los metadatos del evento (el relay del outbox debe copiarlo), y cada consumidor lo extrae y arranca su span como hijo. Sin esto, cada salto arranca una traza nueva y tu "historia única" se rompe en cinco fragmentos conectados solo por grep.

```
from opentelemetry import trace
from opentelemetry.propagate import extract, inject

def publish_with_context(topic, payload):
    headers = {}
    inject(headers)                          # escribe traceparent
    outbox.enqueue(topic, payload, headers=headers)

def consume(event):
    ctx = extract(event.headers)
    with tracer.start_as_current_span(
        "saga.step " + event.topic, context=ctx,
        attributes={
            "saga.id": event.payload["order_id"],
            "saga.step": event.topic,
            "messaging.redelivered": event.redelivered,
        },
    ):
        handle(event)
```

Sobre las trazas, tres métricas cubren la mayoría de las preguntas operativas: un contador de sagas iniciadas/completadas/compensadas por tipo (la *tasa* de compensación es la señal de salud — compensar es normal, una proporción creciente no), un histograma de duración extremo a extremo por tipo (vigila el p99, donde se esconden las sagas atascadas que el reaper aún no segó), y un gauge de sagas en vuelo con más de N minutos, que es la cola del reaper y pertenece al dashboard de guardia. Alerta sobre el gauge y la tasa, no sobre compensaciones individuales: una saga que compensa limpiamente es el sistema funcionando según lo diseñado.

Testing de sagas: mata el proceso en cada paso

La checklist de producción exige tests que maten el proceso en cada paso intermedio; esto es lo que eso significa en concreto. La técnica es la inyección de puntos de crash: la saga bajo test corre contra servicios falsos, uno de los cuales está amañado para lanzar — o para "morir" — en un paso elegido, y el test después reinicia el orquestador y comprueba que el mundo converge.

```
import pytest

class CrashAfter:
    """Envuelve un paso para simular que el proceso muere justo tras el commit."""
    def __init__(self, real_action):
        self.real_action = real_action
    async def __call__(self, ctx):
        await self.real_action(ctx)      # la transacción local confirma...
        raise ProcessKilled()            # ...y el proceso desaparece

@pytest.mark.parametrize("crash_step", [
    "create_order", "reserve_stock", "charge", "confirm_order",
])
```

```
async def test_saga_recovers_from_crash_at_every_step(crash_step, saga_env):
    saga = saga_env.build_saga(crash_wrapper=CrashAfter, at_step=crash_step)
    with pytest.raises(ProcessKilled):
        await saga.execute(saga_env.order_ctx())

    # "Reinicio": un orquestador nuevo sobre el mismo estado durable
    await resume_pending_sagas(saga_env.repo, saga_env.registry)

    final = await saga_env.repo.get_by_context(saga_env.order_ctx())
    if crash_step in ("charge", "confirm_order"):     # pivote o posterior
        assert final.status == "COMPLETED"            # corrió hacia delante
        assert saga_env.orders.state == "CONFIRMED"
    else:
        assert final.status in ("COMPENSATED", "FAILED")
        assert saga_env.inventory.reservations == {}  # nada se filtró
        assert saga_env.payments.charges == []        # nadie fue cobrado
```

La parametrización es la clave: un solo cuerpo de test, ejecutado una vez por paso, comprobando el invariante que de verdad importa — *los crashes antes del pivote no filtran nada; los crashes en el pivote o después completan el flujo*. Compleméntalo con un test de entrega duplicada (alimenta cada evento a su consumidor dos veces y comprueba que los efectos se aplican una — aquí es donde la tabla inbox se gana el sueldo) y, si usas infraestructura real en CI, ejecuta los roles de los fakes contra Postgres y tu broker vía Testcontainers para que los detalles de `ON CONFLICT` y `FOR UPDATE` se ejerciten de verdad. Los equipos en Temporal parten con ventaja: su framework de tests por replay puede inyectar fallos en cualquier frontera de actividad sin escribir a mano la maquinaria de crash.

Versionado: desplegar con sagas en vuelo

Las sagas corren durante minutos o días, así que cada despliegue ocurre con instancias a medio camino. Cambia la definición de la saga sin cuidado — reordena pasos, renombra uno, cambia la firma de una compensación — y el código nuevo despertará sosteniendo estado escrito por el código viejo que ya no entiende.

Las reglas seguras son pocas y mecánicas. Versiona la definición: guarda `saga_type = 'create_order.v2'` (o una columna de versión) en la instancia, mantén registrada la lista de pasos v1 hasta que termine la última instancia v1, y enruta los arranques nuevos a v2. No renombres nunca un nombre de paso ya registrado — el registro de pasos es un contrato con tu propio código de recuperación. Trata las compensaciones como APIs públicas: el despliegue v2 debe seguir pudiendo compensar un paso confirmado por v1, así que las entradas de la compensación viven en el `context` de la saga, no en suposiciones a nivel de código. Y cuando un cambio es de verdad incompatible, gana la estrategia aburrida: *drenar* — despliega v2 junto a v1, deja de arrancar sagas v1, espera a que terminen las que corren, y borra v1. Los motores de workflows hacen esto explícito (Temporal tiene versionado de workers de primera clase exactamente para este problema), pero la estrategia de drenado funciona idéntica para un orquestador artesanal.

Caso de estudio: el flujo de pedido, de extremo a extremo

Para aterrizarlo todo, acompaña a un pedido por un sistema que combina cada patrón de esta guía. El reparto: un servicio de pedidos (con el orquestador embebido), inventario, pagos y un

servicio de notificaciones; Postgres en cada uno; Kafka entre ellos; relays de outbox por todas partes.

El camino feliz. Un cliente envía un pedido. El servicio de pedidos arranca una instancia de saga y, en una transacción local, crea el pedido en `PENDING_STOCK` (lock semántico) y escribe la fila de la instancia. El paso dos llama a inventario, que inserta el id del evento en su inbox, decrementa el stock disponible, registra una reserva y escribe `stock.reserved` en su outbox — una transacción, las cuatro escrituras. El relay publica; el orquestador marca el paso como hecho y avanza el contexto. El paso tres cobra a través del proveedor de pagos con la clave de idempotencia `order-91f4-charge`; este es el pivote. El paso cuatro pasa el pedido a `CONFIRMED` y encola el email de confirmación como paso reintetable. Tiempo total de reloj: unos dos segundos, cuatro transacciones locales, cero locks distribuidos.

El camino interesante. El mismo pedido, pero el proveedor de pagos da timeout — la petición pudo o no haber llegado. La política de reintentos del paso reintenta con la misma clave de idempotencia, que es lo que hace seguro el reintento: si el primer intento sí cobró, el proveedor devuelve el resultado original en lugar de cobrar otra vez. Supón que se agotan todos los reintentos. El pivote nunca confirmó, así que el orquestador pasa a `COMPENSATING` y deshace: libera la reserva (el inbox de inventario se traga el duplicado cuando el mensaje se reentrega), marca el pedido `CANCELLED` — no borrado — y registra la saga como `FAILED` con el error de pago en el contexto. El cliente ve un honesto "el pago no se completó"; los dashboards ven un conteo más en `saga_compensated_total{type="create_order"}`; nadie recibe un aviso, porque una compensación limpia es el sistema funcionando.

El camino de las 3 de la madrugada. Un despliegue reinicia el servicio de pedidos entre el éxito del cobro y el commit de la confirmación. Al arrancar, `resume_pending_sagas` encuentra la instancia: el registro de pasos muestra `charge: DONE`, así que la saga está pasada su pivote y corre hacia delante — confirma, notifica, completa. El cliente nunca se entera. Ese mismo despliegue cinco minutos antes, previo al cobro, habría compensado en su lugar. Ambos desenlaces son correctos, ambos son automáticos, y ambos los eligió código que escribiste a la luz del día en lugar de un ingeniero a las 3 de la madrugada. Esa es la propuesta de valor completa del patrón en una sola frase.

Diseñar las fronteras de los pasos: el oficio detrás del patrón

La mayoría de los bugs de sagas no son errores de código; son pasos dibujados en el lugar equivocado. La frontera de un paso es una promesa: "esta unidad o bien ocurre por completo en un servicio o bien puede deshacerse semánticamente". Tres heurísticas producen buenas fronteras en la práctica.

Un dueño por paso. Un paso debería tocar los datos de exactamente un servicio. En el momento en que un "paso" necesita actualizar dos servicios, es una transacción distribuida anidada con gabardina — divídelo en dos pasos con sus propias compensaciones, o habrás recreado el problema original un nivel más abajo.

Escribe la compensación primero. Antes de implementar un paso, escribe una frase que describa su deshacer: "liberar la reserva", "reembolsar el cargo", "marcar el pedido como cancelado". Si no puedes escribir esa frase — "des-enviar el email", "des-notificar al regulador" — has encontrado un paso que no puede ir antes del pivote. O lo mueves después del pivote (como

paso reintetable) o lo rediseñas a una forma reservable: en lugar de enviar el email, *prográmalo* con un retardo, que es trivialmente cancelable hasta que se dispara.

Empuja el riesgo tarde, empuja las lecturas pronto. Los pasos con probabilidad de fallar por razones de negocio (comprobaciones de crédito, validación de stock) van al principio, donde compensar es barato. Los pasos cuyo fallo es raro y técnico van cerca del pivote. Este orden — la contramedida de la "vista pesimista" aplicada en tiempo de diseño — minimiza tanto el número de compensaciones que llegan a ejecutarse como la ventana de estado intermedio visible.

Un ejemplo trabajado de rediseño de una mala frontera. La versión uno de un flujo de registro tenía un paso "crear cuenta y enviar email de bienvenida" — un paso, dos efectos, uno de ellos incompensable. La versión dos lo divide: el paso uno crea la cuenta en `PENDING` (compensación: marcar `ABORTED`); el pivote activa la cuenta; un paso reintetable después del pivote envía el email. El flujo es ahora plenamente compensable antes del punto de no retorno, y el único efecto incompensable ocurre estrictamente después de él. Ese movimiento de reestructuración — *reserva antes del pivote, efectos secundarios después* — resuelve probablemente la mitad de las objeciones "¡pero este paso no se puede deshacer!" en las revisiones de diseño.

Cuando la compensación falla: dead letters y escalado humano

El mantra "las compensaciones deben acabar triunfando siempre" choca con la realidad: la API de reembolsos lleva seis horas caída, el servicio de inventario rechaza la liberación porque el SKU se retiró a mitad de saga, la compensación ha reintentado cuarenta veces y ya es sábado. Una saga de producción necesita una respuesta diseñada para el día en que el propio deshacer se rompe.

El patrón es una cola de dead letters de compensaciones con escalado humano, y su núcleo es una regla estricta: **una compensación fallida no puede descartarse nunca, solo aparcarse.**

```
async def run_compensation(self, saga_id, step, ctx):
    try:
        await retry_with_backoff(
            step.compensation, ctx,
            max_attempts=10, max_delay=timedelta(minutes=5),
        )
        await self.repo.mark_compensated(saga_id, step.name)
    except RetriesExhausted as e:
        # Aparcarla: registro durable + alerta ruidosa, nunca un log tragado
        await self.repo.park_compensation(
            saga_id=saga_id, step=step.name,
            context=ctx, error=str(e),
        )
        await alerts.page(
            "compensation-parked",
            saga_id=saga_id, step=step.name,
            runbook="https://wiki.internal/runbooks/parked-compensations",
        )
```

La fila aparcada lleva todo lo que un humano necesita para terminar el trabajo a mano o reencolarlo tras la caída: el contexto completo de la saga, el paso que falla, el error. El lado operativo importa tanto como el código: un runbook por tipo de compensación ("cómo liberar una reserva manualmente"), una pequeña acción de administración para reencolar compensaciones aparcadas en bloque cuando el sistema aguas abajo se recupera, y una regla fija: las compensaciones aparcadas con más de un día laborable son incidentes, no backlog. Los equipos

que se saltan esto lo construyen después de su primera caída de seis horas del proveedor de pagos, con peor tooling y más adrenalina.

Sagas de larga duración: días, humanos y plazos

Nada en el patrón dice que una saga termine en segundos. Una solicitud de préstamo, una verificación KYC o un pedido B2B con aprobación manual siguen siendo sagas — transacciones locales, compensaciones, un pivote — salvo que algunos pasos esperan a un humano durante días. A esa escala temporal cambian tres cosas.

Esperar se convierte en un paso. La saga necesita pasos de "esperar señal externa" de primera clase: la aprobación llega como un evento (o un callback de API) que reanuda la instancia. Los orquestadores artesanales lo modelan como un paso en estado `WAITING` que la rutina de reanudación se salta hasta que aterriza la señal; los motores lo traen de serie (signals en Temporal, task tokens y su patrón de callback en Step Functions). El registro de pasos gana disciplina de timestamps: una saga esperando tres días es normal, así que los plazos del reaper pasan a ser por paso en lugar de por saga — treinta minutos para el cobro, cinco días laborables para la aprobación.

Las compensaciones envejecen. Deshacer un paso de hace tres días puede diferir de deshacerlo tres segundos después. Liberar una reserva de stock a los segundos es un `UPDATE`; a los tres días, el periodo de ventas pudo cerrarse, el precio pudo cambiar, el almacén pudo hacer inventario. Las compensaciones de larga vida deberían releer el estado actual y decidir, en lugar de revertir a ciegas — que es la contramedida "relee antes de escribir", ahora obligatoria en vez de opcional.

La gente necesita verlo. Una saga que abarca días es un proceso de negocio, y por los procesos de negocio se pregunta: "¿dónde está atascado el pedido 91f4?" Construye la consulta antes de necesitarla — la tabla `saga_instances` ya la responde con una vista SQL de dos líneas que junta estado, paso actual y antigüedad. Exponer esa vista en solo lectura a los equipos de soporte elimina una categoría entera de interrupciones a ingeniería.

Consultas durante una saga: ¿qué ve el resto del sistema?

La sección de aislamiento cubrió a los escritores chocando; los lectores merecen su propio tratamiento, porque cada pantalla y cada informe de la empresa lee datos tocados por sagas. La pregunta "¿cuál es el estado de este pedido?" tiene una respuesta sutil mientras una saga está en vuelo, y fingir lo contrario produce bugs que ninguna orquestación arregla.

Las reglas prácticas: **estados terminales para las máquinas, estados honestos para los humanos.** Los sistemas aguas abajo que se disparan con el estado del pedido (envíos, facturación, puntos de fidelidad) deben reaccionar solo a estados terminales — `CONFIRMED`, `CANCELLED` — nunca a intermedios. Cablear envíos para dispararse con `PENDING_PAYMENT` "porque casi siempre sale bien" es como un pedido compensado acaba en un camión. Las superficies de cara al humano, en cambio, deben mostrar los estados intermedios con verdad pero con tacto: "procesando tu pago" es honesto; "confirmado" es una mentira que una compensación expondrá.

Las agregaciones necesitan una política, no un accidente: los informes o bien cuentan solo estados terminales (la elección habitual para ingresos) o bien cuentan las sagas en vuelo explícitamente como su propio cubo ("4.200 dólares en pedidos pendientes"). El antipatrón es

contar los intermedios como si fueran terminales y reformular en silencio los números de ayer cuando aterrizan las compensaciones — los equipos de finanzas se dan cuenta, y la confianza perdida sale cara. Si usas read models o un almacén de vistas estilo CQRS, proyecta los estados intermedios con su estado adjunto, y deja que cada consumidor aplique su propia política; quitar el estado al proyectar tira justo la información que los lectores necesitaban.

Auditoría y el rastro de papel

Un efecto secundario agradable de hacer bien las sagas: consigues un log de auditoría gratis. El registro de pasos — cada paso, cada compensación, cada timestamp, cada instantánea del contexto — es precisamente el registro que un auditor, un agente de soporte o una investigación de fraude piden: *¿qué le pasó a este pedido, en qué secuencia, y por qué se reembolsó a este cliente?*

Dos hábitos convierten el registro de pasos de datos de depuración en un rastro con calidad de auditoría. Registra el *porqué*, no solo el *qué*: la fila de la compensación debe llevar el error que la disparó ("pago rechazado: fondos insuficientes"), porque "pedido cancelado" sin causa es una invitación a las conjeturas. Y no mutes nunca filas del registro de pasos — añade las correcciones como filas nuevas. Un registro de pasos solo-apéndice más el historial de eventos del outbox reconstruyen cualquier incidente semanas después, que es la diferencia entre "creemos que el reembolso salió" y una fila con timestamp que lo demuestra. Si tu dominio está regulado, la política de retención de `saga_steps` se convierte en un requisito de cumplimiento y no en una elección de limpieza — decídela deliberadamente.

¿Dónde vive el orquestador?

La orquestación responde a "¿quién coordina?", pero no a "¿dónde corre ese código?" — y la elección de despliegue moldea la propiedad entre equipos más que cualquier propiedad técnica. Hay tres hogares viables.

Embebido en el servicio iniciador. El servicio de pedidos es dueño de la saga de crear pedido: el orquestador es un módulo dentro de él, compartiendo su base de datos para el estado de la saga. Este es el punto de partida por defecto para la mayoría de los equipos. El servicio dueño del resultado de negocio es dueño de la coordinación; no aparece ningún desplegable nuevo; el estado de la saga viaja con la historia de backups y migraciones que el servicio ya tiene. Su límite es la reutilización — cuando tres servicios embeben dos sagas cada uno, los patrones se copian y pegan y divergen.

Un servicio de sagas dedicado. Un desplegable cuyo único trabajo es ejecutar definiciones de sagas. Centraliza la maquinaria (tablas de estado, reaper, tooling de compensaciones aparcadas, dashboards) y da a los operadores un panel único. El riesgo es organizativo, no técnico: un "equipo de workflows" central se convierte con facilidad en un cuello de botella por el que debe pasar cada feature, y el servicio acumula conocimiento de los eventos de todos los dominios — el antipatrón del dios-servicio con mejor marketing. Funciona cuando un equipo de plataforma lo trata como infraestructura de autoservicio, y fracasa cuando se vuelve un guardián.

Un motor. Temporal, Step Functions y compañía son la opción del servicio dedicado comprada en lugar de construida, con la maquinaria de recuperación ya endurecida. La pregunta de la

propiedad no desaparece: las definiciones de workflows siguen necesitando dueños, la disciplina de versionado sigue aplicando, y alguien sigue operando (o pagando) el motor.

El camino de migración entre estas opciones es más suave de lo que parece, siempre que una regla se cumpla desde el primer día: **mantén las definiciones de sagas separadas de la maquinaria de sagas.** Una definición — la lista ordenada de pasos con acciones y compensaciones, como la tabla de `SagaStep` de antes — expresa pura política de negocio. Si nunca importa directamente la capa de persistencia, pasar de embebido a dedicado a motor es un ejercicio de realojo y no una reescritura.

Rendimiento: lo que cuestan las sagas a escala

Una saga multiplica las escrituras. Una operación de negocio que en el monolito era una sola transacción se convierte, en el caso de estudio del pedido: cuatro transacciones locales, cuatro filas de outbox, cuatro publicaciones del relay, cuatro inserts de inbox, más una escritura de estado de saga por transición — más o menos un factor cuatro de amplificación de escritura, antes de los reintentos. A diez pedidos por segundo nadie lo nota. A mil, la propia fontanería de la saga se convierte en una partida de capacidad, y tres técnicas la mantienen asequible.

Agrupa la contabilidad. El relay ya agrupa las publicaciones del outbox; las transiciones de estado de la saga también pueden agruparse. Marcar un paso como hecho no necesita su propio viaje de ida y vuelta — los orquestadores que procesan eventos por lotes (consumiendo una partición de golpe) pueden confirmar un lote de transiciones en una transacción. La semántica del registro de pasos no cambia; su coste por saga cae por el factor del lote.

Particiona por agregado. Todos los eventos de un pedido deben procesarse en orden, pero dos pedidos distintos son independientes — que es exactamente lo que expresa el particionado del broker. Usa el id del agregado (`order_id`) como clave de cada evento de saga, y el sistema escala horizontalmente añadiendo particiones y consumidores mientras cada saga individual sigue siendo estrictamente secuencial. La trampa a evitar es el agregado caliente: un id de almacén como clave de partición de todos los eventos de stock serializa el inventario de toda la empresa detrás de un solo consumidor.

Separa la latencia de la saga de la del usuario. Una lectura errónea común de los "dos segundos" del caso de estudio es que el usuario esperó dos segundos. No debería: la API aceptó el pedido, arrancó la saga y devolvió `202 Accepted` con una URL de estado en bastante menos de 100 ms. La duración de la saga es reloj de backend, solapando colas y reintentos; la latencia de cara al usuario es solo la primera transacción local. Diseñar el contrato del front-end alrededor de "aceptado, aquí puedes mirar" — en lugar de secuestrar la conexión HTTP durante todo el flujo — es lo que permite que el mismo diseño de saga sirva un camino feliz de 2 segundos y uno degradado de 30 sin que los timeouts caigan en cascada hasta los navegadores.

Un número que vale la pena calcular antes del lanzamiento: multiplica los arranques de saga esperados por segundo por los pasos por saga por las escrituras por paso, y compáralo con el throughput de escritura cómodo de tu base de datos. Es una estimación de cinco minutos que ha movido repetidamente a equipos de la sorpresa de "¿por qué Postgres está al 80%?" a una estrategia de particionado aburrida y planificada. Cuando la estimación diga que una sola base de datos no puede con la contabilidad, las tablas de outbox e inbox shardan de forma natural por la misma clave de agregado que los eventos — el patrón escala hacia fuera por el eje que ya habías elegido.

¿Y los motores? La misma aritmética aplica con constantes distintas: Temporal convierte las transiciones de paso en apéndices al historial de eventos contra su propia persistencia (dimensionada y escalada con independencia de tus servicios), y Step Functions Standard factura por transición de estado, lo que convierte el factor de amplificación de escritura directamente en dólares. Ninguno elimina el coste; ambos lo mueven a un sitio con mejor tooling. La contabilidad del patrón saga nunca es gratis — la decisión de ingeniería es solo dónde aterriza la factura y quién opera el contador.

Por último, resiste la tentación de optimizar la contabilidad hasta hacerla desaparecer. Los equipos bajo presión de escritura a veces eliminan el registro de pasos ("podemos inferir el progreso desde las tablas de dominio") o se saltan el inbox ("los duplicados son raros"). Ambos recortes cambian un coste visible y presupuestable por uno invisible que vence durante un incidente: sin el registro de pasos, la recuperación se convierte en arqueología por las tablas de cuatro servicios; sin el inbox, el duplicado que era "raro" llega durante la tormenta de reintentos que sigue a cada caída, exactamente cuando más daño hace. Si la fontanería es demasiado cara a tu escala, shárdala o agrúpala — las dos palancas honestas — en lugar de quitar las piezas cuyo propósito entero es estar ahí el peor día del trimestre.

Errores comunes

Compensaciones que pueden fallar sin plan B. Una compensación debe ser idempotente y reintentable. Si reembolsar puede fallar permanentemente, necesitas una cola de reintentos y una alerta para intervención manual, no una excepción silenciada.

Pasos sin idempotencia. Los brokers entregan al menos una vez. Sin claves de idempotencia, un reintento duplica el cobro. Cada paso y cada compensación deben poder ejecutarse dos veces sin efectos dobles.

Borrar datos al compensar. Cancelar un pedido es marcarlo como CANCELLED, no hacer DELETE. Borrar destruye la trazabilidad y rompe a cualquier consumidor que ya leyó el dato.

Coreografía de ocho pasos. Si nadie puede dibujar el flujo de memoria, el patrón dejó de ayudar. Migra a orquestación antes de que un ciclo de eventos te lo exija.

Estado de saga solo en memoria. Un deploy a mitad de saga no puede significar un pedido cobrado y nunca confirmado.

Sin timeouts. Todo paso que espera respuesta necesita un timeout que dispare compensación o reintento. Una saga esperando eternamente es un bug, no un estado válido.

Checklist de producción

- Cada paso y cada compensación son idempotentes (clave de idempotencia o verificación de estado previo).
- La transacción pivote está identificada y todo lo posterior es reintentable.
- El estado de la saga se persiste en cada transición (o lo gestiona un motor durable).
- Los eventos se publican vía outbox, nunca con dual write.
- Los estados intermedios (PENDING_*) son visibles en los dashboards y tienen alertas de antigüedad: una saga vieja sin terminar es un incidente.

- Las compensaciones tienen reintentos con backoff y una ruta de escalado manual.
- Hay tests que matan el proceso en cada paso intermedio y verifican que la saga se recupera o compensa.

Preguntas frecuentes

¿Necesito una saga para cada llamada entre servicios? No. Una saga es para flujos de *escritura* multipaso que deben terminar consistentes. Una lectura compuesta desde tres servicios no es una saga; tampoco una escritura única seguida de notificaciones best-effort. Si perder la acción posterior es aceptable (un evento de analítica, un calentamiento de caché), basta una cola de reintentos normal. Recurre a una saga cuando un fallo parcial dejaría mal dinero, stock o estado legal.

¿Coreografía u orquestación — hay una regla dura? Una que funciona: cuenta los pasos y los equipos. Hasta tres pasos de uno o dos equipos, la coreografía se mantiene legible y máximamente desacoplada. Más pasos, más equipos, o cualquier conversación que empiece con "espera, ¿qué pasa exactamente cuando...?" — orquesta. Migrar después es normal: muchos sistemas empiezan coreografiados y crían un orquestador solo para su flujo más complejo, dejando eventos para los simples.

¿Dónde muerde de verdad el problema del aislamiento? Donde otro flujo lee estado intermedio y actúa sobre él. El clásico: un job de informes cuenta un pedido que luego se compensa, o una pantalla de cliente muestra "confirmado" para un pedido aún en `PENDING_PAYMENT`. El arreglo es disciplina, no maquinaria — los estados intermedios son estados reales de tu modelo de negocio, así que nómbralos, píntalos con honestidad en las UIs ("procesando") y exclúyelos de los informes hasta que sean terminales.

¿Puede compensarse una compensación? Evita diseñar eso. Las compensaciones deben ser terminales: idempotentes, reintetables y siempre capaces de triunfar al final (quizá vía escalado manual). Una compensación que necesita su propio deshacer significa que la frontera del paso original estaba mal trazada — normalmente dos acciones de negocio escondidas en un solo paso. Sepáralas.

¿Cómo convenzo a mi equipo de que necesitamos esto? Haz un experimento: mata el proceso entre dos escrituras "enlazadas" en staging y enseña cómo queda la base de datos. El patrón no es una moda arquitectónica; es la respuesta honesta a un modo de fallo que todo sistema distribuido ya tiene. La alternativa a una saga diseñada es una accidental, compensada a mano, en producción, con una hoja de cálculo.

¿El event sourcing reemplaza a las sagas? Son ortogonales. El event sourcing cambia cómo un servicio almacena su estado (como un log de eventos); las sagas coordinan cambios de estado *entre* servicios. Un servicio con event sourcing participa en una saga como cualquier otro — su transacción local añade eventos en lugar de actualizar filas. Puedes usar uno, ambos o ninguno.

¿Cómo interactúan las sagas con los reintentos y el backoff en la capa HTTP? Se apilan, y el orden de capas importa. Los reintentos por petición con backoff exponencial y jitter (cubiertos en la guía de reintentos de esta serie) viven *dentro* de un paso y manejan ruido transitorio; la ruta de fallo de la saga maneja desenlaces definitivos. Configura la capa interna para rendirse rápido — un paso que reintenta diez minutos antes de reportar fallo bloquea diez minutos la decisión de toda la saga. Un buen reparto: el paso reintenta durante segundos, la política de reintentos de la propia saga (para pasos reintetables) abarca minutos, y el reaper acota el total en horas.

¿Qué va en el contexto de la saga y qué se queda fuera? Todo lo que un paso posterior o una compensación necesiten: identificadores (id de pedido, de cargo, de reserva), importes tal como se confirmaron, y el error que disparó la compensación. Se queda fuera: cualquier cosa que puedas releer fresca (niveles de stock actuales), cualquier cosa grande (adjunta un id de documento, no el documento) y cualquier dato sensible que sobreviva a su necesidad — la columna de contexto es durable y aparece en registros de pasos, dashboards y auditorías, así que los números de tarjeta y los datos personales no pintan nada ahí. Trata el contexto como un registro público del flujo, porque operativamente eso es exactamente en lo que se convierte.

¿Hay una primera saga natural para construir? Elige un flujo con dos o tres pasos, un pivote claro y compensaciones que sean operaciones de negocio obvias que ya soportas (cancelar, liberar, reembolsar). El flujo de pedido es el clásico por algo. Evita empezar por tu flujo más complejo — la maquinaria (estado, recuperación, reaper, dashboards) es el verdadero entregable de la primera saga, y es mucho más fácil endurecerla sobre un flujo simple y reutilizarla después en todas partes.

Conclusión

El patrón saga es la respuesta pragmática a un hecho incómodo: en microservicios no hay transacciones globales. A cambio de renunciar a la atomicidad entre servicios, obtienes disponibilidad y desacoplamiento, pagando con complejidad explícita: compensaciones, idempotencia y estados intermedios que ahora forman parte de tu modelo de negocio. Empieza con coreografía si el flujo es corto y lineal; pasa a orquestación cuando crezca. Y hagas lo que hagas, publica los eventos con outbox y diseña cada compensación como si fuera a ejecutarse dos veces un viernes por la noche, porque tarde o temprano lo hará.

The Saga Pattern: Distributed Transactions Across Microservices Without 2PC

Guide | Premium | Intermediate | Real Projects

Reading time: ~35 min · July 2026

> Learn how to keep data consistent across microservices without two-phase commit using the saga pattern: choreography versus orchestration, well-designed compensating transactions, pivot transactions, the isolation problem and its countermeasures, and how to combine sagas with the transactional outbox so no event ever gets lost.

The problem: one business operation spread across services

In a monolith, creating an order, charging the customer, and reserving stock fits in a single ACID transaction: either everything happens, or nothing does. Split the system into microservices and that guarantee disappears, because each service owns its own database. The order lives in one service, the payment in another, the inventory in a third. If the charge fails after the order was created, nobody rolls back for you.

The classic answer, two-phase commit (2PC), coordinates an atomic commit across several databases, but it fits microservices poorly: the coordinator is a single point of failure, locks are held for the entire conversation (crushing throughput), and many modern building blocks —Kafka, most NoSQL databases, any third-party API— simply do not support it. You need a different strategy, and that strategy is the saga.

What a saga is

A saga breaks the distributed transaction into a sequence of local transactions. Each step commits in its own service's database and then publishes an event or notifies a coordinator; the next step picks up from there. If a step fails, the saga runs compensating transactions that undo the already-committed steps, in reverse order.

The key phrase is "semantically undo". A compensation is not a ROLLBACK: earlier steps already committed, and other processes may have read that data. Compensating means running a new transaction that reverses the business effect: if you charged, you refund; if you reserved stock, you release it; if you created the order, you mark it cancelled (you don't delete it).

Within a saga it pays to distinguish three kinds of step:

- **Compensatable transactions:** steps that can be undone if something later fails (reserving stock, creating the order in a pending state).

- **Pivot transaction:** the point of no return. Once it commits (the charge, say), the saga no longer goes backwards: it must run forward to completion.
- **Retriable transactions:** steps after the pivot that must not fail permanently; they are retried with backoff until they succeed (sending the confirmation email, setting the final state).

Choreography: the saga as a chain of events

In a choreographed saga there is no coordinator. Each service listens for the events it cares about and reacts by publishing its own. For an order flow: the order service publishes order.created, inventory reserves stock and publishes stock.reserved, payments charges and publishes payment.completed or payment.failed, and the order service listens for the outcome to confirm or cancel.

```
# Payment service: reacts to the inventory service's event
@consumer.on("stock.reserved")
def handle_stock_reserved(event):
    order = event["payload"]
    try:
        charge = payment_gateway.charge(
            customer=order["customer_id"],
            amount_cents=order["total_cents"],
            # Idempotency key: a broker redelivery won't double-charge
            idempotency_key=f"order-{order['order_id']}-charge",
        )
        publish("payment.completed", {"order_id": order["order_id"], "charge_id": charge.id})
    except CardDeclinedError as e:
        # Don't raise: publish the failure to trigger compensations
        publish("payment.failed", {"order_id": order["order_id"], "reason": str(e)})
```

And in the inventory service, the compensation is simply another handler:

```
# Inventory service: compensates when payment fails
@consumer.on("payment.failed")
def handle_payment_failed(event):
    release_reservation(order_id=event["payload"]["order_id"])  # idempotent
    publish("stock.released", {"order_id": event["payload"]["order_id"]})
```

Choreography shines through low coupling: no central service knows about all the others, and adding a new consumer doesn't touch the existing flow. Its big weakness shows up with size: the complete flow is written down nowhere. To know what the saga does you have to read handlers across five services, and a cyclic dependency between events can go unnoticed until it reaches production.

Orchestration: an explicit coordinator

In an orchestrated saga, one central piece defines the steps, invokes each service, and decides what to compensate when something fails. The whole flow reads in a single file:

```
from dataclasses import dataclass
from typing import Awaitable, Callable, Optional

@dataclass
class SagaStep:
    name: str
```

```
    action: Callable[[dict], Awaitable[dict]]                # local transaction
    compensation: Optional[Callable[[dict], Awaitable[None]]] # how to undo it

class CreateOrderSaga:
    def __init__(self, orders, inventory, payments, repo):
        self.repo = repo   # durable saga state
        self.steps = [
            SagaStep("create_order", orders.create_pending, orders.cancel),
            SagaStep("reserve_stock", inventory.reserve, inventory.release),
            SagaStep("charge", payments.charge, None),        # pivot
            SagaStep("confirm_order", orders.confirm, None),  # retriable
        ]

    async def execute(self, ctx: dict) -> dict:
        saga_id = await self.repo.start("create_order", ctx)
        completed: list[SagaStep] = []
        for step in self.steps:
            try:
                ctx = await step.action(ctx)
                completed.append(step)
                # Durable state lets the saga resume after a restart
                await self.repo.mark_step_done(saga_id, step.name, ctx)
            except Exception:
                await self._compensate(saga_id, completed, ctx)
                raise
        await self.repo.mark_completed(saga_id)
        return ctx

    async def _compensate(self, saga_id, completed, ctx):
        for step in reversed(completed):
            if step.compensation:
                # Compensations are retried: they can't fail without a plan B
                await retry_with_backoff(step.compensation, ctx, max_attempts=10)
            await self.repo.mark_compensated(saga_id, step.name)
```

Note self.repo: the saga's state must survive an orchestrator restart. If the process dies between the charge and the confirmation, on startup it has to know exactly where it left off and continue (or compensate). Writing that machinery by hand is doable, but in production most teams delegate to a durable workflow engine such as Temporal or AWS Step Functions, which persists every step and resumes execution automatically. The trend in 2026 is clear: once a saga grows past three or four steps, orchestration wins on visibility, debugging, and control.

The isolation problem

Sagas give up the I in ACID. Between steps, intermediate changes are already committed and visible to the rest of the system, which opens the door to anomalies: another process can read an order that later gets cancelled (a business-level dirty read), or two concurrent sagas can clobber each other's updates. The classic countermeasures:

- **Semantic lock:** the record carries a status flagging that a saga is in flight. An order in PENDING_PAYMENT is not shown as confirmed and cannot be modified by other flows.
- **Commutative updates:** design operations whose order doesn't matter (adding and subtracting balance) so two concurrent sagas can't corrupt each other.
- **Pessimistic view:** reorder steps so the riskiest work happens as late as possible, minimizing the window in which a compensation leaves inconsistent data visible.

- **Reread before writing:** before a critical step, verify the data hasn't changed since the saga read it; if it has, abort or recompute.

```
-- Semantic lock: the status marks the record as "in flight"
UPDATE orders SET status = 'PENDING_PAYMENT'
WHERE id = :order_id AND status = 'CREATED';
-- Every other flow knows a PENDING_* order may still be reverted
```

Sagas and the transactional outbox

One failure mode breaks any event-driven saga: the service commits its local transaction but dies before publishing the event. Result: the saga hangs forever. The fix is the transactional outbox pattern, covered in depth in another guide in this series: the event is written to an outbox table inside the same local transaction, and a relay publishes it afterwards. That makes each saga step —state change plus event— truly atomic. Sagas and outbox don't compete: they complement each other, and in production they almost always travel together.

Building your own orchestrator: a data model that survives restarts

The orchestrator sketched above hides its hardest requirement inside one innocent-looking line: `self.repo`. Durable saga state is what separates a demo from a production system, so it is worth making that repository concrete. The minimal schema needs two tables: one for the saga instance and one for its step log.

```
CREATE TABLE saga_instances (
  id            UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  saga_type     TEXT        NOT NULL,          -- 'create_order'
  status        TEXT        NOT NULL,          -- RUNNING | COMPENSATING | COMPLETED | FAILED
  context       JSONB       NOT NULL,          -- accumulated step outputs
  current_step  TEXT,
  created_at    TIMESTAMPTZ NOT NULL DEFAULT now(),
  updated_at    TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE saga_steps (
  saga_id     UUID  NOT NULL REFERENCES saga_instances(id),
  step_name   TEXT  NOT NULL,
  status      TEXT  NOT NULL,        -- DONE | COMPENSATED | FAILED
  output      JSONB,
  finished_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  PRIMARY KEY (saga_id, step_name)
);

-- The reaper and the dashboards live on this index:
CREATE INDEX idx_sagas_running
  ON saga_instances (updated_at)
  WHERE status IN ('RUNNING', 'COMPENSATING');
```

Three design decisions matter here. First, `context` is a JSONB snapshot of everything the steps have produced so far (the order id, the charge id, the reservation id); a step that needs to be compensated months later must find every identifier it needs inside that column, never in memory. Second, the step log is append-per-step with a composite primary key, which makes

marking a step done naturally idempotent: re-running a recovered saga that crashed after committing a step but before recording it can `INSERT ... ON CONFLICT DO NOTHING` and move on. Third, the partial index means the question every operator asks — "what is in flight and for how long?" — never scans completed history.

Recovery on startup is then a loop, not an art:

```
async def resume_pending_sagas(repo, registry):
    """Called once on service startup, before accepting new work."""
    for instance in await repo.find_unfinished():        # uses idx_sagas_running
        saga = registry.build(instance.saga_type)
        done = await repo.step_names_done(instance.id)
        if instance.status == "COMPENSATING":
            # Crashed mid-undo: keep compensating from where it stopped
            await saga.compensate_remaining(instance, done)
        elif saga.passed_pivot(done):
            # Past the point of no return: only direction is forward
            await saga.run_remaining(instance, done)
        else:
            # Before the pivot the safe default is to roll back
            await saga.compensate_remaining(instance, done)
```

Notice the asymmetry around the pivot: an interrupted saga that already charged the customer is resumed forward, while one that only reserved stock is compensated. Encoding that decision in the recovery routine — instead of leaving it to whoever is on call at 3 a.m. — is precisely why you identified the pivot transaction in the first place.

Timeouts and the reaper: no saga waits forever

Every step that waits on another service needs two timers, not one. The request timeout (seconds) bounds a single call; the saga timeout (minutes or hours, depending on the flow) bounds the whole business operation. The second one is easy to forget because nothing crashes when it is missing — orders just accumulate silently in `PENDING_PAYMENT` until a customer writes in.

The standard countermeasure is a *reaper*: a small periodic job that finds sagas whose `updated_at` is older than the per-type deadline and pushes them to a terminal state.

```
REAPER_DEADLINES = {"create_order": timedelta(minutes=30)}

async def reap_stuck_sagas(repo):
    for saga_type, deadline in REAPER_DEADLINES.items():
        stuck = await repo.find_older_than(saga_type, deadline)
        for instance in stuck:
            if instance.passed_pivot():
                # Money moved: escalate to a human, never auto-cancel
                await alerts.page(
                    "saga-stuck-past-pivot",
                    saga_id=instance.id, minutes=instance.age_minutes(),
                )
            else:
                await repo.mark(instance.id, "COMPENSATING")
                await queue.enqueue("compensate_saga", saga_id=instance.id)
```

The branch on the pivot is again the interesting part. Before the pivot, timing out into compensation is safe and automatic. After it, an automatic cancellation would refund a customer who might already have received their confirmation email, so the correct move is a page, not a

state change. A useful rule of thumb: **the reaper may compensate, but only a human may abandon a saga past its pivot.**

Two operational details save real incidents. Make the reaper's interval much shorter than the shortest deadline (a 30-minute deadline checked hourly is a 90-minute deadline in practice). And emit a metric per reaped saga — a sudden spike in reaped `create_order` sagas is usually the first visible symptom of a broken downstream dependency.

Idempotency in practice: the inbox table

The article has repeated "make every step idempotent" like a mantra; here is what that means mechanically. Brokers deliver at least once, so every consumer in a choreographed saga — and every compensation handler in an orchestrated one — will eventually see the same message twice. The general-purpose defense is the *transactional inbox*: record the message id in the same local transaction that applies its effect.

```
CREATE TABLE processed_messages (
  consumer   TEXT NOT NULL,   -- 'inventory-service'
  message_id TEXT NOT NULL,   -- from the event envelope
  processed_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  PRIMARY KEY (consumer, message_id)
);
```

```
async def handle_stock_reserved(event, db):
    async with db.transaction():
        inserted = await db.execute(
            """INSERT INTO processed_messages (consumer, message_id)
               VALUES ('payment-service', :mid)
               ON CONFLICT DO NOTHING""",
            mid=event["message_id"],
        )
        if inserted == 0:
            return                      # duplicate delivery: ack and move on
        await apply_payment_effects(event, db)   # same transaction
```

Because the dedup insert and the business effect commit atomically, a crash between them is impossible by construction — the classic failure of checking a cache "have I seen this?" before doing the work, then crashing before recording it, simply cannot happen. For steps that call external providers, pair the inbox with the provider's own idempotency mechanism (like the `Idempotency-Key` the charge example already uses): the inbox protects your database, the key protects the third party.

One subtlety: the inbox table grows forever unless you prune it. A retention window slightly longer than your broker's maximum redelivery horizon (a week is a common choice) keeps it small; deleting rows older than that is safe because the broker can no longer redeliver those messages.

Sagas on a durable workflow engine: Temporal

Everything built by hand so far — the state tables, the recovery loop, the reaper — is exactly what a durable execution engine gives you for free. In Temporal, the workflow's own execution state *is* the saga state: the engine records every step in an event history, and if the worker process dies,

another worker replays the history and continues from the last completed activity as if nothing happened. No `saga_instances` table, no reaper, no hand-written recovery.

The saga pattern in Temporal is usually expressed as a growing list of compensations and one `try/except`:

```
from datetime import timedelta
from temporalio import workflow
from temporalio.common import RetryPolicy

@workflow.defn
class CreateOrderSaga:
    @workflow.run
    async def run(self, order: OrderInput) -> str:
        compensations = []          # grows as steps commit
        opts = dict(
            start_to_close_timeout=timedelta(seconds=30),
            retry_policy=RetryPolicy(maximum_attempts=5),
        )
        try:
            order_id = await workflow.execute_activity(
                create_pending_order, order, **opts)
            compensations.append((cancel_order, order_id))

            resv_id = await workflow.execute_activity(
                reserve_stock, order, **opts)
            compensations.append((release_stock, resv_id))

            # Pivot: after this, no more compensating
            await workflow.execute_activity(charge_customer, order, **opts)

            await workflow.execute_activity(confirm_order, order_id, **opts)
            return order_id
        except Exception:
            for comp, arg in reversed(compensations):
                await workflow.execute_activity(
                    comp, arg,
                    start_to_close_timeout=timedelta(seconds=30),
                    # Compensations retry effectively forever
                    retry_policy=RetryPolicy(maximum_attempts=0),
                )
            raise
```

Each `execute_activity` call is a saga step: it runs in a normal Python process, can talk to any database or API, and its result is durably recorded before the workflow moves on. The `reversed(compensations)` loop is the same LIFO unwinding as the hand-rolled orchestrator, but the list itself survives crashes because it is part of the replayed workflow state. Note the two retry policies: forward steps retry a bounded number of times and then trigger the saga's failure path, while compensations retry indefinitely — a compensation that gives up is exactly the "no plan B" mistake from the common-mistakes list.

The trade-offs are real, though. Workflow code must be deterministic (no direct I/O, random numbers, or wall-clock reads outside activities), which is a genuinely different programming model; the engine is another stateful piece of infrastructure to operate (or a vendor bill if you use the hosted version); and testing shifts toward Temporal's replay-based test framework. For a team running two simple sagas, that overhead may not pay for itself. For a team about to build its third hand-rolled recovery loop, it almost certainly does.

Sagas as state machines: AWS Step Functions

The same orchestration logic can live in a managed state machine instead of code. In AWS Step Functions, each saga step is a state (typically invoking a Lambda function), `Retry` handles transient failures with exponential backoff, and `Catch` routes definitive failures to the compensation chain:

```
{
  "StartAt": "ReserveStock",
  "States": {
    "ReserveStock": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:reserve-stock",
      "Retry": [{
        "ErrorEquals": ["States.Timeout", "Lambda.ServiceException"],
        "MaxAttempts": 3, "IntervalSeconds": 2, "BackoffRate": 2.0
      }],
      "Catch": [{
        "ErrorEquals": ["States.ALL"],
        "ResultPath": "$.error",
        "Next": "CancelOrder"
      }],
      "Next": "ChargeCustomer"
    },
    "ChargeCustomer": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:charge-customer",
      "Catch": [{
        "ErrorEquals": ["States.ALL"],
        "ResultPath": "$.error",
        "Next": "ReleaseStock"
      }],
      "Next": "ConfirmOrder"
    },
    "ReleaseStock": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:release-stock",
      "Retry": [{ "ErrorEquals": ["States.ALL"], "MaxAttempts": 10, "BackoffRate": 1.5 }],
      "Next": "CancelOrder"
    },
    "CancelOrder": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:cancel-order",
      "Retry": [{ "ErrorEquals": ["States.ALL"], "MaxAttempts": 10, "BackoffRate": 1.5 }],
      "End": true
    },
    "ConfirmOrder": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:function:confirm-order",
      "End": true
    }
  }
}
```

Read the `Catch` targets carefully and you will see the saga's shape encoded in the wiring: a failure in `ReserveStock` only needs to cancel the order, while a failure in `ChargeCustomer` must first release stock and then cancel — each state's catch points at the right entry in the compensation chain, and the compensations themselves link forward through the remaining undo steps. The console renders all of this as a diagram, which is orchestration's visibility argument made literal:

the flow nobody could draw from memory in an eight-step choreography is now a picture the whole team can point at.

The limits are the mirror image of Temporal's: the definition language is JSON rather than a general-purpose language, so complex branching and data reshaping get awkward; and per-transition pricing on Standard workflows means a chatty saga has a real unit cost. It fits best when your steps are already Lambda functions and your team prefers managed services over operating an engine.

Parallel branches: when the saga is a DAG, not a list

Real flows are rarely purely sequential. Reserving stock and running a fraud check do not depend on each other; forcing them into a sequence adds their latencies. A saga can run independent steps concurrently — it just changes what compensation means, because now a failure in one branch must undo *sibling branches that may still be in flight*.

```
async def execute_parallel_phase(self, ctx):
    results = await asyncio.gather(
        self.inventory.reserve(ctx),
        self.fraud.check(ctx),
        return_exceptions=True,      # never lose a sibling's outcome
    )
    failures = [r for r in results if isinstance(r, Exception)]
    if failures:
        # Compensate every branch that DID succeed
        if not isinstance(results[0], Exception):
            await retry_with_backoff(self.inventory.release, ctx)
        raise SagaStepFailed(failures[0])
    ctx.update(results[0]); ctx.update(results[1])
    return ctx
```

The rules that keep this manageable: only parallelize steps that are all compensatable (a pivot inside a parallel branch means a sibling failure can strand you past the point of no return with another branch half-done); always gather with `return_exceptions=True` so one branch's exception cannot orphan another branch's successful — and therefore compensatable — result; and treat the whole parallel block as a single unit in the step log, recording which branches completed. If you find yourself drawing a diamond-shaped dependency graph with pivots in the middle, that is the signal to move to an engine: Temporal expresses this with ordinary `asyncio.gather` inside a workflow, and Step Functions has a native `Parallel` state with per-branch catches.

Sagas, TCC, and 2PC: picking a consistency strategy

The saga is not the only alternative to 2PC, and it is worth knowing when its cousins fit better. **Try-Confirm-Cancel (TCC)** moves the reservation idea into the protocol itself: every participant exposes three operations, and the coordinator first calls `try` on everyone (reserving resources in a tentative state), then `confirm` on everyone if all tries succeeded, or `cancel` on everyone otherwise.

The practical difference from a saga: in TCC, nothing business-visible happens until the confirm phase, so the isolation problem largely disappears — a tentative hotel hold is invisible to other customers in a way a committed-then-compensated booking is not. The cost is intrusiveness: every participating service must implement the three-operation contract and manage expiring

reservations, which is why TCC tends to appear in domains that already think in holds and captures — payments (authorize/capture/void is TCC in disguise), travel booking, ticketing — and rarely anywhere else.

A quick decision table, honestly stated:

- **2PC/XA**: still fine inside one trust boundary where every participant genuinely supports it (two relational databases you own, one XA-capable broker) and the latency cost of synchronous locking is acceptable. It dies the moment a participant is Kafka, a NoSQL store, or somebody else's HTTP API.
- **TCC**: strong isolation needs, participants you control, a domain with natural reservation semantics. Pay the price of implementing try/confirm/cancel in every service.
- **Saga**: the general-purpose answer for cross-service workflows with heterogeneous participants. Weakest isolation, strongest applicability; the countermeasures from the isolation section are the tax.
- **No pattern at all**: the most underrated option. If two "services" always change together transactionally, they may be one service wearing two deployment units — merging them buys back ACID for free. A saga you did not need is pure operational debt.

Observability: seeing a saga as one trace

A saga scatters one business operation across services, queues, and minutes of wall-clock time, which is precisely the shape of problem distributed tracing exists for. The goal: paste one order id into your tracing UI and see every step, every retry, and every compensation as a single connected story.

The mechanics differ by style. In an orchestrated saga, make the orchestrator's `execute` the root span and each step a child — trivial, since it is one process calling others. In a choreographed saga there is no root process, so propagate the trace context *through the event envelope*: the service that starts the flow injects `traceparent` into the event metadata (the outbox relay must copy it), and every consumer extracts it and starts its span as a child. Without this, each hop starts a fresh trace and your "one story" shatters into five fragments connected only by grep.

```
from opentelemetry import trace
from opentelemetry.propagate import extract, inject

def publish_with_context(topic, payload):
    headers = {}
    inject(headers)                          # writes traceparent
    outbox.enqueue(topic, payload, headers=headers)

def consume(event):
    ctx = extract(event.headers)
    with tracer.start_as_current_span(
        "saga.step " + event.topic, context=ctx,
        attributes={
            "saga.id": event.payload["order_id"],
            "saga.step": event.topic,
            "messaging.redelivered": event.redelivered,
        },
    ):
        handle(event)
```

On top of traces, three metrics cover most operational questions: a counter of sagas started/completed/compensated per type (the compensation *rate* is the health signal — compensations are normal, a rising ratio is not), a histogram of end-to-end saga duration per type (watch the p99, where stuck-but-not-yet-reaped sagas hide), and a gauge of in-flight sagas older than N minutes, which is the reaper's queue and belongs on the on-call dashboard. Alert on the gauge and the rate, not on individual compensations: a saga that compensates cleanly is the system working as designed.

Testing sagas: kill the process at every step

The production checklist demands tests that kill the process at every intermediate step; here is what that looks like concretely. The technique is crash-point injection: the saga under test runs against fake services, one of which is rigged to raise — or to "die" — at a chosen step, and the test then restarts the orchestrator and asserts the world converges.

```
import pytest

class CrashAfter:
    """Wraps a step to simulate the process dying right after commit."""
    def __init__(self, real_action):
        self.real_action = real_action
    async def __call__(self, ctx):
        await self.real_action(ctx)      # local transaction commits...
        raise ProcessKilled()            # ...then the process is gone

@pytest.mark.parametrize("crash_step", [
    "create_order", "reserve_stock", "charge", "confirm_order",
])
async def test_saga_recovers_from_crash_at_every_step(crash_step, saga_env):
    saga = saga_env.build_saga(crash_wrapper=CrashAfter, at_step=crash_step)
    with pytest.raises(ProcessKilled):
        await saga.execute(saga_env.order_ctx())

    # "Restart": a fresh orchestrator over the same durable state
    await resume_pending_sagas(saga_env.repo, saga_env.registry)

    final = await saga_env.repo.get_by_context(saga_env.order_ctx())
    if crash_step in ("charge", "confirm_order"):     # pivot or later
        assert final.status == "COMPLETED"            # ran forward
        assert saga_env.orders.state == "CONFIRMED"
    else:
        assert final.status in ("COMPENSATED", "FAILED")
        assert saga_env.inventory.reservations == {}  # nothing leaked
        assert saga_env.payments.charges == []        # nobody charged
```

The parametrization is the point: one test body, executed once per step, asserting the invariant that actually matters — *crashes before the pivot leak nothing; crashes at or after the pivot complete the flow*. Complement it with a duplicate-delivery test (feed every event to its consumer twice, assert effects applied once — this is the inbox table earning its keep) and, if you use real infrastructure in CI, run the fakes' roles against Postgres and your broker via Testcontainers so the `ON CONFLICT` and `FOR UPDATE` details are exercised for real. Teams on Temporal get a head start here: the replay test framework can inject failures at any activity boundary without hand-writing the crash machinery.

Versioning: deploying while sagas are in flight

Sagas run for minutes or days, so every deploy happens with instances mid-flight. Change the saga's definition carelessly — reorder steps, rename one, change a compensation's signature — and the new code wakes up holding state written by the old code that it no longer understands.

The safe rules are few and mechanical. Version the definition: store `saga_type = 'create_order.v2'` (or a version column) on the instance, keep the v1 step list registered until the last v1 instance finishes, and route new starts to v2. Never rename a recorded step name — the step log is a contract with your own recovery code. Treat compensations like public APIs: the v2 deploy must still be able to compensate a step committed by v1, so compensation inputs live in the saga `context`, not in code-level assumptions. And when a change is truly incompatible, the boring strategy wins: *drain* — deploy v2 alongside v1, stop starting v1 sagas, wait for the running ones to finish, then delete v1. Workflow engines make this explicit (Temporal has first-class worker versioning for exactly this problem), but the drain strategy works identically for a hand-rolled orchestrator.

Case study: the order flow, end to end

To make everything concrete, walk one order through a system that combines every pattern in this guide. The cast: an order service (orchestrator embedded), inventory, payments, and a notification service; Postgres in each; Kafka between them; outbox relays everywhere.

The happy path. A customer submits an order. The order service starts a saga instance and, in one local transaction, creates the order in `PENDING_STOCK` (semantic lock) and writes the instance row. Step two calls inventory, which inserts the event id into its inbox, decrements available stock, records a reservation, and writes `stock.reserved` to its outbox — one transaction, all four writes. The relay publishes; the orchestrator marks the step done and advances the context. Step three charges through the payment provider with idempotency key `order-91f4-charge`; this is the pivot. Step four flips the order to `CONFIRMED` and enqueues the confirmation email as a retriable step. Total wall-clock time: around two seconds, four local transactions, zero distributed locks.

The interesting path. Same order, but the payment provider times out — the request may or may not have gone through. The step's retry policy retries with the same idempotency key, which is what makes the retry safe: if the first attempt actually charged, the provider returns the original result instead of charging again. Suppose all retries exhaust. The pivot never committed, so the orchestrator flips to `COMPENSATING` and unwinds: release the reservation (inventory's inbox swallows the duplicate when the message is redelivered), mark the order `CANCELLED` — not deleted — and record the saga `FAILED` with the payment error in context. The customer sees an honest "payment didn't go through"; the dashboards see one more count on `saga_compensated_total`; nobody gets paged, because a clean compensation is the system working.

The 3 a.m. path. A deploy restarts the order service between the charge succeeding and the confirmation committing. On startup, `resume_pending_sagas` finds the instance: step log shows `charge: DONE`, so the saga is past its pivot and runs forward — confirm, notify, complete. The customer never knows. That deploy happening five minutes earlier, before the charge, would have compensated instead. Both outcomes are correct, both are automatic, and both were chosen by code you wrote in daylight rather than by an engineer at 3 a.m. That is the entire value proposition of the pattern in one sentence.

Designing step boundaries: the craft behind the pattern

Most saga bugs are not coding errors; they are steps drawn in the wrong place. A step boundary is a promise: "this unit either fully happens in one service or can be semantically undone." Three heuristics produce good boundaries in practice.

One owner per step. A step should touch exactly one service's data. The moment a "step" needs to update two services, it is a nested distributed transaction wearing a trench coat — split it into two steps with their own compensations, or you have recreated the original problem one level down.

Write the compensation first. Before implementing a step, write one sentence describing its undo: "release the reservation", "refund the charge", "mark the order cancelled". If you cannot write that sentence — "un-send the email", "un-notify the regulator" — you have found a step that cannot precede the pivot. Either move it after the pivot (as a retriable step) or redesign it into a reservable form: instead of sending the email, *schedule* it with a delay, which is trivially cancellable until it fires.

Push risk late, push reads early. Steps that are likely to fail for business reasons (credit checks, stock validation) belong at the front, where compensating is cheap. Steps whose failure is rare and technical belong near the pivot. This ordering — the "pessimistic view" countermeasure applied at design time — minimizes both the number of compensations that ever run and the window of visible intermediate state.

A worked example of redesigning a bad boundary. Version one of a signup flow had a step "create account and send welcome email" — one step, two effects, one of them uncompensatable. Version two splits it: step one creates the account in `PENDING` (compensation: mark `ABORTED`); the pivot activates the account; a retriable step after the pivot sends the email. The flow is now fully compensatable before the point of no return, and the only uncompensatable effect happens strictly after it. That restructuring move — *reservation before the pivot, side effects after it* — resolves probably half of all "but this step can't be undone!" objections in design reviews.

When compensation fails: dead letters and human escalation

The mantra "compensations must always eventually succeed" collides with reality: the refund API is down for six hours, the inventory service rejects the release because the SKU was retired mid-saga, the compensation has retried forty times and it is now Saturday. A production saga needs a designed answer for the day the undo itself breaks.

The pattern is a compensation dead-letter queue with human escalation, and its core is a strict rule: **a failed compensation may never be dropped, only parked.**

```
async def run_compensation(self, saga_id, step, ctx):
    try:
        await retry_with_backoff(
            step.compensation, ctx,
            max_attempts=10, max_delay=timedelta(minutes=5),
        )
        await self.repo.mark_compensated(saga_id, step.name)
    except RetriesExhausted as e:
        # Park it: durable record + loud alert, never a swallowed log line
        await self.repo.park_compensation(
            saga_id=saga_id, step=step.name,
            context=ctx, error=str(e),
```

```
        )
        await alerts.page(
            "compensation-parked",
            saga_id=saga_id, step=step.name,
            runbook="https://wiki.internal/runbooks/parked-compensations",
        )
```

The parked row carries everything a human needs to finish the job by hand or requeue it after the outage: the full saga context, the failing step, the error. The operational side matters as much as the code: a runbook per compensation type ("how to manually release a reservation"), a small admin action to requeue parked compensations in bulk after the downstream recovers, and a standing rule that parked compensations older than one business day are incidents, not backlog. Teams that skip this build it after their first six-hour payment-provider outage, with worse tooling and more adrenaline.

Long-lived sagas: days, humans, and deadlines

Nothing in the pattern says a saga finishes in seconds. A loan application, a KYC verification, or a B2B order with manual approval is still a saga — local transactions, compensations, a pivot — except some steps wait on a human for days. Three things change at that timescale.

Waiting becomes a step. The saga needs first-class "wait for external signal" steps: the approval arrives as an event (or an API callback) that resumes the instance. Hand-rolled orchestrators model this as a step in state `WAITING` that the resume routine skips until the signal lands; engines have it natively (Temporal signals, Step Functions task tokens and its callback pattern). The step log gains a timestamp discipline: a saga waiting three days is normal, so the reaper's deadlines become per-step rather than per-saga — thirty minutes for the charge, five business days for the approval.

Compensations age. Undoing a three-day-old step may differ from undoing it three seconds later. Releasing a stock reservation after seconds is an `UPDATE`; after three days, the sales period may have closed, the price may have changed, the warehouse may have counted inventory. Long-lived compensations should re-read current state and decide, rather than blindly reversing — which is the "reread before writing" countermeasure, now mandatory instead of optional.

People need to see it. A saga that spans days is a business process, and business processes get asked about: "where is order 91f4 stuck?" Build the query before you need it — the `saga_instances` table already answers it with a two-line SQL view joining status, current step, and age. Exposing that view read-only to support teams eliminates a whole category of engineering interruptions.

Queries during a saga: what does the rest of the system see?

The isolation section covered writers colliding; readers deserve their own treatment, because every screen and report in the company reads saga-touched data. The question "what is this order's status?" has a subtle answer while a saga is in flight, and pretending otherwise produces bugs that no amount of orchestration fixes.

The practical rules: **terminal states for machines, honest states for humans.** Downstream systems that trigger on order status (shipping, invoicing, loyalty points) must react only to terminal states — `CONFIRMED`, `CANCELLED` — never to intermediate ones. Wiring shipping to fire on

`PENDING_PAYMENT` "because it's usually fine" is how a compensated order ends up on a truck. Human-facing surfaces, meanwhile, should show intermediate states truthfully but gently: "processing your payment" is honest; "confirmed" is a lie that a compensation will expose.

Aggregations need a policy, not an accident: reports either count only terminal states (the usual choice for revenue) or count in-flight sagas explicitly as their own bucket ("$4,200 in pending orders"). The anti-pattern is counting intermediates as if terminal and silently restating yesterday's numbers when compensations land — finance teams notice, and the trust lost is expensive. If you run read models or a CQRS-style view store, project intermediate states into them with the status attached, and let each consumer apply its own policy; stripping the status at projection time throws away exactly the information readers needed.

Auditing and the paper trail

A pleasant side effect of doing sagas properly: you get an audit log for free. The step log — every step, every compensation, every timestamp, every context snapshot — is precisely the record an auditor, a customer-support agent, or a fraud investigation asks for: *what happened to this order, in what sequence, and why was this customer refunded?*

Two habits turn the step log from debugging data into an audit-grade trail. Record *why*, not only *what*: the compensation row should carry the triggering error ("payment declined: insufficient funds"), because "order cancelled" without a cause is an invitation to guesswork later. And never mutate step-log rows — append corrections as new rows. An append-only step log plus the outbox's event history reconstructs any incident weeks after the fact, which is the difference between "we believe the refund went out" and a timestamped row that proves it. If your domain is regulated, retention policy on `saga_steps` becomes a compliance requirement rather than a housekeeping choice — decide it deliberately.

Where does the orchestrator live?

Orchestration answers "who coordinates?", but not "where does that code run?" — and the deployment choice shapes team ownership more than any technical property. There are three viable homes.

Embedded in the initiating service. The order service owns the create-order saga: the orchestrator is a module inside it, sharing its database for saga state. This is the default that most teams should start with. The service that owns the business outcome owns the coordination; no new deployable appears; the saga state rides the service's existing backup and migration story. Its limit is reuse — when three services each embed two sagas, patterns get copy-pasted and drift.

A dedicated saga service. One deployable whose only job is running saga definitions. It centralizes the machinery (state tables, reaper, parked-compensation tooling, dashboards) and gives operators a single pane of glass. The risk is organizational, not technical: a central "workflow team" easily becomes a bottleneck that every feature must pass through, and the service accumulates knowledge of every domain's events — the god-service anti-pattern with better marketing. It works when a platform team treats it as infrastructure others self-serve on, and fails when it becomes a gatekeeper.

An engine. Temporal, Step Functions, and their kin are the dedicated-service option bought instead of built, with the recovery machinery already hardened. The ownership question does not disappear, though: workflow definitions still need owners, versioning discipline still applies, and someone still operates (or pays for) the engine.

The migration path between these is smoother than it looks, provided one rule holds from day one: **keep saga definitions separate from saga machinery.** A definition — the ordered step list with actions and compensations, like the `SagaStep` table earlier — expresses pure business policy. If it never imports the persistence layer directly, moving from embedded to dedicated to engine is a re-hosting exercise rather than a rewrite.

Throughput: what sagas cost at scale

A saga multiplies writes. One business operation that was a single transaction in the monolith becomes, in the order-flow case study: four local transactions, four outbox rows, four relay publishes, four inbox inserts, plus a saga-state write per transition — roughly a factor of four in write amplification, before retries. At ten orders per second nobody notices. At a thousand, the saga plumbing itself becomes a capacity line item, and three techniques keep it affordable.

Batch the bookkeeping. The relay already batches outbox publishes; saga-state transitions can batch too. Marking a step done does not need its own round trip — orchestrators that process events in batches (consuming a partition's worth at a time) can commit a batch of step transitions in one transaction. The step log's semantics do not change; its cost per saga drops by the batch factor.

Partition by aggregate. All events for one order must be processed in order, but two different orders are independent — which is exactly what broker partitioning expresses. Key every saga event by the aggregate id (`order_id`), and the system scales horizontally by adding partitions and consumers while each individual saga stays strictly sequential. The trap to avoid is a hot aggregate: one warehouse id as the partition key for every stock event serializes the whole company's inventory behind one consumer.

Separate the saga's latency from the user's. A common misreading of the case study's "two seconds" is that the user waited two seconds. They should not have: the API accepted the order, started the saga, and returned `202 Accepted` with a status URL in well under 100 ms. The saga's duration is backend wall-clock, overlapping queues and retries; user-facing latency is only the first local transaction. Designing the front-end contract around "accepted, here is where to watch" — rather than holding the HTTP connection hostage to the whole flow — is what lets the same saga design serve both a 2-second happy path and a 30-second degraded one without timeouts cascading back to browsers.

One number worth computing before launch: multiply expected saga starts per second by steps per saga by writes per step, and compare against your database's comfortable write throughput. It is a five-minute estimate that has repeatedly moved teams from "why is Postgres at 80%?" surprise to a boring, planned partition strategy. When the estimate says one database cannot carry the bookkeeping, the outbox and inbox tables shard naturally by the same aggregate key as the events — the pattern scales out along the axis you already chose.

What about the engines? The same arithmetic applies with different constants: Temporal turns step transitions into event-history appends against its own persistence (sized and scaled independently of your services), and Step Functions Standard bills per state transition, which

converts the write-amplification factor directly into dollars. Neither removes the cost; both move it somewhere with better tooling. The saga pattern's bookkeeping is never free — the engineering choice is only where the bill lands and who operates the meter.

Finally, resist the temptation to optimize the bookkeeping away. Teams under write pressure sometimes drop the step log ("we can infer progress from the domain tables") or skip the inbox ("duplicates are rare"). Both cuts trade a visible, budgetable cost for an invisible one that comes due during an incident: without the step log, recovery becomes archaeology across four services' tables; without the inbox, the duplicate that was "rare" arrives during the retry storm that follows every outage, exactly when it does the most damage. If the plumbing is too expensive at your scale, shard it or batch it — the two honest levers — rather than removing the parts whose entire purpose is being there on the worst day of the quarter.

Common mistakes

Compensations that can fail with no plan B. A compensation must be idempotent and retriable. If a refund can fail permanently, you need a retry queue and an alert for manual intervention, not a swallowed exception.

Steps without idempotency. Brokers deliver at least once. Without idempotency keys, a redelivery double-charges. Every step and every compensation must be safe to run twice.

Deleting data when compensating. Cancelling an order means marking it CANCELLED, not running DELETE. Deleting destroys traceability and breaks any consumer that already read the data.

An eight-step choreography. If nobody can draw the flow from memory, the pattern stopped helping. Migrate to orchestration before an event cycle forces you to.

Saga state held only in memory. A deploy mid-saga must never mean an order that was charged and never confirmed.

No timeouts. Every step that waits for a response needs a timeout that triggers compensation or retry. A saga waiting forever is a bug, not a valid state.

Production checklist

- Every step and every compensation is idempotent (idempotency key or prior-state check).
- The pivot transaction is identified, and everything after it is retriable.
- Saga state is persisted on every transition (or managed by a durable engine).
- Events are published via the outbox, never with a dual write.
- Intermediate states (PENDING_*) are visible in dashboards with staleness alerts: an old unfinished saga is an incident.
- Compensations retry with backoff and have a manual escalation path.
- Tests kill the process at every intermediate step and verify the saga recovers or compensates.

Frequently asked questions

Do I need a saga for every cross-service call? No. A saga is for multi-step *write* flows that must end consistent. A read composed from three services is not a saga; neither is a single write followed by best-effort notifications. If losing the follow-up action is acceptable (an analytics event, a cache warm-up), a plain retry queue is enough. Reach for a saga when a partial failure would leave money, stock, or legal state wrong.

Choreography or orchestration — is there a hard rule? A workable one: count the steps and the teams. Up to three steps owned by one or two teams, choreography stays readable and maximally decoupled. More steps, more teams, or any conversation that starts with "wait, what actually happens when...?" — orchestrate. Migrating later is normal: many systems start choreographed and grow an orchestrator only for their most complex flow, keeping events for the simple ones.

Where does the isolation problem actually bite? Wherever another flow reads intermediate state and acts on it. The classic: a reporting job counts an order that is later compensated, or a customer-facing screen shows "confirmed" for an order still in `PENDING_PAYMENT`. The fix is discipline, not machinery — intermediate states are real states of your business model, so name them, render them honestly in UIs ("processing"), and exclude them from reports until terminal.

Can a compensation itself be compensated? Avoid designing that. Compensations must be terminal: idempotent, retriable, and always able to succeed eventually (possibly via manual escalation). A compensation that needs its own undo means the original step boundary was drawn wrong — usually two business actions hiding inside one step. Split them.

How do I convince my team we need this? Run one experiment: kill the process between two "linked" writes in staging and show what the database looks like afterwards. The pattern is not an architectural fashion; it is the honest answer to a failure mode every distributed system already has. The alternative to a designed saga is an accidental one, compensated by hand, in production, with a spreadsheet.

Does event sourcing replace sagas? They are orthogonal. Event sourcing changes how one service stores its state (as a log of events); sagas coordinate state changes *across* services. An event-sourced service participates in a saga like any other — its local transaction appends events instead of updating rows. You can use either, both, or neither.

How do sagas interact with retries and backoff at the HTTP layer? They stack, and the layering matters. Per-request retries with exponential backoff and jitter (covered in this series' retries guide) live *inside* a step and handle transient noise; the saga's failure path handles definitive outcomes. Configure the inner layer to give up quickly — a step that retries for ten minutes before reporting failure blocks the whole saga's decision for ten minutes. A good split: the step retries for seconds, the saga's own retry policy (for retriable steps) spans minutes, and the reaper bounds the total in hours.

What belongs in the saga context, and what stays out? Everything a later step or compensation needs: identifiers (order id, charge id, reservation id), amounts as committed, and the error that triggered compensation. What stays out: anything you can re-read fresh (current stock levels), anything large (attach a document id, not the document), and anything sensitive that outlives its need — the context column is durable and shows up in step logs, dashboards, and audits, so card numbers and personal data have no business there. Treat the context as a public record of the flow, because operationally that is exactly what it becomes.

Is there a natural first saga to build? Pick a flow with two or three steps, a clear pivot, and compensations that are obvious business operations you already support (cancel, release, refund). The order flow is the classic for a reason. Avoid starting with your most complex flow — the machinery (state, recovery, reaper, dashboards) is the actual deliverable of the first saga, and it is much easier to harden on a simple flow and then reuse everywhere.

Conclusion

The saga pattern is the pragmatic answer to an uncomfortable fact: in microservices there are no global transactions. In exchange for giving up cross-service atomicity you gain availability and decoupling, paying with explicit complexity: compensations, idempotency, and intermediate states that are now part of your business model. Start with choreography when the flow is short and linear; move to orchestration as it grows. And whatever you do, publish events through an outbox and design every compensation as if it will run twice on a Friday night — because sooner or later, it will.