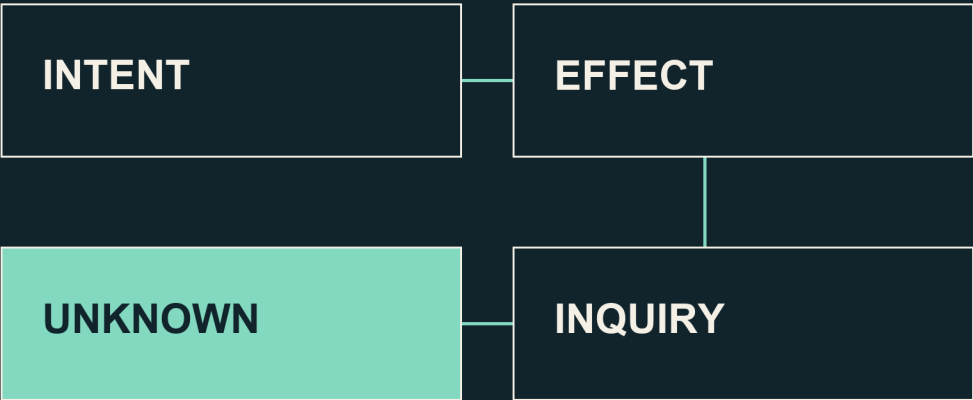


# Payment Reliability Engineering

Prevent Duplicate Charges,  
Recover Unknown States,  
and Reconcile Money Movement



# Payment Reliability Engineering

FREE SAMPLE — opening and Chapters 1–2. Prevent Duplicate Charges, Recover Unknown States, and Reconcile Money Movement

Xiaoqing Wang • Edition 1.0 • October 1, 2026

Copyright © 2026 Xiaoqing Wang. All rights reserved. AI-assisted development is disclosed in the edition note. Provider requirements are dated and qualified. Examples are learning material, not production certification.

## Contents

|                                                          |    |
|----------------------------------------------------------|----|
| How to use this field manual                             | 4  |
| 1. Trace the failure before choosing the fix             | 5  |
| 2. Model intent, operations, and observations separately | 8  |
| What the full manual adds                                | 11 |

# How to use this field manual

A customer has paid. Your order still says pending. The customer taps Pay again. The second attempt succeeds too. A support agent refunds one charge while a fulfillment worker starts shipping. Tonight's reconciliation finds two captures, one refund request, and an order whose status nobody trusts.

This is an illustrative scenario, not a report of a historical incident. It is the running example for this book because the individual components can all be functioning as designed while the business result is wrong.

The promise of this manual is practical: help engineers prevent, diagnose, and recover from payment failures that can cause lost revenue, duplicate charges, inconsistent orders, and expensive production incidents. It assumes you already build backend services and can work with transactions, queues, and HTTP APIs. It does not teach card processing from scratch, provide accounting advice, or replace a processor's integration contract.

Read Chapters 1-3 before changing retry behavior. Use Chapters 4-7 when designing event processing, money records, and reconciliation. Keep Chapters 8-10 beside the incident channel and release checklist. The worksheets are deliberately small enough to use in a design review.

Three kinds of statements appear here. **Engineering patterns** are recommendations whose assumptions are stated. **Documented examples** identify a provider and a source number. **Illustrative exercises** are invented situations used to reason about failure; they make no claim about actual losses or customer incidents. Provider documentation was checked on October 1, 2026. Recheck the linked contract before implementing a provider-specific detail.

The examples are architecture sketches and pseudocode unless explicitly identified as runnable. They omit authentication, complete schemas, legal retention policy, and operational deployment. A correct sketch is useful; a sketch that pretends to be a complete payment service is dangerous.

# 1. Trace the failure before choosing the fix

## A successful API call is only one fact

In the running scenario, the processor knows that a capture happened. The order database knows that it has not applied a payment observation. The browser knows that its request stopped waiting. None of those facts alone establishes the whole business state.

Write the incident as a chain of decisions, not a list of broken components:

1. A checkout creates payment attempt A for order O.
2. The processor accepts A, but the application does not retain the response.
3. A webhook is delayed, lost at your ingestion boundary, or waiting in a failed worker.
4. The order projection remains pending.
5. The user sees a retry action that creates attempt B.
6. B produces another capture for the same intended purchase.
7. Refund and fulfillment decisions read different snapshots.
8. Reconciliation observes money movement that the order projection cannot explain.

Only step 3 concerns webhook delivery. Improving webhook reliability will help, but it will not make step 5 safe. A service that treats missing evidence as evidence of failure can duplicate a charge even with a healthy webhook integration.

Start every reliability review by identifying the **decision that can create another financial effect**. Put the strongest control there. A pending order needs an inquiry path before it needs a fresh payment path. A refund button needs a reservation of refundable value before it needs a request loop.

## Draw the authority map

| Question                                       | Primary evidence                                  | What it does not prove                     |
|------------------------------------------------|---------------------------------------------------|--------------------------------------------|
| Did our business authorize this operation?     | Durable internal operation record                 | That the processor executed it             |
| Did the processor accept or complete it?       | Authenticated provider response or queried object | That fulfillment happened                  |
| Did we fulfill the purchase?                   | Fulfillment record with its own identity          | That funds were settled                    |
| Did settlement reach the expected destination? | Applicable processor/bank reports                 | That every order was represented correctly |

The authority is specific to a question. There is no universal master status. A processor is authoritative about its own object; it does not decide whether your warehouse shipped the right item. Your ledger records your accounting interpretation; it does not make a missing bank deposit appear.

A useful architecture preserves evidence and builds projections from it. A projection such as `order.payment_status` is convenient for a UI. Its convenience must not silently grant it authority to create a new charge.

## Replace "exactly once" with a bounded claim

Ask what is unique, over which scope, and for how long. A queue might deduplicate one message identifier. A database unique constraint might protect one internal operation. A provider idempotency key might protect one endpoint for a limited retention period. These controls cover different boundaries.

The target is not a slogan. It is a demonstrable invariant: **one authorized purchase decision must not create more than its authorized economic effect, even when requests or observations repeat**. Achieving it requires coordinated operation identity, provider semantics, database constraints, and reconciliation.

A local transaction cannot atomically commit inside an unrelated processor. The gap must be represented. Hiding it inside a timeout handler does not remove it.

## The failure-chain worksheet

For one real checkout flow, fill in these fields:

- Business intent: what did the customer agree to buy, at what amount and currency?
- Effect boundary: which command can create a charge, capture, refund, or transfer?
- Durable identity: which record survives a browser refresh and a worker restart?
- Observation gap: where can execution succeed without your database learning it?
- Unsafe inference: which code path interprets that gap as failure?
- Recovery owner: who resolves a gap that outlives automatic inquiry?
- Closure evidence: what proves that both money and fulfillment are explained?

A review is incomplete when the last two answers are "the queue retries" and "the order is green." A queue is a transport mechanism. A green projection is a presentation state. Neither proves closure.

**First implementation action:** find every user-facing and worker-facing path that creates a fresh payment attempt. For each, identify how it excludes an unresolved earlier attempt for the same intent. This is usually more valuable than adding another retry to the webhook consumer.

## 2. Model intent, operations, and observations separately

### One status column cannot carry three histories

A purchase intent answers what the business authorized. An operation answers what you tried to do. An observation answers what you learned. Store them separately even if they live in one database.

A compact conceptual model is:

```
purchase_intent
  intent_id, order_id, amount_minor, currency
  intent_version, authorized_effect_limit

payment_operation
  operation_id, intent_id, kind, request_digest
  provider_scope, idempotency_key, provider_object_id
  execution_state, inquiry_due_at

payment_observation
  observation_id, source, source_object_id
  received_at, effective_at, authenticated_evidence_ref

order_projection
  order_id, applied_intent_version, payment_summary
```

Use exact integer minor units only after establishing the currency's exponent and the API's amount contract. Do not assume every currency has two decimal places. Some providers also have endpoint-specific conventions. Keep currency on every amount; an integer without a currency is not a money value.

provider\_scope includes the merchant/account and environment needed to interpret an object. Sandbox and production identifiers can look alike. Two merchants can use the same internal reference. An identifier without its scope is incomplete evidence.

### Make unknown state explicit

The following is an application model, not a universal provider state machine:

| Execution state     | Evidence                                              | Permitted next decision                               |
|---------------------|-------------------------------------------------------|-------------------------------------------------------|
| Prepared            | Durable command exists;<br>no send has started        | Dispatch under exclusive<br>operation ownership       |
| Dispatching         | Sending may have begun                                | Assume execution may<br>have occurred after a crash   |
| Unknown             | No conclusive outcome is<br>retained                  | Inquiry or contract-safe<br>same-operation retry      |
| Accepted            | Provider acknowledged an<br>operation                 | Follow that operation's<br>completion contract        |
| Completed           | Operation-specific<br>completion evidence             | Apply relevant business<br>consequence once           |
| Definitively failed | Evidence excludes<br>completion under the<br>contract | Authorize replacement only<br>through business policy |
| Review required     | Automatic resolution<br>budget exhausted              | Owned investigation;<br>preserve uncertainty          |

"Accepted" and "completed" are different. An asynchronous refund acknowledgement can mean only that a request entered processing. Likewise, authorization, capture, and settlement are separate stages. Do not name all of them paid and then attempt to infer the missing distinctions during an incident.

Prepared must be committed before dispatch. Mark dispatching durably before the external call. A worker crash after that write may create a false unknown, but that is safer than assuming a possibly sent command was never sent. The resolver decides whether a repeat send is contract-safe.

## Define consequences independently

A successful authorization may justify reserving inventory. A successful capture may justify fulfillment for your product. A settlement report may justify cash reconciliation. These are business decisions, not interchangeable provider statuses. Document which evidence each consequence requires.

An expired order can later receive authentic evidence of a capture. The expiration remains a fact; it does not erase the money movement. Route the

late capture through a policy such as delayed fulfillment or refund review. Silently dropping it because the order is terminal leaves the customer charged and the system unable to explain why.

Similarly, a payment can be fully refunded while the fulfillment record still proves delivery. Preserve both facts. A destructive transition from paid to refunded can erase the event that explains why the goods shipped.

## Invariants that can be checked

Prefer scoped, measurable rules:

- Every outgoing financial command has a durable operation and an immutable request digest.
- Reuse of an operation identity with a different amount, currency, merchant, or semantic purpose is rejected.
- Only one active creation decision exists for a single-pay purchase intent; unresolved operations consume that slot.
- A completion observation applies each internal consequence at most once.
- Confirmed refunds plus reserved unresolved refunds do not exceed the eligible captured amount under the product's refund policy.
- Every unexplained capture is assigned an owner and resolution deadline.
- Every balance correction retains its prior evidence and an explicit reason.

These are engineering invariants. Split tender, multiple captures, subscriptions, tips, and partial shipment require a richer intent model. Do not force them into "one charge per order"; define authorized amounts and allocations explicitly.

## Review exercise: stale success

Illustrative exercise: a worker reads a payment as pending; another worker learns that it succeeded; the first worker later writes failed because its request timed out. A rank table that places success above failure might prevent that overwrite, but ranking alone fails for refund and dispute facts that are not points on the same line.

Use operation-specific transitions and separate money dimensions. If an observation cannot be applied safely, retain it for inquiry. A reducer should be conservative about consequences, not destructive about evidence. The

evidence store may accept a new fact even when the order projection refuses to transition.

## What the full manual adds

The full edition continues into retry decisions, authenticated durable webhook ingestion, money lineage, unknown-state recovery, reconciliation, refund races, incident containment, and a failure-oriented release review. The incident and production-readiness artifacts remain in the paid edition.

This sample contains the opening and the first two complete chapters, selected to demonstrate the manual's reasoning before purchase. The sample is independently usable; the paid edition extends this reasoning into release and recovery artifacts.