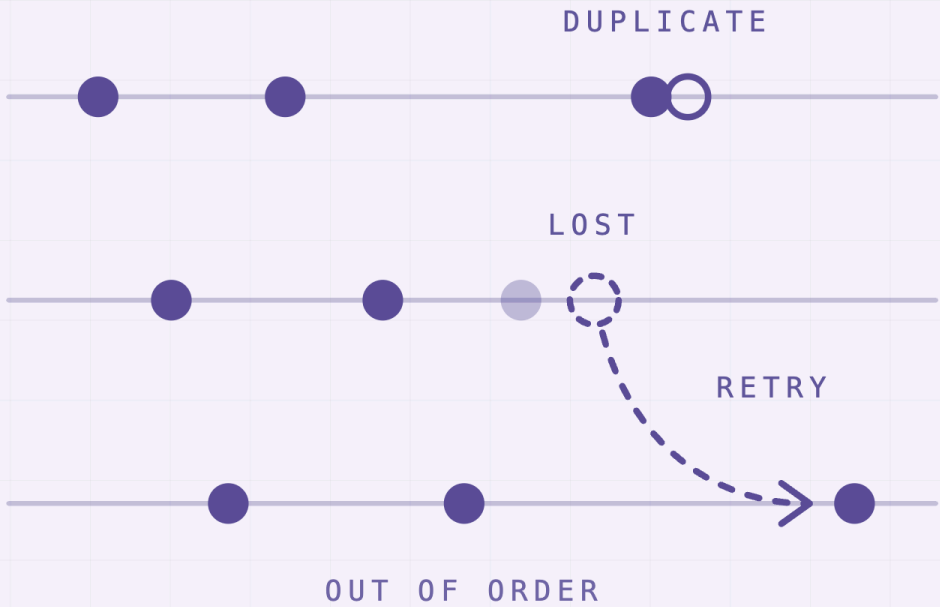




Designing Reliable Event- Driven Systems

Designing Reliable Event-Driven Systems

Messaging, retries, idempotency, outbox, ordering and schema evolution

Muhammet Şafak

Contents

Preface	5
PART I — FOUNDATIONS	12
Chapter 1 — The Lie of the Happy Path	13
Chapter 2 — Messaging Semantics	24
Chapter 3 — A Practical Failure Model	41
PART II — PUBLISHING RELIABLY	59
Chapter 4 — The Dual-Write Problem and the Transactional Outbox	60
Chapter 5 — Relays, Polling and Change Data Capture	78
PART III — CONSUMING RELIABLY	98
Chapter 6 — Retries Done Right	99
Chapter 7 — Idempotency	122
Chapter 8 — Poison Messages and Dead Letters	144
Chapter 9 — Ordering	165
PART IV — WORKFLOWS AND CONTRACTS	185
Chapter 10 — Sagas and Long-Running Workflows	186
Chapter 11 — Event Design	206
Chapter 12 — Schema Evolution	225
PART V — OPERATING AND PROVING	245
Chapter 13 — Observability and Reconciliation	246
Chapter 14 — Recovery, Disasters and Broker Migration	267
Chapter 15 — Testing Event-Driven Systems	291
Chapter 16 — A Reliability Checklist and Closing Thoughts	308
Appendix: Glossary and Pattern Quick Reference	320

Preface

Imagine a Tuesday afternoon at Parcelly, a marketplace that does not exist. The checkout flow has been healthy for months. A buyer places an order, the Orders service writes it to its database, and an `OrderPlaced` event goes out to the message broker. Payments authorizes the card. Inventory reserves the stock. Shipping schedules the parcel. Every dashboard is green.

Then a deploy rolls through Orders. Each instance handles an order the same way: it commits the order to its database, then publishes `OrderPlaced` to the message broker. For a few seconds, instances are shutting down while others start. One instance commits an order and is stopped before the publish call. The order is safely stored, and the event never leaves. Nothing crashes loudly and no alert fires. By the next morning, a buyer is looking at an order marked as placed, but Payments never authorized a card and Inventory never reserved a single item. Orders believes the work is under way, and the rest of the system has never heard of it.

Nobody wrote a bug in the usual sense. Every line of code did what it said. The system failed in the space between the lines: between a database commit and a network call, between a message being delivered and being understood, between "we sent it" and "they did it."

That space is what this book is about.

Why this book exists

Event-driven architecture has a reputation problem, and it is earned from both directions. Its advocates describe loosely coupled services that scale independently and evolve without coordination. Its critics describe distributed spaghetti, where nobody can say what happens when an order is placed. Both are describing the same system on different days.

What separates the two days is rarely the choice of broker, the framework, or even the architecture diagram. It is a short list of unglamorous questions. What happens if this message is delivered twice? What happens if it never arrives? What happens if it arrives after the message that was supposed to follow it? What happens if the consumer fails every single time? What happens when the producer needs to change a field and a hundred consumers are still running the old code?

Those questions are the thesis of this book. Reliable event-driven systems are built around **failure semantics**, not message brokers: for a given seam, what can fail, how you would observe it, and which guarantee survives it. A seam is any place where one component hands work to another, such as a database commit followed by a publish, or a delivery followed by an effect. The patterns here exist because of the network and the database, not because of any one product, and a broker is one more component with seams of its own.

Teams usually meet these questions one at a time, in production, in the order the system happens to fail. Each answer is learned separately, and the lessons rarely end up in one place. The documentation of a broker tells you how to publish and consume. It is much quieter about what a correct service looks like around those two verbs.

This book tries to be that place. It is a guide to the patterns that make event-driven systems reliable: the transactional outbox, idempotent consumers, retries with backoff and jitter, backpressure, dead-letter queues, ordering strategies, sagas, careful event design, schema evolution, observability, recovery from lost data and a change of broker, and testing. I have tried to treat each as a decision with a price instead of a rule to follow. A pattern that is right for Parcelly's payments may be wasteful for its notifications, and the book will say so when it matters.

I also want to be plain about what the book is not. It is not a tutorial for any specific broker. It is not a catalog of every enterprise integration pattern ever named. It does not promise that following it will make your system correct. Reliability is not a property you add; it is a set of failures you have decided, deliberately, how to handle, and a set you have decided, deliberately, to accept. My aim is to make those decisions visible.

Who this book is for

The book is written for **backend and platform engineers** who build or operate services that communicate through events, and for the **tech leads** who have to decide how those services should behave when things go wrong. If you have ever stared at a dead-letter queue that grew overnight, argued about whether a consumer needs to be idempotent, or inherited a topic whose events nobody can fully explain, you are the reader I have in mind.

You do not need to be an expert in messaging. You should be comfortable with the following:

- Relational databases and transactions: what a commit is, what a unique constraint does, and what isolation means in rough terms.
- Services that talk over a network, and the basic vocabulary of timeouts and HTTP-style request and response.
- The general idea of a message broker: something that accepts messages from producers and hands them to consumers.

You do not need experience with any particular product. When a chapter names Kafka, RabbitMQ or Amazon SQS, it does so as an example of a family of systems, and describes behavior that holds in general instead of in a given release. If you have used none of them, the book will still make sense. If you have used one deeply, you will find that some of what you know is specific to it, and the book will try to mark which parts.

If you are a tech lead who does not write code every day, you can read the opening scenario, the trade-offs section and the key takeaways of each chapter and get most of the decision-making value. The pseudocode is there for people who need to see where exactly the transaction boundary sits.

How the book is organized

The book has five parts, and they are meant to be read in order the first time and used in any order afterwards.

Part I, Foundations (Chapters 1 to 3). We start by taking the happy path apart. Chapter 1 shows how a flow that looks simple turns out to hide a dozen decisions. Chapter 2 pins down what messaging guarantees actually mean, and why at-least-once delivery is the realistic baseline. Chapter 3 builds a practical failure model, a small vocabulary for the ways things break, from timeouts and crashes to messages that always fail and the loss of a whole store. The rest of the book uses it as its shared language.

Part II, Publishing Reliably (Chapters 4 and 5). Chapter 4 addresses the dual-write problem, the exact gap in the opening scene, and introduces the transactional outbox. Chapter 5 covers the machinery that moves outbox rows to the broker, comparing polling relays with change data capture, and what each costs you.

Part III, Consuming Reliably (Chapters 6 to 9). This is the longest part, because consumers are where most duplicate effects, stuck queues and lost updates happen. Chapter 6 covers retries: when to retry, when not to, why backoff needs jitter, and how to keep a consumer from being overloaded by its own retries. Chapter 7 is about idempotency and the inbox. Chapter 8 deals with poison messages and dead letters. Chapter 9 turns to ordering, and how little of it you can usually afford to demand.

Part IV, Workflows and Contracts (Chapters 10 to 12). Chapter 10 introduces sagas for workflows that span services and cannot share a transaction. Chapter 11 is about designing events themselves: what goes in them, what stays out, how they are named, and how an envelope survives being written and read in several languages. Chapter 12 covers schema evolution, the quiet discipline that lets producers and consumers change at different speeds.

Part V, Operating and Proving (Chapters 13 to 16). Chapter 13 is about seeing what your system is doing, and about reconciliation, the habit of checking one system's view against another's. Chapter 14 is about recovering from lost data, a lost region and a change of broker. Chapter 15 covers how to test code whose worst bugs only appear under duplication, delay and reordering. Chapter 16 gathers everything into a reliability checklist and closes the book.

An appendix holds a glossary and a quick reference for the patterns, so you can look something up without rereading a chapter.

Every chapter from 1 to 15 follows the same shape. It opens with a short scenario, explains why the problem is hard, presents the pattern, and then moves to two sections I consider the most valuable: **Trade-offs and costs**, which says when not to use the pattern, and **Common mistakes**. Each chapter ends with five to seven key takeaways. If you are short on time, those three parts will take you further than you might expect.

You can read the chapters selectively. If your pain today is a growing dead-letter queue, go straight to Chapter 8, and follow the references back to Chapters 6 and 7 as needed. If you are designing a new service, I would read Part II before you write the first line of publishing code, because the outbox is far easier to start with than to retrofit.

Meet Parcelly

Throughout the book, one fictional system carries the examples: **Parcelly**, an online marketplace where independent sellers list goods and buyers order them. It has six services.

- **Orders** owns the order lifecycle and is where most workflows begin.
- **Payments** authorizes and captures money, and talks to an external payment provider.
- **Inventory** tracks stock and holds temporary reservations.
- **Shipping** schedules shipments and hands parcels to carriers.
- **Notifications** sends email, push messages and SMS, and must never block anything else.
- **Ledger** keeps an append-only, double-entry record of money movements.

Each service owns its own relational database and reads nobody else's. They communicate through a message broker by exchanging events with names such as `OrderPlaced`, `PaymentAuthorized`, `StockReserved`, `PaymentFailed` and `ShipmentScheduled`. The events follow a consistent shape, an envelope that carries an identifier, a type, a schema version, correlation fields and a payload, so that the chapters can talk about them in one voice.

I chose a marketplace on purpose. Taking money, reserving goods and moving parcels is a domain where the cost of a lost message or a duplicate effect is felt quickly. A duplicate email is an annoyance. A duplicate capture is a refund, a support ticket and an apology. A lost reservation is an oversold item. That gradient, from harmless to expensive, is exactly what you need to decide how much reliability machinery each flow deserves.

Parcelly is fiction, and so are its incidents. The scenarios that open each chapter are invented to illustrate a failure shape, and they are always framed as fiction. I will use the first person for judgment, such as "I would add a unique constraint here" or "in my view this is not worth the complexity", but I will not dress up reasoning as war stories. You should be able to trust that when the book says something happens, it is because of how these systems work, not because of an anecdote.

The book is also deliberately **broker-agnostic**. Where a point depends on whether your broker is log-based, with consumers tracking an offset in an ordered partition, or queue-based, with messages acknowledged and redelivered one at a time, I will say which. Where it does not matter, I will not pretend otherwise.

How to read the code

The code in this book is **pseudocode**. It does not belong to any language, it will not compile, and nothing in it calls a real library. This is intentional. The point of each snippet is to show where a transaction begins and ends, where a message is acknowledged, and what happens when a step fails halfway. Those facts are the same in every language, and syntax would only get in their way.

Blocks are short, at most 30 lines. Where a table schema helps, it is written in SQL-like notation. Diagrams use Mermaid, and each is introduced by a sentence telling you what to look for. If you translate a pattern into your own stack, treat the pseudocode as a specification of ordering and atomicity, not as an implementation.

A note on certainty

I will make recommendations in this book, and I will hold some of them firmly. I would not, for example, publish events with a plain dual write when the loss of an event matters to the business. Other recommendations are matters of taste or of context, and I will try to say which is which.

There will also be places where the honest answer is "it depends, and here is on what." Retry budgets, dead-letter retention, partition counts and the right level of ordering all depend on facts about your business and your team that I cannot see. In those places I will name the variables instead of inventing a number. If you finish a chapter knowing which questions to ask in your own system, I will count that as success.

Reliability work has an unfortunate property: when it succeeds, nothing visible happens. Orders get fulfilled, ledgers balance, and no one writes a message to thank you. I hope this book helps you build that kind of quiet.

Let us begin with the happy path, and with the reasons not to trust it.

PART I

Foundations

Chapter 1 — The Lie of the Happy Path

Imagine Parcelly in its first year. Orders calls Payments over HTTP, waits for the answer, then calls Inventory, waits again, then calls Shipping. The code reads top to bottom, a failure is an exception, and a buyer who sees an error page can try again. It is slow when the payment provider is slow, and it falls over when Inventory is down, but nobody is confused about what happened.

Then the team decides to move to events. Orders will publish `OrderPlaced` to a message broker, and the other services will react on their own schedule. The demo is beautiful. A buyer clicks "Place order", and within a second stock is reserved, the card is captured, the order is confirmed and a shipment is scheduled. Everyone applauds.

Now picture the first month in production. A buyer is charged, but the order stays in "pending" forever. Another order is shipped twice. A third has a ledger entry with no matching payment, and nobody can say which service is right. Support asks, "what happened to order 81724?", and the honest answer is that four services each know a piece of it and none of them knows all of it.

Nothing in the demo was wrong. It was incomplete, because a demo is a story in which every step succeeds. The question this chapter answers is what the team gave up when it left request-response, and what "reliable" has to mean before anyone starts building it back.

The problem, and why it is hard

The happy path is the sequence of events that occurs when every component is up, every network call arrives, every message is delivered once and in order, and every consumer finishes what it starts. It is the path you

draw on a whiteboard and the path a demo exercises. It is also the rarest path in a system that runs for months.

The failures are not exotic. Deploys restart processes. Connections drop in the middle of a write. A broker fails over to a replica. A consumer takes longer than expected and its message is handed to someone else. Each of these happens somewhere in a busy system every day. The usual shape of a bad production month is a handful of ordinary events meeting code that was written for the happy path.

What makes this hard is that event-driven systems remove the thing that made failures visible. In a synchronous call the caller waits, and the outcome returns on the same connection: a result or an error, next to the code that caused it. In an event-driven design the producer hands a message to the broker and moves on. If something goes wrong three services and ten seconds later, the producer is long gone, and the evidence is spread across logs nobody has correlated yet.

The obvious response is to make each piece reliable: a good broker, well-tested services, retries on every call. Each helps, and none is sufficient, because the failures that hurt live in the gaps between components. A message written to the database but never published, or processed but never acknowledged, is not a bug in either component. It is a property of the seam. Most of this book is about seams.

For every seam, the book asks the same three questions: what can fail, how would you observe it, and which guarantee survives it. I will call the answers the seam's **failure semantics**. A broker's documentation gives you the semantics of the broker's own seams. The seams around it, between your database and your publish call or between a delivery and its effect, are yours to work out, and Chapter 3 shows how.

Synchronous and asynchronous coupling

The move to events is usually described as "decoupling", a word that promises more than it delivers. It is more accurate to say that coupling changes shape.

A synchronous call couples two services in **time**, in **availability** and in **knowledge**. The callee must be running now, the caller's latency includes the callee's, and the caller knows who the callee is and what it will answer. The benefit is a simple failure model: the caller knows the call failed and can decide immediately to retry, fall back or report an error. The cost is that the chain is only as available as its weakest link. A slow carrier API becomes a slow checkout.

A message through a broker removes the coupling in time and availability. Orders can accept an order while Inventory restarts, because the broker holds the message until Inventory returns. A new consumer, such as an analytics service, can start listening without Orders changing. That gain is real, and it is the reason to adopt the style.

But the coupling has moved, not vanished. Services are now coupled through the **contract of the event**: its name, its fields and its meaning. They are coupled through the **broker's behavior**: what it promises about durability, ordering and delivery. And they are coupled through assumptions about each other's timing that nobody wrote down. Chapter 11 and Chapter 12 deal with the contract, and Chapter 2 deals with the broker.

In short, synchronous systems fail loudly and in a chain. Asynchronous systems fail quietly and in pieces.

The price of temporal decoupling

Temporal decoupling means the producer and the consumer do not have to be active at the same time. It is the main thing you buy with events, so it is worth being exact about the price.

The producer no longer learns the outcome

When Orders called Payments synchronously, it learned whether payment worked before it replied to the buyer. With events, Orders publishes `OrderPlaced` and replies "received". Whether payment succeeds is a question for later, and the answer arrives, if it arrives, as another event. The order now has an intermediate state, "placed but not yet confirmed", and the system has to decide what the buyer sees while it lasts and what happens if it lasts longer than planned.

Nobody owns the whole flow

In the synchronous version, one function contained the sequence, and you could read the business process in one place. In the event-driven version, the process is the sum of reactions scattered across services. Payments reacts to `OrderPlaced`, Orders reacts to `PaymentAuthorized` and `StockReserved`, Shipping reacts to `OrderConfirmed`. The flow exists only as an emergent property, and nobody can point to where it is defined.

That is fine when everything works and expensive when something does not, because "where did this order get stuck?" has no single place to look. Chapter 10 returns to this with sagas, and Chapter 13 covers what you need to observe to answer the question at all.

Failures stop being exceptions

In a call chain, an error propagates up the stack until something handles it. In a message flow there is no stack. A failing consumer can acknowledge the message and lose it, refuse it and have it redelivered, or park it for a human. Each is a policy decision with consequences for services that never see the error. In my view, error handling in a synchronous system is a matter of control flow, and in an asynchronous system it is a matter of design, because the handling is spread across time, processes and sometimes teams.

Time becomes a variable

Between "published" and "processed" there is a gap with no upper bound unless you give it one. During the gap a product may sell out, a buyer may cancel, a price may change. A consumer must decide whether an event that was true when published is still meaningful when handled.

Note: This is why events are named in the past tense. `OrderPlaced` is a fact about a moment that has passed. A consumer that treats it as an instruction to act right now has quietly assumed that no time went by. A command, such as `CapturePayment`, asks one service to do something and may be refused; an event reports something that already happened. Chapter 11 develops the distinction.

What "eventually" really means

Most explanations of **eventual consistency** are one sentence: if no new updates are made, all replicas converge to the same value. That is accurate and nearly useless to someone building a checkout flow, because it leaves out how long "eventually" is and what to do while you wait.

Eventually is not a duration

A system that converges in 200 milliseconds and a system that converges after a nightly batch are both eventually consistent. For a buyer staring at a confirmation page, those are different products. I would ask for a number for every hop that matters, even a rough one. Say Orders must show a confirmed order within a few seconds of capture, and Notifications may take a minute. These figures are illustrations and yours will differ, but they turn "eventually" into a requirement that can be tested and alerted on. A delay with no bound is not a design choice. It is the absence of one.

Eventually is not a guarantee

Convergence happens only if every message that matters is eventually delivered, every consumer processes it, and no step silently drops it. If one message is lost between a database commit and a broker publish, the views do not converge. They stay different until a person or a repair job notices. "Eventually" is a conditional promise, and the condition is that the system has mechanisms making loss detectable and repairable. Choosing a broker does not provide them. Parts II and III of this book prevent most loss, and Chapter 13 detects and repairs the rest.

What to do while you wait

During the window of inconsistency, other readers see an intermediate state. Consider this snapshot of one order, a moment after the buyer was charged:

```
// PaymentCaptured not yet consumed
orders.status      = "CAPTURE_REQUESTED"
payments.status    = "CAPTURED"
inventory.state     = "RESERVED"
shipping.shipment  = none
// LedgerEntryPosted not yet emitted
ledger.entries     = none
```

Each service is right about its own data. The combination looks inconsistent and is normal for a few seconds. It becomes a problem only when it stays that way. The useful operational question is not "are the services consistent right now?" but "how long has this order been in this state, and is that longer than we allow?"

A working definition of reliability

"Reliable" is used so loosely that two engineers can agree a system should be reliable and mean opposite things. I want a definition that can be checked. For event-driven systems I use four properties. They say what is being protected; the three questions above are how you check a seam against them.

No loss. Every event that represents a business fact reaches every consumer that needs it. Loss is the most damaging failure because it is silent: the system keeps running, and the only symptom is an absence nobody notices until a buyer complains. No loss is a property of the whole path, not of the broker. A durable broker does not help if the producer never managed to store the message, or if the consumer acknowledged before finishing the work. Chapter 2 examines what brokers promise, and Chapter 4 and Chapter 5 address the producer side.

No harmful duplication. Some messages will be delivered more than once. This is not a defect of a particular broker but the price of not losing messages: when a sender cannot tell whether a message arrived, the safe move is to send it again. A reliable system does not try to prevent duplicates at the transport. It makes them harmless. The word "harmful" matters. Two identical emails are an annoyance, while a second capture of the same payment is a financial incident. Chapter 7 covers idempotency, the main tool.

Bounded delay. Every event is processed within a time the business tolerates, or the system says loudly that it was not. A message delivered correctly three days late is, for many purposes, a lost message. Bounded does not mean fast. It means a limit exists, someone chose it, and a breach produces an alert. A payment confirmation might need seconds; a weekly statement might tolerate hours. Chapter 6 treats retries and backoff, and Chapter 13 treats lag and alerting.

Explainability. For any order, at any time, someone can reconstruct what happened: which events were produced and consumed, what each service decided and why. This is the least glamorous property and it decides whether the other three can be kept. A system that loses a message but can show exactly where is repairable. A system that misbehaves in a way nobody can trace is not. Chapter 13 covers the identifiers, records and comparisons this needs.

The four pull against each other, which is why a definition with only one of them is dangerous. Chasing no loss with unlimited retries threatens bounded delay. Dropping slow messages to bound delay threatens no loss. Refusing to retry to avoid duplicates brings back the loss. Reliability is a deliberate balance, chosen per flow and written down.

Why the happy path fails in production

With the definition in hand, return to Parcelly's first month and sort the failures by property. A few mechanisms account for most of the damage.

What Parcelly saw	Property at risk	Underlying cause	Chapter
Order accepted, but nothing downstream ever happens	No loss	Event committed to a database but never published	4
Order shipped twice	No harmful duplication	Redelivered message processed again	7
One bad message blocks a partition	Bounded delay	A message fails every time and is retried forever	8
Order "confirmed" before "placed"	Explainability	Related events processed out of order	9
Ledger disagrees with payments	Explainability	No shared identifier, no way to compare views	13

The fourth row is the awkward one. Reordering does not duplicate anything. It damages state, for example when a stale update overwrites a newer one, and none of the four properties names that directly. I file it under explainability because the symptom is a history nobody can explain, and Chapter 9 deals with the state damage itself.

The first row deserves a walk-through, because it is the failure that most often ends the honeymoon.

```
function placeOrder(request):
  transaction:
    orderId = insertOrder(request)
    // commit happens here; the order now exists in the database

    // the process can crash right here, between the two steps
  publish("orders", OrderPlaced(orderId))
```

This block is not safe, and it is shown as the broken baseline. The database commit and the broker publish are two systems with no shared transaction, which the book calls a **dual write**. If the process stops between the two lines, the order exists and nobody hears about it. If the lines are reversed, an event can be published for an order that was never committed. Neither ordering is correct, and both pass every test that does not kill the process at exactly the wrong moment.

This is not a rare bug. It is the default behavior of a design that never had a failure model. Chapter 3 gives the model, and Chapter 4 gives the fix.

A map of the book

The rest of the book answers the questions the definition raises, in the order a team usually meets them.

Question you will ask	Chapter
What does the broker actually promise, and what does it not?	2
What can break, and what will I see when it does?	3
How do I change my database and publish an event without losing one?	4
How do published events get from the database to the broker?	5
How do I retry without making things worse?	6
How do I make repeated processing harmless?	7
What do I do with a message that can never succeed?	8
When does order matter, and how do I keep it?	9
How do I coordinate work across services and undo it?	10
What should an event contain and be called?	11

Question you will ask	Chapter
How do I change an event without breaking consumers?	12
How do I see, measure and repair what is happening?	13
How do I recover from lost data, a lost region or a change of broker?	14
How do I test the failure paths?	15
What do I check before going live?	16

Part I, which this chapter opens, builds the vocabulary and the failure model. Part II covers publishing reliably, Part III consuming reliably, Part IV workflows and contracts, and Part V operating and proving the system. Read the parts in order for a first pass. If you are fighting a specific problem today, the table is an index.

Trade-offs and costs

Moving to events is not wrong, and this chapter is not an argument against it. It is an argument for making the move with open eyes, because the cost is real and is paid in specific places.

You pay in complexity of state. The synchronous flow had two outcomes: done or failed. The event-driven flow has many in-between states, and each must be named, shown to users, monitored and eventually resolved. Somebody has to own that state machine.

You pay in operational skill. Debugging a call chain takes a stack trace. Debugging a message flow takes correlation identifiers, lag dashboards and the ability to reason about timing. A team without those finds that the time to understand an incident grows even if incidents become less frequent. The cost lands on whoever is on call.

You pay in testing effort. The happy path takes little effort to test. Failure paths need duplicated messages, reordered messages and restarts at awkward moments. Chapter 15 shows how to do this without a slow, flaky suite, but the effort is not zero.

You pay in latency for simple things. An operation that was one function call becomes several hops through a broker. When the caller needs the answer now, such as validating a discount code before showing a price, a

synchronous call is the better tool.

I would not use events when two steps must succeed or fail together and live in the same service and database. A local transaction is simpler, stronger and cheaper than any pattern in this book. I would also hesitate when the team has no capacity to operate a broker and the monitoring around it, or when the system is small enough that a single deployable unit with one database does the job. Many good systems mix both styles: a synchronous call for a question that needs an answer, an event for a fact others may care about. I would choose per interaction, not per system.

The four-property definition also has a limit. It is a lens, not a specification. It does not tell you the right delay bound for your business or which duplicates are harmful for a given consumer. I don't know your numbers, and those answers rarely come from engineering alone. What I can offer is the habit of asking for them.

Common mistakes

- **Testing only the happy path.** It is tempting because the feature request described it and it passes quickly. Treat every arrow in the flow diagram as a question about failure, and write at least one test per answer (Chapter 15).
- **Treating the broker as the reliability layer.** Brokers advertise durability, which makes this tempting. A broker covers the segment between acceptance and delivery, and the failures that hurt most sit before and after it. List what the broker promises, then list the gaps around the promise (Chapter 2 and Chapter 3).
- **Using "eventually" as a design.** It sounds like a decision and avoids the harder question of how long. Attach a bound and an alert to every asynchronous hop that matters, and say what the user sees while waiting.
- **Assuming duplicates are too rare to handle.** In a quiet test environment they never appear. Design consumers on the assumption that every message may arrive more than once (Chapter 7).
- **Committing to the database and publishing to the broker as two plain steps.** It reads naturally and works in every demo. The fix is the transactional outbox of Chapter 4.

- **Defining reliability as uptime.** Uptime is cheap to measure, but a service can be up all month and still lose messages or process them late. Measure the four properties directly: what was lost, what was duplicated, how late things were, and whether you could explain them.

Key takeaways

- The happy path is a story in which every step succeeds, and a system tested only against it has been shown to work in the one case that rarely holds for long.
- Moving from synchronous calls to events does not remove coupling; it changes it from time and availability to contracts, broker behavior and unwritten timing assumptions.
- Event-driven failures are quiet and spread across services, so a design without a failure model tends to learn about its seams from production.
- "Eventually" is a conditional promise with no duration attached, so every asynchronous hop that matters needs a bound, an alert and a defined user-facing state for the waiting period.
- A reliable system has no loss, no harmful duplication, bounded delay and explainability, and because these trade against each other, each flow needs a deliberate, written balance.
- The seams between components, such as a database commit followed by a broker publish, cause more damage than the components themselves, and most of this book is about those seams, examined with three questions: what can fail, how would you observe it, and which guarantee survives it.
- Events are not always the right tool: when steps can share a local transaction, or the caller needs an immediate answer, a synchronous call is simpler and stronger.

The next chapter looks at the broker itself. Before we can design around its gaps, we need to know exactly what a queue or a log promises when it says a message was delivered, and what it quietly leaves to you.

Chapter 2 — Messaging Semantics

Imagine a Friday evening at Parcelly, in the middle of a sale. The Notifications service has been sending "your order is confirmed" emails for months without a complaint. As the backlog grows, autoscaling adds consumers, and a few minutes later it removes some again. By the end of the evening, support has two tickets. One buyer received the same confirmation email three times. Another received none, although the order is confirmed.

The team opens the broker dashboard and finds that nothing is broken. Every offset was committed. Every consumer is healthy. The graphs are flat. Yet the same system produced a duplicate for one buyer and a silence for another, and both outcomes are consistent with the documentation.

Somebody says the broker promised "at-least-once delivery", and somebody else says they had assumed the broker handled this. Both statements are true, and neither explains what happened to the two buyers. Suppose Notifications reads a log-based topic as one consumer group, with its client committing offsets automatically, and that a rebalance ran each time the group changed size. Those two facts can produce both tickets, and the chapter shows how.

To explain it, the team needs a precise picture of what a broker promises, what it does not, and which parts of the message's life belong to the application instead. What does a message actually go through between a producer and a finished side effect, and where does each guarantee begin and end?

The problem, and why it is hard

Most engineers meet a broker through its simplest description: you publish a message, and a consumer receives it. That description hides three separate handoffs, and every reliability question in this book lives in one of them.

The first handoff is producer to broker. The second is broker to consumer. The third is consumer to the effect: the database write, the email, the call to a carrier. A broker has a lot to say about the first two and almost nothing to say about the third, because the third happens in code and in systems the broker cannot see.

The difficulty is that each handoff ends with an acknowledgment, and an acknowledgment is a statement about the past made by one party to another across a network. It says "I received this" or "I finished this". It does not say "and nothing else can go wrong". When an acknowledgment is lost, the sender cannot tell whether the work happened. That uncertainty is the root of the three delivery semantics, and we will come back to it after we have the vocabulary.

A second difficulty is that the word "broker" covers two families that behave differently in ways that matter. A **queue** is a broker structure where each message is delivered to one consumer and acknowledged individually. An **event log** is a durable, append-only sequence of events, where consumers read at their own pace and record a position. Statements that are true of one family are often false of the other. The book names the family whenever the distinction matters, and this chapter is where the distinction is built.

A third difficulty is that the guarantees in a product's documentation are scoped. "Delivered at least once" is a statement about the broker's own boundary. It does not cover what the consumer does after receiving the message, and it depends on settings the producer chose when it published. Reading the guarantee without its scope is how a team ends up with the Friday evening above.

Queues and logs

The queue model

In a queue, the broker tracks the state of each message. A message is waiting, delivered to a consumer, or acknowledged and gone. Several consumers can read from the same queue; this arrangement is called **competing consumers**, because each message is delivered to one of them at a time and they compete for the next one.

When a consumer acknowledges a message, the broker deletes it. Retention is therefore short by design: a message lives until it is processed, or until it expires. There is no built-in way to ask the queue for a message that was already acknowledged, because the queue does not keep it.

The consequence is that a queue is a good fit for work that should be done once and forgotten. A Parcelly job such as "render this invoice PDF" is a natural candidate: do the work, then the message has no further use. The cost is that history is not available. If a bug sends the wrong text for an hour, the messages that carried the facts are gone, and the repair has to start from somewhere else.

The log model

In an event log, the broker does not track a per-message state. It stores messages in order, in partitions, and keeps them according to a retention policy that is independent of who has read them. A **partition** is an ordered subsequence of a topic, and a **topic** is a named stream of messages. Each message has an **offset**, its position in the partition.

Consumers keep track of how far they have read. That position is also called an offset, and the consumer, or the broker on its behalf, records it. A **consumer group** is a set of consumers that share the work of one topic: each partition is assigned to one consumer in the group at a time. Different groups read the same topic independently, each with its own offsets.

The log's defining property is that reading does not consume. Ten groups can read the same topic, and a new group can start from the beginning. **Replay**, the deliberate re-reading of history, is the term this book reserves for that act, and it is only possible because the log keeps the data. Parcelly's Ledger is a good example of a consumer that benefits: if a bug is found in how entries are built, the Ledger service can be fixed and pointed back at old events.

The cost is that the log pushes bookkeeping onto the consumer side. In the classic log model, the broker cannot redeliver one failed message while moving on to the next, because it does not track messages individually. A consumer that cannot process the message at offset 500 has to decide what to do before it can safely move to 501.

Choosing between them

Concern	Queue family	Log family
Unit of tracking	Each message	A position per partition
Retention	Until acknowledged or expired	By time or size, independent of readers
Several independent readers	Needs a copy of the message per reader	Natural, one position per group
Replay of history	Not available	Available within retention
One failing message	Redelivered on its own, others continue	Blocks the partition unless handled
Ordering	FIFO in storage at best; lost across competing consumers and redelivery	Strong within a partition

I would not pick a family by tool popularity. I would start from three questions: does anyone need to read this history again, do several consumers need the same facts, and does ordering matter for the same aggregate? Yes to any of them points toward a log. A pure work queue, with no need for history and a need for independent retries of individual messages, points toward a queue.

Many teams end up running both. Events that other services build state from go on a log, and tasks that should happen once go on a queue. That is a legitimate design, and it comes with the cost of operating and understanding two sets of semantics. Chapter 9 returns to what ordering means in each family.

Push, pull and the flow of delivery

Brokers also differ in who initiates delivery. In a **push** model, the broker sends messages to consumers as they become available. In a **pull** model, the consumer asks for messages when it is ready.

Queue-based brokers commonly push (RabbitMQ delivers to consumers, while Amazon SQS consumers poll, so even this varies within the family), with a limit on how many unacknowledged messages a consumer may hold at once. That limit is often called a prefetch count or a window. Log-

based brokers commonly pull: the consumer asks for the next batch from its current offset. Both models can be found in either family, so I would treat this as a tendency and not a law.

The difference matters mostly for what happens when a consumer is slow. In a push model without a limit, the broker keeps sending, the consumer's memory fills with messages it has not started on, and those messages sit in a state where they are neither available to another consumer nor done. In a pull model the consumer controls its own pace, and a slow consumer shows up as growing consumer lag, not as an overloaded process.

Consumer lag is how far a consumer is behind the head of a partition or queue, in messages or time.

The practical rule is that a consumer should never hold more work than it can finish before its claim on the work expires. Chapter 3 treats slow consumers and backpressure as failures in their own right, and Chapter 6 covers how to bound the load. For now, remember that "delivered" and "being processed" are different states, and the gap between them is where time passes.

Acknowledgment, redelivery and offsets

Acknowledgment in a queue

An **acknowledgment**, usually shortened to **ack**, is the consumer's statement that it has finished with a message and the broker may forget it. A **nack**, a negative acknowledgment, says that the consumer could not process the message. Depending on the broker and settings, a nack returns the message to the queue, drops it, or sends it to a dead-letter queue (DLQ).

The key design question is when to ack. There are only two reasonable positions.

- **Ack before processing.** The broker forgets the message immediately. If the consumer crashes mid-work, the message is lost. This produces at-most-once behavior.
- **Ack after processing.** The broker holds the message until the work is done. If the consumer crashes after finishing but before acknowledg-

ing, the message comes back and the work is done twice. This produces at-least-once behavior.

There is no position that gives both properties, because the acknowledgment and the work are two separate actions on two separate systems, and a crash can fall between them.

Visibility timeout and redelivery

How does a queue decide that a consumer has crashed? It cannot observe that directly. Some brokers watch the connection and treat a closed connection as failure. Others use a **visibility timeout**: when a message is delivered, it becomes invisible to other consumers for a set time. If the consumer acknowledges before the time runs out, the message is deleted. If not, the message becomes visible again and is delivered to someone else. That second delivery is a **redelivery**.

The timeout is a guess about how long processing takes, and both ways of guessing wrong cost something. If it is too short, a consumer that is slow but alive will have its message handed to a second consumer while it is still working. Now two consumers process the same message, and the first consumer's eventual ack may fail, be ignored, or, depending on the broker, succeed and delete the message while the second consumer is still working on it. In that last case a failure of the second attempt loses the message. If it is too long, a real crash leaves the message hidden for that whole period, and the delay shows up as latency for the buyer.

Say Parcelly's Inventory consumer normally finishes a reservation in well under a second, and someone sets the timeout to 30 seconds. That number is illustrative. A database hiccup that makes one reservation take 40 seconds will now produce a second delivery of the same `OrderPlaced`, and Inventory will try to reserve the stock twice. Nothing is broken in the broker. The timeout encoded an assumption, and the assumption stopped holding.

Some systems let a consumer extend the timeout while it works. That helps with legitimately long jobs, and it adds a new thing to get wrong: a consumer that extends forever on a message it will never finish.

Offsets in a log

In a log, there is no per-message ack. The consumer instead performs an **offset commit**: it records "I have processed everything before this position". After a restart or a reassignment, the next consumer reads from the committed offset.

The same two positions exist here. Commit the offset before processing, and a crash loses the messages after the last completed one, up to the committed position. Commit after processing, and a crash repeats the messages between the last commit and the crash. Real consumers commit in batches, so the number of repeated messages after a crash is a batch, not one.

Many client libraries can commit automatically. That is convenient and it is also the least explicit way to set your delivery semantics, because, depending on the client, the commit can happen at a moment unrelated to when your processing finished. If processing is handed off to another thread, the client can commit work that never completed. In my view, a consumer that matters should commit explicitly, at a point the team can point to in the code and reason about.

Consumer groups and rebalancing

When consumers join or leave a consumer group, the broker redistributes partitions among them. This is called **rebalancing**. During a rebalance, a partition that was being read by one consumer is reassigned to another, and the new owner starts at the last committed offset.

Log-based groups make the same guess that the visibility timeout makes in a queue, in another form. A consumer that does not return to the broker within a configured interval is presumed dead, its partitions are reassigned, and its uncommitted work is read again by the new owner. A slow consumer that is still alive is the everyday cause of rebalance duplicates.

This is where the duplicates in the opening scenario come from. Consider a consumer that has processed messages 100 to 150 but has only committed up to 120. A deploy restarts it, and the partition moves to a second consumer, which reads from 120. Messages 120 to 150 are processed

twice, and no part of the system malfunctioned. Rebalancing is a normal event that can happen at every deploy, scale-up and crash, so redelivery is an everyday occurrence, not a rare accident.

The silence has the mirror-image cause. If the client committed offsets for messages that the consumer had only handed to a background sender, and the process was stopped before the sender finished, the next owner starts after those messages. They are never read again, and no log line says so. The same rebalance can therefore repeat one buyer's message and drop another's.

There is a related hazard: for a short time, the old owner may not yet know it has lost the partition and may still be working on it. Two consumers can then process the same partition at once. Well-designed clients try to prevent this, but I would not design a consumer that depends on the prevention being perfect.

Delivery semantics

With the mechanics in place, the three delivery semantics are consequences of where you put the acknowledgment.

At-most-once

At-most-once means each message is delivered zero or one times. It is achieved by acknowledging before processing, or by not retrying a failed publish. A message can be lost, and it can never be duplicated.

It is a reasonable choice when losing a message costs less than handling duplicates. A periodic "current location" ping from a Parcelly carrier is one case: the next ping makes the lost one irrelevant. It is the wrong choice for `PaymentCaptured`, where a lost message means money moved and nobody recorded it.

At-least-once

At-least-once means each message is delivered one or more times. It is achieved by acknowledging after processing and retrying until an acknowledgment arrives. A message is never lost while the broker holds it, and it can be delivered again.

This is the semantics the book builds on. It is the only one of the three where the failure is visible and recoverable: a duplicate is something you can detect and ignore, and a lost message is something you may not discover for days. The price is that every consumer must tolerate a **duplicate**, meaning a message processed or delivered more than once. Chapter 7 is about how.

"Exactly-once"

Delivery is a physical act across an unreliable network, and the receiver's acknowledgment can be lost like any other message. A sender that did not get an ack has two options: send again, which risks a duplicate, or do not, which risks a loss. No protocol removes this choice for the two parties alone.

Exactly-once is nevertheless a real and useful outcome, and it is an effect, not a delivery guarantee. When at-least-once delivery meets a consumer whose repeated processing has the same effect as processing once, the combined system behaves as though each message was handled exactly once. The recipe is at-least-once delivery plus an **idempotent consumer**.

Some brokers advertise exactly-once processing, and the claim is legitimate within a boundary. A log-based broker can make a read-process-write cycle atomic when the input, the output and the offset commit all live inside that same broker. Inside that boundary, a crash rolls everything back together. The guarantee also requires every downstream reader to read only committed data, and it relies on a stable producer identity so that an old instance is fenced out after a restart. The boundary ends where your side effects begin. The moment the consumer writes to a relational database, calls a payment provider or sends an email, the broker's transaction no longer covers the effect, and the duplicate problem returns.

For that reason, the book's default remains at-least-once plus idempotent consumers. It works across any broker and any side effect, it does not depend on a feature being enabled, and it survives a migration to a different product, a move that Chapter 14 covers. If you can keep a whole pipeline inside one broker's transactional boundary, the feature may reduce the work, and I would still design the consumers so that they do not depend on it.

Note: Producers can create duplicates too, before a consumer is involved. A producer that times out waiting for a publish acknowledgment will usually publish again, and the broker may now hold two copies of one event. Both copies carry the same `event_id`, provided the producer assigned it once when the event was created and did not mint a new one for the retry. Some brokers de-duplicate a producer's own retries, by sequence number within a producer session or by a deduplication id within a time window. That covers a retry inside one connection. It does not cover a process that restarts and publishes again, such as an outbox relay, which is why the consumer still checks `event_id`. Chapter 4 covers how to publish without losing events, and Chapter 7 covers recognizing duplicates.

What the broker actually guarantees

Everything above describes the broker-to-consumer side. The producer-to-broker side has its own limits, and they depend on configuration more than on the product name.

Producer acknowledgments

When a producer publishes, the broker can acknowledge at different moments, and the useful way to tell the settings apart is by how many nodes hold the message when the producer hears "done". With no acknowledgment, the producer learns nothing about storage. With acknowledgment by the receiving node, one node has accepted the message, whether or not it has flushed it to disk. With acknowledgment after replication, the message exists on a required number of nodes. Flushing to disk is a separate question, which comes below.

The producer's choice matters for what a crash can do. With the first two settings, a broker node that dies right after acknowledging can take the message with it. With replicated acknowledgment, the answer arrives later and the loss window is smaller. The price is higher latency for every publish and a dependency on enough nodes being available to complete the write.

Brokers that offer publisher confirms, in the queue family, and acknowledgment levels, in the log family, are describing the same idea. Check what the acknowledgment you are waiting for actually means in your configuration. In my view, that is one of the most valuable documentation pages a team can read once and write down.

Replication and fsync

Replication protects against the loss of one node: the message exists on several. It does not protect against everything failing together, and its guarantee depends on how many copies had to exist before the producer was told "done". Check both the acknowledgment level and the minimum number of replicas it requires, because "all replicas" can mean one when the others have fallen behind. Also check whether a failover may promote a replica that was not up to date, since that can lose data the producer was told was safe.

There is a related and often misunderstood point about disk. Writing to a file normally places the data in the operating system's cache, and the data reaches the physical disk later, unless the program forces it with a synchronous flush, commonly called fsync. Forcing the flush on every message is slow, so many brokers do not do it for each write. They rely on replication to cover the gap, and they flush on a schedule. Whether a given broker's acknowledgment implies the data is on disk depends on its configuration, and I would not assume it.

I do not know which of these settings suits your system, and it depends on what a lost event costs. A lost `PaymentCaptured` and a lost "page viewed" are different problems, and it would be reasonable to configure them differently.

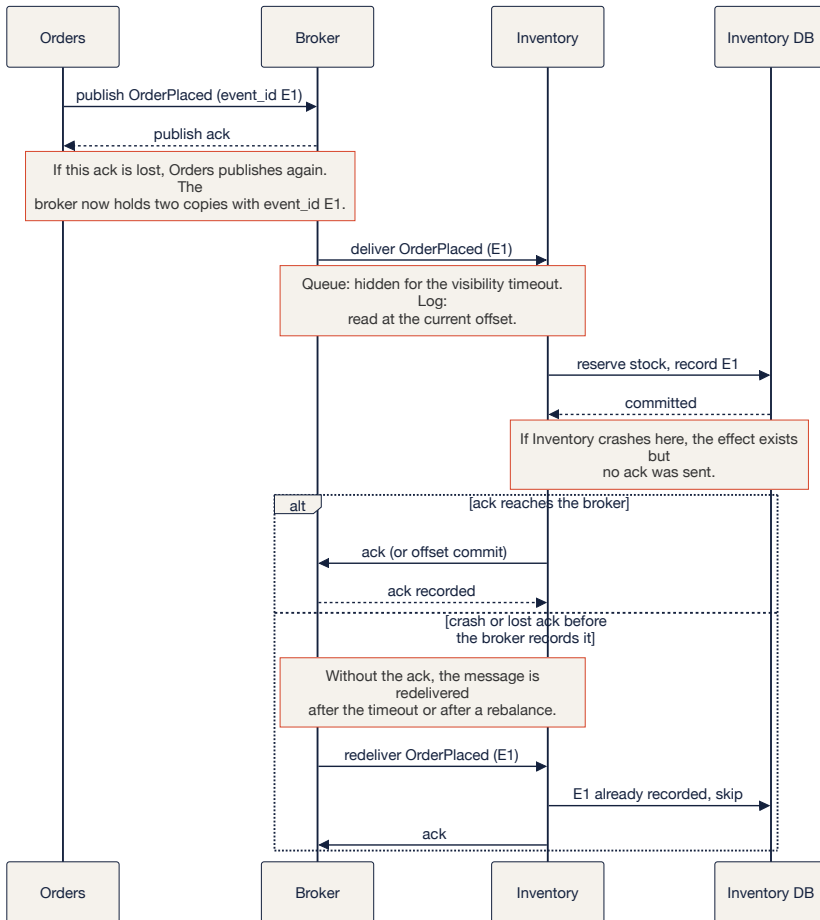
The limit of the guarantee

Put these together and the boundary becomes clear. A broker can promise that a message it accepted will be held and offered to consumers, within its retention or expiry limits, and only as durably as the acknowledgment and replication settings above allow. It cannot promise that a message reaches the broker in the first place, because that depends on what the producer did when its publish timed out. It also cannot promise that a consumer's effect happened exactly once.

Two gaps are left in the application's hands. The first is on the producer side: an order saved in the Orders database and an `OrderPlaced` event published to the broker are two writes, and a crash can fall between them. Chapter 1 named this the dual write, and Chapter 4 removes it. The second is on the consumer side, between the effect and the acknowledgment. Chapter 7 closes that gap. Seen as the three handoffs that opened this chapter, the broker states failure semantics for the first two, and the two gaps are what remains in your hands: the dual write in front of the first handoff, and the gap between effect and acknowledgment in the third.

The life of a message

The diagram below follows one `OrderPlaced` message from the Orders service to a finished effect in Inventory. Look for the two places where a failure leaves the sender without an answer, marked in the notes: the lost publish acknowledgment and the lost consumer acknowledgment. Everything the chapter has described happens inside this one sequence.



The **else** branch of the diagram is the point of the chapter. The redelivery is not an error and no component misbehaved. The only thing that keeps the stock from being reserved twice is that Inventory recorded E1 in the same transaction as the reservation and checked it again on the second delivery.

Here is how a consumer with this property looks in pseudocode. It uses an **inbox**, a table of processed **event_id** values in the consumer's own database, and places the acknowledgment after the commit.

```

function handleOrderPlaced(message):
    // a redelivery of work that already committed is acknowledged,
    not redone
    if inboxContains(message.event_id):
        ack(message)
        return

    transaction:
        reserveStock(message.payload.order_id, message.payload.lines)
        // the unique constraint on event_id makes a concurrent
duplicate
        // fail here, roll back, and come back as a redelivery that
the
        // check above then acknowledges
        inboxInsert(message.event_id)
        // transaction commits here, or rolls back on any error

        // ack comes last: a crash before this line causes a redelivery,
        // which the inbox check turns into an ack rather than a second
reservation
        ack(message)

```

This is the design that makes at-least-once safe, and it is introduced here only to show where the acknowledgment belongs. Chapter 7 builds it out, including the cases where the inbox check alone is not enough.

Trade-offs and costs

Choosing semantics is choosing which failure you would prefer to manage. None of the options removes failure, and each moves it somewhere else.

At-least-once costs you idempotency work in every consumer. Each consumer needs a way to recognize repeated work, usually an inbox table in its own database. That is storage, a unique constraint on the hot path, and a cleanup policy. For a team with a dozen consumers it is a pattern to apply a dozen times. When a consumer's effect is naturally idempotent, such as setting a status to a fixed value, the inbox may be unnecessary, and I would still write down why.

At-most-once costs you completeness. It is simple and fast, and it is acceptable only when someone has decided in advance that a lost message is cheaper than a duplicate. That decision should be explicit and recorded. The usual shape of the mistake is a team that chose it by default, through auto-acknowledgment settings, without realizing it.

Transactional processing inside a broker costs you portability. It can reduce duplicate handling within its boundary, and it ties the design to one product's features, to every participant staying inside that boundary, and to the operational effort of running it. When you cross to a relational database or an external API, you need idempotency anyway. I would not reject the feature, and I would not build correctness on it alone.

Strong producer acknowledgments cost latency and availability.

Waiting for several replicas makes each publish slower and makes the write fail when too few nodes are reachable. For events that must not be lost, that trade is usually right. For high-volume, low-value telemetry, it may not be.

Logs cost consumer-side bookkeeping, and queues cost history. If you choose a log, you own the offset commit policy and the handling of one message that blocks a partition. If you choose a queue, you give up replay. When you need independent retries of individual messages and no history, a log may be more machinery than the work deserves.

When not to apply this chapter's advice: a system with a single producer, a single consumer and tolerance for occasional loss, such as an internal metrics feed, needs neither an inbox nor strong replication. The cost of the guarantees should match the cost of the failure. The mistake is not choosing a weaker guarantee; the mistake is not knowing you did.

Common mistakes

- **Treating "delivered" as "processed".** It is tempting because the dashboard shows deliveries and acknowledgments, and those are the numbers that look healthy. Delivered means the broker handed the message over. Processed means the effect committed. Track both, and make your acknowledgment follow the effect.
- **Acknowledging before the work is done.** This is tempting because it avoids redelivery and makes the queue look clean. It converts every consumer crash into a lost message. Acknowledge after the effect commits, and handle the duplicates that result.
- **Leaving offset auto-commit on by default.** It is tempting because it works in a demo and needs no code. Depending on the client, it can

commit at a moment unrelated to your processing, so a crash can lose or repeat an arbitrary slice. Commit explicitly, after the effect.

- **Setting the visibility timeout by feel.** It is tempting because the default usually seems fine. A timeout shorter than a slow but healthy run causes concurrent duplicate processing. Base it on the slowest legitimate processing time you can observe, plus margin, and extend it deliberately for long jobs.
- **Believing a broker's exactly-once feature covers your database.** It is tempting because the documentation uses the phrase. The guarantee is scoped to the broker's own boundary. Anything outside it, such as a row in your database or an email, needs idempotency.
- **Assuming a published message is a stored message.** It is tempting because the publish call returned. Depending on the acknowledgment level, the message may exist only in one node's memory. Know what your producer waits for, and write it down.
- **Assuming rebalancing is rare.** It is tempting because it happens quietly. Deploys, scale events and crashes can all trigger one, depending on the group protocol and membership settings, and each rebalance can repeat a batch of messages. Design as if redelivery happens daily, because the cost of being wrong is a duplicate effect.

Key takeaways

- A message passes through three handoffs, producer to broker, broker to consumer and consumer to effect, and a broker's guarantees cover only the first two.
- Queues track each message and delete it on acknowledgment, while logs retain messages independent of readers and let each consumer group keep its own offset, which is what makes replay possible.
- Where you place the acknowledgment or offset commit decides the semantics: before processing gives at-most-once, after processing gives at-least-once, and no placement avoids both loss and duplicates.
- Redelivery is routine, not exceptional, because visibility timeouts expire, consumers crash, and rebalances (which deploys commonly cause) repeat batches.

- Exactly-once is an effect built from at-least-once delivery and an idempotent consumer, and a broker's own exactly-once processing stops at the boundary of that broker.
- A producer acknowledgment is only as strong as the replication and flush settings behind it, so the team should know and record what "published" means for each important event.
- Each semantic option moves failure instead of removing it, and the right choice is the one whose remaining failure you have decided you can afford to manage.

Messaging semantics tell you what the broker will and will not do on a normal day. Chapter 3 turns to the abnormal ones, and builds a practical model of what actually breaks: network partitions, crashes, slow consumers, clocks and people.

Chapter 3 — A Practical Failure Model

Imagine Payments at Parcelly in the middle of a capture. Orders has sent a `CapturePayment` command to Payments, and Payments has called the external payment provider to capture the money. The provider's connection goes quiet. Thirty seconds later, Payments' HTTP client gives up and raises a timeout.

Payments now has to answer a question it cannot answer. Did the provider capture the payment before the connection died, or did it never receive the request? Both histories produce exactly the same observation: silence, then a timeout. If Payments retries and the first capture had succeeded, the buyer is charged twice. If Payments reports failure and the first capture had succeeded, Orders cancels an order that is already paid for.

Nothing in this story is exotic. There is no bug, no outage report, no engineer who made a mistake. A network did what networks do, and the system had no way to tell which of two worlds it was living in.

Most of what goes wrong in an event-driven system has this shape: a step happened, or did not, and the evidence is missing. This chapter names the ways that evidence goes missing, so that you can design for each of them on purpose. The question it answers is this: what exactly can break in a system like Parcelly's, what will you see when it does, and where in this book does each failure get handled?

The problem, and why it is hard

A failure model is a written list of the things you assume can go wrong, and the things you assume cannot. Every system has one, whether anyone wrote it or not. The unwritten version usually contains two entries: "the service crashes" and "the network is slow". Those are real, but they are the failures that announce themselves.

The simple case is a failure you can detect, such as a refused connection or a database error. Each tells you the step did not complete.

The hard case is the failure that leaves you uncertain. A timeout does not say "the operation failed". It says "I stopped waiting". The operation may have failed, may have succeeded, or may still be running and about to succeed after you gave up. You need only a timeout and a side effect on the other end to feel it.

This is why the obvious approach, writing code for the happy path and adding a `try` and a `retry` around each call, does not hold. Retrying is a decision about the future, made without knowing the past. If the earlier attempt succeeded, the `retry` is a duplicate. If you decline to `retry`, a message may be lost. Neither choice is wrong in itself; each is wrong in some world you could not distinguish from the one you were in.

There is a second reason. Failures combine: a deploy restarts a consumer while a broker fails over and a producer retries. Listing them one at a time is still the right start, but production delivers them together.

So the work of this chapter is classification. For each failure, I will ask the three questions from Chapter 1: what can fail at this seam, how would you observe it, and which guarantee survives it. In the entries below, "what is actually true" answers the first, "what you see" answers the second, and the pattern named at the end of each entry answers the third. Taken together, these three answers are a failure's semantics, and the rest of the book keeps asking for them.

Ten failures that matter

The list below is the set I would start with for a system built from services, relational databases and a message broker. It is not complete, and no list is.

Network partitions and timeout uncertainty

The opening scenario is the model case. Two parties are connected by a network, and the network drops, delays or reorders traffic. A **network partition** is the extreme form, where two groups of machines cannot

reach each other at all. A timeout is the everyday form, and it is far more common.

What each side can observe is asymmetric. The caller sees only that no answer arrived. The callee may have done the work and lost the reply, or never have seen the request. The same applies to every hop in Parcelly: a service calling an external carrier, a producer sending to the broker, a consumer acknowledging to the broker.

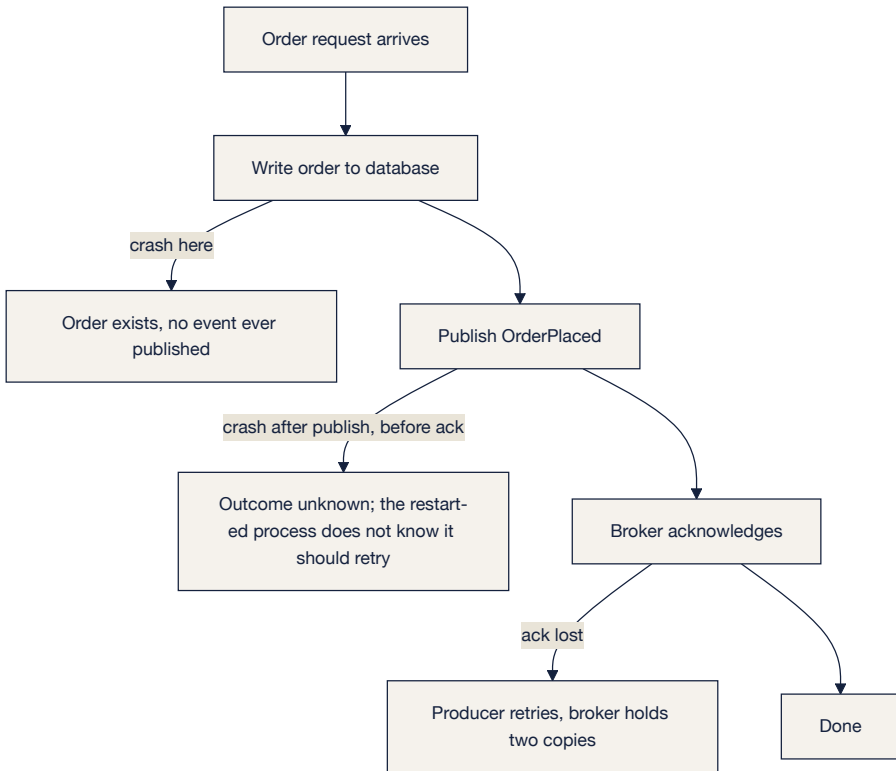
The only honest response to this is to make the operation safe to repeat. If Payments sends the provider a stable idempotency key with each capture, and the provider treats a repeated key as the same request, then retrying after a timeout becomes harmless. The design stops depending on the answer to "did it happen?", because both answers lead to the same final state. Chapter 6 covers when and how to retry, and Chapter 7 covers the idempotency that makes retrying safe.

Some external systems offer no such key. Then you ask the system what happened before acting again, which is a reconciliation step, and Chapter 13 covers it.

Producer crash windows

Consider Orders handling a new order. It must save the order in its database and publish `OrderPlaced` to the broker. These are two systems, and no transaction spans them. This is the dual write from Chapter 1, and the interesting part is what each crash window leaves behind.

The process can crash between the two writes, and the order of the writes decides which damage you get.



The diagram draws the database-first order and shows three windows: a crash before the publish, a crash after it with the outcome unknown, and a lost acknowledgment. If the database commit comes first and the process dies before publishing, the order exists and nobody downstream ever hears of it. Payments never authorizes, Inventory never reserves, and the buyer waits for a confirmation that never comes. Nothing errors. The order sits there.

The reverse order, which the diagram does not draw, has its own window. If you publish first, a crash before the commit leaves a worse state: an event announcing a fact that never became true. Downstream services act on an order that does not exist. And in either order, if the broker's acknowledgment is lost on the way back, the producer will retry and the broker may hold two copies of the same event.

What you see from outside is quiet: orders without events, events without orders, and occasional duplicates. Nothing in the logs says "lost". Chapter 4 introduces the transactional outbox, which removes the window by mak-

ing the state change and the intent to publish one local transaction. Chapter 5 covers how the outbox rows then travel to the broker.

Broker failover

Brokers are replicated so that a machine can die without taking the data with it. When the machine holding the leading copy of a partition or queue dies, another takes over. This is a **failover**, and it is a designed behavior, not a malfunction.

What matters is what the producer was told before the failover. Whether a message acknowledged moments before a leader died survives depends on how the broker was configured to acknowledge: not at all, after the receiving node has accepted it (whether or not it has flushed it to disk), or after a required number of replicas has it. The strongest settings cost latency and throughput. The weakest settings can lose acknowledged messages in a failover. This differs between broker families and between configurations, so I will not claim a specific behavior for a specific product here; the point is that the guarantee has a boundary, and the boundary is a setting someone chose.

During failover, producers also see errors and timeouts, retry, and may publish duplicates of messages the old leader had already accepted. Chapter 2 describes where broker guarantees stop. The rule here is to treat "acknowledged by the broker" as strong evidence and not as proof.

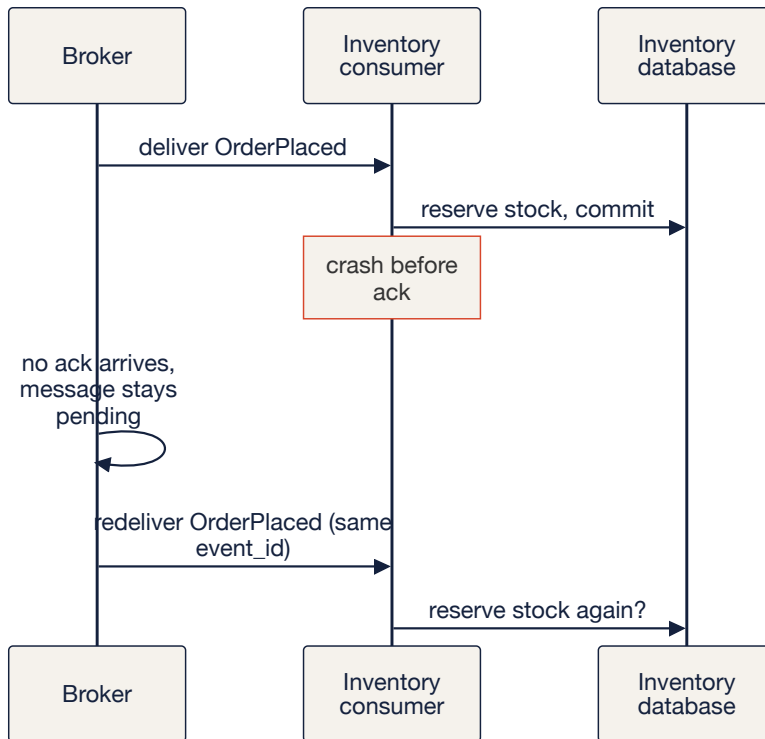
What you see: a burst of publish errors, a dip in consumer throughput, then a surge of redeliveries. If a message was lost, you see nothing at all, which is why Chapter 13's reconciliation matters.

Consumer crash: processed but not acknowledged

Now the consumer side. Inventory receives `OrderPlaced`, reserves stock in its database, and commits. Then the process dies before it can acknowledge the message.

From the broker's view, the message was never finished. On a queue-based broker the broker notices the closed connection or the visibility timeout expires, and the message is redelivered. On a log-based broker the con-

sumer's offset was never committed, so the next consumer to own the partition starts from the old position and reads the message again. In both families, the second delivery arrives for work that has already been done.



The final line of the diagram is the whole problem. A consumer that reserves stock unconditionally on every delivery reserves it twice. This is the ordinary source of duplicates, and it is the reason the book's default is at-least-once delivery combined with idempotent consumers. A consumer can crash at different points, and each point has a different consequence:

```

function handleOrderPlaced(message):
    // crash point A: before any work. Redelivery starts clean.
    if inboxContains(message.event_id):
        ack(message)
        return

    transaction:
        reserveStock(message.payload.order_id, message.payload.lines)
        // the unique event_id makes a concurrent duplicate fail here
        inboxInsert(message.event_id)
    // transaction commits here
    // crash point B: committed, not acked. Redelivery hits the inbox
    check
    // above and is acknowledged without repeating the work.

    ack(message)
    // crash point C: after ack. Nothing to repair.

```

This is a compact version of the handler from Chapter 2, cut down to show the crash points. The effect of crash point A is nothing. The effect of crash point C is nothing. The interesting one is B, and only a consumer that records what it has processed, in the same transaction as the work, survives it. Chapter 7 builds that properly.

What you see: duplicate side effects, such as two reservations, two emails, two ledger entries, and no error anywhere, because each processing attempt succeeded.

Slow consumers and backpressure

Not every failure is a crash. A consumer that is merely slow can do as much damage, and it is harder to notice because nothing is broken.

Suppose Notifications calls an email provider that starts taking 10 seconds per request instead of 100 milliseconds. Notifications still works, but each message now occupies a worker 100 times longer. Messages arrive faster than they are finished, so consumer lag, which Chapter 2 defined, grows steadily.

Slowness then triggers secondary failures. On a queue-based broker, a message that outlasts its visibility timeout reappears while the first worker is still processing it, so a second worker starts the same message. The du-

plicates add load to a system that is already slow. On a log-based broker, a consumer that does not call back to the broker in time may be declared dead and its partitions reassigned, which starts the next failure on this list.

Backpressure is the discipline of bounding the work in flight at each stage so that load stays finite. In a broker-mediated system the broker absorbs the difference, which is the benefit of temporal decoupling, but the buffer is finite, and it runs out in two different ways. When the broker's disk fills, it may throttle or block producers, or stop serving altogether, and which of these your broker does is something to find out before the day it happens. Whichever it is, the producer sees it, and that is real backpressure: loud and visible. When a consumer falls further behind than the retention period of a log, or a queued message outlives its expiry, the broker deletes messages that were never read. That is not delay. It is silent loss: no error is raised, no consumer fails, and the unread `PaymentCaptured` events are gone. Long before either limit, a backlog can already violate a business deadline, such as a payment authorization that expires before it is captured.

Because the broker's buffer only postpones the problem, load has to be bounded in several places at once. A consumer can cap the number of messages in flight, or the batch it prefetches. It can cap its worker concurrency, so that a slow email provider ties up a fixed number of workers and not all of them. It can rate-limit its calls toward the slow dependency, or pause consumption entirely while the dependency recovers. A queue can have a length limit with an explicit overflow policy, so that you choose what is dropped or refused instead of letting the broker choose.

The producing side and the topology set limits too. On the producing side, the outbox absorbs a broker that refuses publishes, because the rows wait in the database until the relay can deliver them. And on a log-based broker, the partition count caps how many consumers in a group can work in parallel, so adding instances stops helping at a number fixed in advance. Each of these limits trades throughput for stability, and each needs a metric, or you will not know it is the one holding you back.

What you see: rising lag and message age, a growing backlog, redelivery counts climbing. Chapter 6 covers backoff, in-flight limits and pausing consumption, Chapter 8 covers what to do with messages that cannot be

processed in time, and Chapter 13 covers the lag and age metrics that tell you early.

Messages that fail every time, and dependencies that fail for every message

So far every failure was a matter of chance: a crash, a lost reply, a bad moment. Some failures are deterministic. A **poison message** is a message that fails on every attempt, whatever the timing. Suppose an `OrderPlaced` arrives at Inventory with a SKU that does not exist in its catalog. Inventory raises an error, the message is redelivered, the error repeats, and nothing about the next attempt is different from the last.

The cause can also sit outside the message. If Payments' provider is down, every capture fails, and every message looks poisonous for as long as the outage lasts. The two cases call for opposite responses. A message that will never succeed should leave the main flow quickly, into a **dead-letter queue (DLQ)**, a separate place where messages that cannot be processed wait for a human or a repair job. A message that fails because a dependency is down should not be dead-lettered at all, because it would succeed after the recovery, and dead-lettering a whole backlog during an outage turns a temporary problem into a manual one.

What goes wrong in between follows a pattern. Retries on a poison message use up attempts and capacity without any chance of success. On a log-based broker, or a queue that preserves order, one such message can block everything behind it. Many consumers retrying against one failing dependency add load to the thing that is trying to recover, a retry storm. And a code bug can fail without any error: the consumer acknowledges, returns, and writes the wrong value, so nothing in the failure path ever runs.

What you see: the same `event_id` appearing in the logs again and again, redelivery counts that never fall, a stuck partition, a DLQ that grows during an outage, and, for the silent bug, nothing at all until a reconciliation check disagrees. Chapter 6 covers how to tell which failures are worth retrying, Chapter 8 covers poison messages and the dead-letter path, and Chapters 13 and 14 cover finding and repairing the damage from a bug that raised no error. The price of any classification is that it can be wrong, so keep the decision visible and reversible.

Clocks and timestamps you cannot trust

Every machine has a clock, and no two clocks agree exactly. They drift, they are corrected by synchronization, and a correction can move a clock backwards. A paused virtual machine wakes up with a clock that is wrong by however long it slept.

The consequence for events is specific. `occurred_at` is a fact about when something happened according to one machine. Comparing the `occurred_at` of an event from Orders with one from Payments tells you something about their clocks as much as about the order of events. If you sort by timestamp to decide which update wins, a service with a clock a few seconds ahead will win every conflict it should lose.

Timeouts have a related trap. A deadline expressed as a wall-clock time, "this authorization is valid until 14:00", is compared by a machine whose idea of 14:00 may differ from the issuer's. For measuring durations on one machine, a monotonic clock, which only moves forward, is the right tool. For ordering across machines, I would not use wall-clock time at all.

What orders events reliably is the structure of the system: on a log-based broker, a partition keyed by the aggregate, giving a total order for events on that key as long as there is a single producer path for each key and the partition count does not change, or an explicit version number on the aggregate that consumers can compare. Chapter 9 covers ordering in depth. Treat `occurred_at` as information for humans and for analytics, and be wary of it as an input to correctness.

What you see: updates applied out of order, "impossible" sequences in logs where an effect appears to precede its cause, and bugs that appear only on one host.

Deploys and rebalances

A deploy is a planned failure. Instances are stopped and started, and for the time between, work is interrupted mid-flight. If your deploy process sends a termination signal and the consumer exits with messages in progress, you have created the "processed but not acknowledged" case on purpose, repeatedly, on every release.

A **rebalance** is the log-family version, and it can occur without a deploy. When members join or leave a consumer group, or when a member is judged unresponsive, partitions are reassigned. During the handover, two consumers can briefly believe they own the same partition, and a consumer that was slow can keep processing messages after its partition has moved elsewhere. The queue-family equivalent is competing consumers whose visibility timeouts expire while work continues.

The damage pattern is a burst of duplicates at deploy time, and sometimes a short stall while partitions are reassigned. A deploy that rolls through all instances in quick succession can trigger several rebalances in a row, and in the worst case consumers spend more time reassigning work than doing it. Two mitigations reduce this, at the price of more configuration to understand: cooperative reassignment, where only the partitions that must move are taken away, and a stable member identity, so that a restarted instance can reclaim its old partitions without a full reshuffle.

Make consumers stop gracefully by finishing or releasing in-flight messages before exit, and assume the graceful path will sometimes fail, so the idempotent consumer remains the real defense. I would also stagger rolling deploys so they do not trigger reassignments in a tight loop. What you see: duplicates and lag spikes that line up with release times.

Human mistakes

The last category is the one most architecture documents leave out. People operate these systems, and people make mistakes under pressure, usually while repairing an earlier failure.

Picture the on-call engineer at Parcelly who finds 40,000 dead letters and presses the button to redrive them all. If the cause was fixed, they flow through. If the cause was not, they fail again, immediately, and flood the system the engineer was trying to heal. If some were already handled by a manual workaround, they are duplicates of work that someone did by hand.

Other forms are equally ordinary. Someone edits a row directly to unblock a customer, bypassing the outbox, so no event is published and downstream services diverge. Someone deploys a producer with a changed pay-

load before consumers can read it. Someone purges a queue in the wrong environment. These are what happens when the repair tools are broad and the safeguards are missing.

I would treat operator actions as part of the failure model and design them like any other component: redrive in small batches with a pause switch, repair actions that go through the same path as normal changes so that events are emitted, and every manual action written to an audit trail. Chapter 8 covers dead letters and redrive, Chapter 12 covers payload changes, and Chapter 13 covers reconciliation, which is how you find out that a human fix went wrong.

What you see: a sudden spike right after someone touched something, and data that disagrees between services with no failed message to explain it.

Losing a whole store

Every failure so far assumed that the stores survive: the database keeps what it committed, and the broker keeps what it acknowledged. The last entry drops that assumption. A cluster or a region is lost. A broker loses data beyond what its replication covered. A consumer's database is restored from a backup taken six hours ago. A producer's database is restored to a point before the last few hundred orders.

These are not one failure, because the damage depends on which store went back in time. If Inventory's database is restored, it forgets reservations made after the restore point, and the broker may no longer hold the events that would recreate them. If Orders' database is restored, it forgets orders that downstream services already acted on, and Payments holds captures for orders that Orders no longer knows about. Neither side raises an error. Each one is consistent with itself and wrong about the other.

One property helps. Because the inbox lives in the same database as the effects, a restore takes both back to the same moment. Messages processed before the restore point are still in the inbox, so a replay skips them, which is correct because their effects survived too. Messages processed after the restore point are the ones at risk, and replaying them is safe: the inbox no longer remembers them, and their effects were lost with it.

External effects are the exception. A capture at the payment provider does not roll back with your database, so those calls need their own idempotency keys, or the replay charges the buyer a second time. The rule is to reconcile before resuming consumption: compare what each service believes, repair the differences, and only then let the flow run.

What you see: nothing, at first, which is the problem. The services are healthy and the dashboards are green, while totals disagree between them. Chapter 13 covers the reconciliation that finds the disagreement, and Chapter 14, Recovery, Disasters and Broker Migration, covers how to restore, replay and resume. Designing for this costs backups that are actually tested, retention that outlasts your recovery time, and the discipline of rehearsing a restore before you need one.

The failure map

Here is the same material in one place. Keep it next to your design reviews.

Failure	What you see	What is actually true	Where it is handled
Network partition or timeout	Caller gets no answer	The operation may or may not have run	Chapter 6, Chapter 7, Chapter 13
Producer crash between write and publish	Orders without events, or events without orders	State and event disagree	Chapter 4, Chapter 5
Lost broker acknowledgment	Duplicate events on the broker	The first publish succeeded	Chapter 7
Broker failover	Publish errors, then re-deliveries; rarely, silent loss	Guarantee depends on acknowledgment settings	Chapter 2, Chapter 13
Consumer crash after work	Duplicate side effects, no errors	Work done, message still pending	Chapter 7
Slow consumer	Growing lag and message age	Consumer alive, falling behind	Chapter 6, Chapter 8, Chapter 13
Retention or expiry while lagging	Nothing; the missing events never arrive	Unread messages were deleted by the broker	Chapter 8, Chapter 13

Failure	What you see	What is actually true	Where it is handled
Poison message or failing dependency	Same message redelivered; stuck partition; DLQ growing in an outage	The failure is deterministic, or the cause is outside the message	Chapter 6, Chapter 8
Consumer bug with no error	Nothing; totals disagree later	The handler succeeded and wrote the wrong result	Chapter 13, Chapter 14
Clock skew	Out-of-order updates, impossible sequences	Timestamps disagree with causality	Chapter 9
Deploy or rebalance	Duplicates and lag at release time	Ownership of work moved mid-flight	Chapter 7, Chapter 13
Operator error	Spike after a manual action; silent divergence	A repair changed state outside the normal path	Chapter 8, Chapter 13
Losing a whole store	Nothing at first; services disagree after a restore	A store went back in time, or its data is gone	Chapter 13, Chapter 14

Two things stand out. First, duplicates appear in the "What you see" column for three different causes (lost acknowledgment, consumer crash, deploy), and redeliveries for a fourth. That is why one pattern, the idempotent consumer, earns a whole chapter. Second, the last column keeps returning to Chapters 13 and 14. Some failures cannot be prevented or even detected at the moment they occur, so the system also needs a way to notice afterwards that two views of the truth disagree.

Writing the failure model down

A failure model is useful only if it is written, current and shared. In my view the practice is more valuable than any one entry, because the act of writing forces the question "what do we assume here?", and the assumptions are where the surprises live.

I would keep it as a short document in the repository, next to the architecture description, with one entry per failure and the same fields each time. The fields mirror the three questions: "where" and "we assume" say what can fail, "observable as" says how you would see it, and "defense" says which guarantee survives. A usable entry looks like this:

```
failure:      consumer crashes after commit, before ack
where:        Inventory, handler for OrderPlaced
we assume:    this happens during every deploy, and at random
otherwise
observable as: redelivered message with the same event_id
effect if unhandled: second stock reservation for the same order
defense:      inbox row inserted in the same transaction as the
reservation
verified by:  test that kills the consumer between commit and ack
owner:        Inventory team
last reviewed: 2026-09-30
```

The fields that matter are the last three. "Verified by" turns the entry from a belief into a claim that can be tested, and Chapter 15 shows how to test it. "Owner" means someone is paged. "Last reviewed" admits that the model rots: services are added, brokers are swapped, and an assumption that was true last year may not be now.

Note: The model should state what you assume cannot happen, too. "We assume the database never loses a committed transaction" is a legitimate entry. It is better to write the assumption down, and know what you would do if it were false, than to leave it implicit.

Use the model in three places. In design reviews, walk each new flow against the list and ask which failures it is exposed to. In incident reviews, find which entry the incident belongs to, and if none, add one. In testing, make each entry a scenario. A failure that appears in the model but in no test is a hope.

Trade-offs and costs

Writing a failure model costs time, and the discomfort of recording what you do not know. A small team with one service and one queue does not need every entry on this list. Start with the three or four failures that would hurt most.

The model also has a boundary. Every defense in this book assumes some failures are out of scope: that the database is durable, that the broker's replication works as configured. Those assumptions are reasonable, and each one is a place where the model can be wrong. Listing them lets you decide whether they are acceptable, which you cannot do while they are

invisible. A bug in your own code is different. It is in scope, as a failure to detect and recover from, because it will happen, and the model has to say how you would notice it and repair what it wrote.

A model with 40 entries will not be read, and one with three will miss cases. I would aim for a list a new team member can read in one sitting.

Designing for every failure at maximum strength is also not the goal. Surviving a broker failover with zero loss costs latency and money, and surviving a double failure costs more. A Notifications email that is sometimes sent twice does not warrant the same defense as a Ledger entry that must never be duplicated. The model's job is to let you spend by risk. If I cannot name the business cost of a failure, I would hesitate to pay a technical cost to prevent it.

The model does not make the system reliable. It makes your assumptions inspectable.

Common mistakes

- **Modeling only crashes.** A crash is tempting to model because it is clean and detectable. The costly failures are the ambiguous ones, such as timeouts and slow consumers. Fix: for every call, write what the caller does when no answer arrives.
- **Treating a timeout as a failure.** It feels natural to read "no response" as "did not happen". Fix: treat it as unknown, and design the operation so that repeating it is safe or checkable.
- **Trusting the broker's acknowledgment as proof.** The word "acknowledged" sounds final. Fix: know what your acknowledgment setting actually guarantees, and reconcile for the cases where it does not.
- **Ordering by timestamp.** `occurred_at` is right there in the envelope and sorting by it takes one line. Fix: order within a partition key by aggregate version, and keep timestamps for humans.
- **Ignoring deploys.** Releases feel like routine, not like failure. Fix: count every deploy as a scheduled consumer crash, and test shutdown while messages are in flight.
- **Leaving operators out.** Repair tools get less review than product code. Fix: put redrive, replay and manual edits in the model, with

batch limits, pause switches and audit trails.

- **Treating a restored database as a clean start.** A restore feels like the end of the incident. Fix: after any restore, reconcile against the other services before consumption resumes, and check external effects that did not roll back.
- **Retrying a failure that will not change.** Retries feel diligent. Fix: separate failures that depend on timing from those that depend on the message, and send the second kind to the dead-letter path early.
- **Writing the model once.** A document written during a design phase is a snapshot. Fix: assign an owner, review it after incidents, and date each entry.

Key takeaways

- A timeout means "I stopped waiting", not "it failed", so design operations that remain correct whichever way the unknown resolves.
- Each seam has its own crash windows, and the damage is silent. A dual write leaves orders without events, or events about orders that do not exist, and a consumer that crashes after committing but before acknowledging gets a duplicate on every redelivery, which makes idempotent consumers a requirement.
- A broker's acknowledgment is evidence with a boundary set by replication and flush settings, so know the boundary and plan to reconcile what falls outside it.
- Slow consumers and deploys cause duplicates and stalls without any component being broken, so bound the load, measure lag and message age, and stop consumers gracefully.
- A message that fails every time, or a dependency that fails for every message, needs a different response from a chance failure, so classify before retrying and keep dead-lettering for what will not recover.
- Wall-clock timestamps disagree across machines, so order events within a partition key by aggregate version and use `occurred_at` for humans.
- Operators and restores are part of the system: a restored store has gone back in time, so reconcile before resuming consumption, and

keep a written failure model that names owners, tests and review dates so repairs do not become the next incident.

The first failure to remove is the one in the producer, where a state change and its event can come apart without anyone noticing. Chapter 4 begins Part II with the transactional outbox, which closes that window.

PART II

Publishing Reliably

Chapter 4 — The Dual-Write Problem and the Transactional Outbox

Imagine Parcelly a few weeks after it launched seller promotions. A support engineer opens a ticket that reads, in full: "Buyer says she completed checkout for order 81724 but the seller never got a shipping label. The order shows as placed." The engineer checks the Orders database and finds the row, status `PLACED`, created at 14:02:11. Then she checks Payments, and there is no authorization for that order. Inventory has no reservation either. As far as the rest of Parcelly is concerned, the order does not exist.

She searches the Orders logs around 14:02 and finds a deploy marker. The service was restarted 200 milliseconds after it committed the order and before it published `OrderPlaced`. The code did exactly what it was written to do: save, then publish. Nothing threw an error, because the process that would have thrown it was already gone. Suppose a quick query shows 11 other orders in the same shape, all within the same minute.

The team fixes the 11 by hand and adds a note to the runbook. Nobody feels good about it. The code is not wrong in any way a reviewer would flag, and yet it can lose work whenever a process dies at the wrong moment. A bigger machine will not change that, and neither will a more careful developer. The question for this chapter is whether a service can change its own database and tell the rest of the system about it in a way that cannot come apart, without a transaction that spans the database and the broker.

The problem, and why it is hard

Chapter 1 defined the dual write: two systems updated without a shared transaction. Orders must write a row to its relational database and publish a message to the broker. The database guarantees atomicity for the first.

The broker guarantees durability, in some configuration, for the second. Nothing guarantees that both happen or neither does.

Chapter 3 sketched the producer crash windows in a diagram. Here I want to walk through them as a table, because the order of the two writes is a design decision most teams make without noticing, and every choice has a failure waiting behind it.

Every ordering breaks

Take the two obvious sequences and ask, for each possible point of failure, what the world looks like afterwards. That is the same exercise as writing down the failure semantics of the seam between the database and the broker: what can fail, how you would observe it, and which guarantee survives it. The table below is that exercise for the producer.

Sequence	Failure point	State of the data-base	State of the broker	Damage
Commit, then publish	Crash after commit, before publish	Order saved	No event	Lost event. Downstream never learns of the order
Commit, then publish	Publish fails or times out, process survives	Order saved	Unknown	Caller must retry or give up. Giving up loses the event
Commit, then publish	Broker accepts, ack lost	Order saved	Event present	Retrying publishes a duplicate
Publish, then commit	Crash after publish, before commit	No order	Event present	Phantom event. Downstream acts on a fact that never became true
Publish, then commit	Commit fails (constraint, deadlock, timeout)	No order	Event present	Same phantom event, now with a visible error in the producer
Publish inside the transaction, commit last	Commit fails after publish	No order	Event present	The transaction block gives the code a false sense of safety, and the broker is not part of it

The last row deserves a comment, because it is the one that looks safest. Many frameworks let you publish from inside a database transaction block. It reads as if the publish were part of the transaction. It is not. The broker call happens immediately and cannot be rolled back, and the commit still happens afterwards. This is the publish-then-commit ordering with better indentation.

The mirror image is the "after commit" callback that many frameworks offer, a hook that runs once the transaction has committed. Teams often believe it is safe, because the publish only happens for committed data. It is the first row of the table: commit, then publish, with the intent held in process memory. A crash between the commit and the hook loses the event, and nothing records that it was ever owed.

There is also a fourth shape that the table does not capture, which is a slow broker. If publishing happens inside the transaction, the database holds locks and a connection while it waits for the broker. A broker that is slow for a minute turns into a database that is slow for a minute. You have coupled the availability of two systems that were supposed to be independent.

Why retrying does not help

The tempting repair is a retry loop around the publish. Retries help with the second row of the table, where the process survives and can try again. They do nothing for the first row, where the process is gone, and they create the duplicate in the third row. Chapter 6 covers retries properly. For now, notice that a retry loop lives in the memory of the process, and the failure we care about is the process disappearing.

The second tempting repair is a periodic job that compares the database with the broker and republishes what is missing. Some teams build this, and I think it is a reasonable safety net, which Chapter 13 returns to as reconciliation. As the primary mechanism it has two problems. The broker is rarely queryable by business key, so "what is missing" becomes an expensive guess. And the job itself performs the same dual write, so it can fail in the same ways, only less often and with more confusion.

Why not a distributed transaction

If the problem is two systems without a shared transaction, the classic answer is to give them one. **Two-phase commit (2PC)**, standardized for databases and transaction managers as XA, coordinates several resource managers. In the first phase each participant prepares, which means it promises it can commit and holds its locks. In the second phase a coordinator tells every participant to commit, or to roll back if anyone could not prepare.

On paper this closes the window. In practice it is rarely the answer for a broker and a service database, for reasons that are mostly operational.

- **Support is uneven.** Many popular brokers do not implement a transaction-manager interface at all. Where one does, the guarantees and the failure handling differ from the database's, and you inherit both sets of rules.
- **Participants hold locks while the coordinator decides.** If the coordinator crashes between the two phases, prepared transactions stay in doubt. Rows stay locked until someone, or something, resolves them. The in-doubt window is exactly the kind of rare, long failure that nobody rehearses.
- **The coordinator becomes a component you must run, secure and recover.** It needs durable logs of its own, and a recovery procedure that has to be right.
- **Availability multiplies.** A transaction that needs the database and the broker and the coordinator to be healthy together is available only when all three are. The independence that made the broker attractive is gone.
- **Throughput and latency suffer.** Two round trips, forced writes at every participant, and locks held across network calls add up.
- **It does not extend to the outside world.** An email provider or a payment provider will not join your transaction. Even a team that adopts 2PC for the broker still needs another answer for every external call.

I would not rule 2PC out in every case. A team that controls both resources, runs a mature transaction manager and has the operational skills to resolve in-doubt transactions can make it work. For most teams building a marketplace out of services, each with its own database and a broker

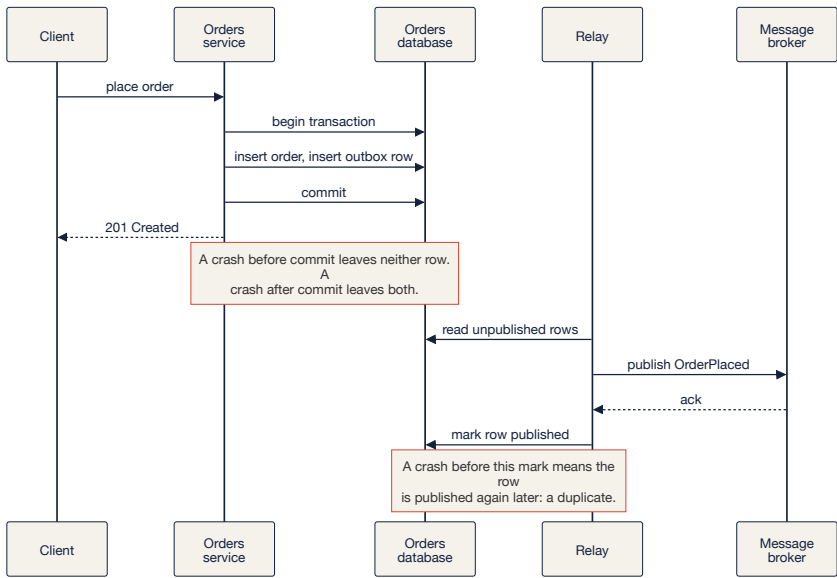
that exposes no XA interface, the cost is high and the benefit is narrow. What they want is smaller than a distributed transaction: they want the state change and the intent to publish to commit together. That is a local problem, and local transactions are well understood.

The transactional outbox

The idea is to turn the dual write into a single write plus a later, repeatable move. Instead of publishing to the broker inside the request, the service writes a row describing the event into a table in its own database, in the same local transaction as the state change. This table is the **transactional outbox**. From here on, the book calls it the outbox.

A separate process, the **relay**, reads outbox rows and publishes them to the broker. Chapter 5 is about the relay. In this chapter we treat it as a function that eventually delivers each row at least once, and look at what that buys us.

The reasoning is short. Either the order and its outbox row commit together, or neither does, because they live in one database and one transaction. Once the row exists, the intent to publish is durable. A crash after commit no longer loses the event, because the row is still there when the service, or its replacement, starts again. A crash before commit leaves no order and no row, so no phantom event exists.



Look at the two notes in the diagram. The first shows the window we closed. The second shows the window we kept: the relay can publish and crash before it records that it did. The outbox does not remove duplicates, it moves the failure to a place where it is bounded and handled. I come back to this below.

The table

The outbox follows the shape the rest of the book uses. It carries every envelope field, so the relay can build the event without asking the producing service anything. This chapter adds four columns beyond the envelope, each for a reason.

```
table outbox:
  id          bigint primary key      // allocation order, not
commit order
  event_id    uuid unique not null    // stable across
redelivery
  aggregate_type text not null        // "order", "shipment"
  aggregate_version bigint not null    // aggregate's version,
bumped in this transaction
  event_type   text not null          // "OrderPlaced"
  schema_version int not null
  occurred_at  timestamp not null     // time of the fact
  created_at   timestamp not null default dbNow() // insert time,
database clock
  producer     text not null          // "orders"
  correlation_id text not null
  causation_id  text not null
  partition_key text not null         // aggregate id, e.g.
"order-81724"
  payload      json not null
  headers      json null              // transport hints, trace
context
  published_at  timestamp null        // null means not yet
published
```

A few of these choices need a defense.

- **id and event_id are different things.** The `event_id` is a UUID the consumers will use as the idempotency key, and it must not change when the event is published twice. The `id` is a database-assigned sequence number the relay can use to find rows. It is not a statement about commit order, and the next section shows why.
- **aggregate_version is the ordering source of truth.** It is the version of the aggregate, incremented in the same transaction that writes the

row. For Parcelly orders it is the `order_version`. Consumers that need per-aggregate order compare this number, not the `id` and not the arrival order. Chapter 9 shows how a consumer uses it. The standard envelope has no field for it, so when the relay builds the event it copies the column into the payload as `order_version`, and that is where consumers read it.

- **aggregate_type and partition_key are separate.** The `partition_key` is what the broker uses for ordering, and it is the aggregate id, such as `order-81724`. The `aggregate_type` tells the relay, and a human reading the table, which kind of thing the key refers to, and it is a convenient way to route rows to different topics.
- **created_at is separate from occurred_at.** The first is when the row was written, and the second is when the fact happened. Usually the two are within milliseconds. They diverge when a service records a fact it learned late, such as a carrier scan that arrives after the event. A relay-lag metric should use `created_at`, because lag is a property of the table, not of the business. Let the database fill it through a column default instead of calling `now()` in application code, so the clock is one clock. It is still insert time, not commit time: a long transaction makes the row look older than the relay's view of it, and that part of the lag is not the relay's fault.
- **headers holds transport concerns.** Trace context and routing hints go here, so the payload stays the business fact. A team that cannot decide whether a field is data or a transport hint should read Chapter 11 before adding it to the payload.
- **published_at is nullable, and null is meaningful.** A row with no `published_at` is a row the relay still owes the broker. Some teams use a `status` column instead, with values such as `PENDING`, `PUBLISHED` and `FAILED`. The two designs carry the same information. I prefer the timestamp because it also tells you when, which is useful when you ask how late the relay was, and because there is less to get wrong. If you need a way to park rows that fail permanently, add a `failed_at` column then, not before.

That moment comes sooner than it sounds. A row the broker will never accept, such as an oversized payload or one the serializer rejects, blocks its key, and in the simplest relay the whole table, until someone repairs it.

The first time that happens, you need a `failed_at` column (or a `FAILED` status, if you chose status columns) and an alert on parked rows. Chapter 5 shows how the relay uses it.

An index on the unpublished rows, in insert order, is the only index the relay needs. Without it, the relay scans a table that grows forever. Chapter 5 covers cleanup.

Writing in the same transaction

The producing code is short, and the shortness is the point. The following function is the whole change to Orders' request path.

```
function placeOrder(request):
  order = buildOrder(request)
  // a new order starts at version 1
  order.version = 1

  transaction:
    orderInsert(order)
    outboxInsert({
      event_id:      newUuid(),
      aggregate_type: "order",
      event_type:    "OrderPlaced",
      schema_version: 1,
      aggregate_version: order.version, // 1 for a new order
      occurred_at:    now(),
      producer:       "orders",
      correlation_id: order.id,
      causation_id:   request.call_id,
      partition_key:  order.id,
      payload:        orderPlacedPayload(order)
    })
  // commit point: the order and its event exist together or not at
  all

  return order
```

Three details matter more than they look.

First, nothing in the function talks to the broker. A broker outage does not fail order placement, and a broker that takes a minute to respond does not hold a database transaction open. That is a real gain in availability: Orders can accept orders while the broker is down, and the outbox grows until the relay can drain it.

Second, the `event_id` is created before the commit and stored in the row. Every later republication of this event carries the same id. This is the property consumers depend on, and it is why generating the id in the relay, at publish time, would be a mistake.

Third, the payload is built inside the transaction, from the state the transaction has written. If the payload were built later by re-reading the order, a concurrent update could change it, and the event would describe a fact that did not exist at the time. An event is a snapshot of what happened, and the snapshot is taken in the same place as the change.

The pseudocode assumes that the order insert and the outbox insert use the same database connection and the same transaction. This is broken by accident more often than it sounds. A data-access layer that opens a second connection for the outbox, or that auto-commits the outbox insert, silently reinstates the dual write, now between two tables. The test for this belongs in your suite: force a failure after the order insert and assert that no outbox row exists.

Commit order, `id` order and the aggregate lock

A new order has no competitor, so `placeOrder` needs no lock. Every later change to an existing order does, and the lock has to come first. Here is `cancelOrder`, which emits `OrderCancelled`.

```
function cancelOrder(order_id, reason):
  transaction:
    // lock the aggregate row before anything else is written
    order = orderSelectForUpdate(order_id)
    order.version = order.version + 1
    orderUpdate(order, status: "CANCELLED")
    outboxInsert({
      event_id:      newUuid(),
      aggregate_type: "order",
      aggregate_version: order.version,
      event_type:     "OrderCancelled",
      partition_key:  order.id,
      payload:        orderCancelledPayload(order, reason)
      // remaining envelope fields as in placeOrder
    })
  // commit point: state, version bump and event together
```

The lock is what makes the version trustworthy, and it is also the only thing that makes `id` order mean anything for one aggregate. Without it, take two transactions on the same order. Transaction B inserts its outbox row and gets `id` 101, then blocks on the order row that transaction A holds. A inserts `id` 102 and commits, and B commits afterwards. Commit order is A then B, and `id` order is B then A. A relay that sees both rows in one poll publishes them reversed. With the lock taken first, B cannot insert until A has committed, so the ids of one aggregate follow its commit order.

Even then, I would not make consumers depend on `id`. Rows of different aggregates interleave, a parallel relay reorders across batches, and nobody outside the Orders database can see the `id`. The `aggregate_version` travels with the event, as `order_version` in the payload, and says in what order the facts happened to this order. Chapter 5 returns to what a relay can and cannot preserve, and Chapter 9 shows the consumer side.

What the relay owes

The relay's contract is deliberately small. It selects unpublished rows, publishes each to the broker, and marks it published once the broker confirms. The simplest version is below, and Chapter 5 covers the production concerns it ignores, such as batching, locking, ordering and failover.

```
function relayOnce():
    // oldest first
    rows = outboxSelectUnpublished(limit: 100)

    for each row in rows:
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
eventFrom(row))
        if result is failure:
            // stop: publishing later rows would overtake this one for
its key
            return
        // if the process dies here, the row stays unpublished
        // and will be published again: this is the duplicate window
        outboxMarkPublished(row.id, now())
```

The comment inside the loop is the whole guarantee. The row is marked only after the broker confirms, which is why the call waits for the acknowledgment and why a failure stops the batch. A crash between the publish and the mark leaves the row unpublished. On the next pass, the

relay publishes it again. The alternative, marking first and publishing second, would make the failure a lost event instead of a duplicate. The outbox picks the safer failure on purpose.

What the outbox guarantees

The outbox gives you three things and takes away one.

- **Atomicity between state and intent.** The state change and the decision to publish commit together. There is no committed order without an outbox row, and no outbox row without a committed order.
- **Durability of the intent.** Once the transaction commits, the event is as durable as the database. The relay can be down, the broker can be down, the producer can be replaced, and the row waits.
- **At-least-once publication.** Every committed row is eventually published, assuming the relay and the broker eventually recover, the relay finds rows by state and not by an id cursor, the cleanup purges only published rows, and a row the broker will never accept is parked and alerted on and not left to block forever (all of this is Chapter 5). A row can be published more than once.
- **Not guaranteed: exactly-once publication.** The relay cannot atomically publish to the broker and mark a row in the database, for the same reason the application could not. The window moved from the request path to the relay, where it is smaller, observable and repairable.

I want to be careful with that summary. The outbox is not a claim that events are never lost. It is a claim that the producer-side loss window of the dual write is closed, provided the application writes through the outbox for every state change that must produce an event. A developer who updates the table by hand, or by a path that skips the outbox, opens the window again. Chapter 3 listed this as a human mistake, and it is the most common way a team with an outbox still loses events.

The duplicate becomes the consumer's problem

Because publication is at-least-once, every consumer of an outbox-published event must tolerate duplicates. This is not a flaw to be engineered away, it is the price of the guarantee, and it is worth being plain about

who pays it.

The producer team does the lighter part. They add a table, write a row, and run a relay. The consumer teams do the harder part, because they must recognize an `event_id` they have already seen and avoid repeating its effect. Payments must not authorize twice. Notifications must not send the same email twice. Ledger must not post the same entry twice.

Warning: Adopting an outbox without idempotent consumers trades a lost-event problem for a duplicate-effect problem. For some events, such as a double charge, the second problem is the more expensive one. Ship the two together, and see Chapter 7 for how a consumer recognizes repeated work.

I would also say this to producer teams in plain words: the outbox is a contract with your consumers. The `event_id` must stay stable for every republication of the same fact, and the contract should be written in the event documentation, not left implicit.

A short failure walk-through

To check the design against Chapter 3's windows, walk Orders through the crashes again.

1. The process dies before the transaction commits. No order, no outbox row, no event. The client sees an error or a timeout and may retry the request, which is a separate problem for the API layer, covered by the idempotency discussion in Chapter 7.
2. The process dies right after the commit. Order and row both exist. The relay publishes the row on its next pass. The client may not have received its response, but the system is consistent.
3. The relay dies after publishing and before marking. The event is in the broker and the row is unpublished. The next pass publishes it again. Consumers see a duplicate with the same `event_id`.
4. The broker is unavailable. The relay fails to publish, leaves the rows unpublished, and tries again later. Orders keeps accepting requests. Lag grows, and so should an alert.

5. The service loses its connection during the commit and does not know the outcome. Either the order and its row exist or neither does, and the service cannot tell which. It must not guess. The safe answer is to let the client's retry carry an API-level idempotency key, so that whichever state the database is in, the retry converges on one order and one event (Chapter 7).

Each case ends either with consistency or with a duplicate, and none ends with a phantom event or a silent loss. That is the whole of what the outbox claims, and it is enough to justify it.

One duplicate sits upstream of the outbox, and no downstream defense sees it. When a client retries after a lost response and the API has no key of its own, Orders creates a second order and a second `OrderPlaced` with a fresh `event_id`. Every consumer's inbox then sees two different events, because they are. The outbox did its job twice. The fix belongs at the API, with the `Idempotency-Key` approach in Chapter 7.

Alternatives and neighbors

The outbox is not the only way to stop the dual write. Two other designs deserve a fair comparison, since a team might reasonably prefer either.

Listen to yourself

In the **listen-to-yourself** pattern, the service does not write to its database first. It publishes the event to the broker, then consumes its own event and applies the state change in the handler. The broker becomes the single write, and the database becomes a projection of what the service has heard.

This removes the dual write, because there is only one write. It also changes the shape of the service. The client's request returns before the state exists, so the caller cannot read its own write unless the system offers a way to wait. Validation must happen before publishing, since an event, once published, is a fact, and an invalid one cannot be rejected afterwards without a compensation. And the handler that applies the event must itself be idempotent, which brings back the same consumer burden.

I would consider this for services that already think in events and can tolerate eventual consistency in their own reads. For Orders, where the buyer's browser expects to see the order right after clicking the button, I would not. The read-your-own-write gap is a product problem that the team would end up solving with workarounds.

Event sourcing

In **event sourcing**, the events are the source of truth. The service stores the sequence of events per aggregate, and current state is derived from them. The write to the event store is the state change, so there is nothing to keep consistent with a separate event. Publishing to the broker then reads from the event store, which is a relay by another name.

Event sourcing solves the dual write by changing what the database is. It also gives you a full history and the ability to rebuild projections. It costs a different programming model, schema evolution of stored events that you can never rewrite casually, and a learning curve for the whole team. It is a significant architectural commitment and I would not adopt it only to avoid an outbox. If you already use it, you still need a relay, and everything in this chapter about at-least-once publication applies.

Comparing the options

Option	What it removes	What it costs	Fits best when
Commit, then publish	Nothing	Silent loss on crash	A lost event is acceptable, such as best-effort metrics
2PC / XA	The window, in theory	Locks, coordinator, uneven support	You control both resources and run the recovery yourself
Transactional outbox	The producer-side window	An extra table, a relay, duplicates	State lives in a relational database and events must not be lost
Listen to yourself	The second write	Eventual consistency in own reads, early validation	The service is already event-first
Event sourcing	The separate state write	A different model of state	History is itself a requirement

For a service that already stores its state in a relational database and needs to tell others about it, the outbox is the smallest change. That is why most of this book assumes it.

Trade-offs and costs

The outbox is cheap to describe and not free to run. These are the costs I would put in front of a team before they commit.

More moving parts. There is a table, a relay process, and a lag to monitor. When something breaks, a new place to look exists. The relay is a component that someone owns, and it needs deploys, alerts and a run-book. Chapter 5 gives the detail.

Extra database load. Every event is an insert in the same transaction as the business write, and every publish is a read and an update.

Transactions get a little longer, and write volume goes up. For most services the increase is small compared with the rest of the work. A service writing very high event volumes should measure it on its own hardware instead of trusting my "small".

Added latency. The event reaches the broker after the transaction commits, and after the relay notices the row. With polling, the delay depends on the polling interval. With a log-based approach it can be shorter, and Chapter 5 compares the two. In either case, the outbox is not the choice for flows where the event must reach the broker within a few milliseconds of the request, and you should know your latency budget before choosing.

Duplicates by design. Covered above. If a consumer cannot be made idempotent, then the outbox improves nothing for it, and the team has to decide whether that consumer should exist in this form.

Ordering is only partly preserved. The relay reads rows by `id`, and `id` order is not commit order unless every transaction locks the aggregate row before it inserts the outbox row. That lock serializes writers of one aggregate, which is a cost of its own on hot aggregates. Parallel relays make the order worse, and the lock does not help across aggregates. The `aggregate_version` column gives consumers something to check, and it costs a version bump on every write. The outbox gives you a place to enforce order, not order for free. Chapter 5 and Chapter 9 return to this.

Growth and cleanup. Published rows accumulate. Without deletion or archiving, the table grows until the relay's query slows down. Retention is a decision, and it interacts with reconciliation, since old rows are also your evidence of what was published.

Only works for state in a database that supports transactions. If the state lives in a store without multi-row transactions, you need another approach, such as a store-specific change stream or an event-sourced design. A store that offers atomicity only within one document has a variant of its own: keep the pending events inside the aggregate document, and let the relay drain them from there.

When not to use it

I would not add an outbox when:

- the event is best-effort and its loss is acceptable, such as a metric or a low-value analytics hint
- the service holds no durable state of its own, so there is nothing to commit alongside the event
- the system is small enough that a single process owns both the state and its consumers, and a direct call is simpler
- the team cannot yet make consumers idempotent, in which case I would fix that first, since the outbox raises the stakes of that gap

Note: "We use an outbox" is a statement about one service's producer path. It says nothing about whether consumers are safe, whether ordering holds or whether events are well designed. Treat it as one pattern in a set, not as a seal of reliability.

Common mistakes

- **Writing the outbox row in a separate transaction.** It is tempting because a framework hook or an event bus abstraction hides the connection. The two writes then become a dual write between two tables of the same database, and the window is back. Use one connection and one transaction, and test the rollback.

- **Publishing from an "after commit" hook.** It is tempting because it only fires for committed data and needs no table. It is commit-then-publish with the intent in memory, so a crash between the two loses the event. Write the row in the transaction and let the relay publish.
- **Inserting the outbox row before locking the aggregate.** The code works in every test with one writer. Under concurrency, `id` order and commit order diverge for one order, and the relay can publish its events reversed. Lock the aggregate row first, bump its version, then insert.
- **Publishing from the request path and keeping the outbox as a backup.** It feels like belt and braces. It produces two publishers, duplicates with different timing, and ordering you cannot reason about. Let the relay be the only publisher.
- **Generating `event_id` in the relay.** It looks harmless, since the relay is the one building the message. A republication then gets a new id, and consumers cannot recognize it as a duplicate. Generate the id when the row is written.
- **Marking the row published before the broker confirms.** It reduces duplicates and feels tidier. It converts a duplicate into a lost event, which is the worse failure. Mark after confirmation, and accept the duplicate.
- **Building the payload after the commit.** Re-reading the aggregate to fill in the event seems to keep the producing code simple. Another transaction can change the data in between, and the event no longer matches the fact. Build the payload from what the transaction wrote.
- **Letting some code paths bypass the outbox.** A migration script, an admin tool or a hand-run SQL fix changes state and emits nothing. Route every state change that matters through the same function, and review manual repairs for the events they should have produced.
- **Forgetting cleanup.** The table works for months, and then the relay's query slows down at the worst time. Decide retention when you create the table.
- **Treating the outbox as the end of the reliability story.** It closes one window. Consumers still need idempotency, retries and a place for messages that never succeed.

Key takeaways

- A dual write cannot be made safe by choosing a better order: commit-then-publish loses events, publish-then-commit invents them, and a publish inside a transaction block is still the second one.
- Two-phase commit can close the window in principle, but for a service database and a broker it usually costs more in locks, coordinator operations and availability than the problem is worth.
- The transactional outbox turns the dual write into one local transaction plus a repeatable move, by writing the event as a row in the same transaction as the state change.
- The outbox row carries the full event envelope, a stable `event_id` created at write time so every republication can be recognized, and an `aggregate_version` bumped under the aggregate's lock, because `id` order is not commit order.
- The outbox guarantees atomicity of state and intent, durable intent and at-least-once publication, and it does not guarantee that a message is published exactly once.
- Duplicates are the price of the guarantee, so an outbox without idempotent consumers trades one failure for another.
- Listen-to-yourself and event sourcing also remove the dual write, but they change how the service reads and stores its state, so I would choose them for what they give you and not merely to avoid a relay.

Publishing is only half of the move: something must read those rows and carry them to the broker without losing, reordering or flooding it. Chapter 5 covers the relay, polling against change data capture, and how to run the outbox in production.

Chapter 5 — Relays, Polling and Change Data Capture

Imagine a Wednesday morning at Parcelly. The Orders team shipped the transactional outbox two weeks ago. Every `OrderPlaced` event is now written to the `outbox` table in the same local transaction as the order itself, and the dual-write window from Chapter 4 is closed. The team considers the problem solved and moves on to the next ticket.

The relay is a small process that wakes up every second, reads rows with an `id` greater than the last one it saw, publishes them and remembers the new highest `id`. It runs as a single instance, and nobody has thought about it since the day it was deployed.

Then a routine deploy replaces the relay at the wrong moment. The old instance has published a batch of 200 rows to the broker but has not yet saved its new position. The new instance starts from the last saved position, reads the same 200 rows and publishes them again. Inventory reserves stock a second time for orders it has already handled. A week later, a cleanup job is added and quietly deletes rows the relay has not yet read. The month after, a long-running transaction commits late and its rows never leave the table.

None of these failures is in the outbox pattern itself. All of them are in the part that moves rows to the broker. How do you run that part so that it is slow to fail, quick to recover, and honest about the duplicates it can still produce?

The problem, and why it is hard

The transactional outbox gives you one guarantee: if the order was committed, the intent to publish its event was committed with it. It says nothing about the second half of the journey. Somebody has to read the rows,

hand them to the broker, and remember that it did. That somebody is the **relay**, the process that moves outbox rows to the broker.

A relay does three things: it finds rows that need publishing, it publishes them, and it records that it has done so. Those are three separate steps, and the broker and the database do not share a transaction. The same dual-write problem we removed from the producer reappears inside the relay, with a narrower window.

The crash windows are few enough to list. The table is the relay's failure semantics: what can fail, how you would observe it, and which guarantee survives it.

Crash happens	State of the row	Result
Before publish	Unpublished	Row is picked up again, event published once. Safe.
After publish, before marking	Unpublished, but the event is on the broker	Row is picked up again, event published twice. A duplicate.
After marking	Published	Nothing to do. Safe.

There is no ordering of the three steps that closes the middle row. If the relay marks first and publishes second, a crash in between loses the event, which is the failure the outbox exists to prevent. So the relay publishes first and marks second, and accepts that a crash between them produces a duplicate. This is the same trade the outbox chapter made: at-least-once delivery, with the duplicate handling pushed to consumers (Chapter 7).

The first design question is therefore not how to avoid duplicates. It is how to keep the window small, how to make sure the rows come out in a sensible order, and how to see the relay falling behind before a customer does. The second design question is a choice between two mechanisms for reading the rows: polling the table, or reading the database's own transaction log. Each moves the difficulty somewhere else.

The polling relay

A **polling relay** queries the outbox table on an interval, publishes whatever it finds, and marks those rows as published. It is the simplest mechanism that works, and I would start here unless the numbers say otherwise.

The basic loop

```
function relayLoop():
  while running:
    // oldest first
    rows = selectUnpublished(limit: 100)
    if rows is empty:
      sleep(pollInterval)
      continue

    // defined below: only broker-confirmed ids
    ackedIds = publishAked(rows)
    // a crash here leaves confirmed rows published but unmarked:
    // they will be published again, which consumers must tolerate

    // one update for the batch
    markPublished(ackedIds, now())
```

The query behind `selectUnpublished` is where most polling relays go wrong, so it is worth a closer look. The obvious shape is "give me rows with an `id` greater than the last one I handled". It looks efficient, and it is unsafe.

Why a cursor on `id` is not safe

Row `ids` are usually assigned when the insert runs, not when the transaction commits. Imagine two `Orders` transactions. Transaction A inserts an `outbox` row with `id` 101 and then spends some time calling a slow function before it commits. Transaction B inserts `id` 102 and commits immediately. The relay polls, sees 102, publishes it, and moves its cursor to 102. A then commits. Row 101 is now behind the cursor and will never be read.

The failure is quiet. Nothing errors, no log line appears, and the event is never published. This is the same loss the `outbox` was built to remove, reintroduced by an optimization.

The fix I prefer is to stop remembering a position and to query by state instead: select rows where `published_at` is null, ordered by `id`. A row that commits late is still unpublished when the next poll arrives, so it is found. The cost is an index on unpublished rows, and a table that must stay small enough for that query to be cheap. A partial index that covers only unpublished rows keeps the query fast regardless of how much history the table holds, if your database supports one. If it does not, the retention work later in this chapter matters more. A partial index does not remove all the

cost: in databases that keep old row versions, every row is inserted, updated and later deleted, and that churn leaves dead rows for the engine to clean up.

State-based reading fixes the loss. It does not fix the order, which is a separate question that the section on preserving order answers.

Batching and the polling interval

Two settings shape the relay's behavior: how many rows it takes per poll, and how long it sleeps when there is nothing to do.

A larger batch amortizes the cost of the query and the marking update, and it improves throughput. It also widens the crash window: a relay that dies halfway through a batch of 500 leaves up to 500 events to republish. A smaller batch republishes less after a crash and, with the row claiming described below, holds locks for less time. Say, 50 to 200 rows is a reasonable first guess; measure before you tune.

The polling interval is a floor on latency. With a one-second interval, an event can wait nearly a second before the relay even looks at it. Shortening the interval cuts latency and raises the number of empty queries the database has to serve. For Parcelly, a second or two of delay between `OrderPlaced` being committed and being published is invisible to buyers. For a flow where a person is waiting at a screen, it may not be.

Marking after publish

The marking step deserves its own care. Publish each row, collect the broker's acknowledgment, and mark only the rows the broker has confirmed. Do not mark a row whose publish has not been confirmed, even if the call returned without an error, because many client libraries send asynchronously and report failure later.

```

function publishAcked(rows):
    ackedIds = []
    for each row in rows:
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
        buildEnvelope(row),
                                key: row.partition_key)
        if result is failure:
            // stop here: later rows with the same partition_key must
            not overtake this one
            break
        ackedIds.append(row.id)
    // only these rows may ever be marked
    return ackedIds

```

This function never touches the table. It returns the ids the broker confirmed, and every relay in this chapter marks exactly those ids and no others. A row that failed, and every row after it in the batch, stays unpublished and is read again on the next poll. Publishing and marking are kept apart on purpose, so that marking can be guarded by a lock or a lease in the one place where it happens. Each event goes to the topic of its aggregate type, with `partition_key` as the message key, the same routing Chapter 4 uses.

The `break` is a deliberate choice, and it is about ordering, which comes up again below. Skipping a failed row and publishing the rows after it is faster, and it can reorder events for one aggregate. Stopping is slower and safe. I would stop, and let monitoring tell me the relay is stuck. This sketch stops the whole batch for simplicity; blocking only the affected key is the refinement described under preserving order below.

One active relay or several

The simplest topology is one relay instance doing all the work. Its ordering behavior is easier to reason about, because it reads in `id` order and publishes in that order, with the caveat about `id` and commit order in the section on preserving order below. Its weakness is availability: if that instance dies, nothing is published until another one starts. For many systems, a restart within a few seconds is acceptable, and a single relay with a supervisor that restarts it is the right answer.

When one relay cannot keep up, or when a gap in publishing is unacceptable, you run several. Now you have a new problem: two relays must not publish the same row at the same time. The standard tool is a row lock

with skip semantics. Many relational databases offer a way to select rows for update while skipping rows that another transaction has already locked, commonly described as `SKIP LOCKED`-style locking. The relay claims a batch inside a transaction and the other relays pass over those rows.

```
function claimAndPublish():
  transaction:
    rows = selectUnpublished(limit: 100, lockRows: true,
skipLocked: true)
    ackedIds = publishAked(rows)
    // unconfirmed rows are not marked; releasing the locks lets
another poll retry them
    markPublished(ackedIds, now())
    // commit point: row locks are released here, and the marks become
visible
```

This works, and it carries costs that are often missed. First, the transaction stays open while the relay talks to the broker, so a slow broker means long-held locks and a growing pile of open transactions. In databases that keep old row versions for concurrent readers, a long-open transaction also holds back the cleanup of dead rows, which hurts exactly the table that churns most. Second, if the transaction aborts after publishing, through a dropped connection or a failed mark, every row it published is published again. A shorter alternative is a brief claim transaction that stamps the rows with a claim time, a publish outside any transaction, and a separate mark transaction, with a claim timeout so a dead relay's rows become claimable again. It costs a state to reason about. Third, parallel relays lose global ordering. Relay A may claim rows 1 to 100 while relay B claims 101 to 200, and B may finish first. For events about different orders, this does not matter. For two events about the same order, it does.

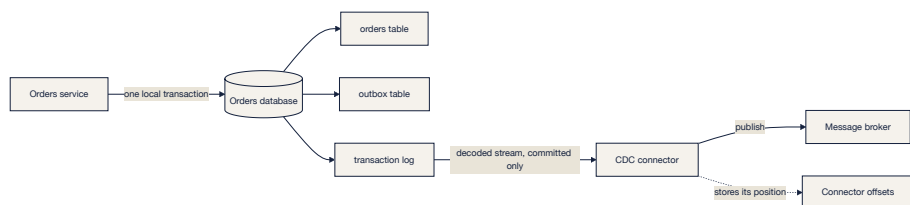
The remedy is to partition the work by `partition_key` instead of by row. Assign each relay a set of keys, for example by hashing `partition_key` into a fixed number of buckets and giving each bucket to exactly one relay at a time. Within a bucket, rows are handled in order by one worker, and across buckets they run in parallel. This is the same idea as partitions on a log-based broker, applied one step earlier. The word "exactly" needs the lease from the failover section below, and bucket handover during a rebal-

ance is the moment per-key order is most at risk, because the old and the new owner may publish for the same key together. Chapter 9 explains why per-key order is the order you can realistically keep.

Change data capture

The second mechanism does not query the outbox table at all. **Change data capture (CDC)** reads a database's own transaction log and turns committed row changes into a stream of change records. A CDC connector, of which Debezium is the familiar example, attaches to that log, notices that a row was inserted into `outbox`, and publishes a message to the broker.

The table keeps its role, because the transactional guarantee still comes from writing it in the same transaction as the state change. What changes is who reads it. No application process polls. The database already records every committed change in an ordered log for its own recovery and replication, and the connector reads that log the way a replica would.



Look at the path from the transaction log to the broker. The change stream that the database exposes to connectors (PostgreSQL calls it logical decoding, MySQL exposes its binary log) contains only committed transactions, and they appear in commit order. That removes the late-commit problem from the polling section entirely: the log is already a total order of commits, so a row that commits late appears late, in the right place.

How it behaves

A few properties follow from the design.

Ordering follows commit order. Events appear in the order the transactions committed. A connector that publishes with the row's `partition_key` as the message key keeps per-key order on a log-based

broker, provided the producer inside it is configured so that a retry cannot reorder requests that are in flight together. Check your client's settings; the same caveat applies to a polling relay that publishes asynchronously.

Latency is low and steady. There is no polling interval. The connector sees a change shortly after commit, so the delay is usually small and does not depend on how often anything wakes up.

The connector tracks its own position. It records how far into the log it has read, and after a restart it resumes from there. Connectors commonly save the position periodically and not after every event, so a crash can re-publish every event since the last saved position. CDC is at-least-once, like polling. The duplicate window has moved, not vanished.

The published flag is optional. Because the connector does not need to find unpublished rows, the table needs no `published_at` column for the relay's sake. Many teams using CDC delete each outbox row soon after it is inserted. The insert has already been recorded in the log, so the connector still sees it. The table stays tiny, and the cleanup problem largely disappears.

The envelope has to be shaped. A raw change record describes a row, not an event. The connector or a transform step must map the outbox columns into the standard envelope: `event_id`, `event_type`, `partition_key` as the message key, `payload` as the body. Most connectors support routing and field mapping for exactly this purpose. Keep the outbox columns aligned with the envelope so that the mapping is mechanical.

What it costs

CDC is not free, and the costs are operational.

You now run a connector, usually inside a connector framework, plus the broker infrastructure it depends on. Someone has to learn how it fails.

The database has to expose its log to the connector, which typically means enabling a particular logging level and creating a replication position that the database retains on the connector's behalf. If the connector stops and nobody notices, the database keeps log segments it cannot discard, and

disk fills. This is a known failure shape in the CDC family, and the details differ by database. I would treat "the connector is down" as a page, because the damage is not limited to the broker.

Schema changes to the outbox table, and to any table the connector watches, need to be planned. A column rename or type change can break the mapping.

Initial setup, snapshots and connector restarts after long outages also deserve rehearsal. The behavior depends on the product and the database, and I would not assume it from a diagram.

Note: CDC is a family of techniques, and products differ in how they read the log, how they store positions and what happens after a long outage. Treat this section as the general shape, and read your connector's and your database's documentation for the exact behavior of the versions you run.

When the log is gone

"The disk fills" is one half of a trade, and the other half is worse. The database either keeps log for the connector's position without limit, which fills the disk, or caps what it keeps, or expires logs on a schedule, which protects the disk. Many databases let you choose. If you cap it and the connector is down longer than the cap, its saved position points into log that no longer exists. Events in the gap are lost, or the connector has to start again from a snapshot.

What a re-snapshot does to an outbox depends on one earlier decision. If the table still holds rows, a snapshot re-emits all of them, so every retained row becomes a duplicate for every consumer. If, as is common with CDC, the rows were deleted right after insert, the snapshot finds nothing, and the events in the gap cannot be recovered from the table at all. That is a silent loss: no error, no duplicate, an order that never reached Inventory. The deletion that kept the table tiny is the same deletion that removes the evidence.

A failover of the database primary adds a third case. Whether the connector's position carries over to the new primary varies by database and by version, and I would find out by rehearsing a failover and not by reading a

diagram.

Three habits follow from this. Choose the retention cap on purpose, as a trade of disk against how long an outage you can survive, and write the number down. Alert on the headroom, meaning how much of the retention window the connector's lag has already used, before it runs out. And decide before the outage how you would recover events from the gap, for example from a reconciliation job (Chapter 13) that compares order state with what was published. Chapter 14 covers recovery after the damage is done.

Choosing between polling and CDC

Neither mechanism is better in the abstract. The table shows where they differ.

Concern	Polling relay	CDC connector
Moving parts	One small process you own	Connector, its framework, database log settings
Latency	Bounded below by poll interval	Usually low and steady
Ordering source	Query order plus your own care	Commit order from the log
Late-committing transactions	Safe with a state-based query	Handled by the log
Load on the database	Repeated queries, row locks	Log reads, retained log segments
Duplicates after crash	Possible, window is the unmarked batch	Possible, window is the unsaved position
Table growth	Needs retention work	Rows can be deleted immediately
Failure that hurts most	Silent lag, stuck rows	Stalled connector, retained log
Skills required	Ordinary application skills	Database log internals, connector operations

My default for a team starting out is a polling relay. It is a few dozen lines of code, the team already knows how to run and debug such a service, and its failure modes are visible in the table it queries. I would move to CDC when one of three things becomes true: polling latency is too high and

cannot be reduced without hammering the database, the outbox volume makes polling queries a measurable cost, or you already run a CDC platform and someone owns it.

I would not adopt CDC only because it sounds more serious. A team that has never operated a replication log and puts its order flow on one has swapped a problem it understood for one it does not. The reverse also holds: a team already running CDC for other purposes gains little from writing a polling relay on the side.

Operating the relay

Choosing the mechanism is the first half of the job. The second half is the work that keeps it healthy.

Retention and cleanup

A polling relay leaves published rows behind. Left alone, the table grows without bound, the unpublished-rows query gets slower, and the database spends effort on rows nobody reads. Cleanup has to be designed, not added later.

Two choices matter: how long to keep published rows, and how to delete them. Keeping them for a while has value. When a consumer reports a missing event, the outbox is the first place you look to learn whether it was ever published. Say, a few days is a reasonable starting point. How long is right depends on how quickly your team notices missing events, which is a question about people and alerting, not about storage.

```
function purgePublishedRows():
    while true:
        deleted = deleteRows(
            where: published_at is not null
              and published_at < now() - retentionPeriod,
            limit: 1000
        )
        // small batches keep locks short and replication lag low
        if deleted is 0:
            return
        sleep(shortPause)
```


Delete in small batches so that the job does not hold locks for long or flood replicas. If the table is large, partitioning by time and dropping whole partitions is cheaper than deleting row by row, where your database supports it. Whatever you choose, the purge condition must include `published_at is not null`. A cleanup that deletes by age alone will, on the day the relay falls behind, delete events that were never published. That is the failure from the opening scenario, and it is a data loss, not a duplicate.

Warning: Never purge outbox rows by age alone. A relay that is stuck for longer than your retention period will have its unpublished rows deleted, and those events are gone for good. Purge only rows that are marked published, and alert on unpublished rows that grow old.

With CDC, the cleanup is different but not absent. Rows can be deleted right after insert, so the table stays small. The thing that grows now is the retained log, and the thing to watch is the connector's lag against it. Deleting at once also means a re-snapshot cannot recover anything, as the earlier section on a vanished log explains, so weigh immediate deletion against a short retention of published rows.

Monitoring the relay

A relay that stops is invisible from the outside. Orders still succeed, the outbox accepts rows, and every other service quietly goes without events. Nobody gets an error. This is why the relay needs its own signals, and I would put three of them in place before launch.

1. **Age of the oldest unpublished row.** This is the single best number, because it measures what users experience: how long the slowest event has been waiting. Alert when it exceeds a threshold tied to the business, not to the technology. If a buyer's confirmation email should arrive within a minute, an oldest-unpublished age of five minutes is an incident.
2. **Count of unpublished rows.** A growing backlog says the relay is slower than the producer, even when the oldest row is still young.

3. Publish error rate and batch duration. These explain the first two. A relay that retries the same row every poll will show a stable error count and a growing age.

For a CDC connector, the same questions apply in different units: how far behind the head of the log the connector's position is, whether it is running, and how much log the database is retaining for it. Chapter 13 returns to these signals and to reconciliation, which catches what lag metrics miss.

Failover and double publishing

A relay must survive its own restarts, and that includes the case where two instances briefly run at once. Deploys, network partitions that make the old instance look dead, and orchestrators that restart a process before the old one has exited all produce this overlap.

For a single-active relay, use a lease: a row or a lock in the database that the active relay renews on a short interval and that expires if the holder stops. A standby takes over after expiry. The detail that trips people is that the old holder may not know its lease has expired. It could be paused by a long garbage collection or a stalled network, wake up, and publish a batch it no longer owns.

```

function relayWithLease():
    while running:
        // returns an increasing number
        token = acquireOrRenewLease(ttl: leaseTtl)
        if token is none:
            sleep(standbyInterval)
            continue
        // small: the batch must fit inside the lease
        rows = selectUnpublished(limit: 20)
        ackedIds = publishAkedWithin(rows, token)
        // marking rows the broker confirmed is true even if the lease
        lapsed meanwhile
        markPublished(ackedIds, now())

function publishAkedWithin(rows, token):
    ackedIds = []
    for each row in rows:
        if leaseRemaining(token) < safetyMargin:
            // stop before publishing, not after: a lapsed lease
            cannot be undone
            break
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
            buildEnvelope(row),
                                key: row.partition_key)
        if result is failure:
            break
        ackedIds.append(row.id)
    return ackedIds

```

The increasing token is a fencing token, and the idea only works where the protected resource checks it. The database can check it: a commit that carries a stale token can be made to fail, and Chapter 7 uses exactly that for consumers and APIs. The broker usually cannot, so the token does not stop a stale relay from publishing. Marking is a different matter. The stale holder marks only rows the broker confirmed, so its marks are true, and rolling them back would only create more duplicates. What a lease must limit is concurrent publishing, because two holders publishing together yield duplicates and can put an older row of a key after a newer one.

Four things limit it. Keep the lease TTL well above the worst-case duration of a batch, and keep batches small, because the lease is renewed once per loop and a slow broker alone can outlast it. Check the remaining lease before each publish, as the sketch does. Where a log-based broker supports producer fencing by epoch, use it, so that the broker itself rejects the old holder. And when the old holder wakes up, let it find out by asking the

database, in a transaction that locks the lease row, before it publishes anything else. A check that does not hold the lock is a check-then-act, and it fails the same way.

That is the honest limit. Failover can produce a double publish, and no lease scheme removes it. What a lease buys you is that the window is short and rare. What removes the harm is a consumer that treats the second copy as the same event, using the `event_id` that is stable across both publishes. The relay must therefore take `event_id` from the outbox row and never generate a new one when it publishes. A relay that mints a fresh identifier on every send turns a harmless duplicate into two apparently distinct events. Some brokers accept a producer-supplied deduplication id and discard repeats within a time window; where yours does, set it to `event_id`. It narrows the window for free and does not replace the consumer's inbox. Chapter 7 covers the consuming side.

Preserving order

The outbox preserves the order of events only as far as the relay does, and the relay can only preserve the order the table holds. Chapter 4 showed that `id` order is not commit order: a transaction can take an `id`, wait on a lock, and commit after a transaction that took a later `id`. The `id` is the order of insertion, not the order of commit.

Two things follow for a polling relay. If a lower `id` for a key commits after a higher one for the same key was already published, the two go out reversed, and the state-based query that rescued the late row is what publishes it late. If both become visible in one poll with inverted ids, they go out reversed in one batch. A relay that publishes "in `id` order" therefore does not deliver per-aggregate events in the order the aggregate changed, unless one of two conditions holds.

The first is that producers lock the aggregate row before inserting the outbox row, so same-key transactions serialize and their ids follow their commit order (Chapter 4). The second is that the order is carried in the data: the `aggregate_version` column, which consumers compare and which Chapter 9 uses. I would do both. The lock is cheap insurance for the relay, and the version is what survives everything after the relay. A CDC connector avoids the first problem because the log is in commit order, and still gives you no help across partitions, so the version stays useful.

Given that, the relay's own choices still matter. Each of these can break the order it has:

- Skipping a failed row and publishing later rows for the same key.
- Running parallel relays without partitioning by key.
- Retrying a failed publish out of band, so a later row arrives first.
- Using different message keys for events of one aggregate.

The safe default is head-of-line blocking per key: if a row for `order-81724` cannot be published, stop publishing later rows for `order-81724`, and let rows for other keys go on. This costs latency for the one blocked aggregate and protects its sequence. If the problem is a row that can never be published, such as a payload the serializer rejects, someone has to repair it, and the monitoring above is how they find out.

Be clear about what the earlier sketch does. With a `break` on the whole batch and an oldest-first query, one unpublishable row is returned first on every poll, and it stalls every key forever. Blocking per key needs two additions, shown as a refinement of `publishAcked` that replaces its `break`: the relay remembers which keys failed in this batch, and a row that fails repeatedly is parked, so that the query stops returning it and also stops returning later rows of its key.

```
function publishAcked(rows):
  ackedIds = []
  blockedKeys = emptySet()
  for each row in rows:
    if row.partition_key in blockedKeys:
      // an earlier row of this key failed: do not overtake it
      continue
    result = publishAndAwaitAck(topicFor(row.aggregate_type),
                                buildEnvelope(row),
                                key: row.partition_key)
    if result is failure:
      blockedKeys.add(row.partition_key)
      // attempts + 1; failed_at = now() after N attempts
      recordFailure(row.id)
      continue
    ackedIds.append(row.id)
  return ackedIds
```

The select then excludes any key that has a parked row, so later rows of that key wait and rows of other keys flow. Parking is the moment Chapter 4's `failed_at` column is needed, together with an `attempts` counter, so a

transient failure is retried a few times before the row is parked. A parked row is one whose `failed_at` is set. Alert on every parked row, because a parked row is a blocked aggregate and a person has to repair or discard it. The cost is two columns, a query that is harder to index, and a repair procedure someone must write before the first incident.

Order is also lost after the relay, in the consumer and the broker. A relay that does its part gives you a sequence worth preserving, not a guarantee that it survives the rest of the path. Chapter 9 follows the order from the broker to the handler.

Trade-offs and costs

Both mechanisms trade simplicity against operational reach, and both leave a duplicate window you cannot close.

A polling relay costs database load and latency in exchange for staying inside code your team already understands. Its weak points are the unpublished-rows query on a table that grows, lock behavior when several relays run, and silent lag when nobody watches the oldest row. If your system publishes at a modest rate and a second of delay is acceptable, the polling relay is hard to beat on cost of ownership.

CDC costs a new component and new skills in exchange for lower latency, commit-order reading and a table that stays small. Its weak points are a stalled connector that makes the database retain log, or a capped log that expires under a stalled connector and loses events, schema changes that break mappings, and recovery procedures that are rarely rehearsed. If you have no one who will own the connector, I would not adopt it, however attractive the architecture diagram looks.

When not to use an outbox relay at all: if an event can be lost without consequence, such as an informational notification where a missed message costs nothing, a direct publish after commit may be acceptable and cheaper. If your events must reflect state in a database that does not offer a local transaction across the state and the outbox, the relay cannot repair that, and you need a different design.

Two costs are common to both. Neither gives you exactly-once publishing: the duplicate after a crash between publish and mark, or between publish and position save, is part of the design. And neither helps if the relay and the producer disagree on the envelope, so keep the outbox columns and the envelope fields in step.

Common mistakes

- **Using a cursor on id instead of a state-based query.** It is tempting because it avoids scanning for unpublished rows. Late-committing transactions then fall behind the cursor and are never published. Query by `published_at is null`, or use a log-based mechanism that already orders by commit.
- **Marking rows before the broker confirms.** Marking first looks like it removes duplicates. It converts a crash into a lost event. Publish, await the acknowledgment, then mark.
- **Generating a new event identifier in the relay.** It feels harmless, since the relay is the one sending. The consumer's inbox can then no longer recognize a republished event as the same one. Copy `event_id` from the row.
- **Purging by age alone.** It keeps the table small and passes every test where the relay is healthy. It deletes unpublished events when the relay is not. Purge only published rows and alert on old unpublished ones.
- **Skipping a failed row to keep throughput up.** Later rows for the same aggregate then overtake it. Block per key instead, park the row that keeps failing, and fix the cause.
- **Believing id order is commit order.** It holds in every single-writer test. Under concurrent writers a lower `id` can commit later, and the relay publishes the pair reversed. Lock the aggregate row before the outbox insert and carry `aggregate_version`.
- **Trusting the lease to fence the broker.** A stale relay that wakes up after its lease lapsed still publishes, because the broker does not check the token. Keep batches inside the TTL, check the lease before each publish and let consumers absorb the rest.

- **Running CDC with a retention cap nobody chose.** The default feels like a detail until the connector is down for a weekend and the position is gone. Pick the cap, alert on headroom, and know what a re-snapshot does to your outbox.
- **Running several relays without partitioning by key.** Row locks prevent double work and say nothing about order. Assign key buckets to single workers.
- **Not monitoring the relay.** The outbox makes the producer reliable, which makes the relay's failure the only one left, and a relay that has stopped looks exactly like a quiet system. Alert on the age of the oldest unpublished row.
- **Treating a connector outage as a broker problem.** In CDC, a stalled connector makes the database retain its log. Page for it, and watch disk, not only lag.

Key takeaways

- A relay does three separate things, find, publish and mark, and the broker and database share no transaction, so a crash between publish and mark always leaves a possible duplicate.
- Publish first and mark second: the other order turns a crash into a lost event, which is the failure the outbox exists to remove.
- A polling relay should select unpublished rows by state, not by an increasing id cursor, because transactions commit in a different order than they allocate ids, and even then per-aggregate order holds only if producers lock the aggregate first and carry an `aggregate_version`.
- To run several polling relays, claim rows with skip-locked-style locking and partition by `partition_key`, so that rows for one aggregate are always handled in order by one worker.
- Change data capture reads committed changes from the database log in commit order and removes polling and the late-commit problem, at the price of a connector, a log-retention trade between a full disk and lost events, and skills your team has to own.
- Start with a polling relay unless latency, load or an existing CDC platform says otherwise, and treat the choice as a trade of simplicity against operational reach.

- Monitor the age of the oldest unpublished row, purge only published rows, park rows that never publish, copy `event_id` from the outbox row, and expect duplicates after failover because a lease limits concurrent publishing and only an idempotent consumer makes the rest harmless.

The relay now delivers every event at least once, in an order worth keeping, and sometimes twice. The next problem sits on the other side of the broker: what a consumer should do when its work fails, which Chapter 6 takes up with retries.

PART III

Consuming Reliably

Chapter 6 — Retries Done Right

Imagine a Tuesday morning at Parcelly, at the start of a seller promotion. Orders sends a `CapturePayment` command, and Payments passes it to the external payment provider. The provider is struggling and answers slowly. Payments' calls begin to time out, so Payments fails the message and the broker redelivers it. Every Payments instance does the same thing, at the same moment, with the same fixed one-second wait.

Within a minute the provider is receiving not the promotion's normal load but the normal load plus every failed call coming back, over and over, in synchronized waves. The provider's slowness, which might have lasted a few seconds, now lasts as long as the retries keep arriving. Meanwhile Payments' consumers are busy retrying messages that will not succeed, so healthy messages queue behind them. Somewhere in that pile, a few captures that had in fact succeeded at the provider are being sent again.

Nobody wrote a bad line of code here. Each instance did the sensible thing: it failed, it waited, it tried again. The damage came from a thousand sensible decisions that nobody coordinated.

This chapter is about making a retry a decision instead of a reflex. The question it answers is this: when a call fails, how do you decide whether to try again, when, how often, and at which layer, so that retrying helps the system recover instead of preventing it?

The problem, and why it is hard

A retry is a promise to try again, not a promise to succeed. In this chapter an **attempt** is any try, so attempts equal one plus the number of retries. The obvious approach, "wrap the call in a loop and retry a few times", fails for four separate reasons, and they compound.

First, **not every failure is worth retrying**. A request rejected because the card was declined will be rejected on the tenth attempt too. Retrying it wastes capacity and delays the moment someone notices the real problem. The retry loop has to know what kind of failure it is looking at, and most loops do not.

Second, **retries add load at the worst moment**. A dependency fails when it is overloaded, degraded or restarting. Retries are extra requests aimed at exactly that dependency. If every caller retries three times, which makes four attempts in all, a struggling service sees up to four times its normal traffic when it can least afford it.

Third, **retries synchronize**. Callers that fail together retry together. A fixed delay turns a single failure into a periodic spike, because every caller wakes up at the same instant. This is the thundering herd, and it is the reason the delay needs randomness and not only length.

Fourth, **a retry repeats an action whose first outcome may be unknown**. Chapter 3 showed that a timeout does not say whether the request was processed. Retrying after a timeout therefore risks doing the work twice. Without idempotency (Chapter 7), a retry is a duplicate waiting to happen.

Everything else in this chapter is a way of answering one of those four problems. Classification answers the first. Backoff with jitter and a retry budget answer the second and third. Idempotency, which Chapter 7 builds, answers the fourth, and the section on its absence below explains why it cannot be skipped.

Classifying failures

Before deciding how to retry, decide whether to. Chapter 1 gave three questions to ask of any seam: what can fail, how you would observe it, and which guarantee survives it. The answers together are a failure's semantics, and a retry decision is those semantics put to work. I find it useful to sort every failure into one of three classes, based on what it says about whether trying again can possibly help, and then to add a fourth signal that says nothing about the message at all.

Transient failures

A **transient failure** is one whose cause is expected to go away by itself. A connection reset, a timeout, a `503 Service Unavailable`, a deadlock victim in the database, a throttling response, a leader election in the broker. The request was not wrong. The world was briefly in the way.

Retrying is the right response, with a delay, because the delay is what lets the cause clear.

Permanent failures

A **permanent failure** is one that will recur on every attempt, because the cause lives in the request or in the data. A payload that fails validation, a reference to a record that stays missing long after it should have arrived (a missing record may be transient in an event-driven system, because the event that creates it can arrive late; Chapter 9 covers that case), a `400 Bad Request`, a card declined for insufficient funds, an event whose schema version the consumer does not understand.

Retrying is wasted effort and sometimes harmful. The correct response is to stop, record why, and send the message somewhere a human or a repair job can deal with it. Chapter 8 covers where permanent failures go. Note the card decline in particular: for Payments it is not an error at all. It is an expected business outcome that becomes a `PaymentFailed` event, and no retry is involved.

Unknown failures

An **unknown failure** is one whose class you cannot tell. A timeout is the classic case, and so is an unexpected exception with a message nobody anticipated. The timeout might have been transient. The exception might be a bug in the consumer that will fail on every attempt.

Unknown is a real class, not a gap in the taxonomy, and the way you treat it is a policy decision. My default is to treat unknown failures as transient, but with a stricter limit than the known transient ones, because the cost of wrongly retrying an unknown failure is a few wasted attempts, while the cost of wrongly dropping it is lost work. When every layer's attempts run out, an unknown failure is handled as a permanent one: it goes to the dead letters with its full history.

When the outcome of an attempt is unknown, ask before you act again. Before a retry, and certainly before a compensation, query the provider for the operation's status by its idempotency key. If the capture happened, the right follow-up is a refund, not a void. If it did not, a retry or a void is safe. Guessing in either direction is how a buyer ends up charged for an order Parcelly believes it cancelled.

Deferred: the dependency is known to be down

The fourth signal is not a verdict about the message. When a circuit breaker (described below) is open, the failure of a call says nothing about the request: the call never reached the dependency. I call this class **deferred**. A deferred message does not consume an attempt, does not count toward any limit, and is never dead-lettered because of it. The consumer stops pulling, or pauses the partition, and the message waits where it is until the dependency is worth calling again.

The distinction matters because the alternative is arithmetic. A breaker that stays open for 30 seconds, in front of a handler that allows three in-handler attempts a second apart, lets a healthy message burn its whole budget in roughly the first three seconds and land in the DLQ for a problem it never had.

Making the classification explicit

The classification has to live somewhere that can be read and reviewed. A table that maps a failure signal to a class and a handling is one of the most valuable small artifacts in a consumer. For Payments' calls to the provider, an illustrative version looks like this:

Signal	Class	Handling
Connection reset, DNS failure	Transient	Retry with backoff
Timeout on a capture call	Unknown	Retry with the same idempotency key, strict limit
429, throttling response	Transient	Retry, honor the provider's wait hint if given
5xx from the provider	Transient	Retry with backoff

Signal	Class	Handling
400, validation failure	Permanent	Do not retry, dead-letter
Card declined	Permanent, business outcome	Emit <code>PaymentFailed</code> , do not retry
Unrecognized error	Unknown	Retry with strict limit, then dead-letter
Circuit breaker open	Deferred	No attempt consumed; pause consumption, resume when the breaker allows

Two cautions. First, status codes are hints, not guarantees; a provider that returns 500 for a malformed request has misclassified it for you, and you will have to learn that from experience with that provider. Second, the table should default to unknown, never to permanent or transient, for any signal it has not seen. A new error nobody classified is not evidence of anything.

The classifier itself is small:

```
// business outcomes such as a card decline are handled by the caller
// before this function is reached; they are not errors
function classifyFailure(error):
  if error is ValidationError:
    return "permanent"
  if error is CircuitOpen:
    // the call never reached the dependency: no verdict on the
    message
    return "deferred"
  if error is ConnectionError or error is Throttled:
    return "transient"
  if error is Timeout:
    // the request may or may not have been processed
    return "unknown"
  return "unknown"
```

Exponential backoff with jitter

Once you decide to retry, the next question is when. Retrying immediately gives a transient cause no time to clear. Retrying after a fixed delay gives it time, but it keeps every caller in step.

Exponential backoff makes the delay grow with each attempt: say, one second, then two, then four, up to a ceiling. Early retries are quick, which suits failures that clear in an instant. Later retries are slow, which relieves a dependency that is genuinely down. The ceiling keeps the delay from growing beyond what the business can tolerate.

Jitter adds randomness to each delay. Without it, callers that failed at the same moment compute the same delays and retry in lockstep, and the herd survives the backoff intact. With it, the retries spread across an interval, and the dependency sees a smear of requests instead of a wall.

There are several jitter schemes. Two are worth knowing by name. In **full jitter**, the delay is a random value between zero and the exponential ceiling for that attempt. In **equal jitter**, half the delay is fixed and half is random, so the minimum wait is never near zero. Full jitter spreads load best, while equal jitter guarantees a floor on the wait. I would start with full jitter, and move to equal jitter only if the dependency suffers from very quick retries.


```

function computeDelay(attempt, baseDelay, maxDelay):
    // the ceiling doubles with each attempt, capped at maxDelay
    ceiling = min(maxDelay, baseDelay * 2 ^ (attempt - 1))
    // full jitter: anywhere between zero and the ceiling
    return randomBetween(0, ceiling)

function callWithRetry(operation, policy):
    attempt = 0
    while true:
        try:
            return operation()
        on error as error:
            failureClass = classifyFailure(error)
            if failureClass == "permanent":
                raise error
            if failureClass == "deferred":
                // no attempt is counted; the consumer pauses, not
retries
                raise error
            attempt = attempt + 1
            if attempt >= attemptLimit(failureClass, policy):
                raise RetriesExhausted
            sleep(computeDelay(attempt, policy.baseDelay,
policy.maxDelay))

function attemptLimit(failureClass, policy):
    // unknown failures get the stricter limit promised above,
    // counted over all attempts so far, whatever their class
    if failureClass == "unknown":
        return policy.maxAttemptsWhenUnknown
    return policy.maxAttempts

```

This loop is correct under the failures it handles, but it is incomplete in two ways that the next sections address. It does not know how much time the caller has left, and it does not know how much retrying the whole system is already doing.

If the provider sends a hint about when to try again, such as a wait value in a throttling response, honor it. The provider knows its own recovery better than your formula does. Treat the hint as a floor under your computed delay, not a replacement for it, and keep the jitter so that callers with the same hint do not return together. A floor is still not a license to sleep past the deadline: if the hint is longer than the time that remains, stop retrying and let the next layer decide.

Deadlines and retry budgets

Backoff with jitter decides the spacing of retries. Two further limits decide how many there may be.

Deadlines

A **deadline** is the moment after which the result is no longer useful. A request-driven flow has a natural one: the buyer is staring at a checkout page, and 10 seconds from now they will leave. A background consumer has a softer one: a shipment label that arrives after the carrier's cutoff is as late as one that never arrives.

Retry logic that counts attempts but ignores time can run far past usefulness. Three attempts with a 30-second timeout each and growing delays between them is minutes of waiting. I prefer to give every retried operation a total time budget, and to stop retrying when the next delay would cross it, whatever the attempt count says.

```
function callWithDeadline(operation, policy, deadline):
    attempt = 0
    while true:
        remaining = deadline - now()
        if remaining <= 0:
            raise DeadlineExceeded
        try:
            // never let one attempt outlive the whole budget
            return operation(timeout: min(policy.attemptTimeout,
remaining))
        on error as error:
            failureClass = classifyFailure(error)
            if failureClass == "permanent" or failureClass ==
"deferred":
                raise error
            attempt = attempt + 1
            delay = computeDelay(attempt, policy.baseDelay,
policy.maxDelay)
            // the next attempt needs time to run, not only time to
start
            needed = delay + policy.minimumUsefulAttempt
            if attempt >= attemptLimit(failureClass, policy) or needed
>= deadline - now():
                raise RetriesExhausted
            sleep(delay)
```

The deadline should also travel. If Orders gives a flow 10 seconds and calls Payments, Payments should pass on what is left, not start a fresh 10 seconds of its own. Otherwise each layer retries inside the previous layer's patience, and the sum quietly exceeds what the user will wait.

Retry budgets

A per-call attempt limit does not protect a dependency when many calls are failing at once. Each call stays within its limit of three retries, and the dependency still receives four times its normal traffic.

A **retry budget** limits retries as a share of overall traffic, not per call. A common form is a token bucket: each successful call adds a fraction of a token, each retry spends one, and when the bucket is empty, calls fail fast without retrying. In steady state, retries are a small fraction of requests. In a widespread outage, the budget runs out within moments and the system stops amplifying its own failure.

The numbers are yours to choose and depend on how much extra load the dependency can absorb. I will not offer a figure, because any figure I gave would be a guess about your dependency. What matters is that the budget exists, that it is shared across the callers in a process, and that exhausting it is visible on a dashboard. A ratio-based budget kept per process is usually enough, because every process capping its own ratio already caps the total. A budget shared across instances is an option, not a strict improvement: it needs a distributed counter, and that counter is a new dependency on the failure path, with a failure mode of its own.

The price is that some calls that would have succeeded on retry will fail instead. That is the point of a budget: it trades individual success for system survival, and it should do so on purpose.

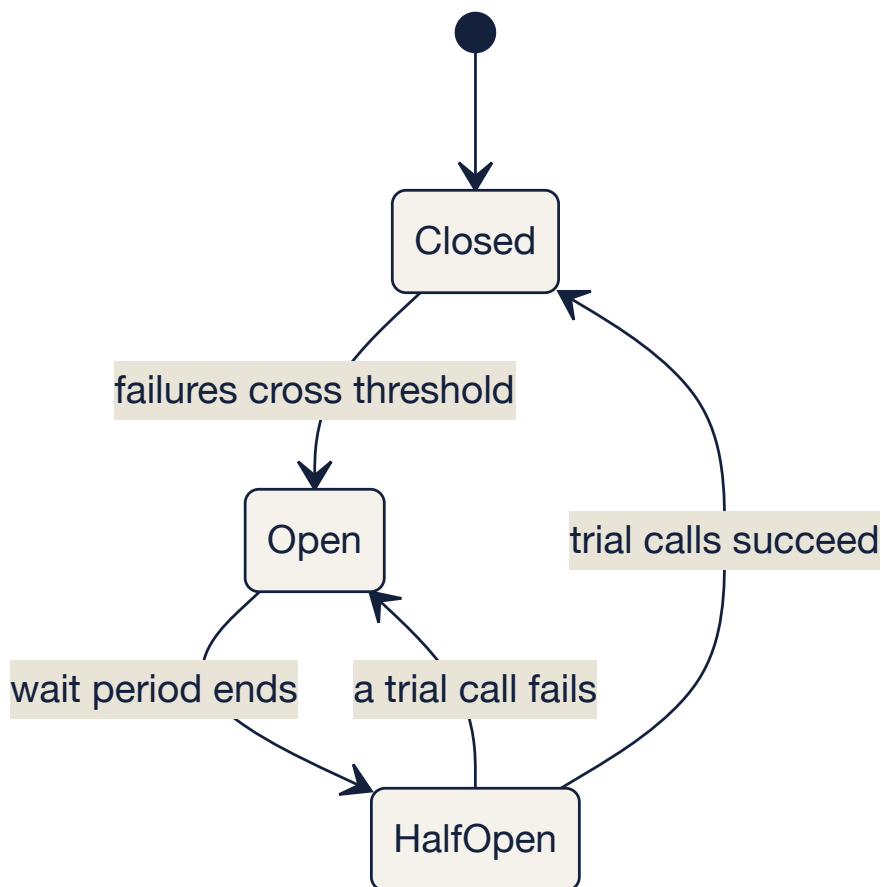
Retry storms and the circuit breaker

A **retry storm** is what happens when retrying becomes the dominant load on a failing dependency. Parcelly's opening scenario was one. It needs three ingredients: a dependency that is slow or failing, callers that retry, and no mechanism that tells the callers to stop. Layers make it worse. If

Orders makes three attempts at Payments, and Payments makes three attempts at the provider, one failed capture can become nine calls to the provider. With another layer of three attempts it becomes 27.

Two defenses address the storm directly. The retry budget above limits how much retrying the system will do. The circuit breaker limits whether it should call at all.

A **circuit breaker** watches the outcomes of calls to a dependency and has three states. It is closed in normal operation, with calls passing through. When failures cross a threshold, it opens, and calls fail immediately without touching the dependency, giving the dependency room to recover. After a wait, it moves to half-open and lets a few trial calls through. If they succeed, it closes. If not, it opens again.



The breaker and the retry policy are complementary and answer different questions. Backoff decides how long to wait before the next attempt on one call. The breaker decides, from the pattern across calls, whether attempts are worth making at all. A breaker that is open turns the retry loop's next attempt into an instant failure. That failure is not transient and it is not unknown: it is the **deferred** signal from the classification above. The retry loop must not count it, back off on it, or spin on it. It hands control back to the consumer, which decides what to stop doing.

A consumer has an extra choice that a request-driven caller lacks. When the breaker is open, a consumer should stop pulling messages from the broker, or pause the partition, instead of consuming messages and failing each of them. Messages that stay in the broker are safe for as long as its retention allows, and they cost storage and show up as consumer lag. Messages that are consumed, retried and failed burn their attempt counts, and might be dead-lettered for a problem that was never theirs. Pausing is the consumer-side form of the breaker, and I would build it before most other things in this chapter.

Pausing raises two follow-up questions. First, with no traffic flowing, nobody tests the half-open state. Resume with a concurrency of one, or send a synthetic probe call, so that the trial calls are made by something that knows they are trials. Second, on a log-based broker a paused consumer must keep its poll loop alive, pausing the partitions but not the polling, or the group will decide the consumer is dead and reassign its partitions (the liveness window described below).

The cost of a breaker is that it can fail wrongly. Thresholds that are too sensitive open on a brief blip and turn a transient failure into an outage. Thresholds that are too lenient open too late. A breaker also adds state that someone must understand when a flow stops working at three in the morning: the first question is whether the dependency is down or only the breaker is open.

Retries under overload: backpressure

A breaker reacts to a dependency that has failed. **Backpressure** is the discipline for the stage before that, when a dependency is slow, saturated or recovering, and the system must decide how much work to hand it.

Chapter 3 places it inside the failure model. This section is the mechanism.

Return to the opening scenario. The provider's slowness did not come from a bug. It came from more work than it could absorb, and every Payments instance kept feeding it. A consumer with no limit on in-flight work behaves exactly that way: it keeps pulling, keeps starting calls, and keeps adding to the pile that is already too heavy. The limits below are the ones I would put in place, roughly in this order.

- **Bound the in-flight work.** Cap the number of messages a consumer holds at once. On a queue-based broker that is the prefetch or in-flight count. On a log-based broker it is the number of records per poll plus the number of handlers running. With no cap, the consumer's memory and the broker's redelivery clock become the limit, and both are worse limits than one you picked.
- **Cut concurrency when the dependency slows down.** Concurrency that is right at 50 milliseconds per call is wrong at five seconds per call, because the same number of slots now holds many times more waiting work. Lower the number of concurrent calls when latency rises, and raise it again, slowly, when latency recovers.
- **Pause instead of failing.** When the dependency is saturated, stop pulling, as with an open breaker. A paused consumer shows up as lag, which is visible and safe. A consumer that pulls and fails burns attempts.
- **Serve live traffic before retries.** Retries and redrive compete with first attempts for the same dependency. When capacity is short, first attempts should win, because a retry is work the system already knows is struggling. Run retry-topic and redrive consumers at a lower concurrency, or let them wait while the main flow has backlog.
- **Shed at the ingress.** Somewhere upstream, a limit on accepted work has to exist, or the queue grows without bound while the dependency recovers. Reject or defer new work, with a clear signal to the caller, rather than queue it indefinitely behind a dependency that cannot serve it.

These limits and the retry budget are two views of one idea. The budget caps how much extra work retrying adds, and backpressure caps how much work, extra or not, is in flight. Neither is enough alone: a budget

with no in-flight cap lets a slow dependency fill every slot with first attempts, and an in-flight cap with no budget lets the slots fill with retries.

The price is lag and rejection. A system that sheds load tells some callers no, on purpose, and a system that pauses delivers some messages late. Both are a trade of throughput for survival, and the person who owns the answer to "which work waits" is the business owner of the flow, not the engineer tuning the numbers. Chapter 8 applies the same rate-limit idea to redrive, and Chapter 9 applies the in-flight bound to keyed lanes.

Where retries live

A message can be retried at several layers, and the layers behave differently. Mixing them without knowing is how a message ends up retried far more often than anyone intended, or not at all.

Application-level retry

The consumer retries inside the handler: the code above, wrapped around a call to the provider. The message stays unacknowledged while the retries run. This layer is cheap, quick and good for failures that clear in milliseconds or seconds. Its limits are that it holds the consumer's capacity while it waits, that a consumer crash loses the retry state (the broker will redeliver, and the attempt count starts over, which is why a message that crashes the process needs the detection Chapter 8 describes), and that long waits risk exceeding the broker's own timeouts.

On a queue-based broker the unacknowledged message is hidden for the **visibility timeout**, after which it is redelivered. If the in-handler retries outlast that timeout, the broker redelivers while the first copy is still being retried, and now two copies are running. On a log-based broker, a consumer that spends too long between polls may be considered dead by its consumer group and have its partitions reassigned, with the same result. The rule I would follow: the longest total time of in-handler retries must sit comfortably inside the broker's timeout, and anything longer belongs in a different layer.

Broker redelivery

The consumer fails the message, and the broker delivers it again. The two broker families differ here, and the difference matters.

On a queue-based broker, redelivery is built in. The consumer nacks the message, or lets the visibility timeout expire, and the message returns to the queue, immediately or after a configured delay, where any competing consumer may receive it.

A log-based broker does not redeliver on its own. The consumer's read position moves forward as it polls, and the committed offset only records where a restarted consumer would resume (Chapter 2 covers this). Failing to process a record and not committing its offset therefore does nothing by itself: the consumer reads the next record, and if it then commits, that commit covers the failed record as well, which is silently skipped. The failed record is read again only after a restart, a rebalance, or an explicit step by the consumer.

On a log, then, the consumer has to choose one of three actions. It can seek back to the failed offset, or pause the partition, and retry in place, which preserves order and stalls everything behind it. It can publish the record to a retry topic and move on. Or it can write it to a dead-letter topic and move on. Each is a deliberate decision in the consumer's code, not something the broker does for you.

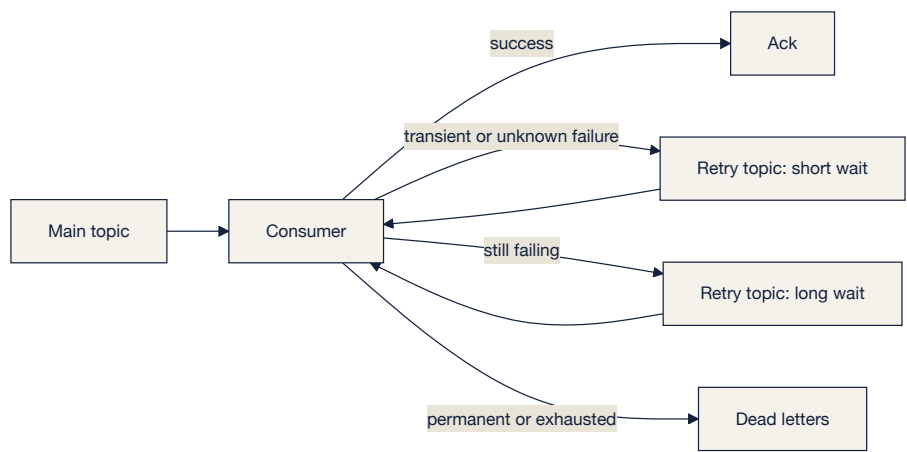
Broker redelivery is durable: it survives a consumer crash. It is also blunt. Many brokers redeliver at once or on a fixed schedule, with no backoff, and the attempt count may be available only as a delivery counter that the consumer has to read and act on. Treat it as the safety net beneath the other layers, not as the main retry policy.

Retry topics and delay queues

When a failure needs a wait longer than in-handler retry can hold, a common structure is to move the message out of the main flow and let it come back later. A consumer that fails a message publishes it, with an incremented attempt count and a not-before time, to a **retry topic** (or retry queue), and acknowledges the original. A separate consumer, or the same one on a different topic, picks it up when the time arrives. Brokers that

support delayed delivery, or per-message delays, give this directly. Where they do not, a common arrangement is a ladder of retry topics, each with a fixed wait, one for each step of the backoff.

The waits are fixed per topic for a reason. On a log, a consumer cannot skip a record that is not yet due: it must wait at the head until the record is ready. If every record in a topic has the same wait, the topic is already in due-time order, and waiting at the head costs nothing extra. A topic that mixed short and long delays would hold the short ones behind the long ones. Per-message not-before times work only on brokers with native delayed delivery.



This layer frees the main flow, so a failing message no longer holds up the messages behind it. It survives restarts, because the state lives in the broker. It also gives you a place to look: the retry topic's depth is a direct measure of how much work is waiting on a recovering dependency.

It costs more machinery. There are more topics to create, name, monitor and secure. The republish is itself a produce operation that can fail, which brings back the dual write from Chapter 1, so the order matters: publish to the retry topic first, then acknowledge the original. If the process dies between the two, the message exists twice. That is a duplicate, which the consumer must tolerate anyway.

Three rules keep the ladder from damaging what it is meant to protect:

- **Own the topic per consumer.** A retry topic belongs to one consumer group or subscription. If another group can read it, that group repro-

cesses a message it never failed.

- **Preserve identity.** The republished message keeps the original key, `event_id` and business headers. A new key breaks ordering, and a new `event_id` breaks the inbox.
- **Size retention to the wait.** Retention on a retry topic must exceed the longest wait plus the lag you expect on it, or the message expires while it waits.

The most important cost is **ordering**, which the next section covers.

Choosing the layers

In practice, a consumer uses all three. A short application-level retry absorbs the blips, such as a connection reset or a deadlock, that clear within a second or two. A retry topic or delay queue covers failures that need minutes. The broker's redelivery covers crashes. A dead-letter queue (DLQ) ends the chain, receiving what has exhausted its attempts or can never succeed; Chapter 8 covers its design.

The layers should be designed together, with one total attempt budget, not three independent ones. If each layer allows three attempts, a message may be tried 27 times, and nobody planned that. The policy at the end of this chapter makes the multiplication explicit and caps it.

Retries and ordering

Retrying a message changes the order in which messages are processed. This follows from the layering above, and it goes unnoticed until it causes damage.

Consider Parcelly's Inventory consumer, which receives `OrderPlaced` and then `OrderCancelled` for the same order, in that order. Suppose the processing of `OrderPlaced` fails transiently and the consumer moves it to a retry topic, then goes on to process `OrderCancelled`. The cancellation finds no reservation to release, because as far as Inventory knows, none exists yet. Then the retried `OrderPlaced` arrives and creates a reservation that nothing will ever release. The final state is wrong, and every individual step looked reasonable.

There are two honest options, and each has a price.

- **Block and retry in place.** On a log-based broker, the consumer seeks back to the failing offset, or pauses the partition, and retries it, with backoff, before touching anything behind it. Order is preserved for that partition. The price is that the whole partition waits behind one message, so a single failing message can stall every aggregate on that partition. This is the right choice when order matters more than throughput, and it is why classifying a permanent failure quickly matters so much: a poison message at the head of a partition blocks everything behind it.
- **Move aside and accept reordering.** The failing message goes to a retry topic and the consumer continues. Throughput is preserved. The price is that the consumer must tolerate out-of-order arrival, through techniques such as version checks, or by also parking every later message for the same aggregate until the earlier one succeeds. Chapter 9 covers these techniques.

I would choose per stream. For an event stream where messages for one aggregate must apply in order, I would retry in place and invest in fast classification and a dead-letter exit. For a stream of independent facts, where order across messages does not matter, moving aside is cheap and keeps the flow moving.

On queue-based brokers without a grouping feature, ordering across competing consumers is weak to begin with, so retries make an existing property slightly worse rather than breaking a guarantee. Even so, a consumer that assumes order should not rely on a queue to supply it. If you use a grouping feature (FIFO queues, message groups, sessions), the trade-off is the same as on a log: a failing message blocks its group until it succeeds or is dead-lettered, the way an in-place retry blocks a partition.

Retries without idempotency

Every retry is a repeated attempt at work whose earlier outcome may be unknown. When the earlier attempt failed before doing anything, the retry is harmless. When it failed after doing the work, the retry does the work twice.

This is not a rare edge. Consider the opening scenario. Payments calls the provider to capture 4,990 minor units of EUR. The provider captures the money and sends its response, and the response is lost. Payments sees a timeout. The timeout is classified as unknown, and the policy says to retry. The second capture request goes out. Whether the buyer is charged once or twice now depends entirely on whether anything makes the second request recognizable as a repeat.

That something is an **idempotency key**: a stable identifier, sent with the request, that lets the receiver recognize it has seen this work before. For the provider call, it is the same key on every attempt for the same capture, which a provider that honors such keys will treat as one capture. For consuming events, it is normally the `event_id`, checked against the inbox in the same transaction as the work. Chapter 7 builds both.

Warning: A retry policy is safe only on top of an idempotent operation. Before you enable retries on any call that has an effect, such as charging a card, sending an email or reserving stock, establish what makes the second attempt harmless. If nothing does, the correct number of automatic retries is zero, and the failure should go to a person. Since an at-least-once broker will redeliver after a crash or timeout whatever the handler wants, "zero" has to be enforced by the handler: it must detect that it has seen this message before, or acknowledge before the effect and accept at-most-once, and then route the failure to a person.

The dependency runs both ways. Idempotency is what makes retries safe, and retries are what make idempotency necessary. A system that retries aggressively without idempotent consumers will produce duplicate effects at a rate proportional to its failure rate, meaning at the moments when the system is already under stress and nobody has time to clean up.

There is also a subtler case. An operation can be idempotent for the same key while still unsafe to retry, because the key changes. If a handler builds a fresh identifier on every attempt, such as a new UUID on each call, the provider sees distinct requests every time. The key must be derived once, from stable inputs such as the `event_id` or the order, and reused on every attempt.

A retry policy for Parcelly

To pull the pieces together, here is what I would write down for Payments' consumer of `CapturePayment`. The numbers are illustrations, not recommendations, and your dependency will need its own.

```
retryPolicy for CapturePayment:
  // illustrative values
  attemptTimeout: 5 seconds
  inHandler:
    maxAttempts: 3
    maxAttemptsWhenUnknown: 2
    baseDelay: 200 milliseconds
    maxDelay: 2 seconds
    jitter: full
  retryTopics:
    steps: [1 minute, 5 minutes, 30 minutes]
    appliesTo: transient and unknown failures; the deadline
caps both
  // after this, a capture is stale
  totalDeadline: 2 hours
  // all layers and redeliveries together
  maxTotalAttempts: 8
  retryBudget: per process, visible on a dashboard
  circuitBreaker: pause consumption when open; deferred calls cost
no attempt
  onExhausted: dead-letter with the full attempt history
  onStale: ask the provider for the capture's status, then
           void if not captured, refund if captured
  idempotencyKey: derived once from command_id, reused on each
attempt
```

Read it as a set of decisions, each with a reason. The in-handler layer absorbs short blips. The retry topics cover a provider that is down for minutes. The deadline says when the work stops being worth doing, and the owner of that decision is the business, not the engineer. A stale capture may need to become an `OrderCancelled`, but the compensation depends on what actually happened at the provider. If the last attempt timed out, the capture may have succeeded, and then the right follow-up is a `PaymentRefunded`, not a `PaymentVoided`. That is why the policy asks for the status first. The key, derived once, is what keeps the whole structure from charging a buyer twice.

Two lines in the policy deserve a second look. `maxTotalAttempts` exists because the layers multiply: three in-handler attempts, times one first pass plus three retry-topic steps, is up to 12 calls to the provider, before count-

ing any redelivery after a crash. A single total of 8 (illustrative) is the number someone actually owns. And the whole retry horizon, two hours here, must be shorter than the time the provider remembers idempotency keys. Providers keep keys for a limited window, and a retry that arrives after the window is a new request carrying an old key.

When Chapter 8's redrive puts a dead capture back, the attempt counter starts again, but a separate redrive counter stays, so the history is not lost and a message that keeps coming back is visible as one.

Notice what is missing: a single "retry count" setting. In a system with layers, that one number hides at least four decisions.

Trade-offs and costs

Retries are one of the cheapest ways to improve a system's apparent reliability, and one of the easiest to overuse. Their costs are specific.

Latency. Every retry adds wait time. Backoff and jitter spread load, but they also stretch the time to success. A flow that retries for minutes may complete correctly and still be useless to the buyer who left after 10 seconds. Where a human is waiting, I would favor a quick failure and a clear message over a long silent retry.

Amplification. Retries multiply load, and layered retries multiply it again. A policy that is reasonable in each service may be dangerous in the chain. Someone has to own the end-to-end attempt budget, because no single service can see it.

Capacity. A consumer that retries in place occupies its slot while it waits. Under a long outage, all consumers can end up waiting, and the system stops making progress even on work that would succeed. Pausing consumption through a breaker, bounding in-flight work with backpressure, and moving long waits to retry topics are the usual defenses.

Hidden failures. A retry that eventually succeeds looks like a success from the outside. If you do not count retries and attempts, a dependency can degrade for weeks behind a policy that quietly absorbs it. Chapter 13 covers what to measure: attempt counts, retry rate and the age of the oldest retried message belong on a dashboard.

Complexity. Retry topics, delay queues and budgets are infrastructure that someone must build, name, monitor and explain. For a small system with one consumer and one dependency, an in-handler retry with backoff and a dead-letter exit may be all you need, and a ladder of retry topics may be more machinery than the problem deserves.

When not to retry. Do not retry permanent failures. Do not retry effectful operations that are not idempotent. Do not retry past the deadline. Do not retry when a circuit breaker says the dependency is down, and do not charge that message an attempt for it. Do not retry inside a layer that already sits inside another layer's retry, unless you have worked out the multiplication. And do not retry as a way of avoiding a decision: a failure that keeps coming back is information, and the right response is often to stop and look.

Common mistakes

Retrying everything the same way

It is tempting because one loop is the shortest code. It treats a card decline like a connection reset and wastes capacity on failures that cannot succeed. Classify first, and give each class its own handling.

Fixed delays without jitter

A constant wait feels predictable and simple. It synchronizes every caller that failed together into a repeating spike. Use exponential backoff with jitter, and in my view full jitter as the starting point.

Counting attempts but not time

An attempt limit looks like a safety bound. With long timeouts and growing delays it can still run for minutes past usefulness. Set a deadline for the whole operation and stop when the next delay would cross it.

Retrying at every layer

Each layer's owner adds a sensible retry, and the multiplication is invisible to all of them. Decide one total attempt budget for the chain, and put retries where they cost least.

Retrying while the breaker is open

It is tempting to keep the loop going, because the loop was working. A consumer that pulls messages and fails them against a dead dependency consumes their attempts and may dead-letter them wrongly. Treat the open breaker as a deferred signal that costs no attempt, pause consumption, and let the messages wait in the broker.

Letting a consumer pull without a limit

An unbounded consumer is the default, and it feels like maximum throughput. Against a slow dependency it fills every slot with waiting work and feeds the retry storm. Bound in-flight messages, lower concurrency when latency rises, and let first attempts win over retries.

Ignoring the broker's timeouts

In-handler retries feel local to the handler. If they outlast the visibility timeout or the group's liveness window, the broker redelivers or reassigns, and two copies of the work run at once. Keep total in-handler time well inside the broker's limits.

Retrying past a message that must stay in order

Moving a failing message to a retry topic is a quick fix for a stalled partition. It reorders that aggregate's events. Decide the ordering requirement first, then choose between blocking and moving aside.

Retrying non-idempotent effects

The retry loop takes an afternoon to add, and idempotency gets postponed. The first timeout after a successful call produces a duplicate effect. Establish what makes the second attempt harmless before the first retry is enabled.

Generating a new key on every attempt

It looks like diligence, since each request is uniquely identified. It defeats the purpose, because the receiver cannot recognize a repeat. Derive the key once from stable inputs and reuse it.

Not counting retries

If only final failures are logged, a degrading dependency is invisible until the retries stop being enough. Record each attempt, with its class and delay.

Key takeaways

- A retry is a promise to try again, not a promise to succeed, so the first question on any failure is whether trying again can possibly help.
- Classify failures as transient, permanent or unknown, write the classification down as a table, and treat unrecognized failures as unknown with a stricter limit.
- Use exponential backoff with jitter, bound the whole operation with a deadline, and share a retry budget so that retries cannot become the dominant load on a failing dependency.
- A circuit breaker decides whether to call at all; an open breaker is a deferred signal that costs no attempt, and for a consumer the strongest form is to pause consumption. Backpressure does the same job earlier, by bounding in-flight work and letting live traffic win over retries and redrive.
- Retries can live in the handler, in broker redelivery and in retry topics, and they should be designed together under one attempt budget, with in-handler time kept inside the broker's timeouts.
- Moving a failing message aside reorders its aggregate's events; choose between blocking in place and tolerating reordering according to the ordering requirement of the stream.
- A retry policy is safe only on top of idempotent operations, with the idempotency key derived once and reused on every attempt.

Retries decide how often you try again, but they cannot decide whether trying twice is harmless. Chapter 7 builds the other half: the idempotent consumer, which turns a duplicate from a disaster into a non-event.

Chapter 7 — Idempotency

Imagine Parcelly's Ledger service in the middle of a rolling deploy, while its consumers are restarted one at a time. A consumer has read a `PaymentCaptured` event, written a ledger entry for 4,990 minor units of EUR, and is about to acknowledge the message when the process receives its shutdown signal. The ack never leaves the machine. The broker, doing exactly what at-least-once delivery promises, hands the same message to the next consumer that asks for it.

The second consumer reads the event, finds nothing unusual, and writes a second ledger entry. The ledger is append-only and correct by construction, so it now records, faithfully and permanently, that the buyer paid twice. Nothing has crashed. No alert has fired. Every component behaved as designed, and the books are wrong.

Notifications has its own version of the same afternoon. A buyer receives the "your parcel is scheduled" email three times, because the message was redelivered after a slow handler outran its visibility timeout. That one is embarrassing. The ledger one is expensive.

Earlier chapters established that duplicates are normal: the broker redelivers, the relay publishes twice after a failover, and the retries of Chapter 6 are duplicates by design. This chapter answers the question those chapters leave open. When the same message arrives twice, what exactly should a consumer do so that the second arrival changes nothing?

The problem, and why it is hard

The goal sounds modest: process each message once, as far as the world can tell. The difficulty is that the consumer cannot know whether it has seen a message before without remembering, and remembering is itself a write that can fail at the wrong moment.

Start with where duplicates come from, because the list is longer than the broker.

Source	How the duplicate appears
Broker redelivery	The ack was lost, the visibility timeout expired, or a rebalance moved the partition before the offset was committed
Relay	A relay published a row, crashed before marking it published, and published it again after restart (Chapter 5)
Application retry	A consumer retried a call, or a producer retried a publish whose first attempt had in fact succeeded (Chapter 6)
Redrive and replay	Someone moved dead letters back, or deliberately re-read an event log from an earlier offset
Humans and clients	A buyer double-clicked "Pay", a mobile app retried on a flaky network, a webhook sender delivered twice

Most of these produce the *same* message twice, with the same `event_id`. A few produce two different messages that express the same intent, and that distinction will matter when we choose a key.

The tempting solution is to look before acting. Ask whether this message has been processed, and if not, process it. I want to name this as the **check-then-act** anti-pattern, because it feels correct and fails quietly.

```
function handlePaymentCaptured(message):  
  // UNSAFE: the check and the action are separate steps  
  if inboxContains("ledger-poster", message.event_id):  
    ack(message)  
    return  
  
  postLedgerEntry(message.payload)  
  inboxInsert("ledger-poster", message.event_id)  
  ack(message)
```

Read it under failure. If the process dies after `postLedgerEntry` and before `inboxInsert`, the entry exists and the inbox does not know, so redelivery posts it again. Reverse the two writes and the failure reverses too: a crash after `inboxInsert` and before the ledger write marks the message as handled when it was not, and the effect is lost forever. Neither order is safe, because the two writes are not one atomic act.

There is a second failure that has nothing to do with crashes. Two consumers can receive the same message at nearly the same moment, for example when a slow first attempt outlives its visibility timeout. Both run the check, both find nothing, and both proceed. The window between "check" and "act" is small, and at sufficient volume it is hit routinely.

So the real problem has two halves. The memory of what was processed must be written atomically with the effect, and the memory must be able to arbitrate between concurrent attempts. The rest of the chapter builds both, starting with the case where you need neither.

Natural and forced idempotence

An operation is **idempotent** when applying it twice leaves the system in the same state as applying it once. Some operations are idempotent by their shape, and I would look for these before building any machinery.

Setting a value is idempotent: `set order.status = "PAID"` can run 10 times and the order is still `PAID`. Deleting a row by id is idempotent. Inserting a row with a primary key that you derive from the event, and ignoring the conflict, is idempotent. Writing the full new state of an aggregate ("the shipment is now `DISPATCHED`, with this tracking number") is idempotent. I call this **natural idempotence**: the duplicate has no extra effect because of what the operation is, not because of anything you added.

Other operations are not. `balance = balance + 4990` applies twice as two deposits. `send email` sends two emails. `append a ledger entry` appends two. `increment the counter of units sold` counts the unit twice. Calling a payment provider's capture endpoint without a key may charge twice. For these, idempotence has to be added, which I call **forced idempotence**: you record that the work was done and consult the record before doing it again.

Natural idempotence has two honest limits. The first is that it quietly depends on order. Setting `status = "PAID"` is idempotent, but a delayed duplicate of an older event can overwrite a newer status: a redelivered `OrderPlaced` that arrives after `OrderConfirmed` and sets the status back

to `PLACED`. The operation is idempotent and the system is still wrong. Chapter 9 covers ordering and version checks; for now, notice that "set to a value" is safe against duplicates only when the value cannot go stale.

The second limit is that it is tempting to overestimate. A handler that "only updates a row" often also writes an audit record, emits a follow-up event, or bumps a counter, and each of those is an increment in disguise. When I review a consumer, I list every effect it has and ask of each one separately whether it survives a second run. One non-idempotent effect is enough to need the inbox.

My rule of thumb is to prefer natural idempotence where the effect permits it, and to write down in the handler why it does. When the effect is a counter, an append, an email, or an external call, use the forced kind. And when in doubt, use the inbox anyway: its cost is small and predictable, and the cost of being wrong is not.

Choosing the idempotency key

An **idempotency key** is what the consumer uses to recognize repeated work. Before building the table, you have to decide what goes in it, and the choice shapes what duplicates you can catch.

The `event_id`

For event consumption the default is the `event_id` from the envelope. It is unique per event and, crucially, **stable across redelivery**: every copy of the message the broker hands you carries the same value. The producer assigns it once, when the event is written to the outbox (Chapter 4), so a relay that publishes twice publishes the same `event_id` twice. That is exactly the property a dedup check needs.

The `event_id` identifies a *message*. It answers "have I handled this specific fact before?" and nothing broader.

The business key

Sometimes two different messages describe the same intent. Suppose `Orders` retries a `CapturePayment` command after a timeout, and because of a bug or a design choice each retry is written as a new command with a

fresh `command_id`. Payments now receives two commands with different ids that both mean "capture the payment for order 81724". Deduplicating on the message id catches neither.

The remedy is a **business key**: an identifier derived from the meaning of the work rather than from the envelope. For capture it might be the `payment_id` plus the operation, say `capture:payment-5521`. For a ledger entry it might be the payment id plus the entry kind. A business key says "this unit of work may happen once", regardless of how many messages ask for it.

Aspect	event_id / command_id	Business key
Catches redelivery of the same message	Yes	Yes
Catches two messages for the same intent	No	Yes
Who assigns it	The producer, automatically	A human decision per operation
Risk	Misses logical duplicates	Too coarse a key rejects legitimate work
Stored where	Generic inbox table	Often a unique constraint on the domain table

The risk in the second-to-last row deserves a concrete case. If the business key for refunds is only `payment_id`, a second, legitimate partial refund of the same payment is rejected as a duplicate. The key must include everything that makes two requests *genuinely different*: here, perhaps a refund idempotency key supplied by the caller. Choosing the key means deciding what "the same work" means in your domain, which is a design decision and not a framework setting.

In my view the layers work well together. Use the `event_id` in a generic inbox as the baseline for every consumer, and add a business-key unique constraint on the domain table wherever two different messages could plausibly mean the same thing. The inbox protects against the broker and the relay. The constraint protects against the rest of the system, including its bugs.

The inbox pattern

The **inbox** is the table of processed message ids on the consumer side. Chapter 4 gave the producer a transactional outbox so that a state change and its event commit together. The inbox is the mirror image: the consumer commits its state change and its record of having handled the message together, in one local transaction in its own database.

```
table inbox:
  consumer      text not null
  event_id      uuid not null
  processed_at  timestamp not null
  primary key (consumer, event_id)
```

The `consumer` column lets one service run several independent consumers against one table, so that the Ledger's posting consumer and its reporting consumer each remember their own history. That is why the primary key is the pair `(consumer, event_id)`. With `event_id` alone, the first consumer to handle a message would make the second consumer's insert fail, and the second consumer's effect would be skipped as if it were a duplicate. A service with a single consumer per table can drop the column and key on `event_id`, which is the simpler form used elsewhere in the book.

The handler writes the inbox row and the business change in the same transaction.

```
function handlePaymentCaptured(message):
  if inboxContains("ledger-poster", message.event_id):
    // fast path only; this check is not the safety net
    ack(message)
    return

  transaction:
    postLedgerEntry(message.payload)
    // primary key (consumer, event_id): a concurrent duplicate
    // fails here, the transaction rolls back, and the
    // message is redelivered and finds the committed row
    inboxInsert("ledger-poster", message.event_id)
  // commit point: effect and inbox row exist together or not at all

  ack(message)
```

Walk through the failures. If the process dies before the commit, neither the ledger entry nor the inbox row exists, so redelivery processes the message from scratch. If it dies after the commit and before the ack, both exist, and redelivery takes the fast path: the check finds the row, the message is acknowledged, and nothing is written. The effect and its memory can never disagree, because they commit as one.

The check at the top is an optimization. It saves a transaction in the common duplicate case, and it can be removed without losing correctness. The safety net is the unique constraint inside the transaction, which is where the real arbitration happens.

Inside a transaction, the order of the writes does not change atomicity, so the warning about reversing the two writes applies only to the unsafe version above. It does change cost. In this version a concurrent duplicate does all the work before it fails on the insert, and in some databases a violation aborts the whole transaction, so the handler cannot recover inside it. I would claim first: insert the inbox row with "on conflict do nothing" semantics, look at whether a row was inserted, and skip the effect if not. The duplicate then waits on the key, does no work, and never raises. The next section's example uses that form.

A word on `ack`, because the term hides a difference between the two families. On a queue-based broker it acknowledges one message. On a log-based broker it is usually a commit of an offset, which means "everything up to here", and a consumer that processes a partition in parallel can commit past a message it has not finished, and lose it. The inbox protects against duplicates, not against that, and Chapter 9 looks at it where ordering and parallelism meet.

Follow-up events from a consumer

Many consumers do more than change their own state: they announce what they did. Ledger posts an entry and emits `LedgerEntryPosted`. That follow-up event is an effect like any other, and it has to survive a crash exactly once, not zero times and not twice.

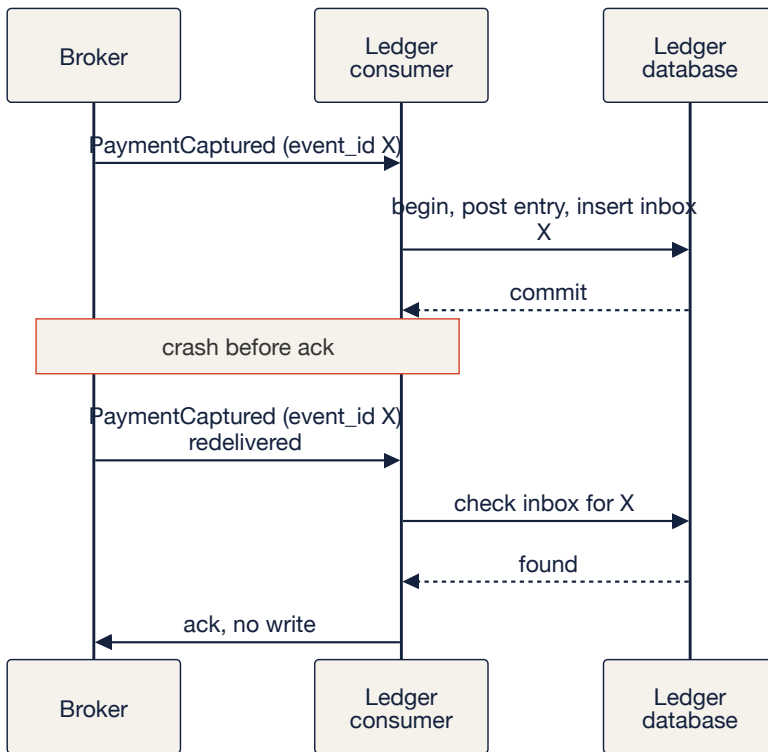
A handler that publishes the follow-up directly after its commit loses it permanently on a crash. The redelivery takes the inbox fast path, finds the row, acknowledges, and skips the publish. The state is right and the event

never existed. The fix is Chapter 4's outbox applied on the consuming side: the inbox row, the state change and the outbox insert commit in one local transaction.

```
function handlePaymentCaptured(message):
  transaction:
    claimed = inboxInsertIfAbsent("ledger-poster",
    message.event_id)
    // on conflict do nothing: a concurrent duplicate waits on the
    key,
    // then sees claimed = false and does no work
    if claimed:
      postLedgerEntry(message.payload)
      outboxInsert({
        event_type:    "LedgerEntryPosted",
        causation_id:  message.event_id,
        partition_key: message.partition_key
        // remaining envelope fields as in Chapter 4
      })
    // commit point: inbox row, ledger entry and follow-up event, or
    none of them
    ack(message)
```

If the process dies before the commit, nothing exists and the redelivery does all of it. If it dies after the commit, the follow-up sits in the outbox and the relay publishes it, while the redelivery finds the inbox row and does nothing. The consumer's outbox has the same at-least-once publication as the producer's, so whoever consumes `LedgerEntryPosted` needs its own inbox. Chapter 10 builds on this rule when a saga step reacts to an event and emits the next one.

This is also why the inbox lives in the *same database* as the state it protects. If the processed-message record sat in a cache or in a different database, we would be back to two writes with no shared transaction, which is the dual write of Chapter 4 on the consumer side. The inbox is not a technique you can bolt on from the outside of the service's data store.



Look at the note in the middle: the crash window is between the commit and the ack, and the inbox row is what turns the redelivery into a no-op.

Concurrent duplicates

The sequence above assumes the duplicate arrives after the first attempt has finished. Concurrency is harder, and the unique constraint is what saves you.

Picture two consumers processing the same message at once. Both start a transaction. Both do the work. Both try to insert the inbox row. The database lets exactly one of them commit; the other receives a unique-constraint violation, its transaction rolls back, and its ledger entry vanishes with it. The loser then leaves the message unacknowledged, or acknowledges it after re-checking, and either way the ledger holds one entry.

That is the whole mechanism, and it is worth saying what it does not require: no distributed lock, no coordination between the two consumers, no assumption about timing. The database's constraint is a single point of

arbitration that both attempts must pass through.

```
function handleSafely(message):
  try:
    transaction:
      applyChange(message)
      inboxInsert("ledger-poster", message.event_id)
      // commit point; a losing concurrent attempt rolls back here
      ack(message)
    on UniqueViolation as e:
      if e.constraint == "inbox_pkey":
        // someone else committed this message first; the work is
done
        ack(message)
      else:
        // a business constraint failed: a real error, not a
duplicate
        raise e
```

Whether a unique violation should be acknowledged or retried is a small decision that is worth making deliberately. If the violation is on the inbox key, the message has been handled by a committed transaction and acknowledging it is correct. If it is on some other constraint inside `applyChange`, the error means something else entirely, and acknowledging it would hide a real bug. In my view the handler should distinguish them, which the handler can do by checking the name of the violated constraint, as the code above does.

Longer handlers and in-progress work

The pattern above holds the whole handler inside one transaction. That is fine when the work is a few writes. It is uncomfortable when the handler calls out to something slow, because the transaction stays open, locks stay held, and the visibility timeout may expire while you wait. The expired message is redelivered to a second consumer, which starts the same slow work in parallel.

There are two sound responses. The first is to keep the transaction short: do the slow external work *outside* it, using the external-effects approach in the next section, and put only the local state change and the inbox row inside the transaction. The second, when the work genuinely must be claimed, is to make claiming explicit with a status.

```
table inbox:
  consumer      text not null
  event_id      uuid not null
  status        text not null    // "processing" or "done"
  lease_until   timestamp null
  lease_token   bigint not null  // increases on every takeover
  primary key (consumer, event_id)
```

A consumer inserts the row as `processing` with a lease, and that claim commits in its own short transaction before the slow work starts. A duplicate that finds a live lease must not acknowledge the message, because the first attempt may still fail and the acknowledgement would erase the only retry. It queues the message with a delay, longer than the remaining lease, and looks again on the next delivery. One that finds an expired lease may take over with a newer lease token, which the original holder must check before it commits its local changes; one that finds `done` acknowledges.

The token protects only what can check it. The local transaction can: it locks the inbox row, compares the token, and rolls back if a newer one exists. The slow external call cannot be rolled back, so the effect gets the deterministic key from the next section, and a stale holder that wakes up and repeats the call is recognized by the downstream. This is the fencing idea from Chapter 5's relay lease, with the same limit: a fence works only where the protected resource checks the token.

The scheme works, and it adds a state machine that somebody has to reason about, including the question of what happens when a worker dies holding a lease. I would reach for it only when the first approach is not available, and I would keep the lease short and renewable.

Warning: Do not rely on a prior `SELECT` or a row lock to keep duplicates out. Whether a second consumer sees the first one's uncommitted work depends on the database's isolation level, and a check that is correct under one level fails under another. Raising the level to serializable does not remove the need for the constraint either: it replaces a silent duplicate with a serialization failure, which the handler must treat as retryable and run again from the start. Locks reduce contention; the unique constraint inside the transaction is the guarantee.

Dedup windows and retention

An inbox that keeps every id forever grows forever. At some point you will want to delete old rows, and deletion changes the guarantee in a way that deserves a name: the **dedup window**. A message whose inbox row has been deleted can be processed again if it ever reappears.

So how long must the window be? It has to cover the longest time a duplicate could plausibly arrive after the original. I don't know that number for your system, and it depends on a short list of things.

- **Broker retention and redelivery horizon.** A queue that redelivers for four days, or a log that retains a week, means duplicates can arrive that late.
- **Dead-letter and redrive practice.** If a team may redrive dead letters a week after they failed, the window must outlast that (Chapter 8).
- **Replay.** Re-reading an event log from an old offset creates duplicates of events as old as you replay. Either the window is long enough, or a replay is run against a consumer that is deliberately reset, which is a procedure you design in advance.
- **Relay failover.** Double publishes after a failover are usually close in time to the original, so they sit well inside any window you would choose.

A common shape is a retention period comfortably longer than the broker's own retention, say a few weeks as an illustration, with a scheduled job that deletes in batches. Partitioning the inbox table by time makes deletion cheap, because dropping a whole partition costs far less than deleting millions of rows.

Keep in mind that the inbox is not the only line of defense. A business-key constraint on the domain table lives as long as the data does, so it protects against duplicates arriving after the inbox row is gone. This is another reason to use both layers: the inbox gives you cheap, bounded-memory protection against the common case, and the constraint covers the tail.

If the system can accept it, there is an alternative to a time window: derive the key from something monotonic. The rule is narrower than it first looks. It applies to state-carrying events, the kind Chapter 9 distinguishes from deltas, where each event holds the full new state of an aggregate. For

those, the consumer stores the highest aggregate version applied and rejects anything that is not newer. The offset is a position within one partition and the version is a property of one aggregate, so the two are not interchangeable: an offset says nothing about aggregates that live on other partitions.

For delta events, "reject if not newer" is wrong, because a skipped version means a lost change. Such a consumer applies version `last + 1`, buffers a later one until the gap fills, and treats anything at or below `last` as a duplicate; Chapter 9 describes the buffering and its limits. Either way the memory is bounded with no TTL, and it depends on ordering guarantees that Chapter 9 examines. It suits aggregates with a version counter and does not suit consumers fed from several partitions.

External side effects

The inbox protects effects that live in your own database. Many effects do not. An email leaves your system the moment you send it, and a call to the payment provider charges a card whether or not your transaction later commits. No local transaction can roll these back, so the atomicity trick does not apply.

Consider Payments handling a `CapturePayment` command. It must call the external provider and then record the result.

```
function handleCapturePayment(command):  
    // unsafe for the crash we care about: the provider call has no  
    key  
    result = provider.capture(command.payload.payment_id)  
    transaction:  
        recordCapture(command.payload.payment_id, result)  
        inboxInsert("payments-capture", command.command_id)  
    ack(command)
```

Under a crash between the provider call and the transaction, the money has moved and Payments has no record. On redelivery the handler calls the provider again. Whether the buyer is charged twice now depends entirely on the provider.

Passing the key downstream

The fix is to carry the idempotency key to the system that performs the effect, so that it can recognize the repeat. Payment providers commonly support this: you send a key with the request, and a second request with the same key returns the first result instead of acting again. The details of the header or field vary by provider, and I would read your provider's documentation for its exact behavior and for how long it remembers keys.

The detail that matters most is *how you choose the key*. It must be **deterministic from the work, not generated per attempt**. A random key created inside the handler looks identical to a fresh request on every retry, and protects nothing.

```
function handleSafely(command):
    try:
        if inboxContains("payments-capture", command.command_id):
            // local memory, kept as long as the inbox row: the long-
            term guard
            ack(command)
            return

            key = "capture-" + command.command_id
            // same command, same key, on every attempt and every
            redelivery
            result = provider.capture(command.payload.payment_id, key)
            // an "in progress" or conflict answer for a concurrent
            request with
            // this key is a retryable failure, not a result to record

            transaction:
                recordCapture(command.payload.payment_id, result)
                inboxInsert("payments-capture", command.command_id)
            // commit point
            ack(command)
        on UniqueViolation as e:
            if e.constraint == "inbox_pkey":
                // a concurrent attempt recorded it first
                ack(command)
            else:
                raise e
```

Now trace the failures again. A crash after the provider call and before the commit leaves the money moved and Payments unaware. The redelivered command runs the same code with the same key, the provider returns the original result, and the transaction records it. A crash during the provider call, where the response is lost, is the same situation: the retry carries the

same key, and the provider either performs the capture once or returns the earlier outcome. Two consumers racing on the same command send the same key at once, and a provider that rejects the second as "in progress" has not failed the work: the handler treats that answer as retryable and looks again on the next delivery.

The provider's memory and the inbox are not the same protection. A provider remembers keys for a limited time, and I would read its documentation for how long. A redelivery that arrives after that period would make a second capture. The local inbox row, kept at least as long as the dedup window described earlier, is what stops the handler from calling the provider at all, so the provider's key retention is never the window you rely on.

If the work is keyed by the business and not the message, use a business-derived key here too, such as the payment id plus the operation. Then even two commands with different ids reach the provider as one request.

When the downstream has no key

Not every downstream accepts one. Plain SMTP has no idempotency key, and many email and SMS providers do not offer one reliably. You then face a choice between two imperfect outcomes, and I would make it explicitly, per effect.

Send before recording, and a crash can lead to a duplicate email. Record before sending, and a crash can lead to a missing one. These are at-least-once and at-most-once behaviors for the effect. For a "your parcel is scheduled" notification, a rare duplicate is cheap and a lost message is a support ticket, so I would send first and accept the occasional repeat. For a one-time voucher code that grants value, I would lean the other way and add a way to detect and repair a missing send.

Some effects can be made safer by recording the *intent* in the local transaction and performing the effect from a separate worker, which is the out-box idea applied to commands. It does not eliminate the problem at the provider boundary, because the worker can still crash between the call and the mark. It does move the problem into one place where you can see it, retry it deliberately, and reconcile it. Chapter 13 covers reconciliation:

comparing your records with the provider's and repairing the differences, which is the final defense for exactly the effects that cannot be made idempotent.

Idempotency-Key and HTTP APIs

The same problem appears at your own front door. A buyer's phone submits "place order", the network drops the response, and the app retries. Orders cannot tell a retry from a second order unless the client says so.

The widely used convention is an `Idempotency-Key` request header: the **client** generates a unique value for each logical operation and sends the same value on every retry of it. This is the other half of the definition given earlier: for event consumption the key is normally the `event_id`, which the producer assigned, and for client-facing requests it is supplied by the caller. The server's job is to make a repeated key behave like the first request.

A server needs more than the key. It stores the key together with a fingerprint of the request and the response it produced.

```

function handleCreateOrder(request):
    key = request.header("Idempotency-Key")
    if key is missing:
        return error(400, "Idempotency-Key required")
    try:
        transaction:
            record = idempotencyFindForUpdate(request.caller, key)
            if record is none:
                token = idempotencyInsert(request.caller, key,
fingerprint(request.body),
                                                    "in_progress", lease_until:
dbNow() + LEASE)
            else if record.fingerprint != fingerprint(request.body):
                return error(422, "key reused with a different
request")
            else if record.status == "completed":
                return record.stored_response
            else if record.lease_until > dbNow():
                return error(409, "request in progress")
            else:
                // the holder crashed or stalled: take over with a
newer token
                token = idempotencyTakeOver(request.caller, key,
lease_until: dbNow() + LEASE)
                // commit point: the key is claimed, fenced by an increasing
token
                on UniqueViolation as e:
                    if e.constraint == "idempotency_pkey":
                        // a concurrent first claim won
                        return error(409, "request in progress")
                    raise e
                transaction:
                    // locks the row; raises LeaseLost
                    requireLeaseToken(request.caller, key, token)
                    response = createOrderWithOutbox(request.body)
                    idempotencyComplete(request.caller, key, response)
                // commit point: on LeaseLost the order and its outbox rows roll
back
                return response

```

The fingerprint check matters. A client that reuses a key for a different body has a bug, and silently returning the first response would hide it. The in-progress state matters for the concurrency case: two simultaneous requests with one key must not both execute. The row lock only orders a takeover of an existing row; the first claim is arbitrated by the unique constraint, which is why the losing insert is caught and answered with 409.

The lease matters for the crash case, and its token matters for the slow case. A holder paused past its lease may wake up and try to finish, but `requireLeaseToken` sees the newer token and its whole transaction, order

and outbox rows included, rolls back. Here the token works as a fence because the protected resource, the local database, checks it inside the transaction, the same idea as Chapter 5's relay lease. The lease length only shortens the overlap; the token is what makes the stale holder's commit fail. The stale caller gets an error, and its retry receives the stored response. Both checks use the database's clock, `dbNow()`, not the application server's, for the reasons Chapter 3 gives about clocks. Scoping the key to the caller keeps one client from colliding with another.

The pseudocode shows the success path, and the failure path needs a rule too. If the request fails for a deterministic reason, such as a validation error that returns a 4xx, the server stores that response as the completed one. A retry with the same key then gets the same 4xx and does not run the validation again against data that may have changed. If it fails for a transient reason, such as a timeout or an unavailable dependency, the claim is released or left to expire, so that the client's retry can run the work. Caching a transient failure as the final answer would turn a blip into a permanent error for that key.

The unique constraint on `(caller, key)` plays the same arbitrating role as the inbox's primary key. The pattern is the same pattern: an atomic claim, a stored outcome, and a deterministic answer for a repeat. Where the API produces events, the order row, the outbox rows and the idempotency record's completion can all be written in one local transaction, so a retried request neither creates a second order nor emits a second `OrderPlaced`. Keep the stored key for at least as long as clients may retry, which is a decision about client behavior and not about brokers.

A connection worth stating: the key the client supplied can flow into the event. Orders can put it in the event headers or derive the `OrderPlaced` `event_id` from it, so that downstream consumers see one identity from the buyer's tap to the ledger entry. If the `event_id` is derived, it must come deterministically from the caller and the key together, for example as a name-based UUID over both, so that two callers who happen to pick the same key never share an identity. Chapter 11 discusses what belongs in an event; whether the key belongs there depends on how far you want idempotency to follow the request.

Trade-offs and costs

The inbox adds a write to every message. Each consumed message costs one extra insert, inside the same transaction, on the hot path. For a consumer doing a handful of writes the overhead is small. For a very high-volume, very cheap consumer, such as one that updates a read-model cache, it may be the dominant cost, and natural idempotence is often the better choice there.

Inbox storage and cleanup are an ongoing job. The table grows with traffic, and the retention policy is a decision you own and revisit. Too short a window re-admits old duplicates; too long a window costs storage and index size. Someone must watch the cleanup job, because when it stops, the table grows quietly until it becomes a performance problem.

A unique constraint on the hot path can become a contention point. With a well-distributed key it is cheap. With a key that clusters, or with many consumers hitting one hot aggregate, it can queue up. Keying the table by consumer and event id, and keeping the handler transaction short, keeps this manageable.

Business keys are a modeling decision, and a wrong one is hard to see. Too fine, and logical duplicates pass through; too coarse, and legitimate work is rejected. The failure is silent in both directions. I would test the key with deliberately awkward cases, such as two legitimate operations on one aggregate, before relying on it.

Idempotency does not cover everything. It makes a repeat harmless; it does not fix a handler that is wrong the first time, it does not order messages, and it does not undo an effect the provider has no key for. An inbox also does not protect against a duplicate that has *changed*: if a producer republishes "the same" fact with a different `event_id` and no business key exists, nothing recognizes it.

When not to use the inbox. If every effect of the consumer is naturally idempotent and cannot go stale, such as overwriting a denormalized field with the latest full state, the inbox is cost without benefit, and I would skip it and record the reasoning in the code. Where it is also unsuitable is a consumer that cannot write to a transactional database of its own, such

as a stateless forwarder. That consumer needs its downstream to deduplicate, with the key passed along, or it needs to be redesigned so that it owns a store.

Common mistakes

- **Checking and acting in separate steps.** It is tempting because the code reads naturally: "have I seen this? no, then do it." It fails under a crash between the two and under concurrency. Put the inbox write in the same transaction as the effect and let the unique constraint arbitrate.
- **Writing the inbox row after acknowledging, or before the effect commits.** Acking first loses the message if the work then fails. Recording before the effect commits marks work as done that never happened. The effect, the inbox row and the commit are one unit; the ack follows the commit.
- **Keeping the inbox in a different store from the effect.** A cache or a separate database feels cheaper, and it recreates the dual write. Keep it in the same database as the state it protects.
- **Generating a fresh key per attempt.** A random value created inside the handler or the client's retry loop makes every retry look new. Derive the key from the command, the event or the business operation, so it is identical on every attempt.
- **Using only `event_id` and assuming that covers all duplicates.** It does not catch two different messages for the same intent. Add a business-key constraint where the domain has such a risk.
- **Choosing a dedup window by guessing.** A window shorter than the broker's retention, or than your redrive practice, silently re-admits old duplicates. Derive the window from those numbers and write the derivation down.
- **Treating a unique violation as a generic error.** Acknowledging every violation hides real bugs; retrying every violation loops forever on a handled message. Catch the inbox violation on its own.
- **Forgetting the external effect.** The inbox cannot roll back an email or a card charge. Pass a deterministic key to the provider, or decide deliberately between a possible duplicate and a possible loss.

- **Publishing follow-up events directly from the handler.** A crash between the commit and the publish loses the event, and the redelivery takes the inbox fast path and never repeats it. Write the follow-up to the outbox in the same transaction as the inbox row and the state change.
- **Acknowledging a message whose lease belongs to someone else.** A duplicate that sees a live lease and acks has removed the only retry if the first attempt then fails. Requeue it with a delay instead.

Key takeaways

- Duplicates come from the broker, the relay, retries, redrive and replay, and from clients, so every consumer should be written as though any message can arrive twice.
- Check-then-act is unsafe because the check and the effect are separate steps, and a crash or a concurrent attempt can slip between them.
- The inbox row and the business change must commit in one local transaction in the consumer's own database, and the unique constraint on the key is what actually arbitrates concurrent duplicates. A consumer that announces what it did adds an outbox row to that same transaction.
- Use the `event_id` as the baseline idempotency key for event consumption, and add a business key wherever two different messages can express the same intent.
- The dedup window must outlast the broker's retention, your redrive practice and any replay you plan, and a business-key constraint on the domain table protects you after inbox rows are deleted.
- For external side effects, send a deterministic idempotency key derived from the work and not from the attempt, keep the local inbox row as the long-term guard because the provider forgets keys, and where the downstream has no key, choose consciously between a possible duplicate and a possible loss.
- At the API boundary the caller supplies the idempotency key, and the server stores the key, a request fingerprint and the response, so that a repeat returns the first outcome.

Idempotency makes a repeated message harmless, but it assumes that processing eventually succeeds. Some messages never will: they fail on every attempt, and the retries and redeliveries pile up behind them. Chapter 8 deals with those poison messages and with the dead letters they leave behind.

Chapter 8 — Poison Messages and Dead Letters

Imagine Parcelly the morning after a new field, `gift_message`, shipped on `OrderPlaced`. Notifications reads it to build the confirmation email, and one seller's test order carries a gift message with a character the email template cannot encode. The consumer throws on that single message. The message comes back. The consumer throws again.

By lunchtime the message has been delivered several thousand times. Each attempt fails in the same place, in the same way, within milliseconds. The consumer is busy, healthy by every infrastructure metric, and has not acknowledged anything useful in hours. On a queue-based broker, other messages are still served by the remaining workers, so the damage looks small. On a log-based topic, one partition has stopped moving because the consumer keeps seeking back to the failing record instead of stepping past it, and every order on that partition waits behind it.

Someone eventually notices, and someone suggests moving the message somewhere else so work can continue. That is the short half of the answer. The hard half is what happens next: who looks at that message, how they decide what it means, and how it gets back into the system without causing a second incident. This chapter is about both halves. What should a system do with a message it cannot process, and how does a team keep that answer from becoming a landfill?

The problem, and why it is hard

A **poison message** is a message that fails every time it is processed. The word "every" carries the definition. A message that fails because a database was briefly unreachable is not poison; it will succeed once the database returns. A poison message fails because something about the message, or about the code reading it, makes success impossible until a hu-

man or a deployment changes something. In the vocabulary of Chapter 3, the failure's semantics say that trying again cannot help, and that is exactly why retrying it is wasted effort.

The usual causes fall into a few groups:

- **Malformed or unexpected content.** A required field is missing, a value is out of range, or a payload does not parse. Schema changes that were not compatible with an old consumer produce this (Chapter 12 covers how to avoid it).
- **A bug in the consumer.** The message is valid, but the code mishandles a case nobody tested, such as a null where one was never expected.
- **A violated business assumption.** A `PaymentCaptured` event arrives at a consumer that has no record of the matching authorization, perhaps because data was deleted, or because events crossed in flight and retries have already outlasted any reasonable reordering window. Within that window the same symptom is transient, as Chapter 6 notes and Chapter 9 covers in depth, and belongs to retries rather than the DLQ. Past it, the consumer cannot proceed and cannot invent the missing record.
- **A permanent downstream rejection.** An external carrier refuses an address as invalid. Trying again produces the same refusal.

Chapter 6 introduced the distinction between transient and permanent failures, and this chapter picks up where that one ends. A retry policy that is working correctly will, eventually, give up on a message. Retries are bounded by attempts, by time, or by budget. What happens at that point is a design decision, and if nobody makes it, the broker makes it for you.

The infinite redelivery loop

If the consumer does nothing special, the broker's default behavior takes over. A queue-based broker redelivers an unacknowledged message, either immediately after a nack or after the visibility timeout expires, and it usually keeps a redelivery count that grows with every attempt.

A log-based broker works differently, and the difference matters. There is no per-message redelivery: the consumer holds an offset and reads forward. Declining to commit an offset does not make the broker hand the

record back; the consumer's in-memory position keeps moving, and the failed record is silently skipped until the next restart or rebalance rewinds the consumer to the last committed offset. A loop on a log exists only because the consumer builds one: it seeks back to the failed offset and reads it again, or it stops advancing the partition while it retries in place. A consumer that does neither, and commits past the failure anyway, loses the record without any signal.

In both families the outcome can be the same: the message returns, fails and returns again, or, on a log with careless offset handling, it vanishes. The correct log-family tools are an explicit seek or pause, a retry topic that receives the failed record so the main partition can advance, and a dead-letter topic for records given up on.

The loop has three costs. The first is wasted capacity: workers spend their time on work that cannot succeed. The second is noise: logs fill with identical stack traces, and real problems hide among them. The third is the cost that matters most, **head-of-line blocking**. On a log, a consumer processes a partition in order, so a record it will not skip blocks every record behind it. On a plain queue, one poison message does not block others by itself, but a handful of them can occupy every worker. If the queue groups messages for ordering (FIFO queues, message groups, sessions), a poison message blocks its group, which is the same blast radius as a log partition, only narrower.

The two bad escapes

Teams under pressure usually reach for one of two escapes, and both are wrong for the same reason: they hide the failure instead of handling it.

The first is to **drop the message**: catch the exception, log it, acknowledge and move on. The pipeline flows again, but the business event is gone. For `OrderPlaced` that means an order that exists in Orders and in no other service. For `PaymentCaptured` it means a ledger that does not match reality. Dropping is the correct choice only when the message carries no value, and a consumer cannot know that at the moment of failure.

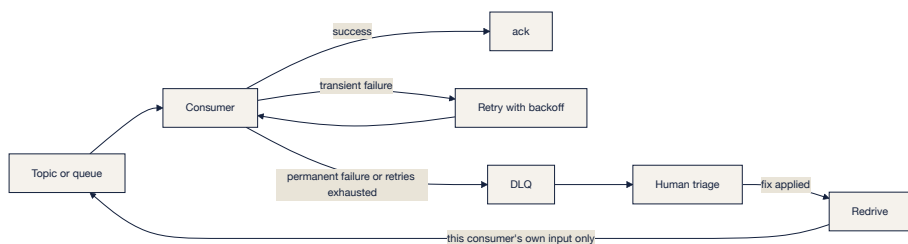
The second is to **retry forever** and trust that someone will fix the cause. This is the infinite loop above, and it converts one bad message into an outage for everything behind it.

The third option sits between them. Set the message aside, with enough context that a person can understand it later, and let the rest of the stream continue. That option is the dead-letter queue.

The dead-letter queue pattern

A **dead-letter queue (DLQ)** is a separate destination where a consumer or a broker places messages it has given up on. On a log-based broker the same idea is usually a **dead-letter topic**: an ordinary topic that receives failed records. When a statement holds for both families, this chapter says DLQ.

The pattern has a simple core. After a message exhausts its retries, or fails in a way the consumer classifies as permanent, the consumer moves it to the DLQ and acknowledges the original. The stream continues. The failed message is not lost; it is parked where it can be examined.



Look at the last two arrows in the diagram. The DLQ is not an exit. It is a waiting room with a door back to the failing consumer's own input, and the whole difficulty of this chapter sits in the distance between those two doors. The door does not lead to a shared topic: other consumers already handled the event, and they should not see it again.

Where the move happens

There are two ways to implement the move, and the choice affects what you can store.

Broker-managed dead-lettering. Many queue-based brokers can be configured so that a message delivered more than some number of times is moved to a designated queue automatically. This costs little to set up and

works even when the consumer crashes before it can do anything. It gives you a delivery count, but usually not the reason the consumer failed, because the broker does not know it.

Consumer-managed dead-lettering. The consumer catches the failure, builds a record that includes the reason, publishes it to the DLQ, and only then acknowledges the original. This lets you classify failures and dead-letter a permanent error on the first attempt instead of the fifth. It costs code in every consumer, and it must be written carefully: publish to the DLQ first and acknowledge second, so a crash between the two produces a duplicate in the DLQ, never a loss.

I would use both where the broker offers the first. The broker-managed rule exists only on some queue-based brokers, and not every queue type limits deliveries. Where it exists, it is the backstop for consumers that crash or hang in ways code cannot catch, and together with the consumer-managed path it means a poison message cannot loop forever, even if your own handling has a bug.

Log-based brokers generally have neither a per-message delivery counter nor a broker-side DLQ, and that leaves a gap. A message that crashes the process, such as a deserializer failure before the handler runs, an out-of-memory error or a native crash, never reaches your `try` block. The consumer restarts at the committed offset, reads the same record and crashes again. Chapter 6 noted that the attempt count starts over after a crash. For a poison record that is the whole problem.

The remedy is to count before you process, so the count survives the crash:

```
function pollLoopStep(record):
    key = record.partition + ":" + record.offset
    // the counter is written before any work, so a crash still counts
    attempts = incrementDurableCounter(key)
    if attempts > MAX_PROCESS_STARTS:
        // already tried too many times: do not run the code that
        crashes
        deadLetterRaw(record, "exceeded process starts")
        clearDurableCounter(key)
        return
    // deserialization is inside the guarded path, not before it
    handleMessage(deserializeGuarded(record))
    clearDurableCounter(key)
```

The counter is keyed by partition and offset (or `event_id` if you can read it without deserializing). The price is a durable write per record in the hot path, and a limit that has to be set high enough that an ordinary deploy restart does not push healthy records over it. The alternative is to accept that a crash-looping poison record stalls its partition until a person notices, which is why the lag alarm described below matters.

```
function handleMessage(message):
  try:
    processBusinessLogic(message)
    ack(message)
  on PermanentFailure as failure:
    // classified permanent: retrying cannot help, so skip the
    retry budget
    deadLetter(message, failure)
  on error as failure:
    // Chapter 6's classifier: anything unclassified is "unknown"
    // and gets the stricter attempt limit, never an escaped
    exception
    limit = attemptLimitFor(classifyFailure(failure))
    if message.attempt_count >= limit:
      deadLetter(message, failure)
    else:
      // nack with backoff on a queue; retry topic on a log
      (Chapter 6)
      retryLater(message)
```

The `deadLetter` step is where the context needed for triage is captured, and the order of its last two lines is the point.

```
function deadLetter(message, failure):
    record = {
        original_message: message.body,
        original_headers: message.headers,
        original_key:      message.key,
        original_time:     message.timestamp,
        source:             message.source_name,
        source_position:    message.position,
        consumer_input:     CONSUMER_INPUT,
        redrive_count:      message.redrive_count,
        error_class:        failure.class,
        error_message:      truncate(failure.message, 1000),
        attempt_count:      message.attempt_count,
        consumer:           CONSUMER_NAME,
        consumer_version:   BUILD_VERSION,
        first_failed_at:    message.first_failed_at,
        failed_at:          now(),
        correlation_id:     message.correlation_id,
        stack_trace_ref:    failure.trace_pointer
    }
    // publish first, ack second: a crash between them yields a
duplicate
    // dead letter, which is recoverable; the reverse order loses the
message
    publish(DLQ_NAME, record)
    ack(message)
```

If `publish` itself fails, `deadLetter` raises and the message is not acknowledged. On a queue-family broker that is enough: the message returns through normal redelivery. On a log-family broker it is not, because an uncommitted offset does not bring the record back; the consumer must seek back to the failed record's position, or pause the partition, and retry the dead-letter publish before it reads further. Either way the goal is the same failure mode: the message stays in the system until it has a safe new home.

What to store with a dead letter

A dead letter with no context is a puzzle. A team that finds a message in a DLQ six weeks later needs to answer four questions: what was it, where did it come from, why did it fail, and is it still safe to process? The record should answer them without opening a source repository.

Field	Why it is there
Original body and headers, byte for byte	Redrive must reproduce the message exactly, including <code>event_id</code>

Field	Why it is there
Source name and position (topic and partition and offset, or queue name)	Locates the message in history, supports ordering decisions
Original key (record key or routing key) and original timestamp	Redrive must publish with the same key, or the message lands on another partition and loses its per-key order
Consumer input (the failing consumer's own queue or redrive topic)	The only place a redrive may publish to
<code>redrive_count</code>	Separate from <code>attempt_count</code> : how many times this message has already come back
<code>error_class</code> and <code>error_message</code>	Groups failures by cause; truncate long messages
Stack trace or a pointer to the log line	Lets an engineer see where it failed without reproducing it
<code>attempt_count</code>	Distinguishes "failed once, permanent" from "failed after many retries"
<code>first_failed_at</code> and <code>failed_at</code>	Shows how long the message has been waiting and when the problem started
Consumer name and version	Ties the failure to a deploy; often the fastest route to the cause
<code>correlation_id</code>	Connects the message to the rest of its business flow

Two cautions apply. First, keep the original payload unmodified. If you wrap it in an envelope of your own, the redrive tool must unwrap it exactly, or `event_id` and `causation_id` will drift and the idempotency guarantees from Chapter 7 are lost. Second, remember that a DLQ is a copy of production data, often kept for a long time. It inherits the access controls and retention obligations of the original, and payloads with personal data need the same care in the DLQ as anywhere else.

Parking lot versus DLQ

The word DLQ gets used for two different things, and separating them makes the operation easier to reason about.

A **parking lot** is a holding place for dead letters that need a human decision. The DLQ is where the system puts a message automatically; the parking lot is where a person puts a message deliberately after looking at

it and deciding that it will not be fixed by a redrive. It is a distinct place with a distinct purpose.

Consider what each one contains. The DLQ holds messages that failed and have not been examined. It should drain: its expected state is empty or near empty, and anything old in it is a symptom. The parking lot holds messages that someone examined and set aside: an order that was cancelled before its event arrived, a payload that needs manual correction, a record that must be reconciled by hand with another team. Those messages are not waiting for a fix to deploy. They are waiting for a decision.

Mixing them produces a predictable failure. If decided-and-parked messages stay in the DLQ, the DLQ's depth never returns to zero, the alert threshold is raised to stop the noise, and the next real poison message arrives unnoticed. Keeping them apart lets the DLQ alarm stay sharp.

A parking lot can be as modest as a database table with a status, an owner, a reason and a review date. It does not need to be a broker destination at all, because nothing is going to consume it automatically. The review date matters most: a parked message with no date for its next look is already forgotten.

Ownership and alarms

A DLQ without an owner is a drain that fills quietly. The first decision, before any tooling, is who is responsible for each one.

My default is that the team that owns the consumer owns the DLQ. They wrote the code that failed, they know what the message should have meant, and they are the ones who can fix and deploy. One DLQ per consumer, or per consumer group and source, makes that ownership obvious. A single shared DLQ for the whole company makes it ambiguous, and ambiguity means nobody looks. Where a platform team provides the tooling, the platform team owns the mechanism and the consuming team owns the contents.

What to alarm on

Alarm on **arrival**, not on depth alone. A single message appearing in a DLQ that is normally empty is worth a notification, because the first dead letter is the cheapest moment to find a bug. A depth threshold alone tells you nothing until the queue is already deep.

Three signals together cover most cases:

- **Any new arrival** in a DLQ that is supposed to be empty. Route this to a channel a human reads during working hours.
- **Arrival rate**. A sudden burst usually means a bad deploy or an incompatible schema change, and justifies paging someone.
- **Age of the oldest message**. This catches the slow failure: the DLQ that receives one message a day, which nobody acts on, until the oldest is a month old and the business context is gone.

I don't know the right paging threshold for your system, and nobody can tell you without knowing the business value of the messages in question. A DLQ attached to `PaymentCaptured` consumers deserves a lower threshold than one attached to `Notifications`, because a lost email is an annoyance and a lost capture is a ledger discrepancy. The thresholds should reflect that difference, and the difference should be written down next to the alarm.

Note: Alarm on consumer lag as well. A consumer stuck on a poison record in a log partition does not necessarily produce dead letters, because it never gives up. Rising lag with no throughput is the signal that the loop described at the start of this chapter is running.

Redrive: getting messages back

Redrive means giving dead letters back to the consumer that failed on them, for reprocessing. The target is that consumer's own input: its retry or redrive topic, or its own queue. It is never the shared topic the message originally came from, because every other consumer of that topic has already processed the event, and republishing there would deliver it to all of them a second time. The original key travels with the message, so it stays on the same partition and keeps its per-key position. The attempt counter

starts again at zero, and the separate `redrive_count` goes up by one, so a message that keeps coming back is visible instead of looking new every time. Redrive is the operation that makes a DLQ valuable and also the one most likely to cause a second incident. Chapter 3 described the engineer who presses a button and redrives 40,000 messages at once. The remedy is to treat redrive as a designed procedure, not as a convenience button.

Fix the cause before the redrive

A message that failed because of a consumer bug will fail again until the bug is fixed and deployed. Redriving before the fix moves the messages in a circle and refills the DLQ with the same entries, now with higher attempt counts and a confusing history. The order of work is: understand the cause, deploy the fix, redrive a small sample, watch it succeed, then redrive the rest.

Redrive must be safe to repeat

A redriven message is a duplicate by definition: the system has seen this `event_id` before, and the consumer may have seen it too. If the original processing partly succeeded before failing, some effects may already exist. Chapter 7 is the reason this is survivable. An idempotent consumer with an inbox treats the redriven message like any other redelivery and applies its effects once.

Two points are worth stating. Redrive must preserve the original `event_id`, since that is the idempotency key. A redrive tool that generates a fresh identifier for each message defeats the inbox, and the effect can be applied twice. And a consumer that is not idempotent should not be the target of a bulk redrive at all; fix that first, or accept that every redriven message is a manual risk assessment.

Redrive in controlled batches

A safe redrive tool has a few properties, none of them exotic:

1. **A selection filter.** Redrive by error class, time window, or `correlation_id`, not only "everything".
2. **A rate limit.** Messages return to the consumer's input at a pace it and its dependencies can absorb. A flood of redriven messages is a self-inflicted retry storm, and the in-flight limits and load shedding in "Re-

tries under overload: backpressure" (Chapter 6) apply to redriven traffic exactly as they do to live traffic.

3. **A pause switch.** An operator can stop the redrive in the middle if failures reappear.
4. **A dry run.** The tool reports what it would move before it moves anything.
5. **An audit record.** Who redrove what, when, and why, kept somewhere durable.

```
function redriveBatch(dlqName, filter, batchSize, maxPerSecond):
    // an operator procedure, not a consumer: run it with a human
    watching
    batch = readFromDlq(dlqName, filter, batchSize)
    for each record in batch:
        if redriveIsPaused():
            return "paused"
        // the failing consumer's own input, never the shared source
        topic;
        // the original key keeps the message on its partition, and
        the
        // original headers and event_id keep the consumer's inbox
        effective
        publish(record.consumer_input, record.original_message,
                key: record.original_key,
                headers: record.original_headers
                + { attempt: 0, redrive_count:
record.redrive_count + 1 })
        // mark this record as redriven, one by one; a crash leaves a
        few
        // already-published records unmarked, which is a duplicate,
        not a loss
        markRedriven(record, operator(), now())
        sleep(1 second / maxPerSecond)
    return "batch complete"
```

Notice the order inside the loop: a record is marked redriven after the publish succeeds, not before. If the tool crashes in the middle, a few messages are redriven and still unmarked in the DLQ, which leads to duplicates, not to loss. A consumer with an inbox absorbs those duplicates; one without it turns a tooling crash into a data problem. The record is marked, not deleted, so the audit trail and `redrive_count` survive; cleanup follows the retention rule in "Preventing the landfill" below.

Redrive and ordering

Redrive breaks ordering unless you design for it. A dead-lettered message was skipped; later messages for the same aggregate were processed in the meantime. When the redriven message arrives, it arrives after them. This is Chapter 6's Inventory example with a longer gap: if `OrderPlaced` is dead-lettered instead of retried, `OrderCancelled` still finds no reservation to release, and the redriven placement, hours later, still creates one that nothing will release. Preserving the key does not repair this: the key puts the message on the right partition, but the partition has moved on.

Whether that matters depends on the consumer. A consumer that is **commutative**, meaning the final state is the same whichever order the events arrive, is unaffected. A consumer that depends on state transitions is not. Chapter 9 covers how to detect and handle out-of-order arrival with version numbers or sequence checks, and a state-based consumer should use them before anyone redrives into it. If it does not, the redrive procedure must include a check that the world has not moved on since the message was dead-lettered.

What a dead letter does to an ordered stream

Ordered streams force the hardest decision in this chapter, because the choice that protects ordering is the one that stops the business.

Consider a log-based topic keyed by `order_id`, where Ledger posts entries. Consumers process each partition in sequence. Message 41 for `order-81724` fails permanently. The consumer has two options:

- **Stop and wait.** Hold the partition at message 41 until someone fixes the cause. Ordering is preserved exactly. Every other order that hashes to that partition also stops, even though their messages are fine. The blast radius is the whole partition.
- **Dead-letter and continue.** Park message 41, process 42 and onward. The partition flows. But 42 may have depended on 41, and the consumer has then processed it against a state that does not include 41.

Neither choice is free. Stopping trades availability for correctness; continuing trades correctness for availability. There is no setting that gives you both, and it is better to choose on purpose per consumer than to inherit

whatever the framework default happens to be.

My rule of thumb has three parts:

1. **If the consumer is order-insensitive or commutative**, dead-letter and continue. Notifications is the typical case: a failed email on one order has no bearing on the next.
2. **If later messages for the same key depend on the failed one**, do not continue for that key. Dead-letter the failed message, then also divert the following messages for the same `partition_key` to a holding area until the first is resolved. Other keys continue. This keeps the blast radius at one aggregate instead of one partition, at the price of an extra mechanism: per-key tracking of "blocked" keys.
3. **If the consumer is a financial ledger, stop**. The cost of processing out of order is a record that is wrong in a way that is hard to detect and expensive to repair, and a paused partition is loud and obvious. Correctness over speed was the stated rule for Ledger, and here it applies in its plainest form.

The per-key variant in the second rule deserves a sketch, since it is the most useful and the least common. It assumes a small `key_block` table with one row per blocked key (`partition_key`, `blocked_event_id`, `state`), and that one consumer at a time handles a given key's live traffic, which is what a partition or a message group gives you.

```

function handleOrderedMessage(message):
    key = message.partition_key
    parked = false
    transaction:
        // select ... for update; none if the key is open
        block = lockBlockRow(key)
        if block exists and message.event_id !=
block.blocked_event_id:
            // park under the same lock that resolveBlockedKey takes,
so a
            // message cannot be parked after the block was cleared

            // held messages are unique on event_id; a repeat is
ignored
            appendToHeldMessages(key, message)
            parked = true
    if parked:
        ack(message)
        return
    try:
        // the inbox absorbs a repeat
        processBusinessLogic(message)
        // this was the redriven message
        if block exists:
            // durable: a crash from here on is recoverable
            markResolving(key)
        ack(message)
        if block exists:
            resolveBlockedKey(key)
    on PermanentFailure as failure:
        transaction:
            // state = "blocked"
            markBlocked(key, message.event_id)
            deadLetterRecord(message, failure)
        // blocked key and dead letter commit together
        ack(message)

```

The release is a separate procedure, and a periodic sweep makes it restartable.

```

function resolveBlockedKey(key):
  loop:
    transaction:
      block = lockBlockRow(key)
      if block is none or block.state != "resolving":
        // another resolver cleared or re-blocked it
        return
      next = oldestHeldMessage(key)
      if next is none:
        // emptiness check and clear under the lock
        clearBlocked(key)
        return
    try:
      // the inbox absorbs a repeat after a crash
      processBusinessLogic(next)
      transaction:
        removeFromHeld(key, next)
    on PermanentFailure as failure:
      transaction:
        block = lockBlockRow(key)
        if block is none or block.state != "resolving":
          // the key moved on while next was in flight
          return
        // state = "blocked", anchored on next
        markBlocked(key, next.event_id)
        deadLetterRecord(next, failure)
        removeFromHeld(key, next)
      return

function sweepResolvingKeys():
  // runs at worker start and on a timer
  for each key in keysInState("resolving"):
    resolveBlockedKey(key)

```

Three details in this sketch decide whether it works. The first is the redrive itself. The redriven message is the very one the key is blocked on, so it has to bypass the block: the check compares its `event_id` with the blocked one. Without that exception the redrive is parked behind the messages it was supposed to precede, and the key never unblocks.

The second is the race with live traffic. While `resolveBlockedKey` drains the held messages, new messages for the same key keep arriving. They must still be held, not processed, until the held set is empty. That only holds if the decision to park and the decision to clear are serialized: both take the key's row lock, and the park is written in the same transaction as the check. If the park were a separate step, a live message could see a

blocked key, then find the held set already emptied and cleared, and be parked behind a block that no longer exists. Nobody would drain it, and later messages would be processed ahead of it.

The third is a crash in the middle of the release. Once the redriven message is processed, the `resolving` state is written before the ack, so the work to do is recorded in the database and not only in a message that will never come back. The sweep picks up any key left in that state and continues; a message processed shortly before the crash is absorbed by the inbox. If a held message fails permanently during the release, the key blocks again, anchored on that message, with its own dead letter, and the rest of the held set waits for its redrive. The per-key block is a loop, not a one-time event.

The sketch is not complete. A sweep can overlap a live release, so two resolvers may work on the same key. The state check under the lock keeps either of them from overtaking a re-block: once one resolver anchors the block on a failed message, the other finds the key no longer `resolving` and stops. What remains is duplicate work on the same held message, which the inbox absorbs; a lease on the `resolving` state would save that work, not correctness. It also puts no bound on the held set, and it leaves out how an operator discards a held message that is no longer meaningful. Treat it as the shape of the mechanism, not a finished one.

This is real machinery with real failure modes of its own, such as the held-messages store growing without bound, so I would build it only for streams where per-key order matters and the dead-letter rate is low enough that the held set stays small. It is also a mechanism that does not know whether the held messages are still meaningful once the failed one is resolved. When the state transitions themselves have to be validated, not merely sequenced, the version buffer in Chapter 9 is the stronger tool: it holds an event until its predecessor has actually been applied, by version number, and it works without a dead-letter record to anchor it.

Preventing the landfill

A DLQ turns into a landfill when messages enter faster than anyone decides what they mean. The symptom is familiar: thousands of entries, many different errors, nobody sure which matter, and an unwritten rule

that you do not touch it. At that point the DLQ is no longer a safety mechanism; it is a place where lost data is stored with a polite name.

Landfills have causes, and each has a counter.

- **No owner.** The counter is the ownership rule above, written down.
- **No triage cadence.** Dead letters should be reviewed on a schedule, not only when an alarm fires. For a low-volume DLQ, a weekly review is a reasonable starting point.
- **Retention that outlives usefulness, or that expires before triage.** A dead letter for a stock reservation that expired last month cannot be redriven meaningfully. Set a retention period based on how long the message stays valid for the business, and decide in advance what happens at expiry: archive, discard with an audit entry, or escalate. The opposite error is as common: a retention shorter than the triage cadence deletes messages before anyone has looked. If the DLQ is a broker queue whose retention is shorter than your review interval, the broker removes the evidence first. Set the DLQ retention above the longest gap between reviews, and read your platform's documentation on what the retention clock starts from. If it starts at the original enqueue time and not at the moment of dead-lettering, a message that waited a long time before failing arrives already close to expiry. That is one more reason the record stores the original timestamp.
- **Everything is dead-lettered.** If routine business outcomes, such as a validation rejection that is a normal part of the flow, are sent to the DLQ, the real failures drown. A rejected order is a business event with its own handling, not a dead letter.
- **Transient failures dead-lettered too early.** If the retry policy gives up after seconds during a brief outage, every message in flight during that outage lands in the DLQ. The remedy is in Chapter 6: a retry policy whose attempts and deadline are sized to the outage you expect to survive.

I would also give every dead letter a disposition. Triage ends with one of a short list of outcomes: redriven after a fix, corrected and redriven, parked for a manual decision, or discarded with a recorded reason. "Still sitting there" is not a disposition.

Trade-offs and costs

A DLQ moves a failure from a place where it is loud, the blocked consumer, to a place where it is quiet. That is both the benefit and the hazard. The stream flows, and the failed message can be forgotten. Everything in this chapter that is not the DLQ itself, the alarms, the owner, the review cadence and the retention, exists to counteract that quietness.

Specific costs to weigh:

- **Operational load.** Someone has to triage, decide and redrive. A DLQ that nobody has time for is worse than none, because it hides the problem behind the appearance of a mechanism.
- **Data governance.** Dead letters hold production payloads, sometimes with personal data, outside the usual lifecycle. They need access controls, retention rules and deletion paths.
- **Ordering risk.** Dead-lettering a message from an ordered stream skips it. Unless you build per-key blocking or stop the partition, later events are applied without it. Per-key blocking itself adds a held-messages store, a redrive bypass, a release loop that must race safely with live traffic, and a sweep that restarts an interrupted release.
- **Stored identity.** Safe redrive needs the original key, partition and consumer input on every record, and a `redrive_count` beside the attempt count. That is more to store and more to keep consistent, and without it redrive can only be a guess.
- **Tooling investment.** A safe redrive tool with filtering, rate limits, per-record disposition and an audit trail is a small project. Buying or building it is part of the cost of having a DLQ.

When not to use one: if a message can be regenerated or its loss is harmless, a DLQ adds more process than value. A cache-warming event or a purely cosmetic update can be dropped with a metric counting the drops. And where the right response to failure is to halt, as in the ledger example, a DLQ that lets the stream continue works against you. In that case, stop the consumer, alert loudly, and fix the cause.

There is also a limit to what the pattern can do. A DLQ holds messages that failed visibly. It does nothing for messages that were processed incorrectly without raising an error, which is the territory of Chapter 13 and its

reconciliation work.

Warning: Never acknowledge the original message before the dead letter is durably stored. If the DLQ publish fails after the ack, the message exists nowhere, and no alarm will tell you.

Common mistakes

- **Dead-lettering with no context.** It is tempting because moving the raw body is the shortest code. The fix is the metadata record described above, including the exact original headers, so that redrive and diagnosis are both possible.
- **Retrying permanent failures.** A parse error does not improve with time, yet a blanket retry policy treats it like a timeout. Classify failures, and send permanent ones straight to the DLQ instead of spending the retry budget.
- **Redriving before fixing the cause.** The pressure to "get the messages moving" is real and the result is the same messages back in the DLQ. Deploy the fix, redrive a sample, then the rest.
- **Redriving everything at once.** A flood of old messages competes with live traffic and can start a new incident. Rate-limit, batch and keep a pause switch.
- **Alarming on depth only.** It feels like the obvious metric, but it fires late. Alarm on first arrival, on rate and on the age of the oldest message.
- **One shared DLQ for everything.** It is cheap to create and impossible to own. Give each consumer its own, and assign the owner by name.
- **Letting parked and unexamined messages mix.** The DLQ never drains, the alert is tuned down, and real poison goes unnoticed. Keep a separate parking lot with owners and review dates.
- **Ignoring order on redrive.** A redriven message arrives after later events for the same aggregate. Use version checks from Chapter 9, or confirm the state has not moved on.
- **Redriving to the shared topic.** It looks like "putting the message back where it came from", and it delivers the event a second time to every

other consumer of that topic. Redrive to the failing consumer's own input, with the original key.

- **Waiting forever on a crash-looping record.** On a log broker a consumer that never gives up produces no dead letters and no alarm except lag. Count attempts durably by partition and offset, so that a restart does not reset the count.

Key takeaways

- A poison message fails every time it is processed, so retrying it forever turns one bad message into an outage for everything queued behind it.
- Dropping a failed message and retrying it forever are both ways of hiding a failure; a dead-letter queue sets the message aside with context so the rest of the stream can continue.
- A dead letter should carry the original body and headers unchanged, its source position, the error, the attempt count, and the consumer version, because the person reading it weeks later has only that record.
- Publish to the DLQ before acknowledging the original, so a crash produces a recoverable duplicate and never a lost message.
- Every DLQ needs a named owning team, an alarm on first arrival, rate and age, and a regular triage cadence, or it becomes a landfill.
- A redrive is only safe when the cause is fixed, the original `event_id` and key are preserved, the message goes to the failing consumer's own input and never a shared topic, the consumer is idempotent, and messages return in rate-limited batches with a pause switch.
- On an ordered stream, dead-lettering one message forces a choice between stopping the partition and continuing past a gap; choose per consumer on purpose, stop for ledgers, and block per key only where the machinery is worth its cost.

Dead-lettering and redrive both lean on one question this chapter kept deferring: what does it mean for messages to arrive in the right order, and what can you build when they do not? Chapter 9 takes that question on directly.

Chapter 9 — Ordering

Imagine a Thursday evening at Parcelly. A buyer places an order, then changes their mind within seconds and cancels it. Orders emits `OrderPlaced`, then `OrderConfirmed`, then `OrderCancelled`, all for `order-81724`, and all in the right order. The relay publishes them in that order too.

The topic that feeds Shipping was created with a default setting: messages are spread across six partitions without regard to their content. `OrderConfirmed` lands on partition 2 and `OrderCancelled` on partition 5. Shipping runs one consumer per partition. The consumer on partition 5 is idle and fast, so it reads the cancellation first and finds a shipment that does not exist yet. It shrugs and acknowledges. A moment later the consumer on partition 2, which was busy with a slow batch, reads the confirmation and schedules a shipment for an order that no longer exists.

Nothing crashed, no message was lost, and no message was duplicated. Every component did what it was configured to do. A carrier will arrive on Monday to collect a parcel that nobody ordered. How much order does a system like this need, where does it get lost, and what can you build when you cannot have it?

The problem, and why it is hard

Order is not one property that a system has or lacks. It is a chain, and every link can break it independently. The relay can publish rows in a different sequence than the one the database committed (Chapter 4 and Chapter 5 cover that link). The broker can spread related messages across partitions or queues. Consumers can process in parallel. A retry can put a message behind its successors (Chapter 6), and a dead letter can skip it entirely (Chapter 8).

Link	How order is lost	Where to look
Producer and relay	Parallel relays, concurrent commits, ids that do not match commit order	Chapters 4 and 5
Broker	Related messages on different partitions or queues	This chapter
Consumer	Several workers handling messages from one partition or queue	This chapter
Failure handling	Retry topics, nacks, dead letters, redrive	Chapters 6 and 8
Topology change	Partition count changes, consumer rebalances	This chapter

The obvious response is to demand a global order: every event in the system, in one sequence, seen by every consumer. It sounds like the safe default. It is expensive and, for most systems, unnecessary.

Why global order is expensive

A single total order means a single sequence point. On a log-based broker, that is one partition, which is read by at most one consumer in a consumer group at a time. Your throughput ceiling becomes the speed of one writer and one reader. Adding capacity does nothing, because capacity comes from running things in parallel and parallelism is what a total order forbids.

A system that spans several services makes this worse. Orders, Payments and Inventory each have their own database and their own outbox. A global order across all of them would need a coordinator that stamps every event, and that coordinator would sit on the write path of every service. You would be buying back the coupling that the architecture was designed to remove.

Why global order is mostly unnecessary

Ask what the business actually needs. Parcelly does not care whether `PaymentCaptured` for `order-81724` came before `OrderPlaced` for `order-90311`. Those orders do not influence each other. What matters is that for one order, `OrderPlaced` is seen before `OrderConfirmed`, and `OrderConfirmed` before `OrderCancelled`.

That is **per-aggregate ordering**. An aggregate is the consistency boundary whose identifier serves as the partition key. If every event about one aggregate is seen in the order it was produced, and events about different aggregates are free to interleave, most state machines work correctly. This is the order you can afford, and in my view it is the order you should design around.

There is one boundary even per-aggregate order does not cross. Events for the same order, produced by different services and carried on different topics, have no order relative to each other. `PaymentAuthorized` and `StockReserved` may arrive at `Orders` in either sequence, which is exactly why the order flow used throughout this book requests capture only after both have arrived. Do not treat a shared partition key as a promise across topics.

Ordering per aggregate

The mechanism is simple to state and demanding to keep: choose a key, send everything with the same key down one path, and process that path one message at a time.

The partition key

On a log-based broker, the producer hashes the `partition_key` to pick a partition. Every message with the same key lands on the same partition, and a partition is an ordered sequence. The envelope in this book carries `partition_key` for this reason, and the outbox stores it so that the relay can set it without asking the producing service (Chapter 4).

The guarantee has a condition: **same key, same partitioner, same partition count**. The partitioner is the hash function plus the way the key is serialized to bytes before hashing. If two producers use different ones, one `order_id` goes to two partitions and per-key order is gone, with no error anywhere. This is the most concrete polyglot failure in ordering. Client libraries in different languages do not all share one default hash function, so a service written in one language and a relay written in another can send the same `order_id` to different partitions. Two defences work. Pin the partitioner explicitly in every producer, using the same algorithm and the same key encoding. Or take the decision out of the producers: have

the relay compute the partition itself from the stored `partition_key`, so that a single piece of code owns the mapping. The second is natural when every event already passes through the outbox relay from Chapter 5.

Queue-based brokers are different. Competing consumers on one queue give you no ordering guarantee once there is more than one of them, because the broker hands the next message to whichever consumer is free. Some queue-based brokers offer a grouping feature that serializes messages sharing a group identifier, for example message groups on Amazon SQS FIFO queues, and others offer a mode where a single consumer is active on a queue at a time. These features work, and each costs parallelism in a different way. Check what your broker promises about redelivery and visibility timeouts under that feature, because the guarantee often has fine print. Say which family you rely on, and what that family really promises.

Choosing the key

The key is the aggregate identifier, and the choice is a statement about where consistency lives. For Parcelly's order lifecycle, the key is `order_id`. Every event that moves an order through its states carries it, which gives Shipping, Orders and Ledger a consistent per-order sequence.

A few questions help when the answer is not obvious:

- Which entity's state does the consumer change? Key by that entity.
- What is the smallest unit within which two events can conflict? That is the aggregate.
- Is the same stream consumed by several services with different aggregates? Then either the producer picks the finest common one, or each consumer needs its own view.

Inventory shows the tension. Stock levels belong to a SKU, so Inventory's own commands naturally key by `sku`. The events flowing from Orders to Inventory key by `order_id`. A reservation event carries both identifiers, and Inventory re-keys internally if it needs per-SKU order. Choosing one key for every topic in the system is tempting, and it wastes the point of having aggregates.

Hot partitions

A partition key spreads load evenly only if keys are evenly busy. They are not. Imagine a flash sale at Parcelly in which one seller's single product receives most of a day's orders in an hour. If Inventory keys its own topic by `sku`, that one key produces most of the traffic, all of it lands on one partition, and one consumer is asked to do the work of the whole group while the others idle.

Adding partitions does not help. A single key always maps to a single partition, and adding partitions moves other keys around it. There are four honest responses:

- **Make the aggregate smaller.** If the business rule allows, key by something finer, such as a reservation id rather than the SKU. Ordering within the SKU is given up in exchange for spread, and this is only acceptable if no rule requires per-SKU order.
- **Spend on the consumer, not on the partitions.** Make each message cheap to handle, so one consumer can keep up with the hot key.
- **Isolate.** Route known hot keys to a dedicated topic or partition, so they do not starve ordinary traffic that shares the partition.
- **Accept relaxed ordering for that stream.** Add a random suffix to the key to spread the load, and rely on version checks (later in this chapter) to repair order at the consumer. This is the most flexible and the most work. Note what salting does and does not buy: it spreads load on the broker, but the version check serializes the key again on the aggregate's row lock in the consumer's database. Throughput improves only if the expensive work happens before the serialized step, or if the operations commute.

I would measure key skew before choosing. A key distribution that looks even in a quiet week can be badly lopsided on a busy day, and the first sign is usually consumer lag on exactly one partition (Chapter 13).

Parallelism and order pull in opposite directions

A consumer group scales by giving each partition to one consumer, so the number of partitions is the ceiling on parallelism. If Shipping has six partitions, six consumers can work in parallel. A seventh would sit idle.

Teams often try to raise throughput inside one consumer by handing messages to a thread pool. This looks harmless and breaks order. Two messages for the same order, read one after the other, are handed to two workers, and the slower worker finishes second. The partition delivered them in order and the processing did not preserve it.

The way to get both is to parallelize across keys and serialize within a key. Inside the consumer, hash the key to a fixed lane, and give each lane exactly one worker.

```
function pollLoop():
  while running:
    // bounded lanes: a slow lane pauses intake instead of filling
    memory
    if anyLaneAbove(HIGH_WATER):
      pause(assignedPartitions)
    else if allLanesBelow(LOW_WATER):
      resume(assignedPartitions)
    for message in poll(maxRecords: POLL_LIMIT):
      // the same key always maps to the same lane, so per-key
      order survives
      lanes[hash(message.partition_key) mod
      LANE_COUNT].enqueue(message)
      // commits are per partition and happen here, on the polling
      thread
      for partition in assignedPartitions:
        commit(partition, contiguousDoneOffset(partition) + 1)

function runLane(lane):
  while running:
    message = lane.dequeue()
    try:
      handleMessage(message)
    on error as failure:
      // in place or aside, as decided in Chapter 6
      decideInPlaceOrAside(message)
    markDone(message.partition, message.offset)
```

Three details carry the design. The first is the offset. Commits are per partition, and `contiguousDoneOffset` is the highest offset below which every message of that partition is done. When lanes finish at different speeds, committing the offset of the fastest lane would, after a crash, skip messages that a slower lane had not finished. The commit happens in the poll loop because many log clients are not safe to call from several threads. The price of in-consumer parallelism is that bookkeeping, and the duplicates after a crash, which an inbox from Chapter 7 handles.

The second is the bound. Without one, a slow lane keeps the commit point still while the poll loop keeps filling memory, and the gap between the committed offset and the work in flight is exactly the number of messages redelivered after a crash. Pausing the partitions when a lane passes its high-water mark turns overload into lag, which is visible and safe, instead of memory growth, which is neither. The mechanism, and the choice between pausing and shedding, is the one described in "Retries under overload: backpressure" in Chapter 6.

The third is failure. If `handleMessage` fails and the lane retries in place, every key that shares the lane waits behind it and the committed offset stops advancing. Sending the message aside lets the lane move on and re-orders that key. This is the same decision Chapter 6 made for whole partitions, now made per lane, and the answer depends on the consumer in the same way.

On a queue-based broker the same idea appears one level up. If you need per-key order across competing consumers, you either use the broker's grouping feature or accept that the queue will deliver out of order and handle it in the consumer. Do not assume that a queue with several consumers preserves order because the messages went in orderly. It does not.

Detecting out-of-order arrival with versions

Sometimes the broker cannot give you the order you need, because of a retry topic, a redrive, parallel consumers, or a change of partition count. In those cases the consumer has to notice for itself. For that, the events must carry something a consumer can compare.

Timestamps are the wrong thing to compare. Chapter 3 explained that `occurred_at` reflects one machine's clock, and that using it to decide which update wins lets a service with a fast clock win every conflict it should lose. Use `occurred_at` for humans and analytics.

What works is a **version number** (also called a sequence number): a counter that the aggregate increments on every change, written in the same transaction as the state change and carried with the event. Because only the owning service changes the aggregate, and it does so inside a transaction, the versions form a gapless sequence per aggregate: 1, 2, 3. Gapless needs a rule in the other direction as well: **every version incre-**

ment produces exactly one published event. An internal update that bumps the version without publishing anything leaves a permanent gap, and every consumer will wait for an event that does not exist.

The standard envelope has no field for it, and I would not change the envelope for one chapter. I would carry it in the payload, as `order_version` for Orders events. It is the aggregate version (Chapter 4), the same number the outbox row stores. I leave the deeper discussion of envelope design to Chapter 11. The counter belongs to one producer and one aggregate. A consumer compares `order_version` values only among Orders events for the same order, and never against an event from Payments or Inventory, which would need a counter of its own. A dependency between services is not solved with a shared sequence. It is solved with a state or precondition check, as the capture step does when it waits for both `PaymentAuthorized` and `StockReserved`. The outbox row gets it for free, since the service writes the new version, the state change and the outbox row in one transaction.

The consumer's decision

A consumer that keeps a view of an aggregate also keeps the last version it applied. For each arriving event it compares versions and takes one of three paths.

Arriving version, compared to last applied	Meaning	Action
Equal or lower	Duplicate or stale	Ignore, count it, ack
Exactly one higher	The next event	Apply, record the new version
More than one higher	A gap: something is missing or late	Buffer the event, do not apply

```

function handleOrderEvent(message):
    key = message.partition_key
    transaction:
        view = loadOrderView(key)
        // loadOrderView locks the row; a concurrent worker waits
here.
        // For the first event there is no row yet: last_version
defaults
        // to 0 and the key's unique constraint settles a concurrent
insert.
        version = message.payload.order_version
        if version <= view.last_version:
            countIgnored(message)
            // a lower version was already applied in sequence: a
duplicate
            else if version == view.last_version + 1:
                applyToView(view, message)
                view.last_version = version
                saveOrderView(view)
                // applying may unblock held events (drain is defined
below)
                drainHeldEvents(key)
            else:
                // insert-if-absent: a redelivered held version must not
fail
                holdEvent(key, version, message)
                // the commit covers the view and the held event together
                ack(message)

```

Two properties make this safe. The version check and the state update commit atomically, so two workers handling the same key cannot both apply version 4. And the check is idempotent: a duplicate is a stale version like any other and gets ignored, so it complements the inbox from Chapter 7 rather than replacing it. `holdEvent` has to be idempotent as well. A duplicate delivery of an already held version would otherwise hit the uniqueness constraint on the buffer, fail the transaction, leave the message unacknowledged and turn into a poison loop.

Two preconditions are often overlooked. The first is the start of the sequence. A consumer that begins mid-stream, such as a new service or a replay from a later offset, sees version 5 first and would wait forever for versions 1 to 4. Seed its view from a snapshot or from the owning service's API, together with the version that snapshot reflects, and start comparing from there.

The second is that gap detection only works if the consumer sees every version. A consumer that subscribes to a subset of an aggregate's events, say only `OrderCancelled`, will see versions 3 and 7 and conclude, wrongly, that 4 to 6 are missing. For such consumers the rule weakens to "never apply anything not newer than what I have", which still protects against stale overwrites but cannot detect gaps.

Last-writer-wins traps

The simplest conflict policy is that the last write wins. It is simple and it hides three different traps, depending on what "last" means.

Last to arrive. The view is overwritten by whichever message the consumer handled most recently. A redelivered `OrderPlaced` that arrives after `OrderConfirmed` sets the status back to `PLACED`, which is the failure Chapter 7 pointed at. Arrival order is the one thing the broker topology may not promise.

Last by timestamp. Better in appearance, worse in practice. Clock skew between services means that the winning event may be the one with the faster clock. Two events written within milliseconds on different machines have a meaningless timestamp order.

Last by version. This works, and it has its own limit. Dropping a stale event is only correct if the newer event already contains everything the stale one said. That is true for events that carry state ("the order is now `CANCELLED`"). It is false for events that carry changes, such as "add 4,990 to the balance". Ignoring a stale change event loses the change.

One clarification keeps this from contradicting the table above. With gap-less versions applied strictly in sequence, a lower version is always a duplicate, because it was already applied; ignoring it is correct even for deltas. The delta trap appears when a consumer jumps to the newest version and skips a gap, since the skipped changes are then gone.

This distinction decides what you do on a stale or out-of-order version:

- **State-carrying events** (the full state, or a state that overwrites the previous one): ignoring the stale event is correct, because the later one supersedes it.

- **Delta or transition events** (increments, ledger entries, anything that does not supersede): never drop them. Buffer and reorder them, or process them in any order if the operation is commutative. Ledger in particular must not discard an entry because it is older.

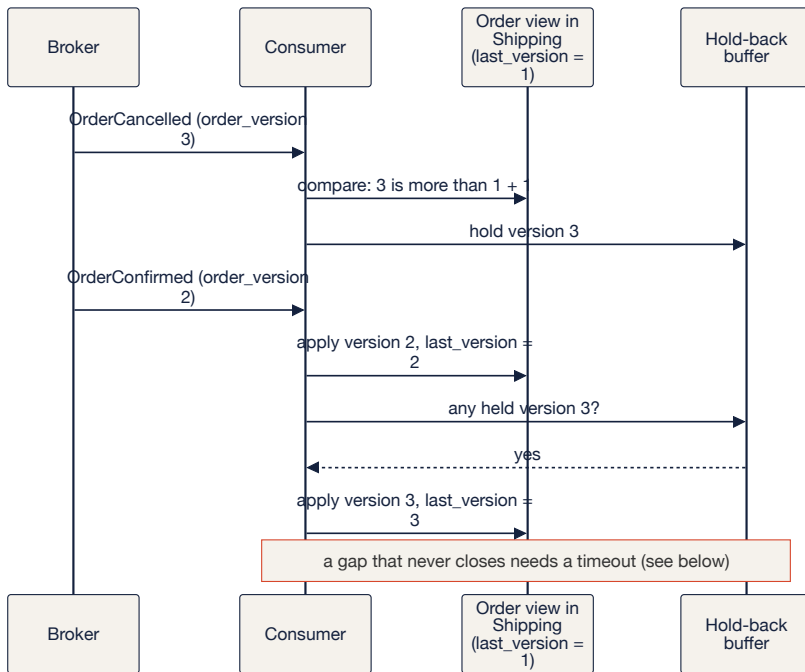
A related trap is the deleted aggregate. If a consumer deletes its row when it sees `OrderCancelled`, a late `OrderPlaced` finds no row, passes the version check with nothing to compare against, and creates a new order. Resurrection is the characteristic bug of ignoring order around deletes. Keep a tombstone: the key, the last version, and a deleted marker, for at least as long as stale events can still arrive. Chapter 8 showed the same resurrection when a redriven `OrderPlaced` arrives after a cancellation, and a tombstone with a version check is what stops it.

Finally, ignoring is a decision that deserves a trace. A stale event dropped silently is a bug that nobody can see. Count ignored events per consumer and keep the key and versions in a log line, so that a spike in the count shows up before a customer notices (Chapter 13).

Buffering and reordering

When the consumer sees a gap, it has two ways forward. It can apply the event anyway and risk a wrong state, or it can hold the event back until the missing one arrives. The second option is a **hold-back buffer**.

The sequence diagram below shows Shipping receiving two events from Orders for `order-81724`, version 3 before version 2, as in the opening scenario. Both come from one producer and one aggregate, which is what makes comparing their versions meaningful. Shipping acts only on `OrderConfirmed`, but for this to work it reads the whole Orders stream for the order and records the version of every event, including the types it otherwise ignores. That is what makes the view's `last_version = 1` real, and what makes gap detection possible at all. Watch where version 3 waits, and what releases it.



The release step is a loop. After applying any event, the consumer checks whether the buffer holds the next version, applies it, and repeats until the buffer has no successor.

```

function drainHeldEvents(key):
    // runs inside the same transaction that applied the triggering
    event
    while true:
        view = loadOrderView(key)
        next = findHeldEvent(key, view.last_version + 1)
        if next is null:
            return
        applyToView(view, next.message)
        view.last_version = next.version
        saveOrderView(view)
        deleteHeldEvent(key, next.version)
  
```

The buffer itself is a table keyed by the aggregate and the version, so it survives a restart. An in-memory buffer disappears with the process, and the broker has already considered those messages acknowledged.

Cleanup needs one more rule. Whenever `last_version` jumps, for example after replacing the view with fetched state, held rows with `version <= last_version` are obsolete, and the consumer deletes them in the same

transaction. Otherwise they sit in the table forever, and the drain loop never reaches them.

The gap that never closes

Buffering is optimistic: it assumes the missing event is merely late. Sometimes it is lost, dead-lettered, or was never sent to this consumer. Without a bound, the aggregate stays blocked forever, and the held set grows.

Every buffer needs a timeout and a decision about what happens when it expires. There are three reasonable answers, and I would pick by the cost of being wrong:

- **Fetch the missing state.** Ask the owning service for the current state through an API and replace the view. This works when the producer exposes such an API, and it makes the buffer a fast path with a safety net.
- **Alert and wait.** A human looks. Right for Ledger, where a wrong entry costs more than a slow one.
- **Mark the view suspect and let reconciliation repair it.** Right for read models that can tolerate staleness (Chapter 13).

Chapter 8's per-key blocking is the version-free form of this mechanism. There, a dead-letter marker blocks the key, later events wait in a held store, and a special bypass lets the redriven event through to release them. Here a dead-lettered event creates a gap, later events wait in the buffer, and the redriven event releases them by itself, because it arrives as `last_version + 1` and the drain loop does the rest. The versions replace both the marker and the bypass, which is why this is the stronger design where the producer can supply them.

The costs are real: a buffer table to size and clean, a timeout policy to tune, and a new class of stuck aggregate that someone needs to be paged about. If your events carry full state and your consumers can safely apply "newest wins", the buffer is unnecessary and I would leave it out.

Retries, dead letters and order

Chapter 6 and Chapter 8 each ended with an ordering question that they left for here. The short form is that every failure-handling mechanism moves a message in time, and that is a reordering.

- **Retry in place** on a log partition blocks the successors, so order is preserved and the whole partition pays in latency.
- **A retry topic or delay queue** lets successors go ahead, so the failed message returns later than events it preceded.
- **A visibility timeout** on a queue-based broker returns a message to the queue after the timeout, where it competes with messages that were sent after it.
- **A dead letter** skips the message, and a **redrive** brings it back after everything that followed.

The version check is what makes the second to fourth options survivable. Take the example Chapter 6 gave: Inventory receives `OrderPlaced` and then `OrderCancelled` for one order from Orders, and `OrderPlaced` is moved to a retry topic after a transient failure. With the aggregate version from Orders, `OrderPlaced` is version 1 and `OrderCancelled` is version 2. The cancellation arrives first, and Inventory has no view for the order yet, so `last_version` defaults to 0 and 2 is more than $0 + 1$. The cancellation is held, not applied against a reservation that does not exist. When the re-tried `OrderPlaced` arrives as version 1, it is applied, and the drain loop releases the cancellation. The stray reservation Chapter 6 described does not appear.

This only works for consumers that keep versions and a buffer. A consumer that does not must choose, as Chapter 6 argued, between blocking the partition and tolerating reordering. And Chapter 8's advice for ledgers still stands: for a consumer where a late or missing entry means a wrong financial record, stop and wait. A paused consumer is loud, and a Ledger that quietly applies entries out of order is not.

Warning: A retry topic combined with a consumer that has no version check reorders events silently. Nothing fails, no alert fires, and the state is wrong. If you route failures aside, make the version check part of the same change.

Changing the number of partitions

The partition for a key is a function of the key and the number of partitions, usually a hash modulo the count. Change the count, and the function changes.

Suppose, as an illustration, that a key hashes to 17. With four partitions it maps to partition 1, and with six partitions it maps to partition 5. A hash of 20 moves from partition 0 to partition 2, while 14 and 15 stay where they were. Most keys move. Under modulo hashing, a key keeps its partition going from four to six only when its hash modulo 12 is 0, 1, 2 or 3, so about two thirds of keys land somewhere new. The failure is not rare: it hits every key that has messages in flight during the change.

The danger is the overlap. Messages for a moving key written before the change sit on the old partition, still unconsumed. Messages written after the change go to the new partition. Two consumers now process the same key at the same time, with no order between them. If a consumer also keeps local state per key, such as a cache or an aggregation window, that state is on the wrong instance.

Ways to deal with it, in rough order of preference:

1. **Provision enough partitions at the start.** Over-provision against the expected peak parallelism, because the number is cheap to hold and expensive to change. Many log-based brokers allow adding partitions but not removing them, so a mistake in one direction is permanent.
2. **Drain before the change.** Stop producers or pause the stream, let consumers reach the end of the old partitions, then change the count. Order is preserved at the cost of a window of unavailability.
3. **Migrate to a new topic.** Create a topic with the right count, run a cutover in which consumers finish the old topic before they begin the new one, and retire the old one. It is the safest and the most work.
4. **Rely on version checks.** If every consumer detects and buffers out-of-order events, the transition is survivable without a pause, and the transition itself is also a good test of whether the check works.

Rebalances and revocation

Consumer group rebalances have a similar shape at a smaller scale, and they happen on every deploy. When a partition moves to a different consumer, in-flight work on the old one may overlap with the first messages on the new one. What the consumer does on revocation decides how large that overlap is.

On revocation, the consumer stops dispatching new messages from the revoked partitions, lets the lanes finish what they hold or abandons it, and commits the contiguous offset before giving the partition up. Whether a rebalance revokes every partition or only the ones that move depends on the assignment strategy of your client, and an all-at-once strategy stalls the whole group for the duration; check which one you run. A consumer that is slow to notice it lost a partition keeps processing as a zombie while the new owner does the same work, so expect some overlap whatever you do.

The inbox and the version check keep that overlap harmless only for effects inside the consumer's own database transaction. They do nothing for an email or a carrier API call made by both consumers. Those external side effects need their own idempotency key, as Chapter 7 describes, because the inbox protects local state and nothing else.

What each Parcelly consumer needs

Not every consumer needs the same ordering, and treating them alike is a source of both bugs and waste. A table, drawn for Parcelly, shows how I would decide.

Consumer	Ordering need	Mechanism	If a message is missing
Notifications	None; each message is an independent fact	Any order, no key needed	Dead-letter and continue
Orders status view	Per order, state-carrying	Key by <code>order_id</code> , ignore stale versions	Fetch current state
Inventory reservations	Per order, transitions	Key by <code>order_id</code> , versions plus hold-back buffer	Hold the key, alert

Consumer	Ordering need	Mechanism	If a message is missing
Shipping	Per order, <code>OrderConfirmed</code> before <code>OrderCancelled</code>	Key by <code>order_id</code> , versions plus hold-back buffer	Hold the key, alert
Ledger	Per order, no gaps, no drops	Key by <code>order_id</code> , stop on gap in the Payments-owned per-order sequence	Stop and wait for a human

The Ledger row names its counter on purpose. Ledger consumes Payments events, so it compares a sequence that Payments increments per order, not `order_version`, which belongs to Orders. A counter is only comparable among events from the one service that owns it.

The first row is the cheapest and deserves more credit than it gets. The best way to handle ordering is to design the consumer so that it does not matter. An operation that is commutative, such as adding an item to a set or taking the maximum of two values, gives the same result in any order. A consumer built from such operations needs no key, no buffer and no version. When you can reshape a handler that way, I would do it before reaching for machinery.

Trade-offs and costs

You are trading parallelism for order, and the exchange rate is the aggregate. A coarser key gives stronger ordering and less spread. A finer key gives spread and weaker ordering. There is no key that gives both, and the decision belongs to whoever knows the business rules.

Hot keys are a design cost, not an operations accident. A partition key chosen from a business identifier inherits the skew of the business. A flash sale reveals it. You can mitigate, and you cannot remove it without giving up order for that key.

Versions require discipline from producers. The aggregate must increment the version in the same transaction as the state change, for every change, from every code path. A single path that forgets leaves a gap that consumers will wait on forever. The reverse also holds: every increment must publish exactly one event, or the sequence has a gap that no con-

sumer can close. Backfills, admin tools and data migrations are where this fails, and a consumer that starts mid-stream needs a seeded view before it can compare anything.

Buffers add state and a new failure. A hold-back table needs sizing, cleanup, a timeout, an owner, and an alert. It is also one more place where a stuck aggregate can hide. For consumers that can apply "newest wins", skip it.

Version checks do not order across aggregates or topics. They protect one aggregate's sequence. If a rule needs to know that an event on topic A came before an event on topic B, a version number does not provide it. That is a workflow problem, and Chapter 10 treats it as one.

When not to bother. If the consumer is commutative, if the events carry full state and "newest wins" is correct, or if a late event is harmless and a periodic reconciliation will repair it, skip the machinery. Doing less is a decision with a cost, and it should be a recorded one.

Common mistakes

- **Asking for global ordering.** It feels like the safe default and caps throughput at one reader. State the real requirement, per-aggregate order, and design for that.
- **Keying by something that is not the aggregate.** Keying by event type, by service, or by a random value happens because the default does it for you. The fix is to key by the identifier of the entity whose state the consumer changes.
- **Using timestamps to order events.** They look like natural version numbers. Clock skew makes the order meaningless, so carry a version from the aggregate instead.
- **Parallelizing inside a consumer without keyed lanes.** A thread pool raises throughput and breaks the partition's order. Hash the key to a lane and give each lane one worker.
- **Dropping stale delta events.** A version check suits state-carrying events. Applied to increments and ledger entries, it discards real changes. Classify each event as state or delta before choosing a policy.

- **Forgetting tombstones.** Deleting a row on cancellation lets a late event recreate the order. Keep the key and last version after deletion for as long as stale events can arrive.
- **Holding events in memory.** A buffer that dies with the process loses events that the broker already considers delivered. Persist it, in the same transaction as the view.
- **Changing the partition count in production without a plan.** The mapping changes for about two thirds of keys when going from four to six partitions, and the order changes silently with it. Drain, migrate, or guard with versions.
- **Mixing partitioners across producers.** Services in different languages may hash the same key differently. Pin one partitioner everywhere, or compute the partition in the relay.
- **Unbounded lanes.** A slow lane then fills memory while the commit point stands still. Bound each lane, pause the partitions, and commit per partition from the poll loop.
- **Trusting the inbox for external effects on rebalance.** The inbox covers the consumer's own database only. An email or a carrier call needs its own idempotency key.
- **Assuming shared keys order events across topics.** The same `order_id` on two topics gives no sequence between them. Make the consumer wait for both, or model the dependency in a workflow.

Key takeaways

- Global ordering costs you all parallelism and is rarely needed; per-aggregate ordering is the order worth designing for.
- A partition key that identifies the aggregate gives ordered events per entity on a log-based broker, but only with the same partitioner and the same partition count everywhere; queue-based brokers need a grouping feature or a consumer that tolerates disorder.
- Parallelism inside a consumer must be keyed, with one worker per lane, bounded, and committed per partition, or it silently undoes the order the partition delivered.
- Version numbers incremented by the aggregate inside its transaction let a consumer detect stale, duplicate and missing events, while time-

stamps cannot.

- Ignoring a stale event is correct for state-carrying events and wrong for deltas and ledger entries, which must be buffered or applied in any order.
- Retry topics, redrive and partition changes all reorder messages, so any consumer that depends on order needs a version check before it uses them.
- Every hold-back buffer needs a timeout, an owner and an alert, because a gap that never closes is a stuck aggregate; Chapter 8's per-key blocking is the version-free form of the same idea.

Ordering gives each event a place within one aggregate, but a business process spans several aggregates and services, and some of its steps have to be undone when a later one fails. Chapter 10 turns to sagas, where the order of steps is the design and compensation is how a workflow recovers.

PART IV

Workflows and Contracts

Chapter 10 — Sagas and Long-Running Workflows

Picture a Tuesday afternoon at Parcelly. A buyer places an order for a ceramic lamp. Orders emits `OrderPlaced`, Payments authorizes the card, and Inventory reserves the last lamp in the seller's stock. Then the card is declined and Payments emits `PaymentFailed`. At the same moment a second buyer is staring at a product page that says "out of stock", because the reservation for the first order is still holding the lamp.

Nobody did anything wrong. Each service did its own job, each local transaction committed, and every message was delivered. The trouble is that the business operation, "sell this lamp to this buyer", is spread over four services and several minutes, and no transaction covers it. Half of it happened. The other half will not, and somebody has to give the lamp back.

The timing matters. Nothing has been captured, so the undo is cheap: release the reservation and let the order end. Had the same lamp already been paid for and the failure come later, giving it back would be a different and more expensive act, and the chapter returns to that line.

Now suppose the same failure arrives while the seller is cancelling the listing, while the buyer is editing the delivery address, or while the capture response is lost on its way back. This chapter is about the machinery that lets a flow like this finish cleanly or undo itself cleanly. How do you get a multi-service workflow to end in a state the business accepts, when you cannot hold a lock across it and any step can fail?

The problem, and why it is hard

Inside one database, a transaction gives you atomicity: either every change happens or none does. Across services you do not have that. Each service owns its database, and the message broker sits between them.

Chapter 4 showed how to make a single service's state change and its published event atomic. It did not make the next service's reaction atomic with the first service's change, and it cannot.

The old answer was distributed transactions, usually two-phase commit. They hold locks in every participant until a coordinator decides, and if the coordinator fails after participants have voted, they wait holding those locks until it returns. They need participants that support the protocol, and they make the slowest or least available participant the limit for all of them. An external payment provider or a carrier API does not take part in your transaction at all. For a flow that includes a call to a third party, that option is closed before you start weighing it.

A **saga** is the alternative. It splits one business operation into a sequence of local transactions, one per service. When a step fails, the saga does not roll back, because the earlier transactions have already committed and other people may have read their results. It runs **compensations**: new local transactions that semantically undo the earlier steps. A compensation is a forward action that moves the business back to an acceptable state, not a time machine.

Three things make this hard in practice.

First, the intermediate states are visible. Between "stock reserved" and "payment captured", the order is neither confirmed nor cancelled, and everything that reads it sees that in-between state. Atomicity is gone; so is isolation.

Second, compensation is a design decision, not a mechanism. Some steps have a clean inverse, such as releasing a reservation. Others have only an approximate one, such as refunding a captured payment, which costs fees and takes days. Some have none: an email that went out stays sent.

Third, the saga itself can fail. The process driving the workflow can crash between steps, a message can arrive twice, a reply can never arrive, and a compensation can fail just as an action can. Everything Chapters 4 to 9 built for single messages now has to hold for a whole flow.

Note: A useful test before adopting a saga: list each step before the pivot (the step after which the saga only moves forward, defined later in this chapter) with its compensation and say what it costs. Steps after the pivot are meant to have none. If several steps before it read "none" or "a support ticket", the boundary between those services may be in the wrong place, and moving the steps into one service's local transaction can be cheaper than coordinating them.

Parcelly's order as a saga

Take the flow from the happy path and name its steps as a saga. Each step is a local transaction in one service, followed by an event that is written to that service's transactional outbox in the same transaction.

Step	Service	Local transaction	Compensation
1	Orders	Create the order in a pending state, emit <code>OrderPlaced</code>	Mark cancelled, emit <code>OrderCancelled</code>
2	Payments	Authorize the card, emit <code>PaymentAuthorized</code>	Void the authorization, emit <code>PaymentVoided</code>
3	Inventory	Reserve stock, emit <code>StockReserved</code>	Release the reservation, emit <code>StockReleased</code>
4	Payments	Capture the payment, emit <code>PaymentCaptured</code>	None inside the saga: this is the pivot (a refund belongs to a separate cancellation flow)
5	Orders	Confirm the order, emit <code>OrderConfirmed</code>	None: retried until it succeeds

The two Payments steps are not database transactions in the strict sense. Each is a call to an external provider with a stable idempotency key, followed by a local record of the outcome, which is the pattern of Chapter 7. The saga's guarantees rest on that pairing, and a timeout on the provider call carries the uncertainty Chapter 3 described.

Steps 2 and 3 run in parallel, and capture is requested only after both `PaymentAuthorized` and `StockReserved` have arrived. Shipping reacts to `OrderConfirmed` and Notifications reacts to what Shipping emits. We will treat those last two as outside the saga for now and come back to why.

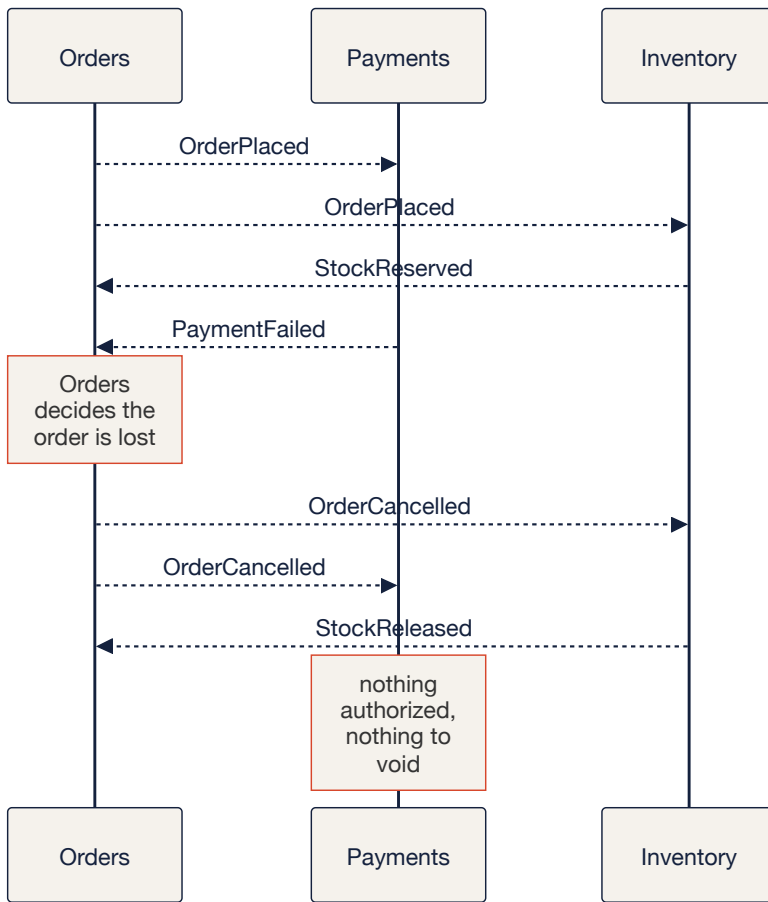
The failure branches follow the table from the event flow. If `PaymentFailed` arrives, no authorization exists, so the only compensation is for Inventory to release a reservation that may or may not have been made, emitting `StockReleased`. If `StockReservationFailed` arrives, an authorization is held and nothing is captured, so Payments voids it and emits `PaymentVoided`. A refund appears only after capture. Before capture, the undo is a release or a void, and calling it a refund would be wrong in the ledger and wrong in the buyer's bank statement.

In both failure branches Orders reacts and emits `OrderCancelled`, and the other services compensate in response to it. That sentence is the whole choreography design. Whether it is the whole design you want is the next question.

Choreography

In a **choreographed** saga there is no central owner. Each service listens for events, does its local transaction, and publishes the next event. The workflow exists only as the sum of those subscriptions.

The diagram below shows the payment failure branch as a choreography. Look at who decides what: nobody gives an instruction, and each reaction is triggered by an event the service already subscribes to.



The strengths are real. Services stay loosely coupled: Inventory knows what `OrderCancelled` means and nothing about Payments. There is no single component whose failure stops every workflow. Adding a consumer, such as an analytics service that reacts to `OrderConfirmed`, requires no change to the others.

The weaknesses show up as the flow grows. The workflow definition is smeared across subscriptions in several codebases, and "what happens after `StockReserved`?" cannot be answered by reading one file. Cyclic dependencies creep in: Orders listens to Payments, which listens to Orders. Every new branch adds events and every service must be updated in a careful order. Most importantly, nobody owns the question "where is order 81724 right now?". The answer is a join across the logs of several services, which is the problem Chapter 1 named and Chapter 13 returns to with tracing and reconciliation.

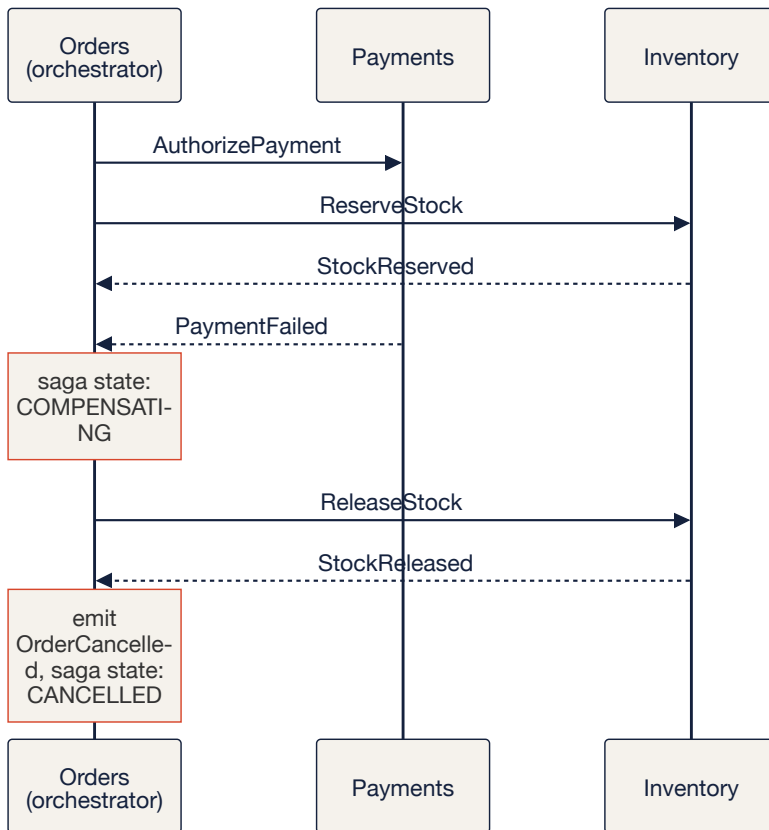
I find choreography works well when the flow is short, has few branches, and no step depends on a conjunction of earlier steps. Two or three services reacting to one event is a natural fit. The Parcelly order flow is already past that line, because capture needs two earlier results to have arrived.

Orchestration

In an **orchestrated** saga, one component owns the workflow. It holds the state, decides the next step, sends commands to participants, and reacts to their replies. The participants know how to do their local job and report the outcome; they do not know where they sit in the larger flow.

This section introduces two commands beyond the canonical ones: `AuthorizePayment` (Orders to Payments) and `ReserveStock` (Orders to Inventory). Confirmation needs no command, because it is the orchestrator's own local transaction. Like all commands in this book they carry a `command_id` in place of `event_id`, and the replies are the ordinary events. In the choreographed version Payments and Inventory started on `OrderPlaced`; here the orchestrator tells them to.

The diagram shows the same failure as before, but now every arrow out of Orders is a command. Look at how the saga state lives in one place and how the compensating commands name exactly the work to be undone.



The orchestrator is the place where the question "what happens next?" is answered. It is a state machine, and the state machine can be read, tested and drawn. A new branch is a change in one place. The orchestrator knows which steps completed, so a compensation targets what actually happened instead of everything that might have.

The costs are the mirror image. The orchestrator knows every participant and becomes a coupling point. If it is careless, domain logic leaks out of the services and into it, and the participants turn into remote procedure calls with extra steps. It is also a component that must be correct, always on, and itself recoverable, since a crashed orchestrator with its state lost is a saga nobody will ever finish.

In my view, an orchestrator that lives inside the service that owns the business process is the most defensible form. At Parcelly that is Orders, which already owns the order lifecycle. A standalone orchestration service that

owns workflows for everyone tends to become the thing every team must change to ship anything.

Choosing between them

Neither style is better in general. The choice depends on the shape of the flow and on who must answer questions about it.

Question	Choreography	Orchestration
Where is the flow defined?	Spread over subscriptions	In one state machine
Coupling between services	Through event contracts only	Orchestrator knows each participant
Adding a step or branch	Touch several services	Touch the orchestrator and the participant
Parallel steps and joins	Awkward; each service tracks what it has seen	Natural; the orchestrator holds the join
"Where is order 81724?"	Join across logs	Read one row
Single point of failure	None, but no owner either	The orchestrator, which must be recoverable
Risk when it grows	Cyclic, hidden dependencies	A god service full of domain logic
Good fit	Short flows, few branches	Long flows, conditions, time-outs, joins

Mixing the two is normal and sensible. An orchestrated order saga can still emit `OrderConfirmed` as a plain event that Shipping and Notifications react to in a choreographed way. The saga should cover the part of the flow that needs coordinated undo; notifications and analytics do not, and pulling them in only adds states.

Compensations and steps that cannot be undone

Design the compensation of a step when you design the step, not when the first incident asks for it. For each step, write down what the business wants when the saga fails after it.

A few rules keep compensations honest.

A compensation is a new business action, not an undo of the database. `StockReleased` does not delete the reservation row's history. It records that the reservation ended. `PaymentRefunded` does not erase the capture; the ledger holds both entries. That is why Ledger is append-only and why compensations show up there as real events.

Order the steps by how hard they are to undo. Put steps that are easy to reverse first and steps that are hard or impossible to reverse last. Authorization and reservation are cheap to undo, which is why they come before capture. Capture is where a refund starts costing money and time. This leads to the idea of a **pivot step**: the point after which the saga no longer compensates and only moves forward. In *Parcelly*, capture is the pivot. Before it, failure means release and void. After it, a failure in confirmation is not retreat but retry, because the business has the buyer's money and owes an order.

Some actions have no inverse. An email that has been sent, a carrier that has received the shipment, a webhook a seller's system has acted on: you can send a follow-up, but you cannot unsend. These steps belong after the pivot, and their failure should be handled by retry and by correction, not compensation. This is also why Notifications is purely reactive and never part of the saga: its steps cannot be compensated and should not be able to block anything.

Compensations must be retried until they succeed. An action failure leads to a compensation. A compensation failure leads to nothing, except a stuck saga and an inconsistent system. Treat a compensation as something that must eventually complete, with the retry discipline of Chapter 6 and the poison-message handling of Chapter 8. A compensation that fails permanently should reach a human through a parking lot, with the saga state recorded, not be dropped.

Compensations must be idempotent, and must tolerate arriving early. The first is Chapter 7's territory: a duplicate `ReleaseStock` must not release twice. The second is subtler. Suppose Orders sends `ReserveStock`, times out, and starts compensating by sending `ReleaseStock`. The reservation was slow, not lost, and the release can reach Inventory first. If Inventory treats a release for an unknown reservation as a no-op, the late `ReserveStock` then succeeds and the lamp is held forever. The safe shape

is for Inventory to record the release against the order, so that the later reservation sees it and refuses. Some texts call this a tombstone. Whatever the name, the compensation must leave a mark the action can read.

Saga state and timeouts

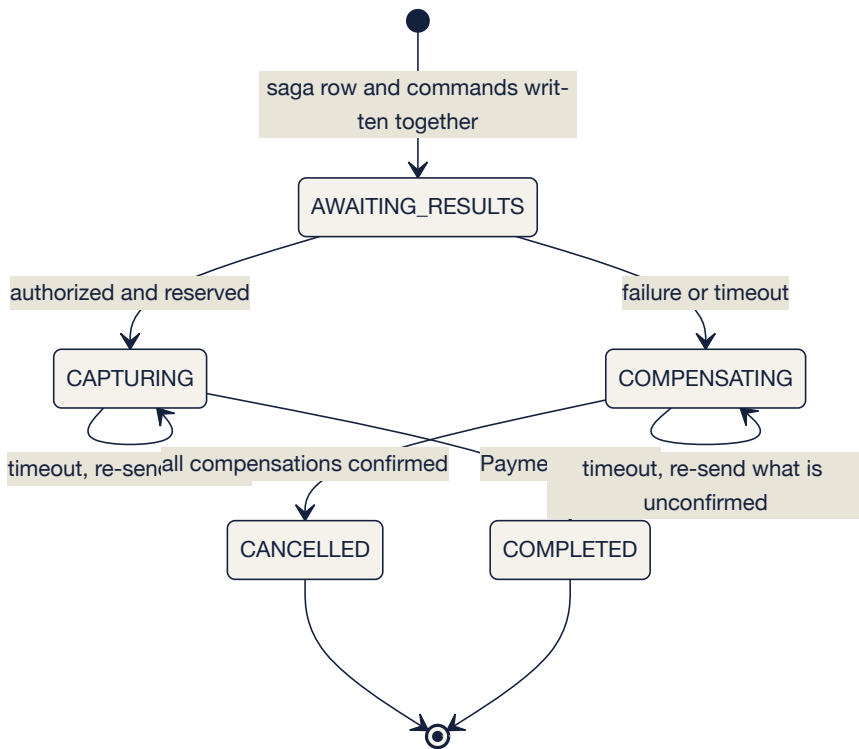
An orchestrated saga needs durable state, and it belongs in the same database as the service that runs it, so that state changes join the same local transaction as the outbox and inbox writes.

```
table saga_instance:
  saga_id      text primary key
  order_id     text unique not null
  state        text not null    // see below
  authorized   boolean not null
  reserved     boolean not null
  stock_released boolean not null // compensation confirmed
  auth_voided  boolean not null // compensation confirmed
  rounds       int not null     // deadlines spent in this state
  deadline_at  timestamp not null
  version      int not null
  updated_at   timestamp not null
```

The `state` column is a small set of named positions: `AWAITING_RESULTS`, `CAPTURING`, `COMPLETED`, `COMPENSATING`, `CANCELLED`. The saga row and the first commands are written in one transaction, so there is no separate "started" position that anyone could observe. The two booleans `authorized` and `reserved` record which results have arrived, which is how the join works: capture is requested when both are true and the state is `AWAITING_RESULTS`. The two flags `stock_released` and `auth_voided` record which compensations have been confirmed, so "all compensations confirmed" is something the row can answer. A compensation that is not needed starts out confirmed: after `PaymentFailed` there is no authorization to void, so `auth_voided` is true on entry to `COMPENSATING`.

A `unique` constraint on `order_id` stops two sagas from running for one order. The `version` column guards against two handlers updating the same row at once, and the handlers below depend on it.

A state machine drawn once is worth more than any prose description. The next diagram shows the legal transitions; look at the exits that do not lead to `COMPLETED`, and at the two states that loop on their own timeout.



Notice that **CAPTURING** has no exit to **COMPENSATING**. That is the pivot expressed in the state machine. A capture whose reply never came is re-sent, and if the provider reports a permanent decline, it is handled as a separate, explicit business path with a human in it.

The loops matter as much as the exits. A saga in **CAPTURING** or **COMPENSATING** is waiting for a reply, and a reply can be lost like any other message. If only **AWAITING_RESULTS** had a deadline, a saga stuck in either of the other two would sit there with nothing to wake it. So every non-terminal state has a deadline, and the deadline is reset on each transition.

Timeouts

Every wait in a saga needs a deadline, because a reply that never comes is also a failure, and the one most likely to be forgotten. Chapter 3 described how a timeout hides three possibilities: the action failed, the action succeeded and the reply was lost, or the action has not finished yet. The saga cannot tell which, so it needs a policy for each wait, written down.

A timeout is a message too. This chapter introduces one internal event for it, `SagaTimedOut`, emitted by the orchestrator's own timer. The usual implementation is a timer process that selects sagas whose `deadline_at` has passed and feeds each one a `SagaTimedOut` event through the same handler as every other input. That way the timeout takes the same path as everything else, with the same inbox, the same state checks and the same outbox.

```
function handleSagaTimedOut(message):
  if inboxContains(message.event_id):
    // fast path only; the unique constraint is the safety net
    ack(message)
    return
  try:
    transaction:
      inboxInsert(message.event_id)
      saga = loadSaga(message.payload.saga_id)
      if saga.state == "AWAITING_RESULTS":
        saga.state = "COMPENSATING"
        saga.rounds = 0
        sendOutstandingCompensations(saga, message.event_id)
      else if saga.state == "COMPENSATING":
        saga.rounds = saga.rounds + 1
        sendOutstandingCompensations(saga, message.event_id)
      else if saga.state == "CAPTURING":
        // same command_id as the first attempt: a repeat, not
        new work
          saga.rounds = saga.rounds + 1
          outboxInsert(CapturePayment, saga.order_id,
            command_id: commandIdFor(saga.saga_id,
              "capture"))
        // any other state (COMPLETED, CANCELLED): a stale
        timeout, recorded and dropped

        // sets a clock for a non-terminal state; escalates past
        MAX_ROUNDS
          armNextDeadline(saga)
          saveSaga(saga, expected_version: saga.version)
          ack(message)
      on UniqueViolation:
        // a concurrent duplicate committed first; the work is done
        ack(message)
      on VersionConflict:
        // another handler changed the row; nothing above committed
        nack(message)
```

The `AWAITING_RESULTS` branch sends compensating commands without asking which step completed. That is deliberate. At timeout time the orchestrator does not know whether a late `StockReserved` is on its way, so

it asks for the undos that are still outstanding and relies on the participants to treat "nothing to undo" as success. Chapter 7 is why that is safe.

`armNextDeadline` does two things inside the same transaction. It sets `deadline_at` to `now() + timeoutFor(saga.state)`, so no non-terminal state is left without a clock. And once `saga.rounds` passes `MAX_ROUNDS`, say five, the saga stops being a timing problem and becomes a human one: it raises an alert and records the saga state in the parking lot of Chapter 8, and the rounds keep running. Terminal sagas get no new deadline. Because the deadline, the state change and the commands commit together, a crash cannot leave a saga that has moved on but is no longer watched.

```
function sendOutstandingCompensations(saga, causing_event_id):
    // asked for again each round until its confirmation has arrived
    if not saga.stock_released:
        outboxInsert(ReleaseStock, saga.order_id,
                     command_id: commandIdFor(saga.saga_id, "release",
causing_event_id))
    if not saga.auth_voided:
        outboxInsert(VoidAuthorization, saga.order_id,
                     command_id: commandIdFor(saga.saga_id, "void",
causing_event_id))
```

A saga in `CAPTURING` is never turned around. The pivot has passed, so the only moves are to re-send the capture with the same key and to ask a person.

Choosing the length of a timeout is not something I can do for you. It depends on how long the slowest healthy path takes, how long a buyer will wait, and what holding stock costs a seller. A deadline set to a very short value turns slow into failed and floods the system with compensations. A very long one holds reservations while buyers watch an out-of-stock page.

Late arrivals need a rule as well. If `PaymentAuthorized` arrives after the saga is `CANCELLED`, the orchestrator must issue `VoidAuthorization` again, because someone is holding funds on a card for an order that does not exist. The late event proves that an authorization exists. The void sent at timeout may still be in flight, it may have been processed just after the authorization completed, or the provider call may have run before Payments checked for the mark that the void leaves. Sending the void again settles all three.

That re-send only works if it is not mistaken for a repeat. Here two rules about identity meet. A command whose repeat must not cause new work, such as `CapturePayment`, keeps one `command_id` for the saga and the step, so a retry is recognized and answered with the first result. A compensation that may legitimately be issued again takes its `command_id` from the message that triggered it: saga, step and the causing `event_id`. The re-issued void is then a new command. It also has to be idempotent on the target state, not only on the command, so Payments reads

`VoidAuthorization` as "no live authorization may remain for order 81724" and checks that, instead of replaying the cached answer "nothing to void" from the first attempt. A state machine that ignores events arriving in the wrong state loses these. Handle them explicitly: decide, per state and per event, whether to apply, ignore, or compensate.

Semantic locks

Because sagas give up isolation, other flows can read and change an order mid-saga. A buyer may edit the delivery address while capture is pending, a seller may cancel, a support agent may refund. Chapter 9's version numbers help with stale events, but they do not stop two flows from making incompatible business changes.

A **semantic lock** is a flag in the data that says "a saga is working on this". It is not a database lock. It is a status value that other code is written to respect. In Parcelly the order status `PENDING` is one: while it holds, the order cannot be edited, cannot be shipped and cannot be refunded, and the order page shows "processing". Likewise, the reservation in Inventory is a lock on the lamp, which is why the second buyer saw "out of stock".

The semantic lock has to be released on every path: on completion, on compensation and on timeout. A saga that stalls without releasing its lock leaves the order frozen, which is a quiet failure that surfaces as support tickets. The timeout above is the guard, and Chapter 13's reconciliation is the second line.

Two further countermeasures are worth knowing. One is to make updates commutative where possible, so that order does not matter. The other is to design a conservative view: for example, show the seller a stock count that already subtracts pending reservations, so nobody promises what is about

to be held. Neither removes the need to decide, for every status, what other flows may do while a saga is in flight. I treat that table, status by allowed operation, as part of the design of any saga.

Outbox and idempotency inside a saga

The saga handler is where the earlier chapters meet. It consumes a message, changes state, and issues the next command or event. Done carelessly, it is a dual write twice over.

The shape that works combines Chapters 4 and 7. Each handler runs one local transaction that does three things: record the message in the inbox, update the saga row, and write the outgoing commands or events to the transactional outbox. The relay publishes from there.

```
function handleStockReserved(message):
  if inboxContains(message.event_id):
    ack(message)
    return
  try:
    transaction:
      // unique event_id: a concurrent duplicate fails here
      inboxInsert(message.event_id)
      saga = loadSagaByOrder(message.payload.order_id)
      if saga.state in ["CANCELLED", "COMPENSATING"]:
        // late success after we gave up: undo it
        outboxInsert(ReleaseStock, saga.order_id,
                     command_id: commandIdFor(saga.saga_id,
"release", message.event_id))
      else if saga.state == "AWAITING_RESULTS":
        saga.reserved = true
        if saga.authorized:
          saga.state = "CAPTURING"
          outboxInsert(CapturePayment, saga.order_id,
                       command_id:
commandIdFor(saga.saga_id, "capture"))
        // a conflict rolls back the whole transaction, inbox
row included
        saveSaga(saga, expected_version: saga.version)
        // any other state: irrelevant, recorded and dropped
        // commit point: inbox, saga state and outbox rows are one
unit
        ack(message)
      on UniqueViolation:
        ack(message)
      on VersionConflict:
        // the redelivery will see the other handler's flag
        nack(message)
```


This is the rule Chapter 7 introduced for consumers that publish: the inbox row, the state change and the outbox insert commit in one transaction. If the process crashes before commit, nothing happened and the broker redelivers. If it crashes after commit and before ack, the redelivery finds the inbox row, is acknowledged and the work is skipped; a concurrent duplicate loses at the unique constraint and is acknowledged the same way. A stale or irrelevant message is recorded and acknowledged too, so it never returns. The saga advances once per message, regardless of how many times the message is delivered.

The `expected_version` check is what makes the join correct.

`PaymentAuthorized` and `StockReserved` can be handled at the same moment by two workers, and each loads a row with both flags false. Without the check, the second write overwrites the first, and the saga waits in `AWAITING_RESULTS` until its timeout compensates a healthy order. With it, one transaction loses, rolls back with its inbox row, and its redelivery loads a row that already holds the other flag. Loading the row with a row lock achieves the same thing.

The commands the saga sends are themselves messages that will be delivered at least once, so the participant needs the same protection, and the `command_id` is where it comes from. For an action that must not repeat, derive it from the saga and the step name. That lets Payments recognize a retried `CapturePayment` and answer with the first result. If the orchestrator regenerates a random `command_id` on each attempt, every retry looks like new work, and the buyer can be charged twice. Compensations that may be re-issued follow the other rule from the section on late arrivals: the id comes from the triggering message, and the participant is idempotent on the target state. The `causation_id` and `correlation_id` carried in the envelope also make each saga traceable: the `correlation_id` is the flow, and each `causation_id` is a step in it.

Warning: A compensation that is sent without an idempotency key is as dangerous as a payment that is. A retried `RefundPayment` without a stable key can refund the buyer twice, and the second refund is money that does not come back.

Where workflow engines fit

Everything above can be built with a table, a timer and a few handlers, and for a flow of the size of Parcelly's order I would start there. At some point the machinery becomes the larger part of the code: durable timers, retries per step, versioning of running workflows, a visual history of each instance. That is the problem workflow engines address.

The class of tool is broad. Some are libraries or services that persist the workflow's progress and resume it after a crash, with code that looks like a sequential program. Others interpret a declarative definition of states and transitions. Cloud providers offer managed versions, and several open-source projects exist. I will not compare products, because the landscape changes faster than a book and the choice depends on your language, your operational skills and your tolerance for running another critical component.

What I would ask of any engine is a short list. Where does workflow state live, and can you inspect it? How does it make a step's side effect and its own progress atomic, or does it leave you the dual write? What happens to a workflow already running when you deploy a changed definition? How do compensations and timeouts get expressed? An engine that stores progress but runs your step in a separate transaction still needs idempotent steps; the engine moves the problem, it does not remove it.

The honest summary is that an engine saves you building and operating the state machine, and costs you a dependency, a model to learn, and a new thing that can be down. For one or two small sagas, a table is cheaper. For many long-running workflows with human steps and long waits, an engine pays for itself.

Trade-offs and costs

A saga trades atomicity for availability and independence, and the price is paid in several currencies.

Complexity. Every saga multiplies states. A flow with three steps, each with an action and a compensation, plus timeouts and late arrivals, has far more reachable states than the happy path suggests, and each one needs a decision and a test. Chapter 15 returns to how to test them.

Visible intermediate states. There is no isolation. Buyers, sellers, support tools and other services see an order that is half done. You design for that with semantic locks and clear statuses, and you accept that some users will see it anyway.

Eventual consistency, on purpose. Between a failure and the last compensation, the system is inconsistent, for seconds or, with a bad timeout, hours. That is acceptable only if the business has agreed to it.

Operational ownership. Someone must watch for sagas stuck in a state beyond their deadline, compensations that cannot complete, and the age of the oldest running saga. Without an owner and an alarm, sagas fail silently.

Compensation is not free. A refund has fees and a delay, a cancelled shipment may carry a fee, and a message to a buyer that "your order was cancelled" is itself a visible event. Design the flow so the common failure is cheap to undo.

When not to use a saga: when the steps can share a local transaction; when failure is rare and a manual fix is cheaper than building compensations, which is a legitimate choice for a low-volume internal process; when the "workflow" is really independent reactions that need no coordinated undo, such as sending an email or updating a search index. And when you cannot define the compensation. If a step cannot be undone and cannot be postponed, the honest design question is whether the boundary between the services is in the wrong place.

Common mistakes

- **Building a saga where one transaction would do.** It is tempting because the services already exist and the diagram looks modern. If two steps share an owner and a database, merge them into one local transaction.
- **Designing the happy path and leaving compensations for later.** Later arrives as an incident. Write the compensation beside the step, and name the event it emits.
- **Calling every undo a rollback or a refund.** Before capture the compensation is a release or a void; a refund exists only after capture.

Using the wrong one hides the real state in the ledger and in the buyer's statement.

- **Letting a saga wait forever.** A missing deadline is invisible until a customer calls. Give every wait in every non-terminal state a timeout and a named outcome, including the waits for a capture reply and for compensation confirmations.
- **Ignoring events that arrive in the wrong state.** A late `PaymentAuthorized` for a cancelled order is a held card. Decide per state and event whether to apply, ignore or compensate.
- **Non-idempotent compensations.** A retried release or refund that acts twice causes a second incident while fixing the first. Give actions that must not repeat a stable `command_id`, derive the id of a re-issuable compensation from its cause, and make participants idempotent on the target state. One fixed id for a re-sent void replays "nothing to void" and leaves the authorization held.
- **Putting the whole business in the orchestrator.** The orchestrator should decide order and outcome; the participants should own the rules. If the orchestrator validates stock levels, it is doing Inventory's job.
- **Putting uncompensable steps early.** Sending the carrier handoff before capture means a failure leaves you with a shipment and no money. Order steps so that the hard-to-undo ones come last.
- **Skipping the lock release.** A semantic lock set on `PENDING` and never cleared on timeout or compensation freezes the order. Release it on every exit path.

Key takeaways

- A saga replaces a distributed transaction with a sequence of local transactions and compensations, and it gives up isolation to do so.
- A compensation is a new business action, such as `StockReleased` or `PaymentVoided`, not a rollback, and before capture it is never a refund.
- Choreography fits short flows with few branches, orchestration fits flows with joins, timeouts and many branches, and the two are often mixed, with the orchestrator inside the service that owns the process.

- Order steps by how hard they are to undo, and treat the first step that cannot be compensated, such as capture, as a pivot after which the saga only moves forward.
- Every wait in every non-terminal state needs a deadline, every state needs a rule for late and duplicate events, and every compensation must be idempotent and tolerate arriving before the action it undoes.
- Inside a handler, one local transaction should combine the inbox, the saga state and the outbox, guarded by a version check so concurrent handlers cannot overwrite each other, and a `command_id` that is fixed for an action that must not repeat, such as capture, but derived from the triggering message for a compensation that may be re-issued.
- A workflow engine can own the state machine, timers and history, but it does not remove the need for idempotent steps and a clear ownership of stuck workflows.

A saga assumes that events describe what happened in a form every participant can read and trust. Whether they do depends on what each event carries, what it is called and who owns it, which is the subject of Chapter 11.

Chapter 11 — Event Design

Imagine that Parcelly's Shipping team wants to schedule shipments faster. Today it receives `OrderConfirmed`, which carries an `order_id` and little else, and then calls Orders over HTTP to fetch the delivery address and the order lines. Orders is slow on Monday mornings, so Shipping slows down with it. A tech lead proposes the obvious fix: put the whole order into the event, so Shipping never needs to ask.

Within a week the event has grown. Orders adds the buyer's email address, the full address, the seller's payout account and a field called `internal_status` that its own admin screen uses. Notifications starts reading the email from the event instead of its own table. The Ledger team notices that the payout account now travels through the broker and sits in a topic with a long retention period. Then Orders renames `internal_status` while refactoring, and three consumers fail on the same afternoon.

Nobody in this story made a reliability mistake in the sense of earlier chapters. The outbox worked, the consumers were idempotent, the retries were sensible. The damage came from the design of the message itself: what it says, what it carries, and who is allowed to change it. This chapter asks how to design an event so that it stays useful, safe and changeable for years after the first consumer reads it.

The problem, and why it is hard

Chapter 1 pointed out that services in an event-driven system are coupled through the contract of the event: its name, its fields and its meaning. That coupling is invisible at the time you write the event, because at that moment there is one producer and one consumer, and both are written by people who sit near each other. The contract feels like an internal detail.

It stops being internal the moment a second consumer subscribes, and the producer usually learns about it late or never. On a log-based broker, a new consumer group can start reading the topic without telling anyone. On a queue-based broker, someone binds a new queue to the same exchange or topic subscription. From that point on, every field in the payload is a promise to people you may never meet.

Three things make this hard in practice.

- **Events are easy to add to and hard to take from.** Adding a field costs a line of code. Removing it requires finding every consumer, including the ones in repositories you do not read and the ones that replay old history.
- **The design pressure points in the wrong direction.** The producing team wants to ship, so it emits whatever it has at hand. The consuming teams want to avoid extra calls, so they ask for more fields. Nobody is paid to say no.
- **Mistakes are durable.** An event written today may sit in an event log for months and be replayed into a consumer that does not exist yet. A badly named event or a leaked personal field stays in storage long after the code that produced it is gone.

So event design is not decoration. It is the part of the system where a few hours of thought decide how expensive every later change will be.

Events, commands and documents

The first design decision is what kind of message you are sending. Three kinds are common, and they have different rules.

An **event** states a fact that already happened: `PaymentCaptured`. The producer does not care who listens, and it does not expect anything in return. Events are named in the past tense, because they report something that has already happened; a later event can counteract it, but cannot unmake it. A consumer may ignore an event, and that is a legitimate response.

A **command** asks one specific service to do something: `CapturePayment`. It has a single intended recipient and an expectation that the recipient acts. A command can be rejected, and the sender usually cares about the

outcome. Commands are imperative, carry a `command_id` in place of `event_id`, and are never called events. An orchestrated saga, which Chapter 10 covers, drives its participants with commands of this kind.

A **document message** carries a block of data without saying that anything happened or that anyone should act, such as a nightly catalog snapshot from a seller. The receiver decides what it means. Document messages are legitimate, but they are the kind most often disguised as events, and the disguise causes confusion.

Kind	Says	Recipient	Naming	Can the receiver refuse?
Event	A fact happened	Anyone subscribed	Past tense: <code>OrderPlaced</code>	Yes, by ignoring it
Command	Do this	One named service	Imperative: <code>CapturePayment</code>	Yes, by replying with a failure
Document	Here is some data	Whoever reads it	Noun: <code>CatalogSnapshot</code>	Not applicable

The common failure is the command in event clothing. Suppose Orders publishes `SendConfirmationEmail`, an imperative name, and calls it an event because it travels through the topic. Orders now depends on Notifications behaving a certain way, but has no way to hear a refusal. If Notifications is down, Orders does not know. If a second service subscribes, it faces an instruction that is not addressed to it.

I would apply a quick test. Ask whether the producer would still be correct if nobody consumed the message. For `OrderPlaced` the answer is yes: the order was placed regardless. For `SendConfirmationEmail` the answer is no, because the message only has meaning if somebody acts. That second kind belongs on a channel where the recipient is named and failure comes back to the sender.

Part of this is also a question of direction. Events point from the owner of a fact to anyone interested. Commands point from a coordinator to an executor. When an event name starts to sound like an instruction to a particular service, the direction has been reversed.

What a command looks like

The two kinds also fail differently, and the envelope has to reflect that. A lost event is silent by design: the producer never expected anyone in particular to act, so nobody notices the absence. A lost or late command is different. The sender is waiting for an outcome, and the only thing that reveals the loss is its own deadline, which is why the saga in Chapter 10 puts a deadline on every non-terminal state. A command can also arrive after the sender has given up and compensated, so the receiver needs a way to refuse it.

A command therefore carries a smaller, different envelope from an event:

```
{
  command_id:      "cmd-capture-saga-7731",          // stable across
retries of the same intent
  command_type:    "CapturePayment",
  target:          "payments",                      // one named
recipient
  reply_to:        "orders.saga-replies",           // where the
outcome goes
  expects:         ["PaymentCaptured", "PaymentCaptureFailed"],
  correlation_id:   "order-81724",
  causation_id:    "e3",
  expires_at:      "2026-10-06T14:05:00Z",          // optional; judged
on the receiver's clock
  payload:         { order_id: "81724", amount_minor: 4990, currency:
"EUR" }
}
```

`command_id` replaces `event_id`, and Chapter 10 explains how it is derived: from the saga and the step for an action that must not repeat, and with the causing `event_id` for a compensation that may be issued again. `reply_to` and `expects` give the outcome a path back, so a refusal is something the sender can hear. `expires_at` lets a receiver drop a command that arrives after the sender has stopped waiting. It is optional because it costs something: it is compared against the receiver's clock, and Chapter 3 gave the reasons that clock cannot be trusted to the second. Set it with a margin, and treat it as protection against hours-old commands, not against seconds.

Domain events and integration events

Inside a service, code often raises events about its own aggregates: `OrderLineAdded`, `DiscountApplied`, `AddressCorrected`. These **domain events** describe the internal model, and they change whenever the model changes. They are useful within the service, for example to update a read model or trigger a rule in the same transaction.

An **integration event** is the version that crosses the service boundary. It is deliberately chosen, named for what other services need to know, and kept stable. `OrderPlaced` is an integration event. It is not a dump of whatever the order aggregate looked like at that moment.

The temptation is to publish domain events straight to the broker, because the machinery already exists. In my view this is the single most common way to leak internal structure. Consider what happens when the Orders team splits one aggregate into two, or merges two line types. If domain events were the public contract, every consumer breaks, though nothing about the business changed.

The fix is a translation step in the producing service. Domain events remain free to change. A small mapping function decides which of them become integration events, with which names and which fields. That function is where you decide what the public contract is, and the transactional outbox from Chapter 4 is a natural place to call it: the integration event is what gets written into the outbox row.

```

function toIntegrationEvent(domainEvent, context):
    // the mapping is the public contract; internals stay behind it
    if domainEvent.kind == "OrderSubmitted":
        return {
            event_id:      newEventId(),
            event_type:     "OrderPlaced",
            schema_version: 1,
            occurred_at:    domainEvent.occurred_at,
            producer:       "orders",
            correlation_id:  context.correlation_id,
            causation_id:   context.causation_id,
            partition_key:  domainEvent.order_id,
            payload: {
                order_id:    domainEvent.order_id,
                buyer_id:   domainEvent.buyer_id,
                line_count:  domainEvent.lines.length,
                total_minor: domainEvent.total_minor,
                currency:    domainEvent.currency
            }
        }
    // most domain events never leave the service
    return nothing

```

Note that the internal name `OrderSubmitted` and the public name `OrderPlaced` differ. That is allowed, and healthy. The public name is what other teams learn. The internal name can follow the code.

Thin events and fat events

How much should the payload carry? The two ends are worth naming.

A **thin event**, sometimes called a notification, carries an identifier and little more: "order 81724 was placed". A consumer that needs details calls back to the producer. A **fat event** carries enough state for consumers to act without calling back. This style is often called event-carried state transfer, because the event moves a piece of state from one service to another.

Each side has a failure mode that the earlier chapters help explain.

Thin events bring back the synchronous coupling that events were meant to remove. Shipping can only proceed when Orders is reachable at that moment, which is exactly the Monday-morning problem from the opening. They also create a race: by the time the consumer calls back, the data may

have changed, and the consumer reads a newer state than the one that triggered the event. If the callback fails, the consumer falls into Chapter 6's world of retries against a dependency that may be slow for a long time.

Fat events cost in other places. The payload grows, and so do the number of fields you have promised to keep. Consumers start to treat the event as a replica of the producer's data, and the replica ages. A consumer that cached the address from `OrderConfirmed` on Monday may use it on Friday, after the buyer corrected it, unless it also reads the later event. And everything in the payload is copied to every consumer and to every retained log segment, which raises the stakes for sensitive fields.

	Thin	Fat
Payload	Identifier and event type	Identifier plus the state consumers need
Consumer needs producer to be up	Yes, to fetch details	No
Staleness risk	Low at read, but races with later changes	Consumer holds a snapshot that ages
Contract size	Small	Large
Privacy exposure	Low	Higher
Replay	Needs the producer's current data, which may differ from then	Self-contained, reproduces what was known

I tend to prefer a middle path: **carry what the consumer needs to decide and act, and nothing it merely might want**. For `OrderConfirmed`, Shipping needs the order id, the lines it must pack and a delivery address, so those belong in the event. It does not need the buyer's payment method, the seller's payout details or the discount rules that produced the total. A useful way to find the boundary is to ask each consuming team what it would do with each field. A field with no named consumer and no stated use is a candidate for removal.

A second detail matters here. If the event carries state, say in the payload which moment that state belongs to. Using the aggregate version (Chapter 4), which Chapter 9 calls `order_version` for Parcelly orders, a consumer can tell whether the snapshot it holds is newer or older than a second event; `occurred_at` is only a rough hint, for the reasons given below.

Chapter 9 covers how to detect out-of-order arrival with such numbers, so this chapter only notes that a fat event without a version invites the last-writer-wins trap described there. Do not confuse this number with `schema_version` in the envelope: the aggregate version counts changes to one order and moves on every change, while `schema_version` names the breaking version of the payload's shape and moves rarely (Chapter 12).

The choice does not have to be global. A team can start thin for rarely used events and add fields when a real consumer asks. Starting thin and growing is cheaper than starting fat and shrinking, because shrinking means a breaking change, which Chapter 12 treats at length.

Naming events

The Parcelly event names used throughout the book follow a small set of rules, and this section states them with their reasons.

- **Past tense**, because the event is a fact: `PaymentCaptured`, not `CapturePayment`, which is the command.
- **Subject first, PascalCase**: the aggregate comes first, then what happened. `ShipmentDispatched`, not `DispatchedShipment`. Sorted alphabetically, events about one aggregate cluster together in catalogs and dashboards.
- **Business language, not technical language**. `OrderConfirmed` says something a product owner understands. `OrderRowUpdated` says something about a table.
- **No technology and no consumer in the name**. `OrderPlacedKafkaEvent` leaks the broker. `OrderPlacedForShipping` leaks a consumer, and the next consumer will have to wonder if the event is for it.
- **No service prefix unless two services emit facts about the same noun**. Suppose Orders and Shipping can both report a `DeliveryDelayed`: one means the seller has not handed the parcel over, the other that the carrier is late. Consumers cannot tell them apart by name, so qualifying becomes necessary (`OrderDeliveryDelayed`, `ShipmentDeliveryDelayed`). The example is illustrative; the Parcelly services in this book do not emit both. Otherwise the producer already appears in the `producer` field.

The most important naming rule is the one about meaning. `OrderUpdated` is a name that fails the fact test: it says that something changed and nothing else. A consumer must compare the new state with the old to learn what happened, and it cannot tell a changed address from a changed total. Prefer names that state the business fact: `OrderCancelled`, `ShipmentAddressCorrected`. When you cannot find a business name, it is often a sign that the event describes a database change rather than a business one.

Another rule is stability. Once consumers depend on an event name, re-naming it is a breaking change with the same cost as removing a field. Spend the extra 10 minutes on the name before the first release.

The envelope

Every event in this book carries the same envelope: `event_id`, `event_type`, `schema_version`, `occurred_at`, `producer`, `correlation_id`, `causation_id`, `partition_key` and `payload`. The envelope is the part of the message that tooling can rely on without understanding the business. A relay, a dead-letter handler, a tracing system and a replay tool can all work with it without parsing the payload.

Each field answers a question that some component asks:

Field	The question it answers	Who asks
<code>event_id</code>	Have I seen this exact event before?	Inbox, deduplication (Chapter 7)
<code>event_type</code>	How do I interpret the payload?	Routing, deserialization
<code>schema_version</code>	Which breaking version of the shape does the payload have?	Consumers, schema registry (Chapter 12)
<code>occurred_at</code>	When did the fact happen, as the producer saw it?	Reconciliation, analytics, debugging
<code>producer</code>	Which service owns this fact?	Ownership, support, routing of questions
<code>correlation_id</code>	Which business flow does this belong to?	Support, reconciliation, log search
<code>causation_id</code>	What directly caused this?	Reconstructing the chain

Field	The question it answers	Who asks
<code>partition_key</code>	Which aggregate does this event concern?	Broker, ordering (Chapter 9)
<code>payload</code>	What are the details?	Business logic

Two points need care. `occurred_at` is the producer's clock, and Chapter 3 explained why you should not order events by it. It tells you roughly when the fact happened, and it does not serve as a sequence. The `partition_key` is the aggregate identifier, and its choice was the subject of Chapter 9.

`correlation_id` is not a trace id. Trace context (W3C Trace Context) travels in headers next to the envelope, and Chapter 13 shows how it is captured into the outbox. Traces get sampled and expire; `correlation_id` is the business key that survives both, and it is the one a support engineer searches by.

The envelope across languages

Parcelly's services are not all written in one language. Orders and Payments run on the JVM, Notifications is Node, and the Ledger team's reconciliation jobs are Python. The field list above says what the envelope contains. It does not say how each field is written on the wire, and that is where a second language breaks a contract that looked fine in the first.

Field	Wire type	Rule
<code>event_id</code> , <code>correlation_id</code> , <code>causation_id</code> , <code>partition_key</code>	string	Identifiers are strings, even when they look numeric. A 64-bit id as a JSON number is silently rounded by a JavaScript consumer
<code>event_type</code>	string	A stable, language-neutral name such as <code>PaymentCaptured</code> , never a class or package name, and never a type header a framework fills in by default
<code>schema_version</code>	integer	Small, and an integer in every language
<code>occurred_at</code>	string	RFC 3339 in UTC with a <code>Z</code> suffix, fixed at millisecond precision, no offsets. Parsers disagree about everything else

Field	Wire type	Rule
Money in <code>payload</code>	integer or string	Integer minor units (<code>total_minor</code>) within a documented range, or a decimal string. Never a float. JavaScript numbers are exact only up to $2^{53} - 1$, so a field that can exceed that travels as a string
Enum-like fields	string	Treat the set as open: a consumer must handle a value it has not seen (Chapter 12)

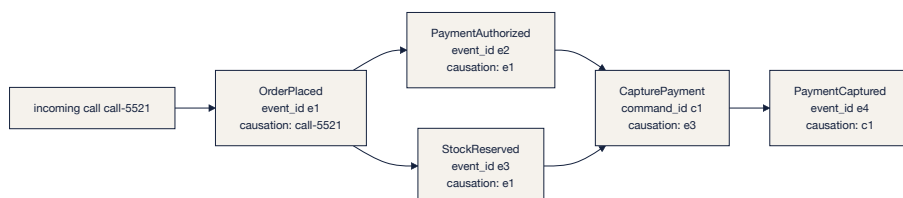
Two placement rules complete the contract. The envelope lives in the message body, and broker headers only mirror what the body already says, because headers are the part that gets dropped when a message is copied, dead-lettered or bridged to another broker. And the content type is declared once, in the schema, not guessed per message. What happens to fields a consumer does not know, and how formats differ in decoding them, is a compatibility question and belongs to Chapter 12.

Correlation and causation

The two identifiers that most often confuse readers are `correlation_id` and `causation_id`. They answer different questions.

The `correlation_id` is shared by every event in one business flow. When a buyer places order 81724, everything that follows carries `correlation_id: "order-81724"`: the authorization, the reservation, the capture, the shipment. You search by it when a buyer writes in and asks what happened.

The `causation_id` points at the one message that directly caused this one. It is the `event_id` of an event, or the `command_id` of a command. It does not repeat across the flow. Each hop sets it to the identifier of its immediate trigger, and following the chain backwards reconstructs the path.



Look at the last two steps. `CapturePayment` is triggered by two events, and Orders must pick one for `causation_id`, here the one that completed the condition. The ambiguity is real, and I would write down the rule so that all producers choose in the same way. The `PaymentCaptured` event names the command, not the event before it, which is why the envelope allows both kinds of identifiers in that field.

Every event in the flow shares `correlation_id: "order-81724"`. The first event in a flow sets the correlation identifier, either by deriving it from the aggregate or by carrying one from the incoming request. Later events copy it unchanged.

This book keeps the aggregate-scoped `order-81724` throughout, and that is a choice with a consequence. An aggregate-scoped id is simple, but every flow on that order merges under it: a refund three weeks later shares the id with the original checkout, and the two cannot be told apart. The alternative is a **flow-scoped** id, minted fresh for each flow (`checkout-81724-1`, `refund-81724-1`), with `order_id` kept searchable in the payload. It costs one more thing to generate and to propagate, and it is worth that cost when an aggregate can start a second flow you must tell apart from the first. The rest of this chapter holds either way. Copying is the whole job, and a consumer that forgets to copy it breaks the trail for everything downstream.

```
function buildFollowUpEvent(trigger, eventType, payload):
  // trigger is the message being handled: an event or a command
  return {
    event_id:      newEventId(),
    event_type:    eventType,
    schema_version: currentVersionOf(eventType),
    occurred_at:   now(),
    producer:      thisServiceName,
    correlation_id: trigger.correlation_id, // copied, never
    regenerated
    causation_id:  identifierOf(trigger), // event_id or
    command_id
    partition_key: aggregateIdOf(payload), // this event's
    aggregate, not the trigger's
    payload:       payload
  }
```

The trigger's `partition_key` can be copied only when both messages concern the same aggregate. A follow-up about a payment or a shipment that inherited the order's key would be ordered against the wrong stream (Chapter 9). Parcelly keys almost everything by order, which is why the mistake is hard to spot.

The cost of getting this wrong is paid at the worst time. During an incident, the question is "what happened to this order across five services?" Without a consistent `correlation_id`, an engineer searches five logs by hand. Chapter 13 shows how tracing and reconciliation use these fields, so the work of this chapter is to make sure they are present and correct from the first event.

What does not belong in an event

An event leaves its producer's database and enters a system with different access rules, different retention and many readers. So the design question is also a question about what you are willing to copy into that system.

Personal data. An event with an email address, a phone number or a full delivery address is stored wherever the broker stores it, possibly for the length of the retention window, and read by every consumer. In an event log, deleting one person's data later is awkward, because the log is append-only and consumers have already copied the fields. Some teams respond with encryption schemes or separate topics. I would start earlier: put an identifier in the event, such as `buyer_id`, and let a consumer that has a legitimate need fetch the personal fields from the owning service at that moment, subject to that service's access rules. This leans toward thin events for sensitive fields, and I think the trade is worth it. If a fat event truly needs an address, as Shipping does, then the event should carry it deliberately, to that audience, with the retention period chosen in the knowledge that it is there.

Which fields count as personal data, and what obligations follow, depends on your jurisdiction and your legal advisers. I do not know your constraints. The design habit is the same everywhere: every sensitive field in an event should have a named reason.

Secrets and credentials. Tokens, keys and payout account numbers do not belong in a payload. A message is logged, copied to dead-letter destinations, shown in admin tools and attached to tickets. Assume every payload will be read by a person.

Internal detail. Table names, internal status codes, feature flags, database ids that only make sense inside one service, and values whose meaning depends on the producer's code. `internal_status` in the opening scenario is the example. A field like that is a public promise the moment somebody reads it.

Instructions to specific consumers. A flag such as `skip_email_if_gift` tells Notifications how to behave. The event should state the fact, and the consumer decides what to do with it.

Warning: Anything that enters an event log should be treated as permanent and widely readable. A secret or a personal field written once is copied to dead-letter destinations, backups and replay tools, and removing it everywhere is far harder than leaving it out.

A short checklist helps when reviewing a new event: does each field have a consumer who needs it, is each field something the producer is willing to keep stable, and could the field harm someone if it were read by a person who should not see it?

Granularity

How big is one event? There are two opposite mistakes.

Too fine. Publishing a `FieldChanged` event for every column update produces a stream of noise. Consumers have to reassemble the meaning from many messages, and the order of those messages starts to matter in ways that Chapter 9 warned about. A change that was one business fact, the buyer correcting an address, now arrives as four events that can be seen in a half-applied state.

Too coarse. An `OrderChanged` event that fires on every modification and carries the whole order forces each consumer to diff the new state against the old, and to guess what the business meaning was. It also means every consumer wakes for every change, including those it does not care about.

I would use the business fact as the unit: one event per decision or occurrence that someone in the business would name. A buyer placing an order is one fact. A payment being captured is another. When one transaction produces several facts, publish several events through the same transactional outbox write, so they are committed or rolled back together.

A practical test is to ask if a consumer could react meaningfully to this event alone. If the answer is "only after the next three events arrive", the granularity is too fine. If most consumers ignore most of the payload, or the event name needs "and" to describe it, it may be too coarse and might hide two facts.

Ownership and the public contract

Every event has exactly one owner: the service that owns the aggregate the event describes. Payments owns `PaymentCaptured`. Orders owns `OrderPlaced`. Consumers may read the event, but only the owner defines it. A consumer that wants a different shape asks the owner, and does not publish a variant of its own under the same name.

Treating an event as a **public contract** has practical consequences, and I would write them into the team's working agreements.

- **The owner documents the event:** its name, meaning, when it is emitted, when it is not, each field with its type and meaning, and the guarantees about ordering and partition key.
- **The event has a version and a change process.** Chapter 12 describes how to evolve schemas without breaking consumers. The ground rule belongs here: the owner does not change a published event without knowing the consequences for its readers.
- **Consumers declare themselves.** A short registry of who consumes what turns a silent subscription into a known dependency. It does not need to be fancy. A list in the repository that a pull request has to update is enough to start.
- **Consumers read only what they need.** The tolerant reader rule from Chapter 12 starts here: ignore fields you do not use, so that additive changes do not break you.

- **Emission is part of the behavior.** A service that stops emitting `OrderCancelled` in some code path has changed its public interface as much as one that removes an HTTP endpoint.

The words "contract" and "owner" both carry weight. A contract can be tested. An owner can be paged. If nobody can say who owns an event, nobody can say who is allowed to change it, and the opening scenario repeats.

Trade-offs and costs

Discipline costs speed. Mapping domain events to integration events, naming carefully, writing documentation and keeping a consumer list all slow the first release. The payoff arrives later, as cheaper changes. For a prototype with one producer and one consumer written by the same person, the ceremony may not pay off, and I would keep it light until a second team appears.

Fat events trade coupling to time for coupling to shape. Consumers no longer need the producer to be reachable, but they depend on the fields it copies. A larger payload means a larger contract, more to keep stable and more to protect.

Thin events are cheap to publish and expensive to consume. Every consumer pays a callback, and the producer must serve that read load, which usually peaks at the same time as the event. If the producer's API is slow, the thin event moves the problem from the producer to every consumer.

Careful envelopes cost bytes and attention. Nine fields on every message add up at high volume, and every producer must fill them in correctly. The alternative is to rebuild the missing information during an incident, which costs more. How much this matters depends on your message volume. The usual approach is to keep the envelope small and the payload lean.

Wire-level rules cost flexibility. Strings for ids and money, a fixed timestamp format and an envelope in the body make a polyglot system boring in the good sense. They also mean converting at the edges, and a team

that wants native types in its own language pays that conversion on every message.

A command's expiry costs a clock dependency. `expires_at` protects against stale commands but is judged on the receiver's clock. The margin you add is time a genuinely late command is still accepted.

Anonymized or identifier-only events limit what consumers can do alone. A consumer that must contact the buyer needs a call to the owner or a local copy of the data it has been given a reason to hold. That is friction, and it is chosen friction.

When not to put this much structure around an event. A purely internal event inside one service, consumed by the same codebase and never retained, does not need an envelope, a version or an owner list. Keep the full treatment for events that cross a boundary, and in particular for events that outlive a deploy.

Common mistakes

- **Publishing domain events directly.** It is tempting because the machinery exists, and the result is that internal refactors become breaking changes. Add an explicit mapping to integration events and treat that mapping as the contract.
- **Calling a command an event.** The past tense is dropped or added cosmetically, and the producer ends up depending on a specific recipient without a way to hear a refusal. Send commands to a named recipient and expect an outcome back.
- **Updated events.** `OrderUpdated` says nothing about what changed, so each consumer diffs state and guesses. Name the business fact, and if no name exists, question whether the event is needed.
- **Dumping the whole aggregate into the payload.** It saves one conversation and creates a public contract the size of the database table. Ask each consumer what it does with each field, and cut what nobody uses.
- **Putting personal data and internal fields in because they were at hand.** Fix this at review time with a checklist, because after publication the data is already in storage you do not control.

- **Encoding by language default.** A class name as `event_type`, ids as numbers, money as floats, timestamps with whatever offset the runtime picked. It works until the second language reads it. Write the wire rules down once, as the table above does.
- **Copying the trigger's `partition_key`.** It is right when both messages concern one aggregate and wrong for a follow-up about another. Derive the key from the event's own aggregate.
- **Regenerating `correlation_id` at each hop.** Someone writes a helper that creates a new identifier by default. Copy it from the trigger and test that every consumer does so.
- **Using `occurred_at` as the order.** A timestamp is a clock reading, and clocks disagree. Use the partition key and a version number as Chapter 9 describes.
- **No owner.** An event emitted by "the platform" or by two services under one name has nobody who can approve a change. Assign one owning service per event type.

Key takeaways

- An event states a fact, a command requests an action from a named recipient, and a document carries data; mixing these three leads to hidden coupling and lost refusals, and a command needs a reply path and an optional expiry because a lost one is noticed only by the sender's deadline.
- Domain events describe a service's internal model, while integration events are the deliberately chosen public contract, and an explicit mapping between them keeps refactoring from becoming a breaking change.
- Thin events keep contracts small but reintroduce runtime coupling and callback races, while fat events remove the callback but enlarge the contract, so carry what consumers need to act and nothing they merely might want.
- Name events in the past tense with the subject first, use business language, and avoid `Updated`-style names that hide the fact.
- The envelope exists so that tooling can deduplicate, route, trace and order without reading the payload; its wire encoding (string ids, RFC

3339 UTC timestamps, safe money, language-neutral type names) is what keeps it readable in every language, and `correlation_id` is copied through the flow while `causation_id` points at the one direct trigger.

- Keep personal data, secrets, internal state and consumer-specific instructions out of payloads, because anything in an event log is copied widely and removed with difficulty.
- Every event has one owning service, a documented meaning and a list of known consumers, which is what turns a message into a contract that can be changed on purpose.

An event that is designed well still has to change, because the business will change, and consumers will upgrade at different speeds while old events stay in the log. Chapter 12 turns to schema evolution: how to change a contract that other teams already depend on without breaking them.

Chapter 12 — Schema Evolution

Imagine that Parcelly's Payments team wants to support partial captures. A seller ships half of a two-item order, and the buyer should be charged for that half now and the rest later. The `PaymentCaptured` event carries a single field, `amount_minor`, which until now has always meant "the whole order". The team renames it to `captured_minor` and adds `remaining_minor`, which is cleaner and says what it means. The tests in the Payments repository pass, and the deploy goes out late on a Thursday.

Ledger is not in that repository. It reads `amount_minor` to post a double-entry record, finds nothing, and rejects every new event as malformed. Notifications reads the same field for a receipt and prints an empty amount. A reporting job that replays last month's events now meets two populations in the same topic, old names and new names, and its parser handles only one.

Nothing here is a bug in Payments. The code did what the team asked. The failure is that an event is read by programs the author never meets, and some of them read events written months ago. How do you change a contract whose readers you do not control, while the old writes are still sitting in the log?

The problem, and why it is hard

An event schema is a contract between a producer and every consumer, present and future. A function signature in your own codebase is a contract too, but a compiler checks both sides and you can change both in one commit. An event schema has no compiler between the two sides. Producer and consumers deploy on different days, owned by different teams, and the broker accepts whatever bytes arrive.

Three properties make this harder than ordinary API versioning.

First, **events outlive the code that wrote them**. On a log-based broker, events stay for the configured retention period, and some systems keep them indefinitely. On a queue-based broker the window is shorter, but a backlog or a dead-letter queue (DLQ) can hold messages for days. A consumer deployed today may be asked to read something written under a schema from a year ago.

Second, **replay makes the past a live input**. Replay, defined in Chapter 2, is how you rebuild a projection, backfill a new service or recover after a bug. It hands today's code every historical schema at once. A consumer that handles only the latest shape works until the day someone replays, and then it fails on the oldest events first. Shape is only half of it: replay also applies today's business rules and side effects to old facts, such as sending a receipt again or calling the payment provider again. A consumer rebuilt by replay has to know that it is replaying, through an explicit mode or an offset range, suppress its external effects, and hold rules that are valid for historical facts. Chapter 7 covers the deduplication side.

Third, **you cannot upgrade everyone at once**. Rolling deploys mean old and new versions of the same service run side by side for minutes or hours. A change that is correct once everything has been upgraded is still an incident during the upgrade.

The obvious approach is to treat the schema as an internal detail and fix consumers afterward. That works while there is one consumer and one team. It stops working when the producer does not know how many consumers exist, which is the situation event-driven design is meant to create. Chapter 11 argued that a published event is a public contract. Schema evolution is how you keep that promise over time.

Compatibility, defined

The same words are used loosely in many places, so here are the definitions the book uses.

- **Backward compatibility:** a consumer using the new schema can read events written with the old schema. New readers, old data.
- **Forward compatibility:** a consumer using the old schema can read events written with the new schema. Old readers, new data.
- **Full compatibility:** both at once.

- **Transitive** variants of each check the new schema against every earlier version, not only the latest.

The direction is easy to flip in your head, so I ask a concrete question instead: "if I deploy this change, who breaks?"

Compatibility	Who reads what	Safe deploy order	Protects against
Backward	New reader, old data	Consumers first, then producers	Replay of history, lagging backlogs
Forward	Old reader, new data	Producers first, then consumers	Consumers that cannot be upgraded on demand
Full	Both directions	Any order	Mixed fleets, unknown consumers
None	Neither guaranteed	Coordinated, with a migration plan	Nothing; see the migration section

A caveat on names. Registries and serialization libraries use these words with small differences. Read the definitions in the tool you use before relying on a setting, and check which variant your registry applies by default: a non-transitive rule is not enough for a replayable log. Test with real sample events from each version.

For a log with replay, backward compatibility against the whole history is the property you need, because a replaying consumer can meet any version ever published. For a queue with short retention, checking against the previous version may be enough. That is a retention decision as much as a schema decision.

Safe and breaking changes

A change is **additive** when old consumers can ignore it and new consumers can cope with its absence. A change is **breaking** when some reader, old or new, can no longer interpret some event. The verdicts below assume readers behave as described in the tolerant reader section.

Change on <code>PaymentCaptured</code>	Usual verdict	Why
Add an optional field (<code>payment_method</code>)	Safe	Old readers ignore it; new readers handle absence

Change on <code>PaymentCaptured</code>	Usual verdict	Why
Add a required field with no default	Breaking	New readers cannot interpret old events
Remove an optional field	Safe only if no one reads it	Readers that depend on it see nothing
Remove a required field	Breaking	Old readers expect it
Rename a field	Breaking	Reads as a removal plus an addition; a few formats can resolve a rename with aliases
Change a type (<code>int</code> to <code>string</code>)	Breaking	Parsers fail or silently coerce
Widen a numeric type	Depends on the format	Some formats promote, others do not
Change the meaning of a field	Breaking, and the worst kind	Nothing fails; results are wrong
Add a value to an enumeration	Risky	Old readers may reject or mishandle it

Two rows deserve a longer look.

Changing meaning is the dangerous change

Renaming a field breaks loudly. Ledger fails on the first event, an alert fires, and someone reads the stack trace. Changing what a field means, while leaving name and type alone, breaks quietly. Imagine Payments decides that `amount_minor` will now mean "amount captured in this event" instead of "total order amount". Every parser still succeeds. Ledger posts partial amounts as if they were totals, and the error surfaces weeks later in a reconciliation (Chapter 13) as a gap nobody can explain.

No schema tool catches this, because the schema is unchanged. Only a human reviewing the change does. My rule: if the meaning changes, it is a new field or a new event type, never an edit to the old one.

Enumerations are a hidden contract

Adding an enumeration value is additive in the schema and often not in the consumers. A consumer that maps `payment_method` to a handling branch, with an `else` branch that assumes "card", will mishandle a new

`"bank_transfer"` without any error. Producers can document that enumerations are open. Consumers can treat unknown values as unknown and route them to a visible error or a safe default, instead of folding them into a known case.

Schema registries and formats

A **schema registry** is a service that stores the schemas used by a system, assigns them identifiers, and can check a proposed schema against earlier ones. It does two jobs worth separating.

1. **Lookup.** A message carries, or points to, the identifier of the schema it was written with. A consumer fetches that schema and decodes the bytes. This is what makes compact binary formats workable: field names do not travel with every message.
2. **Gatekeeping.** Before a producer registers a new schema, the registry compares it to earlier versions under a chosen compatibility rule and rejects it if it breaks the rule.

The second job is the one I would defend hardest. Rules that live in a review checklist get skipped on the day the team is busy. A check in the deploy pipeline does not get busy.

Subjects, topics and event types

A registry checks compatibility per **subject**, and whether a subject is a topic or an event type decides whether several event types can share a topic. Sharing is what per-aggregate ordering across event types needs (Chapter 9): the `PaymentAuthorized` and `PaymentCaptured` of one payment stay in order only if they travel on the same partition of the same topic. The Parcelly examples assume one topic per aggregate type, such as `payments.payment-events`, carrying several event types keyed by aggregate id, with compatibility configured per event type rather than per topic. The cost is that consumers filter by `event_type`, and that a versioned topic (Strategy 3) moves every event type of that aggregate at once. A topic per event type is simpler to configure and gives up ordering across types.

Three families of format

The book is format-agnostic, so this is orientation, not a product comparison. Avro, Protobuf and JSON Schema differ in what the data looks like on the wire, which shapes what is safe to change.

Family	On the wire	Fields identified by	Practical consequence	Trap
Avro-style	Compact binary, no field names	Name, resolved between writer and reader schema	Reader needs the writer's schema; defaults carry compatibility	Aliases let a reader resolve a re-named field, so a rename is not always breaking. An absent field is filled with the reader's default, so absence is invisible
Protobuf-style	Compact binary	Field number	Renaming is cheap; reusing a number is dangerous	Cheap only on the binary wire: ProtoJSON and generated code use names, so a rename breaks source and JSON. A proto3 scalar without explicit presence decodes an absent field as 0
JSON Schema	Text JSON	Name; schema optional at runtime	Easy to debug; compatibility is a convention unless enforced	With a closed model (<code>additionalProperties: false</code>) adding a field breaks old readers, and registries judge open and closed models differently

Three ideas carry over to all of them.

- **Defaults make additions safe, if the default is honest.** A new field with a defined default lets a new reader process an old event. Without one, an addition becomes a break. For a field the reader must be able to detect as missing, such as an amount, the default has to be "no value": an Avro union of `null` and the type with a `null` default, or proto3 `optional` or a wrapper type. A default of 0 decodes an old event as a capture of zero, and the reader cannot tell the difference. The reader then checks presence, not value.
- **Identity should not be reused.** Whether a field is identified by name or number, giving an old identity a new meaning is a break. Retire identifiers; do not recycle them.
- **Unknown fields should be tolerated.** Whether a decoder keeps, ignores or rejects fields it does not know depends on the format and its

settings. Find out before you design around it.

If you run JSON without a schema language, you still get most of the benefit by keeping schema files in version control, validating in the producer's tests, and keeping real example events per version. The registry adds central lookup and enforcement. It does not add the thinking.

Where the version lives

The standard envelope carries `schema_version` as an integer next to `event_type`, and a consumer reads the pair to know which shape to expect. Keeping it in the envelope means a router, a metrics counter or a DLQ tool can see it without decoding the payload. "How many `schema_version: 1` events are still arriving?" becomes a cheap query instead of a project.

Two numbers are easy to confuse here, so this chapter gives them separate names. The **breaking version** is the envelope's `schema_version`. It moves only for a change that an old reader cannot handle: a rename, a removal, a type change or a change of meaning. The **registry revision** is the identifier the registry assigns to each registered schema, and it moves on every registered change, additive ones included. A consumer rejects an event whose breaking version it does not know. It reads an event with a newer registry revision but a known breaking version as the latest shape it knows, and ignores what it does not recognize. Neither is the aggregate version of Chapter 4, which counts changes to one order and has nothing to do with the shape of the payload.

If every additive change bumped `schema_version`, each consumer not yet upgraded would reject every new event, and forward compatibility would be gone. The price of the split is two numbers to keep straight. One example of the same line is BabelQueue (babelqueue.com), an open-source, language-independent envelope standard that freezes its envelope at `schema_version 1`, allows new fields only as optional and ignorable, requires a version bump for a rename, removal or type change, has consumers reject a version they do not know, and keeps unknown fields from breaking processing and preserved when the message is published again.

An event as Payments publishes it during the dual-publish step of the migration below:

```

{
  event_id:      "9c04b6e2-...",
  event_type:    "PaymentCaptured",
  schema_version: 1,
  occurred_at:   "2026-10-05T11:20:44Z",
  producer:      "payments",
  correlation_id: "order-90412",
  causation_id:  "5ad1e07c-...",
  partition_key: "order-90412",
  payload: {
    order_id:      "order-90412",
    amount_minor:  4990,
    captured_minor: 4990,
    currency:       "EUR"
  }
}

```

Compatibility across languages

A schema can be perfectly compatible by the registry's rules and still break between two languages, because the rules describe structure and the break is in how a runtime reads it. Chapter 11 fixed the encoding of the envelope. These are the traps in the payload and in what a consumer does with fields it does not know.

Unknown fields are lost on re-emit. Suppose a Node consumer in Notifications deserializes each `PaymentCaptured` into a class with `order_id`, `captured_minor` and `currency`, and copies the event to a partner-facing topic. When Payments later adds `payment_method`, the copy silently drops it. Nothing fails. The field is never there. Protobuf binary decoders generally keep unknown fields; many typed JSON mappers do not. Rule: a hop that forwards a message preserves unknown fields, by carrying the raw payload or a generic map, or it publishes a new event of its own under its own contract.

Numbers. `captured_minor` fits comfortably in any language's integer. An `int64` identifier or counter does not fit in a JavaScript JSON number beyond $2^{53} - 1$, and the consumer rounds it without error. Rule: document the range of every numeric field, and send anything that can exceed it, and any decimal, as a string.

Timestamps. Avro's `timestamp-millis` and `timestamp-micros` hold the same kind of value at different scales. A writer in microseconds read as milliseconds yields a date tens of thousands of years away, and still parses. Rule: fix the unit in the schema and never change it; a unit change is a breaking version.

Enumerations. An unknown value is handled differently by generated code: some keep it, some map it to a default, some throw. Rule: treat every enumeration as open and decide in the consumer what unknown means.

Null and absent. Many decoders do not distinguish them, and some fill the gap with a default. Rule: for fields whose absence matters, use the presence-aware forms from the defaults bullet above.

None of these needs a particular version of a particular library. All of them need a person to test with a consumer in each language Parcelly runs, which is a cost the contract tests of Chapter 15 can carry.

Versioning strategies

There are three ways to introduce a change, from cheap to expensive. I would try them in order and climb only when the lower rung cannot hold the change.

Strategy 1: add an optional field

If the change is something new that old readers may ignore, add an optional field. Adding `payment_method` to `PaymentCaptured` is the model case. It registers a new registry revision, and the compatibility rule decides whether the registry accepts it. It does not change `schema_version`, because no old reader loses anything. Bumping the breaking version for an additive change would make every consumer not yet upgraded reject the new events, which defeats the point of adding the field in a compatible way.

This strategy cannot express a rename, a type change, a removal of something required or a change of meaning. For those, move up.

Strategy 2: introduce a new event type

When the meaning changes, say so with a new name. Suppose Parcelly treats partial capture as a different business fact from a full capture. The honest model is a new event, named by Chapter 11's rules. This chapter introduces `PaymentPartiallyCaptured` as an additional event.

The old event keeps its meaning and its consumers keep working. The cost is that both events now exist, and a consumer that wants "all captures" must subscribe to both. This is the right trade when the new fact is conceptually different, and the wrong one when it is the same fact with a better shape. That second case is a representation problem, and an upcaster fits it better.

Strategy 3: a versioned topic

When a breaking change touches the whole stream, put the new shape on a new topic, for example `payments.payment-events.v2`. Consumers migrate on their own schedule, and the old topic is retired when its last reader leaves. With the layout described above, the new topic carries every event type of the aggregate.

This is the clearest option and the most expensive. Partition keys, ordering and offsets of the two topics are unrelated, so a consumer moving from one to the other must decide where to start reading and how to avoid processing a fact twice. Chapter 9's ordering rules apply to each topic separately, and nothing orders events across them. On a queue-based broker the equivalent is a second queue with its own bindings, dead-letter handling and alerts. I would reach for a versioned topic only for a real break that an upcaster cannot bridge, and only when I can name who will delete the old one.

Upcasting and the tolerant reader

Two techniques let consumers cope with several versions without forcing the producer to freeze.

Upcasting

An **upcaster** converts an event written under an old schema into the current schema before business logic sees it. Business code is written against one shape, and the conversion for each historical version lives in one place.

```
function upcastPaymentCaptured(event):
    p = event.payload
    if event.schema_version == 1:
        // during the migration a v1 event carries either field, or
        both

        // check presence, not value
        if hasField(p, "captured_minor"):
            captured = p.captured_minor
        else:
            // v1 only ever described full captures, so the
            // order amount is the captured amount; may be absent
            captured = p.amount_minor
        return { order_id: p.order_id, captured_minor: captured,
currency: p.currency }
    else if event.schema_version == 2:
        // v2 carries captured_minor only
        return { order_id: p.order_id, captured_minor:
p.captured_minor, currency: p.currency }
    else:
        // an unknown newer breaking version must not be guessed at
        raise UnsupportedSchemaVersion
```

The comment on the version 1 branch carries the real work. An upcaster encodes a statement about the past, and that statement has to be true. If some old events were partial in practice, the upcaster quietly reports a partial amount as the whole capture. Test upcasters with real archived events, not invented ones.

`UnsupportedSchemaVersion` needs a classification of its own. For the current code it is permanent, since retrying changes nothing, but a consumer deploy cures it. So it is held for redrive after the consumer is upgraded (Chapter 8), not sent to the parking lot, where it would wait for a human who has nothing to decide.

The consumer side, inside the handler before the business logic, is the simplest place to start. Rewriting history into a new topic is useful when many consumers would otherwise need the same upcaster. Rewriting stored events in place is a different decision: event-sourced systems often refuse

to, because the log is evidence. I would respect that unless there is a legal or security reason, and then I would run it as a data migration with a plan.

The tolerant reader

A **tolerant reader** takes only what it needs from an event and ignores the rest. It reads `order_id` and `captured_minor` and does not care that there is also a `payment_method`. It is liberal in what it accepts and strict about what it depends on.

```
function readPaymentForLedger(event):
    payload = upcastPaymentCaptured(event)

    if not hasField(payload, "order_id") or
        not hasField(payload, "captured_minor") or
        not hasField(payload, "currency"):
        // a missing dependency will fail on every attempt,
        // so it is a poison message, not a retry case
        raise InvalidEvent

    return {
        order_id: payload.order_id,
        amount:   payload.captured_minor,
        currency: payload.currency
    }
```

Tolerance has two sides. Ignoring unknown fields is safe when producers add things. Quietly substituting zero for a missing amount is dangerous. The reader above refuses to continue when a field it depends on is absent, and it asks about presence, not value. That is only honest if the decoder can tell an absent field from a zero, which is why the defaults above matter. That refusal is a classification: the event can never succeed, so it takes the permanent-failure path from Chapter 6 and ends as a dead letter, as Chapter 8 describes.

Note: Tolerance is an agreement between both sides. The producer promises to add and not to change meaning. The consumer promises to ignore what it does not understand and to refuse what it cannot do safely. If either side breaks its half, the other cannot save the system.

The registry checks the producer before it publishes. The tolerant reader protects the consumer after something slips through. I would not rely on one in place of the other.

Migrating a breaking change

Sometimes a break cannot be avoided: the old shape is wrong, a regulation forces a field out, or two services must be reshaped together. The answer is to turn one breaking deploy into several non-breaking ones. This is the **expand and contract** pattern applied to events: expand the contract to support both shapes, migrate every reader, then contract by removing the old shape.

A migration plan for PaymentCaptured

Say Parcelly really must rename `amount_minor` to `captured_minor`, and nothing else about the event changes. The value keeps its meaning: the amount captured for the whole order. This is a pure rename, and the plan below is only safe because of that. Partial captures are a change of meaning and are handled separately, after the steps.

1. **Register the addition.** The new registry revision keeps `amount_minor` and adds `captured_minor` as an optional field whose default is "no value", not 0. Under a backward-compatibility rule the registry accepts it. `schema_version` stays 1, because no old reader loses anything.
2. **Upgrade consumers first.** Every consumer learns to read `captured_minor` and to fall back to `amount_minor` when it is absent. It also learns to read breaking version 2, which does not exist yet, so that the bump in step 5 finds nobody who rejects it. New readers exist before new data does.
3. **Dual-publish.** Payments writes both fields in each event, still at `schema_version` 1, or, if the change needs a new event type or topic, publishes the old and new form of the same fact. This is not the dual write of Chapter 4: both messages come from the same transactional outbox transaction, so the usual guarantee holds.
4. **Observe.** Measure who still reads the old field or old topic. Producer metrics on `schema_version` show what is still written; consumer-group activity on the old topic, the consumer list from Chapter 11 and

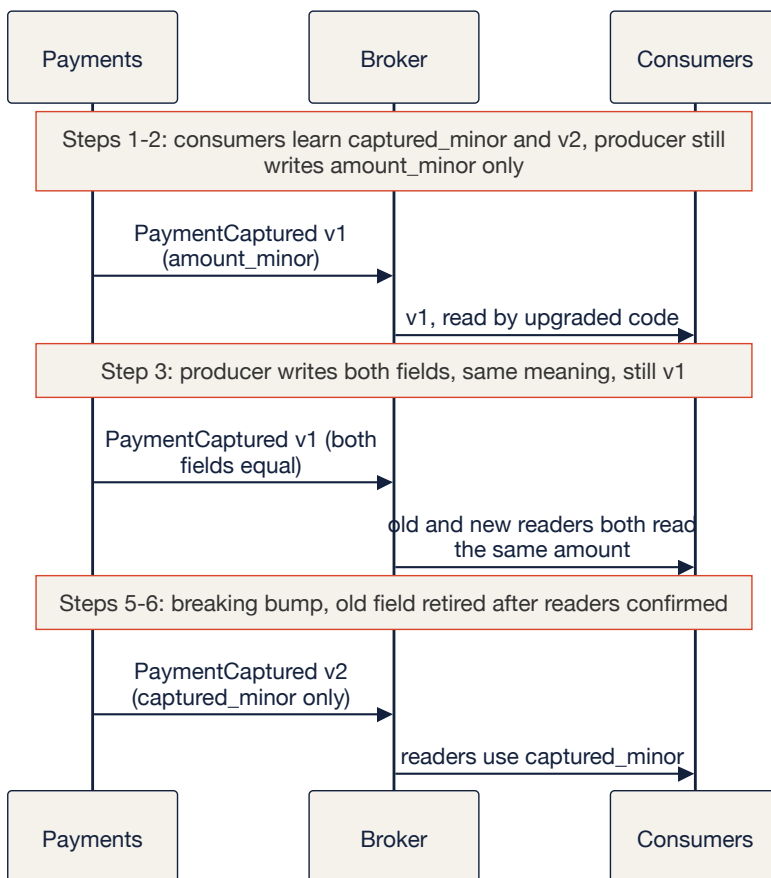
the contract tests of Chapter 15 show who still reads it. A consumer that never moves is a conversation to have, not a number to ignore.

5. **Stop writing the old shape.** Payments publishes `captured_minor` only and bumps `schema_version` to 2, because this is the one step an old reader cannot survive. A consumer that missed step 2 rejects the event, and it goes to the DLQ for redrive after that consumer is upgraded (Chapter 8). Step 4 exists to keep that list empty. `amount_minor` stays in the schema as deprecated until the oldest event carrying it has left retention.

6. **Contract.** Remove the old field and retire the old topic if there was one. Remove the fallback from consumers only once the oldest event without `captured_minor` has left retention, or move it into a permanent upcaster.

Partial captures stay out of this plan. Once a capture can be partial, `amount_minor` has no honest value. If it carries the captured amount, its meaning has changed in place, and a reader that skipped step 2, whether unknown or rolled back, records the partial amount as if it were the order total, with no error. If it carries the order total, the same reader records too much. The registry passes both versions, because the structure is unchanged. That is the lesson: a structural compatibility check cannot see a change of meaning. Put the new meaning in a new field, a new event type (Strategy 2, `PaymentPartiallyCaptured`) or a new major version on a new topic. Then do not publish partial captures at all until every reader has moved to the new form, and publish them only in the new form afterward.

The order is the heart of the plan. For a backward-compatible change, consumers go first; for a forward-compatible one, producers may go first. When you cannot tell which applies, run the plan as written and keep both fields for a while.



Dual-publishing across topics

When the plan uses a versioned topic, step 3 publishes the fact twice, once to each topic. A consumer that reads both must not process the same business fact twice. I would give the copies distinct `event_id` values, because they are distinct messages, and carry the original's id in a documented field. Whichever you pick, the consumer's idempotency key for that fact must be defined on purpose. Chapter 7 covers the keys; the point here is that a migration is exactly when idempotency assumptions get tested.

Warning: During dual-publishing, never let a consumer read both the old and the new stream and apply both. A consumer with two sources for one fact needs one deduplication rule, written down, or it will double-post a ledger entry.

Rollbacks and history

Rolling back is part of the plan. If version 2 events are already published, rolling the producer back does not remove them from the log. A consumer rolled back to code that cannot read version 2 now fails on events that already exist. So consumers must be able to read version 2 before any producer writes it, and a consumer rollback target must be able to read every version in retention, version 2 included. Code that reads only version 1 is a rollback target to avoid, even if it was the last release before the change.

Backward compatibility handles old events without touching them. If you want to delete the fallback code, you have three choices: keep the upcaster until old events expire, rewrite the stored events into a new topic, or snapshot state elsewhere and let history lapse. Each costs something: permanent code, a migration you must trust, or the ability to replay. I do not know which your system should pick without knowing how often you replay and how long your regulators expect you to keep records.

Deprecating events

Events are easier to add than to remove, because a consumer you do not know about may depend on any of them. A reasonable lifecycle has four stages.

1. **Announce.** Mark the event deprecated in the registry or event catalog, name the replacement, give a retirement date and a contact.
2. **Measure.** Find out who still reads it: consumer group membership on a log, bindings on a queue, registry lookups. Treat the list as evidence, not proof, because a monthly batch job may be absent today and present next week.
3. **Stop producing.** After the date, stop publishing.
4. **Retire.** Remove the schema and the topic or queue once retention has aged out the last event and the readers have gone.

Two cautions. A deprecation without a deadline never ends, so put the date in writing and keep it. And a deprecated event that is replayed after retirement will fail, so decide what replay should do for the affected period before you retire.

Ownership matters. The producer's team owns the event and its depreciation, as Chapter 11 described. The consumers own their migrations. A policy that lets the producer decide alone, or lets consumers refuse indefinitely, will fail. The useful policy is notice from the producer and a date met by consumers.

Trade-offs and costs

- **A registry is another service to run.** It sits in the path of every producer's deploy and, depending on setup, of consumers' decoding. It needs availability, access control and an owner, and clients need a plan for when it is unreachable.
- **Compatibility rules slow producers down, deliberately.** A team that could have renamed a field in an afternoon now runs a six-step migration. That is the price of independent deploys.
- **Upcasters accumulate.** Each historical version adds a branch and a test, and the code does not get simpler until old events leave retention. Indefinite retention means indefinite upcasters.
- **Tolerant readers hide mistakes.** A consumer that ignores unknown fields also fails to notice when a producer starts sending something it should care about. Tolerance without monitoring turns loud failures into silent drift.
- **Dual-publishing multiplies traffic and deduplication work** for as long as the migration lasts.
- **Two version numbers instead of one.** The breaking version and the registry revision keep additive changes from failing consumers, and cost a team the effort of telling them apart in dashboards and conversations.
- **Version numbers are not documentation.** `schema_version: 4` says something changed, not what or who is affected. A change log kept with the schema is worth more than the number.
- **Cross-language rules cost convenience.** Strings for large numbers, raw payload preservation and fixed timestamp units mean extra conversion code in each language, and a test per language.

When not to use these patterns

Do not build the full machinery for an event with one producer, one consumer and one team that deploys both together. A coordinated deploy is cheaper, and the risk you would insure against does not exist. Revisit when a second consumer appears, especially one owned by another team.

Do not treat a registry's strict mode as a substitute for design. A sequence of safe additions can leave an event with 40 optional fields and no clear purpose. If an event keeps growing, it is probably two events, and Chapter 11's granularity rules apply.

Do not create a versioned topic for every change. A system that does ends up with many half-retired streams and nobody who knows which is current.

Common mistakes

- **Treating a rename as a small change.** It is tempting because the new name is cleaner and the tests pass. To every reader it is a removal plus an addition. Add the new name beside the old, migrate readers, then remove the old.
- **Changing the meaning of a field in place.** Name and type stay, so no tool objects. Give the new meaning a new field or event type.
- **Adding a required field with no default.** It feels like an addition, but old events lack it. Make it optional with a default, or bring it in with a new version and an upcaster.
- **Testing compatibility only against the latest version.** Replay reads everything in retention. Check against every version still there, and keep sample events from each in consumer tests (Chapter 15).
- **Upgrading producers before consumers.** The producer's team owns the change and is ready first. For a backward-compatible change, consumers go first. Confirm the order before the deploy.
- **Failing loudly on unknown fields.** A strict parser turns every additive change into an incident. Ignore what you do not use; reject only what you depend on.
- **Silently defaulting a missing field.** Substituting zero for a missing amount hides a defect, and a decoder that fills in 0 hides it before

your code runs. Use a "no value" default for such fields, check presence, and raise an error so the failure path holds the event for a human.

- **Bumping `schema_version` on every change.** Each additive change then becomes a hard failure for every consumer not yet upgraded. Bump the breaking version only for changes an old reader cannot handle, and let the registry revision count the rest.
- **Forwarding through a typed class.** The copy drops every field the class does not declare, and nothing complains. Preserve unknown fields on re-emit, or publish a new event under your own contract.
- **Reusing a field number or topic name.** A recycled number reads old data as the new field, and a reused topic mixes two shapes. Retire identifiers.
- **Deprecating without a date or an owner.** The event lingers and the next team is afraid to remove it. Give every deprecation a date, a replacement and a person who answers questions.
- **Rolling back a consumer without checking the schema.** Verify that the rollback target can read every version in retention.

Key takeaways

- An event schema is a contract with readers you do not control and with data written long ago; backward compatibility means new readers can read old data, forward means old readers can read new data, and a log with replay needs backward compatibility against every version still in retention.
- Adding an optional field is safe, while renames, type changes, new required fields and changes of meaning are breaking, and a change of meaning is the worst because nothing fails loudly; only a breaking change moves `schema_version`, and the registry revision counts the rest.
- A schema registry earns its place by enforcing compatibility in the deploy pipeline, but it checks structure and not meaning, so a human still reviews what a field is supposed to say.
- Prefer the cheapest versioning strategy that holds the change: an optional field first, a new event type when the meaning is a different

fact, and a versioned topic only when the whole stream must break.

- Upcasters and tolerant readers let consumers cope with several versions, provided the upcaster encodes a true statement about the past and the reader checks presence, not value, and refuses what it cannot safely interpret.
- Across languages the quiet breaks are lost unknown fields, large integers, timestamp units, closed enums and null versus absent, so preserve what you do not know and write the representation down.
- Migrate a breaking change as a sequence of non-breaking ones, with the bump at the one step an old reader cannot survive, and plan the rollbacks in the same document.

A contract that evolves safely is only reassuring if you can see what is flowing through the system, which is where Chapter 13 begins.

PART V

Operating and Proving

Chapter 13 — Observability and Reconciliation

Imagine a Thursday morning at Parcelly. Support forwards a ticket: a buyer paid for an order two days ago, the card statement shows the charge, and the order page still says "Awaiting confirmation". Support opens the Orders admin screen and sees `PAYMENT_PENDING`. They ask the Payments team, who find a `PaymentCaptured` row in their database and a clean outbox. They ask the Orders team, whose consumer logs show no error for that order. Everyone's dashboard is green.

Three engineers now search logs in three services by order number. One log line uses `order-81724`, another uses `81724`, and the Payments relay logs only its internal row id. After an hour they find the event in the Payments outbox, marked published. After another hour they learn that the broker topic feeding Orders had a consumer group that fell behind during a deploy. The broker's retention deleted a stretch of messages the group had not read, and the group's offset-reset policy was set to jump to the latest position, so it resumed after the gap without raising an error. The event existed, was published, and was never read.

Nothing here was a code bug in the sense a reviewer would flag. The system did what it was built to do, and nobody could see it doing it. Worse, the ticket exists because a buyer noticed before any alarm did, and nobody knows how many other orders look the same. This chapter asks two questions. What does a team need to see to answer "where is this order?" in minutes, and how does a team find the orders nobody has complained about yet?

The problem, and why it is hard

In a synchronous system, a failure usually arrives with a stack trace attached to a request. In an event-driven system, the work of one business flow is spread across several services, several databases, a broker and a

number of moments in time. No single process holds the whole story, and no single log contains it.

Three properties make this hard. First, **the hops are decoupled in time**. The producer finishes and forgets, and the consumer may run seconds or hours later, in a different process, with none of the producer's call stack. Tools that follow a request through nested calls lose it at the first message.

Second, **the worst failures are silent**. Chapter 3 described unread messages that a broker deletes when a consumer lags beyond retention. Chapter 8 described messages that were processed incorrectly without raising an error. In both cases nothing fails, so nothing alerts. A system can only notice an event that never arrived by knowing that it should have.

Third, **healthy-looking components can add up to a broken flow**. The relay is running, the broker is up, the consumer reports no errors, and an order is still stuck. Component health answers "is this process alive?" and the business asks "did this order complete?". Those are different questions, and the gap between them is where this chapter works.

Chapter 3 described a failure's semantics as three questions: what can fail, how you would observe it, and which guarantee survives it. This chapter is about the second question, and about what to do when the answer turns out to be "you would not".

The approach has two halves. Observability gives you the means to follow a flow and to see pressure building. **Reconciliation**, comparing two systems' views of the truth and repairing the differences, gives you a way to find what no signal reported. A team that has only the first half sees the loud failures. A team that has only the second finds problems days late. I would build both.

Carrying context across asynchronous hops

Every event already carries two identifiers from Chapter 11: `correlation_id`, shared by every event in a business flow, and `causation_id`, which points at the direct trigger. These are the backbone of the story. Distributed tracing adds a second thread, which records timing and the parent-child relationship between units of work.

A **trace** is a tree of timed spans, each one a unit of work in one process. Inside a process the tracing library keeps the current span in ambient state. Across a message, that state disappears unless you put it in the message. The standard approach is to serialize the trace context, a trace id, a span id and a sampling flag, into message headers. W3C Trace Context defines a common format for this, with two headers, `traceparent` and `tracestate`, and most tracing libraries follow it. Header capacity and propagation differ by broker and by bridge: some limit the number or size of headers, and some bridges drop them, so check the path your messages actually take.

Where the context is captured matters more than it first appears. Chapter 4 put a `headers` column in the outbox for exactly this. The relay runs later, in a different process, with no ambient span, so a relay that injects the context at publish time would attach the relay's own trace instead of the one that placed the order. The context must be captured when the outbox row is written, inside the producing transaction.

```
function writeOutboxEvent(event):
  // runs inside the caller's transaction
  headers = {}
  injectTraceContext(currentSpan(), headers)
  headers.correlation_id = event.correlation_id
  outboxInsert(event, headers)

function relayPublish(row):
  // the relay forwards stored headers verbatim and never
  // recomputes them: a republication must carry the original
  context
  publish(topicFor(row.event_type), row.event, row.headers)
```

On the consumer side, there are two ways to relate the new span to the old one. A consumer span can be a child of the producer span, which draws one tree. Or it can start a new trace and hold a **link** to the producer's context. The right choice depends on the case.

Case	Relationship	Why	What it costs
Single message, consumed promptly	Child of the producer span	One waterfall from request to effect	If delivery is late, the trace stays open for the whole delay and the durations mislead

Case	Relationship	Why	What it costs
Batch or fan-in	New trace, one link per producer	A span has only one parent, so a batch consumer cannot be a child of twelve producers	The backend must support links, and someone has to navigate them
Fan-out	One producer span, one child consumer span per subscriber	The tree stays a tree	The trace grows with every subscriber added
Replay, redrive or late delivery	New trace with a link	An hours-old parent would stretch the original trace	The sampling decision is made again, so a sampled producer trace can link to an unsampled consumer trace

I would still default to links, because lateness and redelivery are the normal case this book plans for. The price is the single end-to-end waterfall that most tracing UIs draw for parent-child. For a consumer that always runs within moments of its producer, parent-child is the better choice and I would take it.

```
function startConsumerSpan(message):
  producerContext = extractTraceContext(message.headers)
  span = newSpan("consume " + message.event_type)
  if producerContext exists:
    span.addLink(producerContext)
  span.setAttribute("event_id", message.event_id)
  span.setAttribute("correlation_id", message.correlation_id)
  // from a broker attribute where one exists; otherwise from a
  header
  // that the retry path increments on every republish
  span.setAttribute("delivery_attempt", deliveryAttempt(message))
  return span
```

Notice that the span records `event_id` and the delivery attempt. Log-based brokers keep no delivery count, and only some queue-based brokers expose one, so decide where the number comes from before you rely on it. When a message is redelivered, you get a second span with the same `event_id`. The pair tells you that the message was delivered twice, and the `outcome` field described below tells you whether it was applied twice.

Tracing has limits, and I would be careful not to lean on it for the guarantee. Backends sample, so the trace for the one order you care about may not exist. Context headers can be dropped by a bridge, a dead-letter copy or a hand-written producer. One way to avoid depending on headers is to carry the trace identifier in the body: the BabelQueue envelope (babelqueue.com), an open-source, language-neutral standard, carries a `trace_id` in the message body that the first producer mints, later producers copy unchanged, and that survives dead-lettering, so the trail holds even when a hop drops headers. A trace that ends at an outbox row never reaches a consumer that has not read the row. For this reason the identifiers in the envelope matter more than the tracing library does: `correlation_id` must appear in every log line and every database row that handles the event, and then it works even when tracing is off.



Read the diagram for the thing that does not change: the `correlation_id` stays the same at every hop, while each service stores a fresh trace context in its own outbox.

Reconstructing the story of one order

With identifiers in place, the question "where is order 81724?" becomes a query. The cheapest implementation is a convention rather than a product: every service writes one structured log line per event handled and per event published, with `correlation_id`, `event_id`, `event_type`, `causation_id`, outcome and timestamp as fields. Searching those logs for `correlation_id = "order-81724"` gives you every hop in every service.

```
log {
  at: "2026-03-14T09:41:09Z", service: "orders",
  action: "consumed", outcome: "applied",
  event_type: "PaymentCaptured", event_id: "3f2a9c1e-...",
  occurred_at: "2026-03-14T09:41:07Z",
  correlation_id: "order-81724", causation_id: "b81d77f0-...",
  delivery_attempt: 1, lag_ms: 1840
}
```

The `outcome` field deserves attention. Use a small closed set: `applied`, `duplicate_skipped`, `stale_ignored`, `failed_retry` and `dead_lettered`. Chapter 9 argued that an ignored stale event deserves a trace, and Chapter 7 covers the inbox check that produces `duplicate_skipped`. Once those outcomes are fields, "how many events did this consumer drop as stale last hour?" is a count instead of an investigation.

Sorting by `occurred_at` gives a timeline, and following `causation_id` gives a tree: `OrderPlaced` caused `PaymentAuthorized`, which with `StockReserved` caused `CapturePayment`, and so on. In the scenario from the opening, the tree would have shown `PaymentCaptured` published by Payments and no `consumed` line in Orders. The first missing hop is the place to look. That is more useful than any single log line: a gap in the chain is evidence that something was lost, where a long chain only shows that the flow progressed.

If you operate a saga from Chapter 10, its `saga_instance` row is a second source. It answers "what does the orchestrator believe the state is?", and the logs answer "what actually happened?". When they disagree, you have found something worth a ticket.

I would also keep an **event journal** for the most important flows: an append-only record of event envelopes, each with a hash of its payload instead of the payload itself, written by each service next to its inbox. It costs storage and some write load, and in return you can answer questions after the broker has forgotten the events, including whether a rebuilt event still matches the original (see the repair section below). Keep the retention long enough for your slowest reconciliation to run.

Metrics that matter

Metrics tell you about pressure before any single message goes wrong. The list of useful ones in an event-driven system is short, and most of them have already been hinted at in earlier chapters. I would start with these.

Signal	What it measures	Why it matters
Consumer lag (messages)	How far behind the head the consumer is	Backlog size; useful for capacity

Signal	What it measures	Why it matters
Message age, consumer side	Time since the oldest unprocessed message was produced	What the business experiences
Outbox age	Age of the oldest unpublished row	Relay health (Chapter 5)
End-to-end latency	Time from the flow's first <code>occurred_at</code> to the effect in the final consumer	The flow's real deadline
Retry rate and attempt counts	Share of messages needing more than one attempt	Degradation hidden by retries (Chapter 6)
DLQ arrivals and age	New dead letters, and the oldest unreviewed one	Failures needing a human (Chapter 8)
Ignored and duplicate counts	Stale events dropped, duplicates skipped	Ordering and redelivery pressure
Drain rate and time to catch up	Processing rate against arrival rate; lag divided by the net drain rate	Whether a backlog clears before its deadline or the retention limit (Chapter 6)

Lag, and why message age is better

Consumer lag in messages is the number most dashboards show first, and it is the easiest to misread. A lag of 10,000 messages means little on its own: if the consumer drains 5,000 a second, it is two seconds behind, and if it drains 10 a second, it is a quarter of an hour. Lag in messages answers a capacity question. Lag in time answers the business question.

Message age is the time since the oldest message that is still waiting was produced. Some queue-based brokers expose the age of the oldest message directly; on the others, derive it from a timestamp on the oldest waiting message. On a log-based broker, derive it from the timestamp of the record at the consumer's committed offset. In both cases the number is in seconds, so it can be compared to a deadline that the business chose.

The timestamp you subtract from matters, because the candidates measure different things. The broker's append time measures how long a message waited at the broker, which is the health of the consumer.

`occurred_at` measures the time since the fact happened, including the time a row sat in the outbox waiting for the relay, which is the business's view of delay. The producer's create-time depends on the setup: depending on configuration a log broker stores either that or the append time, and a

relay that republishes an old row gets a fresh create-time unless it sets one explicitly. I would use append time for the consumer-age gauge and `occurred_at` for end-to-end latency. Using `occurred_at` for the gauge makes a healthy consumer look very late the moment the relay republishes an old row. The age of a group is the maximum across its partitions, and alerts should use that maximum.

One trap is worth naming. A partition with no new traffic has a lag of zero and, with a naive calculation, an age of zero or an age that grows forever. Decide how an idle partition reports, and test it. The same function must survive the retention failure described below: a committed offset that has fallen behind the start of the log points at deleted data, so the check comes first and is reported on its own. A consumer that is stuck on a poison record, as in Chapter 8, shows zero throughput and a rising age, and that combination should be what pages someone.

```
function reportConsumerAge(partition):
    committed = committedOffset(partition)
    head = headOffset(partition)
    if committed >= head:
        // nothing waiting: report zero, and report liveness
        separately
        emitGauge("message_age_seconds", 0, partition)
        return
    logStart = logStartOffset(partition)
    if committed < logStart:
        // retention already deleted unread data: report how much, as
        a
        // gauge that follows the condition and not how often we
        checked
        emitGauge("unread_offsets_lost", logStart - committed,
partition)
        // the gauge clears at the next commit, so also leave a record
        logLostRange(partition, committed, logStart)
        committed = logStart
        record = readRecordAt(partition, committed)
        // append time: how long this record has waited at the broker
        emitGauge("message_age_seconds", now() - appendTime(record),
partition)
```

Retention is the deadline you cannot see

Message age has a second, less obvious job. On a log-based broker, a consumer that falls behind by more than the retention period loses messages silently: the broker deletes data that was never read, and the consumer

continues from wherever its offset-reset policy puts it. On a queue-based broker the same happens when messages expire. Chapter 3 named this failure, and no error accompanies it.

That reset policy is a decision, and each option has a price. Resetting to the earliest available offset replays whatever is left, so the loss is partial and the consumer must absorb a flood of old messages. Resetting to the latest offset skips ahead and makes the loss silent, which is what happened in the opening scenario. Failing, so that the consumer stops and waits for a person, turns a silent loss into an incident. I would choose failing for any flow where a gap costs money, and accept that the consumer is down until someone decides. Chapter 14 covers how to recover from the gap once you have noticed it: resetting to a timestamp and replaying, with the side effects suppressed.

The defense is arithmetic. Compare message age to the retention or expiry period, and alert when the age reaches a fraction of it, say half. At that point the deadline is still ahead, and a person has time to add consumers, pause a producer or extend retention. After the deadline passes, the only evidence is the absence of events, and that is what reconciliation exists to detect. I would put the ratio on the same dashboard as the age itself, because a lag that is acceptable for the business and fatal for retention is a real situation.

For a CDC connector, the equivalent signal is how much of the database's log the connector has not yet consumed, compared with how much the database is willing to keep for it. A connector that is down for longer than the database retains its log may need a fresh snapshot to recover, and that is an outage of its own.

Latency, retries and the rest

End-to-end latency needs a definition before it needs a graph. I would measure from `occurred_at` of the first event in a flow to the moment the final effect is committed, for example from `OrderPlaced` to `OrderConfirmed`. Report percentiles, not an average: the slow tail is the buyers who write to support. The final consumer sees only the envelope it consumed, and that envelope carries the time of its own fact, not the start of the flow. Either propagate the flow's start time in a header, or look it up by `correlation_id` in the event journal or the owning service. Then label

the measurement with the final `event_type`. Both timestamps come from different services' clocks, so expect some skew in the result; it is small next to a business deadline, and it is a reason not to alert on sub-second differences.

Drain rate is the observable form of backpressure. Divide the lag by the difference between the rate at which the consumer processes messages and the rate at which they arrive, and you get the time to catch up. If arrivals exceed processing, the lag never clears, and that is the number to alert on, long before the age reaches the retention limit. Chapter 3 defined backpressure and Chapter 6 covers how to bound the work a consumer takes on.

Retry rate and attempt counts surface the problem Chapter 6 called hidden failures: a dependency that has degraded for weeks while retries quietly absorb it. The attempt count histogram is more informative than the error count, because a message that always succeeds on the third attempt produces no errors at all.

DLQ arrivals and the age of the oldest unreviewed dead letter come from Chapter 8. Add **duplicate and stale counts** from Chapters 7 and 9. A rising duplicate count right after a deploy is normal, since ownership of work moves, and a rising count at a steady hour points at slow handlers. On a queue-based broker the usual cause is a visibility timeout shorter than the handler's run time. On a log-based broker it is a rebalance, or a consumer that exceeds its poll timeout and loses its partitions.

Alerts that mean something

The failure of most alerting setups is not too little data. It is that the data pages people for causes instead of for effects, and that, after a few weeks of noise, nobody trusts the pager. I would hold alerts to one standard: if this fires at three in the morning, a named person can say what to do about it.

Alert on **symptoms tied to a business deadline**. "Oldest unpublished outbox row is older than two minutes" is a symptom that something between a state change and its announcement is stuck. "Consumer CPU is above 80

percent" is a cause, and it may or may not matter. Use the cause metrics on dashboards where someone investigates, and keep the pager for the symptoms.

Give each alert an **owner and a threshold with a reason**. Chapter 8 made the same point for DLQs: a lost email and a lost capture deserve different thresholds, and the difference should be written down next to the alarm. The reason matters because thresholds drift, and a number with no stated reason gets raised the first time it is annoying.

Alert on **absence as well as presence**. A relay that crashes emits no error; it emits nothing. A consumer group that has lost all its members shows lag, but a topic that has stopped receiving events altogether shows no lag and no errors. For flows with a steady rhythm, such as orders during business hours, an alert such as "no `OrderPlaced` events observed for 10 minutes" catches a producer-side silence that every other signal misses. The threshold has to follow the traffic pattern, and for flows with no rhythm it is the wrong tool.

Finally, two tiers are enough for most teams: a page for anything that threatens a deadline or loses data, and a ticket for slow pressure such as a growing DLQ of low-value messages. A third tier tends to be ignored.

Warning: Raising an alert threshold to stop noise is how a team trades an annoyance for a blind spot. If an alert fires too often, fix its cause or its definition. Do not raise the number until it stops firing.

Reconciliation

Metrics and traces observe what the system does. They cannot tell you about an event that never entered the system, a handler that committed the wrong state without an error, or a repair that someone applied by hand. For those, you compare the system with an independent record of what should be true. That is reconciliation.

A reconciliation job takes two views of the same facts and reports the differences. The pairs are usually obvious once you look for them:

- **Orders against Payments:** every order in `PAID` state has a captured payment, and every captured payment belongs to an order that knows

about it.

- **Payments against Ledger:** every `PaymentCaptured` has a matching `LedgerEntryPosted` for the same amount in the same currency.
- **Your records against a provider's:** the payment provider's settlement report against your own capture records. Chapter 7 pointed here for effects that cannot be made idempotent, where reconciliation is the last line of defense.
- **A read model against its source:** a search index or a view that Chapter 9 said could be marked suspect and repaired later.

Three difference types cover most cases: **missing** (the source has it, the derived side does not), **stale** (both have it but the derived side disagrees, usually because it is behind), and **extra** (the derived side has something the source does not know about). Each points to a different cause. Missing suggests lost or never-consumed events, stale suggests ordering or lag, or now and then a handler that wrote the wrong value, and extra suggests a bug or a manual change.

The part that deserves care is that events in flight look exactly like discrepancies. A payment captured four seconds ago and not yet in Ledger is normal. A job that reports it will produce noise and teach people to ignore it. So the job compares only records older than a **grace period** chosen from your normal end-to-end latency, and it confirms a mismatch a second time before reporting.

```

function reconcileCaptures(window_start, window_end):
    upper = min(window_end, now() - gracePeriod)
    suspects = []
    for each batch in readCapturesInWindow(window_start, upper,
batchSize):
        // list per payment_id
        entries = fetchLedgerEntries(keysOf(batch))
        for each capture in batch:
            found = entries[capture.payment_id]
            if count(found) is 0:
                suspects.add({ key: capture.payment_id, kind:
"missing" })
            else if count(found) > 1:
                suspects.add({ key: capture.payment_id, kind: "extra"
})
            else if differs(found[0], capture):
                suspects.add({ key: capture.payment_id, kind: "stale"
})
        for each batch in readLedgerEntriesInWindow(window_start, upper,
batchSize):
            captures = fetchCaptures(keysOf(batch))
            for each entry in batch:
                if captures[entry.payment_id] is null:
                    suspects.add({ key: entry.payment_id, kind: "extra" })
            // a second look after a delay removes events that were merely in
flight
            sleep(recheckDelay)
            confirmed = filter(suspects, stillMismatched)
            for each item in confirmed:
                // upsert on (key, kind): first_seen_at survives, last_seen_at
moves
                upsertDiscrepancy(item.key, item.kind, now())
            // anything open in this window that no longer mismatches is
closed here
            resolveOpenDiscrepanciesInWindow(window_start, upper, except:
confirmed)
            return count(confirmed)

```

Several details of this block carry the correctness. The window is bounded and the cutoff is applied, so the job reads data that has had time to settle. Each batch is a keyed lookup instead of a join across databases, because Parcelly services do not read each other's tables, so the job fetches the other side through that service's own interface or a published extract. The job only records discrepancies and does not change anything. And because it upserts into a discrepancy table keyed by item and kind, running it twice, or after a crash halfway through a window, produces the same table, with no growing pile of duplicate rows. The same function closes discrepancies once the sides agree, which is the step the repair code below relies on. A payment posted to Ledger twice is caught by the forward pass, which

counts the matches. The reverse pass is what finds an "extra" entry with no capture behind it, such as a posting for a payment that was never captured, and a one-directional check never would.

Choose the cadence by the damage a discrepancy does. Payments against Ledger is financial and deserves a frequent run, with a daily full comparison as a backstop. A search index may need a nightly run. The job's own health matters too: alert if it has not completed in its expected window, because a reconciliation that silently stopped is worse than none, as people believe it is watching.

Note: A reconciliation job is a consumer of the data it checks, and it can be wrong. Treat a discrepancy as a lead for investigation first. The first weeks of a new job are mostly spent finding false positives caused by timing, rounding or a comparison rule that does not match the business.

Repair and backfill

Finding a discrepancy is half the work. The other half is changing the state so that it matches, without creating a second discrepancy.

The rule I would hold to is that **repairs go through the same path as normal changes**. If Ledger is missing an entry for a captured payment, do not insert the row with a database console. Chapter 3 treated operator actions as part of the failure model, and a hand-edited row emits no event, so every other consumer of that fact stays wrong, and the next reconciliation reports an inconsistency of your own making. Instead, republish the fact, or send the command that causes it, through the producer's outbox, so that consumers handle it as they handle everything else.

A repair can be retried, interrupted and run twice by two engineers, and it must stay safe in each case. Chapter 7's idempotency answers this only if the repair event carries the **original** `event_id`. A repair event with a new id is a different event to every consumer. If the first event was merely delayed, it arrives after the repair, and Ledger posts the payment twice. So the repair republishes the fact that already exists: the same envelope, with the same `event_id` and `causation_id`, and the inbox treats whichever copy arrives second as a duplicate.

That requires the producer to keep the original id. The simplest way is to store `event_id` on the business record (here, the capture) when the outbox row is written, together with a hash of the payload. If the outbox row still exists, it is marked unpublished and the relay sends it again. If it was purged, the envelope is rebuilt from the record with the stored id.

Rebuilding has a limit. The record is the current state, and the event is a fact from the past. If the record changed since, say a partial refund adjusted the amount, a payload rebuilt from it differs from the original while carrying the same `event_id`. That is why the hash is stored: the repair rebuilds the payload, compares its hash with the stored one, and stops for a human if they differ. A record that has no stored id or hash, usually one written before the habit existed, has to fall back to an id derived from its business key and to fields that cannot change, and then the consumer must also be protected by a unique constraint on that key, as Chapter 7 describes. I would add that constraint to Ledger anyway, as a second line of defense.

A republish also **fans out**. The topic delivers the envelope to every subscriber, not only to Ledger. A subscriber whose inbox still holds the `event_id` skips it. One whose inbox row was purged, because the event is older than its dedup window (Chapter 7), applies it a second time, and Notifications could send the same email again. So there are two safe paths. If the event is younger than the shortest dedup window among all subscribers, republish through the producer's outbox as above. If it is older, deliver the repair only to the consumer that missed it, through that consumer's own repair queue or topic, or by resetting the offsets of that one group, and rely on a business-key constraint as the protection. Chapter 8 applies the same principle to redrive.

```

function repairMissingLedgerEntry(discrepancy):
    capture = loadCapture(discrepancy.item_key)
    if ageOf(capture.event_id) > shortestDedupWindow():
        // a broadcast could be applied twice where the inbox row is
gone
        return sendRepairToConsumer("ledger", capture, discrepancy)
    try:
        transaction:
            original = outboxFindByEventId(capture.event_id)
            if original exists:
                // same envelope and event_id; the relay publishes it
again
                outboxMarkUnpublished(original.id)
            else:
                // purged: rebuild the identical envelope, never a new
id
                envelope = rebuildOriginalEnvelope(capture)
                if hashOf(envelope.payload) != capture.payload_hash:
                    // the record changed since the event was
published
                    raise RepairNeedsReview(discrepancy.id)
                outboxInsert(envelope)
                recordRepairAttempt(discrepancy.id, actor, now())
                // commit point: the repair is durable and the relay delivers
it
            on UniqueViolation as e:
                if e.constraint != "outbox_event_id_key":
                    // a different constraint: a real error
                    raise e
                // a concurrent repair inserted the same event_id first
                return
        // resolution is recorded by the next reconciliation, not here

```

Two details carry the safety. Marking an unpublished row unpublished again is harmless when repeated, and the unique violation from a second concurrent insert is treated as "already handled" instead of an error. The republished envelope also keeps its original `causation_id`, so the causation chain stays truthful, and the link to the discrepancy lives in the repair record. The last comment is the important one. The repair function does not mark the discrepancy resolved. The next reconciliation run does that, when the two sides agree. This keeps a repair that was published and then lost from being reported as fixed.

Bulk repair, usually called **backfill**, adds scale. A backfill of 30,000 events into a live consumer competes with the normal traffic and can push the flow's latency past its deadline. Run it in small batches, with a rate limit and a pause switch, as Chapter 8 recommended for redrive, and watch

message age and consumer lag while it runs. On a log-based broker you may prefer a deliberate replay into a separate consumer group that rebuilds a derived view, then switches over. That is cleaner when the consumer is a projection, and wrong when the consumer has side effects such as sending email.

Repair through the producer fits a handful of records. When the damage is wider, such as a consumer group that skipped a whole retention gap, a restored database or a lost broker cluster, republishing record by record stops being the right tool. Chapter 14 covers recovery by replay, disasters and broker migration.

Every repair needs a record: who ran it, which discrepancies it covered, what it published and when. The audit trail is how the next engineer learns that the spike in duplicates at 14:10 was a backfill, and it is how a regulator or finance team later learns that a Ledger entry was a correction.

Business invariants

Reconciliation compares two systems. An **invariant check** compares one system with a rule the business holds true. Invariants are often cheaper to evaluate and catch problems that no second system could, such as a bug that makes two systems agree on the wrong answer.

For Parcelly, a few examples show the shape:

- No order is `CONFIRMED` without a captured payment.
- No reservation stays held longer than its deadline.
- No saga instance sits in `AWAITING_RESULTS` past its `deadline_at`.
- Every captured payment has exactly one Ledger entry, not zero and not two.
- The sum of Ledger entries for an order balances to zero.

Each one is a query that should return no rows. Run it on a schedule, and treat each returned row as a discrepancy with an owner. The saga example is the one Chapter 10 left open: the timeout handler is the first guard against a frozen order, and an invariant query on stuck instances is the second, because the handler that should have fired may itself have failed.

```
table invariant_violation:
  invariant_name  text not null
  item_key        text not null
  first_seen_at   timestamp not null
  last_seen_at    timestamp not null
  resolved_at     timestamp null
  primary key (invariant_name, item_key)
```

Keying by invariant and item means a violation that persists is updated, not duplicated, and `first_seen_at` tells you how long a problem has been open. That age is itself a metric worth alerting on.

I would write invariants with the product and finance people rather than for them. Engineers know what the system can do, and they do not always know what the business considers a defect. A rule such as "a refund never exceeds the capture" is obvious to finance and may be written nowhere in the code.

Trade-offs and costs

Observability and reconciliation are not free, and they do not run themselves.

Telemetry has a price. Structured logs for every handled event multiply log volume, and high-cardinality labels such as `order_id` on metrics can overwhelm a metrics backend. Keep identifiers in logs and traces, and keep metric labels to small sets such as service, topic and event type. Sample traces, and accept that the trace you need may not exist.

An event journal duplicates data. It is worth it for flows where "what happened?" has legal or financial weight. For a low-value flow such as a cosmetic notification, the logs are enough, and the journal is a cost with no reader.

Reconciliation reads production data. A comparison job can load the source database, and one that is careless about batching can slow the service it protects. Run it against a replica where you have one, and bound its batch size and rate. It also adds a dependency: if the other side's interface is down, the job cannot run.

Alerts cost attention. Every page spends the credibility of the next one. A small set of well-owned alerts does more than a large set that nobody reads.

Repair has its own risk. A repair mechanism is a path to change production state outside the normal request flow. Without audit, approvals for the dangerous kinds, and idempotent design, it replaces one class of problem with a worse one.

When not to build all this: a system with one service and one consumer, low stakes and short flows does not need an event journal, invariant tables or a nightly reconciliation. Structured logs with `correlation_id` and an alert on message age are the minimum I would hold at any scale. The rest grows with the cost of a silent failure. If a lost event costs a ledger discrepancy, build reconciliation first. If it costs a stale search result, build it last.

Common mistakes

- **Alerting on health, not on flow.** Green CPU and memory graphs say the processes are alive and say nothing about whether orders complete. It is tempting because infrastructure metrics exist already. Add the symptom alerts: message age, outbox age and invariant violations.
- **Watching lag in messages only.** A message count has no deadline attached. Convert to age, and compare the age to the business deadline and to the retention period.
- **Injecting trace context in the relay.** The relay has no ambient span, so it attaches the wrong one, or none. Capture the context when the outbox row is written and forward it unchanged.
- **Inconsistent identifiers.** One service logs `order-81724`, another `81724`. Searching by correlation then fails exactly when it is needed. Define the format once in the event design and enforce it with a shared logging helper. Keep that one format for synthetic traffic too: Chapter 15 marks probes with a separate header and never with a special `correlation_id`.
- **Using one timestamp for every age.** Producer create-time, broker append time and `occurred_at` measure different delays. Use append time for the consumer-age gauge and `occurred_at` for end-to-end latency.

- **Republishing a repair to every subscriber.** A consumer whose inbox row is purged applies the event again. Check the event against the shortest dedup window, or deliver the repair to the one consumer that missed it.
- **Reconciling without a grace period.** The job reports every event in flight as a discrepancy, the team learns to ignore it, and the one real finding is lost in the noise. Compare only settled records and recheck before reporting.
- **Repairing by hand-editing rows.** It fixes one table, emits no event, and leaves every downstream consumer wrong. Repair through the producer and the outbox so that the change travels like any other.
- **A repair that is not idempotent.** A second run duplicates the correction. Republish with the original `event_id`, so that the inbox recognizes a second run and a late original alike.
- **Letting reconciliation decay.** The job breaks after a schema change, nobody notices, and the team believes it is covered. Alert when a run is late or fails, and review the discrepancy backlog on a schedule.

Key takeaways

- A flow spread across services needs identifiers that survive every hop: `correlation_id` in every log line and row, and trace context stored in the outbox row so that the relay forwards the right one.
- Tracing helps but is sampled and can be dropped, and links versus parent-child is a trade-off by case, so the envelope identifiers must be sufficient on their own to reconstruct an order's story.
- Message age, measured against a business deadline and against broker retention, is more useful than lag in messages, and it is the signal that warns you before a lagging consumer silently loses events; the offset-reset policy decides whether such a loss is silent or loud.
- Alert on symptoms tied to deadlines, give every alert an owner and a stated reason for its threshold, and include alerts for silence, because the failures that lose data often produce no signal at all.
- Reconciliation finds what no metric reports: it compares settled records across two views of the truth, classifies differences as missing, stale or extra, and records them without changing anything.

- Repairs must travel the normal path, republishing the original `event_id` so that a late original and a repair cannot both take effect, with an audit trail, and a discrepancy is closed by the next successful comparison, not by the repair itself. A republish reaches every subscriber, so old events go only to the consumer that missed them.
- Business invariant checks, such as no confirmed order without a captured payment, catch defects that two agreeing systems would hide.

Observation and reconciliation tell you that the system is misbehaving, and repair fixes a handful of records. Chapter 14 covers what to do when the damage is wider than a repair can reach: recovering by replay, surviving a disaster and moving to a different broker.

Chapter 14 — Recovery, Disasters and Broker Migration

Imagine a Tuesday at Parcelly. The managed broker cluster that carries every event in the marketplace becomes unreachable. The provider's status page says that the zone hosting it has failed, and that a replacement cluster will be available within the hour. When it is, the topics exist and are empty, or they hold data up to a point somewhere in the last few minutes. Nobody can say exactly where that point is.

The services are fine. Orders, Payments, Inventory, Shipping and Ledger each have their own database, and each database is intact. Every service has a relay that keeps retrying its publish, and the outboxes are filling up. Consumers are idle, waiting for a broker that is not there. In the meantime the team has to decide what to do when it comes back, and the options all sound plausible. Point the consumers at the new cluster and let them run. Republish everything from the outboxes. Restore the last backup of something. Each option is right for one store and wrong for another.

This chapter is about the day after a store goes wrong, or goes away, or is replaced on purpose. It asks one question. When the broker, the database or the region is no longer the thing it was, which of the guarantees you built are still standing, and how do you use them to get back to a state you can trust?

The problem, and why it is hard

The earlier chapters defended against failures that leave the stores intact: a crash between two steps, a slow consumer, a duplicate, a poison message. A recovery is different because the thing that went wrong is the memory itself. A store went back in time, or lost part of its history, or is being swapped for another one. Every defense in the book relies on some store remembering something, so the question becomes which memory survived and which did not.

That is hard for two reasons. First, the stores fail independently and in different directions. A database restored to six hours ago and a broker that lost its last two minutes disagree with each other in a way that neither one can detect. Both are consistent with themselves. Chapter 3 called this losing a whole store, and noted that nothing raises an error at first.

Second, recovery work is rare, and rare work is done badly under pressure. The commands that move a consumer group backward, the settings that stop a replay from sending a second round of email, the order in which services are restarted: all of these are decisions that are cheap to make in advance and expensive to improvise.

The thesis of this chapter is the thesis of the book at its barest. The broker is not the thing that survives. What survives a lost cluster, a restored database or a change of product is the failure semantics you designed around it: the transactional outbox (Chapter 4) that remembers what was never confirmed, the inbox that recognizes what was already done, the stable identifier, the contract and the habit of comparing two views of the truth. If those are in place, the broker is replaceable. If they are not, the broker was carrying the system's correctness, and nobody knew.

Recovering by replay

Replay, defined in Chapter 2, is the deliberate re-reading of history from an event log. It applies to the log family only, because a queue deletes what it delivers. Chapter 15 uses replay as a test technique. Here it is a recovery tool, and it has a procedure.

The situation it fits is the one that opened Chapter 13: a consumer group fell behind, retention deleted messages it had not read, and the group moved on past a gap. If the data is still retained, the first recovery to consider is not to reproduce events one by one from the producer. It is to move the group's position back to before the gap and read again.

Resetting a group to a point in time

A consumer group's position is stored as an offset per partition. Moving it backward is an administrative action on the broker, and most log-based brokers can target either an explicit offset or a timestamp, which the broker resolves to the first offset at or after that time. I prefer a timestamp.

An offset is a number that means something to one partition of one topic on one cluster, and the timestamp is what you know: the deploy started at 09:40, so go back to 09:30. Check your broker's documentation for how it resolves a timestamp, since the producer's time and the broker's append time are not always the one it uses.

The procedure is short and the order matters.

1. Stop the group. Resetting the offsets of a group that is running either fails or is overwritten by the next commit, depending on the broker.
2. Record the current offsets before changing anything. This is the only way back if the reset was wrong.
3. Choose the target with a margin. Go back earlier than you think you need to, because duplicates are cheap and a gap is not.
4. Reset to the target, restart the group and watch the age of the oldest unprocessed message (Chapter 13) as it drains.
5. Keep watching until the group is back at the head and then reconcile, because the reset repairs only what the log still held.

Choosing a margin is safe only because of the inbox (Chapter 7), which turns the re-read of messages the group did process into no-ops. A group whose consumers are not idempotent should not be reset, whatever the procedure says.

Limiting replay to one consumer group

A log lets 10 groups read the same topic, and each has its own offsets. That is the property that makes the reset safe to target. Reset the group that missed the events, and no other group notices. Reset the smallest set of groups that missed something, and name them in the plan before running it. Payments' consumer of `OrderPlaced` keeps its position while Inventory's group re-reads the last day.

A second form of the same discipline is a separate group for the replay. If a projection must be rebuilt, for example Orders' search view, start a new group with a different name that reads from the beginning into a fresh table, and switch the reader over when it has caught up. The old group never moves, so there is nothing to undo if the rebuild goes wrong. It costs storage and a second consumer for a while, which is a small price for a rollback that is free.

Suppressing side effects

A replay applies today's code to yesterday's facts. If a handler sends email, calls the payment provider or notifies a carrier, replaying makes it do so again. Chapter 12 raised this, and the rule it gave is the starting point: a consumer rebuilt by replay has to know that it is replaying.

The first idea people have is a flag in the message, a header that says "this is a replay". That does not work, because a replayed message is the original message. The broker holds the bytes the producer wrote, and nothing marks them as being read for the second time. The mode belongs to the consumer, not to the message. The replaying group runs as a separate deployment with a configuration value such as `mode: replay`, or the effect layer checks the group name.

With the mode known, there are four options for the effects themselves, and each has a price.

- **Gate effects at the edge.** Every external call goes through one function that checks the mode and, in replay, does nothing. It is the clearest design and the cheapest to audit, but it only works if no handler bypasses the gate. A new handler that calls a provider directly defeats it, so the rule needs a test (Chapter 15).
- **Put idempotency keys on external calls.** If the capture call carries a key derived from `event_id`, the provider recognizes the second attempt. Chapter 7 covered this. It protects only for as long as the provider remembers the key, so it is a second line of defense and not the one to rely on for a replay of last month.
- **Pause the outbox of derived events.** A replayed handler writes new rows to its own outbox in the same transaction, and those rows go out to the next service. If the new events carry fresh identifiers, downstream consumers treat them as new facts. Either derive the identifier of each outgoing event deterministically from the input `event_id` and the handler's name, so that the second run produces the same identifiers and downstream inboxes absorb them, or hold the relay of the replaying service until you have checked what it produced.
- **Replay into a different environment.** For a projection with no effects, a new table is the cleanest answer, as above. For a handler with

effects, a different deployment with its effect layer connected to a stub is a way to test the replay before running it for real.

Warning: Resetting a group that sends email, charges cards or calls carriers, with no gate and no keys, turns every historical event into a new action. The inbox does not help if you reset it as well, and for external effects it never could, because the provider does not roll back with your database. Decide the suppression method before you move the offsets, not after the first complaint.

The dedup window and the inbox

The inbox changes what a replay does, and this works for you or against you depending on what you want.

If the consumer's database is intact and the range you are re-reading is inside its dedup window (Chapter 7), the inbox recognizes every `event_id` it has already processed and skips it. That is what you want after a gap: the group processes only what is missing, and the rest is cheap. It is also why a replay can repair the consumer that never saw an event, without disturbing the ones who did.

If the range is older than the window, the inbox rows are gone. The consumer applies old events a second time. For a consumer protected by a business-key constraint, such as one ledger entry per capture, the second application fails harmlessly. For one without it, the effect happens twice. So the dedup window is part of the recovery plan: before a replay, state the oldest event in the range, the shortest window among the groups that will see it, and what protects the consumer where the two do not overlap. Chapter 7 gave the rule that the window must outlast any replay you plan. This is the plan.

The inbox can also be the obstacle. A consumer with a bug wrote wrong state. You fix the bug and replay, and every message is skipped, because every `event_id` is in the inbox. That case comes up shortly.

Replay, repair and redrive

Three tools move messages again, and they differ in what they act on.

Tool	What it acts on	Chapter	Right when
Repair	A handful of records, republished through the producer's outbox with the original <code>event_id</code>	13	A few items are missing or wrong, and the producer can say which
Redrive	Dead letters, moved back to the consumer that rejected them	8	Messages failed visibly and the cause is fixed
Replay	A range of the log, re-read by one group	14	Many keys are affected, the data is still in retention, and the consumer is idempotent

The distinguishing question is who knows what is wrong. With repair, the producer or a reconciliation job names the specific items. With redrive, the system itself set aside the messages. With replay, nobody has a list: the damage is a time range, and the cheapest way to cover a range is to read all of it again and let the inbox sort out what was done. Replay trades precision for coverage.

Retention is a recovery boundary

Replay reaches only as far back as the log holds data. Retention, a setting chosen long ago for disk cost, is therefore also the limit of the cheapest recovery you have. An outage that outlasts retention, or a bug discovered after it, moves you from the cheap path to republishing from producers and reconciling.

I would set retention against the time the team needs to notice and fix a problem, not against disk alone. If a reconciliation job runs daily and a fix takes a day to ship, a retention of one day leaves nothing. Retention also interacts with the cost side: longer retention increases the number of old schemas a replay can meet (Chapter 12), and it extends how long personal data lives in the log. I don't know the right number for your system. It depends on how fast you notice, how long the repair takes, and what the log is allowed to hold.

A consumer that was wrong without failing

Chapter 3 has a row for a consumer bug with no error: the handler succeeded and wrote the wrong result. This is the recovery that most needs replay, and the one where the inbox gets in the way.

Imagine that Ledger's consumer of `PaymentCaptured` converted amounts with the wrong exponent for one currency, for a week. Every message was acknowledged. Nothing failed, and the inbox holds every `event_id`. Reconciliation (Chapter 13) eventually reports that Ledger's totals for that currency disagree with Payments. The fix ships, and now the wrong rows exist and must be corrected.

Replaying the topic into the same consumer does nothing, because the inbox says each event is done. There are two honest routes, and the choice depends on what Ledger is.

If the consumer's state is derived and can be thrown away, such as a projection, rebuild it. Start a new group into a new store with an empty inbox, replay from the first affected event, verify it against the source with reconciliation and switch over. The wrong state is discarded, not corrected.

If the consumer's state is a record that must not be rewritten, as in an append-only ledger, replay still helps, but not to overwrite. Read the affected range with the fixed code in a correction mode. For each event, compute what the entry should have been, compare it to what is posted net of earlier corrections, and post a correcting entry for the difference, with an identifier derived from the original `event_id` and the correction's own name, so that running the correction twice is a no-op. The unique constraint on the ledger entry's own identifier does that work, so the correction needs no inbox row, and the loop treats a duplicate key as "already corrected" and moves on to the next event.

The wrong entries stay, as history. The corrections are new facts that say why. This is slower and more explicit than a rebuild, and it is the only form that an auditor accepts.

```

function correctLedgerRange(from_time, to_time, correction_name):
    for each event in readLog("payments.captured", from: from_time,
to: to_time):
        original = ledgerEntryFor(event.event_id)
        posted = original.amount_minor + sumCorrections(original.id)
        // the fixed logic
        should_be = buildEntry(event)
        if posted == should_be.amount_minor:
            continue
        correction_id = deriveId(event.event_id, correction_name)
        try:
            transaction:
                postLedgerEntry({
                    id: correction_id,           // unique: re-running
this correction fails
                    amount_minor: should_be.amount_minor - posted,
                    corrects: original.id
                })
            // commit point: the correction is durable
            on DuplicateKey(constraint: "ledger_entry_id"):
                // an earlier run already made this correction; skip it
                continue

```

The common thread is that replay finds the affected range cheaply, and the decision about what to do with each event belongs to the consumer's own semantics.

Disaster scenarios

A disaster is a failure where the loss is not a message but a store. The useful way to organize the possibilities is by which store went back in time. Write each scenario down in the pattern Chapter 3 used: what broke, how it shows, and which guarantee still holds. The last of the three is the useful one, because it names what you are relying on, and a flow with nothing to put there is exposed in that scenario. Before the scenarios, a word about what you are aiming for.

Objectives for each store

Two numbers describe a recovery. The **recovery point objective (RPO)** is how much data you can afford to lose, measured as time: a backup every six hours means an RPO of up to six hours. The **recovery time objective (RTO)** is how long you can be without the service. Both are decisions, and the usual mistake is to make one for the whole system.

They differ because the stores differ. For the stores in this chapter, a sketch looks like this.

Store	What a loss costs	Typical way back	Shapes
A service database	State that exists nowhere else, unless an event carries it	Backup plus log shipping	RPO per database, set by the business value of the data
The outbox	Events not yet published	Same database, so same RPO	Any row lost is an event that never existed
The inbox	Memory of what was processed	Same database as the effects	Moves back with them (Chapter 3)
The broker	Events in flight and retained history	Replication, or republishing from producers	RPO is set by replication, RTO by how fast consumers drain
A derived projection	Nothing, if it can be rebuilt	Replay	RTO is the replay time, RPO is zero if the log holds everything
An external provider	Effects outside your control	Their records, and reconciliation	Outside both of your objectives

The point of the table is that an RTO for the replayed projection is a drain calculation. If the log holds a month and the group can process a day's worth per hour, the rebuild takes about 30 hours, and that is its RTO (suppose those figures; yours come from measurement, as in Chapter 13). A team that has never measured its drain rate does not know its RTO.

I would also tie objectives to the failure model from Chapter 3, flow by flow. The payment capture flow has a different tolerance from the notification flow, and the Ledger's tolerance for a gap is closer to zero. A uniform objective ends up too strict for notifications and too loose for the ledger.

Restoring a database from backup

Suppose Orders' database is restored to a point six hours back. What does that do to the machinery of this book?

The **outbox** goes back with the state. Rows written in those six hours are gone, along with the orders they announced, though Payments and Inventory may have acted on the events already. Rows that were unpub-

lished at the restore point but were published later in the old timeline are unpublished again, so the relay publishes them a second time. That is a harmless duplicate, because the `event_id` is the same.

There is a sharper hazard. If order identifiers come from a sequence, the restored sequence is rewound, and the next order will receive an identifier that was already given, to a different order, in the lost timeline. Payments now holds a capture for `order-81724` that belongs to a buyer Orders has forgotten, and the new `order-81724` belongs to someone else. I would prefer opaque random identifiers for aggregates for this reason. If you must keep a sequence, advance it past the highest value ever issued before accepting traffic, a number you can read from downstream services.

The **inbox** of a consumer whose database is restored goes back together with the effects. Chapter 3 pointed out that this is correct: messages processed before the restore point are still in the inbox and their effects survived, and messages after it are forgotten in both places. That property holds only when the inbox and the effects live in the same database.

What does not go back is the consumer's **offset**, which lives in the broker. The group is positioned at the head, past six hours of events that the restored database has forgotten. The consumer will never see them again unless someone moves the group back, so the restore plan includes a reset of that group.

The target needs care, because the restore point is a time on the database's clock, while a timestamp reset works on the event times in the log. At the moment of the restore the group was not at the head. It was at some position, and the event at that position was produced some time before, by as much as the group's message age (Chapter 13). So the target is the event time at the group's position at that moment, minus a margin. If the target is set from the restore point alone, a group that was an hour behind skips the hour it had not yet processed when the database went back.

Read the position from the message age history or from the recorded offsets, and keep that history outside the database you are about to restore. Some teams store the offset in the same database as the effects, in the same transaction, so that the two rewind together. It removes the manual step and takes over the offset bookkeeping from the broker, which is a cost to weigh.

External effects stay where they were. The capture that Payments sent to the provider in the lost timeline exists in the provider's records and not in Payments' database. This is where reconciliation (Chapter 13) is required. Compare your records with the provider's, and repair the differences.

The order in which the system is restarted matters, and it is worth writing as a procedure.

```
function resumeAfterRestore(service, restore_point):
    // 1. nothing moves while the stores disagree
    pauseConsumers(service)
    pauseRelay(service)

    // 2. compare before anything is published or applied
    findings = reconcileAgainstNeighbors(service, since:
restore_point)
    repairThroughNormalPath(findings)

    // 3. start from where each group was, not from when the incident
was
    for each group in consumerGroupsOf(service):
        position_time = eventTimeAtPosition(group, at: restore_point)
        resetToTimestamp(group, position_time - safetyMargin)

    // 4. re-reading derives events and commands again; their ids must
be
    //    deterministic, so the relay stays paused until they are
checked
    resumeConsumers(service)
    verifyDerivedOutboxRows(service)
    resumeRelay(service)

    // 5. a restore is not finished until reconciliation is quiet
    scheduleReconciliation(service, within: "one hour")
```

The steps are the point. Pausing both sides first keeps the restored service from making its inconsistent state worse, and reconciling before resuming follows Chapter 3. The reset turns the restore into a replay problem, with one addition.

A restored Orders that re-reads the lost hours produces new outbox rows, including a `CapturePayment` command, and a fresh `command_id` would look new to Payments' inbox and could capture twice. So the ids of derived events come from the input `event_id`, as in the side-effect section. A command triggered by two events, as `CapturePayment` is, takes its id from

the business key instead, such as `order_id` plus the command type (Chapter 7), because which of the two is read last can change on replay. The relay is held until someone has checked what was produced.

Losing broker data

A broker can lose data without losing the cluster. In a log family, a replica that was behind may be promoted to leader, and the messages it did not have are gone. The acknowledgment level and the minimum number of in-sync replicas decide whether the producer was told that such a message was safe (Chapter 2). Some brokers allow or forbid promoting an out-of-date replica by configuration, and I would find the default for yours and write it down. Retention can do the same quietly, as Chapter 13 showed.

What you see: consumer positions that jump or are reported as out of range, a topic whose end offset is lower than it was, and, after the fact, reconciliation findings of kind "missing". The broker's acknowledgment is evidence with a boundary, and the boundary is the number of copies that existed at that moment.

What survives: the outbox, if you kept the rows. A relay that purges rows right after publishing has no memory of what it sent. One that keeps published rows for a period, say a few days as an illustration, can mark the rows from the affected window as unpublished and send them again. The republish keeps the original `event_id`, so consumers that did receive an event treat the second copy as a duplicate. This is the same step as Chapter 13's repair, applied to a range, and it carries the same fan-out caution: events older than a subscriber's dedup window need delivery to the one consumer that missed them, not to everyone.

If the outbox rows are gone, the producer's own records are the next source: rebuild the envelope from the business record with its stored `event_id` and payload hash, and stop for a human where the hash differs. And if neither exists, the event cannot be recovered and reconciliation will keep reporting it. At that point the right question is a business one: is the missing fact reconstructed from another system, or accepted as lost and explained? Designing so that this question is rarely asked is what keeping the outbox rows for a while buys.

Losing a region

A region loss combines the earlier two, and adds a problem of its own. Assume that Parcelly runs a standby cluster in a second region, fed by a replication tool that copies topics continuously, and that databases replicate asynchronously.

The first property to state is that **asynchronous replication has an RPO**, and it is the replication lag at the moment of failure. Whatever had been written in the primary region and not yet copied is gone from the standby's point of view. That is the measure of what you lose, and it is a number that varies minute by minute. It is also worth alerting on, against a stated limit, in the way Chapter 13 treats message age: you are watching your RPO.

The second property is that **offsets do not travel**. The standby holds a copy of the messages, but the copy has its own offsets, assigned when the replica wrote it. A consumer's committed position in the primary says nothing about the standby. Some replication tools offer offset translation; I would check what yours guarantees, and I would not design a recovery that depends on it being exact.

The portable plan is to start each group on the standby from the event time at the position it had reached when the primary failed, minus a margin, and let the inbox absorb the overlap. It is the rule from the database restore. A group that was 40 minutes behind starts 40 minutes back. Events that were never replicated are not on the standby at all, and the outbox window below covers them.

Since the primary's own commit information is lost with the region, the message age history has to be kept outside it. If it is gone, fall back to the oldest point inside the consumers' dedup windows and accept the extra duplicates. An alert threshold is not enough, because the alert may have been firing at the time.

The third property is that **duplicates at failover are certain, not possible**. A consumer that restarts earlier than its true position will process events twice. A relay that was mid-publish when the region failed may publish a second time from the surviving database.

Gaps are the other side: events that the primary acknowledged and the standby never received. If the producer's database survived and is later than the standby's broker, it holds rows marked published that the standby never saw. For these, widen the republish window of the outbox as in the broker-loss case: mark the rows from the last replication-lag interval as unpublished, plus a margin, and send them again. The replication lag sizes this window, and only this window. Duplicates cost nothing at the consumer. Gaps cost the thing you are trying to protect.

If the producer's database is itself lost with the region and restored from an older replica, the case reduces to restoring from backup, with the re-wound sequences and the forgotten orders that come with it.



Read the diagram from the left. The two steps that deal with the unknown, the republish window and the timestamp margin, both err toward duplication. Every step that follows exists to make duplication harmless.

Broker migration

Migration is a disaster you schedule. A team moves from one broker to another because of cost, an end of support, a managed offering or a change in the access pattern, and the move has the same ingredients as the disasters above: data on one side, a new empty place on the other, positions that do not carry over, and a period when two systems are alive at once. The difference is that you have time to be careful, and no emergency forcing you to improvise.

The book's tools make a migration a controlled change. The reason is that several of them were placed between the application and the broker on purpose.

The relay as the single switch point

With a transactional outbox (Chapter 4), a producing service never talks to the broker. It writes to its own database, and the relay (Chapter 5) moves rows to the broker. So the choice of broker is encoded in one process. Changing where events go means changing what the relay pub-

lishes to, and the producer's code does not change. That is a smaller change than one spread over every service, a side benefit of a pattern Chapter 4 recommends for another reason.

The relay also owns the envelope. It takes `event_id` from the outbox row and puts it in the message it publishes. On the new broker the same event has the same `event_id`, and that equality is what makes the rest of the migration safe.

Dual-publish

The safest first step is for the relay to publish each row to both brokers. For a while, every event exists on the old broker, where today's consumers read it, and on the new one, where nobody reads it yet.

Dual-publishing is itself a dual write, and Chapter 4 explains why those are dangerous. Here the risk is smaller, because the outbox row is the source of truth and a crash leaves the row to be sent again.

The design question is what the relay waits for. If it marks a row published only after both publishes succeed, an unreachable shadow broker leaves rows unpublished and stops the live flow that consumers depend on, for the sake of a broker nobody reads. So track the two targets separately. The row is published, in the sense of Chapter 5, when the old broker has it, and a second marker records the new broker. One pass, in `id` order, feeds the new broker the rows that lack the second marker, which keeps the order per key, and the purge waits for both. The cost is one more column and one more pass, and in exchange an outage of the new broker shows up as a growing gap in the comparison and not as an outage of Parcelly.

During this phase the old broker stays the one that consumers trust. The new broker is a shadow, and its only job is to be compared with the original.

Mirroring and bridges

An alternative to dual-publish in the relay is to copy messages between brokers with a tool: a mirroring tool for log-to-log, or a bridge or connector for log-to-queue and queue-to-log. This needs no relay change and also carries the history that existed before the migration started, which dual-publishing does not.

Two cautions. A mirror or bridge is a consumer and a producer, so it is at-least-once and it produces duplicates, as every consumer-producer pair does. And it must pass `event_id` through, in the message body or a header, unchanged. A bridge that mints its own identifier turns every copied event into a new event, and the inbox cannot recognize any of them. Test this on a sample before you rely on it. Whether a mirror preserves partitioning, keys and headers, and in what order across partitions, differs by tool, and the answer is in its documentation, not in this book.

Offsets do not translate

A consumer group's offset on the old broker is meaningless on the new one. Even between two brokers of the same kind, the copy has its own numbering. Between a queue and a log there is no offset at all on one side.

The answer is the one that the region-loss section reached: pick a safe starting point and rely on idempotency. For each group, choose a timestamp such that every event the group had not yet finished processing on the old broker was published after it. Take the group's lag at the moment of the switch, convert it to time, and add a margin. Start the new group at that timestamp on the new broker. It re-reads events the old group already handled, and the inbox skips them.

The dedup window applies here as it does to a replay: the oldest event the new group will re-read must be inside every consumer's window.

Ordering and partitions

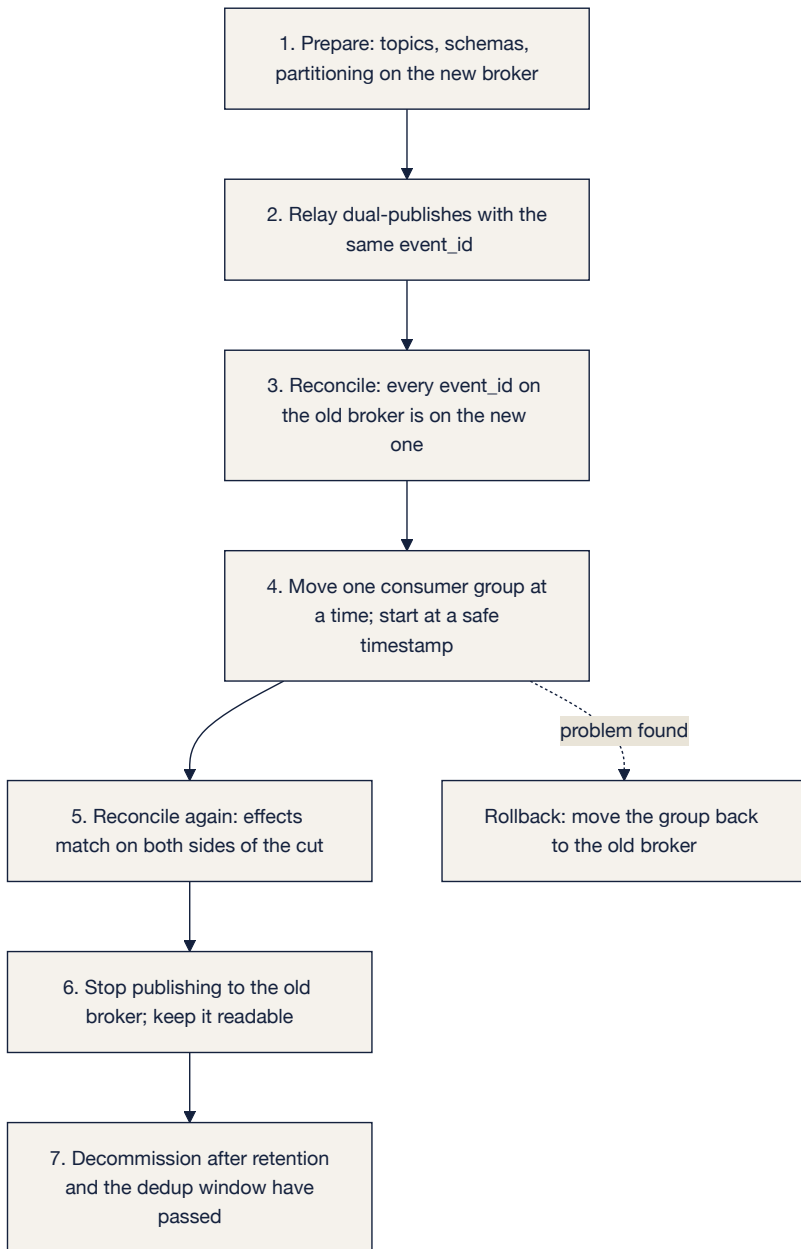
Chapter 9 gave the condition for per-key order: same key, same partitioner, same partition count. A migration changes at least one of the three. The new broker may use a different partitioner, so one `order_id` lands on a partition where it never did, and the new topic may be created with a different number of partitions because that is the moment people reconsider the number.

Two things follow. Compute the partition in the relay from the stored `partition_key`, as Chapter 9 suggested, so one piece of code owns the mapping, and test that events for one key land on one partition in the new broker. And plan the cutover so that no key has events in flight in both places. If a consumer starts reading the new topic while the old topic still

holds unprocessed events for the same `order_id`, the new events can be handled before the old ones: `OrderCancelled` before `OrderConfirmed` again, the failure from Chapter 9. There are two ways to avoid it. Drain the old backlog to zero before starting the new group, which pauses processing for the length of the drain. Or make the consumers tolerant by design, with state-carrying events and an aggregate version that rejects anything older, as Chapter 9 described. I would drain first for any flow where order matters and the pause is affordable.

The cutover

The order of operations is the part people ask about: consumers first, or producers first? With dual-publishing the question mostly dissolves, because producers are already writing to both before a single consumer moves. The order becomes this.



Reconciliation brackets the move. Before it, compare the set of `event_id` values on the two brokers over a window: anything on the old broker and missing on the new one is a dual-publish defect, found while nobody depends on the new broker. After it, compare the effects: a group moved at 14:00 should have produced the same state it would have produced on

the old broker, and Chapter 13's reconciliation across services is what verifies that. A migration that is declared finished because the consumers are healthy has verified only that nothing crashed.

Move consumer groups one at a time, beginning with the least critical, say Notifications, and ending with the Ledger. Each group is a small, separately reversible step.

Rollback

A migration without a rollback plan is a one-way door. With this design, the plan comes almost free, for as long as one condition holds: the old broker keeps receiving every event. While the relay dual-publishes, the old broker is complete, so rolling back means pointing the group back at the old broker at a timestamp with a margin. The inbox absorbs the events the group handled in the new place.

The condition is why step six comes late. Once the relay stops writing to the old broker, a rollback needs the flow reversed, with the relay writing to the old one again and the old broker's consumers starting from the point where it stopped. That is still possible because the outbox holds the rows, and it is slower. Rollback is also bounded by retention on the old broker and by the dedup window, so I would decommission only when both have passed since the last event it received. Retiring the old broker on the day the last consumer moves removes the rollback at the moment the new broker's problems are most likely to appear.

Between queue and log

When the migration crosses families, the semantics change, and a plan that treats it as a change of address is wrong.

Aspect	Queue to log	Log to queue
History	Gained: replay becomes possible (and retention becomes a decision)	Lost: no replay, so recovery relies on the outbox and producers
Acknowledgment	Per-message ack becomes an offset commit; one slow message can now block its partition (Chapter 9)	An offset commit becomes per-message ack, with visibility timeouts and redelivery

Aspect	Queue to log	Log to queue
Ordering	Group order, where the queue offers it, becomes per-partition order; choose the key	Per-partition order must be recreated, for example with message groups where the queue supports them
Dead letters	Redrive is something you build or configure; the log has no built-in dead-letter queue (DLQ), only a dead-letter topic you create (Chapter 8)	The queue's DLQ features appear, and its limits and retention differ
Delays and expiry	Delayed delivery and per-message expiry may not exist and need a substitute	Delay and expiry may be native; retention-based recovery goes away

Every row is a place where a behavior your consumers relied on can change silently. I would write the list of behaviors each consumer depends on before choosing the target, and test the ones that matter, such as a poison message that must not block its partition, in a rehearsal (Chapter 15).

Rehearsing recovery

None of this is worth much if the first time you run it is under pressure. A restore that has never been done is a belief. A replay procedure that has never run against realistic volume does not have a duration.

Two rehearsals earn their cost. A **restore test** takes a backup, restores it into a scratch environment, and checks what the restore does to the out-box, the inbox and the offsets, and then runs the resume procedure and times it. A **game day** removes something on purpose in a safe place, a broker, a zone, a consumer database, and has the team recover using only the runbook. Chapter 15 covers how to design these drills. The reason belongs here: only running recovery shows which steps you forgot.

A runbook for this chapter's scenarios has to answer a short list of questions, written down and not carried in someone's head. Chapter 16 puts them into its runbook guidance.

- Which store is affected, and which went back in time?
- What is the recovery point, and how was it determined?
- Which consumer groups must be reset, to what timestamp and with what margin?

- Which effects must be suppressed during the replay, and by what mechanism?
- Is every range to be re-read inside every affected consumer's dedup window?
- What is paused, in what order, and who resumes it?
- Which reconciliations confirm that the recovery worked, and who reads them?
- How is the whole thing undone if it is going wrong?

Trade-offs and costs

Retention is a recovery budget. Longer retention keeps the cheap recovery available for longer, at the costs listed earlier. Shorter retention pushes you to republish from producers.

Keeping outbox rows costs storage and a cleanup job. The same rows that make a republish window possible are the ones a leaner design deletes immediately. With change data capture (CDC) and immediate deletion, the window does not exist (Chapter 5). If you want the safety, keep published rows for a stated period and monitor the table.

Replay is a blunt instrument. It re-reads everything in the range, including what was fine, and it relies on every consumer being idempotent and every effect being gated. For a few known records, repair is cheaper and safer. Replay earns its place when the damage is wide and nobody holds a list.

Side-effect suppression is code that has to be tested. A gate that exists but is bypassed by a new handler is worse than none, because it creates confidence. The cost is a permanent test and a rule in code review.

A migration with dual-publish costs twice. Two brokers run at once, the relay does more work and tracks two targets, and the duplicates on the old side rise. In return you get verification before anyone depends on the new broker, and a rollback. If the system tolerates a maintenance window, a direct cutover is cheaper: stop the relay, drain the old consumers, point the relay at the new broker and start the consumers there. With an outbox that needs an outage and no loss, because the rows wait. It gives up the verification and the rollback, and I would not choose it for the ledger.

When not to build all of this. A flow where a gap is cheap, such as an internal metrics feed, does not need a republish window, a runbook or a game day. Match the recovery to the cost of the failure, and write down which flows you decided to leave uncovered. Their absence from the plan should be a decision.

Common mistakes

- **Resetting a group without checking its side effects.** It is tempting because the reset is a single command and the data is there. The result is a second round of email and a second charge attempt. Decide how effects are suppressed before moving any offset.
- **Resetting too many groups.** Starting everyone over feels thorough. Every group you reset re-reads history and carries the risk of its own effects. Reset only the groups that missed something.
- **Relying on a replay header.** It is tempting because a flag on the message looks like the cheapest signal. A replayed message is the original message and carries no sign of being read again. The mode belongs to the consumer deployment or the effect layer.
- **Replaying into a consumer with the bug's inbox.** After a consumer bug, the inbox skips every message as done. Rebuild a derived state into a fresh store, or post explicit corrections for an append-only one.
- **One recovery objective for the whole system.** It is tempting because it is a single number to agree on. It is too strict for notifications and too loose for Ledger. Set objectives per flow and per store.
- **Restoring a database and resuming at once.** The service is healthy and the dashboards are green, so traffic is resumed. The restored service then publishes rewind state, reuses identifiers and misses events its offsets skipped. Pause, reconcile, reset offsets, then resume.
- **Reusing sequence-based identifiers after a restore.** Sequences are the default and nothing looks wrong until two orders share an identifier. A rewind sequence hands out an identifier that already belongs to another order. Use opaque identifiers, or advance the sequence past the highest value ever issued.
- **Trusting offset translation.** It is tempting because the tool advertises it and it would save the timestamp arithmetic. A replication tool that

copies messages may or may not map offsets exactly. Plan recovery that works without it, using timestamps and the inbox.

- **A bridge that mints new identifiers.** A bridge is often configured to wrap each message in its own envelope, and that passes a test with one consumer. The copied events look new, so the inbox recognizes nothing and every duplicate becomes an effect. Pass `event_id` through unchanged and test it.
- **Decommissioning the old broker with the last consumer move.** It feels like the migration is done. It removes the rollback exactly when it is most likely to be needed. Wait for retention and the dedup window to pass.
- **Never restoring a backup.** The backup job is green. The first restore is then the real one. Restore into a scratch environment on a schedule and time it.

Key takeaways

- When a store is lost, restored or replaced, what survives is the failure semantics around it, meaning the outbox, the inbox, the stable `event_id`, the contract and reconciliation, and the broker itself is replaceable.
- Replay is a recovery tool for the log family: stop one consumer group, record its offsets, reset it to a timestamp measured from where the group actually was, with a margin, and let the inbox absorb the duplicates, as long as the data is still within retention.
- A replayed message looks like the original, so the replaying consumer must know its mode and gate its effects, derive downstream identifiers deterministically or hold its outbox, and never rely on a header.
- The dedup window is part of every recovery plan, because a range older than the window re-admits duplicates, and a consumer bug turns the inbox from a protection into an obstacle that needs a rebuild or explicit corrections.
- Recovery point and recovery time objectives belong to each store and flow, and a restored database rewinds its outbox, inbox and sequences while the offsets stay where the broker left them, so the resume procedure pauses, reconciles, resets offsets and only then resumes.

- Asynchronous replication across regions has an RPO equal to its lag and carries data but not consumer offsets, so a failover starts each group from the event time of its own position at failure, sizes a republish window of the outbox by the replication lag, and treats duplicates as certain and gaps as the real risk.
- A broker migration uses the relay as the single switch point, dual-publishes with the same `event_id`, moves one consumer group at a time from a safe timestamp, reconciles before and after, and keeps the old broker readable until rollback is no longer needed.

A recovery procedure that has never been run is a hypothesis, and so is the code that is supposed to make a system survive failure. Chapter 15 covers how to test event-driven systems, including how to rehearse the restores, failovers and replays that this chapter described.

Chapter 15 — Testing Event-Driven Systems

Imagine the Inventory team at Parcelly on the Friday before a long weekend. Their suite is green: 400 tests, a few minutes of runtime, coverage that nobody argues about. The `OrderPlaced` handler has a test that builds an event, calls the handler, and checks that stock went down by two. It passes every time. The tests use an in-memory stand-in for the broker, which delivers each message once, in order, immediately.

On Saturday a deploy restarts the consumers. Handlers run slower than usual while they warm up, and one outruns its visibility timeout, so the broker hands the same `OrderPlaced` message to a second consumer while the first is still processing it. Both read the inbox, both find nothing, and both go on to reserve the stock. The unique constraint that should have made one of them fail was missing from the real schema, and no test noticed, because the tests ran on a hand-written fake database that never enforced constraints and never ran two handlers at once. The suite had never been wrong about anything it checked. It never checked the one thing that mattered.

Meanwhile, a test in the Orders suite fails one run in 20. The team has stopped looking at it. It waits two seconds for an event to arrive and sometimes the event takes three.

Both stories have the same root: the tests describe a system that does not exist. This chapter answers a practical question. How do you test code whose worst bugs appear only under duplication, delay and reordering, without building a suite that is slow, flaky or ignored?

The problem, and why it is hard

Chapter 1 argued that the happy path is the one you can test without trying and the one you are inclined to believe. Chapters 2 to 14 then built defenses against failures that the happy path never shows. Defenses are code, and code that is never exercised under the failure it handles is a hope. Chapter 3 introduced a failure model with a "verified by" field for each entry. This chapter is where that field gets filled in.

Three properties make this testing hard.

The failures are interleavings, not inputs. A function test supplies an input and checks an output. The duplicate-reservation bug needs a particular timing: two deliveries overlapping in time, or a commit followed by a crash and a redelivery. No input value triggers it. The test has to control when things happen, which means the test harness must own the broker's behavior, the clock and the crash points.

Asynchrony hides the moment of truth. After a handler runs, the effect may land in another service seconds later. A test cannot assert immediately, and it cannot know how long to wait. Sleeping a fixed time is wrong in both directions: too short and the test is flaky, too long and the suite is slow.

Fakes lie in exactly the places that matter. Idempotency rests on a unique constraint. Atomicity rests on a transaction. Ordering rests on partition assignment. A fake that skips these gives you a green suite and no information. The cost of realism is speed, so the craft is deciding where to pay it.

Adapting the test pyramid

The classic test pyramid still holds: many fast tests at the bottom, few slow ones at the top. What changes is what each layer is for, and which fault each layer is responsible for catching.

Layer	What runs	Real or fake	Catches
Handler tests	One handler against a real database	Real database, no broker	Wrong effect, missing inbox, broken atomicity, bad duplicate and order handling

Layer	What runs	Real or fake	Catches
Contract tests	Event shapes between producer and consumer	No services at all	Schema changes that break a consumer
Component tests	One service with a real broker, outbox and relay	Real broker, other services stubbed	Wiring: outbox to relay to topic, acks, retries, dead-lettering
Flow tests	A few services together, a failure injected	Real everything, small scale	Cross-service behavior such as compensations and timeouts
Production probes	Synthetic events in the live system	Real	Deploy and configuration problems no pre-production environment shows

Two choices in the table deserve a sentence each. The handler layer uses a real database engine, because the inbox, the outbox write and the version check all depend on constraints and transactions that a fake would have to reimplement, and reimplementing them is how the Inventory suite got fooled. The broker, by contrast, is left out at that layer. A handler is a function from a message and a state to a new state and some outgoing messages. If you can call it directly, you can test it quickly.

The shape of the pyramid also follows from where the failures live. Most of the defenses in this book are consumer-side or producer-side logic, which the handler layer reaches cheaply. Only the wiring needs a broker, and there the number of tests is small because each one proves a property of the configuration, not of the business logic.

Testing handlers

A handler test should state the contract in the language of the book: given this message and this state, the effect happens once and the outgoing event is written to the outbox. Then the same test gets a second act, where the message arrives again.

```
function testReservationIsIdempotent():
  db = freshDatabase()
  seedStock(db, "sku-17", available: 5)
  message = buildEvent("OrderPlaced", order_id: "order-1",
                      lines: [{ sku: "sku-17", qty: 2 }])

  handleOrderPlaced(db, message)
  // same event_id: the broker redelivered it
  handleOrderPlaced(db, message)

  assert stockAvailable(db, "sku-17") == 3
  assert reservationCount(db, "order-1") == 1
  assert outboxCount(db, "StockReserved", "order-1") == 1
```

The last assertion is the one teams forget. A duplicate that does not re-serve twice but emits `StockReserved` twice still causes harm downstream, and checking the outbox catches it.

That test covers sequential duplicates, which the fast-path check in the in-box handles. The dangerous case is the concurrent duplicate, where two workers both pass the check. To test it, the harness runs two handlers at the same time and uses a barrier to hold both right before the transaction, so that the interleaving is forced instead of hoped for.

```
function testConcurrentDuplicatesCommitOnce():
  db = freshDatabase()
  seedStock(db, "sku-17", available: 5)
  message = buildEvent("OrderPlaced", order_id: "order-1",
                      lines: [{ sku: "sku-17", qty: 2 }])
  barrier = newBarrier(parties: 2)
  setHook("before-transaction", barrier.wait)

  results = runConcurrently(
    handleOrderPlaced(db, message),
    handleOrderPlaced(db, message))

  // one commits; the other fails on the unique constraint and
  // would be redelivered, which is allowed
  assert stockAvailable(db, "sku-17") == 3
  assert reservationCount(db, "order-1") == 1
  assert results.committed == 1
```

This test fails if someone removes the unique constraint and relies on the read-then-insert check, which is the exact regression that is hardest to spot in review. Chapter 7 explained why the constraint is the safety net. A test like this makes the explanation executable.

Crash points work the same way. If the handler calls `crashPoint("before-ack")` where the ack would go, a test can arm that point and raise a simulated crash there.

```
function testCrashBetweenCommitAndAck():
  db = freshDatabase()
  seedStock(db, "sku-17", available: 5)
  message = buildEvent("OrderPlaced", order_id: "order-1",
                      lines: [{ sku: "sku-17", qty: 2 }])
  armCrash("before-ack")

  crashed = false
  try:
    handleOrderPlaced(db, message)
  on SimulatedCrash:
    // the process died after commit; the broker saw no ack
    crashed = true
    assert reservationCount(db, "order-1") == 1
    // if the crash point was renamed, the test must fail, not pass
    assert crashed

    // a crash point fires once per arming; disarm it for the
redelivery
    disarmCrash("before-ack")
    handleOrderPlaced(db, message)
    assert reservationCount(db, "order-1") == 1
    assert stockAvailable(db, "sku-17") == 3
```

Read the assertion inside the `on` block. It checks that the commit survived the crash, which is half of the story. The other half is the redelivery that follows. A crash test that checks only the second half would also pass for a handler that never committed anything.

Injecting duplicates, reordering and delay

Individual tests encode the cases you thought of. The failure model also contains the ones you did not, so I would add a layer of tests that perturb a whole stream of events and check that the outcome does not change.

The idea is a convergence property. Take a recorded, correctly ordered stream for one aggregate. Compute the expected final state by folding the events directly, with no broker involved. Then deliver the same events through the handler in a perturbed schedule, with some shuffled within a window and some duplicated, and assert that the final state equals the expected one.

```

function testOrderViewConvergesUnderAnyDelivery(seed):
    // versions 1 to 6, true order
    events = recordedOrderStream("order-1")
    // reference, no broker, no handler
    expected = foldEvents(events)
    random = randomSource(seed)
    // a failure must be reproducible
    log("seed", seed)

    schedule = shuffleWithin(events, random, window: 3)
    schedule = duplicateSome(schedule, random, rate: 0.3)

    db = freshDatabase()
    for each message in schedule:
        handleOrderEvent(db, message)

    // assumes held events are released when a gap closes (Chapter 9)
    assert heldEventCount(db, "order-1") == 0
    assert loadOrderView(db, "order-1") == expected

```

Run it for a few hundred seeds in CI. Two details keep it honest. The seed is logged, so a failure on seed 4172 is a failure you can rerun on a laptop instead of a story. And the reference result comes from a different code path than the handler, so a bug in the handler cannot also appear in the expectation.

Delay is the third perturbation. A delayed message is one that arrives after its successors, or after a timeout has fired. Time is the thing to control here: handlers and sagas that read the system clock directly cannot be tested for timeouts without waiting. I would pass the clock in as a dependency, and let the test advance it. Chapter 10 listed the states a saga can be in. A saga test then becomes a table: for each state, deliver each event that could arrive, including the late ones, and check the resulting state and the compensation emitted.

Saga state	Arriving event	Expected outcome
AWAITING_RESULTS	PaymentAuthorized	Stays in <code>AWAITING_RESULTS</code> , waiting for stock
AWAITING_RESULTS, payment held	Timeout fires	<code>COMPENSATING</code> ; commands <code>ReleaseStock</code> and <code>VoidAuthorization</code> written to the outbox

Saga state	Arriving event	Expected outcome
COMPENSATING or CANCELLED	Late <code>StockReserved</code>	No saga transition. A late <code>StockReserved</code> means Inventory reserved before <code>ReleaseStock</code> reached it, so that command releases the stock. Had the release arrived first, its tombstone would have refused the reservation (Chapter 10)
COMPLETED	Duplicate <code>PaymentCaptured</code>	Ignored

The third row is the one that matters. Late arrival after a timeout is the case that sagas get wrong, because the happy path and the timeout path were each tested alone and the collision never was. The table has to be written by someone asking "what else could arrive in this state?" and not by someone tracing the flow diagram.

Note: If your tests rely on a randomized schedule, keep a few failing seeds as fixed regression cases once they are found. A bug that was reproduced once should stay reproduced.

Contract tests and compatibility checks

Handler tests build their own events, which means they test the consumer against the consumer's idea of the producer. When the producer changes a field, nothing in the consumer's suite notices. Chapter 12 described compatibility rules and a schema registry that enforces them. That covers structure. Contract tests cover the part the registry cannot see: what each consumer actually depends on.

In a **consumer-driven contract**, each consumer writes down the fields and types it reads from an event, and the producer's pipeline checks that its schema still provides them. The direction matters. The producer cannot know which fields are load-bearing, and the consumer can. A contract is therefore a small file owned by the consumer and stored where the producer's CI can read it.

```

contract inventory-reads-order-placed:
    consumer:      inventory
    event_type:    OrderPlaced
    schema_version: 1
    requires:
        payload.order_id      text, required
        payload.lines[].sku    text, required
        payload.lines[].qty    integer, greater than zero
    tolerates:      unknown fields
    sample:         minimalEventFrom(requires)

function verifyContracts(contracts, producerSchemas):
    for each contract in contracts:
        schema = producerSchemas.versionOf(contract.event_type,
                                              contract.schema_version)
        for each requirement in contract.requires:
            // optional is not enough: a consumer that reads the field
            // must be able to rely on it being there
            if not schema.guarantees(requirement):
                raise ContractBroken(contract.consumer, requirement)

```

Two halves are needed. The producer's pipeline runs `verifyContracts` against every contract on file. The consumer's pipeline decodes each sample event in its own contracts with its real deserializer and runs the handler on it. Together they catch a producer that stops providing a field and a consumer that starts reading a field it never declared. The second catch holds only if the sample is built from the contract and contains only the declared fields; a sample copied from the producer's full event lets the consumer read an undeclared field and still pass.

Samples should cover every version still in retention, as Chapter 12 recommended. Replay reads whatever is there, and a consumer that handles only the newest shape fails on the first old event. This is also where the registry's compatibility check belongs in CI: a producer's schema change should fail the build before it can be registered, not after it reaches the topic.

Be honest about the limit. A contract proves that `amount_minor` exists and is an integer. It does not prove that it still means "the amount in minor units". Changing meaning without changing structure passes every contract test, and only a human reviewer, or a behavioral test in the consumer, notices. I would not treat green contracts as a statement about meaning.

Eventually-consistent assertions

An end-to-end test publishes something and then asserts on an effect elsewhere. The effect lands eventually, so the assertion needs a form that tolerates that. The standard form is a poll with a deadline.

```
function eventually(check, timeout, interval):
    deadline = now() + timeout
    while true:
        observation = check()
        if observation.satisfied:
            return
        if now() >= deadline:
            // report what was last seen, not only that time ran out
            raise AssertionError(observation.description)
        sleep(interval)
```

The structure matters more than it looks. The check runs once more after the deadline is reached, so a condition met during the last sleep still passes, the interval is short relative to the timeout, and the failure message carries the last observation. A message that says "timed out" tells you nothing. One that says "order-1 stuck at AWAITING_RESULTS, 1 held event" tells you where to look.

Flaky async tests have a short list of causes, and most are fixable at the cause.

- **Fixed sleeps.** Replace with `eventually`. The timeout becomes a ceiling and the test finishes as soon as the condition holds.
- **Timeouts tuned to a fast laptop.** Set the ceiling generously, since it costs nothing when the condition is met quickly, and keep it separate from performance expectations. A latency budget is a different test with a different environment.
- **Shared state between tests.** A leftover message in a shared topic makes the next test see an event it did not publish. Give every test its own correlation id and filter by it, or its own topic or queue.
- **Order assumed where none is promised.** If a test asserts on the order of two events from different partitions, it asserts on something the broker never promised (Chapter 9).
- **Asserting that nothing happened.** "No second email was sent" cannot be polled into success, since waiting longer might still show one.

Prefer to drive the system to quiescence, then assert. Quiescence is a property of the whole path, not of one consumer: the outbox is empty, the relay is idle, the lag of every consumer involved is zero, no retry or timer is scheduled, and the retry and delay queues are empty. Only then is there nothing left that could produce the effect, including a downstream hop whose message is still in flight.

For the last case, the harness needs a way to say "run until nothing is pending". In a handler or component test, the harness can pump the consumer until the queue is drained and the injected clock has no timers left to fire. This is both faster and more deterministic than waiting for wall-clock time, and it is the main reason I keep most behavioral tests at the handler layer and use real time only where wiring is the thing under test.

Fault injection and chaos

Everything so far tests code in isolation. The next question is whether the assembled system behaves as the failure model claims. Fault injection puts a failure into a running environment on purpose and checks the system's response.

Start with the failure model from Chapter 3 and pick entries, not tools. For each, write the hypothesis in the entry's own terms: "if a consumer is killed between commit and ack, the redelivery changes nothing". Then inject it: kill the process at that moment, pause the relay, delay the broker's responses, cut the connection between a service and its database, restart a broker node mid-publish. Afterward, check the invariants, not the logs: stock matches reservations, the ledger has one entry per capture, no outbox row is stuck.

Add recovery drills to that list, because a defense that has never been exercised is a hope. A backup that has never been restored is the classic example. Restore one into a scratch environment and check what a restore does to the outbox, the inbox and the consumer offsets, which Chapter 14 covers: outbox rows resurrected, inbox rows rewound so that duplicates are no longer recognized, offsets that no longer match. Fail over to a standby cluster, or rehearse it, and compare the recovery time and the data lost with the recovery objectives chosen for that flow in Chapter 14. And test catching up: pause a consumer long enough to build a backlog,

resume it while traffic keeps arriving, and check that the backlog clears before the flow's deadline and before retention expires. That checks the drain rate from Chapter 13 and the limits on in-flight work from Chapter 6.

I would run this in layers of increasing blast radius. Flow tests in a dedicated environment inject faults on a schedule in CI, small and repeatable. A periodic game day in a larger environment injects them as a team exercise, partly to find bugs and partly to find out whether anyone notices. The alerts from Chapter 13 are as much under test as the code. If you killed a consumer and no alert fired, you found a bug in monitoring.

Chaos in production is a possible next step, not a starting point. It makes sense only when the system has been shown to survive the same fault in pre-production, when blast radius can be bounded, and when someone can stop the experiment instantly. A team without those should not begin there, and I would not be embarrassed to stay at game days for a long time. The value of a fault-injection practice is in the cycle of hypothesis, injection and check, not in how dramatic the injection is.

Warning: A fault injection run that shares a broker, a database or a payment provider account with real traffic is an outage with a plan. Isolate the environment, or restrict injection to synthetic traffic, before turning anything off.

Replay as regression testing

An event log is a recording of what actually happened, and replay, the deliberate re-reading of history, turns it into a regression suite. Take a slice of real events, run them through the new version of a consumer in a scratch environment, and compare the derived state with what the old version produced. Differences are either intended, as in a bug fix, or regressions. Replay also has an operational use, recovering after a gap or a restore by re-reading history, and Chapter 14 covers that. This section is only about using it to test.

This catches things handwritten fixtures never will: event shapes from three schema versions ago, sequences nobody imagined, a field that was always present in the tests and is null in a small fraction of history. It is

also the closest thing to a test of Chapter 12's claim that old events still decode.

A workable procedure has five steps.

1. **Select a slice.** A window of time, or the full history of a sample of aggregates. A sample that includes every schema version, plus known incidents, beats a random one.
2. **Mask it.** Replayed events contain buyer data. Anonymize before they leave their original boundary, or keep the scratch environment under the same controls as production.
3. **Disable effects and emitted events.** The replaying consumer must not charge cards, send emails or call carriers. Effects go to stubs that record what they were asked to do, and the stubs' records are part of the output. Its outbox and relay must publish to a scratch broker, or be captured, and never point at production topics, or the replay republishes history into live consumers.
4. **Run the old and new versions on the same slice.** Compare derived state and emitted events, aggregate by aggregate.
5. **Classify every difference.** Intended, harmless or a regression. A difference that nobody can explain is a regression until shown otherwise.

Replay applies to the log family, where history is still there. On a classic queue, consumed messages are gone. The equivalent is a captured sample: a tap that copies a fraction of messages, anonymized, to a store you control. It is a weaker tool because the capture is only as good as the sampling, but it serves the same purpose.

Replay also has a cost limit. Large histories take real time and storage, and the comparison tooling is something your team has to build and keep working. I would reserve full-history replay for changes that touch how events are interpreted, and use a smaller slice for routine releases.

Verifying in production

No pre-production environment reproduces production's traffic shape, data, configuration and quirks. Some failures, such as a missing permission on a topic after a deploy, a stale secret or a misrouted queue, appear

only in the real system. The answer is not to give up on pre-production tests but to add a last layer that runs where the failures are.

A **synthetic event** is a deliberately injected, explicitly labeled piece of traffic that travels the real pipeline and is checked at the end. For Parcelly, it is an order placed every few minutes by a probe, flowing from `OrderPlaced` through to `ShipmentScheduled`.

```
function runOrderProbe():
  probeId = newId()
  // recorded first, so a crash of the probe itself leaves a trail
  recordProbe(probeId, startedAt: now())
  try:
    // goes through the real API, outbox, relay and broker; the
    flag
    // travels in a header and is copied like correlation_id
    orderId = placeSyntheticOrder(probeId, headers: { synthetic:
    "true" })
    eventually(
      check: flowReached(orderId, "ShipmentScheduled"),
      timeout: probeBudget,
      interval: probeInterval)
  finally:
    // runs on success, on timeout and on error alike
    cancelSyntheticOrder(probeId)
```

The label is the safety mechanism, and it is a dedicated `synthetic` header, not a prefix on `correlation_id`. An identifier that also controls behavior is a trap, and Chapter 13 asks for one uniform id format, `order-81724`, so that searching by correlation works everywhere. The price is that a hop which forgets to copy the header turns a probe into real traffic without any error. Every service copies it with the same helper that copies `correlation_id`, the probe also uses a dedicated synthetic buyer account that Payments and Shipping recognize independently, and a test drops the header on purpose and checks that no real charge or shipment results.

Each service recognizes the flag and changes its edges, not its core logic: Payments talks to the provider's sandbox, Shipping skips the carrier hand-off, Notifications suppresses delivery, and the Ledger books to a test account that finance excludes from reports. These branches are code that runs only for probes, so a mistake in them is a bug that only production exposes. Inventory needs the same treatment, or every probe reserves real stock: reserve against a dedicated synthetic SKU, or skip the reservation

for labeled orders. I would test the branches themselves at the handler layer, and test the probe by breaking something on purpose and checking that it fails.

Cleanup has one more trap. A probe that times out has not necessarily failed to finish; its events may still be in flight, and a late `StockReserved` or `PaymentAuthorized` can arrive after the cancellation. That is why the probe is recorded up front and every service keys on the `synthetic` flag: a periodic sweep can find probes older than the budget and cancel whatever they left behind, and the cancellation is itself idempotent, so running it twice is harmless.

A probe also needs a budget, and the budget needs an owner. A failing probe pages someone, so the budget has to be generous enough to avoid false alarms and tight enough to be meaningful. Chapter 13 covers end-to-end latency and how to alert on it. The probe is the cheapest source of that measurement, since it exercises the whole path, including the parts that no real order happened to touch in the last hour.

Production verification includes the reconciliation jobs from Chapter 13. A reconciliation run is an invariant test over real data, executed continuously, and it catches the failures that no one thought to simulate. I count it as the last layer of the pyramid.

Trade-offs and costs

Realism costs time. Real databases, real brokers and injected faults make tests slower. The suite that was fast was fast because it skipped what mattered. I would budget the slow layers explicitly: handler tests in seconds, component tests in a few minutes, flow tests and fault injection on a schedule rather than every commit.

Contract tests add coordination. Each consumer owns files that live in the producer's pipeline, and someone has to keep them current. For a system with two services and one team, a shared test of the schema may be enough. Contract testing begins to pay when producer and consumer teams deploy independently, and it earns little before that.

Randomized schedules add investigation work. A failing seed is a puzzle and a real defect at once. Teams that dislike it end up lowering the number of seeds until the test no longer finds anything. The cure is logging seeds and keeping fixed regressions, not abandoning the technique.

Chaos and probes have a blast radius. Both touch live or shared systems. When not to use them: when the system does not yet survive the fault in isolation, when no one can stop the run immediately, or when the labeled traffic cannot be kept apart from real money and real buyers.

Replay is expensive to set up. The infrastructure for masking, effect stubs and comparison is its own project. A small system with a short history gets more from handler tests and crash points. I do not know where your break-even point is, since it depends on how often consumers change and how costly a wrong interpretation of history would be, but a team that has already had one incident from an unexpected old event can usually guess.

Common mistakes

- **Testing against a broker fake that never misbehaves.** It is tempting because it is fast and the first tests pass. Make the fake duplicate, delay and reorder, or put the failure into the handler tests with explicit calls, as in this chapter.
- **Faking the database's constraints.** A hand-written fake does not enforce uniqueness, isolation or atomicity, so the tests pass whether or not the defense exists. Use the real engine, in a container or a scratch instance, for any test that depends on a constraint.
- **Sleeping for a fixed time.** It looks like the simplest way to wait. Poll for the condition with a ceiling and report the last observation on failure.
- **Never testing the crash between commit and ack.** It feels exotic, yet it happens on every deploy. Add a crash point to each handler and a test that arms it.
- **Testing only the final state, not the outbox.** A duplicate that does not change state can still emit a second event. Assert on what was written to the outbox.

- **Contract tests that check presence but not requiredness.** A field that is optional in the producer's schema can be absent. Make the contract say "required" and check it.
- **Randomized tests without seeds.** A failure that cannot be reproduced gets deleted. Log the seed and keep failing seeds as fixed cases.
- **Trusting a backup that was never restored.** The backup job reports success, so the defense feels tested. Restore it into a scratch environment on a schedule and check the outbox, inbox and offsets afterward.
- **Running probes against real money.** Synthetic traffic that reaches the live payment provider or a real carrier is not a probe. Route labeled traffic to sandboxes at the edge, and test that routing.

Key takeaways

- Tests that assume a broker delivering each message once, in order, test a system that does not exist, so every defense in this book needs a test that applies the failure it defends against.
- Handler tests belong on a real database engine, because the inbox, outbox and version checks depend on constraints and transactions that fakes do not reproduce.
- Duplicates, reordering and delay are best tested as a convergence property over a perturbed stream, with the seed logged so that any failure can be reproduced.
- Consumer-driven contracts catch structural breaks before deploy, but they cannot see a change in meaning, so a human review of what a field is supposed to say remains part of the process.
- Replace fixed sleeps with polling against a condition and a ceiling, assert on quiescence rather than on silence, and inject the clock so that timeouts can be tested without waiting.
- Fault injection should start from the failure model's entries and check invariants afterward, increasing in blast radius only as the system proves it survives each step, and it should include recovery drills for restores, failovers and catching up.
- Replay of real history and synthetic probes in production catch what handwritten fixtures and pre-production environments cannot, and they need masking, effect stubs and labeled traffic to be safe.

A system that has been tested this way can be trusted a little more, though never completely, and that leaves the question of what a team should check before it trusts a design at all. Chapter 16 gathers the book into a reliability checklist, and closes with what the checklist cannot do.

Chapter 16 — A Reliability Checklist and Closing Thoughts

Imagine a design review at Parcelly on a quiet Thursday. The team is about to add a new flow: sellers can offer partial refunds, and Payments will emit a new event for it. The diagram is clean. Someone has drawn the arrows, named the event and listed the consumers. The meeting is scheduled for 30 minutes, and it looks as if it will end in 20.

Then a tech lead asks a plain question: "What happens if the refund is issued and the event is never published?" The room goes quiet, not because nobody knows the pattern, but because nobody had asked. The outbox exists elsewhere in the codebase. The idempotent consumer exists elsewhere too. Whether this new flow uses them is a different matter, and the diagram does not say.

This is how reliability usually erodes. The team knows the patterns. What fades is the habit of checking, for every new flow, that each pattern is actually in place. So the question for this last chapter is practical: what would you ask, flow by flow, to find out whether the system can keep its promises on a bad day, and who has to own the answers?

The problem, and why it is hard

Reliability is a property of seams, and seams are not visible in any single file. A producer can be correct, a consumer can be correct, and the flow between them can still lose events, because the handoff was never examined. Chapter 1 made this argument at the start of the book, and the chapters since have worked through the seams one at a time.

The difficulty now is that the knowledge is spread across 15 chapters. Nobody holds that much in their head during a review. Under deadline pressure, the questions that get skipped are the ones whose answer is "it

works until the day it does not": the dedup window, the dead-letter owner, the compatibility check, the alert that fires on age and not on count.

A checklist is the usual answer, and it has known limits. It cannot think. Someone who does not understand an item can still tick it, and a completed list then lends false confidence. I would use the list below as a prompt for conversation, not as a gate a form can pass. Every item points to the chapter that explains why it is there, so a reader who disagrees can find the reasoning and argue with the premise.

Treat each item as a question about a specific flow, such as the Parcelly order flow, and not about the system in general. "We use an outbox" is not an answer. "Orders writes `OrderPlaced` to its outbox in the same transaction as the order row" is.

The method behind the list

The list is not a new idea. It is the thesis of the book applied to a review: reliable event-driven systems are built around **failure semantics**, not message brokers: for a given seam, what can fail, how you would observe it, and which guarantee survives it.

That makes three questions, and every item below is one of them asked about one seam. An item that names a failure asks what can fail. An item that names a metric, an alert or a test asks how you would observe it. An item that names a setting, a constraint or a pattern asks which guarantee survives it. A seam whose items answer only one of the three is half finished, and a seam with no answer to the second question will fail without telling anyone.

The checklist

The list is grouped by where the failure lives. The right-hand column names the chapter to reread when an answer is "no" or "I am not sure".

Producer

Check	Chapter
The state change and the event are written in one local transaction, with no direct publish in the business path.	4

Check	Chapter
The outbox row carries the full envelope, and the <code>event_id</code> is created at write time and never regenerated.	4
The team has written down that publication is at-least-once and that duplicates are the price.	4
The relay publishes first and marks second, and selects by state, not by an increasing id cursor.	5
Several relays, if any, claim rows safely and keep one aggregate's rows on one worker.	5
The age of the oldest unpublished row is measured, and only published rows are purged.	5

If a team can answer only one question in a review, I would want it to be the first, because every later guarantee assumes the event exists at all. Where a flow does not need an outbox, because its steps can share a local transaction, the team should say so and say why (Chapter 1).

Broker

Check	Chapter
For each important event, the team has recorded what "published" means: the replication and flush settings behind a producer acknowledgment.	2, 3
Acknowledgment or offset commit happens after the effect, so the semantics are at-least-once and not an accidental at-most-once.	2
Visibility timeouts, session timeouts and in-handler retry time are set together, so slow work is not redelivered while it still runs.	2, 6
The partition key identifies the aggregate, and the team knows what ordering its broker family offers.	9
Every producer, in every language, uses the same partitioner and key encoding, or the relay computes the partition itself.	9
Retention exceeds the longest realistic outage plus the time to notice it.	3, 13
Consumers stop gracefully on deploy, so rebalancing is not a routine source of duplicates.	3

Most of these items are settings, not code. They stay at their defaults until an incident shows what the defaults were, and by then they are hard to trace.

Consumer

Check	Chapter
The inbox row and the business change commit in one local transaction, with a unique constraint on the key.	7
The dedup window outlasts retention, redrive and planned replay, and a business-key constraint covers the domain table beyond it.	7
External side effects carry a deterministic idempotency key derived from the work, not from the attempt.	7
Failures are classified as transient, permanent or unknown, and retries use exponential backoff with jitter, a deadline and a shared budget.	6
In-flight work is capped, concurrency drops when the dependency slows down, and the consumer pauses instead of failing, so overload shows up as lag; a limit at the ingress rejects or defers new work with a clear signal to the caller.	3, 6
A poison message ends in a DLQ with body, headers, source position, error, attempt count and consumer version, published before the original is acknowledged.	8
A consumer stuck on one record is detected, by lag with no throughput and by a per-position crash counter, and does not loop until someone notices.	8, 13
Redrive preserves the original <code>event_id</code> , runs in rate-limited batches and has a pause switch.	8
Per-aggregate order is protected by keyed lanes, version checks and hold-back buffers that have timeouts.	9
Every workflow step before the pivot has an idempotent compensation, and every wait has a deadline.	10

This is the longest group because most of the book's patterns land here. If it feels heavy, that is accurate: at-least-once delivery hands the consumer the burden.

Contract

Check	Chapter
Each event states a fact, is named in the past tense, and has one owning service.	11
The integration event is a deliberate public contract, mapped from the domain event, with no personal data or secrets in the payload.	11
A command is named in the imperative, has one intended recipient, carries a <code>command_id</code> , has a reply path so that a refusal reaches the sender, and is never called an event; the sender waits with a deadline of its own.	10, 11

Check	Chapter
<code>correlation_id</code> is copied through the flow and <code>causation_id</code> points at the direct trigger.	11
The envelope's wire types, such as timestamps, identifiers and money, are fixed for every language that reads or writes it.	11
Every change is classified as compatible or breaking, and compatibility is checked in the deploy pipeline.	12
A change of meaning is reviewed by a human, because no tool detects it.	12
Consumers ignore fields they do not use and tolerate values they have not seen, a hop that forwards a message preserves unknown fields, and an event whose <code>schema_version</code> they do not know is rejected or parked.	12
Breaking changes migrate as a sequence of non-breaking steps, with rollbacks planned in the same document, and the consumers still reading the old version are known.	11, 12

Operations

Check	Chapter
Trace context and <code>correlation_id</code> survive every asynchronous hop, so one order's story can be rebuilt from its identifier.	13
Consumer lag, message age, outbox age, retry rate and DLQ arrivals are measured, and alerts fire on age and rate.	13
Reconciliation jobs compare the source of truth with derived state on a schedule, and repairs are reviewed before they run.	13
Business invariants, such as "every captured payment has a ledger entry", are checked directly.	13
Handlers are tested with duplicates, reordering and delay, and contract tests run in CI.	15
Fault injection and replay-based regression tests exist for the flows that matter most.	15
Each store has a stated recovery point objective (RPO) and recovery time objective (RTO), and the team knows which store goes back in time on a restore.	14
A replay procedure names the consumer groups, the timestamp and its margin, the side effects it suppresses, and the dedup window every replayed range must fit inside.	7, 14
A broker migration plan uses the relay as the single switch point, dual-publishes with consumer deduplication, and has cutover and rollback steps.	14
Restores and game days have been rehearsed, and the runbooks below were written from what the rehearsal showed.	14, 15

Check	Chapter
A written failure model names owners, tests and review dates.	3

Questions for a design review

The checklist verifies a design that exists. These questions help shape one that does not exist yet. I would ask them in this order, because each answer constrains the next.

1. **What is the fact, and who owns it?** If you cannot name the event in the past tense and name one owning service, the design is not ready for a broker (Chapter 11).
2. **Does this need an event at all?** If the steps can share a local transaction, or the caller needs an immediate answer, a synchronous call may be the better tool (Chapter 1).
3. **What happens between the commit and the publish?** Ask for the crash window, not for the pattern. "We publish right after commit" is a loss waiting for a deploy (Chapter 4).
4. **What happens when this message arrives twice, and what when it arrives late?** They are different failures with different fixes (Chapters 7 and 9).
5. **What is the bound on "eventually"?** Name a duration, an alert and a user-facing state for the wait (Chapter 1).
6. **What happens when this message can never succeed?** Ask where it ends up, who sees it and how it gets back (Chapter 8).
7. **What happens when the consumer, or the dependency behind it, is slower than the producer?** Ask what is capped, what pauses and which work waits, and who owns that choice (Chapters 3 and 6).
8. **If a later step fails, what undoes the earlier ones?** Name the compensation and the pivot after which nothing is undone (Chapter 10).
9. **How will we change this contract in a year?** If nobody has thought about it, the answer will be improvised in an incident (Chapter 12).
10. **How will we know it is broken, and how will we know it is repaired?** Detection and verification are separate questions (Chapters 13 and 15).

11. **If a store, the broker or a region is lost, how much do we lose, and how long until we are back?** Ask for an RPO and an RTO per store, and for the replay or restore that follows (Chapter 14).
12. **Who gets paged, and what do they do first?** If the answer is a name, you have an owner. If it is "the team", you may not.

Some of these have no clean answer on a given day, and that is fine. "We do not know yet, and here is how we will find out" is a better outcome than a confident answer nobody checked.

The organizational side

Every pattern in this book has a human half. An outbox needs someone who watches the age of its oldest row. A DLQ needs someone who reads it. A schema registry needs someone who decides what "compatible" means. When the human half is missing, the pattern still exists on the diagram and no longer does its job.

Ownership

I would write down three kinds of ownership.

- **Event ownership.** Every event has one owning service and team, which documents what the event means and who consumes it (Chapter 11).
- **Operational ownership.** Every DLQ, parking lot, reconciliation report and alert has a named owner. An alert owned by nobody is a notification, and notifications get ignored.
- **Cross-service flows.** The Parcelly order flow crosses six services. Some team has to own the flow as a whole, including its timeouts and stuck workflows, even when no single service does (Chapter 10).

Runbooks

A runbook is what you write assuming the reader is tired and has not seen the system for six months. For each alert, say what it means, what to look at first, which actions are safe, which are not, and whom to call if the first steps do not help.

Some runbooks are tied to an alert and some to an event that happens so rarely that nobody on the team has done it recently. Chapter 14 adds the second kind. Its section on rehearsing recovery lists eight questions that a recovery runbook has to answer, written down and not carried in someone's head:

1. Which store is affected, and which went back in time?
2. What is the recovery point, and how was it determined?
3. Which consumer groups must be reset, to what timestamp and with what margin?
4. Which effects must be suppressed during the replay, and by what mechanism?
5. Is every range to be re-read inside every affected consumer's dedup window?
6. What is paused, in what order, and who resumes it?
7. Which reconciliations confirm that the recovery worked, and who reads them?
8. How is the whole thing undone if it is going wrong?

The table lists the events I would write a runbook for. The last column marks which of the eight questions I would expect each one to answer. It is a starting point, and a team's own system will add questions of its own.

Event	The dangerous action	Questions
A consumer stuck on a poison message (Chapter 8)	Skipping the position without a record of it, or raising the retry limit until the message passes	6, 7, 8
A consumer whose lag is approaching retention (Chapters 13 and 14)	Letting the messages expire unread, or resetting forward without anyone deciding to	2, 3, 5, 7
Offset reset and replay (Chapter 14)	Replaying with side effects live, or past the dedup window	3, 4, 5, 6, 8
Redrive (Chapter 8)	Sending a whole DLQ back at once into a dependency that is still struggling	5, 6, 8
Repair and backfill (Chapter 13)	Running an unreviewed script against production state	4, 7, 8

Event	The dangerous action	Questions
Restoring a database from backup (Chapter 14)	Resuming consumers without resetting their offsets, or reusing identifiers the re-wound sequence hands out again	1, 2, 3, 4, 5, 6, 7, 8
Broker data loss or failover (Chapter 14)	Resuming groups at positions that no longer mean what they meant	1, 2, 3, 5, 6, 7
Region evacuation (Chapter 14)	Promoting a standby without knowing how far behind it was	1, 2, 3, 6, 7, 8
Broker migration cutover (Chapter 14)	Moving consumers before the dual-published copies are deduplicated, or with no way back	3, 5, 6, 8

Two of these deserve particular care. Redrive comes first, because a careless one turns a contained failure into a flood (Chapter 8). Repair and backfill come second, because a script that corrects derived state is a production change and deserves a review, a dry run and a record of what it did (Chapter 13).

On-call

On-call for an event-driven system differs from on-call for a request-response service in one way that matters: the failure is often quiet. A synchronous service returns errors and a graph turns red. An event-driven system may return nothing at all, because the message is sitting in a queue, waiting. Alerts on age, lag and unprocessed work are the primary signal, and the on-call engineer needs enough access to look inside the queue, the DLQ and the outbox without waiting for permission.

Note: Practice the repair before you need it. A team that has redriven a test DLQ once, in daytime, with a colleague watching, will handle the real one differently from a team that has only read the run-book. Chapter 14 makes the same point for restores and game days, and Chapter 15 covers how to design the drills.

Trade-offs and costs

A checklist has costs, and the first is attention. A list of 43 items applied to every flow is too long, and a team that applies all of them to a low-stakes notification flow will conclude that the checklist is the problem. I would weight the list by the cost of the failure. A flow that moves money or changes stock deserves the whole list. A reminder email deserves idempotency and a DLQ owner, and perhaps little else.

The second cost is false assurance. A completed list says that someone considered each item, not that the system is correct. Chapters 13, 14 and 15 exist because considered designs still fail in ways their authors did not imagine. Treat the checklist as the cheapest layer of defense, not the last.

The third cost is drift. A list written once becomes a record of what the team believed at one moment. Revisit it after each incident, and delete items that have stopped earning their place. A short list that people use is worth more than a long one that people sign.

When not to use it: do not use the checklist to replace judgment, and do not demand it for throwaway prototypes. If a flow will be deleted next month, the cost of the full list exceeds the cost of the failure. Say so explicitly, and write down the date by which the prototype either dies or gets a proper review.

Common mistakes

- **Treating the checklist as a gate.** A gate can be audited by counting ticks, which is the temptation. The result is a filled-in form and an undiscussed design. Use the items as questions, and record the answers that surprised someone.
- **Checking patterns instead of flows.** "We have an outbox" feels like progress. Ask whether this flow uses it, and name the table and the transaction.
- **Applying everything everywhere.** It feels thorough, then leads to fatigue and abandonment. Weight the list by the cost of failure, and say which items you skip and why.
- **Leaving the human items for later.** Owners, runbooks and review dates seem less urgent than code, and they are the first thing missing

when an alert fires at night. Assign them in the design review itself.

- **Never revisiting the answers.** A flow that passed review a year ago has gained consumers, versions and people since. Put a review date on the answers, as Chapter 3 suggests for the failure model.
- **Believing the list is complete.** Your system has failure modes this book does not name. Add them to your own list when you meet them.

Key takeaways

- Reliability erodes through skipped questions more often than through unknown patterns, so a fixed set of questions asked on every new flow protects a design better than expertise in the room.
- Every item on the checklist is one of the book's three questions, what can fail, how you would observe it and which guarantee survives it, asked about one seam, so each item should be answered for a specific flow and not for the system in general.
- Weight the list by the cost of failure: flows that move money or stock deserve every item, and low-stakes flows deserve the few that prevent duplicate effects and silent loss.
- Every pattern in this book has a human half, so owners, runbooks, alert recipients and review dates belong in the same design review as the code.
- Event-driven failures are quiet, so on-call depends on alerts for age, lag and unprocessed work, and the rare events, a restore, a replay, a region loss and a broker migration, need runbooks that have actually been run.
- A completed checklist shows that someone thought about each item, not that the system is correct, which is why reconciliation and testing exist as further layers.
- Reliability is not a feature you ship once but a discipline you keep practicing, in small questions, written answers and repairs rehearsed before they are needed.

Parcelling never becomes reliable the way a feature becomes finished. It becomes more reliable each time someone asks what happens when the message arrives twice, writes down the answer, and gives it an owner. No broker offers that, and no library ships it. It is a habit of attention, kept by

people who know that every system they build will one day be misused by a network, a deploy or a tired colleague, and who have decided to design for that day without drama. That is what I would ask you to take from this book: not a set of patterns, but the habit of asking the next question.

Appendix: Glossary and Pattern Quick Reference

This appendix has two parts. Part A is a glossary of the terms the book uses, in alphabetical order, with the chapter where each term is defined or used most fully. Part B is a quick reference for the patterns, written as a table you can scan during a design review.

Neither part replaces the chapters. A definition here is a reminder, and a table row is a pointer. The reasoning, the costs and the cases where a pattern should not be used live in the chapter named in the last column.

A. Glossary

acknowledgment (ack). The consumer's signal to the broker that a message has been handled and can be removed from the queue, or that the consumer's position can advance. The negative form is a **nack**. Chapter 2.

aggregate. The consistency boundary that owns a partition key. Events for one aggregate are ordered with respect to each other; events for different aggregates are not. Chapter 9.

aggregate version. A counter of the changes to one aggregate, stored in the outbox row and carried in the payload, such as `order_version` for Orders. It orders the events of one aggregate and says nothing about the shape of the payload. It is a different number from the **schema version** and the **registry revision**. Chapter 4 defines it; Chapter 9 uses it.

at-least-once. A delivery semantic in which a message is delivered one or more times, so duplicates are possible. It holds while the message is still retained and the consumer acknowledges only after the effect. A consumer that lags beyond retention still loses messages silently. It is the book's default. Chapter 2 defines it; Chapter 3 covers the loss.

at-most-once. A delivery semantic in which a message is delivered zero or one times, so a message can be lost but is never delivered twice. Chapter 2.

backfill. Bulk repair or rebuild that feeds a large set of events into a live consumer, run in small, rate-limited batches. Chapter 13.

backpressure. Bounding the work in flight at each stage so that load stays finite. In a broker-mediated system it appears as limits on a consumer's intake, such as prefetch or records per poll, and as lag that the broker absorbs. Pressure reaches a producer only through broker flow control or explicit throttling. Chapter 3 places it in the failure model; Chapter 6 gives the mechanisms.

backward compatibility. A consumer using the new schema can read events written with the old schema: new readers, old data. Chapter 12.

business key. An identifier derived from the meaning of the work, such as a payment id plus an operation, rather than from the envelope. It protects against the same work arriving in two different messages. Chapter 7.

causation_id. The envelope field that points to the event or command that directly caused this one, where `correlation_id` names the whole flow. Chapter 11.

change data capture (CDC). Reading a database's transaction log to turn committed changes into messages. In the outbox setting, a relay built on CDC reads outbox rows from the log instead of polling the table. Chapter 5.

circuit breaker. A guard around a dependency that stops calls after failures cross a threshold, then lets trial calls through after a wait. Chapter 6.

command. A request to do something, named in the imperative, such as `CapturePayment`. A command is addressed to one service and can be refused. It is not an event. Chapter 11.

compensation. The step that semantically undoes an earlier saga step, such as releasing stock or voiding an authorization. The book does not call it a rollback. Chapter 10.

competing consumers. Several consumers sharing one queue so that each delivery goes to one of them. A redelivered message can reach another consumer. It is the queue-family counterpart of a consumer group. Chapter 2.

consumer. A process that reads messages from a topic or queue and acts on them. Chapter 2.

consumer-driven contract. A file owned by a consumer that lists the fields and types it reads from an event, checked in the producer's pipeline. Chapter 15.

consumer group. A set of consumers that share the partitions of a topic, so that each partition is read by one member at a time. Chapter 2.

consumer lag. How far a consumer is behind the head of a partition or queue, measured in messages or in time. Chapter 2 defines it; Chapters 3 and 13 use it.

consumer outbox. An outbox written by a consumer that publishes a follow-up message, in the same transaction as its inbox row and its state change. A crash before commit leaves nothing behind, and a crash after it is absorbed by the inbox on redelivery. Chapter 7 introduces the rule; Chapter 10 applies it to saga steps.

correlation_id. The envelope field shared by every event in one business flow. Its companion `causation_id` points to the event or command that directly caused this one. Chapter 11.

cutover. The step of a broker migration in which a consumer group starts reading from the new broker, taken one group at a time after the two brokers have been reconciled and with a rollback planned. Chapter 14.

dead letter. A message that has been moved aside because processing it failed. As a modifier the term is hyphenated: dead-letter queue. Chapter 8.

dead-letter queue (DLQ). A separate destination where a consumer or broker places messages it has given up on. On a log-based broker the same idea is a **dead-letter topic**. Chapter 8.

deadline. The moment after which a result is no longer useful. Retries must stop before it. Chapter 6.

dedup window. The period during which a consumer remembers processed ids. A message whose inbox row has been deleted can be processed again. Chapter 7.

deferred. The retry class for a failure that says nothing about the message, because the dependency is known to be down, as when a circuit breaker is open. A deferred message consumes no attempt and is never dead-lettered for it. The consumer pauses and the message waits. Chapter 6.

dual write. Updating two systems, such as a database and a broker, without a shared transaction. Chapter 4.

dual-publish. Publishing the same fact in two forms or to two destinations during a migration: both fields or both topics during a schema change (Chapter 12), both brokers during a broker migration (Chapter 14). In a broker migration the relay keeps a second marker for the new target, so that the old broker stays complete and the new one can be reconciled against it.

duplicate. A message processed or delivered more than once. Chapter 2.

envelope. The standard set of fields that travels with every event, such as `event_id`, `event_type`, `schema_version`, `occurred_at`, `correlation_id` and `causation_id`, separate from the payload. Chapter 11.

event. A message whose content is a fact that has already happened, named in the past tense. Chapter 11.

event_id. The envelope field that identifies one event. It is created when the outbox row is written and never regenerated, and it is the usual idempotency key for consumption. Chapters 4 and 7.

event journal. An append-only record of event envelopes, without payloads, kept by a service next to its inbox so that questions can be answered after the broker has forgotten the events. Chapter 13.

event log. The durable, append-only sequence of events. Chapter 2.

eventual consistency. The property that replicas or services converge on the same state if updates stop, without a bound on how long it takes. Chapter 1.

exactly-once. Used in the book only as a qualified effect: at-least-once delivery plus an idempotent consumer, which the book also calls effectively-once. It is not a delivery guarantee. A broker's exactly-once processing is real inside that broker's boundary and stops at it. Chapter 2.

expand and contract. A migration pattern that turns one breaking change into several compatible steps: support both shapes, migrate every reader, then remove the old shape. Chapter 12.

failover. A designed switch to another copy of a broker partition, a database or a cluster when the active one fails. It can lose what had not yet been replicated. Chapters 3 and 14.

failure model. The written list of a system's seams and, for each, what can fail, how it would be observed and which guarantee survives it, with owners, tests and review dates. Chapter 3.

failure semantics. For a given seam, what can fail, how you would observe it and which guarantee survives it. It is the book's thesis. The Preface states it, Chapter 3 applies it as the failure model, and Chapter 16 turns it into the review checklist.

fat event. An event that carries enough state for consumers to act without calling back to the producer. The opposite is a **thin event**. Chapter 11.

forward compatibility. A consumer using the old schema can read events written with the new schema: old readers, new data. Chapter 12.

grace period. The age a record must reach before a reconciliation job compares it, so that events still in flight are not reported as discrepancies. Chapter 13.

head-of-line blocking. A record that a consumer will not skip blocks every record behind it in the same partition or stream. Chapter 8.

hold-back buffer. A durable table in which a consumer holds an event whose predecessor has not arrived, and releases it when the gap closes or a timeout expires. Chapter 9.

idempotency key. The key a consumer or API uses to recognize repeated work. For event consumption it is normally the `event_id`. Chapter 7.

idempotent consumer. A consumer whose repeated processing of the same message has the same effect as processing it once. Chapter 7.

inbox. The table of processed message ids on the consumer side, written in the same transaction as the handler's effects. Chapter 7.

integration event. An event published as a deliberate public contract, mapped from a domain event and free of personal data and secrets. Chapter 11.

invariant check. A scheduled query for a rule the business holds true, such as every captured payment having exactly one Ledger entry. Each returned row is a discrepancy. Chapter 13.

jitter. Randomness added to a retry delay so that many clients do not retry in step. "Exponential backoff with jitter" is the standard phrase. Chapter 6.

link (trace). A reference from a consumer span to the producer's trace context, used instead of a parent-child relationship when delivery is late, repeated or batched. Chapter 13.

message. A unit of data sent through a broker: an event, a command or a document. Chapter 11.

message age. The time since the oldest message still waiting was produced, compared with a business deadline and with the retention period. Chapter 13.

message broker. The infrastructure that carries messages between services. Two families appear in the book: log-based and queue-based. Chapter 2.

mirroring. Copying messages between brokers with a tool: a mirroring tool between two logs, or a bridge or connector between a log and a queue. It also carries history from before a migration and can keep a standby cluster fed, and the offsets on the copy are its own. Chapter 14.

offset. A consumer's position in a partition of an event log. Chapter 2.

offset commit. The consumer's record, on a log-based broker, that it has processed everything before a position. It replaces the per-message acknowledgment. Chapter 2.

offset reset. Moving a consumer group's committed position, usually backward to a timestamp with a margin, so that history is read again. Offsets belong to one cluster and do not translate to another. Chapter 14.

outbox. The table written in the same local transaction as the state change, from which a relay publishes events. The full term is **transactional outbox**. Chapter 4.

parking lot. A holding place for dead letters that need a human decision, distinct from the DLQ itself. Chapter 8.

partition. A unit of a log-based topic that is ordered and read by one consumer of a group at a time. Chapters 2 and 9.

partition key. The aggregate identifier used to route events to a partition and so to preserve per-aggregate order. Chapter 9.

pivot step. The point in a saga after which the workflow no longer compensates and only moves forward. Chapter 10.

poison message. A message that fails every time it is processed. On a log-based broker one that crashes the process leaves the consumer stuck on one position, so it is detected by lag with no throughput and by a counter per position. Chapter 8.

producer. A process or service that publishes messages, and the owner of the events it emits. Chapter 11.

queue. A broker structure in which each message is consumed and acknowledged individually. Chapter 2.

rebalance. The reassignment of partitions when members join or leave a consumer group, or when a member is judged unresponsive. Two consumers can briefly work on the same partition. Chapter 3.

reconciliation. Comparing two systems' views of truth and repairing the differences. Chapter 13.

recovery point objective (RPO). How much data you can afford to lose, measured as time. A backup every six hours means an RPO of up to six hours, and each store has its own. Chapter 14.

recovery time objective (RTO). How long you can be without the service. For a replayed projection it is a drain calculation. Chapter 14.

redelivery. The broker delivering a message again, for example after a missed ack or an expired visibility timeout. Replay is a different thing. Chapter 2.

redrive. Giving dead letters back to the consumer that failed on them, to its own input and never to the shared topic they came from. The key and `event_id` are kept, the attempt count starts again, and `redrive_count` records the round. Chapter 8.

registry revision. The identifier a schema registry assigns to each registered schema. It moves on every registered change, additive ones included, and is a different number from the **schema version** and the **aggregate version**. Chapter 12.

relay. The process that moves outbox rows to the broker. Chapter 5.

replay. Deliberately re-reading history from an event log. Chapter 2 defines it; Chapters 7, 12, 13, 14 and 15 use it.

retention. How long a broker keeps a message whether or not it has been read: a time or size limit on a log, an expiry on a queue. It bounds replay, redrive and how long a lagging consumer can be down. Chapter 2 defines it; Chapters 3, 13 and 14 use it.

retry budget. A limit on retries as a share of overall traffic rather than per call. Chapter 6.

retry storm. A surge of retries that adds load to a struggling dependency and keeps it down. Chapter 6.

saga. A long-running workflow made of local transactions with compensations, either orchestrated by a coordinator or choreographed through events. Chapter 10.

schema evolution. Changing the shape of events over time without breaking the readers and writers that exist. Chapter 12.

schema registry. A service that stores schemas, assigns identifiers and checks a proposed schema against earlier ones. Chapter 12.

schema version. The integer `schema_version` in the envelope, which Chapter 12 also calls the breaking version. It moves only for a change that an old reader cannot handle, and it is a different number from the **reg-**

istry revision and the **aggregate version**. Chapter 12.

semantic lock. A status flag in the data that says a saga is working on a record, which other code is written to respect. It is not a database lock. Chapter 10.

synthetic event. Labeled traffic injected into the live pipeline by a probe and checked at the end of the flow. Chapter 15.

tolerant reader. A consumer that takes only the fields it needs from an event and ignores the rest. Chapter 12.

tombstone. The key, the last version and a deleted marker that a consumer keeps after deleting an aggregate's row, so that a late event cannot recreate it. Chapter 9.

topic. A named stream of messages in either broker family. Chapter 2.

trace context. The trace id and span id that place a unit of work in a distributed trace, captured into the outbox row and sent in message headers so that the consumer's span can link to the producer's. Chapter 13.

upcaster. Code that converts an event written under an old schema into the current one before business logic sees it. Chapter 12.

version check. A consumer comparing the version in an event with the last version it applied for that aggregate, to ignore stale events and detect gaps. Chapter 9.

visibility timeout. On a queue-based broker, the time a delivered message stays hidden before it is redelivered if it has not been acknowledged. Chapter 2.

B. Pattern quick reference

Each row names the failure the pattern addresses, in the book's terms, and what it costs you. A pattern moves a failure to a place where you can handle it; it does not remove the failure. Chapter 16 turns these rows into a review checklist.

Pattern	Failure addressed	Costs	Chapter
Transactional outbox	Dual write between a database and a broker	Extra table, a relay to run, duplicates pushed to consumers	4
Polling relay	Moving outbox rows to the broker with little infrastructure	Polling delay, load on the database, careful cursor design	5
Change data capture	Publishing outbox rows from the database log with low delay	Operational weight, connector failover, log retention to manage	5
Exponential backoff with jitter	Retries that overload a recovering dependency	Added latency, tuning of delays and attempt limits	6
Retry budget	Retries that amplify an outage	Some calls fail fast that might have succeeded	6
Backpressure and flow control	A dependency or consumer fed more work than it can absorb	Lag, paused consumers and rejected work, a business decision on what waits	6
Circuit breaker	Repeated calls to a failing dependency	State to tune; calls rejected while the breaker is open	6
Retry topic or delay queue	Retry delays too long to wait out inside the handler	More topics, more moving parts, harder ordering	6
Idempotent consumer with inbox	Duplicates from at-least-once delivery	Table writes on every message, retention to manage	7
Business key	The same work arriving in different messages	Key design per operation, and unique constraints	7
Idempotency key passed downstream	Duplicate external side effects	Needs support from the downstream service	7
Dead-letter queue	Poison messages stuck in a redelivery loop	A pile that needs an owner, alarms and a repair path	8
Parking lot	Dead letters that need a human decision	Manual work and a clear ownership rule	8
Redrive	Recovering dead letters after the cause is fixed	Duplicates, reordering, and load on the main flow	8
Consumer outbox	A consumer that changes state and publishes a follow-up, losing one of the two	An outbox on the consumer side and a relay to run	7, 10

Pattern	Failure addressed	Costs	Chapter
Partition key per aggregate	Out-of-order events with- in one aggregate	Hot partitions, limits on parallelism	9
Version check on the consumer	Stale or out-of-order events overwriting newer state	A version field and a deci- sion for each gap	9
Hold-back buffer	Gaps in a sequence that must be applied in order	Memory, timeouts and the gap that never closes	9
Saga (orchestrated or choreographed)	A business workflow that spans services without one transaction	States, timeouts and com- pensations to design and operate	10
Compensation	Undoing completed saga steps after a later failure	Some steps cannot be un- done; the pivot step marks the limit	10
Semantic lock	Concurrent changes to a record that a saga is working on	Blocked operations while the saga runs	10
Wire types fixed per field across languages	The same envelope field read differently by two runtimes	Discipline in every lan- guage and tests that cross languages	11
Standard event envelope	Events that tooling and consumers cannot trace or route	Discipline across every producer	11
Thin or fat event	Choosing between call- backs and carried state	Thin: coupling at read time. Fat: larger contract and copied data	11
Schema registry with compatibility checks	Producers silently break- ing consumers	A service to run and a process to follow	12
Tolerant reader and upcaster	Readers breaking on new fields or old versions	Conversion code kept for every historical version	12
Expand and contract	A breaking change that cannot be avoided	Several deploys, dual-pub- lish, a longer migration	12
Trace context and correlation id	Losing the story of a flow across asynchronous hops	Propagation in every pro- ducer and consumer	13
Reconciliation and repair	Silent divergence between a source system and de- rived state	Comparison jobs, repair tooling and review of results	13

Pattern	Failure addressed	Costs	Chapter
Event journal	Questions about events that the broker has forgotten	A table written on every message and retention to manage	13
Backfill	Derived state that must be rebuilt or filled in from existing data	Rate-limited batches and a pause switch, and the load on a live consumer	13
Invariant checks	Business-level breakage that no technical metric shows	Defining the invariants and owning the alerts	13
Offset-reset replay	A consumer that was wrong, or a store that went back in time, and needs history again	A dedup window that covers the range, suppressed side effects, and retention as the limit	14
Recovery point and time objectives per store	Disasters handled by improvisation, with no agreed loss or downtime	A number to measure and rehearse for every store	14
Cross-cluster replication for recovery	Loss of a broker cluster or a region	A standby to run, a replication lag that is the RPO, and offsets that do not translate	14
Dual-publish and cutover	Moving to another broker without losing events or duplicating effects	A second marker in the outbox, reconciliation, and a slower migration	14
Consumer-driven contract tests	Contract drift between teams	Test upkeep and agreement on who runs them	15
Handler and perturbation tests	Handlers never exercised under duplicates, delay or reordering	Real-database test infrastructure and slower suites	15
Fault injection	Failure-model entries never exercised in a running system	A dedicated environment, a blast radius to bound	15
Replay-based regression test	Changes that break behavior on real history	Curated, anonymized sample data to maintain	15
Synthetic events in production	Silent failures in the live path	Test data that must be isolated from real business data	15

Reading the table

Many rows come in pairs: a pattern that fixes one failure and a second pattern that fixes the failure the first one introduced. The outbox creates duplicates, so the consumer needs an inbox. Retries create load, so they need a budget. A dead-letter queue creates a pile, so it needs an owner and a redrive procedure.

That chain is the shape of the book. If you adopt one row and skip its partner, the failure has moved but not gone. In my view the table is most useful read from right to left: start with the failure your system has today, find the row that names it, and read the cost column before you read the chapter.

ABOUT THE BOOK

Reliable event-driven systems are built around failure semantics, not message brokers.

For every seam, where one component hands work to another, the book asks the same three questions: what can fail, how you would observe it, and which guarantee survives it. At-least-once delivery is treated as a fact rather than a defect, so handlers are idempotent, and duplicates, lost messages, retries and out-of-order arrival are designed for instead of hoped away. Following Parcelly, the book works through the transactional outbox, relays and change data capture, retries, idempotency, dead letters, ordering, sagas, schema evolution, observability and recovery, each with the cost it carries.

ABOUT THE AUTHOR

Muhammet Şafak has been building software since 2008. Over many years Muhammet has held active roles in the healthcare, finance and telecommunications domains, and argues that development should focus on the problem, not the technology. Since 2024 Muhammet has been designing and building high-traffic systems that scale end to end.

Muhammet mentors small and mid-sized software development teams, leads open-source projects, and writes decisions down with their reasons and their cost.

Reliable systems are not built from reliable parts. They are built around the parts that fail.

I
ACCESS

II
EVENTS

III
AGENTS

