

Apache Kafka Reference

From first message to production-grade event streaming pipelines.

Last reviewed 29 Sep 2026 · Lab-tested with Apache Kafka 4.3.1 ·

Prerequisites: [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) · [Python for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/)

Related: [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) · [Databricks](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) · [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) · [Apache Flink](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/) · [Beam and Dataflow](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/beam-dataflow/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/beam-dataflow/) · [Streaming SQL](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/streaming-sql/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/streaming-sql/) · [Real-Time Analytics Databases](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/realtime-olap/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/realtime-olap/) · [Azure and Fabric](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Practice: [Lab 04 — Kafka Streaming](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/04-kafka-streaming) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/04-kafka-streaming)

Overview

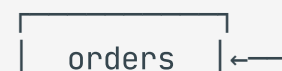
Challenge: In a typical commerce platform, many services react to the same event — billing, inventory, notifications, analytics, fraud detection, and shipping all need to know when an order is placed. Point-to-point integrations or one queue per consumer create a fragile web of dependencies that does not scale.

Solution: Kafka is a durable, **shared log**. The producing service writes each event once, and any number of consumers read it independently, at their own pace.

Without Kafka:

Orders → Billing

With Kafka:

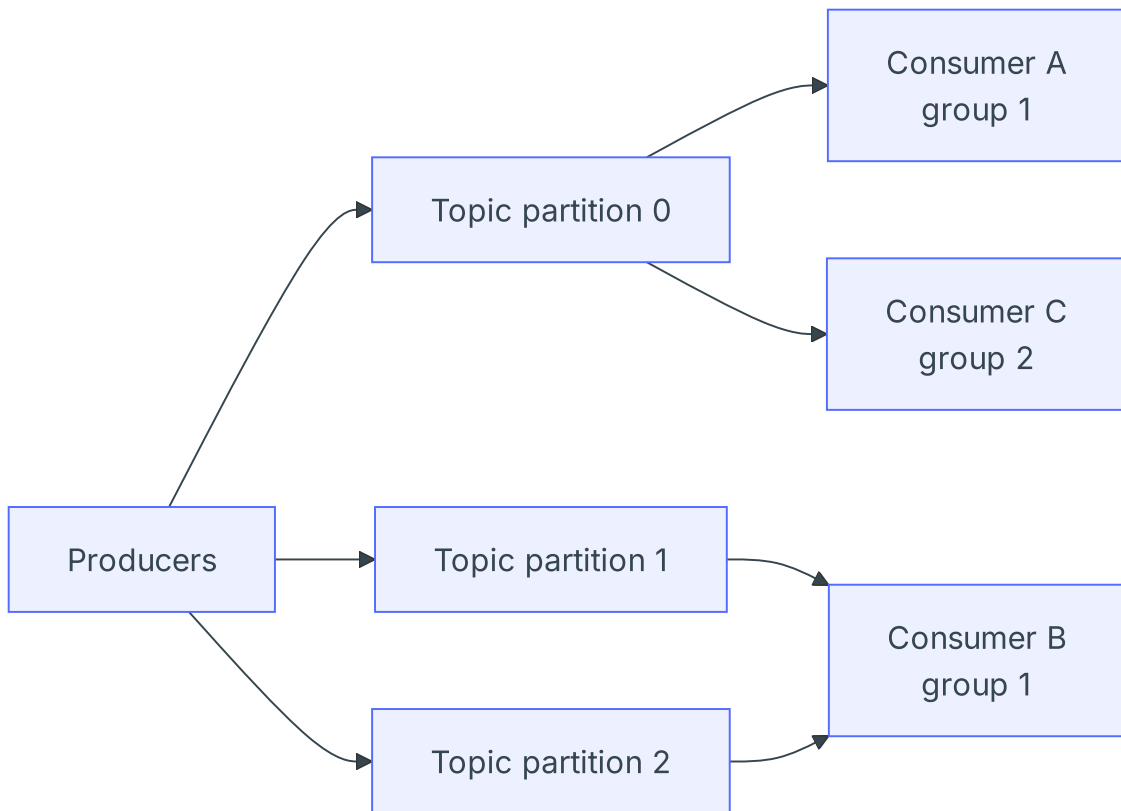


```

Orders service (writes once)
Orders → Inventory    (messy, fragile)  |  topic  |
Orders → Analytics
Orders → Fraud
Orders → Notifications
|-----|-----|-----|
Fraud                               Billing  Inventory  Analytics
                                     (each reads at own pace,
independently)

```

Key property: Kafka retains messages for a configurable period (days to weeks, or indefinitely). Consumers can reprocess history, new consumers can start from the beginning, and a restarted consumer resumes from its last committed position — capabilities that traditional queues, which delete messages once consumed, do not offer.



On this page

Basics - [What is Kafka?](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#what-is-kafka) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#what-is-kafka) - **Core Concepts** (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#core-concepts) - **Topics & Partitions** (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#topics--partitions) - **Producers** (https://s

[arangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#producers](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#producers)) - [Consumers & Consumer Groups](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#consumers--consumer-groups) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#consumers--consumer-groups>)

Intermediate - [Offsets & Delivery Guarantees](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#offsets--delivery-guarantees) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#offsets--delivery-guarantees>) - [Consumer Group Rebalancing](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#consumer-group-rebalancing) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#consumer-group-rebalancing>) - [Kafka CLI](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-cli) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-cli>) - [Schema Registry & Avro](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#schema-registry--avro) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#schema-registry--avro>) - [Kafka with Python](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-with-python) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-with-python>)

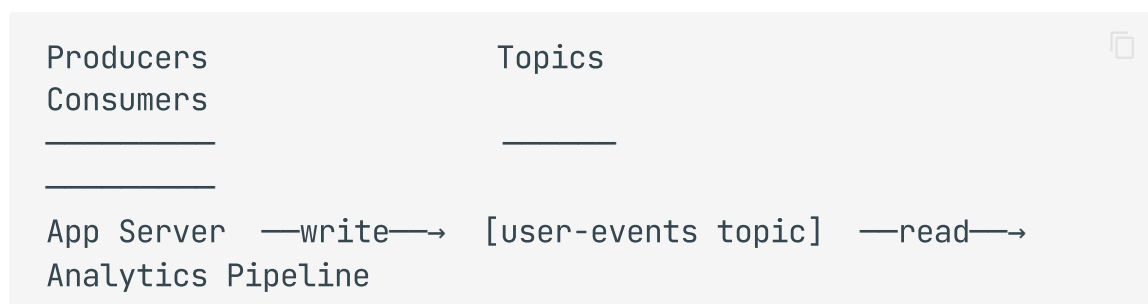
Advanced - [Kafka Connect](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-connect) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-connect>) - [Kafka Streams](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-streams) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-streams>) - [Retention & Compaction](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#retention--compaction) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#retention--compaction>) - [Performance Tuning](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#performance-tuning) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#performance-tuning>) - [Kafka in DE Pipelines](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-in-de-pipelines) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#kafka-in-de-pipelines>)

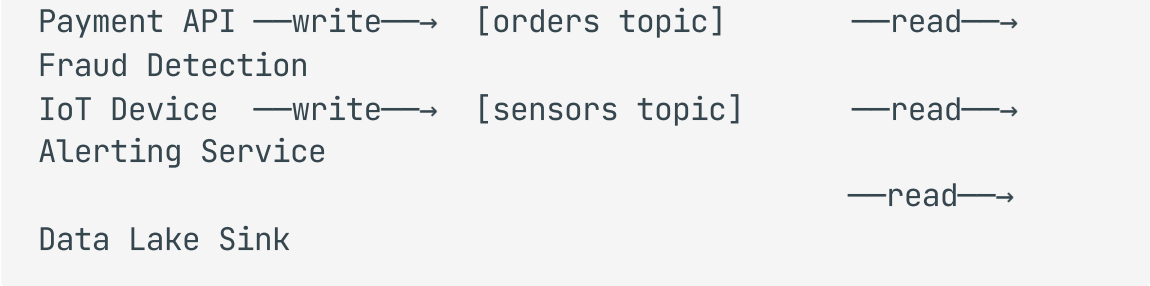
Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#common-pitfalls) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#common-pitfalls>) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#cheat-sheet) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#cheat-sheet>) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#interview-questions) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#interview-questions>) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#further-reading) (<https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/#further-reading>)

What is Kafka?

Kafka is a distributed, fault-tolerant, high-throughput **event log**. It stores a durable, ordered sequence of events that any number of consumers can read at their own pace — independently and repeatedly.

Think of it as a **commit log for events**: producers append events to the end; consumers read from wherever they left off. Nothing is deleted by a consumer reading — events persist until a retention policy removes them.





Kafka vs a message queue (RabbitMQ/SQS):

	Kafka	Message Queue
Message deleted after consume?	No — retained by time/size	Yes
Multiple consumers same message?	Yes	No (one consumer per message)
Replay old messages?	Yes	No
Ordering guarantee	Per-partition	Varies
Throughput	Very high (millions/sec)	Moderate

Core Concepts

Concept	Definition
Event / Message / Record	An immutable unit of data: key, value, timestamp, headers
Topic	A named, ordered, durable log. Like a table, but for events
Partition	A topic split into parallel ordered sublogs; unit of parallelism
Offset	Sequential integer identifying a message's position in a partition
Producer	Application that writes events to a topic
Consumer	Application that reads events from a topic
Consumer Group	Set of consumers sharing the work of reading a topic

Concept	Definition
Broker	A single Kafka server that stores partitions
Cluster	Multiple brokers working together
KRaft (formerly ZooKeeper)	Cluster metadata and controller election. Kafka 4.0 removed ZooKeeper — new clusters run in KRaft mode only
Replication Factor	How many brokers hold a copy of each partition
Leader / Follower	One broker leads each partition; followers replicate it
ISR	In-Sync Replicas — followers caught up to the leader

Topics & Partitions

Topic: order-events (3 partitions, replication factor 2) 

Partition 0: [0: order_placed] [1: order_shipped] [2: order_delivered]

Partition 1: [0: order_placed] [1: order_cancelled]

Partition 2: [0: order_placed] [1: order_shipped]

↑ offset 0 ↑ offset 1

Each partition is an independent, ordered log.
Ordering is guaranteed **WITHIN** a partition, not across partitions.

Partition key

The producer assigns each message a **key**. Kafka hashes the key to determine which partition the message lands in. Messages with the same key always go to the same partition — guaranteeing ordering for that key.

```
# Messages for the same order_id → same partition →
# guaranteed order
producer.send("order-events", key=b"order_123",
value=b"...")
producer.send("order-events", key=b"order_123",
```



```
value=b"...") # same partition

# No key → round-robin across partitions
producer.send("order-events", value=b"...")
```

How many partitions?

More partitions → more parallelism → higher throughput
But: more partitions → more overhead, longer leader election on failure

Rule of thumb:

- Start with partitions = max expected consumers in one group
- Or: target throughput / throughput per partition
- Common range: 6-100 for most use cases
- You can add partitions later but cannot reduce them

Producers

```
from kafka import KafkaProducer
import json

producer = KafkaProducer(
    bootstrap_servers=["kafka:9092"],
    key_serializer=str.encode,
    value_serializer=lambda v: json.dumps(v).encode("utf-8"),
    acks="all",                # wait for all ISR replicas to
    confirm write              # confirm write
    retries=5,
    linger_ms=10,             # batch messages for up to 10ms
    before sending            # before sending
    batch_size=16384,          # max batch size in bytes
)

# Send a message
producer.send(
    topic="order-events",
```

```

        key="order_123",
        value={"order_id": "order_123", "status": "placed",
"amount": 99.50},
    )

    # Flush – wait for all pending messages to be sent
    producer.flush()
    producer.close()

    # With callback – handle success/failure
    def on_send_success(metadata):
        print(f"Sent to {metadata.topic}:
{metadata.partition}@{metadata.offset}")

    def on_send_error(exc):
        print(f"Send failed: {exc}")

    producer.send("order-events", key="order_456", value={...}) \
        .add_callback(on_send_success) \
        .add_errback(on_send_error)

```

Producer acknowledgment modes (acks)

acks	Durability	Throughput	Risk
0	Fire-and-forget	Highest	Data loss on broker failure
1	Leader acknowledged	High	Loss if leader fails before replication
all / -1	All ISR replicas acknowledged	Lower	Safest — no loss if ISR count ≥ 2

Consumers & Consumer Groups

```

from kafka import KafkaConsumer
import json

consumer = KafkaConsumer(

```



```

    "order-events",
    bootstrap_servers=["kafka:9092"],
    group_id="analytics-pipeline",          # consumer group
    auto_offset_reset="earliest",          # start from
beginning if no committed offset
    enable_auto_commit=True,                # auto-commit
offsets every interval
    auto_commit_interval_ms=5000,
    value_deserializer=lambda m: json.loads(m.decode("utf-
8")),
    key_deserializer=lambda k: k.decode("utf-8") if k else
None,
)

# Continuous poll loop
for message in consumer:
    print(f"Topic: {message.topic}, Partition:
{message.partition}, "
          f"Offset: {message.offset}, Key: {message.key}")
    process(message.value)

consumer.close()

# Manual offset commit – more control over exactly-once
semantics
consumer = KafkaConsumer(
    "order-events",
    group_id="analytics-pipeline",
    enable_auto_commit=False,          # disable auto-commit
    bootstrap_servers=["kafka:9092"],
)

for message in consumer:
    try:
        process(message.value)
        consumer.commit()              # commit only after
successful processing
    except Exception as e:
        logger.error("Processing failed: %s", e)
        # don't commit – message will be redelivered on
restart

```

How consumer groups work

Topic: order-events (4 partitions) 

Consumer Group A (analytics):
(alerts):

Consumer A1 → Partition 0
Partition 0, 1, 2, 3
Consumer A2 → Partition 1
Consumer A3 → Partition 2, 3
(3 consumers share 4 partitions)

Consumer Group B

Consumer B1 →
(only 1 consumer –
gets all partitions)

Each group reads ALL messages independently.

Within a group, each partition is assigned to exactly ONE consumer.

Adding more consumers than partitions → some consumers are idle.

Offsets & Delivery Guarantees

Offset management

```
# Where to start consuming when no committed offset exists:   
auto_offset_reset="earliest"    # from the oldest available  
message  
auto_offset_reset="latest"      # from new messages only  
(skip historical)  
  
# Seek to a specific offset manually  
from kafka import TopicPartition  
  
tp = TopicPartition("order-events", 0)  
consumer.assign([tp])  
consumer.seek(tp, 1000)          # start from offset 1000  
  
# Seek to beginning / end  
consumer.seek_to_beginning(tp)  
consumer.seek_to_end(tp)
```

```
# Get current committed offset for a group
consumer.committed(tp)
```

Delivery guarantees

Guarantee	How	Risk
At-most-once	Commit offset before processing	Message lost if consumer crashes after commit but before processing
At-least-once	Commit offset after processing	Duplicate if consumer crashes after processing but before commit
Exactly-once	Transactional producer + consumer, or idempotent consumer	Complex; requires Kafka transactions or deduplication logic

At-least-once + idempotent consumer is the standard production pattern. Design your consumer to safely reprocess the same message (deduplicate by event ID).

Consumer Group Rebalancing

When consumers join or leave a group, Kafka **rebalances** — reassigns partitions to the current set of consumers. During rebalancing, all consumers in the group pause.

```
from kafka import ConsumerRebalanceListener

class RebalanceHandler(ConsumerRebalanceListener):
    def on_partitions_revoked(self, revoked):
        print(f"Partitions revoked: {revoked}")
        consumer.commit()    # commit current offsets
                              before losing partitions

    def on_partitions_assigned(self, assigned):
        print(f"Partitions assigned: {assigned}")

consumer.subscribe(["order-events"],
listener=RebalanceHandler())
```

Reduce rebalance frequency

```
KafkaConsumer(  
    "order-events",  
    bootstrap_servers=["kafka:9092"],  
    group_id="analytics-pipeline",  
    session_timeout_ms=30000,           # consumer declared  
dead after 30s of silence  
    heartbeat_interval_ms=10000,       # send heartbeat  
every 10s  
    max_poll_interval_ms=300000,       # max time between  
poll() calls (5 min)  
    max_poll_records=500,              # limit records per  
poll to stay within interval  
)
```

Kafka CLI

```
# Topics  
kafka-topics.sh --bootstrap-server kafka:9092 \  
  --create --topic order-events \  
  --partitions 6 \  
  --replication-factor 2  
  
kafka-topics.sh --bootstrap-server kafka:9092 --list  
kafka-topics.sh --bootstrap-server kafka:9092 --describe --  
topic order-events  
kafka-topics.sh --bootstrap-server kafka:9092 --delete --  
topic order-events  
  
# Change partition count (can only increase)  
kafka-topics.sh --bootstrap-server kafka:9092 \  
  --alter --topic order-events --partitions 12  
  
# Produce messages (interactive)  
kafka-console-producer.sh --bootstrap-server kafka:9092 --  
topic order-events  
# type JSON and press Enter for each message  
  
# Consume messages  
kafka-console-consumer.sh --bootstrap-server kafka:9092 \  
  --topic order-events --from-beginning
```

```
--topic order-events \  
--from-beginning \  
--group my-test-group  
  
# Consumer group info  
kafka-consumer-groups.sh --bootstrap-server kafka:9092 --  
list  
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \  
--describe --group analytics-pipeline  
# Shows: partition, current-offset, log-end-offset, LAG  
  
# Reset offsets (useful for reprocessing)  
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \  
--group analytics-pipeline \  
--topic order-events \  
--reset-offsets --to-earliest \  
--execute
```

Schema Registry & Avro

Schema Registry is a separate service (from Confluent) that stores and enforces message schemas. Every message is serialized with its schema ID — consumers always know the schema.

```
from confluent_kafka import Producer, Consumer  
from confluent_kafka.schema_registry import  
SchemaRegistryClient  
from confluent_kafka.schema_registry.avro import  
AvroSerializer, AvroDeserializer  
from confluent_kafka.serialization import  
SerializationContext, MessageField  
  
# Define Avro schema  
order_schema_str = """  
{  
    "type": "record",  
    "name": "Order",  
    "namespace": "com.mycompany.events",  
    "fields": [  
        {"name": "order_id",    "type": "string"},
```

```

        {"name": "customer_id", "type": "int"},
        {"name": "amount", "type": "double"},
        {"name": "status", "type": "string"},
        {"name": "created_at", "type": "long", "logicalType":
"timestamp-millis"}
    ]
}
"""

schema_registry = SchemaRegistryClient({"url":
"http://schema-registry:8081"})
serializer = AvroSerializer(schema_registry,
order_schema_str)
deserializer = AvroDeserializer(schema_registry)

# Produce with Avro
producer = Producer({"bootstrap.servers": "kafka:9092"})

order = {"order_id": "abc123", "customer_id": 42, "amount":
99.50,
        "status": "placed", "created_at": 1710000000000}

producer.produce(
    topic="order-events",
    value=serializer(order, SerializationContext("order-
events", MessageField.VALUE)),
)
producer.flush()

```

Schema evolution rules

Change	Backward compatible?	Forward compatible?
Add field with default	Yes	Yes
Remove field with default	Yes	Yes
Add field without default	No	Yes
Remove required field	Yes	No
Change field type	No	No

Kafka with Python

confluent-kafka (recommended — wraps librdkafka, high performance)

```
from confluent_kafka import Producer, Consumer, KafkaError,
KafkaException
import json

# Producer
producer = Producer({
    "bootstrap.servers": "kafka:9092",
    "acks": "all",
    "enable.idempotence": True,    # exactly-once producer
    semantics
})

def delivery_report(err, msg):
    if err:
        print(f"Delivery failed: {err}")
    else:
        print(f"Delivered to {msg.topic()}
[ {msg.partition()} ] @ {msg.offset()}")

producer.produce(
    "order-events",
    key="order_123",
    value=json.dumps({"order_id": "order_123", "amount":
50.0}),
    callback=delivery_report,
)
producer.poll(0)    # trigger delivery callbacks
producer.flush()

# Consumer
consumer = Consumer({
    "bootstrap.servers": "kafka:9092",
    "group.id":          "analytics-pipeline",
    "auto.offset.reset": "earliest",
    "enable.auto.commit": False,
})
```

```

consumer.subscribe(["order-events"])

try:
    while True:
        msg = consumer.poll(timeout=1.0)
        if msg is None:
            continue
        if msg.error():
            if msg.error().code() ==
KafkaError._PARTITION_EOF:
                continue    # end of partition – not an
error
                raise KafkaException(msg.error())

        data = json.loads(msg.value().decode("utf-8"))
        process(data)
        consumer.commit(asynchronous=False)    # synchronous
commit after processing

finally:
    consumer.close()

```

Kafka Connect

Kafka Connect moves data between Kafka and external systems without writing producers/consumers. Configured via JSON, not code.

Source connector — capture changes from Postgres into Kafka with Debezium. It creates the topics `cdc.public.orders` and `cdc.public.customers`, and every insert, update, and delete on those tables becomes an event:

```

{
    "name": "postgres-source-orders",
    "config": {
        "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
        "database.hostname": "postgres",
        "database.port": "5432",
        "database.user": "debezium",
        "database.password":
"${file:/secrets/db.properties:password}",

```

```

        "database.dbname": "myapp",
        "table.include.list":
"public.orders,public.customers",
        "topic.prefix": "cdc",
        "plugin.name": "pgoutput",
        "publication.name": "debezium_pub",
        "slot.name": "debezium_slot"
    }
}

```

Sink connector — write from Kafka to S3:

```

{
    "name": "s3-sink-orders",
    "config": {
        "connector.class":
"io.confluent.connect.s3.S3SinkConnector",
        "tasks.max": "4",
        "topics": "cdc.public.orders",
        "s3.region": "us-east-1",
        "s3.bucket.name": "my-data-lake",
        "s3.part.size": "67108864",
        "flush.size": "1000",
        "storage.class":
"io.confluent.connect.s3.storage.S3Storage",
        "format.class":
"io.confluent.connect.s3.format.parquet.ParquetFormat",
        "partitioner.class":
"io.confluent.connect.storage.partitionner",
        "path.format":
"'year'=YYYY/'month'=MM/'day'=dd/'hour'=HH",
        "locale": "en_US",
        "timezone": "UTC"
    }
}

```

Kafka Streams

Kafka Streams is a Java/Kotlin library for building stream processing applications that read from and write to Kafka. It runs inside your own application — there is no separate processing cluster — and scales by running more instances of the application, which share the work through a consumer group.

Core abstractions

Abstraction	Meaning	Example
KStream	An unbounded stream of independent events	Every order event
KTable	A changelog: the latest value per key	Current status of each order
GlobalKTable	A KTable fully replicated to every instance	Small reference data (currencies, regions)
State store	Local, fault-tolerant storage for aggregations and joins	Running revenue per customer
Changelog topic	Kafka topic that backs up a state store	Restores state after a restart

Example: revenue per customer in 5-minute windows

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "revenue-5m");
// also the consumer group id
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
props.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG, StreamsConfig.EXACTLY_ONCE_V2);

StreamsBuilder builder = new StreamsBuilder();

KStream<String, Order> orders =
    builder.stream("order-events",
        Consumed.with(Serdes.String(), orderSerde)); // key = customer_id

KTable<Windowed<String>, Double> revenue = orders
```

```

        .filter((customerId, order) ->
            "placed".equals(order.status()))
        .groupByKey(Grouped.with(Serdes.String(), orderSerde))

        .windowedBy(TimeWindows.ofSizeAndGrace(Duration.ofMinutes(5),
            Duration.ofMinutes(1)))
        .aggregate(
            () -> 0.0,
            // initial value
            (customerId, order, total) -> total +
            order.amount(), // adder
            Materialized.with(Serdes.String(),
                Serdes.Double())); // backed by a state store

revenue.toStream()
    .map((window, total) -> KeyValue.pair(window.key(),
        total))
    .to("revenue-5m", Produced.with(Serdes.String(),
        Serdes.Double()));

KafkaStreams streams = new KafkaStreams(builder.build(),
    props);
Runtime.getRuntime().addShutdownHook(new
    Thread(streams::close));
streams.start();

```

Joins

Join	Use case	Requirement
KStream-KStream (windowed)	Match events that occur close in time (order + payment)	Co-partitioned topics; a join window
KStream-KTable	Enrich events with the latest reference data	Co-partitioned (same key, same partition count)
KStream-GlobalKTable	Enrich with small reference data on any key	Fits in memory on every instance
KTable-KTable	Maintain a joined, current-state view	Co-partitioned

When to use Kafka Streams — and the alternatives

Option	Choose when
Kafka Streams	JVM teams; Kafka in and Kafka out; want to deploy as a normal application
Apache Flink	Complex event-time processing, large state, many sources and sinks, SQL interface
Spark Structured Streaming	Already on Spark; micro-batch latency (seconds) is acceptable; writes to a lakehouse
Python stream processors (e.g. Faust-streaming, Quix Streams, Bytewax)	Python-first teams with moderate throughput
Plain consumers	Simple, stateless transformations or sinks

Operational notes: state stores live on local disk and are restored from changelog topics after failures (standby replicas speed this up) · repartition topics are created automatically when you re-key a stream · `exactly_once_v2` covers Kafka-to-Kafka processing end to end · scale out up to the number of input partitions

Retention & Compaction

Time-based retention (default)

```
# Keep messages for 7 days (default: 168 hours)
kafka-configs.sh --bootstrap-server kafka:9092 \
  --entity-type topics --entity-name order-events \
  --alter --add-config retention.ms=604800000

# Keep up to 50 GB per partition
kafka-configs.sh --bootstrap-server kafka:9092 \
  --entity-type topics --entity-name order-events \
  --alter --add-config retention.bytes=53687091200
```

Log compaction

Instead of time-based deletion, keep only the **latest message per key**. Perfect for CDC — the topic acts like a changelog, always reflecting current state.

```
kafka-configs.sh --bootstrap-server kafka:9092 \
  --entity-type topics --entity-name customer-state \
  --alter --add-config cleanup.policy=compact

# Tombstone – delete a key by sending null value
producer.send("customer-state", key=b"customer_42",
value=None)
```

Performance Tuning

Property names differ between clients: `confluent-kafka` (`librdkafka`) rejects Java-client names like `buffer.memory` or `max.poll.records`.

Producer

```
# confluent-kafka / librdkafka names
Producer({
  "linger.ms": 20, # wait up to
20ms to batch messages
  "batch.size": 65536, # 64KB batch
size
  "compression.type": "zstd", # compress
batches (lz4/snappy also common)
  "queue.buffering.max.kbytes": 65536, # 64MB local
buffer (Java client: buffer.memory)
})
```

Consumer

```
# confluent-kafka / librdkafka names
Consumer({
  "group.id": "analytics-pipeline",
  "fetch.min.bytes": 1024, # wait for at
least 1KB before returning
  "fetch.wait.max.ms": 500, # ...or 500ms,
whichever comes first
  "max.partition.fetch.bytes": 1048576, # 1MB max per
partition per fetch
```

```
}  
# Batch size per call is controlled in code:  
consumer.consume(num_messages=500, timeout=1.0)  
# (the Java client and kafka-python use max.poll.records  
instead)
```

Broker

```
num.network.threads=8          # threads for handling  
network I/O  
num.io.threads=8               # threads for disk I/O  
socket.send.buffer.bytes=102400  
socket.receive.buffer.bytes=102400  
log.segment.bytes=1073741824  # 1GB per log segment  
log.retention.check.interval.ms=300000
```

Kafka in DE Pipelines

Common patterns

```
Pattern 1: CDC → Data Lake  
  OLTP DB → Debezium (Kafka Connect) → Kafka → S3 Sink  
  (Kafka Connect) → Parquet files → SQL transformations  
  
Pattern 2: Event Stream → Real-time Aggregation  
  App Events → Kafka → Flink/Spark Structured Streaming →  
  Aggregated tables → BI  
  
Pattern 3: Fan-out  
  Single event topic → multiple consumer groups:  
    → Group 1: Analytics pipeline (warehouse load)  
    → Group 2: Alerting service  
    → Group 3: ML feature store  
  
Pattern 4: Kafka as Buffer  
  High-volume API events → Kafka (absorbs spikes) →  
  Consumer (steady write rate to DB)
```

Consumer lag monitoring

Consumer lag = log-end-offset – current-offset = how far behind a consumer is.

```
# Check lag for all groups
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
    --describe --all-groups

# Output:
# GROUP          TOPIC          PARTITION  CURRENT-
OFFSET LOG-END-OFFSET LAG
# analytics-pipeline order-events  0          1500
1502          2
# analytics-pipeline order-events  1          980
985          5
```

High lag = consumer is falling behind producers. Investigate: 1. Consumer processing too slow → optimize or scale out consumers 2. Producer spike → temporary; will catch up 3. Consumer crashed → restart it

Dead Letter Queue (DLQ)

```
dlq_producer = Producer({"bootstrap.servers":
    "kafka:9092"})

while True:
    msg = consumer.poll(timeout=1.0)
    if msg is None or msg.error():
        continue
    try:
        process(msg.value())
    except Exception as e:
        # Send the failed message to a DLQ for later
        # inspection/reprocessing
        dlq_producer.produce(
            "order-events-dlq",
            key=msg.key(),
            value=msg.value(),
            headers={"error": str(e), "original_topic":
msg.topic(),
                    "original_offset": str(msg.offset())},
        )
```

```
dlq_producer.flush()
consumer.commit(message=msg, asynchronous=False)  #
move past the message either way
```

Common Pitfalls

Pitfall	Symptom	Fix
Auto-commit on, processing done afterwards	Messages lost when a consumer crashes mid-batch (offset already committed)	<code>enable.auto.commit=false</code> ; commit after processing succeeds
Assuming exactly-once end to end	Occasional duplicates in the warehouse	At-least-once plus idempotent writes (upsert on an event ID); transactions only where really needed
Too few partitions	Can't scale consumers past the partition count	Size for peak throughput and future consumers; you can add partitions, never remove them
Adding partitions to a keyed topic	Keys move to different partitions — per-key ordering breaks	Choose the partition count up front; if you must change it, migrate to a new topic
Hot keys (one <code>customer_id</code> producing most events)	One partition and its consumer lag while others sit idle	Better key choice, key salting, or a separate topic for the heavy hitter
<code>acks=1</code> / idempotence off for important data	Lost or duplicated messages on broker failover	<code>acks=all</code> , <code>enable.idempotence=true</code> , <code>min.insync.replicas=2</code> with replication factor 3
Slow processing inside the poll loop	<code>max.poll.interval.ms</code> exceeded → rebalance storms	Keep per-message work small; process in batches; hand heavy work to a worker pool
JSON without a schema contract	A producer renames a field and every consumer breaks	Avro/Protobuf + Schema Registry with a compatibility mode (usually <code>BACKWARD</code>)

Pitfall	Symptom	Fix
Poison messages retried forever	One bad record blocks the partition	Retry a few times, then send it to a dead-letter topic with error headers
Retention shorter than your recovery time	Can't replay after an outage or a bug fix	Set retention to cover your worst-case recovery window; archive to S3 for longer history
Nobody watching consumer lag	The "real-time" dashboard is hours behind	Alert on lag (in messages and in time) per consumer group
Using Kafka as a database	Lookups need full scans; data expires	Kafka is a log; sink to a database, warehouse, or lakehouse for querying

Cheat Sheet

Task	Command
Create a topic	<code>kafka-topics.sh --bootstrap-server b:9092 --create --topic t --partitions 6 --replication-factor 3</code>
Describe (leaders, ISR)	<code>kafka-topics.sh --bootstrap-server b:9092 --describe --topic t</code>
Change a topic setting	<code>kafka-configs.sh --bootstrap-server b:9092 --alter --entity-type topics --entity-name t --add-config retention.ms=604800000</code>
Tail a topic	<code>kafka-console-consumer.sh --bootstrap-server b:9092 --topic t --from-beginning --property print.key=true</code>
Produce test messages	<code>kafka-console-producer.sh --bootstrap-server b:9092 --topic t --property parse.key=true --property key.separator=:</code>
Consumer lag	<code>kafka-consumer-groups.sh --bootstrap-server b:9092 --describe --group g</code>
Replay from a point in time	<code>kafka-consumer-groups.sh ... --group g --topic t --reset-offsets --to-datetime 2024-03-15T00:00:00.000 --execute (group must be stopped)</code>
Handy CLI	<code>kcat -b b:9092 -t t -C -o -10 -e (last 10 messages)</code>

Reliable producer: `acks=all` · `enable.idempotence=true` · `compression.type=zstd` · `linger.ms=5-20` · a key when per-entity ordering matters

Reliable consumer: `enable.auto.commit=false` · commit after processing · idempotent sink · dead-letter topic · lag alerting · `auto.offset.reset=earliest` for pipelines

Topic defaults for production: replication factor 3 · `min.insync.replicas=2` · retention sized to your replay needs · `cleanup.policy=compact` for "latest value per key" topics

Schema compatibility modes: `BACKWARD` (new consumers can read old data — the default) · `FORWARD` (old consumers can read new data) · `FULL` (both)

Ecosystem map: Kafka Connect (Debezium CDC, S3/Snowflake/Iceberg sinks) · Schema Registry · Kafka Streams (Java) · Flink / Spark Structured Streaming · managed services: Confluent Cloud, Amazon MSK, Redpanda, Aiven

Interview Questions

Q: What is a Kafka partition and why does it exist? A: A partition is a single ordered log within a topic. Partitions enable parallelism — multiple consumers in a group can read from different partitions simultaneously. With 6 partitions, you can have up to 6 consumers processing in parallel. Messages within a partition are strictly ordered; ordering across partitions is not guaranteed. Partition count is a key capacity decision — you can increase it but never decrease.

Q: What are the three delivery guarantee modes in Kafka? A: (1) At-most-once: messages may be lost, never duplicated — achieved by committing offsets before processing. (2) At-least-once (default): messages are never lost but may be duplicated — achieved by committing offsets after processing. (3) Exactly-once: no loss, no duplicates — requires idempotent producers (`enable.idempotence=true`) and transactional consumers. Exactly-once is the hardest to achieve and has performance overhead.

Q: What is consumer lag and how do you respond to it? A: Consumer lag is the difference between the latest offset published to a partition and the offset the consumer has processed. Growing lag means the consumer is falling behind the producer. Responses: (1) scale out — add more consumers (up to partition count); (2) optimize consumer processing — parallelize or batch; (3) increase batch size (`max.poll.records`); (4) check if the producer is having a burst — lag may be temporary.

Q: When would you use a partition key and what happens if you don't use one? A: Without a key, messages are distributed round-robin across partitions — good for even load distribution but no ordering guarantee across messages for the same entity. With a key

(e.g., `customer_id`), all messages for that key go to the same partition — guaranteeing ordering per key, enabling stateful processing. Use keys when message ordering per entity matters (e.g., order state transitions must be processed in order).

Q: What is the difference between Kafka and a traditional message queue like RabbitMQ?

A: In a queue, each message is consumed by exactly one consumer and deleted after acknowledgment. In Kafka, messages are written to a log and retained for a configurable period — any number of consumer groups can read them independently, and consumers can rewind and reprocess. Kafka scales to millions of messages/sec; queues are better for task distribution and work queues where retention isn't needed.

Q: How does Kafka guarantee ordering? A: Only within a partition. Messages with the same key are hashed to the same partition, so all events for one `order_id` are read in the order they were written. There's no ordering across partitions. With retries enabled, the idempotent producer (the default since Kafka 3.0) keeps ordering intact within a partition even when sends are retried.

Q: What are replication, ISR, and `min.insync.replicas`? A: Each partition has a leader and follower replicas on other brokers (the replication factor). The ISR is the set of replicas fully caught up with the leader. With `acks=all`, a write succeeds only once every in-sync replica has it, and `min.insync.replicas` sets how many that must be — with RF=3 and `min.insync.replicas=2`, the cluster tolerates one broker failure without losing acknowledged data, and rejects writes rather than risk loss if two replicas are down.

Q: What is log compaction and when would you use it? A: Instead of deleting data by age, a compacted topic keeps at least the latest message for each key and removes older versions in the background (a message with a null value, called a tombstone, deletes the key). It turns a topic into a changelog of current state — used for CDC "latest row" topics, Kafka Streams state stores, and Kafka's own `__consumer_offsets` topic.

Q: How does Kafka achieve exactly-once processing? A: Three pieces. The idempotent producer attaches a producer ID and sequence numbers so broker-side retries don't create duplicates. Transactions let a producer write to several partitions *and* commit consumer offsets atomically, so a read-process-write cycle either fully happens or doesn't. Consumers use `isolation.level=read_committed` so they never see aborted writes. This covers Kafka-to-Kafka pipelines; for external sinks you still need idempotent writes or transactional connectors.

Q: How would you get changes from a Postgres database into a data lake in near real time? A: CDC with Debezium running on Kafka Connect: it reads Postgres's write-ahead log (via logical replication) and publishes every insert, update, and delete as an event to a topic per table, with before/after images. A sink — an S3/Iceberg sink connector, or a Spark/Flink job — writes those events to the lake and applies them with `MERGE` to keep a current-state table, often alongside an append-only history table. Avro plus Schema Registry handles schema changes, and log compaction on the topics keeps the latest state per key.

Further Reading

- [Apache Kafka documentation](https://kafka.apache.org/documentation/) (https://kafka.apache.org/documentation/)
- [Confluent Developer courses](https://developer.confluent.io/courses/) (https://developer.confluent.io/courses/) — free, from fundamentals to internals
- [confluent-kafka Python client](https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html) (https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html) and the [librdkafka configuration reference](https://github.com/confluentinc/librdkafka/blob/master/CONFIGURATION.md) (https://github.com/confluentinc/librdkafka/blob/master/CONFIGURATION.md)
- [Debezium documentation](https://debezium.io/documentation/) (https://debezium.io/documentation/)
- *Kafka: The Definitive Guide, 2nd Edition* — Gwen Shapira, Todd Palino, Rajini Sivaram & Krit Petty (O'Reilly)
- [kcat](https://github.com/edenhill/kcat) (https://github.com/edenhill/kcat) — the netcat of Kafka, great for debugging

Previous: [Trino](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/trino-federation/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/trino-federation/) • **Next:** [Apache Flink](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/flink-reference/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026