Kafka: From the Original Problem to Your First Event

November 5, 2026 kafka java spring-boot architecture event-driven

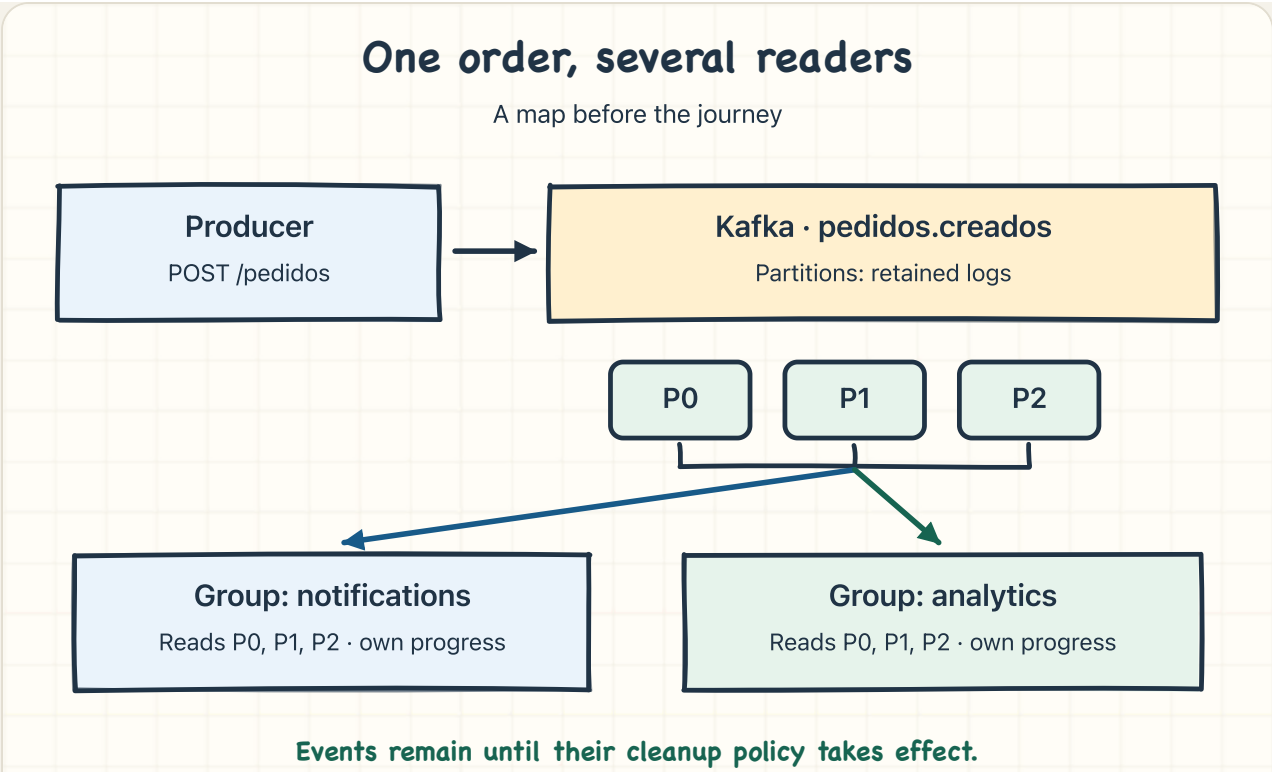
Checkout is working, but the email service takes eight seconds and the sales dashboard is down. You want to acknowledge the order without waiting for both, and tomorrow you may need to rebuild today's sales. Let us work through that problem with one order and a small runnable app.

Apache Kafka is a distributed platform for publishing, retaining, and reading event streams. A producer records a fact; Kafka stores records in ordered sequences called partitions; consumers read them and keep track of progress. Separate groups can move independently and revisit retained history. Your applications decide what the facts mean: notify a customer, build a view, or calculate a metric.



Our order will connect the explanation, the code, and the experiments.

My goal is to help you choose Kafka for a specific reason, follow a record through the system, and operate a small example. We will identify the need, publish a first order early, and then open the mechanism behind that result.



Kafka retains the stream; consumer groups interpret it for different purposes.

Your starting point	A route through this same article
New to Kafka	Read the problem and origins, run the first event, then work through foundations, parts, and experiments.
Already integrating applications	Try the first event, examine the decision criteria, then compare groups, replay, and failures.
Designing or operating systems	Start with internals, open the deeper notes, and evaluate ecosystem, contracts, and operations.

The lab uses **Java 21, Spring Boot 4.1.2, and Kafka 4.4.0 KRaft**. The complete project contains its source and tests. Both editions use the same verified code. Spanish names remain in that shared artifact: **`PedidoCreado`** means OrderCreated, **`pedidoId`** means order ID, and the **`productor`** and **`consumidor`** profiles select producer and consumer roles.

▼ IN THIS ARTICLE

01. 1. the Problem Before Kafka
02. 2. Why Kafka Appeared When It Did
03. 3. Publish Your First Event
04. 4. When Kafka Is Worth Its Cost

05. 5. Foundations You Need, without Requiring an Architecture
06. 6. the Parts: Open the Order Card
07. 7. the Internal Journey: From Send to Commit
08. 8. the Explained Lab and Its Experiments
09. 9. What Happens When Something Fails
10. 10. from the Lab to an Architectural Decision

1. the Problem Before Kafka

An Order Method with Too Many Responsibilities

Imagine a `createOrder` method that saves the order, calls the email service, and updates a sales dashboard. These actions happen around the same business moment, so putting them together feels natural. Their responsibilities are different, though. Recording the order belongs to checkout; sending the email communicates a result; aggregating sales builds a view for another reader.

When you chain them together, checkout inherits their waiting time. It also inherits awkward failure combinations. If analytics returns an error after the order has been saved, what do you tell the customer? Retrying everything might create another order or send another email. Doing nothing could leave the dashboard incomplete. You can encounter this problem with a modest workload; you do not need millions of requests to make the boundaries matter.

A durable work queue may be exactly what you need. Put an email task on it and let a worker finish it later. Kafka is not a prerequisite for asynchronous email. Microservices are not a prerequisite for messaging either: different modules in one application can publish and consume. Start by naming what must survive, which failure it must survive, and how long it must remain recoverable.

A Second Reader Changes the Question

Suppose the notification worker receives an order message and sends the email. A month later, analytics needs the same order facts to calculate hourly sales. It is not competing for the right to send the email. It needs an independent interpretation of the purchase. If you put both workers on the same competing-consumer queue, each receives only part of the work that both actually need.

You can publish copies into two queues. That is a valid publish/subscribe design. The next question is what happens when a third reader arrives later and needs purchases from before it existed. Is there a historical copy? How long does it last? Can the new reader scan it without moving the existing reader's progress? Those questions need explicit answers in the architecture. A broker's product name is not an answer by itself.

This is where a **retained sequence of facts**, with an independent reading position for each group, becomes interesting. A producer appends to that sequence. Different readers work through it at different speeds. Reading a page does not tear the page out of the notebook. Separating retained data from reading progress is the central idea I want you to keep as the terminology gets more specific.

Communication, Buffering, and Usable History

I separate three needs. **Communication** means making a fact available to another component. **Buffering** gives a reader time to catch up when it is temporarily busy or unavailable. **Usable history** lets a new or corrected reader revisit records that are still retained. Different systems can offer some or all of these properties. Compare the actual needs instead of adding features to a list with no business context.

> **Takeaway:** identify who needs the fact, what can wait, and what history must remain available. Kafka is one possible implementation of those requirements.

---

2. Why Kafka Appeared When It Did

Two Paths Needed a Shared Foundation

Kafka was built at LinkedIn to move user activity and operational data toward readers with different needs. Some applications reacted continuously; others loaded data for offline analysis. A custom path for every destination multiplied connections, formats, and recovery procedures.

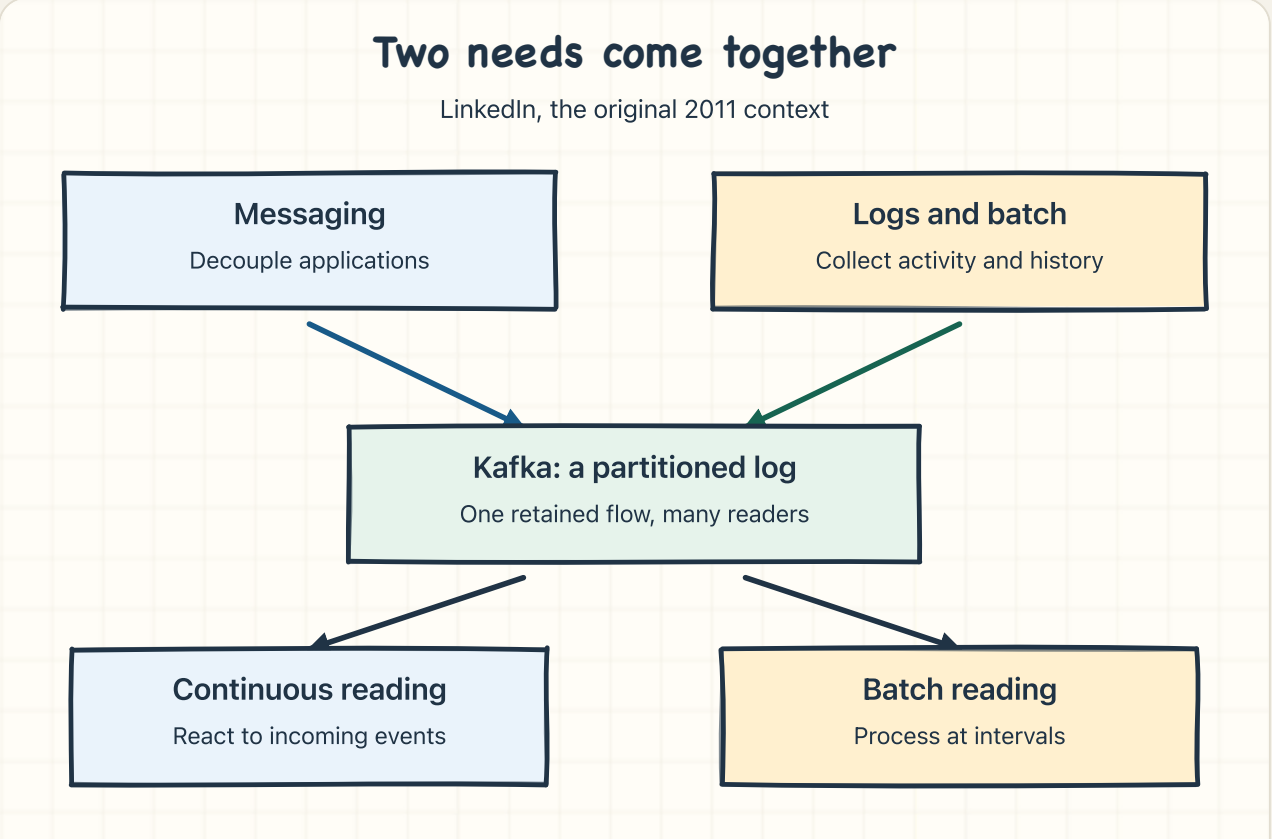
Jay Kreps, Neha Narkhede, and Jun Rao described the design in *Kafka: a Distributed Messaging System for Log Processing*, presented at NetDB 2011. The original paper compares enterprise messaging and the log aggregators of that period. The requirement was to move a large flow and make it useful to both continuous applications and offline systems.

Earlier or contemporary approach	What it offered	The need that encouraged another design
Enterprise messaging; APIs such as JMS and brokers such as ActiveMQ or RabbitMQ	Application communication, delivery, and routing	A suitable combination of throughput, shared history, and independent readers
Log aggregators such as Scribe and Flume	Collect activity and move it to data systems	Also serve continuous readers with their own progress
Batch processing with Hadoop	Analyze large datasets at intervals	React to that same activity while it continues arriving
Database logs	Order changes for recovery and replication	Apply change sequences to integration across systems

These are different kinds of tools: JMS is an API, while Hadoop is a processing environment. The table compares jobs that needed connecting. It also does not describe every capability in their current releases. For example, RabbitMQ Streams now provides retained logs and non-destructive reading.

The Combination Kafka Chose

Kafka retained records in **partitioned logs** and let readers traverse them by position. Partitioning distributes work; batching spreads network and write overhead. Separating storage from reading progress makes a new reader possible without adding another direct call to the producer.



The original need: use the same activity in systems moving at different speeds.

Reference	What it helps explain
NetDB, June 2011: paper by Kreps, Narkhede, and Rao	The original problem and design choices
LinkedIn, July 2011: Narkhede's public presentation	Activity, logs, and online/offline consumption in practice
Kreps, December 2013: *The Log*	A shared abstraction behind integration, replication, and processing
Kafka 4: KRaft replaces ZooKeeper	The metadata coordination used by our current lab

The LinkedIn presentation and The Log help trace that path. The current documentation explains the move to KRaft.

The shape of the requirement still matters when a purchase feeds notifications, metrics, and new business views. Your scale may be smaller. Evaluate the properties you need and the cost of supporting them. We will first observe the mechanism, then decide how it fits your app.

---

3. Publish Your First Event

Start with Docker and Compose, Java 21, and three terminal sessions. Maven comes through the included Wrapper. On Windows, use **`mvnw.cmd`** instead of **`./mvnw`**; shell examples assume macOS, Linux, or a compatible shell.

In the first terminal, get the project and build the executable:

```
git clone https://github.com/workmanuelherreramesen-lab/manuelherreramesen-profile-site.git
cd manuelherreramesen-profile-site/examples/spring/kafka
java -version
docker compose version
docker compose up -d --wait
./mvnw -DskipTests package
java -jar target/kafka-pedidos-1.0.0.jar --spring.profiles.active=productor
```

Wait for **`Started PedidosApplication`**. The producer serves **`localhost:8091`** and Kafka uses **`localhost:9092`**; both ports must be available. Use the same lab directory in every terminal. The first run can take longer while images and dependencies download. If Docker fails, inspect **`docker compose logs kafka`** before continuing.

In the second terminal, start the reader. Its profile selects the consumer component, and the last argument avoids starting another HTTP server:

```
java -jar target/kafka-pedidos-1.0.0.jar \
  --spring.profiles.active=consumidor \
  --spring.main.web-application-type=none
```

Wait for partition assignment, then publish from the third terminal:

```
curl -i -X POST http://localhost:8091/pedidos \
  -H 'Content-Type: application/json' \
  -d '{"pedidoId":"pedido-4821"}'
```

HTTP 202 returns event identity, partition, and offset. The reader logs the matching coordinates. The explained lab includes a complete captured response and a walkthrough of the classes. For now, identify the three boundaries: the request creates an event, the broker acknowledges publication, and the consumer interprets the record.

Try a small change: stop the consumer with **`Ctrl+C`**, publish **`pedido-4822`**, and restart the same reader. What do you expect to happen to that second order? Observe your terminal before opening the explanation.

> **– Solution**
>
> the reader finds the pending order because Kafka retained its record and the group can resume progress. Recovery depends on that history remaining available. Later we will set a retention window and inspect the group's position.

You now have a working flow. Leave the processes running for the experiments, or return here when you need them. Keep one HTTP producer instance; launching another on the same port while the first remains active produces a port conflict.

---

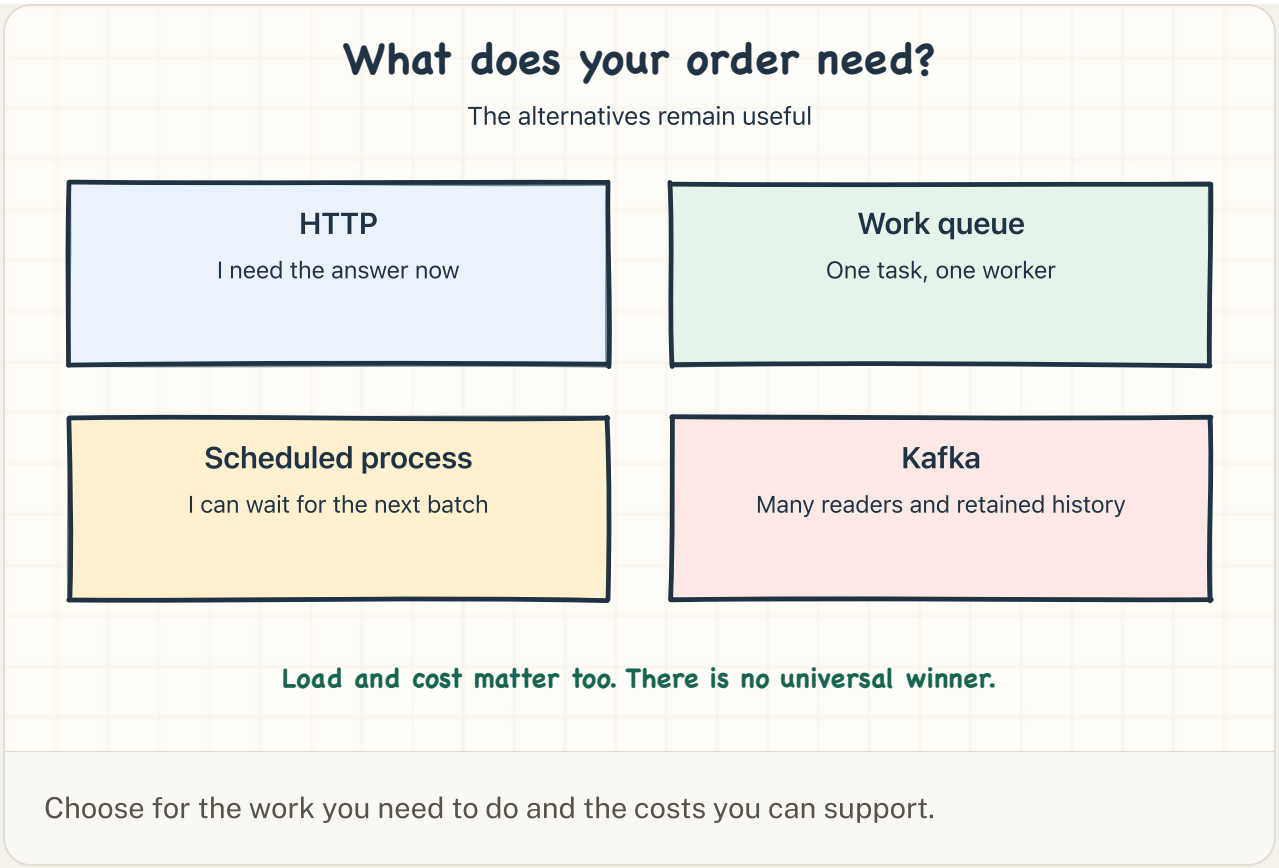
4. When Kafka Is Worth Its Cost

Write the Requirement Before Drawing Topics

Complete this sentence: “These readers need to process these facts, within this delay budget, with this recovery window.” If you cannot identify the readers or the recovery window, you may be selecting a tool too early. You do not need a large architecture document to begin. One purchase, two purposes, and a reconstruction requirement already give you meaningful criteria.

For the store, assume notifications should react promptly, analytics can run a few minutes behind, and analytics must be able to recalculate the last twenty-four hours after a bug fix. That is a reasonable case for investigating Kafka. It does not prove that Kafka is the cheapest choice. It explains which properties a comparison needs to preserve.

If all you need is one email job per purchase, a work queue may be more direct. If checkout must confirm a balance before accepting an order, a synchronous interaction or a local transaction may remain essential. Publishing an event later does not eliminate that decision. Asynchronous communication becomes useful when you can define what is pending and what the customer will see while it remains pending.



Choose for the work you need to do and the costs you can support.

Dominant requirement	First option I would evaluate	What I would check
Get an immediate decision from another system	HTTP/RPC	Latency, timeout, availability, and retry idempotency
Distribute tasks among workers	A work queue	Acknowledgements, retries, routing, and failure handling
Consolidate data at intervals	A scheduled process	Batch window, source load, and recovery
Keep a stream for independent readers and replay	Kafka or another distributed log	Retention, partitions, contracts, and operational ownership

What You Gain When the Model Fits

Independent progress means that one reader can lag behind another while the records remain available. Producers do not need to add another direct call every time a new consumer appears. Retained records provide a basis for recomputation after a logic fix. Partitions give you units of parallel work that can be distributed across the members of a group.

What the Team Must Pay for

The first cost is conceptual. A successful send is not completed business work. A connected consumer is not necessarily caught up. A read record is not necessarily a committed effect. You will design around

partial progress, recovery, and contracts that coexist across releases. If the team cannot observe those states, “we published it” can conceal more than it explains.

The second cost is operational: storage, transfer, capacity, upgrades, permissions, and incident response. Managed infrastructure can remove some server administration, but it does not choose your partition key or make an email operation idempotent. There is also an organizational cost: someone owns the event's meaning, approves changes, and responds when an important reader stops.

Signs That a Simpler Option May Be Better

For a small CRUD application with one integration and no reconstruction requirement, I would start with the simpler design. A nightly job may only require a carefully designed incremental query. A customer interaction requiring an immediate decision needs that response boundary first. And if nobody can own the stream's operation, a running container does not prove that you have a supportable system.

> **Takeaway:** justify Kafka through the properties of the flow and its recovery. Your explanation should still make sense before you mention the product name.

---

5. Foundations You Need, without Requiring an Architecture

An Event Reports; a Command Asks

“Create the order” asks for an action. The request may be rejected because information is missing or a business rule prevents it. “Order created” reports that something has already happened within a boundary you have defined. A consumer may react to that fact, but it should not quietly reinterpret it as a request still waiting for approval. A past-tense name helps communicate the intent; the agreed contract is what makes it real.

Asynchrony separates moments. The producer can finish publishing before the consumer finishes its job. A customer-facing product needs to represent the distinction: “we received your order” and “your order is ready” are different states. HTTP 202 can communicate acceptance, but an API must define what was accepted. In our example it means the broker acknowledged publication, not that every downstream consequence has completed.

The Network Moves Bytes, Not Live Java Objects

A Java instance does not cross the network as a running object. A serializer turns information into bytes, and another program interprets those bytes using an agreed format. We use JSON inside strings because it is easy to inspect and avoids introducing a schema registry into the first exercise. JSON alone is not an automatic contract: two programs can disagree about the meaning of a field while both parse it successfully.

Keep three identities separate. **`pedidoId`** identifies the business entity. **`eventId`** identifies one particular fact, which is useful for tracing and possible deduplication. The Kafka key influences partition selection in the client we use. We choose the order ID as the key, but the concepts remain distinct. One order can generate many events, each with its own event ID and the same partitioning key.

A Log and Its Reading Positions

A log is a sequence that receives appended records. Think of numbered pages in a notebook. The numbering lets a reader say where it wants to continue without requiring the writer to know what the reader understood. In Kafka, that sequence exists per partition. There is no single numbering scheme that orders every record in every topic across the cluster.

A sequence of facts is also different from a table of current state. Answering “what is this order's state now?” may require interpreting several events and building a view. That view can live in a database. Choosing a communication mechanism and choosing a query mechanism are connected decisions, but they are not interchangeable.

Concurrency, Partial Failure, and Idempotency

Two readers can move at different speeds, and two producers can publish concurrently. You need to decide which order matters. The business may require the sequence for one purchase while having no need to order every purchase against every other purchase. Finding that boundary lets you introduce useful parallelism without promising an unnecessary global order.

Partial failure means one participant can complete its action while another cannot determine whether it happened. A broker might store an event but lose the acknowledgement on its way back to the producer. A timeout tells the caller that confirmation did not arrive in time. It does not prove that the record is absent. This uncertainty is one reason retries need a business design rather than just another loop.

An idempotent operation tolerates repetition without multiplying its logical effect. Recording an event ID and applying its consequence once is one possible strategy. Sending the same email twice does not become idempotent just because the text is identical. Identity, durable state, and the boundary protecting the effect all matter.

Eventual consistency describes views that take time to reflect a shared fact. The purchase can have been published while the dashboard still shows the previous total. That is not a reason to accept arbitrary delay. Establish an expectation, measure it, and decide what the product should communicate when it is exceeded. An architecture needs a usable delay budget, not merely the word “eventual.”

Knowledge, Execution Requirements, and Design Choices

To understand the guide	To run the lab	Architectures you may combine with Kafka
Processes, networks, and data formats	Docker with Compose	A modular monolith or microservices
Asynchrony and partial progress	Java 21 and terminal sessions	DDD and hexagonal architecture

To understand the guide	To run the lab	Architectures you may combine with Kafka
Sequences, keys, and reading progress	The included Maven Wrapper	CQRS, Event Sourcing, and data platforms

Kafka needs server processes and compatible clients that can communicate with them. It does not require Spring, Java in every reader, Kubernetes, or a particular cloud provider. The broker image includes its own Java environment; the Java application requires the stated JDK. We chose Spring Boot to connect with the other guides and keep the first lab focused on the flow.

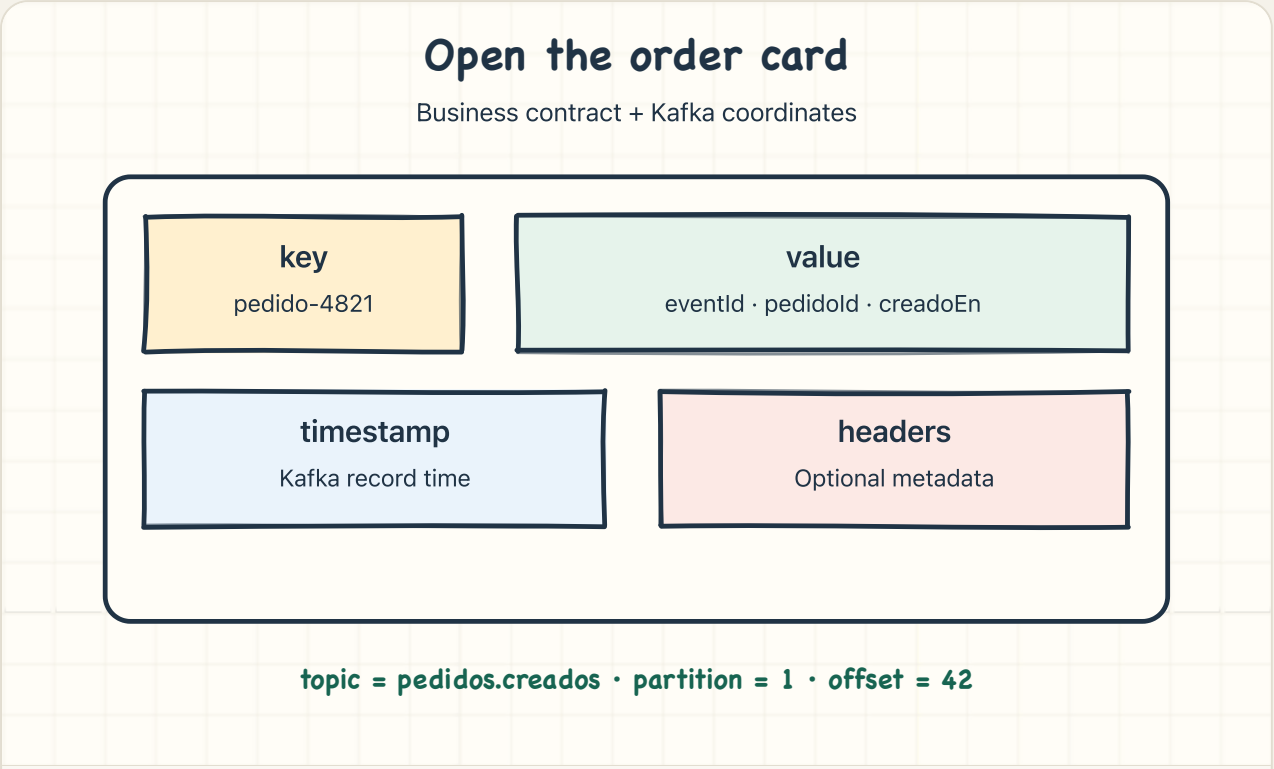
---

6. the Parts: Open the Order Card

Record, Key, and Value

A **record** is the unit we publish and read. Conceptually it includes a key, value, timestamp, and optional headers. Our record's value is the JSON representation of **`PedidoCreado`**. Its key is the order identifier. Headers can carry metadata such as correlation information; they are not a private hiding place for information the reader should not receive. The Apache introduction establishes this terminology.

The Kafka record timestamp and **`creadoEn`** inside our JSON deserve separate names. The second belongs to the event contract; the first is Kafka metadata whose treatment depends on configuration. If you later calculate time windows, decide which clock represents the business fact. Reading something today does not establish that the underlying action happened today.



The business contract and Kafka's storage coordinates answer different questions.

Topic and Partition

A **topic** is a named flow, such as **`pedidos.creados`**. It is not a separate queue for each reader. A topic contains **partitions**, ordered sequences that provide units for distributing storage and work. Our topic has three. A record belongs to one partition; having three partitions does not mean the same record is written to all three. Replication is a separate choice.

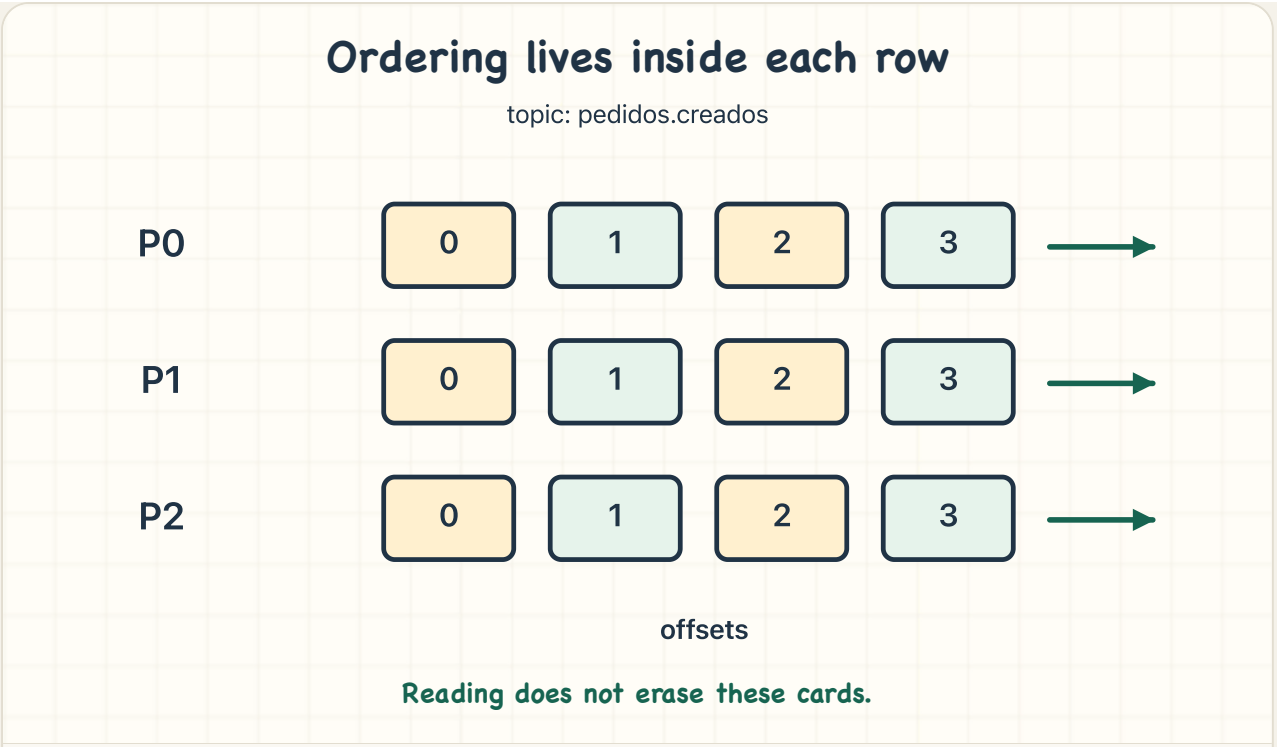
A **key** helps the producer choose a partition. With this client's default partitioning, a stable non-null key maps to the same partition while the partition count and strategy remain unchanged. That gives the events for **`pedido-4821`** a useful local sequence. It does not make uncoordinated producers agree automatically on business order. You must still define the sequence in which they publish.

Increasing the partition count can increase potential parallelism, but it may also change the mapping from keys to partitions. If your design depends on historical ordering per entity, do not treat that change as transparent. Plan how to migrate, drain, or version the flow. A design that is correct with three partitions should explain what happens when it needs six.

Offset: A Local Coordinate

An **offset** is a position inside a partition. To locate a record, use topic, partition, and offset together. Offset 42 in partition zero and offset 42 in partition one are different coordinates. Comparing them does not tell you which purchase happened first. They are also not interchangeable with an event's business identity in an external system.

A consumer has a **current position** as it reads and a **committed offset** that records where the group should resume. The committed value normally refers to the next record to process: after finishing record 42, progress may be committed as 43. The consumer configuration reference is the place to check offset reset, commits, and client reading behavior.



Three partitions are three sequences. There is no total ordering across the rows.

Producer, Broker, and Controller

The **producer** is the client that publishes. Our application uses **`KafkaTemplate`** as a convenient integration over the Java client and observes its asynchronous result. A **broker** is a server process that serves requests and stores partitions. A **cluster** is the coordinated set of processes providing the service. One broker is useful for learning, but it does not provide a second broker holding another copy of the data.

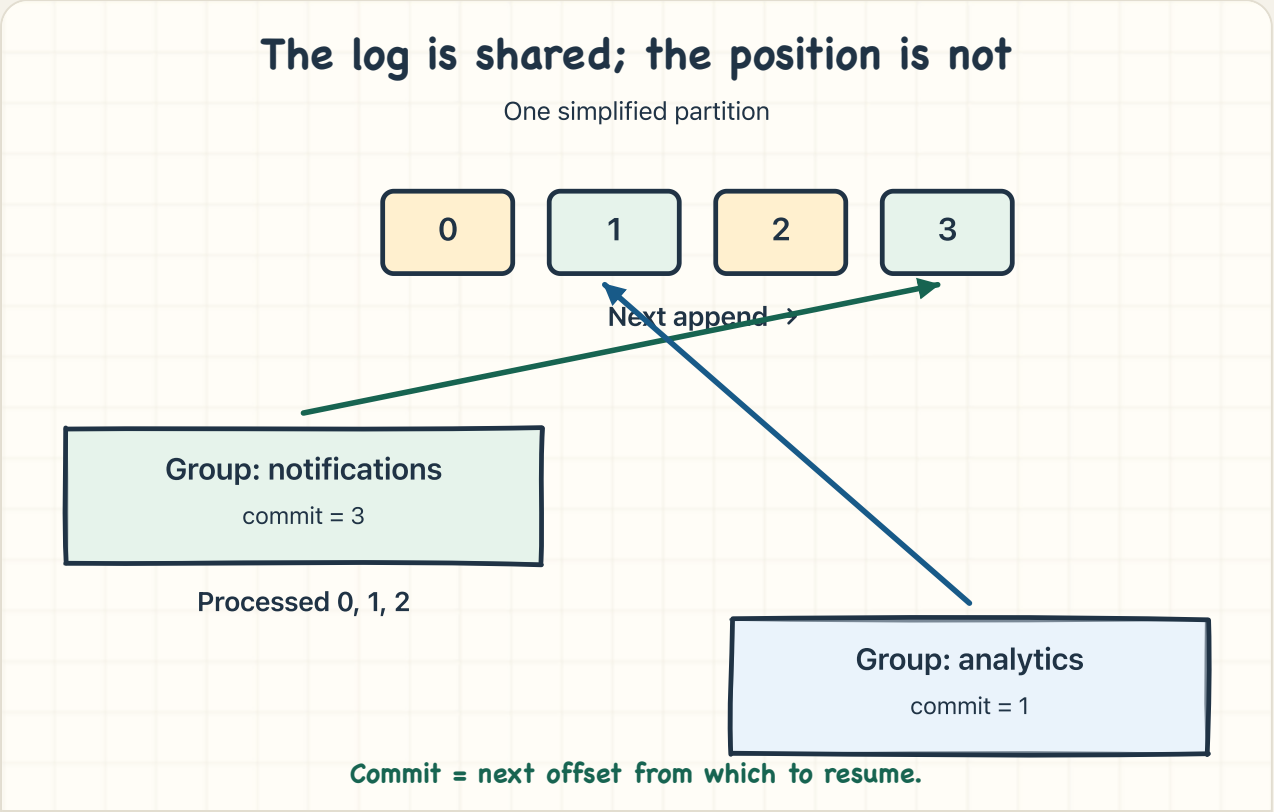
The **controller** manages cluster metadata: registered brokers, partitions, leaders, and related changes. KRaft controllers maintain a metadata log through Raft consensus. This does not mean each purchase is committed through a Raft transaction across every broker. Business partition replication has its own mechanism. Separating metadata coordination from business records prevents a surprisingly common misunderstanding.

Our local process combines broker and controller roles for convenience. The KRaft operations guide explains roles and quorum considerations for real operations. A combined-role container is not a template from which to infer the right topology for a critical service.

Consumer and Group

A **consumer** requests records, interprets them, and does its work. A **consumer group** coordinates members that divide subscribed partitions. In the model used here, a partition is assigned to one member at a time within a group. Another group can read the same partitions from its own positions without competing with the first group for those records.

With three partitions and two consumers in **`avisos-demo`**, one member might receive two partitions and the other one. The specific assignment depends on the assignor and the group's state. Adding a fourth member does not create a fourth partition: at least one member will have no partition from this topic to process. More processes do not remove the parallelism boundary you chose.



Notifications and analytics share records, not progress. Committing does not erase the log.

One possible assignment of three partitions

Unassigned	Member A	Member B
	✓ P0 ✓ P1 ✓ P2	

Tokens represent partition assignments, not records removed by reading.

A joins and can read all three topic partitions in this group.

B joins. After rebalance, one possible assignment gives A two partitions and B one. Ownership does not overlap within the group.

B leaves. After reassignment, A resumes all partitions from group commits. The log still retains its records.

– **Going deeper**

> in **Share Groups**, members share work through record acquisition and acknowledgement, with temporary locks and possible redelivery. More consumers than partitions can therefore collaborate. Our conventional consumer groups assign each partition to one member and commit a log position; records may return after a failure. These are different progress models with their own APIs and configuration. This lab uses `KafkaConsumer`, not `KafkaShareConsumer`. Read Share Consumer in the Kafka 4.4 design before carrying rules from one model into the other.

---

7. the Internal Journey: From Send to Commit

1. Discovering Where to Write

An application starts with addresses in `bootstrap.servers`. They let it discover metadata: which brokers exist and which broker leads each partition. A producer does not necessarily send everything to the first bootstrap address and ask that server to route all records. It uses the metadata to connect to the appropriate destination.

That explains a common Docker failure. The first connection to `localhost:9092` succeeds, but the broker advertises a name that the application cannot resolve. The next connection fails. `listeners` describes where a server listens; `advertised.listeners` describes the addresses it tells clients to use. The listener configuration guide documents the distinction. Our setup provides a host address and an internal Docker address.

Three Responsibilities to Keep Separate

Role	What it coordinates	In the order journey
KRaft controller	Cluster metadata and leadership changes	Records which broker leads each partition
Partition leader	Writes and replication for that log	Receives our `Produce` request
Group coordinator	Membership, assignment, and group offsets	Helps `avisos-demo` resume reading

A broker can perform several roles. The group coordinator is the broker responsible for that group; it need not lead the partition containing our order. KRaft does not hold a vote for every `PedidoCreado`: metadata consensus and event replication follow different paths.

2. Serializing, Partitioning, and Batching

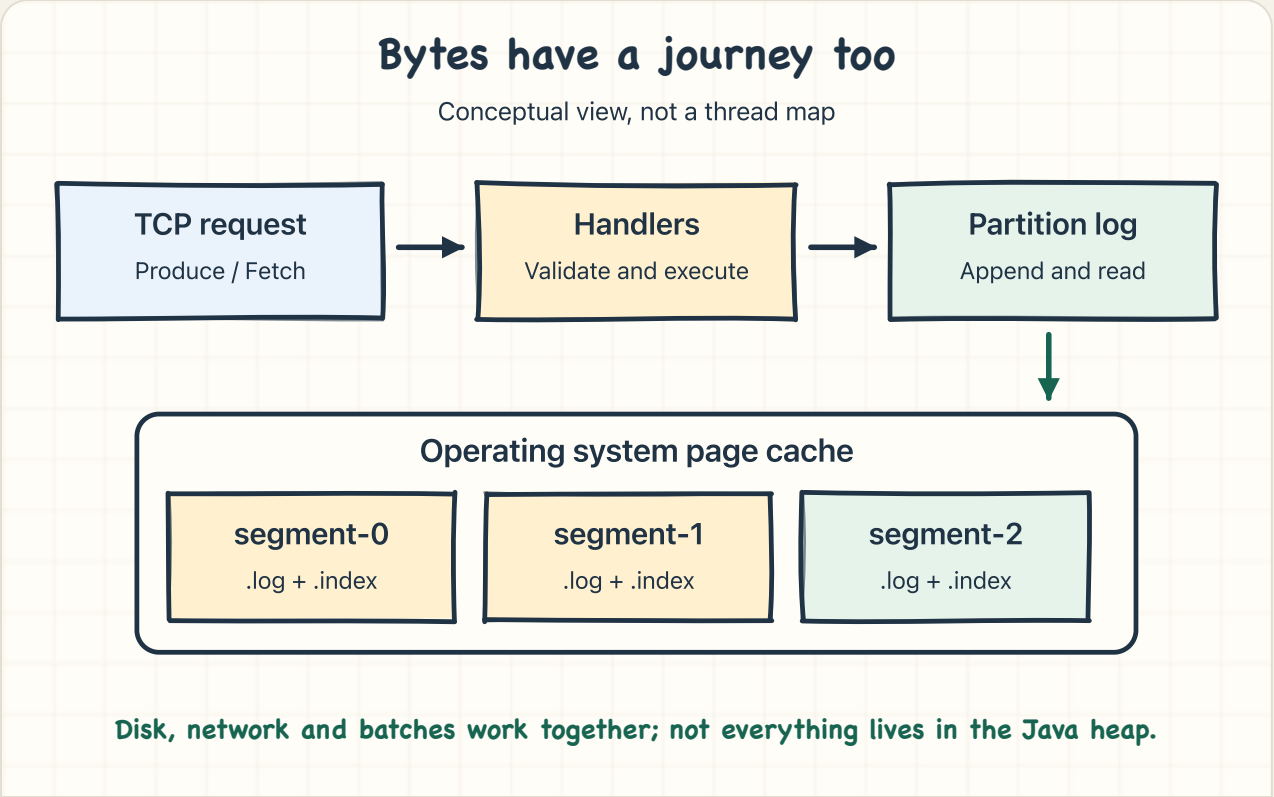
The producer serializes the key and value, selects a partition, and can group records into **batches**. Sending a batch spreads the cost of a network request across many small records. Batch size and the time allowed to form batches influence throughput, memory, and latency. This is one reason a production tuning discussion needs a workload and a delay budget rather than a universally “best” collection of settings.

The producer reference describes acknowledgements, idempotency, timeouts, and batching. Our request has bounded waiting time so it cannot wait indefinitely for the broker. When that time expires, the honest result is uncertainty about confirmation, not proof that no record was written.

3. Receiving and Appending

The broker's network layer accepts requests and passes them to handlers. A `Produce` request ultimately appends records to the relevant logs, subject to validation and server conditions. A `Fetch` request asks for data starting at particular positions. The network layer and message format references describe these interfaces and the batch representation.

A partition log is divided into **segments**. The active segment receives appends, while older segments provide manageable units for retention and access. Indexes help locate positions without treating the topic as one huge in-memory collection. The log implementation is a useful next read when "Kafka stores messages" is no longer a detailed enough mental model.



A log has files, indexes, and an access path. It is not an endless list of retained Java objects.

Kafka makes use of the operating system and its page cache. Sequential access, batching, and efficient transfer paths help move data without keeping a permanent Java object for every retained event. This does not mean all data lives in memory. It also does not mean every acknowledgement forces every replica to perform a physical `fsync`. Replication, persistence, and physical synchronization are connected but distinct mechanisms.

> – Going deeper
>
> on compatible paths, zero-copy avoids copying file bytes into an application buffer and back into the kernel: the OS can transfer them from its page cache to the socket. That reduces copying and CPU work while reading. TLS can require another path for encryption; do not assume this optimization applies to every connection. Producers can also compress batches, trading CPU and latency choices for less network traffic and disk usage. Batching, compression, and caching explain part of the performance; a benchmark with a different key distribution does not establish your capacity.

4. Replicating and Acknowledging

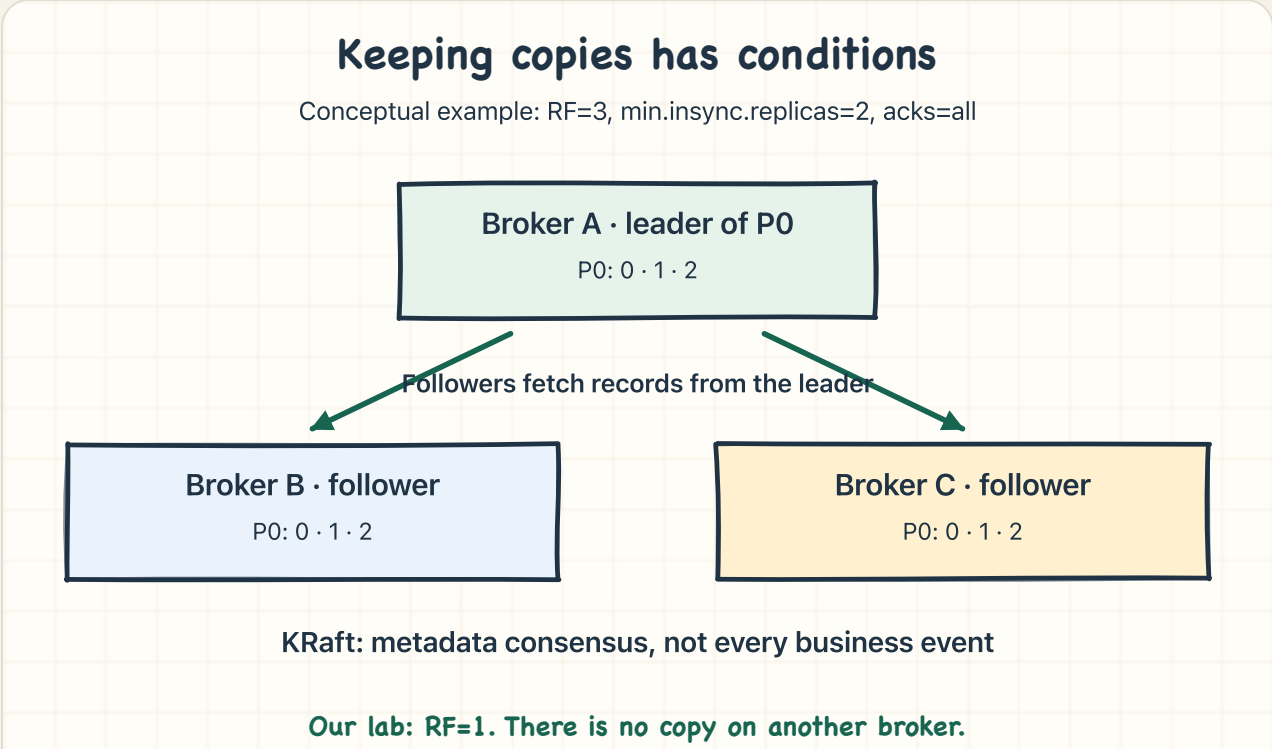
A replicated partition has a leader and follower replicas. The leader accepts writes and followers fetch records to maintain their copies. The **ISR** is the set of replicas considered in sync; the leader is part of that set. Three partitions are not three replicas. You can have three partitions that exist on only one broker.

With **`acks=all`**, the leader waits for acknowledgement from the ISR replicas. **`min.insync.replicas`** sets a minimum condition for accepting the write; it is not a fixed number of acknowledgements to wait for. For replication factor three and a minimum of two:

Current ISR	What happens with **`acks=all`**?
Leader and two followers: 3	Wait for all three ISR replicas
Leader and one follower: 2	Wait for both; the out-of-sync replica does not by itself block the write
Leader alone: 1	The minimum is not met; the write fails

The ISR can change during a request, and an error alone does not prove that no bytes were written. This is an acknowledgement policy tied to replication state, rather than protection against every possible loss. The topic configuration reference spells out the conditions.

Our replication factor is one. **`acks=all`** therefore acknowledges the one available replica; it does not create another. A Docker volume can preserve data across container recreation, but that volume is still on the same machine. Losing that storage leaves no other lab broker with a copy. We are demonstrating communication and recovery of reading progress, not certifying infrastructure fault tolerance.



A conceptual replicated topology. The one-broker lab does not provide this redundancy.

5. Reading Is Not Removing

A consumer requests records through **`poll`**. Its client handles connections, fetching, and group participation. Spring adds a container that runs that activity and invokes our listener method. **`@KafkaListener`** does not make the broker call a Java method inside your process. The consumer application is actively reading through its client.

Receiving records advances the client's local position. Your code can still fail while parsing or applying an effect. We therefore need an explicit progress policy. In our configuration, client auto-commit is disabled and the Spring container uses **`record`** acknowledgement mode: after the listener returns successfully for a record, the container commits the corresponding progress. The listener-container reference explains that policy.

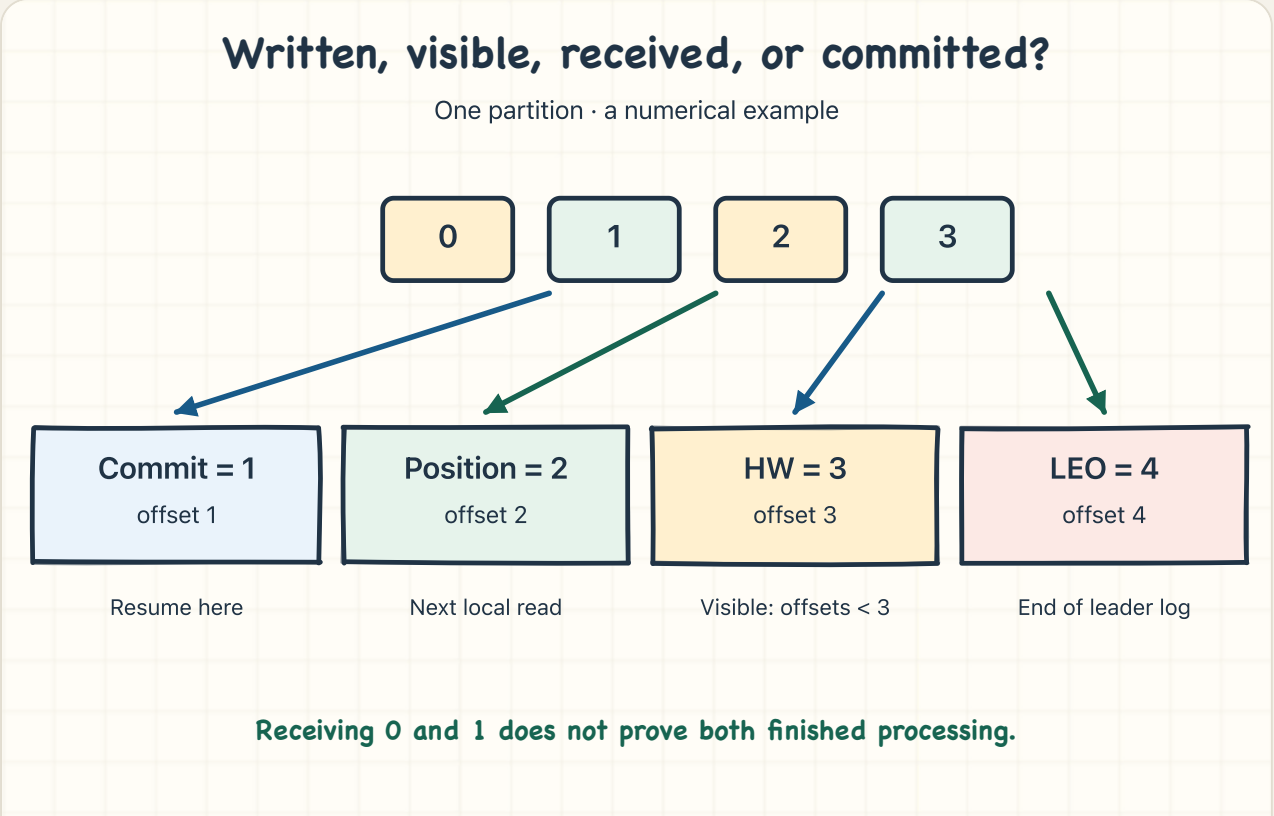
Kafka stores group progress through internal mechanisms, including its offsets topic. Committing does not edit or delete the original business record. Another group can still read it while it remains available. Conversely, a slow group's bookmark does not prevent retention from removing old data. Data lifetime and reading progress have separate responsibilities.

Four Positions on One Partition

Consider a partition containing offsets **`0, 1, 2, 3`**. These boundaries describe different things:

Position	Example value	Meaning
Leader log end offset (LEO)	4	Next offset at the end of its log; record 3 is present
High watermark (HW)	3	Exclusive boundary of replicated data visible to ordinary reads: 0, 1, and 2
Client position	2	Next read starts at 2; `poll` already returned earlier records
Group committed offset	1	A member resuming that partition starts at 1

The consumer could have received 0 and 1, processed only 0, and committed 1. If it stops now, record 1 will be delivered again even though its local position was already 2. Meanwhile, record 3 being present on the leader does not make it readable beyond the HW. A client's position is not proof that your code finished its work.



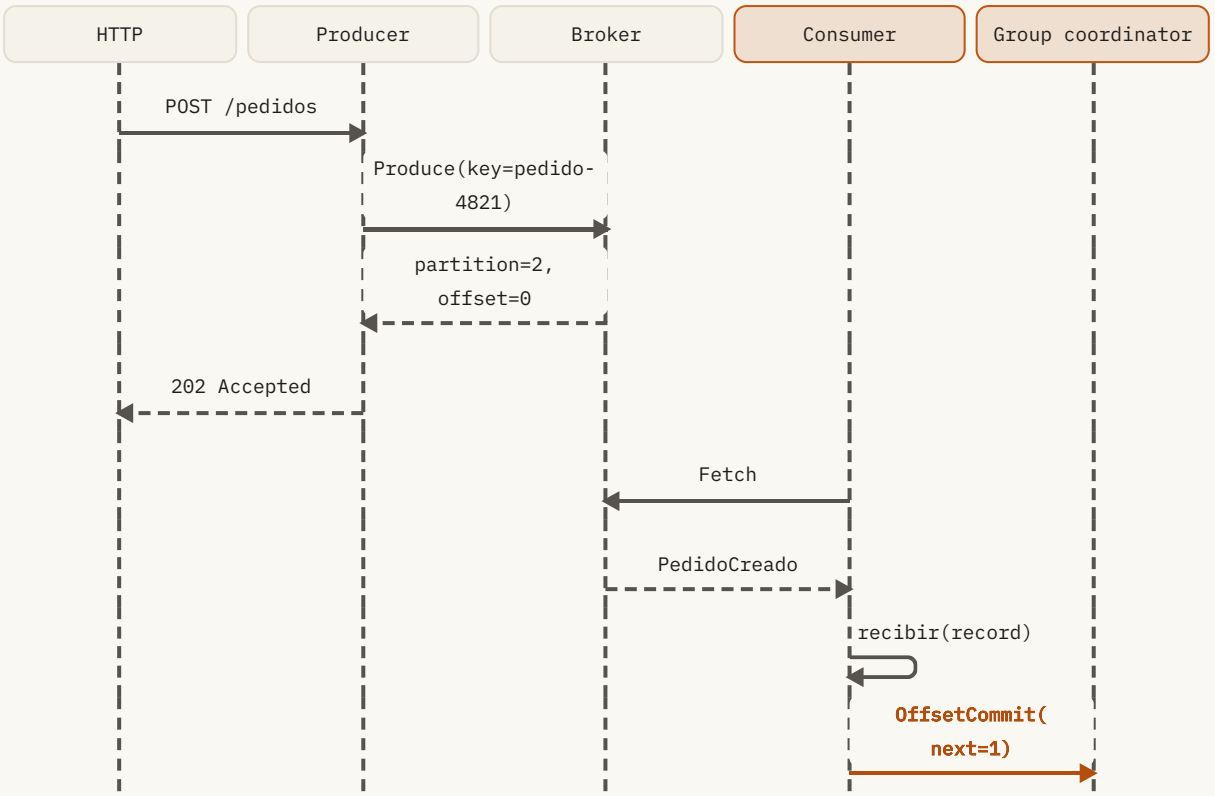
Four different boundaries, all using next-offset or exclusive-limit conventions.

– Going deeper

`isolation.level=read_committed` adds the **last stable offset (LSO)**. It is the smaller of the HW and the first offset belonging to an open transaction. Consumers do not cross that boundary and filter records from aborted transactions. If an open transaction starts at 1 while HW is 3, LSO is 1: later

replicated records may still be unavailable to that reader. LSO is unrelated to your group's commit. See the definitions in the consumer API.

Publishing, reading and committing are separate steps



The request creates an event in our application. Kafka has not acknowledged it yet.

The client sends bytes to the selected partition. This lab has only one replica.

The broker acknowledges publication. These coordinates illustrate a run; your values can differ.

The response confirms acknowledged publication. Consumer work may still be pending.

The consumer requests data. The broker does not directly invoke our Java method.

The record is read and remains in the log. Reading does not remove the card.

The listener parses JSON and writes its log line. A failure here is not a failed publication.

With ack-mode=record, Spring commits after successful return. The group can resume at the next offset.

6. Retention, Compaction, and Recovery

Our topic uses **`cleanup.policy=delete`** with a twenty-four-hour retention setting. Removal works through segments and maintenance activity. Do not interpret the number as a separate timer that deletes every individual record at exactly its twenty-four-hour birthday. It expresses a policy, not that level of per-record temporal precision.

Compaction serves a different purpose: retaining the latest value per key, subject to the cleaner's timing and related rules. It can support state reconstruction, but it does not retain every business transition. If older order versions disappear, the history you wanted for audit may be gone. Tombstones, delayed readers, and cleanup windows need deliberate consideration before a compacted topic becomes part of a state design.

> – **Going deeper**
>
> imagine a compacted topic with `0: A=pending`, `1: B=pending`, `2: A=paid`, and `3: B=null`. Cleaning can remove 0 because 2 retains A's latest value. The null value at 3 is a **tombstone**, marking B for deletion; it is not the JSON text `"null"`. The cleaner can remove B's older value and, after the relevant retention period, the marker itself. Surviving offsets are not renumbered. Cleaning is asynchronous, so reads can still encounter several versions. A rebuilding reader must finish before losing the deletion markers it needs; review `delete.retention.ms` in the topic configuration.

A replay revisits records that still exist. It cannot recover data removed by retention. It also cannot guarantee the same result when processing depends on external information that has changed. If an old calculation looked up a price that is different today, rerunning the same code may produce a different answer. Decide what information belongs in the event and which other inputs need versioning or preservation.

> **Takeaway:** publication, replicated storage, reading, processing, and committing are separate steps. During a failure, identify which happened and which remains uncertain.

---

8. the Explained Lab and Its Experiments

If you ran the first event, Kafka, the producer, and a consumer are already running. We will now inspect their configuration and change one variable at a time. If you closed the session, use that earlier walkthrough to restart it.

The Environment You Just Ran

This is the project's version-and-digest-pinned Compose file:

compose.yml

```
services:
  kafka:
    image:
apache/kafka:4.4.0@sha256:833b867a9585a141da455dc2b57431c65cbccb27f11c91f694444cca9f777e
```

```
0f
    ports:
      - "127.0.0.1:9092:9092"
    environment:
      CLUSTER_ID: "MkU3OEVBNTcwNTJENDM2Qk"
      KAFKA_NODE_ID: 1
      KAFKA_PROCESS_ROLES: broker,controller
      KAFKA_LISTENERS: EXTERNAL://:9092,INTERNAL://:19092,CONTROLLER://:9093
      KAFKA_ADVERTISED_LISTENERS: EXTERNAL://localhost:9092,INTERNAL://kafka:19092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
EXTERNAL:PLAINTEXT,INTERNAL:PLAINTEXT,CONTROLLER:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_CONTROLLER_LISTENER_NAMES: CONTROLLER
      KAFKA_CONTROLLER_QUORUM_VOTERS: 1@kafka:9093
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "false"
      KAFKA_LOG_DIRS: /var/lib/kafka/data
      KAFKA_LOG_RETENTION_HOURS: 24
      KAFKA_HEAP_OPTS: -Xms128m -Xmx256m
    volumes:
      - kafka-datos:/var/lib/kafka/data
    healthcheck:
      test: ["CMD", "/opt/kafka/bin/kafka-topics.sh", "--bootstrap-server",
"kafka:19092", "--list"]
      interval: 5s
      timeout: 10s
      retries: 20
      start_period: 15s
volumes:
  kafka-datos:
```

`EXTERNAL` provides the host connection, and **`INTERNAL`** lets tools inside the Compose network use **`kafka:19092`**. The controller listener serves the KRaft role. These are names we assign to listeners, not three separate products. The external port is bound to the local interface. PLAINTEXT is used because this is a local learning environment.

The volume preserves the log when the container is stopped. Replication settings of one allow internal topics to run on a single broker. Do not carry these settings into a production design without reevaluating the required guarantees. The fixed cluster ID belongs to this lab's storage; changing it over an existing volume is not equivalent to starting a clean new cluster.

Use **`docker compose ps`** and the service logs to inspect the environment. The health check queries Kafka's API, and the volume preserves history across sessions.

Dependencies and Application Entry Point

The project inherits Spring Boot 4.1.2 and targets Java 21. These dependencies provide Kafka integration, HTTP, and validation. Boot manages compatible versions rather than making the example independently pin every transitive library:

pom.xml: dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webmvc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

The entry point is the familiar Boot application. If you want more background on what Boot configures, the site's Spring Boot guide works through that mechanism separately:

PedidosApplication.java

```
@SpringBootApplication
public class PedidosApplication {
    public static void main(String[] args) {
        SpringApplication.run(PedidosApplication.class, args);
    }
}
```

We declare a topic through Spring's administrative integration. The full project contains the imports; the article's synchronized fragments focus on the decisions we are studying. **`NewTopic`** asks for the topic during startup, so the first successful application startup also establishes the flow used by later terminal commands.

ConfiguracionKafka.java: topic

```
public static final String TOPIC = "pedidos.creados";

@Bean
NewTopic pedidosCreados(@Value("${demo.topic:pedidos.creados}") String topic) {
    return TopicBuilder.name(topic)
            .partitions(3)
            .replicas(1)
            .config("cleanup.policy", "delete")
```

```
            .config("retention.ms", "86400000")
            .build();
}
```

The expression **`${demo.topic:pedidos.creados}`** asks Spring for a configurable name, defaulting to **`pedidos.creados`**. Tests use a different name to keep their records separate from your session.

If a topic already exists with different settings, do not assume restarting this application recreates it exactly as shown. Inspect it with the administrative tools. A fresh lab begins with fresh storage; a repeated session may preserve old records and group offsets. Learning to observe that history is part of the exercise, not a setup defect.

The Event and the Configuration

The event contract is deliberately small: event identity, order identity, and creation time. A Java **`record`** expresses that data without suggesting that consumers should mutate a shared object. The field names remain the same in both language editions because both use one tested project.

PedidoCreado.java

```
public record PedidoCreado(UUID eventId, String pedidoId, Instant creadoEn) {}
```

Read the configuration as carefully as the classes:

application.yml

```
server:
  port: ${PORT:8091}
spring:
  application:
    name: kafka-pedidos
  kafka:
    bootstrap-servers: ${KAFKA_BOOTSTRAP_SERVERS:localhost:9092}
    admin:
      fail-fast: true
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
      acks: all
      properties:
        enable.idempotence: true
        max.block.ms: 5000
        request.timeout.ms: 5000
        delivery.timeout.ms: 10000
    consumer:
      group-id: ${KAFKA_GROUP:avisos-demo}
      auto-offset-reset: earliest
```

```
    enable-auto-commit: false
    key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
  listener:
    ack-mode: record
    concurrency: 1
```

The string serializer turns our JSON string into bytes, and the string deserializer returns a string for the listener to interpret. **`acks=all`** and producer idempotency make the publication intent explicit. The timeouts bound waiting. **`enable-auto-commit=false`** leaves progress control to the Spring container, whose **`record`** acknowledgement mode ties the commit to successful listener completion.

`earliest` applies when the group does not have a valid position from which to continue. It is not an instruction to restart from the beginning on every launch. If **`avisos-demo`** has already committed progress, restarting with the same ID respects that progress. We will test the difference by changing only the consumer's group name.

The HTTP Producer

The controller accepts a short **`pedidoId`**, creates a fresh event, serializes it, and publishes using the order ID as the key. It waits for the send result before returning partition and offset. HTTP 202 confirms acknowledged publication. It does not certify that an email has been sent or a sale counted; neither effect exists in this project.

PedidoController.java

```
@RestController
@Profile("productor")
@RequestMapping("/pedidos")
public class PedidoController {
    private final KafkaTemplate<String, String> kafka;
    private final JsonMapper json;
    private final String topic;

    public PedidoController(KafkaTemplate<String, String> kafka, JsonMapper json,
            @Value("${demo.topic:pedidos.creados}") String topic) {
        this.kafka = kafka;
        this.json = json;
        this.topic = topic;
    }

    public record CrearPedido(
            @Pattern(regexp = "[A-Za-z0-9-]{1,40}") String pedidoId) {}
    public record Publicado(UUID eventId, String pedidoId, int particion, long offset)
{}

    @PostMapping
    @ResponseStatus(HttpStatus.ACCEPTED)
```

```
    public Publicado crear(@Valid @RequestBody CrearPedido entrada) {
        if (entrada.pedidoId() == null) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Falta pedidoId");
        }
        var evento = new PedidoCreado(UUID.randomUUID(), entrada.pedidoId(),
Instant.now());
        try {
            var resultado = kafka.send(topic, evento.pedidoId(),
                    json.writeValueAsString(evento)).get(12, TimeUnit.SECONDS);
            var metadata = resultado.getRecordMetadata();
            return new Publicado(evento.eventId(), evento.pedidoId(),
                    metadata.partition(), metadata.offset());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw noConfirmado(e);
        } catch (ExecutionException | TimeoutException |
org.apache.kafka.common.KafkaException
                | org.springframework.kafka.KafkaException e) {
            throw noConfirmado(e);
        }
    }

    private ResponseStatusException noConfirmado(Exception causa) {
        return new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE,
                "Publicación no confirmada; un reintento puede duplicar el evento",
causa);
    }
}
```

If confirmation cannot be obtained, the controller returns 503 while preserving the possibility that retrying could duplicate the event. The blocking wait is intentional in this small lab: it exposes the acknowledgement boundary without introducing another asynchronous HTTP flow. A high-load API might handle concurrency differently, but it would still need to observe the publication outcome.

Each request generates a new **`eventId`**. Repeating a **`POST`** is therefore not business-idempotent. Kafka producer idempotency protects certain protocol retries; it does not recognize that two separate HTTP requests represent the same purchase. This limitation belongs beside the code, where you can see it before replacing a teaching signal with real business work.

An HTTP Error That Explains Uncertainty

The controller throws an exception with context, but its warning must survive conversion to an HTTP response. This advice applies only to the order controller and returns **`ProblemDetail`**, the error format supported by Spring MVC, with a status, detail, and request path. It exposes neither stack traces nor internal exceptions.

```
@RestControllerAdvice(assignableTypes = PedidoController.class)
```

```
public class ErroresHttp {
    @ExceptionHandler(ResponseStatusException.class)
    public ProblemDetail responder(ResponseStatusException error) {
        var problema = ProblemDetail.forStatusAndDetail(error.getStatusCode(),
                error.getReason() == null ? "Solicitud no completada" :
error.getReason());
        problema.setTitle(error.getStatusCode().value() == 503
                ? "Publicación no confirmada" : "Solicitud rechazada");
        return problema;
    }
}
```

The HTTP test verifies **`application/problem+json`**, status 503, and the warning “un reintento puede duplicar” in **`detail`**. You can also observe it by stopping the broker, requesting an order, and starting Kafka again. The producer's waits remain bounded.

Stopping the broker produced this HTTP response. The warning is present in the response body, rather than only in the server log:

```
{
  "detail": "Publicación no confirmada; un reintento puede duplicar el evento",
  "instance": "/pedidos",
  "status": 503,
  "title": "Publicación no confirmada"
}
```

The Consumer

The listener accepts the full record so that its coordinates are observable. It parses the JSON and writes one log line. The group comes from configuration. Running this exact code with a different group changes which reading progress it shares, making it a convenient way to isolate the group concept without adding another business service.

ConsumidorPedidos.java

```
@Component
@Profile("consumidor")
public class ConsumidorPedidos {
    private static final Logger log = LoggerFactory.getLogger(ConsumidorPedidos.class);
    private final JsonMapper json;

    public ConsumidorPedidos(JsonMapper json) {
        this.json = json;
    }

    @KafkaListener(id = "lector-pedidos", topics = "${demo.topic:pedidos.creados}",
            groupId = "${spring.kafka.consumer.group-id}")
```

```
    public void recibir(ConsumerRecord<String, String> record) {
        PedidoCreado evento;
        try {
            evento = json.readValue(record.value(), PedidoCreado.class);
        } catch (JacksonException error) {
            log.error("PEDIDO_INVALIDO topic={} key={} partition={} offset={}",
                    record.topic(), record.key(), record.partition(), record.offset());
            throw error;
        }
        log.info("PEDIDO eventId={} pedidoId={} key={} partition={} offset={}",
                evento.eventId(), evento.pedidoId(), record.key(),
                record.partition(), record.offset());
    }
}
```

A processing error stops the listener container in this lab. That makes an unprocessable JSON record visible instead of quietly treating it as completed after some retry limit. It is a deliberate teaching policy, not a universal production recommendation:

ConfiguracionKafka.java: errors

```
@Bean
CommonContainerStoppingErrorHandler detenerAnteError() {
    return new CommonContainerStoppingErrorHandler();
}
```

The application process may remain alive after its listener stops. Fixing the cause and restarting the reader is required before it can continue; the pending record will be encountered again. Later we will discuss bounded retries and error topics. Neither should be added without deciding what skipping or delaying the event means to the business.

Working Through the Six Experiments

Write a one-sentence prediction before changing anything. Run the instruction, compare the result, and then open the explanation. Your IDs and offsets may differ; the relationships between them are the point of the exercise.

Experiment 1: Publish and Locate the Record

Predict: can a producer publish before any consumer is ready? What exactly does HTTP 202 acknowledge?

Try: publish another order and compare its coordinates with the reader terminal.

```
curl -i -X POST http://localhost:8091/pedidos \
  -H 'Content-Type: application/json' \
```

```
  -d '{"pedidoId":"pedido-4821"}'
```

Observe: HTTP 202 returns **`eventId`**, **`pedidoId`**, **`particion`**, and **`offset`**. The consumer prints a line beginning with **`PEDIDO`**, including the matching identity and storage coordinates. Do not compare your result with a hard-coded partition from someone else's run. Match the result of your own request with the corresponding record observed by the listener.

This capture comes from the verified run on November 6, 2026, with the log trimmed to its message. Your IDs and coordinates may differ:

```
{
  "eventId": "b1f5c34a-e4a1-407a-b0ba-49c02e40cd94",
  "pedidoId": "pedido-4821",
  "particion": 2,
  "offset": 0
}
```

```
PEDIDO eventId=b1f5c34a-e4a1-407a-b0ba-49c02e40cd94 pedidoId=pedido-4821 key=pedido-4821
partition=2 offset=0
```

The subsequent group query reported **`CURRENT-OFFSET=1`**, **`LOG-END-OFFSET=1`**, and **`LAG=0`** for partition 2. The event was at 0; the commit stores 1, the next position.

> **– Solution**
>
> yes, provided Kafka is available and accepts the write. HTTP 202 acknowledges publication, not processing. Producer and consumer meet through the log. The automated test additionally checks that the group's committed offset advances beyond the processed record. Seeing the log line and seeing committed progress are observations of different stages, which is why both are useful.

Experiment 2: Stop the Reader

Predict: what changes when only the reader stops? Who will recover the order published during its absence?

Try: press **`Ctrl+C`** in the consumer terminal. Leave Kafka and the producer running. Publish **`pedido-4822`** using the earlier request with a different ID, then restart the consumer.

Observe: the pending order is processed when the group resumes. You may see other pending records if you made additional requests. The producer does not need to track the reader's availability or manually resend the event when it comes back.

- Solution

retained records and committed progress make recovery possible. The experiment does not cover a reader that remains down beyond the retention window. If the data has been removed, you need another source of recovery or a business decision that accepts that history is no longer available.

Experiment 3: Share Partitions Between Two Consumers

Predict: with two members and three partitions, does each member receive every order or do they divide the work?

Try: keep the first consumer running and start the same command in another terminal. Wait for the rebalance, then inspect membership:

```
docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --describe --group avisos-demo --members --verbose
```

Observe: both members have assignments, with no overlapping ownership within the group. Publish several orders with different keys. One key should not spread across all three partitions. Also, three arbitrarily chosen keys do not necessarily hash into three different partitions, so do not use that assumption to judge the result.

- Solution

ownership is divided by partition. A record is processed by the member responsible for its partition while another member can work elsewhere. The automated test publishes to known partitions to verify sharing without depending on accidental hash distribution among our example order IDs.

Experiment 4: Add an Independent Purpose

Predict: does a consumer in **`analitica-demo`** compete with members of **`avisos-demo`**? Can it read an order that notifications already committed?

Try: start another reader:

```
java -jar target/kafka-pedidos-1.0.0.jar \
  --spring.profiles.active=consumidor \
  --spring.main.web-application-type=none \
  --spring.kafka.consumer.group-id=analitica-demo
```

Observe: if this group is new, **`earliest`** lets it read retained history from the beginning. Publish another order. The notification group and analytics group can both process it. Members inside the notification group still divide their own work among themselves.

> – **Solution**
>
> we did not duplicate the topic for each purpose, and committing did not erase the records. We created independent progress and interpretation. A real analytics application would execute different code; using the same listener here keeps the experiment focused on one changed variable.

Experiment 5: Replay Retained Records

Predict: does restarting **`analitica-demo`** replay the same history as starting a new group? Which position takes priority when a commit already exists?

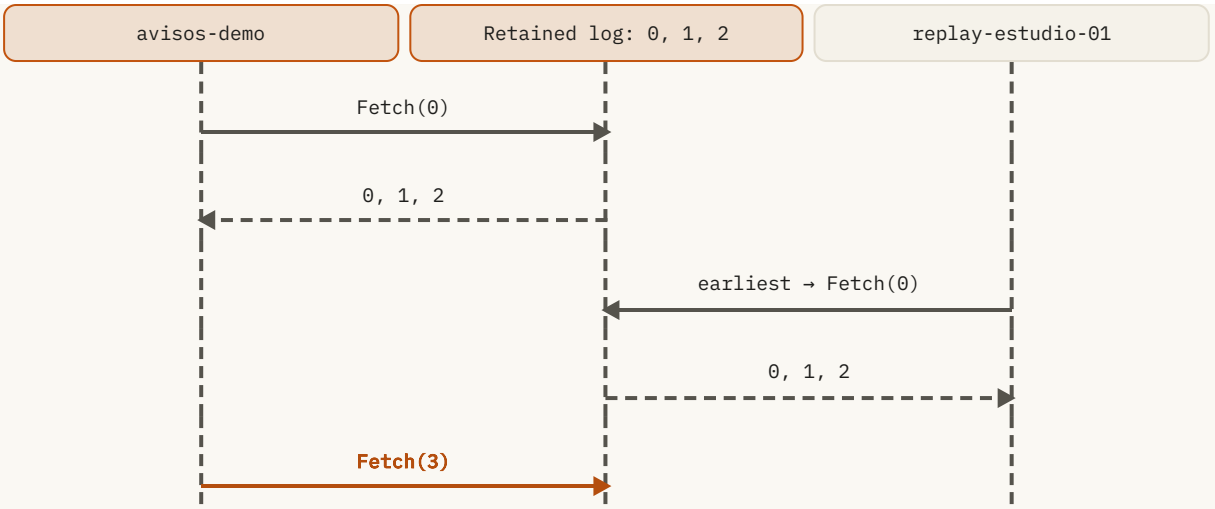
Try: first restart **`analitica-demo`**, then choose a fresh, recognizable group:

```
java -jar target/kafka-pedidos-1.0.0.jar \
  --spring.profiles.active=consumidor \
  --spring.main.web-application-type=none \
  --spring.kafka.consumer.group-id=replay-estudio-01
```

Observe: the new group traverses records that are still retained. If you already used that name, choose another, such as **`replay-estudio-02`**. The tests explicitly distinguish resuming a group with committed offsets from starting a new group with **`earliest`**.

> – **Solution**
>
> replay means another reading, not permission to repeat external consequences without control. Rebuilding analytics into a new view is one useful approach. Replaying emails, payments, or remote mutations requires an effect-protection strategy and a reconstruction mode that understands which actions should not happen again.

> **Replay: another group, the same retained history**



Notifications starts at 0 and processes the three available records.

The cards remain. This group commits its next position: 3.

The replay group is new and has no commits; earliest selects the available beginning.

The second reader sees the same history. It does not move notification commits.

On return, notifications continues at 3. earliest does not reset valid group progress. Neither reader recovers records already removed by retention.

Experiment 6: Keep the Key Stable

Predict: when you repeat a key, what happens to the partition, offsets, and **`eventId`**?

Try: repeat the **`POST`** with **`pedido-4821`** three times, waiting for each response before sending the next. Compare the responses with their corresponding consumer log lines.

Observe: the partition remains the same and offsets increase. Other keys may have records interleaved between them. That does not violate this order's local sequence. If you submit requests concurrently, do not treat click order or HTTP response order as a business sequence your producer never explicitly established.

> – **Solution**
>
> the experiment concerns one producer, a stable key, and a fixed partition count. Consumer behavior matters too. A reader can receive records in order but launch their effects into concurrent tasks that finish out of order. Storage order and effect-completion order need separate answers.

Inspect, Test, and Close the Lab

Inspect the topic and the group's progress with the included tools:

```
docker compose exec kafka /opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server kafka:19092 --describe --topic pedidos.creados

docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --describe --group avisos-demo
```

CURRENT-OFFSET shows committed progress, while **LOG-END-OFFSET** shows the end position reported by the tool. The difference helps you reason about backlog, with the broader model's caveats about transactions and internal records. It is not a duration in seconds. Two thousand records could represent a second of one workload and an hour of another.

Run **./mvnw test** from the lab with the local broker available. To repeat the isolated authoring run, return to the repository root:

```
pnpm spring:ejemplos run --proyecto kafka
pnpm spring:ejemplos check --proyecto kafka
```

The runner prepares dependencies, then tests without Internet egress using a real broker on an internal network. It records evidence tied to the current source. To end the local session, stop the Java applications with **Ctrl+C** and run **docker compose down**. The volume remains. Use **docker compose down -v** only when you deliberately want to remove **all lab data and group offsets** and begin with an empty history next time.

---

9. What Happens When Something Fails

The Acknowledgement That Never Arrived

Our application creates a new **eventId** for each HTTP request. If Kafka accepts a record but its response arrives too late, retrying **POST /pedidos** can publish another event with the same **pedidoId** and a different **eventId**. Producer idempotence handles duplicates from certain protocol retries; it does not recognize two business requests as one purchase.

That is why HTTP 503 carries **publication unconfirmed** and warns that retrying may duplicate the event. It does not claim the order was processed or that repeating the request is safe. For a real API, I would design a persistent idempotency key and a policy for reconciling uncertain outcomes; that state is outside this lab.

Processing and Committing Are Different Operations

Imagine the consumer sends an email and exits before committing its offset. After restart, it may receive the same record again. The first email has already left. Commit before sending, and the reverse failure

can omit the email. Reordering the operations changes the risk; it does not turn independent systems into one atomic transaction.

Policy	Vulnerable point	Possible consequence
Commit before the effect	Failure after commit	Omit the effect
Apply the effect, then commit	Failure before commit	Repeat the effect
Coordinate progress and effect within one transactional boundary	Requires suitable support and design	Stronger guarantees within that boundary

“At least once” means your processing must tolerate the possibility of repetition. It does not mean every record always repeats, or that unlimited duplicate side effects are acceptable. In our lab, a repeated log line is harmless. Replacing that line with a charge would dramatically change what the surrounding design must protect.

> – **Going deeper**
>
> consider a processor reading **`pedidos.creados`**, writing a result to **`pedidos.enriquecidos`**, and advancing its input offset. A Kafka transaction can commit **the output and input progress together**. If it aborts, a **`read_committed`** reader does not see that output, and the processor can read the input again. Kafka Streams can coordinate this model through its exactly-once processing configuration. An email sent over HTTP is outside that transaction: it might already have been sent before the abort. Always name the inputs, outputs, and readers covered by the guarantee. Consult the transaction protocol before extending it to other systems.

A Record That Will Never Succeed Unchanged

Malformed JSON can fail regardless of how many times you retry. A remote service outage may recover after a delay. Treating both failures identically can either block a partition indefinitely or discard recoverable work too early. Classify the error before deciding how many attempts are useful and how long a partition may wait.

The example uses **`CommonContainerStoppingErrorHandler`** to stop the listener on a processing error. This keeps the problem visible and requires intervention. A production design might choose bounded retries and an error topic with enough context to investigate. That topic still needs ownership, retention, permissions, and a recovery procedure. Moving a failure somewhere else does not make it disappear.

Continuing after moving an event to an error topic can change business order. If an order-paid event fails and an order-shipped event proceeds, the consequences may no longer follow the sequence the domain expects. “We have a DLQ” is not a complete answer. Decide whether work may continue, how pending state is represented, and how a corrected event can safely reenter the flow. The Spring Kafka error reference provides mechanisms; your domain defines their policy.

Recover the Lab without Erasing Its History

This optional procedure comes after the six experiments. Keep Kafka, the HTTP producer, and at least one **`avisos-demo`** consumer running, with terminals in **`examples/spring/kafka`**. If you already closed the lab, repeat the first-event startup before continuing.

Inject an invalid record with a known key, then publish a valid order with the same key. With the partition count unchanged, both go to the same partition.

```
docker compose exec -T kafka /opt/kafka/bin/kafka-console-producer.sh \
  --bootstrap-server kafka:19092 --topic pedidos.creados \
  --producer-property acks=all --property parse.key=true --property key.separator=:
<<'EOF'
pedido-error:no-es-json
EOF

curl -sS -X POST http://localhost:8091/pedidos \
  -H 'Content-Type: application/json' -d '{"pedidoId":"pedido-error"}'
```

The listener logs **`PEDIDO_INVALIDO`** with the topic, key, partition, and offset before stopping. Record those coordinates. Stop **every consumer in `avisos-demo`**, including processes that remain alive after their listener stops. Other partitions assigned to that container stop being served too: the error handler stops the listener, rather than only the failed record.

First preserve the record and incident context. In our documented run, the invalid record was at partition 0, offset 0. **Replace both numbers with the ones from your log** before copying the command. This terminal format fits our single-line keys and values; a general recovery tool must also preserve any headers and metadata required by your contract, without relying on text separators.

```
PARTICION=0
OFFSET=0
SIGUIENTE=$((OFFSET + 1))

docker compose exec -T kafka /opt/kafka/bin/kafka-console-consumer.sh \
  --bootstrap-server kafka:19092 --topic pedidos.creados \
  --partition "$PARTICION" --offset "$OFFSET" --max-messages 1 --timeout-ms 5000 \
  --formatter-property print.key=true --formatter-property key.separator=: > registro-
problematico.txt

printf 'topic=pedidos.creados partition=%s offset=%s\n' \
  "$PARTICION" "$OFFSET" > incidente.txt

docker compose exec -T kafka /opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server kafka:19092 --create --if-not-exists \
  --topic pedidos.errores --partitions 1 --replication-factor 1
```

```
docker compose exec -T kafka /opt/kafka/bin/kafka-console-producer.sh \
  --bootstrap-server kafka:19092 --topic pedidos.errores \
  --producer-property acks=all --property parse.key=true --property key.separator=: \
  < registro-problematico.txt
```

Verify the console producer exits successfully and **`pedidos.errores`** contains that key and value. Inspect it with **`kafka-console-consumer.sh`**, changing the topic and using **`--from-beginning --max-messages 1 --timeout-ms 5000`**. If previous incidents exist, locate the new record rather than assuming it is first. Keep **`incidente.txt`** with the review outcome. Do not advance the group until the copy is acknowledged.

Next inspect the group: it must be **`Empty`**, with zero members. The dry run should propose **only that partition**, advancing to the offset immediately after the failed record. Review the output before running **`--execute`**: skipping a record is an explicit recovery decision, not a repair to the original event.

```
docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --describe --group avisos-demo --state

docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --group avisos-demo \
  --topic "pedidos.creados:$PARTICION" --reset-offsets \
  --to-offset "$SIGUIENTE" --dry-run

docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --group avisos-demo \
  --topic "pedidos.creados:$PARTICION" --reset-offsets \
  --to-offset "$SIGUIENTE" --execute
```

Restart a consumer in **`avisos-demo`**. The valid order behind the failure should appear and progress should advance. Another group starting from the beginning will still find the invalid JSON: we neither erased nor repaired history. Other groups' commits remain independent. A real incident also requires investigation, correction, and an agreed way to reintroduce the work; quarantine alone does not close it.

Rebalances and Slow Consumers

A group can reassign partitions when membership or subscriptions change. That process is a **rebalance**. A member that owned a partition may lose it, and another member resumes from committed progress. Do not turn temporary ownership into a permanent assumption in some unrelated background worker.

Processing time affects group health. Slow work between polls can exceed limits and trigger reassignments. Increasing every timeout should not be the first response. Measure the slow operation, consider batch size, and design concurrency while preserving partition ownership. The rebalance protocol has evolved, so check your client's version and configuration before borrowing advice from older material.

A consumer can be connected while falling farther behind every minute. If one hundred records arrive per second and it processes eighty, a restart does not create the missing capacity. Reduce the work, add

useful parallelism, or redistribute the load. If almost all traffic uses one key, adding partitions does not automatically spread that entity across many readers while preserving its local order.

Writing to Two Places

Adding a database introduces another failure window. Save the order and then publish: the process can fail between them, leaving an order with no event. Publish first and then fail to save: now the event may describe an order that was never recorded. Reversing the order does not eliminate the gap.

A **transactional outbox** stores the business change and publication intent in one local transaction. A relay or CDC mechanism later sends pending events. That sending can repeat, so readers still need an idempotency strategy. The Debezium outbox event router is one implementation option. The lab does not need it because it has no database write, but recognizing this boundary prevents a small teaching example from quietly becoming an incomplete production design.

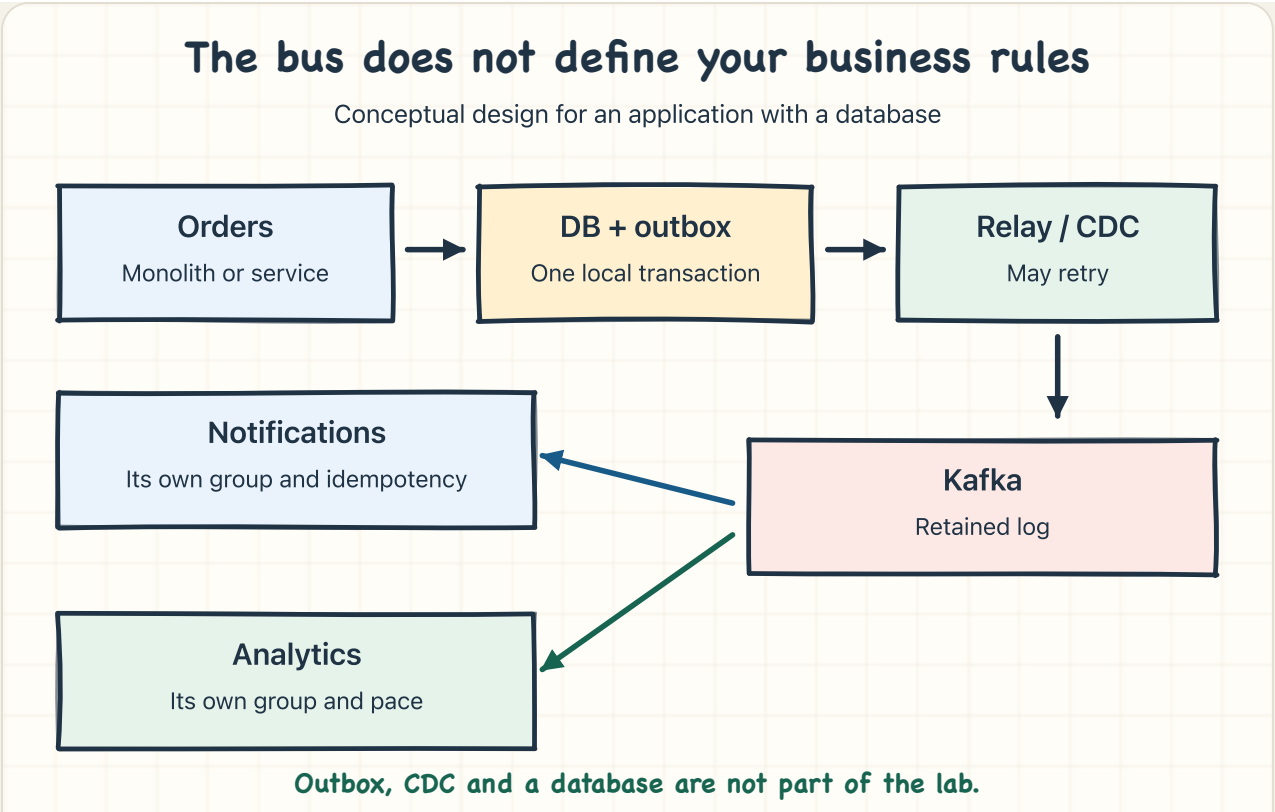
---

10. from the Lab to an Architectural Decision

Where Kafka Fits

A monolith can own the order process and publish facts for outside applications. Microservices can use Kafka across context boundaries. A data platform can integrate sources and build continuous results. These are different uses of a retained stream, not a requirement to divide every application into small services.

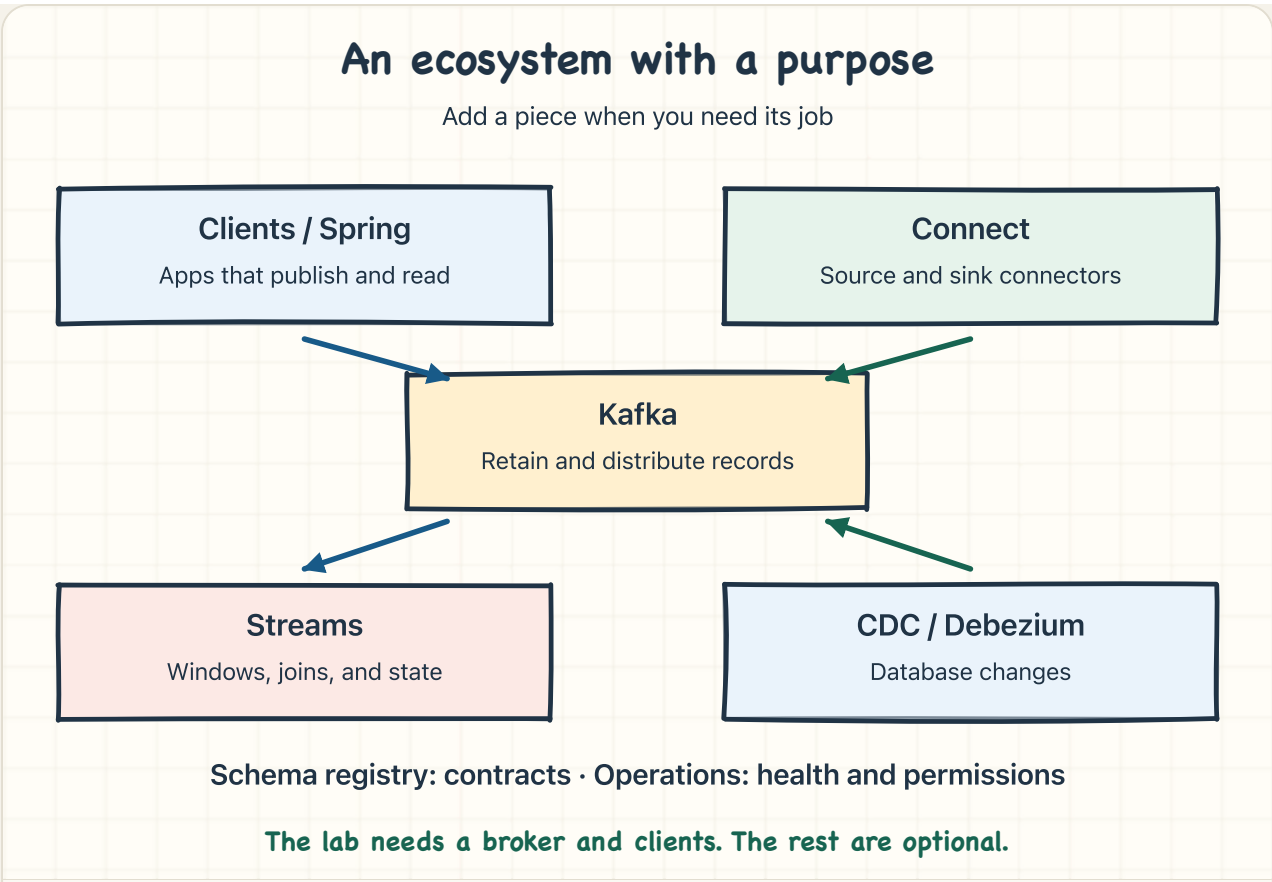
DDD can help name facts and establish which context has authority over them. Hexagonal architecture can keep the Kafka client in an adapter so the domain does not depend on its classes. CQRS separates read and write models. Event Sourcing retains changes as the source of state. None of those patterns automatically appears when a program publishes JSON, and none is required to start the broker.



One possible evolution with a database. Outbox and CDC are additional choices, outside this lab.

The Ecosystem: Add Tools for a Specific Job

This lab works with the broker and clients alone. Extra tools address specific needs; installing all of them does not turn an application into a better architecture.



Clients are enough for our example. Connect, Streams, CDC, and schema registries do different jobs.

Piece	Concrete job	Store example
Clients and Spring Kafka	Publish and consume from your code	Our endpoint and listener
Kafka Connect	Run source connectors into Kafka and sink connectors out of Kafka	Deliver records to a search engine without another custom listener
Kafka Streams	Java library for transformations and stateful processing	Group orders and count sales per store in five-minute windows
CDC, such as Debezium	Capture changes from a database log	Publish table changes or relay an outbox
Schema registry	Version contracts and enforce compatibility under its configured rules	Prevent removing a field that older readers still require
Operations tools	Explore topics, measure health, and manage the cluster	Investigate lag, permissions, or a partition without a leader

Connect has its own runtime and tasks. A source connector reads another system; a sink delivers to one. **Streams** runs inside your application: it is neither another broker nor a SQL engine required for consumption. Its windows, joins, and state stores handle calculations that a simple `println` does not

need. Both belong to Apache Kafka; see the official integration overview and processing concepts for their boundaries.

CDC means **Change Data Capture**. Capturing “a row changed” does not automatically define a domain fact such as **`PedidoCreado`**: you need a contract and a decision about which change represents it. An outbox expresses that intent inside the business transaction; CDC can transport it. Debezium documents that change capture as a separate project, as are schema registry products and many administration interfaces. Kafka does not itself validate our JSON against a business schema; clients, tools, and agreed policies carry that responsibility.

Contracts Across Deployments

Consumers do not necessarily update at the same time as producers. Adding an optional field and renaming a required field have different consequences. Changing what **`creadoEn`** means can break a calculation even if every JSON document remains syntactically valid. Compatibility has both structural and semantic dimensions.

Assign ownership of each topic and a way to announce changes. Document valid examples, identity, time conventions, optionality, and personal data. A schema registry can automate some compatibility checks for formats such as Avro, Protobuf, or JSON Schema; it is not a mandatory part of the broker and does not replace agreement between teams. Confluent Schema Registry compatibility rules provide one concrete example of these external policies.

Keep old readers and old records in mind. A replay may encounter several generations of producer output. If a deserializer only understands today's format, live traffic can work while reconstruction fails halfway through history. Test with genuinely older event forms, not only the happy-path sample emitted by the latest application build.

Security, Capacity, and Operational Signals

The lab uses PLAINTEXT on a local interface. A real environment needs explicit decisions about encryption, authentication, authorization, and network access to listeners. The security overview and ACL reference cover different responsibilities. Being able to connect does not mean a client should be able to read every topic or use every group.

For a first storage estimate, multiply incoming bytes per second by the retention window and replication factor. Then account for compression, indexes, metadata, headroom, and cleanup behavior. Treat that as an estimate rather than a precise reservation. For recovery, check whether readers can process faster than the live incoming rate while clearing backlog. Without spare capacity, they cannot catch up.

Useful signals include publication latency and errors, lag by group and partition, processing rate, the age of pending work, rebalances, disk usage, and replica health. The monitoring catalog helps map those concerns to concrete metrics. A lag alert without workload context can be noisy. An alert tied to a product's maximum acceptable delay makes the reason for intervention clearer.

Five Exercises to Test Your Model

1. Two purposes. Notifications and analytics must both receive every purchase. Would you give them the same `group.id`? Explain the outcome if you did.

- **Solution**

I would use separate groups. A shared group divides partition ownership and work; it does not give every application a copy of every record. Separate groups preserve independent progress and let one purpose rebuild without moving the other's position.

2. The rewind that did not happen. You restart a group with `earliest`, but it does not reread old records. What should you inspect before blaming retention?

- **Solution**

check whether that group has valid committed offsets. `earliest` does not reset them. A new group is a simple replay experiment; changing a real group's progress requires a deliberate administrative procedure, with consumers stopped when the operation requires it.

3. The duplicate email. The email was sent, the process exited before its commit, and the email was sent again after restart. Which guarantee was missing?

- **Solution**

the external effect needed protection against repeated delivery. That requires stable identity and a strategy that coordinates deduplication and effect according to the destination's capabilities. Producer idempotency and a later offset commit do not make an external email atomic with Kafka progress.

4. The fourth consumer. One topic has three partitions and four members in the same group. Does the fourth process guarantee more processing capacity?

- **Solution**

no. The conventional model used here has only three partitions to distribute. A hot key can further concentrate work on one partition. I would inspect distribution and processing time before adding processes.

5. The order with no event. An application saves an order and exits before publication. Does publishing first fix the design?

> – **Solution**
>
> it changes the failure window but does not remove it. I would consider an outbox written with the business change in one local transaction and a recoverable relay. Consumers would still need idempotency because outbox publication can repeat.

A Glossary to Return to

Term	The question it answers
Event	What fact are we communicating?
Topic	Which named flow does it belong to?
Key	How do we relate and partition its records?
Partition	Which ordered sequence stores it?
Offset	Where is it within that sequence?
Broker	Which process serves and stores partitions?
KRaft controller	Who coordinates cluster metadata?
Group	Which readers share assignment and progress?
Commit	Where should the group resume?
Retention	What history remains available under the policy?
Replay	Which available records do we want to traverse again?
Lag	How far is progress behind the end of the log?
ISR	Which replicas are considered in sync?
Idempotency	What prevents repetition from multiplying an effect?

The Discussion I Would Take Back to the Team

Question	What still needs designing if you cannot answer
Do we need independent readers and replay?	The reason to choose a retained log
Which order does the business require?	Keys, publication, and reader concurrency
How far behind may a group fall?	Retention, capacity, and alerts
What happens with duplicates?	Identity and effect protection
Who owns the contract?	Responsibility and evolution
Who restores a broken flow?	Operations and incident procedures

If you can answer these, your Kafka discussion has a firmer foundation than a list of popular technologies. You can adopt it, trial an alternative, or postpone it for a specific reason. That judgment is part of learning the tool too.

Now look at your own application: which fact needs sharing, who should be able to read it again, and what would those two answers lead you to choose today?