Dead Letter Queues: Manejo de Mensajes Envenenados en SQS, RabbitMQ y Kafka

Guia - Proyectos Reales - Edicion ampliada (ES + EN)

PuigFlows - puigflows.com - Julio 2026

Version en Espanol

El problema: un mensaje que nunca se procesa

Imagina un consumidor que lee de una cola y se encuentra un mensaje con JSON corrupto. Lanza una excepción, no confirma el mensaje y el broker lo vuelve a entregar. Otra excepción, otra reentrega. Ese único mensaje —un mensaje envenenado (poison message)— puede quedarse en bucle indefinidamente: quema CPU, ensucia los logs y, en sistemas con orden garantizado como Kafka, bloquea la partición entera, de modo que ningún mensaje posterior avanza hasta que ese se resuelva.

La solución estándar es la Dead Letter Queue (DLQ): una cola aparte a la que se mueven los mensajes que agotan sus intentos de procesamiento. El pipeline sigue fluyendo y tú ganas un lugar donde inspeccionar, corregir y reprocesar los fallos con calma, sin perder datos.

Transitorio o permanente: la decisión que lo cambia todo

Antes de configurar nada, clasifica los errores. No todos merecen reintento:

- Errores transitorios (timeouts, HTTP 503, deadlocks, throttling): suelen resolverse solos. Reintenta con backoff exponencial y jitter, entre 3 y 5 intentos como punto de partida.
- Errores permanentes (JSON inválido, violación de esquema, regla de negocio incumplida): reintentar es inútil. El mensaje debe ir a la DLQ en el primer intento.

El fallo clásico es tratar todo igual. Si reintentas cinco veces un error de validación, solo retrasas lo inevitable y añades carga al sistema justo cuando menos lo necesita. Y al revés: mandar un timeout puntual a la DLQ convierte un parpadeo de red en trabajo manual para un humano.

SQS: maxReceiveCount y redrive policy

SQS trae el soporte más directo. Cada cola puede declarar una redrive policy: si un mensaje se recibe más de maxReceiveCount veces sin ser borrado, SQS lo mueve automáticamente a la DLQ configurada.

```
import json
import boto3

sqs = boto3.client("sqs")

# 1. La DLQ, con la retencion maxima (14 dias) para dar tiempo a investigar
dlq = sqs.create_queue(
    QueueName="pedidos-dlq",
    Attributes={"MessageRetentionPeriod": "1209600"},
)
dlq_arn = sqs.get_queue_attributes(
    QueueUrl=dlq["QueueUrl"], AttributeNames=["QueueArn"]
)["Attributes"]["QueueArn"]

# 2. La cola principal: tras 5 recepciones sin borrado, SQS mueve el mensaje a la DLQ
sqs.create_queue(
    QueueName="pedidos",
    Attributes={
        "RedrivePolicy": json.dumps({
            "deadLetterTargetArn": dlq_arn,
            "maxReceiveCount": "5",
        }),
        "VisibilityTimeout": "30",
    },
)
```

Dos detalles que evitan sustos: la retención de la DLQ empieza a contar desde que el mensaje llegó a SQS, no desde que entró en la DLQ, así que dale siempre el máximo. Y cuando toque reprocesar, SQS ofrece redrive to source: devuelve los mensajes de la DLQ a su cola de origen con una sola llamada, con límite de velocidad configurable.

SQS a fondo: Lambda, lotes parciales y redrive programático

La redrive policy básica funciona igual cuando el consumidor es una función Lambda, pero ahí aparecen matices que producen dead-lettering accidental si no los conoces. Merece la pena verlos con calma, porque la combinación SQS + Lambda es hoy la forma más común de consumir colas en AWS.

Lotes parciales: no castigues a los mensajes inocentes

Lambda lee de SQS en lotes de hasta 10 mensajes por defecto. Si tu función lanza una excepción, el lote entero vuelve a la cola y el ReceiveCount de todos sus mensajes sube, incluidos los que se habían procesado bien. Con un maxReceiveCount bajo, un solo mensaje envenenado puede arrastrar a la DLQ a mensajes perfectamente válidos que tuvieron la mala suerte de compartir lote con él varias veces seguidas.

La solución son las respuestas parciales de lote (partial batch responses): en lugar de fallar todo o nada, informas a Lambda de exactamente qué mensajes fallaron, y solo esos vuelven a la cola.

```
import json

def handler(event, context):
    batch_item_failures = []
    for record in event["Records"]:
        try:
            process(json.loads(record["body"]))
        except Exception:
            batch_item_failures.append(
                {"itemIdentifier": record["messageId"]}
            )
    # Solo los fallidos vuelven a la cola; el resto se elimina
    return {"batchItemFailures": batch_item_failures}
```

Para que Lambda haga caso al valor de retorno tienes que activar ReportBatchItemFailures en el event source mapping; sin eso, la respuesta se ignora en silencio y sigues con el comportamiento todo-o-nada:

```
aws lambda update-event-source-mapping \
  --uuid <uuid-del-mapping> \
  --function-response-types "ReportBatchItemFailures"
```

La aritmética del visibility timeout

El segundo origen de dead-lettering fantasma es un visibility timeout demasiado corto. Si el mensaje vuelve a ser visible mientras Lambda todavía lo está procesando, otro worker lo recibe, el ReceiveCount sube sin que haya habido ningún fallo real, y el mensaje acaba en la DLQ habiéndose procesado bien (quizá varias veces). La recomendación de AWS es concreta: visibility timeout de al menos 6 veces el timeout de la función (más el batch window si lo usas). Y con Lambda como consumidor, configura maxReceiveCount en al menos 5: un throttling por concurrencia agotada también consume un intento, aunque tu código nunca llegara a ejecutarse.

Redrive programático con la API

El botón de la consola está bien para un incidente puntual, pero desde que existe la API de redrive puedes automatizar el reproceso con control de velocidad:

```
resp = sqs.start_message_move_task(
```

```
    SourceArn="arn:aws:sqs:eu-west-1:123456789012:pedidos-dlq",
    # Sin DestinationArn: cada mensaje vuelve a su cola de origen
    MaxNumberOfMessagesPerSecond=10,
)

# Supervisar el avance o abortar si algo va mal
sqs.list_message_move_tasks(
    SourceArn="arn:aws:sqs:eu-west-1:123456789012:pedidos-dlq"
)
sqs.cancel_message_move_task(TaskHandle=resp["TaskHandle"])
```

Dos detalles que ahorran sustos. Primero: el redrive "a origen" solo funciona con mensajes que llegaron a la DLQ mediante la redrive policy; si tu código publicó el mensaje en la DLQ a mano, SQS no sabe de dónde vino y tendrás que indicar un destino explícito. Segundo: MaxNumberOfMessagesPerSecond es tu freno de mano. Si el bug que llenó la DLQ sigue vivo, prefieres descubrirlo devolviendo 10 mensajes por segundo y no 10.000 de golpe.

DLQs en colas FIFO

La DLQ de una cola FIFO debe ser también FIFO. Y hay una interacción sutil con los message groups: mientras un mensaje de un grupo está atascado reintentándose, el resto del grupo espera detrás. Es el mismo efecto de bloqueo de partición que vimos en Kafka, y la conclusión es la misma: en colas FIFO conviene un maxReceiveCount bajo, porque cada reintento del mensaje envenenado retrasa a todos los mensajes de su grupo. Al moverlo a la DLQ desbloqueas el grupo, aunque pagues el precio de procesar los siguientes sin él: si tu dominio no tolera ese hueco, tu consumidor debe detectar el grupo incompleto y esperar.

RabbitMQ: DLX, ciclo de retry y la cabecera x-death

RabbitMQ no cuenta entregas en las colas clásicas. En su lugar, una cola declara un dead letter exchange (DLX): cuando un mensaje se rechaza con requeue=False, expira por TTL o la cola se desborda, RabbitMQ lo publica en ese exchange. Para emular los reintentos con espera se usa el patrón del ciclo de retry: la cola principal manda los rechazos a una cola de retry con TTL, y al expirar el TTL el mensaje vuelve a la principal. Cada vuelta incrementa la cabecera x-death, que hace de contador.

```
import pika

conn = pika.BlockingConnection(pika.ConnectionParameters("localhost"))
ch = conn.channel()

# Topologia: pedidos -(reject)-> retry (TTL 30s) -> vuelve a pedidos
# Tras MAX_ATTEMPTS ciclos, el consumidor publica en pedidos.dlq (terminal)
ch.exchange_declare("pedidos.retry.dlx", "fanout", durable=True)
ch.exchange_declare("pedidos.rework", "fanout", durable=True)

ch.queue_declare("pedidos", durable=True, arguments={
    "x-dead-letter-exchange": "pedidos.retry.dlx",
})
ch.queue_declare("pedidos.retry", durable=True, arguments={
    "x-message-ttl": 30000,                    # espera 30 s
    "x-dead-letter-exchange": "pedidos.rework",  # y reencola
})
ch.queue_declare("pedidos.dlq", durable=True)    # terminal: sin DLX

ch.queue_bind("pedidos.retry", "pedidos.retry.dlx")
ch.queue_bind("pedidos", "pedidos.rework")

MAX_ATTEMPTS = 5

def attempts_so_far(props):
    for death in (props.headers or {}).get("x-death", []):
        if death["queue"] == "pedidos":
            return int(death["count"])
```

```
    return 0

def publish_to_dlq(ch, body, props, exc):
    headers = dict(props.headers or {})
    headers["x-error"] = type(exc).__name__
    headers["x-error-detail"] = str(exc)[:500]
    ch.basic_publish("", "pedidos.dlq", body,
                     pika.BasicProperties(headers=headers, delivery_mode=2))

def on_message(ch, method, props, body):
    try:
        process(body)
        ch.basic_ack(method.delivery_tag)
    except PermanentError as exc:
        publish_to_dlq(ch, body, props, exc)   # sin reintentos: es inutil
        ch.basic_ack(method.delivery_tag)
    except TransientError as exc:
        if attempts_so_far(props) >= MAX_ATTEMPTS:
            publish_to_dlq(ch, body, props, exc)
            ch.basic_ack(method.delivery_tag)
        else:
            ch.basic_reject(method.delivery_tag, requeue=False)  # al ciclo de retry
```

Si usas quorum queues (el tipo recomendado hoy), hay un atajo: el argumento delivery-limit hace que la propia cola cuente las entregas y dead-letter-ee el mensaje al superar el límite, sin inspeccionar x-death a mano.

Backoff por mensaje en SQS: ChangeMessageVisibility como temporizador

SQS no tiene retry topics ni retrasos exponenciales nativos entre reintentos: si el mensaje vuelve a la cola, vuelve a ser visible tras el visibility timeout fijo, siempre el mismo. Pero hay una palanca poco conocida para conseguir backoff exponencial real por mensaje: ChangeMessageVisibility. En lugar de dejar que el timeout expire solo, el consumidor lo ajusta explícitamente según el número de intento antes de soltar el mensaje.

```
def handle(message):
    receive_count = int(
        message["Attributes"]["ApproximateReceiveCount"]
    )
    try:
        process(json.loads(message["Body"]))
        sqs.delete_message(
            QueueUrl=QUEUE_URL,
            ReceiptHandle=message["ReceiptHandle"],
        )
    except TransientError:
        # Backoff exponencial con tope: 30s, 60s, 120s... max 15 min
        delay = min(30 * (2 ** (receive_count - 1)), 900)
        sqs.change_message_visibility(
            QueueUrl=QUEUE_URL,
            ReceiptHandle=message["ReceiptHandle"],
            VisibilityTimeout=delay,
        )
        # No borramos: el mensaje reaparecera tras el delay
    except PermanentError:
        move_to_dlq(message)   # publicar en DLQ + borrar de la cola
        sqs.delete_message(
            QueueUrl=QUEUE_URL,
            ReceiptHandle=message["ReceiptHandle"],
        )
```

Tres observaciones. Primera: ApproximateReceiveCount es tu contador de intentos gratuito; pídelo en ReceiveMessage con AttributeNames. Segunda: el backoff convive con la redrive policy — maxReceiveCount sigue contando recepciones, así que dimensiona ambos juntos (con 5 recepciones y esta escala, el mensaje vive unos 4 minutos de espera acumulada antes de la DLQ; sube maxReceiveCount si necesitas ventanas de recuperación más largas). Tercera: para errores permanentes el patrón correcto sigue

siendo el atajo — publicar en la DLQ a mano y borrar, en vez de quemar recepciones. Eso sí: un mensaje movido a mano no podrá usar el redrive "a origen" de la consola, así que guarda la URL de la cola origen en un atributo del mensaje.

SNS y EventBridge: dónde vive la DLQ en un fan-out

En arquitecturas de eventos sobre AWS es raro que SQS esté solo: lo habitual es SNS o EventBridge repartiendo eventos hacia varias colas. Y ahí surge una pregunta de diseño con una respuesta poco intuitiva: ¿dónde pones la DLQ?

La respuesta corta: en cada suscripción, no en el publicador. Si SNS no consigue entregar a una de las cinco colas suscritas (porque alguien borró la cola, o la policy no permite la entrega), ese fallo es de esa suscripción concreta; las otras cuatro entregas fueron bien. Por eso tanto SNS como EventBridge permiten configurar una DLQ por suscripción o por regla/destino:

```
sns.set_subscription_attributes(
    SubscriptionArn=SUBSCRIPTION_ARN,
    AttributeName="RedrivePolicy",
    AttributeValue=json.dumps({
        "deadLetterTargetArn":
            "arn:aws:sqs:eu-west-1:123456789012:sns-entregas-dlq"
    }),
)
```

Ojo a la diferencia de semántica, porque confundirla cuesta datos:

- La DLQ de la suscripción SNS/regla de EventBridge captura fallos de entrega: el mensaje nunca llegó a la cola de destino. Sin ella, tras agotar los reintentos de entrega el evento se pierde para siempre, y nadie lo nota porque ningún consumidor llegó a fallar.
- La DLQ de la cola SQS (la redrive policy de siempre) captura fallos de procesamiento: el mensaje llegó, pero tu consumidor no pudo con él.

Necesitas ambas capas. EventBridge además reintenta entregas durante hasta 24 horas con backoff antes de rendirse; si tu destino estuvo caído más tiempo (un despliegue desastroso de fin de semana), solo la DLQ de la regla te permite recuperar esos eventos. La lección general para cualquier broker con fan-out: cada salto de la cadena necesita su propio plan para el fallo, porque cada salto puede fallar por razones distintas.

RabbitMQ moderno: colas quorum, delivery-limit y dead-lettering fiable

Todo lo anterior sobre DLX aplica a las colas clásicas, pero desde hace años las colas quorum son la opción recomendada para cargas importantes, y RabbitMQ 4 trajo un cambio que sorprendió a más de un equipo al actualizar: el delivery-limit ahora tiene un valor por defecto de 20.

Las colas quorum llevan la cuenta de reentregas en la cabecera x-delivery-count, bastante más fácil de leer que el histórico x-death de las clásicas. Cuando el contador supera el delivery-limit, el mensaje se dead-letterea si hay DLX configurado… o se descarta si no lo hay. Sí: se descarta. Si migraste a RabbitMQ 4 sin configurar dead-letter-exchange en tus colas quorum, un mensaje envenenado desaparece sin dejar rastro al vigésimo intento. Configúralo siempre, aunque solo sea para poder hacer la autopsia:

```
rabbitmqctl set_policy dlq-pedidos "^pedidos" \
  '{"delivery-limit": 5,
    "dead-letter-exchange": "dlx",
    "dead-letter-strategy": "at-least-once",
    "overflow": "reject-publish"}' \
  --apply-to quorum_queues
```

Dead-lettering at-least-once

El dead-lettering clásico es at-most-once: el broker re-publica el mensaje hacia el DLX sin esperar confirmación, y si en ese instante no hay cola enlazada o el nodo se cae, el mensaje se pierde. Para un log de clicks puede dar igual; para un pago, no. Las colas quorum ofrecen dead-letter-strategy: at-least-once, que re-publica con confirms internos y no suelta el mensaje hasta que la cola destino confirma haberlo persistido.

Los requisitos: la cola origen debe ser quorum, overflow debe ser reject-publish y la cola destino debería ser también quorum. El coste es memoria (los mensajes pendientes de confirmación se retienen), pero es el precio de poder afirmar que ningún mensaje muerto se pierde por el camino.

Un matiz reciente que corrige una injusticia histórica: desde RabbitMQ 4.3 se distinguen dos contadores, delivery-count (solo sube cuando la reentrega se debe a un fallo real del consumidor) y acquired-count (sube con cualquier requeue). Antes, un consumidor que devolvía mensajes por un rebalanceo o un apagado limpio inflaba el contador y podía mandar a la DLQ mensajes perfectamente sanos.

En el consumidor, usa x-delivery-count para decidir sin esperar al límite del broker cuando el error es claramente permanente:

```
def on_message(ch, method, properties, body):
    deliveries = (properties.headers or {}).get("x-delivery-count", 0)
    try:
        process(json.loads(body))
        ch.basic_ack(method.delivery_tag)
    except PermanentError:
        # No tiene sentido esperar 5 intentos: al DLX directamente
        ch.basic_nack(method.delivery_tag, requeue=False)
    except Exception:
        if deliveries >= 4:
            ch.basic_nack(method.delivery_tag, requeue=False)
        else:
            ch.basic_nack(method.delivery_tag, requeue=True)
```

El broker actúa como red de seguridad con su delivery-limit, y tu código toma la vía rápida para los errores que reconoce como permanentes. Ambas capas suman.

Kafka: la DLQ que construyes tú

Kafka no tiene DLQ nativa: un consumidor que falla y no avanza el offset relee el mismo registro para siempre y bloquea la partición. El patrón es construirla en el consumidor: capturar el error, publicar el registro en un topic *.dlq con cabeceras de diagnóstico y confirmar el offset igualmente para desbloquear la partición.

```
import json
import time
import traceback
from confluent_kafka import Consumer, Producer

consumer = Consumer({
    "bootstrap.servers": "localhost:9092",
    "group.id": "pedidos-worker",
    "enable.auto.commit": False,   # commit solo tras decidir que hacer
    "auto.offset.reset": "earliest",
})
producer = Producer({"bootstrap.servers": "localhost:9092"})
consumer.subscribe(["pedidos"])

MAX_ATTEMPTS = 4

def send_to_dlq(msg, exc, attempts):
    origin = "pedidos/" + str(msg.partition()) + "/" + str(msg.offset())
    producer.produce(
        "pedidos.dlq",
        key=msg.key(),
        value=msg.value(),          # payload original, sin tocar
        headers={
            "dlq.error": type(exc).__name__,
```

```
            "dlq.detail": str(exc)[:500],
            "dlq.stack": traceback.format_exc()[-2000:],
            "dlq.origin": origin,
            "dlq.attempts": str(attempts),
            "dlq.ts": str(int(time.time() * 1000)),
        },
    )
    producer.flush()

while True:
    msg = consumer.poll(1.0)
    if msg is None or msg.error():
        continue
    for attempt in range(1, MAX_ATTEMPTS + 1):
        try:
            process(json.loads(msg.value()))
            break
        except PermanentError as exc:
            send_to_dlq(msg, exc, attempt)   # directo, sin reintentos
            break
        except TransientError as exc:
            if attempt == MAX_ATTEMPTS:
                send_to_dlq(msg, exc, attempt)
            else:
                time.sleep(min(2 ** attempt, 30))  # backoff en el proceso
    consumer.commit(msg)   # desbloquea la particion pase lo que pase
```

Ojo con el time.sleep: si el backoff total supera max.poll.interval.ms, el broker expulsa al consumidor del grupo y provoca un rebalanceo. Para esperas largas, usa topics de retry escalonados (por ejemplo pedidos.retry.1m, pedidos.retry.10m) en lugar de dormir en el proceso. Si usas Spring Kafka, DeadLetterPublishingRecoverer implementa todo este patrón de serie.

Kafka avanzado: retry topics escalonados y reintentos no bloqueantes

El consumidor con bucle de reintentos que vimos antes tiene una limitación seria: mientras reintenta, la partición entera está bloqueada. Con backoffs de milisegundos quizá puedas permitírtelo; con backoffs de minutos —los que de verdad ayudan cuando una dependencia está caída— no. El patrón estándar para desbloquear la partición son los retry topics escalonados.

La idea: además del topic principal y la DLQ, creas topics intermedios con retrasos crecientes, por ejemplo pedidos.retry.1m y pedidos.retry.10m. Cuando un mensaje falla en el principal, se publica en el primer retry topic con una marca de tiempo de "no procesar antes de". El consumidor de cada retry topic comprueba esa marca: si aún no toca, pausa la partición y espera; si ya toca, procesa. Si el mensaje agota todos los niveles, acaba en la DLQ. La partición del topic principal nunca se detiene.

```
RETRY_TOPICS = ["pedidos.retry.1m", "pedidos.retry.10m"]
DELAYS = {"pedidos.retry.1m": 60, "pedidos.retry.10m": 600}

def next_destination(topic):
    if topic == "pedidos":
        return RETRY_TOPICS[0]
    idx = RETRY_TOPICS.index(topic)
    if idx + 1 < len(RETRY_TOPICS):
        return RETRY_TOPICS[idx + 1]
    return "pedidos.dlq"

def forward(msg, exc, source_topic):
    dest = next_destination(source_topic)
    headers = dict(msg.headers() or [])
    attempts = int(headers.get("retry.attempts", b"0")) + 1
    not_before_ms = int(time.time() * 1000) + DELAYS.get(dest, 0) * 1000
    producer.produce(
        dest,
        key=msg.key(),
        value=msg.value(),           # payload original, intacto
        headers={
```

```
            "retry.attempts": str(attempts),
            "retry.not.before": str(not_before_ms),
            "dlq.error": type(exc).__name__,
            "dlq.detail": str(exc)[:500],
        },
    )
    producer.flush()

def run_retry_consumer(topic):
    consumer.subscribe([topic])
    while True:
        msg = consumer.poll(1.0)
        if msg is None or msg.error():
            continue
        headers = dict(msg.headers() or [])
        not_before = int(headers.get("retry.not.before", b"0"))
        if time.time() * 1000 < not_before:
            # Aun no toca: pausa, rebobina al mismo offset y espera
            tp = TopicPartition(msg.topic(), msg.partition(), msg.offset())
            consumer.pause([tp])
            consumer.seek(tp)
            wait_s = (not_before - time.time() * 1000) / 1000
            time.sleep(min(wait_s, 30))
            consumer.resume([tp])
            continue
        try:
            process(json.loads(msg.value()))
        except Exception as exc:
            forward(msg, exc, topic)
        consumer.commit(msg)
```

La advertencia que nadie puede ahorrarte: en el momento en que un mensaje salta a un retry topic, pierdes el orden por clave. Un evento posterior de la misma entidad puede procesarse antes que el que está esperando su reintento. Si el orden importa, tienes dos salidas: reintentar bloqueando (aceptando la latencia en esa partición) o detectar en el consumidor que hay un mensaje anterior de esa clave pendiente en retry y apartar también el nuevo. Elegir retry topics es elegir disponibilidad sobre orden; hazlo con los ojos abiertos.

En el ecosistema JVM no hace falta construir nada de esto a mano: Spring Kafka lo trae de serie con la anotación @RetryableTopic, que crea los retry topics y el dead letter topic automáticamente y gestiona la espera con pausas de partición, sin bloquear el hilo:

```
@RetryableTopic(
    attempts = "4",
    backoff = @Backoff(delay = 60000, multiplier = 10.0)
)
@KafkaListener(topics = "pedidos")
public void listen(Pedido pedido) {
    procesar(pedido);
}

@DltHandler
public void dlt(Pedido pedido,
                @Header(KafkaHeaders.ORIGINAL_TOPIC) String origen) {
    log.error("Mensaje muerto procedente de {}", origen);
}
```

Y si quien procesa es un conector de Kafka Connect (un sink hacia una base de datos, por ejemplo), no escribas código: es configuración. Con errors.tolerance el conector deja de morirse ante el primer registro malo y lo desvía a su propia DLQ con el contexto del error en cabeceras:

```
{
  "errors.tolerance": "all",
  "errors.deadletterqueue.topic.name": "connect.pedidos.dlq",
  "errors.deadletterqueue.topic.replication.factor": 3,
  "errors.deadletterqueue.context.headers.enable": true,
  "errors.retry.timeout": 300000,
  "errors.log.enable": true
}
```

Con context.headers.enable activado, Connect añade cabeceras __connect.errors.* (topic y offset de origen,

clase de excepción, mensaje) a cada registro desviado: exactamente los metadatos que dijimos que toda DLQ necesita, gratis.

Elegir plataforma: qué te da y qué te cobra cada broker

Si estás decidiendo dónde vivirá un sistema nuevo, la historia del dead-lettering resume bien la filosofía de cada broker:

- SQS es el más completo de serie: redrive policy declarativa, contador de recepciones gestionado, API de redrive con control de velocidad y métricas en CloudWatch sin instalar nada. A cambio, los reintentos con retraso variable exigen el truco de ChangeMessageVisibility y el máximo de retención son 14 días: si tu proceso de revisión es lento, el reloj corre.
- RabbitMQ da la topología más expresiva: DLX enrutando por routing key, políticas por patrón de colas, y con quorum queues un contador fiable (x-delivery-count) y dead-lettering at-least-once. Lo que no da es herramienta de reproceso: el redrive lo escribes tú. Y exige vigilar la configuración tras cada upgrade mayor, como demostró el delivery-limit por defecto de la versión 4.
- Kafka no te da nada… y te da todo: no hay DLQ nativa, pero la retención larga, los offsets re-consumibles y las cabeceras hacen que construir DLQs y retry topics a medida sea natural. Es la opción con más piezas a tu cargo y también la única donde "reprocesar los muertos de hace tres semanas" es trivial si dimensionaste la retención.

La regla práctica: en AWS gestionado, empieza con SQS y no inventes; si ya operas RabbitMQ, usa quorum queues y póliza con DLX desde el día uno; en Kafka, acepta que la DLQ es código tuyo y trátala como tal, con tests y dueño.

Deserialización y esquemas: la fábrica número uno de mensajes envenenados

La mayoría de los mensajes envenenados no nacen de bugs exóticos. Nacen de algo mucho más mundano: un productor que cambió el formato. Un campo renombrado, un tipo que pasó de entero a string, un JSON truncado por un límite de tamaño. El fallo ocurre antes de tu lógica de negocio, en la capa de deserialización, y eso importa porque muchos frameworks reintentan por defecto errores que jamás se arreglarán solos: un JSON corrupto seguirá corrupto en el intento mil.

- Trata los errores de deserialización como permanentes, siempre. Directo a la DLQ, sin reintentos. No hay dependencia que recuperar ni timeout que esperar.
- Deserializa y valida antes de tocar nada. Si el mensaje va a fallar, que falle antes de producir efectos secundarios a medias.
- Guarda los bytes crudos en la DLQ, no tu interpretación fallida de ellos. El payload original es la única evidencia forense fiable.

Con Pydantic, la frontera queda en una función de una línea de lógica:

```
from pydantic import BaseModel, ValidationError

class Pedido(BaseModel):
    pedido_id: str
    importe_centavos: int
    moneda: str

def parse(raw: bytes) -> Pedido:
    try:
        return Pedido.model_validate_json(raw)
    except (ValidationError, UnicodeDecodeError) as exc:
        raise PermanentError("payload invalido") from exc
```

Un schema registry (Avro, Protobuf o JSON Schema) cambia la forma del problema: los chequeos de

compatibilidad frenan la mayoría de los cambios rompedores en el lado del productor, antes de que lleguen a la cola. Pero "compatible" según el registro no siempre significa "compatible con tu código": un campo nuevo opcional que tu lógica asume presente pasa el chequeo del registro y revienta en tu consumidor. El registro reduce la superficie del problema; los tests de contrato la cierran.

Un consejo operativo que paga dividendos: distingue en los metadatos de la DLQ los fallos de deserialización de los fallos de negocio. Los primeros se resuelven con un despliegue (del consumidor o rollback del productor) y se reprocesan en bloque con confianza; los segundos exigen mirar mensaje a mensaje. Cuando la DLQ mezcla ambos sin etiquetar, cada incidente empieza con una hora de arqueología.

Qué guardar junto al mensaje fallido

Un mensaje en la DLQ sin contexto obliga a arqueología forense. Adjunta siempre, en cabeceras o en un sobre de metadatos: el tipo y mensaje de la excepción, el stack trace, el número de intentos, el origen exacto (cola, o topic/partición/offset), la versión del consumidor que falló y el ID de correlación de la petición. Guarda el payload original intacto: si lo transformas antes de encolarlo, ya no podrás reproducir el fallo tal cual ocurrió.

Diseño del sobre del mensaje: versión, identidad y trazas

Buena parte de lo que hace manejable una DLQ se decide mucho antes del primer fallo: en el diseño del propio mensaje. Un payload desnudo — un JSON con campos de negocio y nada más — es barato de producir y carísimo de operar. Un sobre (envelope) mínimo alrededor cambia las reglas del juego:

```
{
  "message_id": "9c2f4a1e-8d3b-4f6a-9e21-7b5c0d8e4f12",
  "type": "pedido.creado",
  "version": 2,
  "occurred_at": "2026-07-28T09:14:03Z",
  "traceparent": "00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01",
  "data": {
    "pedido_id": "P-48821",
    "importe_centavos": 4990,
    "moneda": "EUR"
  }
}
```

Cada campo del sobre resuelve un problema concreto de la DLQ:

- message_id estable y generado por el productor: es la clave de la tabla inbox y lo que permite afirmar "este mensaje ya se procesó" durante un redrive. Sin él, la idempotencia depende de heurísticas frágiles sobre el contenido.
- type y version: el consumidor puede enrutar y, sobre todo, rechazar limpiamente versiones que no entiende ("version 3: no soportada") en vez de fallar de forma críptica en mitad de la lógica. Y al mirar la DLQ, la distribución de versiones te dice al instante si el problema es una migración de esquema.
- occurred_at: no es el timestamp del broker, es el del hecho de negocio. Imprescindible al reprocesar con días de retraso, cuando "ahora" ya no significa nada.
- traceparent (el estándar W3C Trace Context): propagado hasta la DLQ y el reproceso, conecta el mensaje muerto con la traza distribuida original. El on-call puede saltar del mensaje en la DLQ a la traza exacta en la que murió, con todos los spans y logs correlacionados, en un clic.

Nada de esto es sofisticado; todo es disciplina. Los equipos que la tienen depuran una DLQ en minutos. Los que no, imprimen payloads en la terminal y hacen arqueología con grep.

Monitorización: la edad importa más que el tamaño

Una DLQ sin alertas es un cementerio: cada mensaje que entra y nadie mira es una pérdida de datos silenciosa. Dos reglas prácticas:

- Alerta sobre la edad del mensaje más antiguo (en SQS, ApproximateAgeOfOldestMessage), no solo sobre la profundidad. Diez mensajes de hace un minuto pueden ser ruido; uno de hace tres días es un problema que nadie ha atendido.
- Cualquier mensaje en la DLQ debería ser accionable: o revela un bug que corregir, o un productor que valida mal, o un contrato roto entre servicios. Si un tipo de mensaje llega de forma recurrente y decidís ignorarlo, filtradlo antes, no lo dejéis acumularse.

Y una regla de diseño: la DLQ es terminal. Sin DLX sobre la propia DLQ, sin redrive automático de la DLQ hacia otra DLQ. Los ciclos mensaje-va-y-vuelve son una de las formas más comunes de tirar un broker en producción.

Observabilidad aplicada: alarmas de CloudWatch y reglas de Prometheus

Ya establecimos el principio: la edad importa más que el tamaño. Bajemos a configuración concreta en las dos plataformas de monitorización más comunes.

CloudWatch para SQS

Dos alarmas bastan para cubrir el 95% de los incidentes. La primera salta con el primer mensaje que entra en la DLQ: una DLQ sana está vacía, así que el umbral correcto es cero.

```
cloudwatch.put_metric_alarm(
    AlarmName="pedidos-dlq-tiene-mensajes",
    Namespace="AWS/SQS",
    MetricName="ApproximateNumberOfMessagesVisible",
    Dimensions=[{"Name": "QueueName", "Value": "pedidos-dlq"}],
    Statistic="Maximum",
    Period=300,
    EvaluationPeriods=1,
    Threshold=0,
    ComparisonOperator="GreaterThanThreshold",
    AlarmActions=[SNS_TOPIC_ARN],
)
```

La segunda vigila ApproximateAgeOfOldestMessage, pero en la cola principal: si el mensaje más antiguo de la cola de trabajo envejece, tu consumidor está caído o atascado, y la DLQ ni siquiera llegará a enterarse.

Prometheus para RabbitMQ y Kafka

```
# Hay mensajes muertos sin revisar desde hace 15 minutos
- alert: DLQMessagesWaiting
  expr: rabbitmq_queue_messages{queue=~".*dlq.*"} > 0
  for: 15m
  labels:
    severity: warning

# Estan ENTRANDO mensajes a la DLQ ahora mismo
- alert: DLQGrowthRate
  expr: rate(rabbitmq_queue_messages_published_total{queue=~".*dlq.*"}[5m]) > 1
  labels:
    severity: critical
```

La distinción entre ambas es lo que evita la fatiga de alertas: "hay 3 mensajes de ayer pendientes de revisión" es un warning para horario laboral; "están entrando mensajes a más de uno por segundo" significa que algo está roto en producción ahora y debe despertar a alguien. En Kafka, las métricas equivalentes son el rate de producción al topic .dlq y el lag del consumer group que se supone que lo revisa: si ese lag crece

sin límite, tu DLQ es un cementerio, no una sala de espera.

Completa el cuadro con dos métricas de aplicación que ningún exporter te da gratis: un contador dlq_messages_total etiquetado por cola de origen y tipo de error (te dice qué se rompe y dónde sin abrir ni un mensaje), y un histograma de los intentos consumidos antes de morir. Este último es revelador: si la mayoría de tus mensajes muertos agotaron todos los reintentos, son errores transitorios que no se recuperaron a tiempo y quizá necesitas más margen; si murieron al primer intento, son permanentes y tus reintentos solo añaden ruido y latencia.

Reprocesamiento: deliberado, nunca a ciegas

El orden correcto es: diagnosticar, corregir el bug, desplegar y después redirigir los mensajes, en lotes acotados y vigilando métricas. El redrive masivo a ciegas devuelve los mismos venenos a la cola y duplica el incidente. Dos requisitos previos:

- Idempotencia: un mensaje que falló a mitad de procesamiento puede haber dejado efectos parciales. Reprocesarlo no debe duplicar pagos, pedidos ni correos.
- Orden: si tu dominio depende del orden (eventos de un mismo agregado), ten en cuenta que los mensajes reprocesados llegarán tarde y fuera de secuencia; el consumidor debe tolerarlo, por ejemplo con números de versión.

Idempotencia: la condición previa para reprocesar sin miedo

Reprocesar una DLQ significa, por definición, entregar otra vez mensajes que ya se intentaron. Si tu consumidor cobró la tarjeta y después falló al guardar en base de datos, el reproceso volverá a cobrarla. Antes de tocar el botón de redrive necesitas idempotencia real, y la forma más robusta es la tabla inbox: registrar cada message_id procesado en la misma transacción que el efecto de negocio.

```
CREATE TABLE inbox (
    message_id   text PRIMARY KEY,
    processed_at timestamptz NOT NULL DEFAULT now()
);
```

```
def process_idempotent(message_id: str, payload: dict):
    with db.begin() as tx:
        inserted = tx.execute(
            "INSERT INTO inbox (message_id) VALUES (%s) "
            "ON CONFLICT DO NOTHING",
            (message_id,),
        ).rowcount
        if inserted == 0:
            return  # ya procesado antes: ack y seguir
        apply_business_logic(tx, payload)  # misma transaccion
```

La clave está en la atomicidad: el registro en inbox y el efecto de negocio se confirman juntos o se revierten juntos. Con esto, el peor caso de cualquier reproceso —o de cualquier reentrega del broker, que en sistemas at-least-once ocurren solas— es un no-op silencioso. Es el mismo patrón que protege contra los duplicados del día a día; la DLQ solo lo vuelve obligatorio.

Un redrive casero con freno de mano

Para RabbitMQ y Kafka no hay API de redrive gestionada: la herramienta la escribes tú. Las tres propiedades que debe tener: límite de cantidad, límite de velocidad y aborto automático si los mensajes vuelven a fallar en masa (señal inequívoca de que el bug sigue vivo).

```
def redrive(dlq, destination, max_messages=100,
            rate_per_second=5, abort_after_failures=10):
```

```
    moved, failed = 0, 0
    for msg in dlq.read(limit=max_messages):
        if failed >= abort_after_failures:
            log.error("demasiados refallos: el bug sigue vivo, abortando")
            break
        try:
            destination.publish(msg.body, headers=msg.headers)
            dlq.ack(msg)
            moved += 1
        except Exception:
            failed += 1
        time.sleep(1.0 / rate_per_second)
    return moved, failed
```

Empieza siempre con una muestra pequeña (max_messages=10) y observa. Si la muestra pasa limpia, sube el volumen. Un redrive es un experimento controlado, no una compuerta que se abre.

Autoscaling y colas: KEDA y el efecto acordeón

Un detalle operativo que conecta las DLQs con la capacidad: si escalas tus consumidores según la profundidad de la cola (el patrón habitual con KEDA en Kubernetes), un mensaje envenenado sin DLQ no solo bloquea el procesamiento — infla la métrica de escalado. La cola crece porque nada avanza, el autoscaler añade réplicas, y ahora tienes veinte pods reintentando el mismo mensaje corrupto en lugar de cinco. Pagas más por fallar más rápido.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: pedidos-consumer
spec:
  scaleTargetRef:
    name: pedidos-consumer
  minReplicaCount: 1
  maxReplicaCount: 20
  triggers:
    - type: aws-sqs-queue
      metadata:
        queueURL: https://sqs.eu-west-1.amazonaws.com/123456789012/pedidos
        queueLength: "5"      # mensajes por replica deseados
        awsRegion: eu-west-1
```

Con la DLQ en su sitio, el mensaje envenenado sale del camino en unos pocos intentos, la profundidad refleja trabajo real y el escalado vuelve a tener sentido. Dos consejos: no incluyas la DLQ en la métrica de escalado del consumidor principal (su profundidad no es demanda, es deuda), y ponle a la DLQ su propio ScaledObject si su archivador necesita absorber picos — con maxReplicaCount bajo, porque archivar no es urgente.

Multi-tenant: cuando el veneno viene de un solo cliente

En sistemas multi-tenant aparece un modo de fallo propio: un solo cliente — una integración suya mal configurada, un formato exótico, un volumen anómalo — genera el 99% de los mensajes envenenados y su ruido entierra los fallos de los demás. Tres defensas escalonadas:

- Etiqueta el tenant en los metadatos del mensaje muerto desde el día uno. Es la dimensión que convierte "hay 4.000 mensajes en la DLQ" en "el tenant ACME tiene 3.960 y el resto se reparte 40", que es un incidente completamente distinto.
- Alerta por tenant, no solo global. Un umbral global alto para no despertar a nadie por un cliente ruidoso conocido, y umbrales por tenant para detectar que un cliente pequeño de repente falla al 100%, cosa que el agregado esconde.
- Si un tenant satura de forma crónica, sepáralo: su propia cola con su propia DLQ. El aislamiento por

colas es la versión de mensajería del patrón bulkhead, y evita que el ruido de uno degrade la operación de todos.

La versión resumida: una DLQ compartida entre tenants es aceptable; una DLQ donde no puedes filtrar por tenant es una trampa que se cierra el día que más la necesitas.

Testing: si no pruebas el camino del fallo, no existe

El camino feliz se prueba solo con usar la aplicación. El camino de la DLQ solo se ejecuta cuando algo se rompe, así que si no lo cubres con tests, su primera ejecución real será en producción, de madrugada y con público. Con Testcontainers (brokers reales en Docker) y LocalStack (SQS local) puedes cubrirlo en CI sin mocks que mientan.

```
import pytest
from testcontainers.rabbitmq import RabbitMqContainer

@pytest.fixture(scope="module")
def rabbit():
    with RabbitMqContainer("rabbitmq:4.1-management") as c:
        yield c

def test_envenenado_llega_a_dlq_con_metadatos(rabbit):
    setup_topology(rabbit)          # colas + DLX como en produccion
    publish(rabbit, "pedidos", b"esto no es json")
    run_consumer_until_idle(rabbit)

    dead = get_one(rabbit, "pedidos.dlq", timeout=10)
    assert dead.body == b"esto no es json"   # payload intacto
    assert dead.headers["dlq.error"] == "PermanentError"
    assert "dlq.stack" in dead.headers
```

Los cuatro tests que de verdad importan:

- Un error permanente va a la DLQ al primer intento, sin reintentos de por medio.
- Un error transitorio se reintenta N veces con backoff y solo entonces se rinde.
- El mensaje muerto conserva el payload original byte a byte y todos los metadatos.
- Reprocesar un mensaje ya procesado es un no-op (la idempotencia funciona).

Y uno más sutil que casi nadie escribe: mientras hay mensajes fallando, el consumidor sigue avanzando con el resto. Es la propiedad que justifica toda esta arquitectura; merece su assert.

Para SQS, LocalStack reproduce la redrive policy completa: crea en el test una cola con maxReceiveCount 2, publica un mensaje que siempre falla y afirma que aparece en la DLQ tras exactamente dos recepciones, ni una más.

El consumidor de la DLQ: archivar antes que expirar

Dijimos que la DLQ no debe tener su propia DLQ y que su consumidor debe ser trivial. Este es el patrón concreto: un archivador que copia cada mensaje muerto a almacenamiento barato y duradero (S3, GCS, un bucket con lifecycle) antes de que la retención lo borre. Con eso, la cola deja de ser el único custodio de los datos y la presión del reloj desaparece.

```
def archive_dlq_message(message):
    key = "dlq/pedidos/{}/{}.json".format(
        time.strftime("%Y/%m/%d"),
        message["MessageId"],
    )
    s3.put_object(
        Bucket="dlq-archive",
        Key=key,
        Body=json.dumps({
```

```
            "body": message["Body"],
            "attributes": message.get("MessageAttributes", {}),
            "receive_count":
                message["Attributes"]["ApproximateReceiveCount"],
            "archived_at": time.time(),
        }),
    )
    # Solo tras confirmar el archivado se borra de la cola
    sqs.delete_message(
        QueueUrl=DLQ_URL,
        ReceiptHandle=message["ReceiptHandle"],
    )
```

Detalles que importan: la clave particionada por fecha permite listar "todos los muertos del martes" sin escanear el bucket; se archiva el mensaje completo con metadatos, no solo el body; y el borrado de la cola ocurre después del put confirmado — el orden inverso pierde datos si el archivador muere a mitad. Sobre el archivo puedes montar lo que la cola nunca te dará: consultas con Athena, retención de años a coste de céntimos y reproceso selectivo por fecha o tipo de error.

Seguridad y retención: la DLQ también es datos de producción

Una DLQ acumula precisamente los mensajes que nadie está procesando, y eso la convierte en el rincón olvidado con datos reales: payloads con emails, direcciones, importes, a veces tokens. Tres reglas:

- Misma protección que la cola principal. Cifrado en reposo (SSE-KMS en SQS, TLS y cifrado de disco en RabbitMQ y Kafka) y control de acceso igual de estricto. Es sorprendentemente común encontrar la cola principal bien cerrada y la DLQ abierta "porque es de debugging".
- Retención explícita y finita. Los 14 días de SQS son el máximo disponible, no un objetivo. En Kafka, configura retention.ms en el topic .dlq; en RabbitMQ, un TTL en la cola muerta. Y alinea ese plazo con tu política de datos personales: el derecho de supresión del GDPR también aplica a los mensajes muertos, que son los que más fácilmente se quedan olvidados en una esquina.
- Cuidado con lo que copias a los metadatos. El stack trace que guardas en dlq.stack puede llevar el payload entero interpolado en el mensaje de la excepción. Redacta los campos sensibles antes de escribir los metadatos, igual que harías en los logs.

Errores comunes

- Confirmar antes de procesar: un consumidor que hace ack y luego lanza una excepción pierde el mensaje sin dejar rastro. El ack va siempre después del éxito (o de la decisión consciente de enviar a la DLQ).
- Reintentar errores permanentes: cinco intentos contra un JSON corrupto son cinco fracasos garantizados y latencia extra para todo lo demás.
- DLQ sin retención suficiente: si los mensajes caducan antes de que alguien los mire, la DLQ solo aplaza la pérdida de datos.
- Redrive a ciegas sin corregir la causa: los mensajes vuelven, fallan y regresan a la DLQ, ahora con las métricas duplicadas.
- Sin metadatos de error: obliga a reproducir el fallo para entenderlo, justo lo que la DLQ debía evitar.

Runbook: el día a día de operar una DLQ

La arquitectura es la mitad del trabajo; la otra mitad es qué hace un humano cuando suena la alarma. Este es el runbook que debería estar enlazado desde esa alerta:

- 1. Clasifica sin abrir mensajes. Mira los metadatos agregados: ¿un solo tipo de error o varios? ¿Una

sola cola de origen o muchas? ¿El inicio coincide con un despliegue, un pico de tráfico o una caída de un proveedor? El 80% de los diagnósticos se cierra aquí.

- 2. Decide si sigue entrando. Si el rate de entrada a la DLQ es cero, el incidente ya pasó y trabajas con calma. Si sigue entrando, el bug está vivo: tu prioridad es cortar la hemorragia (rollback, feature flag, hotfix), no reprocesar.
- 3. Corrige la causa antes de tocar la DLQ. Un redrive sin fix es un bumerán: los mensajes volverán, con sus contadores de reintento agotados y menos margen.
- 4. Reprocesa por lotes crecientes. Muestra de 10, observa, luego el resto con límite de velocidad. Vigila el rate de refallos durante todo el proceso.
- 5. Cierra con higiene. DLQ a cero, alarma recuperada, y una nota en el postmortem: qué tipo de error fue, cuántos mensajes, cuánto tardó cada fase. Esos datos calibran tus reintentos y tus umbrales para la próxima vez.

Y una tarea semanal de diez minutos que evita sustos: revisar las DLQs con mensajes antiguos sin reclamar. Si llevan ahí más de una semana, o se decide reprocesarlos o se decide descartarlos, pero se decide; el limbo indefinido es la puerta a perderlos por retención.

Caso práctico: la migración que llenó una DLQ a las 2:14

Para cerrar, un incidente real (con los nombres cambiados) que muestra todas las piezas funcionando juntas.

Un pipeline de pedidos con SQS y Lambda. El equipo productor despliega a las 2:03 una versión que renombra el campo importe a importe_centavos. La coordinación con el equipo consumidor existía… en un ticket planificado para la semana siguiente. A las 2:07 entran los primeros pedidos con el esquema nuevo; el consumidor los valida con Pydantic, la validación lanza error permanente y los mensajes van directos a la DLQ con sus metadatos.

- 2:14 — La alarma de umbral cero se dispara con el primer mensaje muerto. El on-call abre el dashboard.
- 2:19 — Sin abrir ni un mensaje, los metadatos cuentan la historia: dlq.error es ValidationError en el 100% de los casos, dlq.detail dice "importe: field required", y el inicio exacto coincide con el despliegue de las 2:03 del productor. Diagnóstico: cinco minutos.
- 2:31 — Decisión: en lugar de pedir rollback al otro equipo, el consumidor saca un hotfix que acepta ambos campos (el patrón expand de expand-contract). Mientras tanto, los pedidos con esquema viejo siguen fluyendo con normalidad: la DLQ está aislando el problema, no bloqueando el pipeline.
- 2:58 — Hotfix desplegado. Redrive con la API de SQS a 10 mensajes por segundo. Los 312 mensajes muertos se reprocesan sin un solo refallo; la tabla inbox garantiza que ninguno se aplica dos veces.
- 3:20 — DLQ a cero. Edad máxima que alcanzó el mensaje más antiguo: 51 minutos. Pedidos perdidos: cero. Clientes afectados: cero.

La moraleja no es que el incidente fuera elegante, sino todo lo que no pasó: nadie restauró backups, nadie escribió consultas SQL de madrugada para reconstruir pedidos, nadie descubrió el problema por un cliente enfadado a la mañana siguiente. Cada pieza de esta guía —clasificación de errores, metadatos, alarma de umbral cero, idempotencia, redrive controlado— convirtió lo que habría sido una pérdida de datos en 66 minutos de operación rutinaria.

Checklist de producción

- Errores clasificados en transitorios y permanentes; solo los transitorios se reintentan.
- Entre 3 y 5 intentos con backoff exponencial y jitter antes de dead-letter.
- DLQ terminal, con la retención al máximo que permita el broker.

- Metadatos completos en cada mensaje muerto: excepción, stack, intentos, origen, versión, correlación.
- Alerta sobre la edad del mensaje más antiguo, además de la profundidad.
- Procedimiento de redrive documentado: corregir primero, reprocesar en lotes acotados después.
- Consumidores idempotentes, porque los reintentos y el redrive garantizan duplicados tarde o temprano.

Una DLQ bien montada convierte el peor escenario de un sistema de mensajería —el mensaje que nunca avanza— en una tarea rutinaria de operación: el pipeline no se detiene, el fallo queda aislado con todo su contexto y el reprocesamiento es una decisión controlada en lugar de una emergencia.

Cuándo no usar una DLQ

Después de una guía entera a favor, el contrapunto honesto: hay casos donde una DLQ es la herramienta equivocada.

- Datos de bajo valor y alto volumen. Para métricas, telemetría o clicks, el coste de operar la DLQ supera el valor de los mensajes. Descarta con un log estructurado y un contador; nadie va a reprocesar un ping de telemetría de hace tres días.
- Cuando el orden es sagrado. Si saltarse un mensaje corrompe el estado (proyecciones de event sourcing, réplicas, CDC), apartar un evento y seguir con el siguiente es corrupción de datos con pasos intermedios. Ahí el patrón correcto es parar la partición, alertar con severidad máxima y arreglar hacia adelante; el bloqueo no es un fallo del diseño, es el diseño.
- Flujos request-reply síncronos. Si hay un usuario esperando la respuesta, un mensaje que se va a una DLQ para revisión en horas es simplemente un timeout con papeleo. El error debe volver al llamante, que decidirá si reintenta.
- Como sustituto del transactional outbox. La DLQ protege el lado del consumidor. Si tu problema es que el productor pierde eventos al publicar (dual write), ninguna DLQ lo va a arreglar: eso se resuelve con el patrón outbox, que es otra guía de esta serie.

La prueba rápida: ¿alguien va a mirar estos mensajes muertos y actuar? Si la respuesta sincera es no, no construyas la DLQ; construye el log y el contador, y ahórrate la falsa sensación de seguridad.

Preguntas frecuentes

¿Una DLQ por cola o una global?

Por cola (o por topic). Una DLQ global mezcla contextos de fallos que no tienen nada que ver, complica los permisos y convierte el reproceso selectivo en un ejercicio de filtrado. La única excepción razonable es una flota de colas pequeñas y homogéneas gestionadas por el mismo equipo.

¿Cuántos reintentos son los correctos?

Para errores transitorios, entre 3 y 5 con backoff exponencial cubre la inmensa mayoría de los casos; más allá, la probabilidad de que el reintento número 8 triunfe donde fallaron 7 es mínima y solo retrasas la señal de alarma. Para errores permanentes, cero: cada reintento de un error permanente es ruido. Si usas Lambda con SQS, recuerda el mínimo de 5 por los intentos que consume el throttling.

¿Reintentos en el consumidor o delegados al broker?

Ambos, en capas: reintentos rápidos in-process (milisegundos, para blips de red) y el límite del broker como red de seguridad (maxReceiveCount, delivery-limit). Lo que no debes hacer es contar solo en tu código ignorando el contador del broker, porque un crash del proceso resetea tu cuenta pero no la suya.

¿Qué hago con un mensaje que también falla al reprocesarlo?

Si tras corregir el bug un mensaje sigue fallando, apártalo a una cola de estacionamiento (parking lot) para análisis manual y no lo devuelvas a ciclar. Un mensaje que sobrevive a dos diagnósticos es un caso para un humano, no para un tercer redrive.

¿La DLQ puede tener su propia DLQ?

No en cascada. El consumidor de una DLQ debe ser tan trivial que no pueda fallar por lógica de negocio: leer, archivar, alertar. Si ni eso funciona, el problema es de infraestructura y ninguna cola adicional te va a salvar; te salvan la retención larga y las alarmas.

¿Cómo evito que la DLQ se convierta en un vertedero?

Con un SLA operativo, no con tecnología: todo mensaje muerto se revisa en menos de X horas laborables, hay un dashboard visible con la edad del más antiguo, y la responsabilidad de revisarlo tiene nombre y rotación. Una DLQ sin dueño es solo la papelera de reciclaje más cara de tu arquitectura.

¿Los mensajes de la DLQ cuentan para el autoscaling?

No deberían. La profundidad de la DLQ es deuda pendiente de revisión humana, no demanda de procesamiento: inclúyela en el escalado y tendrás réplicas de más cada vez que un incidente llene la cola muerta. Escala el consumidor principal con la cola principal y trata la DLQ como una cola administrativa aparte.

¿Reproceso la DLQ automáticamente cada noche?

Tentador y peligroso. Un reproceso programado sin diagnóstico convierte los errores permanentes en un bucle eterno de madrugada y enmascara la señal de alarma. Si quieres automatizar algo, automatiza el reintento de una clase concreta de error ya diagnosticada como transitoria de larga duración, con límite de edad y de intentos, y deja el resto para un humano.

Las ideas que deben quedarse contigo

Si dentro de seis meses solo recuerdas cinco cosas de esta guía, que sean estas:

- Clasifica cada error como transitorio o permanente antes de decidir nada más; esa distinción gobierna reintentos, DLQ y alertas.
- Los reintentos tienen límite y las DLQs son terminales: un mensaje muerto espera a un humano (o a un archivador), nunca cicla solo.
- El payload viaja intacto y los metadatos del fallo viajan aparte; sin autopsia posible no hay diagnóstico rápido.
- La alarma correcta de una DLQ es umbral cero para la entrada y edad máxima para la espera; el tamaño es la métrica menos interesante.
- Reprocesar exige idempotencia demostrada, causa corregida y lotes con freno; en ese orden, sin excepciones.

Todo lo demás — quorum queues, retry topics, partial batches, archivadores — son implementaciones de estas cinco ideas en la plataforma que te haya tocado. Las plataformas cambian; las ideas, de momento, no.

English Version

The problem: a message that never gets processed

Picture a consumer reading from a queue that hits a message with corrupt JSON. It throws, never acknowledges the message, and the broker redelivers it. Another exception, another redelivery. That single message — a poison message — can loop forever: it burns CPU, floods your logs and, in systems with ordering guarantees like Kafka, blocks the entire partition, so nothing behind it makes progress until it is dealt with.

The standard answer is the Dead Letter Queue (DLQ): a separate queue that receives messages once they exhaust their processing attempts. The pipeline keeps flowing, and you get a place to inspect, fix and reprocess failures calmly, without losing data.

Transient or permanent: the decision that changes everything

Before configuring anything, classify your errors. Not all of them deserve a retry:

- Transient errors (timeouts, HTTP 503, deadlocks, throttling): they usually clear on their own. Retry with exponential backoff and jitter — 3 to 5 attempts is a solid starting point.
- Permanent errors (invalid JSON, schema violations, broken business rules): retrying is pointless. The message should go to the DLQ on the first attempt.

The classic mistake is treating everything the same. Retrying a validation error five times only delays the inevitable and piles load onto the system exactly when it needs it least. And the reverse is just as bad: sending a one-off timeout to the DLQ turns a network blip into manual work for a human.

SQS: maxReceiveCount and the redrive policy

SQS has the most direct support. Every queue can declare a redrive policy: once a message has been received more than maxReceiveCount times without being deleted, SQS automatically moves it to the configured DLQ.

```
import json
import boto3

sqs = boto3.client("sqs")

# 1. The DLQ, with maximum retention (14 days) to leave time for triage
dlq = sqs.create_queue(
    QueueName="orders-dlq",
    Attributes={"MessageRetentionPeriod": "1209600"},
)
dlq_arn = sqs.get_queue_attributes(
    QueueUrl=dlq["QueueUrl"], AttributeNames=["QueueArn"]
)["Attributes"]["QueueArn"]

# 2. The main queue: after 5 receives without a delete, SQS moves the message to the DLQ
sqs.create_queue(
    QueueName="orders",
    Attributes={
        "RedrivePolicy": json.dumps({
            "deadLetterTargetArn": dlq_arn,
            "maxReceiveCount": "5",
        }),
        "VisibilityTimeout": "30",
    },
```

```
)
```

Two details that prevent surprises: DLQ retention is counted from the moment the message first entered SQS, not from when it landed in the DLQ, so always set it to the maximum. And when it is time to reprocess, SQS offers redrive to source: it moves DLQ messages back to their original queue with a single API call, with a configurable rate limit.

SQS in depth: Lambda, partial batches and programmatic redrive

The basic redrive policy works the same when the consumer is a Lambda function, but that combination introduces subtleties that cause accidental dead-lettering if you are not aware of them. They are worth a close look, because SQS + Lambda is by far the most common way to consume queues on AWS today.

Partial batches: do not punish the innocent messages

Lambda reads from SQS in batches of up to 10 messages by default. If your function throws, the whole batch returns to the queue and the ReceiveCount of every message in it goes up — including the ones that were processed successfully. With a low maxReceiveCount, a single poison message can drag perfectly valid messages into the DLQ just because they were unlucky enough to share a batch with it several times in a row.

The fix is partial batch responses: instead of failing all-or-nothing, you tell Lambda exactly which messages failed, and only those go back to the queue.

```
import json

def handler(event, context):
    batch_item_failures = []
    for record in event["Records"]:
        try:
            process(json.loads(record["body"]))
        except Exception:
            batch_item_failures.append(
                {"itemIdentifier": record["messageId"]}
            )
    # Only the failed ones return to the queue; the rest are deleted
    return {"batchItemFailures": batch_item_failures}
```

For Lambda to honor the return value you must enable ReportBatchItemFailures on the event source mapping; without it, the response is silently ignored and you keep the all-or-nothing behavior:

```
aws lambda update-event-source-mapping \
  --uuid <mapping-uuid> \
  --function-response-types "ReportBatchItemFailures"
```

The visibility timeout arithmetic

The second source of phantom dead-lettering is a visibility timeout that is too short. If the message becomes visible again while Lambda is still processing it, another worker receives it, the ReceiveCount goes up without any real failure, and the message ends up in the DLQ despite having been processed correctly (possibly more than once). AWS's recommendation is concrete: set the visibility timeout to at least 6 times the function timeout (plus the batch window if you use one). And with Lambda as the consumer, set maxReceiveCount to at least 5: a concurrency throttle also consumes an attempt, even though your code never ran.

Programmatic redrive with the API

The console button is fine for a one-off incident, but since the redrive API exists you can automate reprocessing with rate control:

```
resp = sqs.start_message_move_task(
    SourceArn="arn:aws:sqs:eu-west-1:123456789012:orders-dlq",
    # No DestinationArn: each message returns to its source queue
    MaxNumberOfMessagesPerSecond=10,
)

# Monitor progress, or abort if things go wrong
sqs.list_message_move_tasks(
    SourceArn="arn:aws:sqs:eu-west-1:123456789012:orders-dlq"
)
sqs.cancel_message_move_task(TaskHandle=resp["TaskHandle"])
```

Two details that save you surprises. First: redrive "to source" only works for messages that reached the DLQ through the redrive policy; if your code published the message to the DLQ manually, SQS does not know where it came from and you must specify an explicit destination. Second: MaxNumberOfMessagesPerSecond is your handbrake. If the bug that filled the DLQ is still alive, you would much rather find out at 10 messages per second than at 10,000 all at once.

DLQs for FIFO queues

The DLQ of a FIFO queue must itself be FIFO. And there is a subtle interaction with message groups: while one message of a group is stuck retrying, the rest of the group waits behind it. It is the same partition-blocking effect we saw in Kafka, and the conclusion is the same: on FIFO queues you want a low maxReceiveCount, because every retry of the poison message delays every message in its group. Moving it to the DLQ unblocks the group, at the cost of processing the following messages without it: if your domain cannot tolerate that gap, your consumer must detect the incomplete group and wait.

RabbitMQ: DLX, the retry cycle and the x-death header

RabbitMQ does not count deliveries on classic queues. Instead, a queue declares a dead letter exchange (DLX): when a message is rejected with requeue=False, expires via TTL, or the queue overflows, RabbitMQ publishes it to that exchange. To emulate retries with a delay you use the retry-cycle pattern: the main queue dead-letters rejects into a retry queue with a TTL, and when the TTL expires the message flows back to the main queue. Each loop increments the x-death header, which acts as your counter.

```
import pika

conn = pika.BlockingConnection(pika.ConnectionParameters("localhost"))
ch = conn.channel()

# Topology: orders -(reject)-> retry (TTL 30s) -> back to orders
# After MAX_ATTEMPTS cycles, the consumer publishes to orders.dlq (terminal)
ch.exchange_declare("orders.retry.dlx", "fanout", durable=True)
ch.exchange_declare("orders.rework", "fanout", durable=True)

ch.queue_declare("orders", durable=True, arguments={
    "x-dead-letter-exchange": "orders.retry.dlx",
})
ch.queue_declare("orders.retry", durable=True, arguments={
    "x-message-ttl": 30000,                    # wait 30 s
    "x-dead-letter-exchange": "orders.rework",  # then requeue
})
ch.queue_declare("orders.dlq", durable=True)    # terminal: no DLX

ch.queue_bind("orders.retry", "orders.retry.dlx")
ch.queue_bind("orders", "orders.rework")

MAX_ATTEMPTS = 5

def attempts_so_far(props):
    for death in (props.headers or {}).get("x-death", []):
        if death["queue"] == "orders":
            return int(death["count"])
    return 0
```

```
def publish_to_dlq(ch, body, props, exc):
    headers = dict(props.headers or {})
    headers["x-error"] = type(exc).__name__
    headers["x-error-detail"] = str(exc)[:500]
    ch.basic_publish("", "orders.dlq", body,
                     pika.BasicProperties(headers=headers, delivery_mode=2))

def on_message(ch, method, props, body):
    try:
        process(body)
        ch.basic_ack(method.delivery_tag)
    except PermanentError as exc:
        publish_to_dlq(ch, body, props, exc)   # no retries: they are pointless
        ch.basic_ack(method.delivery_tag)
    except TransientError as exc:
        if attempts_so_far(props) >= MAX_ATTEMPTS:
            publish_to_dlq(ch, body, props, exc)
            ch.basic_ack(method.delivery_tag)
        else:
            ch.basic_reject(method.delivery_tag, requeue=False)  # into the retry cycle
```

If you use quorum queues (the recommended type today), there is a shortcut: the delivery-limit argument makes the queue itself count deliveries and dead-letter the message once the limit is exceeded, with no manual x-death inspection.

Per-message backoff in SQS: ChangeMessageVisibility as a timer

SQS has no retry topics and no native exponential delays between retries: if a message returns to the queue, it becomes visible again after the fixed visibility timeout, always the same one. But there is a little-known lever to get real per-message exponential backoff: ChangeMessageVisibility. Instead of letting the timeout expire on its own, the consumer explicitly adjusts it based on the attempt number before releasing the message.

```
def handle(message):
    receive_count = int(
        message["Attributes"]["ApproximateReceiveCount"]
    )
    try:
        process(json.loads(message["Body"]))
        sqs.delete_message(
            QueueUrl=QUEUE_URL,
            ReceiptHandle=message["ReceiptHandle"],
        )
    except TransientError:
        # Exponential backoff with a cap: 30s, 60s, 120s... max 15 min
        delay = min(30 * (2 ** (receive_count - 1)), 900)
        sqs.change_message_visibility(
            QueueUrl=QUEUE_URL,
            ReceiptHandle=message["ReceiptHandle"],
            VisibilityTimeout=delay,
        )
        # No delete: the message reappears after the delay
    except PermanentError:
        move_to_dlq(message)   # publish to DLQ + delete from queue
        sqs.delete_message(
            QueueUrl=QUEUE_URL,
            ReceiptHandle=message["ReceiptHandle"],
        )
```

Three observations. First: ApproximateReceiveCount is your free attempt counter; request it in ReceiveMessage via AttributeNames. Second: the backoff coexists with the redrive policy — maxReceiveCount still counts receives, so size the two together (with 5 receives and this scale, the message accumulates about 4 minutes of waiting before the DLQ; raise maxReceiveCount if you need longer recovery windows). Third: for permanent errors the right pattern is still the shortcut — publish to the DLQ manually and delete, rather than burning receives. One caveat: a manually moved message cannot use the console's redrive "to source", so store the source queue URL in a message attribute.

SNS and EventBridge: where the DLQ lives in a fan-out

In event architectures on AWS, SQS rarely stands alone: usually SNS or EventBridge is fanning events out to several queues. And that raises a design question with an unintuitive answer: where do you put the DLQ?

The short answer: on each subscription, not on the publisher. If SNS fails to deliver to one of the five subscribed queues (because someone deleted the queue, or the policy does not allow delivery), that failure belongs to that specific subscription; the other four deliveries went fine. That is why both SNS and EventBridge let you configure a DLQ per subscription or per rule/target:

```
sns.set_subscription_attributes(
    SubscriptionArn=SUBSCRIPTION_ARN,
    AttributeName="RedrivePolicy",
    AttributeValue=json.dumps({
        "deadLetterTargetArn":
            "arn:aws:sqs:eu-west-1:123456789012:sns-delivery-dlq"
    }),
)
```

Mind the difference in semantics, because confusing them costs data:

- The SNS subscription / EventBridge rule DLQ captures delivery failures: the message never reached the destination queue. Without it, once delivery retries are exhausted the event is lost forever — and nobody notices, because no consumer ever got to fail.
- The SQS queue's DLQ (the usual redrive policy) captures processing failures: the message arrived, but your consumer could not handle it.

You need both layers. EventBridge additionally retries deliveries for up to 24 hours with backoff before giving up; if your target was down longer than that (a disastrous weekend deploy), only the rule's DLQ lets you recover those events. The general lesson for any broker with fan-out: every hop in the chain needs its own plan for failure, because every hop can fail for different reasons.

Modern RabbitMQ: quorum queues, delivery-limit and reliable dead-lettering

Everything above about DLX applies to classic queues, but quorum queues have been the recommended option for anything important for years now, and RabbitMQ 4 shipped a change that surprised more than one team on upgrade: delivery-limit now defaults to 20.

Quorum queues track redeliveries in the x-delivery-count header, which is far easier to read than the historical x-death of classic queues. When the counter exceeds the delivery-limit, the message is dead-lettered if a DLX is configured… or dropped if there is none. Yes: dropped. If you migrated to RabbitMQ 4 without configuring a dead-letter-exchange on your quorum queues, a poison message silently vanishes on its twentieth attempt. Always configure it, if only to be able to do the autopsy:

```
rabbitmqctl set_policy dlq-orders "^orders" \
  '{"delivery-limit": 5,
    "dead-letter-exchange": "dlx",
    "dead-letter-strategy": "at-least-once",
    "overflow": "reject-publish"}' \
  --apply-to quorum_queues
```

At-least-once dead-lettering

Classic dead-lettering is at-most-once: the broker re-publishes the message to the DLX without waiting for confirmation, and if at that moment no queue is bound or the node goes down, the message is lost. For a click log that may be acceptable; for a payment, it is not. Quorum queues offer dead-letter-strategy: at-least-once, which re-publishes with internal publisher confirms and does not let go of the message until the

destination queue confirms it has persisted it.

The requirements: the source queue must be a quorum queue, overflow must be reject-publish, and the destination queue should be a quorum queue too. The cost is memory (messages pending confirmation are retained), but that is the price of being able to state that no dead message is lost along the way.

A recent nuance that fixes a historical injustice: since RabbitMQ 4.3 there are two separate counters, delivery-count (which only increases when the redelivery is due to an actual consumer failure) and acquired-count (which increases on any requeue). Before that, a consumer returning messages because of a rebalance or a clean shutdown inflated the counter and could send perfectly healthy messages to the DLQ.

In the consumer, use x-delivery-count to decide without waiting for the broker limit when the error is clearly permanent:

```
def on_message(ch, method, properties, body):
    deliveries = (properties.headers or {}).get("x-delivery-count", 0)
    try:
        process(json.loads(body))
        ch.basic_ack(method.delivery_tag)
    except PermanentError:
        # No point waiting for 5 attempts: straight to the DLX
        ch.basic_nack(method.delivery_tag, requeue=False)
    except Exception:
        if deliveries >= 4:
            ch.basic_nack(method.delivery_tag, requeue=False)
        else:
            ch.basic_nack(method.delivery_tag, requeue=True)
```

The broker acts as the safety net with its delivery-limit, and your code takes the fast path for errors it recognizes as permanent. The two layers add up.

Kafka: the DLQ you build yourself

Kafka has no native DLQ: a consumer that fails and never advances its offset re-reads the same record forever and blocks the partition. The pattern is to build it in the consumer: catch the error, publish the record to a *.dlq topic with diagnostic headers, and commit the offset anyway to unblock the partition.

```
import json
import time
import traceback
from confluent_kafka import Consumer, Producer

consumer = Consumer({
    "bootstrap.servers": "localhost:9092",
    "group.id": "orders-worker",
    "enable.auto.commit": False,   # commit only after deciding what to do
    "auto.offset.reset": "earliest",
})
producer = Producer({"bootstrap.servers": "localhost:9092"})
consumer.subscribe(["orders"])

MAX_ATTEMPTS = 4

def send_to_dlq(msg, exc, attempts):
    origin = "orders/" + str(msg.partition()) + "/" + str(msg.offset())
    producer.produce(
        "orders.dlq",
        key=msg.key(),
        value=msg.value(),          # original payload, untouched
        headers={
            "dlq.error": type(exc).__name__,
            "dlq.detail": str(exc)[:500],
            "dlq.stack": traceback.format_exc()[-2000:],
            "dlq.origin": origin,
            "dlq.attempts": str(attempts),
            "dlq.ts": str(int(time.time() * 1000)),
        },
```

```
    )
    producer.flush()

while True:
    msg = consumer.poll(1.0)
    if msg is None or msg.error():
        continue
    for attempt in range(1, MAX_ATTEMPTS + 1):
        try:
            process(json.loads(msg.value()))
            break
        except PermanentError as exc:
            send_to_dlq(msg, exc, attempt)   # straight to the DLQ, no retries
            break
        except TransientError as exc:
            if attempt == MAX_ATTEMPTS:
                send_to_dlq(msg, exc, attempt)
            else:
                time.sleep(min(2 ** attempt, 30))  # in-process backoff
    consumer.commit(msg)   # unblocks the partition no matter what
```

Watch out for that time.sleep: if total backoff exceeds max.poll.interval.ms, the broker kicks the consumer out of the group and triggers a rebalance. For long waits, use tiered retry topics (for example orders.retry.1m, orders.retry.10m) instead of sleeping in-process. If you are on Spring Kafka, DeadLetterPublishingRecoverer implements this whole pattern out of the box.

Advanced Kafka: tiered retry topics and non-blocking retries

The retry-loop consumer we saw earlier has a serious limitation: while it retries, the whole partition is blocked. With millisecond backoffs you might get away with it; with minute-long backoffs — the ones that actually help when a dependency is down — you will not. The standard pattern to unblock the partition is tiered retry topics.

The idea: besides the main topic and the DLQ, you create intermediate topics with increasing delays, for example orders.retry.1m and orders.retry.10m. When a message fails on the main topic, it is published to the first retry topic with a "do not process before" timestamp. The consumer of each retry topic checks that mark: if it is not due yet, it pauses the partition and waits; if it is due, it processes. If the message exhausts every tier, it lands in the DLQ. The main topic's partition never stops.

```
RETRY_TOPICS = ["orders.retry.1m", "orders.retry.10m"]
DELAYS = {"orders.retry.1m": 60, "orders.retry.10m": 600}

def next_destination(topic):
    if topic == "orders":
        return RETRY_TOPICS[0]
    idx = RETRY_TOPICS.index(topic)
    if idx + 1 < len(RETRY_TOPICS):
        return RETRY_TOPICS[idx + 1]
    return "orders.dlq"

def forward(msg, exc, source_topic):
    dest = next_destination(source_topic)
    headers = dict(msg.headers() or [])
    attempts = int(headers.get("retry.attempts", b"0")) + 1
    not_before_ms = int(time.time() * 1000) + DELAYS.get(dest, 0) * 1000
    producer.produce(
        dest,
        key=msg.key(),
        value=msg.value(),           # original payload, untouched
        headers={
            "retry.attempts": str(attempts),
            "retry.not.before": str(not_before_ms),
            "dlq.error": type(exc).__name__,
            "dlq.detail": str(exc)[:500],
        },
    )
```

```
    producer.flush()

def run_retry_consumer(topic):
    consumer.subscribe([topic])
    while True:
        msg = consumer.poll(1.0)
        if msg is None or msg.error():
            continue
        headers = dict(msg.headers() or [])
        not_before = int(headers.get("retry.not.before", b"0"))
        if time.time() * 1000 < not_before:
            # Not due yet: pause, rewind to the same offset and wait
            tp = TopicPartition(msg.topic(), msg.partition(), msg.offset())
            consumer.pause([tp])
            consumer.seek(tp)
            wait_s = (not_before - time.time() * 1000) / 1000
            time.sleep(min(wait_s, 30))
            consumer.resume([tp])
            continue
        try:
            process(json.loads(msg.value()))
        except Exception as exc:
            forward(msg, exc, topic)
        consumer.commit(msg)
```

The warning nobody can spare you: the moment a message jumps to a retry topic, you lose per-key ordering. A later event for the same entity can be processed before the one waiting for its retry. If ordering matters, you have two ways out: retry blocking (accepting the latency on that partition) or detect in the consumer that an earlier message for that key is pending in retry and set the new one aside as well. Choosing retry topics is choosing availability over ordering; do it with your eyes open.

In the JVM ecosystem you do not need to build any of this by hand: Spring Kafka ships it out of the box with the @RetryableTopic annotation, which creates the retry topics and the dead letter topic automatically and handles the waiting with partition pauses, without blocking the thread:

```
@RetryableTopic(
    attempts = "4",
    backoff = @Backoff(delay = 60000, multiplier = 10.0)
)
@KafkaListener(topics = "orders")
public void listen(Order order) {
    process(order);
}

@DltHandler
public void dlt(Order order,
                @Header(KafkaHeaders.ORIGINAL_TOPIC) String source) {
    log.error("Dead message coming from {}", source);
}
```

And if the failing component is a Kafka Connect connector (a sink into a database, say), do not write code: it is configuration. With errors.tolerance the connector stops dying on the first bad record and diverts it to its own DLQ with the error context in headers:

```
{
  "errors.tolerance": "all",
  "errors.deadletterqueue.topic.name": "connect.orders.dlq",
  "errors.deadletterqueue.topic.replication.factor": 3,
  "errors.deadletterqueue.context.headers.enable": true,
  "errors.retry.timeout": 300000,
  "errors.log.enable": true
}
```

With context.headers.enable on, Connect adds __connect.errors.* headers (source topic and offset, exception class, message) to every diverted record: exactly the metadata we said every DLQ needs, for free.

Choosing a platform: what each broker gives you and what it charges

If you are deciding where a new system will live, the dead-lettering story sums up each broker's philosophy well:

- SQS is the most complete out of the box: a declarative redrive policy, a managed receive counter, a redrive API with rate control and CloudWatch metrics without installing anything. In exchange, variable-delay retries require the ChangeMessageVisibility trick, and maximum retention is 14 days: if your review process is slow, the clock is ticking.
- RabbitMQ gives you the most expressive topology: a DLX routing by routing key, policies by queue pattern, and with quorum queues a reliable counter (x-delivery-count) plus at-least-once dead-lettering. What it does not give you is a reprocessing tool: the redrive is yours to write. And it demands watching configuration across major upgrades, as version 4's default delivery-limit demonstrated.
- Kafka gives you nothing… and gives you everything: there is no native DLQ, but long retention, re-consumable offsets and headers make building custom DLQs and retry topics natural. It is the option with the most pieces on your plate — and also the only one where "reprocess the dead from three weeks ago" is trivial if you sized retention right.

The practical rule: on managed AWS, start with SQS and do not reinvent; if you already operate RabbitMQ, use quorum queues and a DLX policy from day one; on Kafka, accept that the DLQ is your code and treat it as such, with tests and an owner.

Deserialization and schemas: the number one factory of poison messages

Most poison messages are not born from exotic bugs. They are born from something far more mundane: a producer that changed the format. A renamed field, a type that went from integer to string, a JSON truncated by a size limit. The failure happens before your business logic, in the deserialization layer, and that matters because many frameworks retry by default errors that will never fix themselves: a corrupt JSON will still be corrupt on attempt one thousand.

- Treat deserialization errors as permanent, always. Straight to the DLQ, no retries. There is no dependency to recover and no timeout to wait out.
- Deserialize and validate before touching anything. If the message is going to fail, let it fail before producing half-applied side effects.
- Store the raw bytes in the DLQ, not your failed interpretation of them. The original payload is the only reliable forensic evidence.

With Pydantic, the boundary fits in a function with one line of logic:

```
from pydantic import BaseModel, ValidationError

class Order(BaseModel):
    order_id: str
    amount_cents: int
    currency: str

def parse(raw: bytes) -> Order:
    try:
        return Order.model_validate_json(raw)
    except (ValidationError, UnicodeDecodeError) as exc:
        raise PermanentError("invalid payload") from exc
```

A schema registry (Avro, Protobuf or JSON Schema) changes the shape of the problem: compatibility checks stop most breaking changes on the producer side, before they ever reach the queue. But "compatible" according to the registry does not always mean "compatible with your code": a new optional field that your logic assumes is present passes the registry check and blows up in your consumer. The registry shrinks the

problem's surface; contract tests close it.

One operational tip that pays dividends: distinguish deserialization failures from business failures in the DLQ metadata. The former are fixed with a deploy (of the consumer, or a producer rollback) and can be confidently reprocessed in bulk; the latter demand a message-by-message look. When the DLQ mixes both without labels, every incident starts with an hour of archaeology.

What to store alongside the failed message

A DLQ message without context forces forensic archaeology. Always attach, as headers or a metadata envelope: the exception type and message, the stack trace, the attempt count, the exact origin (queue, or topic/partition/offset), the version of the consumer that failed, and the request correlation ID. Keep the original payload untouched: if you transform it before enqueueing, you can no longer reproduce the failure exactly as it happened.

Designing the message envelope: version, identity and traces

A good part of what makes a DLQ manageable is decided long before the first failure: in the design of the message itself. A naked payload — a JSON with business fields and nothing else — is cheap to produce and very expensive to operate. A minimal envelope around it changes the rules of the game:

```
{
  "message_id": "9c2f4a1e-8d3b-4f6a-9e21-7b5c0d8e4f12",
  "type": "order.created",
  "version": 2,
  "occurred_at": "2026-07-28T09:14:03Z",
  "traceparent": "00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01",
  "data": {
    "order_id": "P-48821",
    "amount_cents": 4990,
    "currency": "EUR"
  }
}
```

Each envelope field solves a concrete DLQ problem:

- message_id, stable and generated by the producer: it is the key of the inbox table and what lets you state "this message was already processed" during a redrive. Without it, idempotency depends on fragile heuristics over the content.
- type and version: the consumer can route and, above all, cleanly reject versions it does not understand ("version 3: unsupported") instead of failing cryptically in the middle of the logic. And when you look at the DLQ, the version distribution instantly tells you whether the problem is a schema migration.
- occurred_at: not the broker timestamp — the business fact's timestamp. Essential when reprocessing days late, when "now" no longer means anything.
- traceparent (the W3C Trace Context standard): propagated into the DLQ and through the reprocess, it connects the dead message with the original distributed trace. The on-call can jump from the message in the DLQ to the exact trace where it died, with all spans and logs correlated, in one click.

None of this is sophisticated; all of it is discipline. Teams that have it debug a DLQ in minutes. Teams that do not print payloads in the terminal and do archaeology with grep.

Monitoring: age matters more than depth

A DLQ without alerts is a graveyard: every message that lands there unseen is silent data loss. Two practical rules:

- Alert on the age of the oldest message (in SQS, ApproximateAgeOfOldestMessage), not just on depth.

Ten messages from a minute ago may be noise; one from three days ago is a problem nobody has looked at.

- Every DLQ message should be actionable: it either reveals a bug to fix, a producer validating badly, or a broken contract between services. If one message type keeps arriving and you decide to ignore it, filter it out upstream — do not let it pile up.

And one design rule: the DLQ is terminal. No DLX on the DLQ itself, no automatic redrive from one DLQ into another. Message ping-pong loops are one of the most common ways to take down a broker in production.

Applied observability: CloudWatch alarms and Prometheus rules

We already established the principle: age matters more than depth. Let us get down to concrete configuration on the two most common monitoring platforms.

CloudWatch for SQS

Two alarms cover 95% of incidents. The first fires on the very first message that enters the DLQ: a healthy DLQ is empty, so the correct threshold is zero.

```
cloudwatch.put_metric_alarm(
    AlarmName="orders-dlq-has-messages",
    Namespace="AWS/SQS",
    MetricName="ApproximateNumberOfMessagesVisible",
    Dimensions=[{"Name": "QueueName", "Value": "orders-dlq"}],
    Statistic="Maximum",
    Period=300,
    EvaluationPeriods=1,
    Threshold=0,
    ComparisonOperator="GreaterThanThreshold",
    AlarmActions=[SNS_TOPIC_ARN],
)
```

The second one watches ApproximateAgeOfOldestMessage — but on the main queue: if the oldest message in the work queue is aging, your consumer is down or stuck, and the DLQ will not even get to hear about it.

Prometheus for RabbitMQ and Kafka

```
# Dead messages sitting unreviewed for 15 minutes
- alert: DLQMessagesWaiting
  expr: rabbitmq_queue_messages{queue=~".*dlq.*"} > 0
  for: 15m
  labels:
    severity: warning

# Messages are ARRIVING at the DLQ right now
- alert: DLQGrowthRate
  expr: rate(rabbitmq_queue_messages_published_total{queue=~".*dlq.*"}[5m]) > 1
  labels:
    severity: critical
```

The distinction between the two is what prevents alert fatigue: "there are 3 messages from yesterday pending review" is a business-hours warning; "messages are arriving at more than one per second" means something is broken in production right now and should wake someone up. In Kafka, the equivalent metrics are the production rate to the .dlq topic and the lag of the consumer group that is supposed to review it: if that lag grows without bound, your DLQ is a graveyard, not a waiting room.

Complete the picture with two application metrics no exporter gives you for free: a dlq_messages_total counter labeled by source queue and error type (it tells you what is breaking and where without opening a single message), and a histogram of attempts consumed before dying. That last one is revealing: if most of your dead messages exhausted all their retries, they were transient errors that did not recover in time and you may need more slack; if they died on the first attempt, they are permanent and your retries only add

noise and latency.

Reprocessing: deliberate, never blind

The right order is: diagnose, fix the bug, deploy, and then redrive the messages, in bounded batches while watching your metrics. Blind bulk redrive sends the same poison right back into the queue and doubles the incident. Two prerequisites:

- Idempotency: a message that failed mid-processing may have left partial effects behind. Reprocessing it must not duplicate payments, orders or emails.
- Ordering: if your domain depends on order (events for the same aggregate), remember that reprocessed messages arrive late and out of sequence; the consumer has to tolerate that, for example with version numbers.

Idempotency: the precondition for fearless reprocessing

Reprocessing a DLQ means, by definition, delivering messages that were already attempted. If your consumer charged the card and then failed to save to the database, the reprocess will charge it again. Before touching the redrive button you need real idempotency, and the most robust form is the inbox table: record every processed message_id in the same transaction as the business effect.

```
CREATE TABLE inbox (
    message_id   text PRIMARY KEY,
    processed_at timestamptz NOT NULL DEFAULT now()
);
```

```
def process_idempotent(message_id: str, payload: dict):
    with db.begin() as tx:
        inserted = tx.execute(
            "INSERT INTO inbox (message_id) VALUES (%s) "
            "ON CONFLICT DO NOTHING",
            (message_id,),
        ).rowcount
        if inserted == 0:
            return  # already processed: ack and move on
        apply_business_logic(tx, payload)  # same transaction
```

The key is atomicity: the inbox record and the business effect commit together or roll back together. With this in place, the worst case of any reprocess — or of any broker redelivery, which in at-least-once systems happen on their own — is a silent no-op. It is the same pattern that protects against everyday duplicates; the DLQ merely makes it mandatory.

A home-grown redrive with a handbrake

For RabbitMQ and Kafka there is no managed redrive API: you write the tool yourself. The three properties it must have: a quantity limit, a rate limit, and automatic abort if messages start failing again en masse (the unmistakable sign that the bug is still alive).

```
def redrive(dlq, destination, max_messages=100,
            rate_per_second=5, abort_after_failures=10):
    moved, failed = 0, 0
    for msg in dlq.read(limit=max_messages):
        if failed >= abort_after_failures:
            log.error("too many re-failures: bug still alive, aborting")
            break
        try:
            destination.publish(msg.body, headers=msg.headers)
            dlq.ack(msg)
            moved += 1
```

```
            except Exception:
                failed += 1
            time.sleep(1.0 / rate_per_second)
    return moved, failed
```

Always start with a small sample (max_messages=10) and watch. If the sample comes through clean, raise the volume. A redrive is a controlled experiment, not a floodgate.

Autoscaling and queues: KEDA and the accordion effect

An operational detail connecting DLQs with capacity: if you scale your consumers on queue depth (the usual pattern with KEDA on Kubernetes), a poison message without a DLQ does not just block processing — it inflates the scaling metric. The queue grows because nothing advances, the autoscaler adds replicas, and now you have twenty pods retrying the same corrupt message instead of five. You pay more to fail faster.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: orders-consumer
spec:
  scaleTargetRef:
    name: orders-consumer
  minReplicaCount: 1
  maxReplicaCount: 20
  triggers:
    - type: aws-sqs-queue
      metadata:
        queueURL: https://sqs.eu-west-1.amazonaws.com/123456789012/orders
        queueLength: "5"      # desired messages per replica
        awsRegion: eu-west-1
```

With the DLQ in place, the poison message gets out of the way within a few attempts, depth reflects real work and scaling makes sense again. Two tips: do not include the DLQ in the main consumer's scaling metric (its depth is not demand, it is debt), and give the DLQ its own ScaledObject if its archiver needs to absorb spikes — with a low maxReplicaCount, because archiving is not urgent.

Multi-tenant: when the poison comes from a single customer

Multi-tenant systems have a failure mode of their own: a single customer — a misconfigured integration of theirs, an exotic format, an anomalous volume — generates 99% of the poison messages, and their noise buries everyone else's failures. Three layered defenses:

- Label the tenant in the dead message's metadata from day one. It is the dimension that turns "there are 4,000 messages in the DLQ" into "tenant ACME has 3,960 and the remaining 40 are spread out" — a completely different incident.
- Alert per tenant, not just globally. A high global threshold so nobody gets woken up by a known noisy customer, and per-tenant thresholds to detect that a small customer is suddenly failing at 100% — something the aggregate hides.
- If a tenant saturates chronically, separate them: their own queue with their own DLQ. Queue-level isolation is messaging's version of the bulkhead pattern, and it keeps one customer's noise from degrading everyone's operations.

The short version: a DLQ shared across tenants is acceptable; a DLQ you cannot filter by tenant is a trap that springs shut on the day you need it most.

Testing: if you never test the failure path, it does not exist

The happy path gets tested just by using the application. The DLQ path only runs when something breaks, so if you do not cover it with tests, its first real execution will be in production, in the small hours, with an audience. With Testcontainers (real brokers in Docker) and LocalStack (local SQS) you can cover it in CI without mocks that lie.

```
import pytest
from testcontainers.rabbitmq import RabbitMqContainer

@pytest.fixture(scope="module")
def rabbit():
    with RabbitMqContainer("rabbitmq:4.1-management") as c:
        yield c

def test_poison_message_reaches_dlq_with_metadata(rabbit):
    setup_topology(rabbit)          # queues + DLX as in production
    publish(rabbit, "orders", b"this is not json")
    run_consumer_until_idle(rabbit)

    dead = get_one(rabbit, "orders.dlq", timeout=10)
    assert dead.body == b"this is not json"   # payload intact
    assert dead.headers["dlq.error"] == "PermanentError"
    assert "dlq.stack" in dead.headers
```

The four tests that actually matter:

- A permanent error goes to the DLQ on the first attempt, with no retries in between.
- A transient error is retried N times with backoff and only then gives up.
- The dead message preserves the original payload byte for byte, plus all its metadata.
- Reprocessing an already-processed message is a no-op (idempotency works).

And a subtler one almost nobody writes: while messages are failing, the consumer keeps making progress with the rest. That is the property that justifies this whole architecture; it deserves its own assert.

For SQS, LocalStack reproduces the full redrive policy: create a queue with maxReceiveCount 2 in the test, publish a message that always fails, and assert it shows up in the DLQ after exactly two receives, not one more.

The DLQ consumer: archive before it expires

We said the DLQ should not have its own DLQ and that its consumer must be trivial. This is the concrete pattern: an archiver that copies every dead message to cheap, durable storage (S3, GCS, a bucket with lifecycle rules) before retention deletes it. With that, the queue stops being the sole custodian of the data and the clock pressure disappears.

```
def archive_dlq_message(message):
    key = "dlq/orders/{}/{}.json".format(
        time.strftime("%Y/%m/%d"),
        message["MessageId"],
    )
    s3.put_object(
        Bucket="dlq-archive",
        Key=key,
        Body=json.dumps({
            "body": message["Body"],
            "attributes": message.get("MessageAttributes", {}),
            "receive_count":
                message["Attributes"]["ApproximateReceiveCount"],
            "archived_at": time.time(),
        }),
    )
    # Delete from the queue only after the archive is confirmed
    sqs.delete_message(
        QueueUrl=DLQ_URL,
        ReceiptHandle=message["ReceiptHandle"],
```

```
    )
```

Details that matter: the date-partitioned key lets you list "everything that died on Tuesday" without scanning the bucket; you archive the full message with metadata, not just the body; and the queue delete happens after the confirmed put — the reverse order loses data if the archiver dies midway. On top of the archive you can build what the queue will never give you: Athena queries, years of retention at pennies, and selective reprocessing by date or error type.

Security and retention: the DLQ is production data too

A DLQ accumulates precisely the messages nobody is processing, and that turns it into the forgotten corner holding real data: payloads with emails, addresses, amounts, sometimes tokens. Three rules:

- Same protection as the main queue. Encryption at rest (SSE-KMS on SQS, TLS and disk encryption on RabbitMQ and Kafka) and equally strict access control. It is surprisingly common to find the main queue locked down tight and the DLQ wide open "because it is for debugging".
- Explicit, finite retention. SQS's 14 days is the available maximum, not a goal. In Kafka, set retention.ms on the .dlq topic; in RabbitMQ, a TTL on the dead queue. And align that window with your personal-data policy: the GDPR right to erasure applies to dead messages too — and they are the ones that most easily end up forgotten in a corner.
- Watch what you copy into the metadata. The stack trace you store in dlq.stack can carry the entire payload interpolated into the exception message. Redact sensitive fields before writing the metadata, just as you would in your logs.

Common pitfalls

- Acking before processing: a consumer that acks and then throws loses the message without a trace. The ack always comes after success (or after the conscious decision to dead-letter).
- Retrying permanent errors: five attempts against corrupt JSON are five guaranteed failures plus extra latency for everything else.
- Insufficient DLQ retention: if messages expire before anyone looks at them, the DLQ merely postpones the data loss.
- Blind redrive without fixing the cause: messages come back, fail again and return to the DLQ, now with doubled metrics.
- No error metadata: it forces you to reproduce the failure just to understand it — exactly what the DLQ was supposed to prevent.

Runbook: the day-to-day of operating a DLQ

Architecture is half the job; the other half is what a human does when the alarm rings. This is the runbook that should be linked from that alert:

- 1. Classify without opening messages. Look at the aggregated metadata: one error type or several? One source queue or many? Does the start line up with a deploy, a traffic spike or a provider outage? 80% of diagnoses close right here.
- 2. Determine whether it is still flowing in. If the DLQ inflow rate is zero, the incident is over and you can work calmly. If messages keep arriving, the bug is alive: your priority is stopping the bleeding (rollback, feature flag, hotfix), not reprocessing.
- 3. Fix the cause before touching the DLQ. A redrive without a fix is a boomerang: the messages will come back, with their retry counters exhausted and less slack.

- 4. Reprocess in growing batches. A sample of 10, observe, then the rest with a rate limit. Watch the re-failure rate throughout.
- 5. Close with hygiene. DLQ at zero, alarm recovered, and a note in the postmortem: what error type it was, how many messages, how long each phase took. That data calibrates your retries and thresholds for next time.

And one ten-minute weekly task that prevents scares: review DLQs holding old, unclaimed messages. If they have been there over a week, either decide to reprocess them or decide to discard them — but decide; indefinite limbo is the door to losing them to retention.

Case study: the migration that filled a DLQ at 2:14 a.m.

To close, a real incident (names changed) that shows every piece working together.

An order pipeline on SQS and Lambda. The producer team deploys at 2:03 a version that renames the amount field to amount_cents. Coordination with the consumer team existed… in a ticket planned for the following week. At 2:07 the first orders with the new schema arrive; the consumer validates them with Pydantic, validation raises a permanent error and the messages go straight to the DLQ with their metadata.

- 2:14 — The zero-threshold alarm fires on the first dead message. The on-call opens the dashboard.
- 2:19 — Without opening a single message, the metadata tells the story: dlq.error is ValidationError in 100% of cases, dlq.detail says "amount: field required", and the exact start matches the producer's 2:03 deploy. Diagnosis: five minutes.
- 2:31 — Decision: instead of asking the other team for a rollback, the consumer ships a hotfix that accepts both fields (the expand step of expand-contract). Meanwhile, orders with the old schema keep flowing normally: the DLQ is isolating the problem, not blocking the pipeline.
- 2:58 — Hotfix deployed. Redrive through the SQS API at 10 messages per second. All 312 dead messages reprocess without a single re-failure; the inbox table guarantees none is applied twice.
- 3:20 — DLQ at zero. Maximum age reached by the oldest message: 51 minutes. Orders lost: zero. Customers affected: zero.

The moral is not that the incident was elegant, but everything that did not happen: nobody restored backups, nobody wrote 3 a.m. SQL to reconstruct orders, nobody learned about the problem from an angry customer the next morning. Every piece in this guide — error classification, metadata, the zero-threshold alarm, idempotency, controlled redrive — turned what would have been data loss into 66 minutes of routine operations.

Production checklist

- Errors classified as transient or permanent; only transient ones get retried.
- 3 to 5 attempts with exponential backoff and jitter before dead-lettering.
- A terminal DLQ, with retention set to the broker maximum.
- Full metadata on every dead message: exception, stack, attempts, origin, version, correlation ID.
- Alerts on the age of the oldest message, in addition to depth.
- A documented redrive procedure: fix first, reprocess in bounded batches afterwards.
- Idempotent consumers, because retries and redrive guarantee duplicates sooner or later.

A well-built DLQ turns a messaging system's worst-case scenario — the message that never makes progress — into routine operations: the pipeline keeps moving, the failure sits isolated with its full context, and reprocessing becomes a controlled decision instead of an emergency.

When not to use a DLQ

After an entire guide in favor, the honest counterpoint: there are cases where a DLQ is the wrong tool.

- Low-value, high-volume data. For metrics, telemetry or clicks, the cost of operating the DLQ exceeds the value of the messages. Drop with a structured log and a counter; nobody is going to reprocess a three-day-old telemetry ping.
- When ordering is sacred. If skipping a message corrupts state (event sourcing projections, replicas, CDC), setting an event aside and continuing with the next is data corruption with extra steps. There, the correct pattern is to stop the partition, alert at maximum severity and fix forward; the blocking is not a flaw in the design — it is the design.
- Synchronous request-reply flows. If a user is waiting for the response, a message that goes to a DLQ for review within hours is just a timeout with paperwork. The error must go back to the caller, who decides whether to retry.
- As a substitute for the transactional outbox. The DLQ protects the consumer side. If your problem is the producer losing events at publish time (the dual write), no DLQ will fix it: that is solved with the outbox pattern, covered in another guide in this series.

The quick test: is anyone going to look at these dead messages and act? If the honest answer is no, do not build the DLQ; build the log and the counter, and spare yourself the false sense of safety.

Frequently asked questions

One DLQ per queue, or a global one?

Per queue (or per topic). A global DLQ mixes failure contexts that have nothing to do with each other, complicates permissions and turns selective reprocessing into a filtering exercise. The only reasonable exception is a fleet of small, homogeneous queues owned by the same team.

How many retries is the right number?

For transient errors, 3 to 5 with exponential backoff covers the vast majority of cases; beyond that, the odds that attempt number 8 succeeds where 7 failed are minimal and you are only delaying the alarm signal. For permanent errors, zero: every retry of a permanent error is noise. If you use Lambda with SQS, remember the minimum of 5 because throttling consumes attempts.

Retries in the consumer, or delegated to the broker?

Both, in layers: fast in-process retries (milliseconds, for network blips) and the broker limit as the safety net (maxReceiveCount, delivery-limit). What you must not do is count only in your code while ignoring the broker's counter, because a process crash resets your count but not the broker's.

What do I do with a message that fails again on reprocessing?

If after fixing the bug a message still fails, set it aside in a parking-lot queue for manual analysis and do not send it back around. A message that survives two diagnoses is a case for a human, not for a third redrive.

Can the DLQ have its own DLQ?

Not as a cascade. The consumer of a DLQ should be so trivial it cannot fail on business logic: read, archive, alert. If even that fails, the problem is infrastructure and no additional queue will save you; long retention and alarms will.

How do I keep the DLQ from becoming a dumping ground?

With an operational SLA, not with technology: every dead message gets reviewed within X business hours, there is a visible dashboard with the age of the oldest one, and the responsibility for reviewing it has a name and a rotation. A DLQ without an owner is just the most expensive recycle bin in your architecture.

Do DLQ messages count toward autoscaling?

They should not. DLQ depth is debt awaiting human review, not processing demand: include it in scaling and you will have extra replicas every time an incident fills the dead queue. Scale the main consumer on the main queue and treat the DLQ as a separate administrative queue.

Should I reprocess the DLQ automatically every night?

Tempting, and dangerous. A scheduled reprocess without a diagnosis turns permanent errors into an eternal small-hours loop and masks the alarm signal. If you want to automate something, automate retrying one specific error class already diagnosed as long-lived transient, with age and attempt limits, and leave the rest to a human.

The ideas that should stay with you

If six months from now you only remember five things from this guide, make them these:

- Classify every error as transient or permanent before deciding anything else; that distinction governs retries, DLQs and alerts.
- Retries have a limit and DLQs are terminal: a dead message waits for a human (or an archiver), it never cycles on its own.
- The payload travels untouched and the failure metadata travels alongside it; without a possible autopsy there is no fast diagnosis.
- The right DLQ alarm is zero-threshold on arrival and maximum age on waiting; depth is the least interesting metric.
- Reprocessing requires proven idempotency, a fixed root cause and rate-limited batches; in that order, no exceptions.

Everything else — quorum queues, retry topics, partial batches, archivers — is an implementation of these five ideas on whatever platform you happen to run. Platforms change; the ideas, so far, do not.