

# **Dominando o Transactional Outbox no CAP Java**



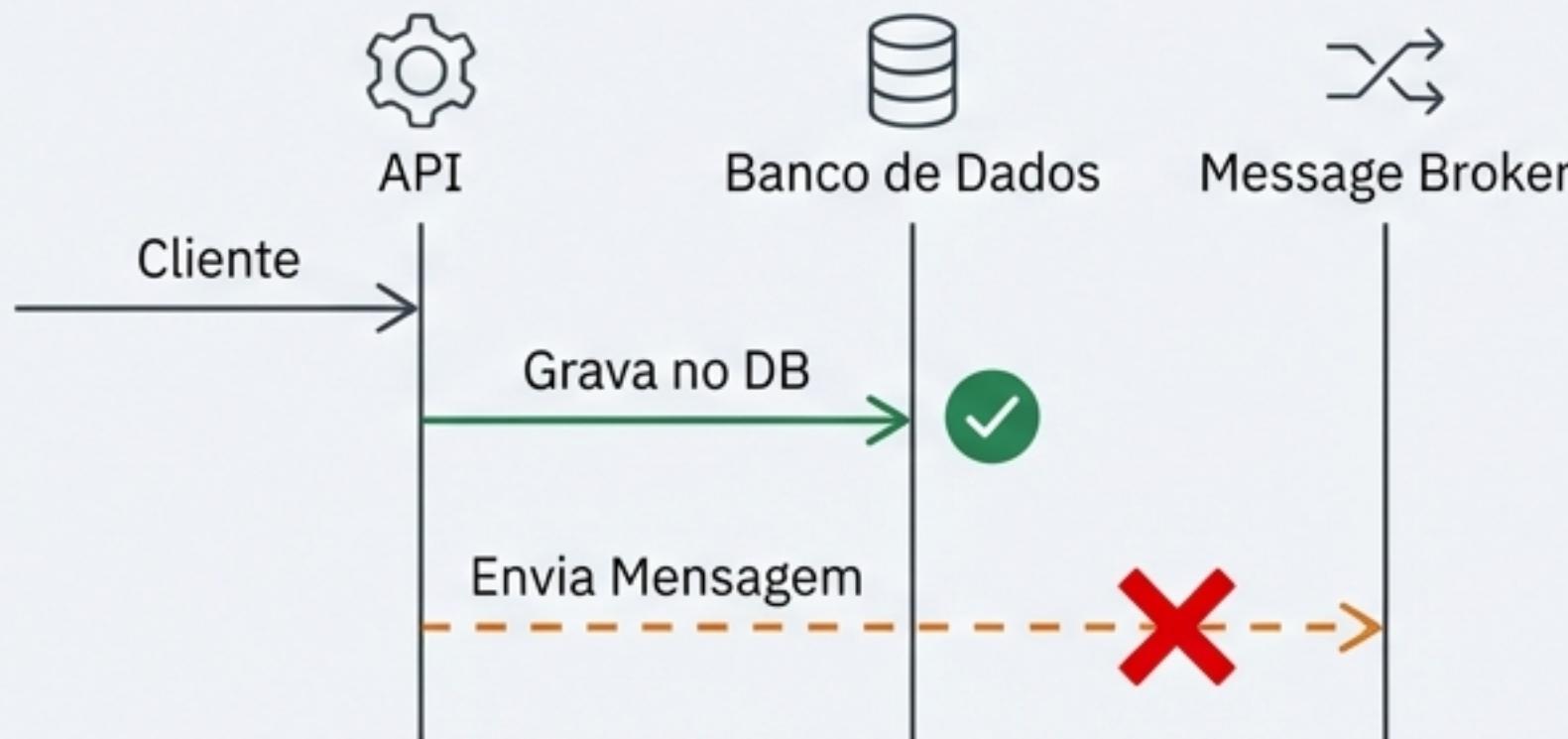
**Garantindo Mensagens Resilientes e Transações  
Consistentes em Arquiteturas Orientadas a Eventos**

# O Desafio Crítico: O Risco da Inconsistência Atômica

Em sistemas distribuídos, uma operação de negócio geralmente envolve duas ações: persistir dados no banco e emitir uma mensagem/evento. O que acontece se uma dessas ações falhar?

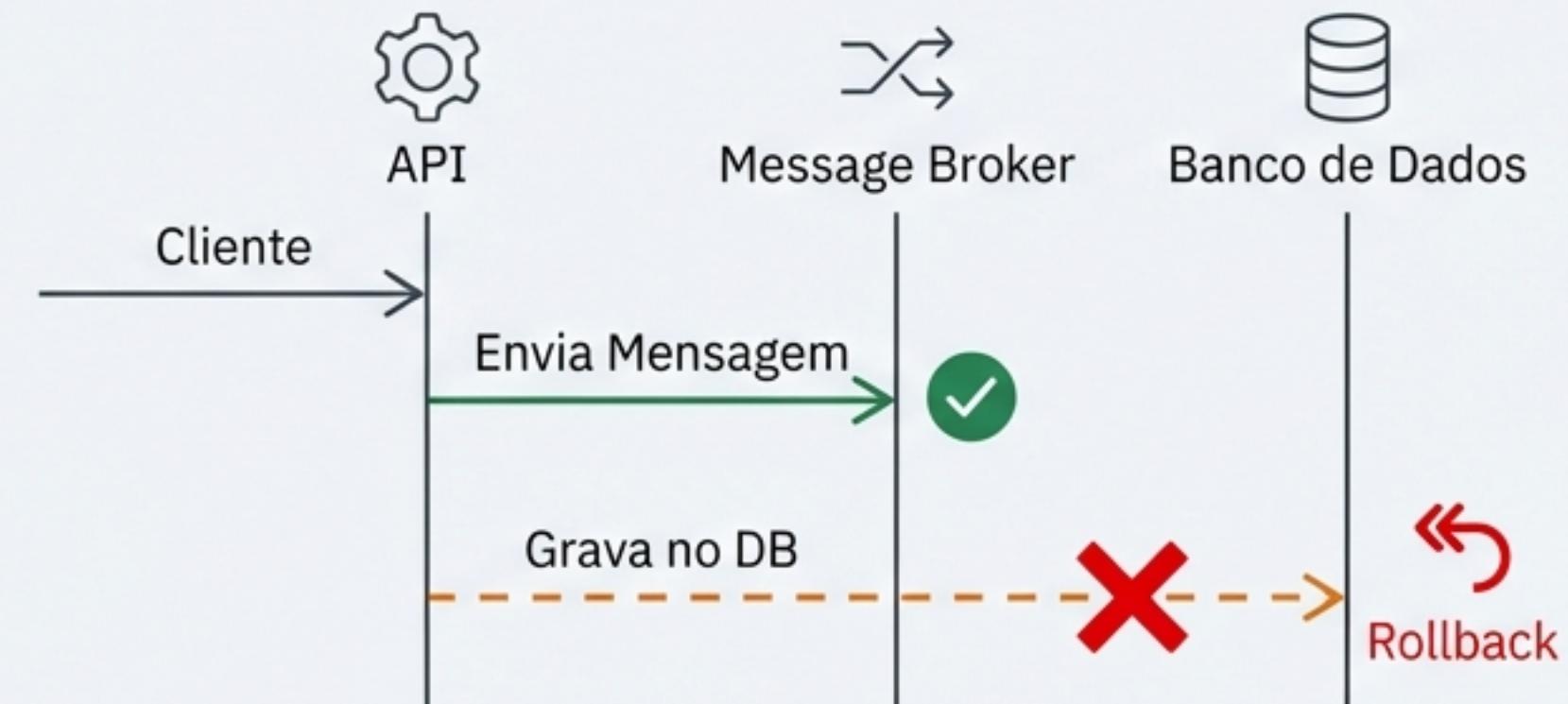
## Cenário 1: Mensagem Perdida

SUCESSO no Banco, FALHA no Broker



## Cenário 2: Mensagem Fantasma

SUCESSO no Broker, FALHA no Banco

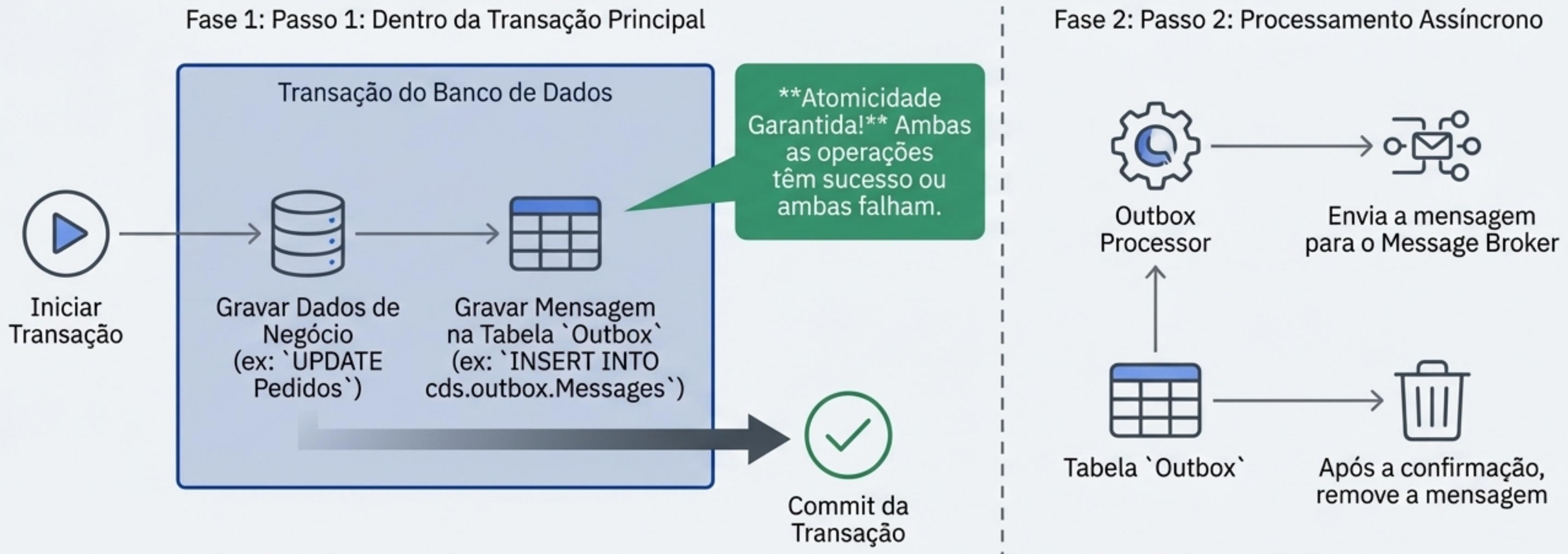


Dado salvo, mas o evento nunca é disparado. O sistema fica em um estado inconsistente e silencioso.

Evento disparado para uma transação que nunca foi concluída. Consumidores reagem a um estado que não existe.

# A Solução Elegante: O Padrão Transactional Outbox

O padrão une a escrita do dado de negócio e o registro da "intenção de enviar uma mensagem" na mesma transação atômica do banco de dados. A consistência é garantida no ponto de commit. A entrega da mensagem ocorre de forma assíncrona por um processo separado.



# Outbox no CAP Java: Escolhendo sua Estratégia

O CAP Java oferece duas implementações do padrão Outbox. A escolha correta depende do seu caso de uso e dos requisitos de resiliência.

Característica	In-Memory Outbox (Padrão)	Persistent Outbox 
Mecanismo	As mensagens são mantidas em memória até o commit da transação.	As mensagens são salvas em uma tabela dedicada no banco de dados. 
Resiliência	<b>Mensagens são perdidas</b> se a aplicação sofrer um crash após o commit, mas antes do envio. 	<b>Entrega garantida.</b> As mensagens sobrevivem a reinicializações e falhas da aplicação. 
Configuração	Nenhuma configuração necessária. É o comportamento padrão.	Requer ativação explícita e migração do schema do banco de dados. 
Caso de Uso	Desenvolvimento, prototipagem, eventos não críticos.	<b>Ambientes de produção</b> , eventos de negócio críticos.

# Passo a Passo: Ativando e Configurando o Outbox Persistente

Para habilitar a persistência, declare a dependência do serviço `outbox` e ajuste as configurações conforme necessário. O CAP adicionará automaticamente a entidade `cds.outbox.Messages` ao seu modelo.

## 1. Ativação no `package.json`

```
// package.json
"cds": {
  "requires": {
    "outbox": {
      "kind": "persistent-outbox"
    }
  }
}
```

## 2. Configuração no `application.yaml`

```
# application.yaml
cds:
  outbox:
    services:
      DefaultOutboxOrdered:
        maxAttempts: 10
      DefaultOutboxUnordered:
        maxAttempts: 10
```



**Atenção:** Após adicionar o Outbox Persistente, é essencial migrar o schema do banco de dados de todos os tenants para criar a tabela `cds.outbox.Messages`.

# A Anatomia do Outbox Persistente: O Ciclo de Vida da Mensagem

Após o commit da transação, um processador assíncrono assume. Ele tenta enviar cada mensagem, com uma lógica de retentativa com backoff exponencial em caso de falha.



# Cenário Avançado 1: Gerenciando Versões de Eventos em Deployments Contínuos

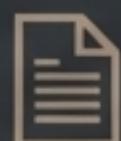
Em um deploy blue/green, versões antigas e novas da sua aplicação coexistem. Para evitar que uma instância antiga (consumidor) tente processar um evento de formato novo e incompatível, podemos **versionar** os eventos do Outbox. O coletor só processará eventos com versão menor ou igual à sua própria.

## 1. Ative o Filtro de Recursos do Maven no `pom.xml`

```
<!-- srv/pom.xml -->
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering> ----->
    </resource>
  </resources>
</build>
```

## 2. Use a Versão do Projeto no `application.yaml`

```
# application.yaml
cds:
  environment:
    deployment:
      version: ${project.version}
```



No startup da aplicação, você verá um log confirmando a versão configurada:  
Configured deployment version: [your.project.version]

## Cenário Avançado 2: Isolando Outboxes em um Banco de Dados Compartilhado

Quando múltiplos microserviços usam o mesmo banco de dados, seus outboxes padrão (`DefaultOutboxOrdered`, `DefaultOutboxUnordered`) colidiriam. A estratégia correta é desativar os padrões e **criar outboxes customizados e com namespaces únicos** para cada serviço.

**Passo 1:** Desativar os outboxes padrão para evitar conflitos

```
# application.yaml
cds:
    # Passo 1: Desativar os outboxes padrão para evitar conflitos
    outbox:
        services:
            DefaultOutboxUnordered.enabled: false
            DefaultOutboxOrdered.enabled: false
```

**Passo 2:** Definir outboxes customizados com nomes únicos por serviço

```
# Passo 2: Definir outboxes customizados com nomes únicos por serviço
Service1CustomOutboxOrdered:
    maxAttempts: 10
    ordered: true
Service1CustomOutboxUnordered:
    maxAttempts: 10
    ordered: false
```

**Passo 3:** Apontar as configurações de messaging e audit log para os outboxes customizados

```
# Passo 3: Apontar as configurações de messaging e audit log para os outboxes customizados
auditlog:
    outbox.name: Service1CustomOutboxUnordered
messaging:
    services:
        MessagingService1:
            outbox.name: Service1CustomOutboxOrdered
```

# Além de Mensagens: Usando o Outbox para Chamadas de Serviço CAP

Você pode encapsular qualquer chamada de serviço CAP (como uma chamada a um serviço OData remoto) em um Outbox. Isso transforma a chamada em uma operação assíncrona e resiliente, com a mesma garantia de retentativas.

## Uso Básico

```
OutboxService myCustomOutbox = ...;  
CqnService remoteS4Service = ....;  
  
// "outboxedS4Service" agora executa chamadas  
// de forma assíncrona via outbox  
CqnService outboxedS4Service =  
myCustomOutbox.outboxed(remoteS4Service);
```

## Lidando com Retornos (Recomendado)

```
// Use a interface AsyncCqnService para um  
// tratamento assíncrono mais adequado  
AsyncCqnService asyncOutboxedS4 =  
AsyncCqnService.of(remoteS4Service, myCustomOutbox);
```



**Atenção ao Comportamento Assíncrono:** Métodos em um serviço 'outboxed' que originalmente possuem um valor de retorno **sempre retornarão 'null'**. A execução real acontece em background. Use a API `AsyncCqnService` para um tratamento mais robusto.

# Estratégias de Resiliência: Lidando com Erros Recuperáveis vs. Irrecuperáveis

Nem todo erro é igual. Uma falha de rede é temporária e deve ser retentada. Um erro de '400 Bad Request' indica um problema nos dados e retentativas não resolverão o problema. Você pode controlar o comportamento do Outbox capturando exceções.

```
@On(service = "<OutboxServiceName>", event = "myEvent")
void processMyEvent(OutboxMessageEventContext context) {
    try {
        // Lógica de processamento que pode falhar (ex: chamada a serviço externo)
    } catch (Exception e) {
        if (isUnrecoverableSemanticError(e)) { // ex: verifica se é um HTTP 4xx
            // 1. Tome medidas específicas (ex: logar, notificar)

            // 2. Informe ao Outbox que a mensagem foi tratada e não deve ser retentada
            context.setCompleted(); ←
        } else {
            // 3. Para todos os outros erros (ex: rede), relance a exceção.
            // O Outbox fará a retentativa automaticamente.
            throw e; ←
        }
    }
}
```

3. Para todos os outros erros (ex: rede), relance a exceção.  
O Outbox fará a retentativa automaticamente.

# O Fim da Linha: Implementando uma Dead Letter Queue (DLQ)

Mensagens que excedem o `maxAttempts` são consideradas 'mortas'. Elas permanecem no banco de dados, mas não são mais processadas. Podemos criar um serviço CDS para expor e gerenciar essas mensagens de forma controlada.

```
// srv/outbox-dead-letter-queue-service.cds
using from '@sap/cds/srv/outbox';

@requires: 'internal-user' // Proteja o acesso a este serviço!
service OutboxDeadLetterQueueService {

    @readonly
    entity DeadOutboxMessages as projection on cds.outbox.Messages;

    // Ações para gerenciar as mensagens "mortas"
    actions {
        action revive(); // Reviver a mensagem para novas tentativas
        action delete(); // Deletar permanentemente a mensagem
    }
}
```

# Operando a DLQ: Reviver ou Remover Mensagens com Falha

A implementação das ações `revive` e `delete` permite que uma equipe de operações analise a falha e decida o que fazer: dar uma segunda chance à mensagem (após corrigir a causa raiz) ou removê-la em definitivo.

Implementação da Ação `revive()` 

```
@On
public void reviveOutboxMessage(DeadOutboxMessagesReviveContext context) {
    // ... (código para obter o ID da mensagem)
    Map<String, Object> key = analysisResult.rootKeys();

    Messages deadOutboxMessage = Messages.create();
    deadOutboxMessage.setAttempts(0); // Zera o contador

    db.run(Update.entity(Messages_.class).entry(key).data
        (deadOutboxMessage));
    context.setCompleted();
}
```

Implementação da Ação `delete()` 

```
@On
public void deleteOutboxEntry(DeadOutboxMessagesDeleteContext context) {
    // ... (código para obter o ID da mensagem)
    Map<String, Object> key = analysisResult.rootKeys();

    db.runDelete.from(Messages_.class).ById(key.get(Messages.ID));
    context.setCompleted();
}
```

# Observabilidade Essencial: Métricas do Outbox com Open Telemetry

Você não pode gerenciar o que não pode medir. O Outbox do CAP Java se integra nativamente com Open Telemetry, expondo KPIs vitais para monitorar a saúde do sistema, detectar problemas e entender o comportamento do fluxo de mensagens.

Métrica (KPI)	Descrição	Tipo
<code>com.sap.cds.outbox.coldEntries</code>	<b>Número de entradas "mortas"</b> que não serão mais retentadas. Um indicador chave para a DLQ. Exemcz de montata pro fluxo.	Gauge
<code>com.sap.cds.outbox.remainingEntries</code>	Número de entradas pendentes de entrega. <b>Mede o "tamanho da fila".</b>	Gauge
<code>com.sap.cds.outbox.maxStorageTimeSeconds</code>	Tempo máximo que uma mensagem permaneceu no Outbox. Ajuda a identificar gargalos.	Gauge
<code>'com.sap.cds.outbox.incomingMessages'</code>	Número total de mensagens que entraram no Outbox.	Counter
<code>'com.sap.cds.outbox.outgoingMessages'</code>	Número total de mensagens enviadas com sucesso pelo Outbox.	Counter

# Checklist para o Sucesso com o Transactional Outbox



**Sempre use o Outbox Persistente em produção.** A garantia de entrega é fundamental para eventos de negócio críticos.



**Configure `maxAttempts` de forma consciente.** O valor deve refletir a criticidade do serviço e a natureza das possíveis falhas.



**Isole outboxes por serviço** em cenários de banco de dados compartilhado para evitar conflitos.



**Implemente uma estratégia de DLQ.** Tenha um plano para analisar e gerenciar mensagens que falham persistentemente.



**Monitoreativamente as métricas de telemetria.** Fique de olho em `coldEntries` e `remainingEntries` para detectar problemas proativamente.

v1.2.3

**Utilize o versionamento de eventos** em ambientes com deploy contínuo para garantir compatibilidade retroativa.

# O Outbox como Fundamento da Resiliência

O Transactional Outbox não é apenas um recurso; é um padrão arquitetural que desacopla a lógica de negócio da entrega de mensagens, trocando a comunicação síncrona e frágil por um modelo assíncrono e robusto. Ao dominá-lo, você constrói sistemas que não apenas funcionam, mas que sobrevivem e se recuperam de falhas inerentes ao mundo distribuído.

