READ MODE · ANSWER FIRST

Idempotency and exactly-once semantics explained

Learn at-most-once, at-least-once, idempotency keys, outbox and inbox patterns, Kafka transactions, and exactly-once boundaries.

CHEAT SHEET · 13 ESSENTIAL IDEAS

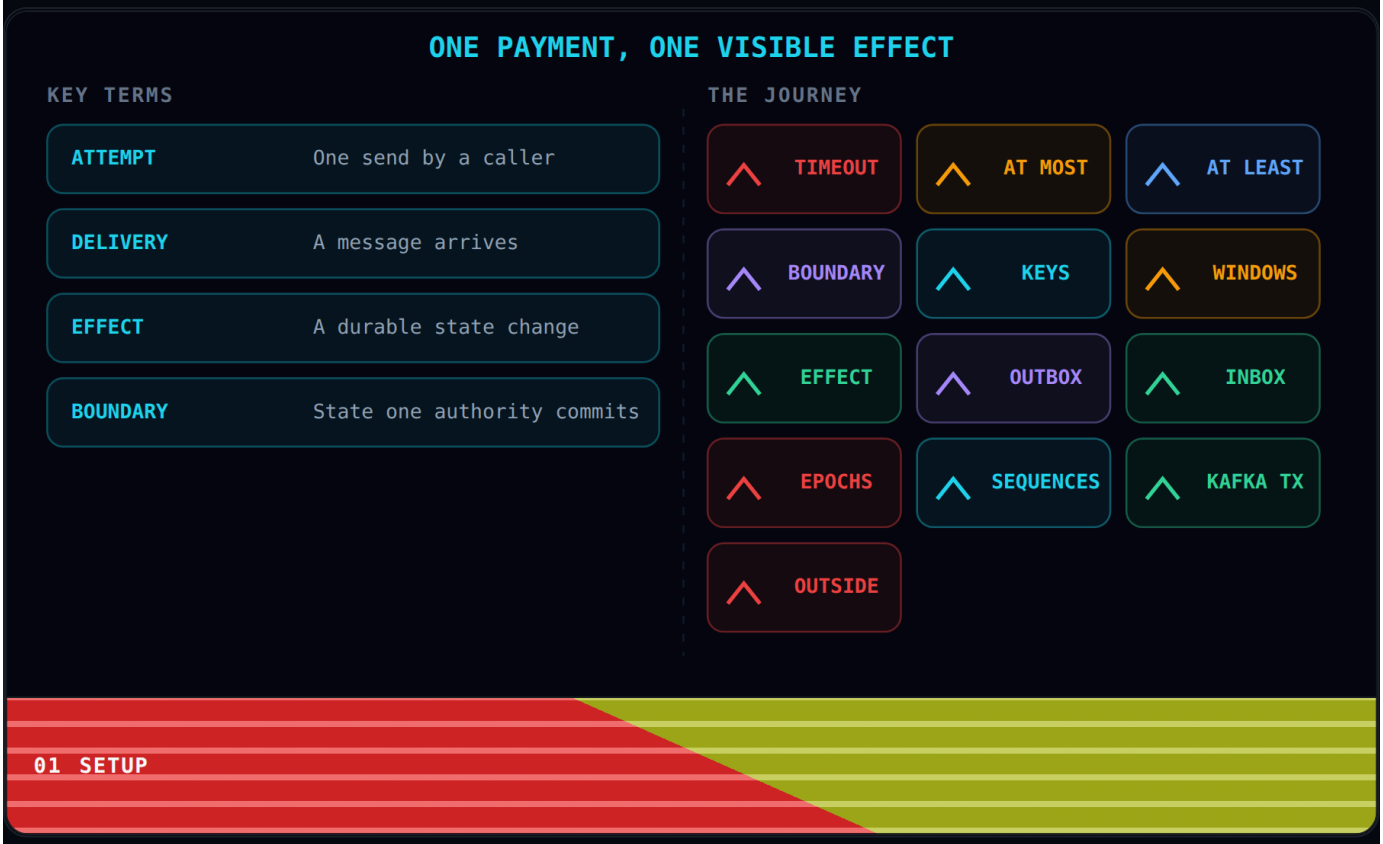
The whole story in 13 lines

The practical goal is not one delivery but one durable effect across every boundary that the request crosses.

1. A lost reply leaves the client unable to tell whether the server committed the effect.
2. At-most-once avoids duplicate delivery by refusing to retry, so it can lose work.
3. At-least-once retries until delivery succeeds, so consumers must expect duplicates.
4. Exactly-once always names the state and effects controlled by one commit boundary.
5. An idempotency key maps repeated identical requests back to the first stored result.
6. A deduplication window forgets old keys, so a late retry can become new work again.
7. Effectively-once combines repeated delivery with durable guards that expose one effect.
8. A transactional outbox commits the business row and event row in one database transaction.
9. An inbox records an event ID beside the consumer effect so redelivery becomes a no-op.
10. A producer epoch fences an older process that wakes after a replacement has taken over.
11. Per-partition sequence numbers let a broker reject duplicates and detect missing batches.
12. Kafka transactions atomically publish Kafka records and consumed offsets to read-committed consumers.
13. External APIs remain outside Kafka transactions unless they provide their own idempotent boundary.

STAGE 1 OF 15

Setup



Imagine paying once while the network can lose any reply. We will follow one command and ask what the client, service, broker, and bank can actually prove.

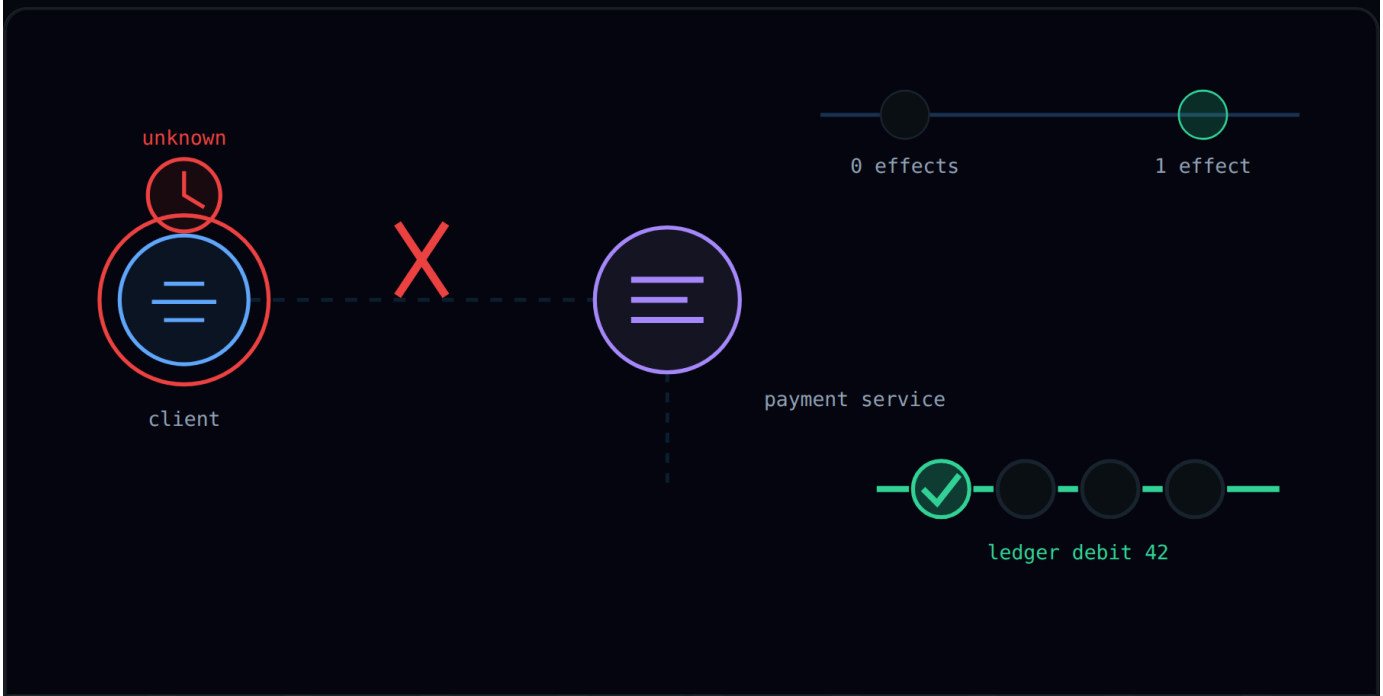
Before we begin, let us separate three ideas that sound similar but describe different moments in a request. An attempt is one send, a delivery reaches a receiver, and an effect changes durable state such as a balance or receipt.

Our fourth term is the boundary, which names the state one authority can commit together. Later safeguards will protect repeated work whenever this payment crosses into another authority.

Think of pay_8472 as one ticket that never changes its name, even when the client sends it again. We will follow it through retries, database rows, producer metadata, and Kafka logs while asking who remembers that identity.

Now let us begin with the uncomfortable moment that creates every retry: the server may finish while its reply disappears. Once you understand why the client cannot distinguish that outcome from failure, the need for every later safeguard becomes much easier to see.

The Ambiguous Timeout



02 THE AMBIGUOUS TIMEOUT

Let us begin by watching payment pay_8472 move from the client toward the payment service. The service may change the ledger before the network delivers any answer back to the client.

The service writes one debit and creates a receipt before sending a success reply across the same unreliable network. At this point the durable effect already exists, so losing the reply cannot undo the ledger change that the client has not yet learned about.

The client waits until its timer expires. Did the request vanish before the service or did only the success reply vanish afterward?

PAUSE AND PREDICT

Which lost network leg can leave a committed debit behind?

- The request
- The reply

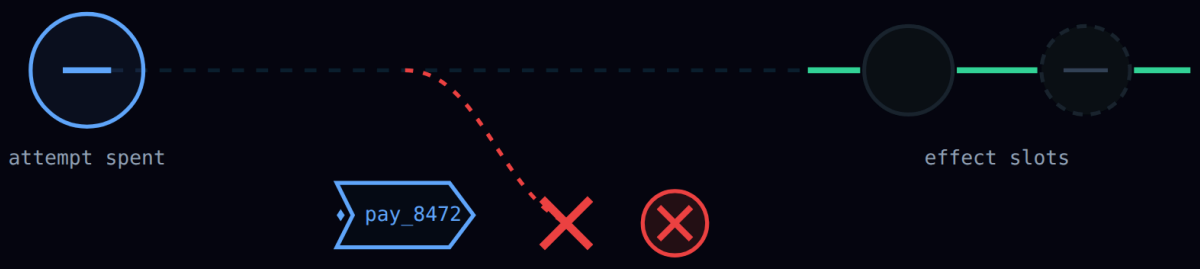
Skip this check

The reply-loss path leaves a durable ledger mark beside a client that still sees only a timeout. The request-loss path leaves no debit, which exposes why the same timeout can describe two opposite outcomes.

Switch the Failure point control through both network legs. Compare the ledger result after Request lost with the committed but unknown result after Reply lost.

A timeout means the outcome is unknown, not necessarily that the operation failed. The next stage tests the simplest response to that uncertainty by refusing to send the command twice.

At-Most-Once



03 AT-MOST-ONCE

Now that we understand why a timeout is uncertain, consider the simplest policy: send the payment ticket once and never retry it. This policy protects the receiver from duplicate deliveries because the sender creates only one possible copy.

If that ticket arrives, the service performs one effect and no second copy waits behind it. At-most-once therefore gives us a clean success path, but only when the network happens to deliver the single attempt.

The empty retry slot guarantees that no duplicate can follow the first attempt, but it also removes the sender's recovery option. What happens when the only ticket disappears before reaching the service?

PAUSE AND PREDICT

What remains after the only at-most-once attempt is dropped?

- No payment effect
- A retry copy

Skip this check

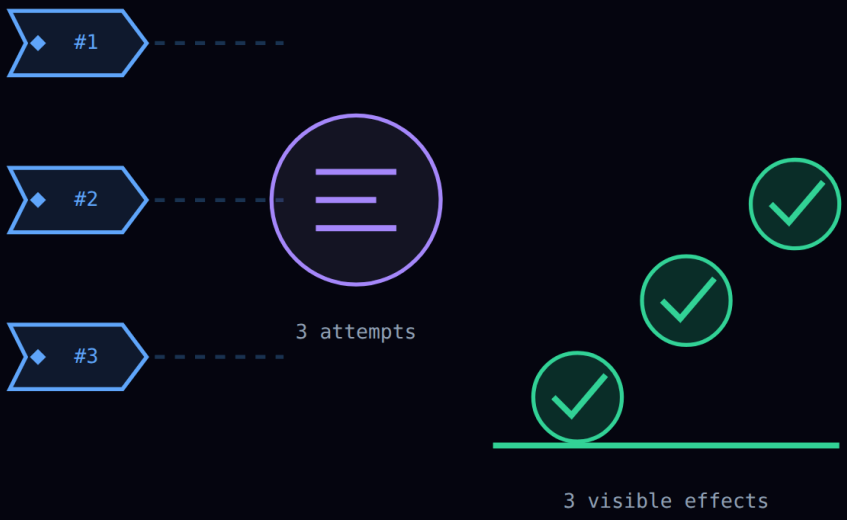
A dropped one-use ticket falls away before the service and both effect slots remain empty. The policy prevents duplicate delivery precisely because it accepts that this payment can disappear permanently.

Switch the Delivery control between Delivered and Dropped. Compare the completed effect with the permanently empty slot caused by the lost one-use ticket.

Avoiding retries removed duplicate delivery, but it also removed any chance to recover a lost request. Next we will make the opposite trade by retrying until something arrives, then study where the duplicate problem moves.

STAGE 4 OF 15

At-Least-Once



04 AT-LEAST-ONCE

We just saw that refusing to retry can lose the payment even though it prevents duplicate delivery. At-least-once makes the opposite trade by retrying until one delivery returns a usable answer.

Attempt one commits the payment but loses its reply, so the client sends attempt two with the same command. To the network these are separate deliveries, even though the person intended only one payment.

The service receives two ordinary requests with no built-in history connecting them to one intention. Without durable memory, how could it know that the second ticket describes old work rather than a new payment?

PAUSE AND PREDICT

What can an unguarded receiver infer from the second delivery?

- **Nothing about intent**
- **It must be a retry**

Skip this check

Three delivered copies now terminate in three separate effect marks, even though the sender meant one payment. At-least-once recovers delivery while leaving the receiver responsible for duplicate suppression.

Adjust the Attempts control at least once. Notice how every additional unguarded delivery adds another durable effect instead of strengthening a one-effect promise.

At-least-once improved the chance of delivery but moved duplicate handling into the receiver. Before adding a guard, next we must draw the exact state boundary that any stronger promise can honestly control.

Exactly-Once Needs a Boundary

05 EXACTLY-ONCE NEEDS A BOUNDARY

We have reached the central question of the lesson. Repeated delivery can create repeated effects, so an exactly-once promise must first name the state one commit authority actually controls.

Inside one database or Kafka transaction, several writes can become visible together because one coordinator decides whether all of them commit or abort. Atomicity works here because every participating state change accepts that coordinator's decision.

The bank owns different storage, exposes a separate protocol, and does not automatically participate in Kafka's transaction. Can Kafka's commit decision include that external charge merely because our application called both systems?

PAUSE AND PREDICT

Can Kafka atomically commit its records together with this bank charge?

- **Yes, one application called both**
- **No, the bank is separate**

Skip this check

The Kafka bracket closes around broker-managed records and offsets, but it breaks when stretched across the separate bank authority. Exactly-once remains precise only when its scope stops at that ownership line.

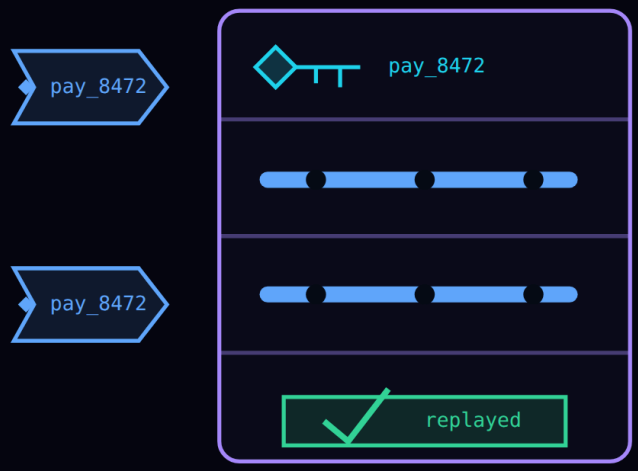
Switch the Boundary control through both scopes. Compare the closed Kafka commit with the broken bracket produced by the unsupported Kafka + bank claim.

Once the transaction scope is honest, operations outside it need their own compatible protection instead of borrowed guarantees. We will start with an idempotency key, which gives repeated requests a stable identity the external service can remember.

STAGE 6 OF 15

Idempotency Keys

Try it in Watch mode. Make a reused key fail safely instead of replaying the first receipt.



06 IDEMPOTENCY KEYS

Now that the boundary is honest, let us protect the operation that sits outside its atomic guarantee. The client attaches key pay_8472 to every retry of the same intended payment.

The first request stores a fingerprint of the payload and the resulting receipt beside key pay_8472. Saving both matters because the service must later distinguish a true retry from a different payment that accidentally reused the same key.

A retry arrives with the same key. Should the service execute it again or compare its body with the stored operation?

PAUSE AND PREDICT

What should happen when the same key returns with an unchanged body?

- **Execute again**
- **Replay the stored result**

Skip this check

The retry reaches the same stored key and payload fingerprint, so the service returns the first receipt without another debit. A changed fingerprint instead stops at a conflict gate.

Switch the Payload control through Same body and Changed body. Compare a receipt replay with the conflict that prevents one key from naming different payment details.

The key turns a retry into a lookup instead of another execution, but storing every completed identity forever would grow without limit. Next we will place that memory on a timeline and see exactly when a late retry becomes dangerous again.

Deduplication Windows

07 DEDUPLICATION WINDOWS

An idempotency key works only while the service remembers it, so let us place that memory on a five-minute ruler. The highlighted window is the exact period during which the service promises to recognize this payment as old work.

Attempt two arrives after one minute and lands inside the remembered region, so the service finds pay_8472 and replays the first receipt. The retry is harmless because both the operation identity and its stored result still exist.

Attempt three waits much longer, moving toward the edge where old keys are pruned to bound storage. When it finally arrives, is pay_8472 still a known duplicate or has the service already forgotten it?

PAUSE AND PREDICT

What happens to a retry arriving after the five-minute window?

- It is still replayed
- It may become new work

Skip this check

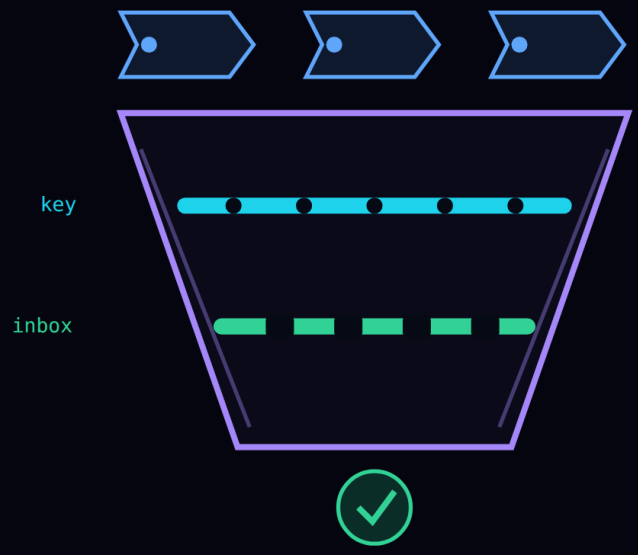
At seven minutes, the retry marker sits beyond the five-minute memory edge and the service treats the key as new. At three minutes, the stored receipt still absorbs it.

Switch the Retry age control between 3 min and 7 min. Compare the remembered duplicate with the expired key that can cross into new work.

Retry safety lasts only as long as the deduplication memory, which means clients and servers must agree on a retry horizon. Next we will combine several bounded defenses so duplicate deliveries can still produce one practical end effect.

STAGE 8 OF 15

Effectively-Once



08 EFFECTIVELY-ONCE

The deduplication window showed us that no guard lasts forever, even when its stored identity is durable. Effectively-once therefore starts with an honest premise: duplicate deliveries still enter the system and must be handled.

An API idempotency key protects command creation, while a consumer inbox records event identities before applying downstream projection updates. These guards sit at different boundaries because the command retry and the event redelivery are different duplicate paths.

Three copies can cross the delivery side even when the user intended one payment, so every effect boundary needs memory of the same operation. If one layer forgets that identity, how many debits can the three deliveries produce?

PAUSE AND PREDICT

Which protection path reduces three deliveries to one end effect?

- **No defenses**
- **Idempotency key only**
- **Key plus inbox**

Skip this check

The complete pair of guards narrows three delivered copies into one durable end effect while keeping every input copy honest. Removing either mesh lets duplicate work escape at an unprotected boundary.

Switch the Defenses control through None, Key only, and Key + inbox. Compare how the output count changes when each boundary gains or loses durable memory.

Effectively-once is a composition of bounded safeguards rather than proof that duplicate messages never existed. Next we will make the producer's first local step atomic by writing business state and an outgoing event together with a transactional outbox.

STAGE 9 OF 15

Transactional Outbox

09 TRANSACTIONAL OUTBOX

Let us build effectively-once one local, atomic step at a time before connecting the full path. The payment service first writes its business row and event row inside one database transaction.

The database commit makes the payment row and outbox event visible together. This removes the unsafe gap where the business change succeeds but the process crashes before recording anything that can later publish its event.

A separate relay reads the durable outbox row and publishes it to the broker, then records that it made progress. What happens if the relay crashes after publishing the event but before saving that progress marker?

PAUSE AND PREDICT

What can the restarted relay safely do with the unmarked outbox row?

- Skip it forever
- Publish it again

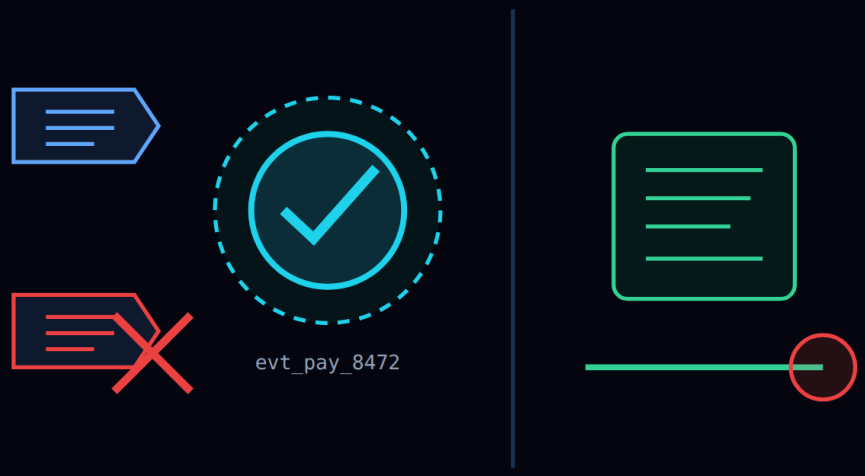
Skip this check

The restarted relay publishes the durable row again, leaving two broker copies beside one atomic database change. The outbox closes the loss gap but deliberately preserves at-least-once delivery.

Toggle the Relay crash control through both states. Compare one ordinary publish with the duplicate broker copy created after a crash interrupts progress tracking.

The outbox prevents event loss by making publication safely repeatable, but that choice intentionally permits more than one broker copy. Next the inbox will complete the pattern on the consumer side by absorbing that expected redelivery.

Inbox Pattern



10 INBOX PATTERN

The outbox deliberately allows a repeated publish, so the receiver now needs its own durable memory of completed work. The receipt service checks event ID evt_pay_8472 before changing its projection.

On the first delivery, the consumer inserts the inbox event ID and the receipt projection row in one local transaction. Either both records commit or neither does, so the memory of processing cannot become separated from the effect it protects.

The same event arrives again after a crash. Can it insert a second receipt while the inbox ID already exists?

PAUSE AND PREDICT

What should the consumer do when the inbox already contains this event ID?

- **Insert another receipt**
- **Treat it as a no-op**

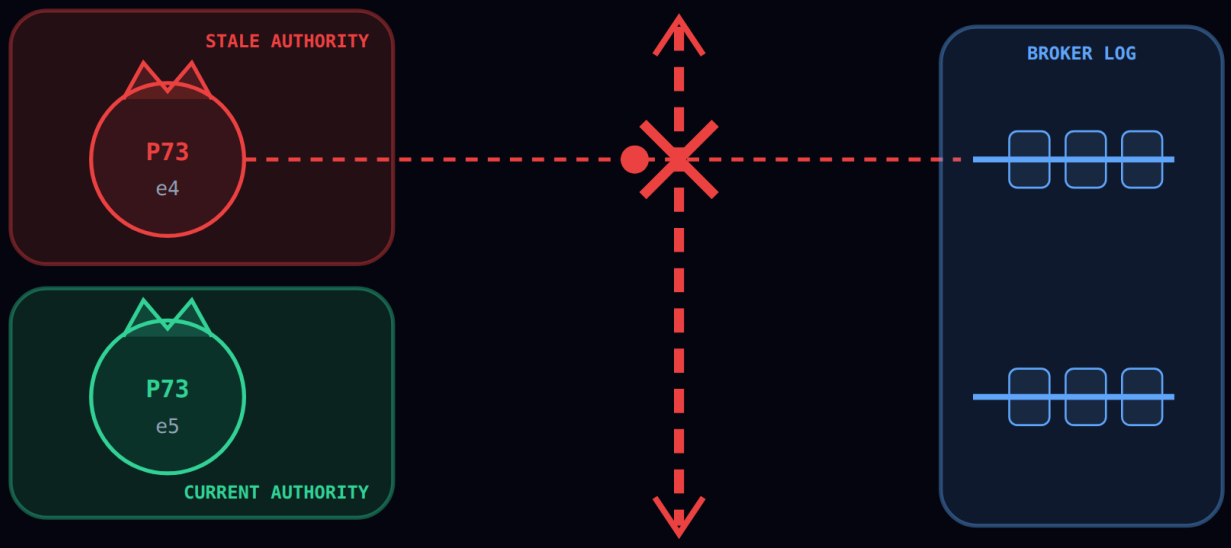
Skip this check

The repeated event collides with its durable inbox stamp and cannot create another projection row. The first delivery creates both records, while every later copy preserves the single receipt.

Switch the Delivery control between First and Redelivery. Compare the atomic row creation with the duplicate path that stops at the existing inbox stamp.

The outbox and inbox now make producer and consumer retries safe across their local database boundaries. Next we move inside Kafka, where epochs and sequence numbers handle duplicates before application guards receive records.

Producer Epochs



11 PRODUCER EPOCHS

We handled duplicate events at the consumer, but Kafka faces a different problem: an old producer process can wake after its replacement has taken over. Both processes may know the same transactional identity, so Kafka needs a way to decide which session still has authority.

The coordinator gives replacement producer 73 a newer epoch, e5, while the paused process still carries e4. The shared producer ID says they belong to the same logical producer, and the epoch says which incarnation is current.

Both processes know the same transactional identity. Which one should the broker trust when they race to append?

PAUSE AND PREDICT

Which producer incarnation should cross the broker fence?

- Zombie epoch e4
- Current epoch e5

Skip this check

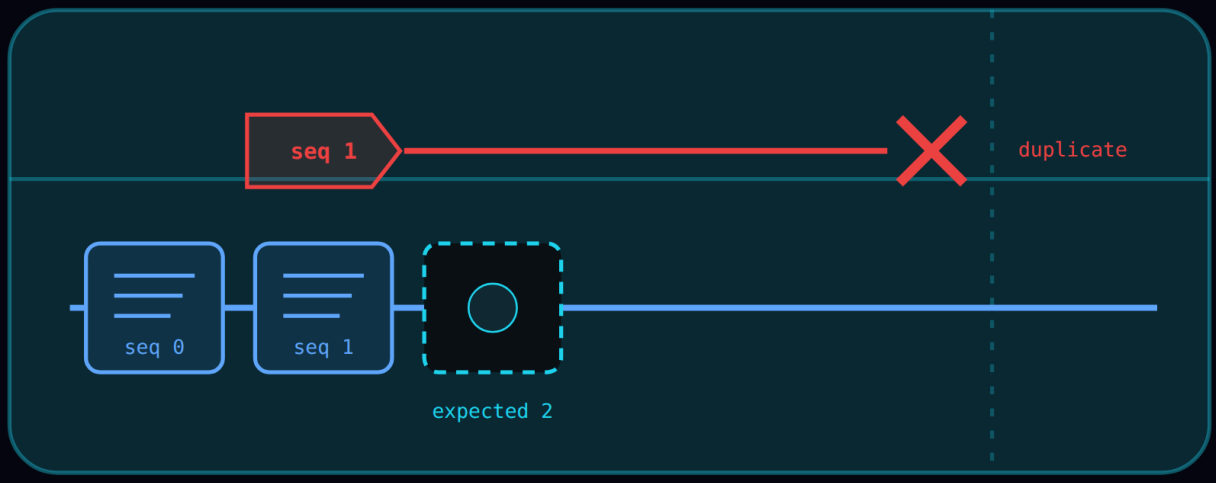
Current epoch e5 crosses the authority fence and reaches the broker log, while zombie epoch e4 stops before any append. The epoch converts process replacement into a strict authority check.

Switch the Instance control between Current e5 and Zombie e4. Compare the accepted append with the obsolete producer that collides with the epoch fence.

Epochs fence an obsolete producer process, but the legitimate live session can still retry a batch after losing a response. Next Kafka will number each partition's writes so the broker can distinguish a repeated batch from the next expected batch.

STAGE 12 OF 15

Sequence Numbers



12 SEQUENCE NUMBERS

The epoch tells Kafka which producer session is legitimate when old and new processes both claim authority. Within that session, the producer labels every batch for a partition with a growing sequence number.

The broker has accepted sequences 0 and 1 for this producer and partition, so the ordinary next slot expects sequence 2. That small counter turns an ambiguous network retry into a concrete ordering question the broker can answer locally.

A response is lost and a batch returns. Is it the expected next batch, a duplicate retry, or evidence that a batch went missing?

PAUSE AND PREDICT

How should the broker classify sequence 1 after it already accepted sequence 1?

- The next batch
- A duplicate batch
- A missing-batch gap

Skip this check

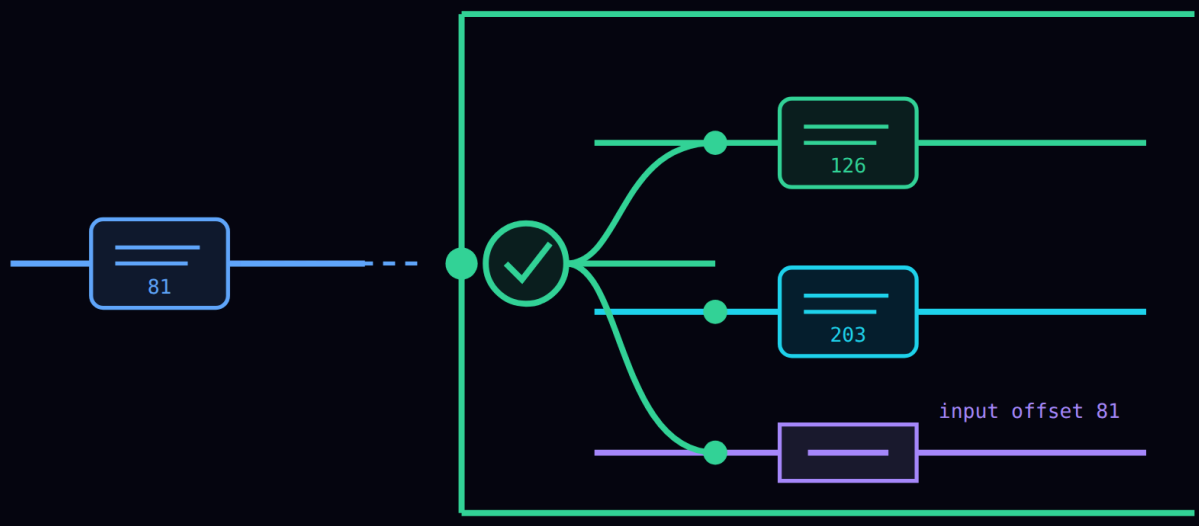
Repeat 1 lands on an occupied slot and is rejected, while Next 2 fits the expected opening exactly. Gap 4 exposes missing positions instead of allowing the log order to jump silently.

Switch the Sequence control through Next 2, Repeat 1, and Gap 4. Compare the accepted slot with duplicate rejection and missing-batch detection.

The epoch selects the legitimate producer session, while sequence numbers order that session's batches within each partition. Next Kafka will group records across several logs together with one consumed offset, moving from duplicate append protection to transactional visibility.

STAGE 13 OF 15

Kafka Transactions



13 KAFKA TRANSACTIONS

Sequence numbers protect individual appends, so let us widen the view beyond one partition and one batch. A Kafka transaction groups the output records from input offset 81 with that consumed offset.

The processor writes a receipt at offset 126 and an audit record at offset 203 before asking the transaction coordinator to finish the bundle. These records may already occupy physical log positions, but read-committed consumers still wait for the transaction outcome before treating them as visible.

A read-committed consumer waits for the transaction marker. Should these records and input progress appear if the processor aborts?

PAUSE AND PREDICT

What does a read-committed consumer receive after this transaction aborts?

- Records and offset
- Neither records nor offset

Skip this check

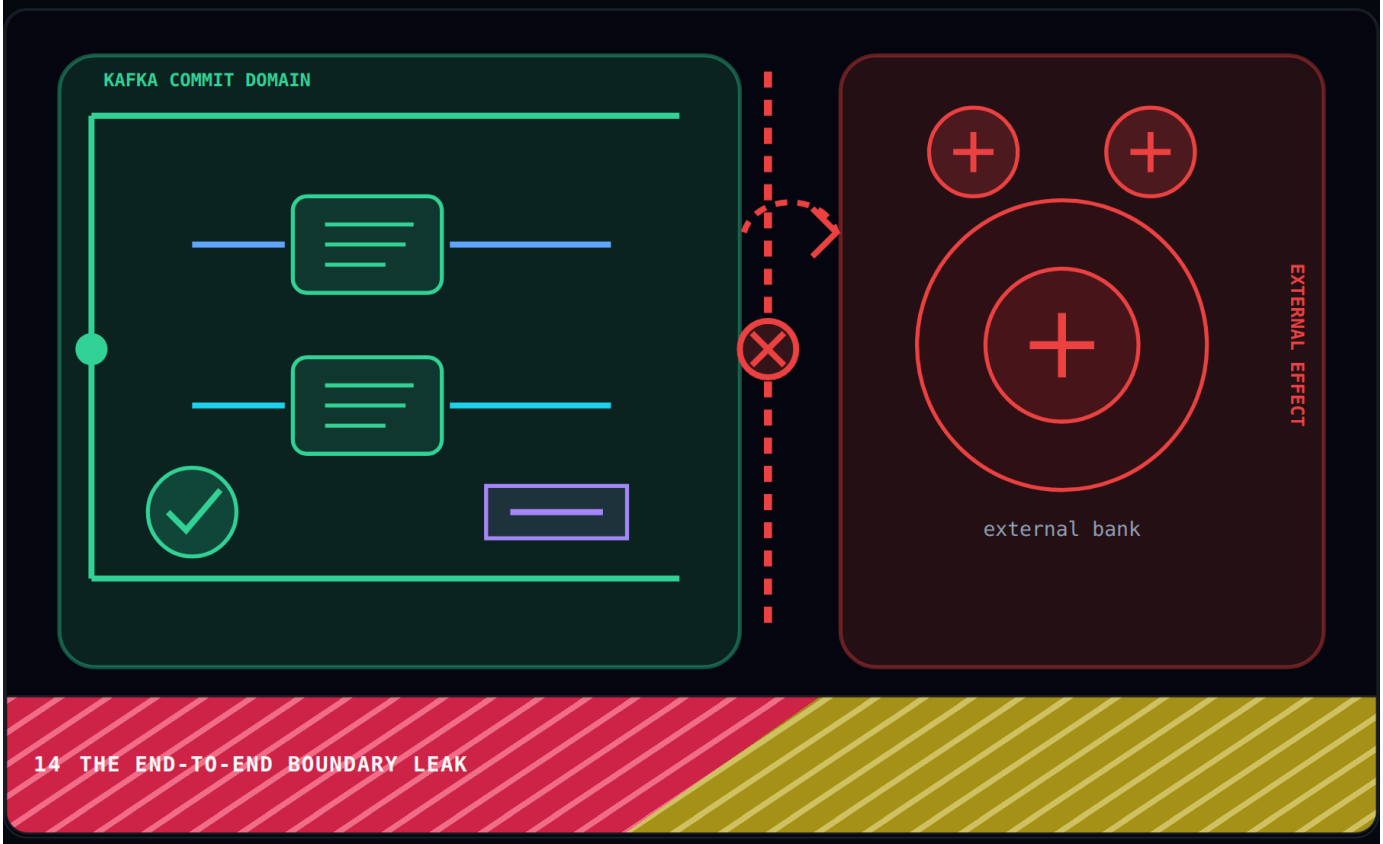
The commit marker reveals two output records and offset 81 as one bundle, while an abort crosses out every member. Kafka therefore couples result visibility with input progress.

Switch the Outcome control between Commit and Abort. Compare the complete readable bundle with the state where both output records and input progress remain hidden.

Kafka can make its output records and consumed offset appear together to read-committed consumers, which is a real exactly-once guarantee for Kafka-managed state. The final stage will place an external bank charge beside that boundary and test what the guarantee does not include.

The End-to-End Boundary Leak

★ If you remember one thing · Exactly-once is real only inside the boundary that commits every participating effect.



Here is the final test of our boundary, where one command crosses between two independent authorities. Kafka protects its own records and offsets, but the same payment command also creates a bank charge outside that transaction bracket.

The processor charges the bank and then crashes before committing its Kafka transaction. Kafka correctly retries the input because no Kafka commit became visible, but the bank has already changed a ledger that Kafka cannot roll back.

The second processor attempt is safe for Kafka records and offsets because Kafka controls their transaction history. What prevents that retry from sending a second charge to the independent bank API?

PAUSE AND PREDICT

What protects the bank from the processor retry?

- **Kafka transaction state**
- **A bank idempotency key**

Skip this check

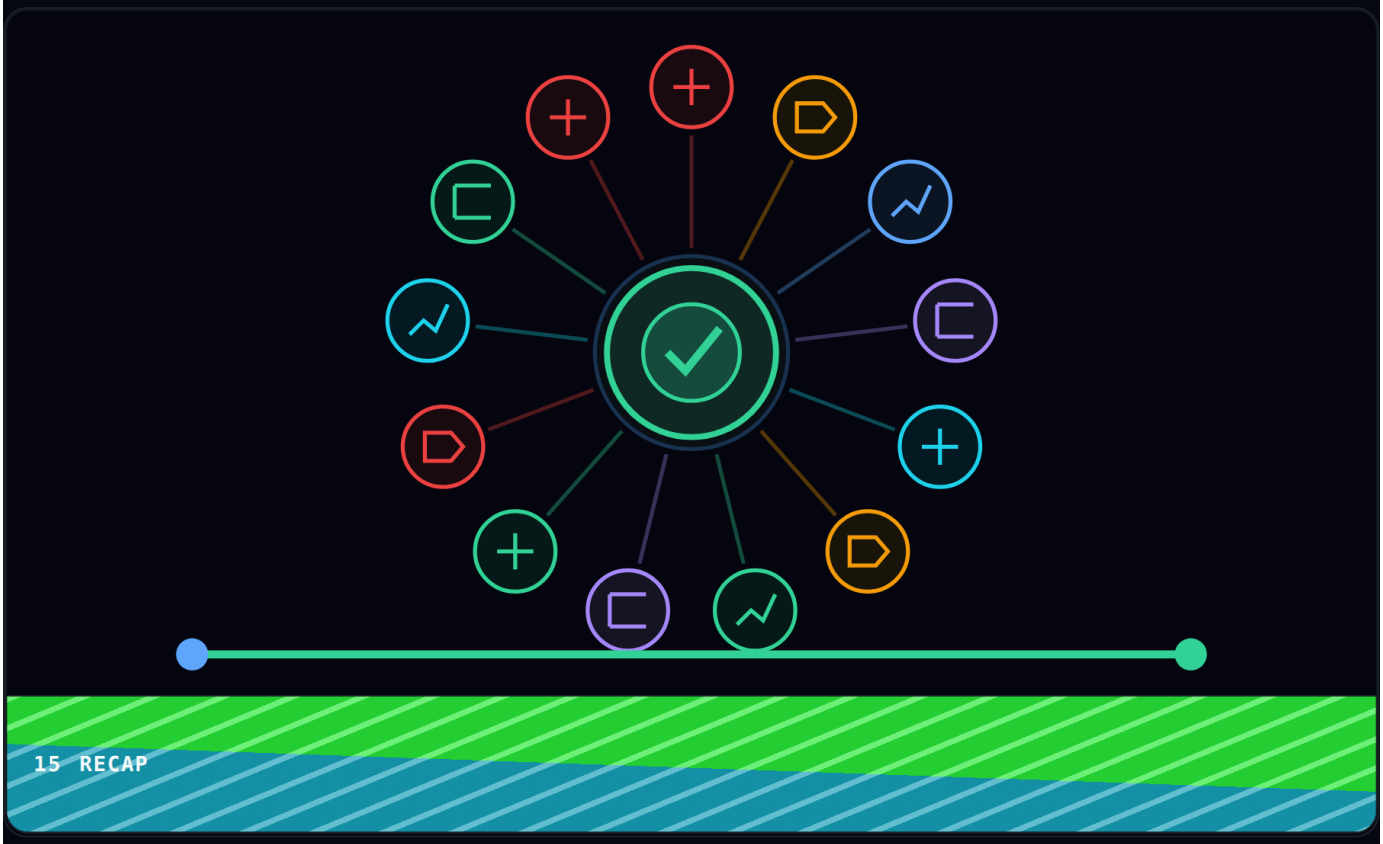
One committed Kafka bundle now sits beside two bank charge marks because the bank owns a separate retry history. This contrast is the boundary leak that a Kafka-only guarantee cannot close.

Toggle the External key control through both states. Compare the duplicate bank charges with the single charge produced when the bank remembers the request identity.

Exactly-once was real inside Kafka and effectively-once required another guard outside it. Now let us step back and connect the whole pattern.

STAGE 15 OF 15

Recap



Let us retrace the payment from the beginning and rebuild the protection story one layer at a time. A missing reply could not tell the client whether the payment failed or committed successfully.

At-most-once avoided duplicate delivery by accepting possible loss, while at-least-once recovered through retries that could repeat effects. Neither guarantee alone promised one business outcome, because delivery policy and effect protection answer different questions.

We then drew an explicit boundary because exactly-once is meaningful only for state controlled by one commit authority. Whenever a workflow crosses into another database, broker, or API, the guarantee must either expand through a shared protocol or stop honestly at that edge.

The idempotency key named one intended payment and turned matching retries into a replay of the first receipt. Storing a payload fingerprint also prevented a caller from reusing the key for a different operation and silently changing its meaning.

The deduplication window bounded how long the service remembered that key, which means retry safety ended when the stored identity expired. The retention period was therefore part of the correctness contract, not merely an internal cleanup preference.

Effectively-once combined repeated delivery with durable guards so three arriving copies produced one visible result. The duplicates still existed inside the system, but careful composition prevented them from multiplying the user-facing effect.

The transactional outbox committed the payment row and outgoing event row together, removing the unsafe dual-write gap. Its relay then published at least once, deliberately choosing possible redelivery over the risk of losing an event after the business change committed.

The inbox completed the producer-consumer pair by recording the event ID beside the consumer effect in one local transaction. A redelivered event then found a durable stamp and became a no-op instead of

creating a second receipt.

Producer epochs fenced the zombie process, while per-partition sequence numbers classified the current producer's next, repeated, and missing batches. Together they protected Kafka's append path from two different sources of duplicate or out-of-order work.

Kafka transactions placed output records and consumed offset 81 under one commit decision for read-committed consumers. A commit revealed the complete bundle, while an abort hid both the output and the corresponding input progress.

The external bank remained outside Kafka's transaction, exposing the exact point where another idempotency key was required. Kafka could retry its own work safely, but only the bank could recognize and collapse repeated charge requests in the bank ledger.

The worked values came from one deterministic illustration checked against Kafka, AWS, and Stripe primary documentation. The example was simplified for teaching, but its boundaries and failure modes reflect the real contracts those systems document.

If you remember one design habit, remember that every mechanism has both a scope and a lifetime. Strong systems make those limits explicit in the API contract and operating model, so callers know exactly which retries remain safe and for how long.

Retries are normal in distributed systems, so good designs expect them instead of treating them as rare accidents. One effect comes from durable identities, atomic local writes, and a commit boundary that promises only what it truly controls.