



Event Streams and Kafka

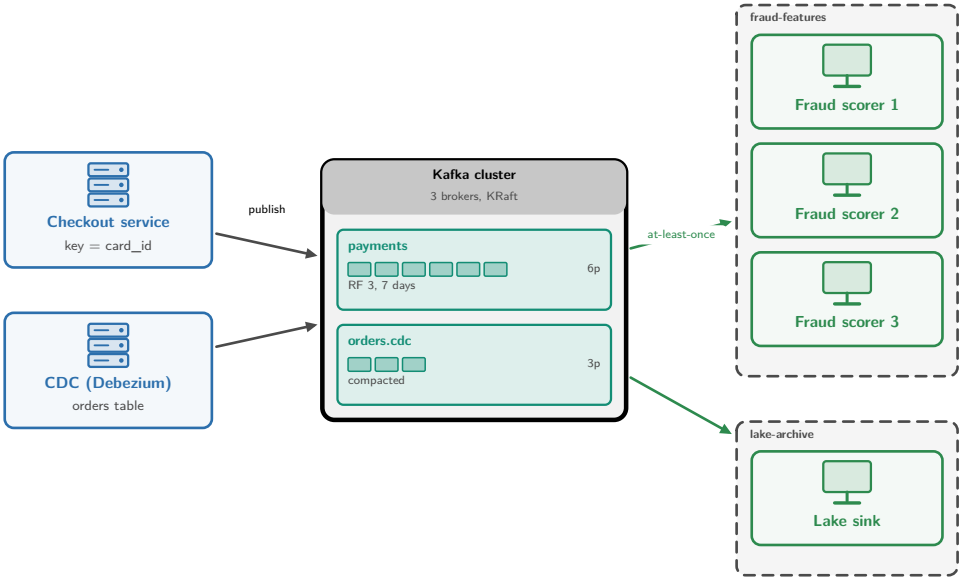
Streaming and real-time ML | Topics, partitions, keys and ordering, consumer groups and offsets, retention and compaction, replication and acks, schema registry and compatibility; commands, configs and sizing (Apache Kafka 4.x, KRaft only, as of 2026)

#Kafka #EventStreams #Partitions #ConsumerGroups #SchemaRegistry

The log, partitions and keys

An accountant's ledger books		A Kafka topic
A set of ledger books for one subject	≈	A topic with N partitions
Lines are only ever appended, never erased	≈	Append-only records, immutable
Line number in one book	≈	Offset in one partition
All entries of one client go in the same book	≈	Same key → same partition → ordered
Each auditor keeps a bookmark per book	≈	Each consumer group commits an offset per partition
Old books are archived after 7 years	≈	retention.ms deletes old segments
WHERE THE ANALOGY BREAKS A ledger has one reader at a time; a Kafka partition can be read by any number of groups, each with its own bookmark, and old lines vanish after retention even if nobody read them.		

Kafka as a set of numbered ledgers



One topic, six partitions, two independent consumer groups; within a group each partition has one owner

Rule	Consequence for ML pipelines
Order is guaranteed only inside a partition	per-user sequence features need key = user_id
Partition = murmur2(key) mod N (positive hash)	adding partitions remaps keys: plan N up front
Null key: sticky partitioner fills one batch, then moves	no ordering; fine for independent events
One partition → at most one consumer per group	max parallelism of a group = number of partitions
Consumers > partitions → extra consumers idle	scale partitions before scaling pods
A hot key overloads one partition	salt the key (key#0..7) and re-aggregate downstream

Ordering and parallelism rules

Partitions needed for a target throughput

FORMULA $N = \max(T / P, T / C)$

WHERE

- N = partitions needed (round up, add headroom) (partitions)
- T = target throughput of the topic at peak (MB/s)
- P = measured producer throughput into one partition (MB/s)
- C = measured consumer throughput from one partition (your processing speed) (MB/s)

WORKED

T = 60
P = 10
C = 5
N = max(60 / 10, 60 / 5)
N = 12 partitions

RESULT N = 12 partitions

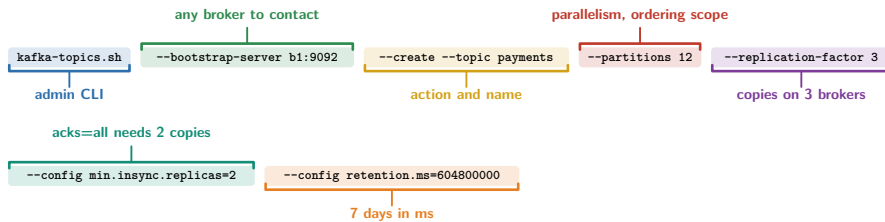
READ IT 12 partitions; round up to a number with many divisors (12, 24) and plan for 2x growth, since adding partitions later remaps keys.

MISTAKES

- C is your slowest consumer (feature computation, model call), not the client library benchmark
- Too many partitions: longer leader elections, more open files, more memory per consumer
- Parallelism also caps at partitions: 12 partitions → at most 12 consumers per group

Sizing the partition count (rule of thumb from measured per-partition throughput)

Consumer groups, offsets and commands



Creating a production topic, part by part

<code>kafka-topics.sh --bootstrap-server b1:9092 --describe --topic payments</code>	Partitions, leaders, replicas, ISR; under-replicated partitions show ISR < replicas
<code>kafka-consumer-groups.sh --bootstrap-server b1:9092 --describe --group fraud-features</code>	Per partition: CURRENT-OFFSET, LOG-END-OFFSET, LAG, owning consumer
<code>kafka-consumer-groups.sh ... --group fraud-features --reset-offsets --topic payments --to-datetime 2026-10-01T00:00:00.000 --execute</code>	Replay from a time (group must be stopped); --dry-run first; also --to-earliest, --shift-by -1000
<code>kafka-configs.sh ... --alter --entity-type topics --entity-name payments --add-config retention.ms=1209600000</code>	Change a topic config live (here 14 days)
<code>kafka-console-consumer.sh ... --topic payments --from-beginning --property print.key=true --max-messages 5</code>	Peek at records with their keys
<code>kafka-get-offsets.sh ... --topic payments --time -1</code>	Log-end offset per partition (-2 = earliest)

```
kafka-consumer-groups --describe (abridged): lag per partition
GROUP      TOPIC      PART  CURRENT  LOG-END  LAG
fraud-features payments  0      918200   918260   60
fraud-features payments  1      904115   904130   15
fraud-features payments  2      611020   702020  91000
# partition 2 is stuck: hot key or a slow / dead consumer
```

Commit after processing — At-least-once: a crash between processing and commit replays the records, so downstream must tolerate duplicates.

Commit before processing — At-most-once: a crash after the commit loses those records; acceptable only for disposable metrics.

Auto-commit (`enable.auto.commit=true`, every 5 s) — Commits offsets returned by poll on a timer: simple, but can commit records not yet processed by async code.

Transactional commit — Offsets committed inside a producer transaction with the output records: exactly-once read-process-write (sheet 07).

What triggers a rebalance	How to make it painless
A consumer joins or leaves	Kafka 4.0+: <code>group.protocol=consumer</code> (KIP-848), incremental and broker-driven
No heartbeat within <code>session.timeout.ms</code> (45 s default)	Older clients: cooperative-sticky assignor
Processing a batch takes longer than <code>max.poll.interval.ms</code> (5 min default)	Static membership: <code>group.instance.id</code> so restarts do not rebalance
Partitions added to a subscribed topic	Lower <code>max.poll.records</code> if processing is slow

⚡ Durability, retention and configs

client.properties	
1 # producer	
2 acks=all	1 wait for all in-sync replicas; default since Kafka 3.0
3 enable.idempotence=true	
4 linger.ms=10	2 retries cannot duplicate; default since 3.0
5 batch.size=65536	
6 compression.type=zstd	3 batch up to 10 ms for throughput; zstd cuts bytes
7 # consumer	
8 group.id=fraud-features	
9 auto.offset.reset=earliest	4 new group starts from the oldest data (default: latest)
10 enable.auto.commit=false	
11 isolation.level=read_committed	5 commit yourself after processing: at-least-once
12 max.poll.records=500	
13 group.protocol=consumer	6 skip aborted transactional records (default read_uncommitted)
	7 KIP-848 rebalance protocol, GA in Kafka 4.0

Producer and consumer settings for an ML feature pipeline

Setting	Default	Use for ML
<code>cleanup.policy=delete + retention.ms</code>	7 days	event topics; set long enough to replay a backfill or a retrain
<code>retention.bytes (per partition)</code>	-1 (no limit)	cap disk when volume spikes
<code>cleanup.policy=compact</code>	-	latest value per key: user profile, feature snapshot, state changelogs
<code>compact.delete</code>	-	latest per key but nothing older than retention
Tiered storage (KIP-405, production-ready in 3.9)	off	months of history on object storage for replay without big broker disks
Retention and cleanup policies		

Quantity	Formula	Example
Storage	$S = r \times t_{ret} \times RF$	20 MB/s $\times$ 604,800 s $\times$ 3 $\approx$ 36.3 TB
Replay time	$t_{replay} = \frac{L}{C_{tot}}$	lag of 7.2 GB at 40 MB/s total = 180 s

where S = disk used across the cluster (MB); r = average write rate into the topic (MB/s); t_{ret} = retention time (7 days = 604,800 s) (s); RF = replication factor (copies); L = backlog to reprocess (MB); C_{tot} = total consumption rate of the group (MB/s)
Disk needed for a topic, worked: 20 MB/s, 7 days, RF 3 $\rightarrow$ 20 $\times$ 604,800 $\times$ 3 = 36.3 TB before compression

⚠

Durability needs three settings together
replication.factor=3, min.insync.replicas=2 and acks=all: a write is acknowledged only when at least 2 copies have it, so one broker can fail with no loss. acks=1 or min.insync.replicas=1 can lose acknowledged writes on a leader crash.

Schemas and the managed landscape			
Mode	New schema can read	Safe change	Deploy first
BACKWARD	data written with the previous schema	delete a field, add an optional field with a default	consumers
FORWARD	old schema can read new data	add a field, delete an optional field	producers
FULL	both directions	add or delete optional fields with defaults only	either
*_TRANSITIVE	same, against all earlier versions	same, checked against every version	-
NONE	no check	anything (breaks consumers silently)	-
Schema registry compatibility modes (Confluent Schema Registry and compatible ones; default BACKWARD)			

🔗

Wire format and subjects
Serialized records start with a magic byte 0 and a 4-byte schema id, then the Avro or Protobuf payload; consumers fetch the schema by id. Default subject name is <topic>-value (Topic-NameStrategy). Register schemas in CI and set compatibility per subject so a feature-breaking change fails the build, not the model.

Service	Model	Notes for ML teams
Apache Kafka 4.x (self-managed)	brokers + KRaft	full control; you run upgrades, rebalancing, monitoring
Amazon MSK	managed Kafka: provisioned, Express brokers, Serverless	IAM auth; MSK Connect for connectors
Confluent Cloud	managed Kafka (Kora engine)	schema registry, connectors, managed Flink SQL; owns WarpStream (2024)
Amazon Kinesis Data Streams	shards, not Kafka API	per shard about 1 MB/s or 1,000 records/s in, 2 MB/s out; on-demand mode
Google Pub/Sub; Managed Service for Apache Kafka	Pub/Sub: per-message ack; Kafka: managed	Pub/Sub ordering keys optional; Dataflow (Beam) processes both
Azure Event Hubs	partitions, Kafka-compatible endpoint	Kafka clients connect without code changes
Redpanda, WarpStream	Kafka API compatible	Redpanda: C++, no JVM; WarpStream: diskless on object storage
Kafka-compatible and managed event streams (as of 2026; check current quotas)		

✓ Myths and recap

MYTH OR FACT?
"Kafka keeps the order of all events in a topic"

MYTH: Only within one partition
Choose the key so events that must stay ordered share a partition.

MYTH OR FACT?
"More consumers always means faster consumption"

MYTH: Parallelism stops at the partition count
A 6-partition topic keeps a 10-consumer group's 4 extra members idle.

MYTH OR FACT?
"A consumed message is deleted"

MYTH: Retention, not reading, deletes data
Every group keeps its own offset; replay is a reset of offsets.

MYTH OR FACT?
"acks=all alone guarantees no loss"

MYTH: It needs min.insync.replicas $\geq$ 2
With min ISR 1, 'all' may mean only the leader.

Myth or fact? The verdict and the reason are printed under each card

☐ **Key choice = ordering + load balance**
same entity $\rightarrow$ same partition; watch hot keys

☐ **Partitions cap consumer parallelism**
size with $N = \max(T/P, T/C)$, plan growth

☐ **Commit after processing**
at-least-once by default; make sinks idempotent

☐ **RF 3, min ISR 2, acks=all**
the durability triple

☐ **Retention long enough to replay**
your backfill and retraining depend on it

☐ **Schemas enforced in CI**
BACKWARD by default: upgrade consumers first

Tick only what you could explain to a colleague

Q1 **Where is ordering guaranteed in Kafka?**
A Inside one partition only; records with the same key go to the same partition.

Q2 **Max useful consumers in one group for a 12-partition topic?**
A 12; any extra consumer stays idle.

🌐 Tadata | tadata.fr | contact@tadata.codes

Q3 What is consumer lag?

A Log-end offset minus committed offset per partition: records still to process.

Q4 Why plan the partition count up front?

A Adding partitions changes $\text{murmur2}(\text{key}) \bmod N$, so keys move and per-key order breaks across the change.

Q5 What does log compaction keep?

A At least the latest record for each key; a tombstone (null value) deletes the key.

Q6 BACKWARD compatibility: what can change and who deploys first?

A Delete fields or add optional fields with defaults; upgrade consumers first.

Q7 What replaced ZooKeeper in Kafka 4.0?

A KRaft, Kafka's built-in Raft metadata quorum; ZooKeeper mode was removed.

Q8 Two causes of a rebalance storm?

A Processing slower than `max.poll.interval.ms`, and pods restarting without static membership.

Q9 How do you replay a day of events for a backfill?

A Stop the group, run `kafka-consumer-groups.sh --reset-offsets --to-datetime ... --execute`, restart; needs retention covering that day.

Q10 Kinesis shard write limit?

A About 1 MB/s or 1,000 records/s per shard (check current AWS quotas).