Kafka Client Patterns

Version Kafka 4.4 · kafka-javascript 1.11 · verified 2026-11-02

Keys, acks, idempotence, batching, consumer groups, offset commits, transactions, share groups, Schema Registry, retries and lag — from the client side.

Contents

Mental model 2

Topics, partitions and keys 2

Producer acks and idempotence 3

Batching and compression 4

Consumer groups and rebalancing 5

Offset commits 6

Exactly-once with transactions 8

Share groups vs consumer groups 9

Schemas: Avro, Protobuf, JSON Schema 10

Retries, dead letters and lag 11

Further reading 12

Mental model

Kafka is a replicated, append-only log split into partitions, and consumers are just readers that remember an offset. Every guarantee — ordering, delivery, exactly-once — is scoped to one partition and one consumer group, never to the topic as a whole. So design the key (which partition a record lands in) and the group (who shares the work) first; everything else is tuning.

Topics, partitions and keys

Choice	Consequence
Same key	Same partition, so ordered
No key	Spread across partitions, no order
More partitions	More parallel consumers
Adding partitions later	Keys remap to new partitions

The producer hashes the key to pick a partition, so all records for one user id arrive in order at one consumer. There is no ordering across partitions. Samples use `@confluentinc/kafka-javascript` 1.11 through its KafkaJS-compatible API: it wraps librdkafka and is actively released, while KafkaJS's last release, 2.2.4, dates from February 2023.

```
import pkg from "@confluentinc/kafka-javascript";
const { Kafka } = pkg.KafkaJS;


const kafka = new Kafka({
```

```
  kafkaJS: { brokers: ["localhost:9092"] },
});
const producer = kafka.producer();
await producer.connect();
await producer.send({
  topic: "orders",
  messages: [{ key: "user-42", value: "{}" }],
});
```

> **Gotcha:** increasing a topic's partition count changes which partition each key hashes to. New records for `user-42` can land on a different partition than old ones, so per-key order breaks across the change. Size partitions up front.

Producer acks and idempotence

Setting (JS `kafkaJS`)	Default	Use
`acks`	`-1` (all in-sync replicas)	Keep
`idempotent`	`false`	Set `true`
`transactionalId`	none	Exactly-once (below)

`acks: -1` waits until every in-sync replica has the record, so a leader crash cannot lose an acknowledged write. Retries can still duplicate a record whose ack was lost; idempotence makes the broker drop those duplicates and keeps per-partition order across retries.

```
// kafka: the client from the first sample
const producer = kafka.producer({
  kafkaJS: { acks: -1, idempotent: true },
});
```

> **Gotcha:** the Java producer enables idempotence by default (`enable.idempotence=true`); librdkafka and therefore this JS client default it to `false`. Code ported from Java silently loses the guarantee unless you set it.

Batching and compression

Setting	JS default	Effect
`linger.ms`	5	Wait to fill a batch
`compression`	none	Per batch: gzip, snappy, lz4, zstd
`flush()`	—	Wait for queued sends

Records are batched per partition and sent when a batch fills or `linger.ms` elapses, so compression works on whole batches. Awaiting each `send()` serialises your throughput behind the round trip; the client's docs recommend firing sends and then calling `flush`.

```
import pkg from "@confluentinc/kafka-javascript";
const { Kafka, CompressionTypes } = pkg.KafkaJS;
const kafka = new Kafka({ kafkaJS: {
```

```
  brokers: ["localhost:9092"] } });
const producer = kafka.producer({
  kafkaJS: { compression: CompressionTypes.ZSTD },
  "linger.ms": 20,
});
await producer.connect();
for (const value of ["a", "b", "c"])
  producer.send({ topic: "logs",
    messages: [{ value }] });
await producer.flush({ timeout: 5000 });
```

Consumer groups and rebalancing

Concept	Rule
One partition	One consumer in the group
Consumers > partitions	The extras sit idle
Join or leave	Triggers a rebalance
`group.protocol`	`classic` (default) or `consumer`

Each group gets every record once, split across its members by partition. A rebalance moves partitions between members; under the newer `consumer` protocol the broker computes the assignment incrementally instead of stopping the whole group.

```
// kafka: the client from the first sample
const consumer = kafka.consumer({
```

```
  kafkaJS: { groupId: "billing" },
  "group.protocol": "consumer",
});
await consumer.connect();
await consumer.subscribe({ topics: ["orders"] });
await consumer.run({
  eachMessage: async ({ partition, message }) => {
    console.log(partition, String(message.value));
  },
});
```

> **Gotcha:** processing that outlasts the max poll interval (`rebalanceTimeout`, 300000 ms by default here) gets the consumer evicted. Its partitions move, and another member reprocesses the uncommitted records.

Offset commits

Strategy	Delivery	Risk
Auto-commit (default, 5 s)	At-least-once	Replays up to 5 s
Commit after processing	At-least-once	Duplicates on crash
Commit before processing	At-most-once	Loss on crash

This client auto-commits, every 5 seconds, only offsets whose `eachMessage` ran to completion — so the default is at-least-once and your

handler must tolerate duplicates. Commit manually when you need the commit to follow a side effect such as a database write.

```
// kafka: the client from the first sample
const consumer = kafka.consumer({ kafkaJS: {
  groupId: "billing", autoCommit: false } });
await consumer.connect();
await consumer.subscribe({ topics: ["orders"] });
await consumer.run({
  eachMessage: async (p) => {
    /* write to the database first */
    const { topic, partition, message } = p;
    const n = Number(message.offset) + 1;
    await consumer.commitOffsets([{ topic,
      partition, offset: String(n) }]);
  },
});
```

> **Gotcha:** the committed offset is the next one to read, not the last one processed. Committing `message.offset` replays that record after every restart; commit `offset + 1`.

Exactly-once with transactions

Side	Required setting
Producer	`transactionalId` (implies idempotence)
Consumer	`autoCommit: false`
Downstream readers	`read_committed` isolation

Exactly-once in Kafka covers read-process-write *within Kafka*: the output records and the consumer's offset commit go into one transaction, so both land or neither does. Readers must use `read_committed`: this JS client defaults to it, but the Java consumer defaults to `read_uncommitted`, which also returns records from aborted transactions.

```
// kafka, consumer: as above, autoCommit false
const producer = kafka.producer({
  kafkaJS: { transactionalId: "billing-tx-1" },
});
await producer.connect();
const tx = await producer.transaction();
await tx.send({ topic: "invoices",
  messages: [{ key: "user-42", value: "{}" }] });
await tx.sendOffsets({ consumer, topics: [{
  topic: "orders",
  partitions: [{ partition: 0, offset: "101" }],
```

```
}] });
await tx.commit();  // or tx.abort()
```

> **Warning:** a side effect outside Kafka — an email, an HTTP call, a database row — is not in the transaction. For those you still need an idempotent consumer, keyed on something like the record's offset.

Share groups vs consumer groups

	Consumer group	Share group
Unit of work	Partition	Individual record
Consumers > partitions	Idle	All share the work
Order	Per partition	None
Ack	Offset commit	Per record: accept, release, reject

Share groups (KIP-932, "Queues for Kafka") are production-ready since Kafka 4.2. Records are acquired under a time-limited lock (30 s by default, `share.record.lock.duration.ms`) and delivery attempts are counted, so a poison record stops being redelivered. The JS client's promisified API has no share consumer as of 1.11; use the Java client.

```
bin/kafka-console-share-consumer.sh \
  --bootstrap-server localhost:9092 \
  --topic orders --group billing-queue
bin/kafka-share-groups.sh \
```

```
  --bootstrap-server localhost:9092 \
  --describe --group billing-queue
```

> **Note:** Kafka 4.4 adds a dead-letter queue for share groups (KIP-1191): set `errors.deadletterqueue.topic.name` on the group once the cluster runs `share.version=2`.

Schemas: Avro, Protobuf, JSON Schema

Compatibility	New schema must…	Upgrade first
`BACKWARD` (default)	Read data in the latest schema	Consumers
`FORWARD`	Be readable by the latest schema	Producers
`FULL`	Both	Either
`*_TRANSITIVE`	Hold against all versions	—

Kafka stores bytes; Schema Registry stores the schema and checks each new version against the subject's compatibility rule before producers may use it. The serializer prefixes each record with a schema id. Use `@confluentinc/schemaregistry`, which supports Avro, Protobuf and JSON Schema.

```
import sr from "@confluentinc/schemaregistry";
const { SchemaRegistryClient, AvroSerializer,
  SerdeType } = sr;
const registry = new SchemaRegistryClient({
  baseURLs: ["http://localhost:8081"] });
```

```
const ser = new AvroSerializer(registry,
  SerdeType.VALUE, { useLatestVersion: true });
const value = await ser.serialize("orders",
  { id: "o-1", total: 9.5 });
```

Retries, dead letters and lag

Tool	Purpose
Retry topic (`orders.retry`)	Re-try later without blocking the partition
Dead-letter topic (`orders.dlt`)	Park records that keep failing
`kafka-consumer-groups.sh --describe`	`LAG` per partition

A consumer group cannot skip a bad record and come back to it: the partition is blocked until you commit past it. Catch the failure, send the record with error headers to a retry or dead-letter topic, then commit and move on. Watch lag — log-end offset minus committed offset — per partition; steadily growing lag means consumers are too slow.

```
bin/kafka-consumer-groups.sh \
  --bootstrap-server localhost:9092 \
  --describe --group billing
# TOPIC  PARTITION  CURRENT-OFFSET
#   LOG-END-OFFSET  LAG  CONSUMER-ID ...
```

Further reading

- Apache Kafka — Design: message delivery semantics
- Apache Kafka — Share groups
- KIP-932 — Queues for Kafka
- confluent-kafka-javascript — Introduction
- confluent-kafka-javascript — Migrating from KafkaJS