# Apache Kafka

## Interview Preparation Guide

*Java Edition*

Comprehensive Q&A | Code Examples | Best Practices | Cheat Sheet

### Topics Covered

- Core Kafka Architecture & Concepts
- Producers, Consumers & Consumer Groups
- Topics, Partitions & Offsets
- Kafka Streams & Connect
- Replication, Fault Tolerance & Reliability
- Performance Tuning & Configuration
- Security & Monitoring
- Advanced Patterns & Best Practices

**⭐ = Frequently Asked in Interviews**

# Table of Contents


1. Introduction to Apache Kafka ..... 3
1. Introduction to Apache Kafka ..... 3
2. Topics, Partitions & Offsets ..... 4
2. Topics, Partitions & Offsets ..... 4
3. Kafka Producers ..... 6
3. Kafka Producers ..... 6
4. Kafka Consumers & Consumer Groups ..... 8
4. Kafka Consumers & Consumer Groups ..... 8
5. Replication, Fault Tolerance & Reliability ..... 10
5. Replication, Fault Tolerance & Reliability ..... 10
6. Performance Tuning & Configuration ..... 12
6. Performance Tuning & Configuration ..... 12
7. Kafka Streams ..... 13
7. Kafka Streams ..... 13
8. Kafka Connect ..... 15
8. Kafka Connect ..... 15
9. Advanced Kafka Concepts ..... 16
9. Advanced Kafka Concepts ..... 16
10. Kafka Security & Monitoring ..... 18
10. Kafka Security & Monitoring ..... 18
11. Patterns, Best Practices & Common Pitfalls ..... 19
11. Patterns, Best Practices & Common Pitfalls ..... 19
12. Scenario-Based Interview Questions ..... 21
12. Scenario-Based Interview Questions ..... 21
CHEAT SHEET: Quick Reference ..... 22
CHEAT SHEET: Quick Reference ..... 22
- Key Configuration Cheat Sheet ..... 22
- Key Configuration Cheat Sheet ..... 22
  - Producer Configs ..... 22
  - Producer Configs ..... 22
  - Consumer Configs ..... 22
  - Consumer Configs ..... 22
  - Broker / Topic Configs ..... 22
  - Broker / Topic Configs ..... 22
- Kafka CLI Quick Reference ..... 22
- Kafka CLI Quick Reference ..... 22
- Frequently Asked Questions Summary ..... 23
- Frequently Asked Questions Summary ..... 23
- Kafka Delivery Guarantees at a Glance ..... 24
- Kafka Delivery Guarantees at a Glance ..... 24

# 1. Introduction to Apache Kafka

### ⭐ FREQUENTLY ASKED Q1. What is Apache Kafka? Why is it used?

**Answer:**

Apache Kafka is an open-source, distributed event streaming platform originally developed by LinkedIn and later donated to the Apache Software Foundation. It is designed for high-throughput, fault-tolerant, real-time data streaming.

Key reasons Kafka is used:

- **Event Streaming:** Real-time ingestion and processing of millions of events per second
- **Decoupling:** Producers and consumers are independent; producers don't know about consumers
- **Durability:** Messages are persisted to disk and replicated across brokers
- **Scalability:** Horizontally scalable via partitioning across multiple brokers
- **Fault Tolerance:** Replication ensures data availability even when brokers fail
- **Replay:** Consumers can re-read past messages by resetting offsets

□ **Note:** *Kafka is not a traditional message queue — it is a distributed commit log optimized for sequential disk I/O.*

### ⭐ FREQUENTLY ASKED Q2. What are the core components of Kafka?

**Answer:**

- **Broker:** A Kafka server that stores messages. A Kafka cluster consists of multiple brokers
- **Topic:** A named, ordered, immutable stream of records. Topics are split into partitions
- **Partition:** An ordered, immutable sequence of records. Enables parallelism and scalability
- **Producer:** A client that publishes (writes) records to a Kafka topic
- **Consumer:** A client that subscribes to topics and reads records from partitions
- **Consumer Group:** A group of consumers that jointly consume a topic; each partition is assigned to one consumer
- **ZooKeeper / KRaft:** Manages cluster metadata, broker leadership, and configuration (ZooKeeper is legacy; KRaft is the modern replacement)
- **Offset:** A sequential number identifying each record within a partition

### Q3. What is the difference between Kafka and traditional message queues (e.g., RabbitMQ)?

**Answer:**

| Feature | Apache Kafka | Traditional MQ (RabbitMQ) |
|---|---|---|
| Model | Log-based (distributed commit log) | Queue-based (push model) |
| Retention | Configurable (days/size based) | Deleted after acknowledgment |
| Replay | Yes — reset offset to re-read | No — message gone after consume |
| Ordering | Per-partition ordering guaranteed | Varies by queue type |
| Scalability | Horizontal via partitions | Limited vertical/cluster scaling |
| Consumer model | Pull-based | Push-based |
| Throughput | Very high (millions/sec) | Moderate |
| Use case | Streaming, audit logs, event sourcing | Task queues, work distribution |

# 2. Topics, Partitions & Offsets

### ⭐ FREQUENTLY ASKED Q4. What is a Kafka Topic? How does partitioning work?

**Answer:**

A Topic is a category or feed name to which records are published. Topics in Kafka are:

- Multi-subscriber: multiple consumers can read from the same topic
- Durable: records are retained based on retention policy (time or size)
- Partitioned: each topic is divided into one or more partitions for parallelism

Partitioning mechanics:

- **Ordering:** Order is guaranteed only within a single partition, not across partitions
- **Key-based routing:** Records with the same key always go to the same partition (murmur2 hash by default)
- **Round-robin:** If no key, records are distributed across partitions round-robin
- **Custom partitioner:** You can implement a custom Partitioner interface

```
// Example: creating a topic with 3 partitions, replication factor 2
NewTopic topic = new NewTopic("orders", 3, (short) 2);
adminClient.createTopics(Collections.singletonList(topic)).all().get();
```

### ⭐ FREQUENTLY ASKED Q5. What is an Offset in Kafka? How is it managed?

**Answer:**

An offset is a unique, sequential integer assigned to each record within a partition. Offsets are:

- Immutable: once assigned, an offset never changes
- Per-partition: offset sequences are independent for each partition
- Consumer-tracked: consumers store their current offset to know what to read next

Offset storage options:

- **Auto-commit:** Kafka auto-commits offsets every auto.commit.interval.ms (default 5000ms)
- **Manual commit:** Use commitSync() or commitAsync() for fine-grained control
- **External store:** Store offsets in a database for exactly-once semantics

```
// Manual offset commit example
consumer.poll(Duration.ofMillis(100)).forEach(record -> process(record));
consumer.commitSync(); // blocks until broker confirms
```

### Q6. How do you choose the right number of partitions for a topic?

**Answer:**

Choosing partition count involves trade-offs:

- **Parallelism:** More partitions = more consumer threads can read in parallel
- **Throughput:** Each partition is handled by one consumer in a group; more partitions = higher throughput

- **Overhead:** Each partition has memory/file handle overhead on brokers and ZooKeeper
- **Rule of thumb:** Start with max(target throughput / single partition throughput, consumer parallelism)
- **Ordering requirements:** More partitions mean less strict ordering across the topic

 **Note:** *You can increase partition count after creation, but you cannot decrease it. Increasing partitions changes the key-to-partition mapping, which can break ordering guarantees for keyed messages.*

# 3. Kafka Producers

### ⭐ FREQUENTLY ASKED Q7. How does the Kafka Producer work internally?

**Answer:**

The Kafka Producer follows a batching, async send pipeline:

- **Serialization:** Key and value are serialized using configured serializers
- **Partitioner:** Determines which partition to route the record to
- **RecordAccumulator:** Records are batched into a buffer grouped by topic-partition (batch.size, linger.ms)
- **Sender thread:** A background thread drains batches and sends them to brokers
- **Acknowledgment:** Broker sends ack based on acks config; retries happen on failure

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
props.put("acks", "all");              // strongest durability
props.put("retries", 3);
props.put("linger.ms", 5);              // wait up to 5ms to fill a batch
props.put("batch.size", 16384);         // 16KB batch
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
ProducerRecord<String, String> record = new ProducerRecord<>("orders",
"key1", "value1");
producer.send(record, (metadata, exception) -> {
    if (exception != null) exception.printStackTrace();
    else System.out.println("Sent to partition " + metadata.partition());
});
producer.flush(); producer.close();
```

### ⭐ FREQUENTLY ASKED Q8. What are the acks settings in Kafka Producer? What are the trade-offs?

**Answer:**

| acks Value | Behavior | Durability | Performance |
|---|---|---|---|
| acks=0 | Fire and forget — no ack waited | Lowest (data loss possible) | Highest throughput |
| acks=1 (default) | Waits for leader broker ack | Moderate (loss if leader dies before replica sync) | Good throughput |
| acks=all (-1) | Waits for all in-sync replicas (ISR) to ack | Highest (no data loss) | Lower throughput, higher latency |

 **Note:** *Use acks=all with min.insync.replicas=2 for production systems requiring no data loss.*

### ⭐ FREQUENTLY ASKED Q9. What is idempotent producer? How do you enable exactly-once semantics?

**Answer:**

An idempotent producer ensures that retries do not result in duplicate messages. Each message gets a unique sequence number and producer ID (PID). If a retried batch has the same sequence number, the broker deduplicates it.

```
props.put("enable.idempotence", "true");
// Also automatically sets: acks=all, retries=MAX_INT, max.in.flight=5
```

Exactly-Once Semantics (EOS) — full pipeline:

- **Idempotent producer:** Deduplicates retries within a session
- **Transactional producer:** Atomic writes across multiple partitions/topics
- **Isolation level:** Consumer reads only committed transactions (read_committed)

```
props.put("transactional.id", "my-transactional-id");
producer.initTransactions();
producer.beginTransaction();
producer.send(new ProducerRecord<>("topic-a", key, val));
producer.send(new ProducerRecord<>("topic-b", key, val));
producer.commitTransaction(); // or abortTransaction() on failure
```

# 4. Kafka Consumers & Consumer Groups

### ⭐ FREQUENTLY ASKED Q10. How does the Kafka Consumer work? What is a Consumer Group?

**Answer:**

Kafka Consumers use a pull model — they periodically poll the broker for new messages. The consumer maintains its position (offset) in each partition.

Consumer Group concepts:

- Each consumer group has a unique group.id
- Within a group, each partition is assigned to exactly ONE consumer
- If consumers > partitions, some consumers will be idle
- If consumers < partitions, some consumers will read from multiple partitions
- Multiple groups can independently read the same topic (broadcast/fan-out pattern)

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "order-processors");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());
props.put("auto.offset.reset", "earliest"); // or latest
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("orders"));
while (true) {
    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("offset=%d key=%s value=%s%n",
            record.offset(), record.key(), record.value());
    }
}
```

### ⭐ FREQUENTLY ASKED Q11. What is Consumer Rebalancing? What triggers it?

**Answer:**

Rebalancing is the process of redistributing partition assignments among consumers in a group. It ensures all active consumers have a fair share of partitions.

Triggers for rebalancing:

- A new consumer joins the group
- A consumer leaves (explicitly or due to crash / heartbeat timeout)
- A subscribed topic gets new partitions added

- Consumer calls unsubscribe()

Impact of rebalancing:

- During rebalance, ALL consumers in the group stop consuming (stop-the-world)
- May cause duplicate processing if offsets weren't committed before the rebalance

Minimizing rebalance impact:

- **session.timeout.ms:** Increase to avoid false timeouts (default: 45000ms)
- **heartbeat.interval.ms:** Should be ~1/3 of session.timeout.ms
- **max.poll.interval.ms:** Increase if processing is slow (default: 300000ms)
- **Static membership:** Use group.instance.id to allow consumers to rejoin without triggering full rebalance

 **Note:** *Kafka 2.4+ introduced incremental cooperative rebalancing (CooperativeStickyAssignor) which only revokes partitions that need to move, avoiding stop-the-world rebalances.*

## Q12. What is the difference between auto.offset.reset=earliest vs latest?

**Answer:**

- **earliest:** Consumer starts reading from the beginning of the partition (offset 0 or lowest available). Used when you want to process all historical data.
- **latest (default):** Consumer starts reading from the next new message produced after subscription. Historical messages are skipped.
- **none:** Throws NoOffsetForPartitionException if no committed offset is found for the consumer group.

 **Note:** *This setting only applies when a consumer group has NO committed offsets. If a committed offset exists, reading resumes from that offset regardless of auto.offset.reset.*

## Q13. What is the difference between commitSync() and commitAsync()?

**Answer:**

| Method | Behavior | Performance | Error Handling |
|---|---|---|---|
| commitSync() | Blocks until broker confirms commit; retries on failure | Lower throughput | Automatic retry; throws exception on final failure |
| commitAsync() | Non-blocking; fires and forgets commit to broker | Higher throughput | No automatic retry; use callback for error logging |
| Best practice | Use commitAsync() in loop; commitSync() on shutdown | | |

# 5. Replication, Fault Tolerance & Reliability

### ⭐ FREQUENTLY ASKED Q14. How does Kafka replication work?

**Answer:**

Kafka replicates each partition across multiple brokers for fault tolerance:

- **Leader:** Each partition has one leader that handles all reads and writes
- **Followers:** Follower replicas passively copy data from the leader
- **ISR (In-Sync Replicas):** The set of replicas that are fully caught up with the leader
- **Replication Factor:** Number of total replicas (including leader). Recommended: 3 in production

Leader election:

- If a leader fails, Kafka elects a new leader from the ISR
- If ISR is empty and unclean.leader.election.enable=true, an out-of-sync replica can become leader (data loss risk)
- unclean.leader.election.enable=false (default since Kafka 0.11) prevents data loss at the cost of availability

```
// Set replication factor when creating a topic
NewTopic topic = new NewTopic("payments", 6, (short) 3); // 6 partitions, RF=3
```

### ⭐ FREQUENTLY ASKED Q15. What is min.insync.replicas and why is it important?

**Answer:**

min.insync.replicas (minISR) is a broker/topic configuration specifying the minimum number of in-sync replicas that must acknowledge a write for it to be considered successful (only enforced when acks=all).

Example scenario with RF=3:

- min.insync.replicas=2 means at least 2 replicas (leader + 1 follower) must be in-sync
- If only 1 replica is in-sync, the producer gets NotEnoughReplicasException
- This prevents data loss even if a broker fails immediately after write

 **Note:** *Golden rule for production: replication.factor=3, min.insync.replicas=2, acks=all. This tolerates 1 broker failure with zero data loss.*

### Q16. What happens when a Kafka broker goes down?

**Answer:**

When a broker fails:

- ZooKeeper (or KRaft controller) detects the failure via missed heartbeats
- For each partition where the failed broker was the LEADER, a new leader is elected from ISR
- Producers and consumers automatically reconnect and redirect to new partition leaders
- For partitions where the failed broker was a FOLLOWER, no leader election is needed; ISR shrinks

- When the broker recovers, it re-joins as a follower and catches up (log truncation may occur)

# 6. Performance Tuning & Configuration

### ⭐ FREQUENTLY ASKED Q17. What are the key producer configurations for high throughput?

**Answer:**

| Config | Default | Recommended for Throughput | Effect |
|---|---|---|---|
| batch.size | 16384 (16KB) | 65536–131072 (64–128KB) | Larger batches = fewer network round trips |
| linger.ms | 0 | 5–50ms | Wait to fill batches; increases latency but boosts throughput |
| compression.type | none | lz4 or snappy | Reduces message size; LZ4 is fastest |
| buffer.memory | 33554432 (32MB) | 67108864 (64MB) | More buffer for high-volume producers |
| max.in.flight.requests.per.connection | 5 | 5 (or 1 for ordering) | Higher = more concurrent requests |
| acks | 1 | 1 (throughput) / all (durability) | Trade-off between speed and safety |

### Q18. What are the key consumer configurations for high throughput?

**Answer:**

| Config | Default | Tuning Tip |
|---|---|---|
| fetch.min.bytes | 1 | Increase to 1024+ to reduce fetches; waits for more data |
| fetch.max.wait.ms | 500ms | Max wait for fetch.min.bytes to be met |
| max.poll.records | 500 | Increase for batch processing; adjust with processing speed |
| max.partition.fetch.bytes | 1048576 (1MB) | Per-partition fetch size; increase for large messages |
| enable.auto.commit | true | Set false for manual commit control |
| auto.commit.interval.ms | 5000ms | Only relevant when auto.commit=true |

### Q19. How does Kafka achieve such high throughput?

**Answer:**

Kafka achieves high throughput through several design choices:

- **Sequential disk I/O:** Kafka appends records sequentially to log files, which is much faster than random reads/writes
- **Zero-copy transfer:** Uses sendfile() system call to transfer data from disk to network without copying to user space
- **Batching:** Producers batch multiple records into a single network request
- **Compression:** Batches can be compressed (gzip, snappy, lz4, zstd)
- **Page cache:** Kafka relies on the OS page cache, avoiding double-buffering in JVM heap
- **Partitioning:** Multiple partitions allow concurrent parallel reads/writes

# 7. Kafka Streams

### ⭐ FREQUENTLY ASKED Q20. What is Kafka Streams? How is it different from Kafka Consumer?

**Answer:**

Kafka Streams is a client-side library for building real-time stream processing applications. Unlike a simple consumer, Kafka Streams provides:

- **Stateful processing:** Maintains local state stores (RocksDB by default)
- **Windowing:** Time-based and session-based windowed aggregations
- **Exactly-once processing:** Built-in EOS support
- **Join operations:** Stream-stream, stream-table, and table-table joins
- **KStream / KTable abstraction:** KStream = unbounded event stream; KTable = changelog/snapshot

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> orders = builder.stream("orders");
KTable<String, Long> orderCounts = orders
    .groupByKey()
    .count(Materialized.as("order-counts-store"));
orderCounts.toStream().to("order-counts", Produced.with(Serdes.String(),
Serdes.Long()));
KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
```

### Q21. What is the difference between KStream and KTable?

**Answer:**

| Aspect | KStream | KTable |
|---|---|---|
| Semantics | Infinite unbounded event stream | Changelog / snapshot of latest values |
| Each record | Appended (INSERT semantics) | Upserted by key (UPDATE semantics) |
| Null value | Treated as regular event | Treated as DELETE (tombstone) |
| State | Stateless by default | Stateful — latest value per key stored |
| Use case | Processing individual events (clicks, transactions) | Maintaining running state (counts, user profiles) |
| Analogy | Log file | Database table |

### Q22. What is a State Store in Kafka Streams?

**Answer:**

A State Store is a local storage mechanism used by Kafka Streams tasks to maintain stateful data (counts, aggregations, joins). Key characteristics:

- **Default implementation:** RocksDB (persistent) or in-memory HashMap
- **Changelog topic:** State changes are backed up to a Kafka topic for fault recovery
- **Recovery:** On restart, state is restored from the changelog topic
- **Interactive queries:** State stores can be queried externally via queryableStoreName

```
// Creating a persistent state store
StoreBuilder<KeyValueStore<String, Long>> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("my-store"),
        Serdes.String(), Serdes.Long());
builder.addStateStore(storeBuilder);
```

# 8. Kafka Connect

## Q23. What is Kafka Connect? What are Source and Sink Connectors?

**Answer:**

Kafka Connect is a framework for reliably streaming data between Kafka and external systems (databases, file systems, cloud services, etc.) without writing custom producers/consumers.

- **Source Connector:** Reads data FROM an external system INTO Kafka (e.g., JDBC Source reads from a database and writes to Kafka)
- **Sink Connector:** Reads data FROM Kafka and writes TO an external system (e.g., JDBC Sink writes Kafka messages to a database)
- **Workers:** Kafka Connect workers are distributed, fault-tolerant, and scalable processes that run connectors
- **Tasks:** Each connector is split into tasks; tasks run in parallel across workers

Common connectors:

- Debezium (CDC from MySQL, PostgreSQL, MongoDB)
- JDBC Connector (relational databases)
- S3 Sink Connector
- Elasticsearch Sink Connector
- HDFS Connector

# 9. Advanced Kafka Concepts

## Q24. What is Log Compaction in Kafka?

**Answer:**

Log compaction is a retention mechanism that retains only the LATEST record for each key, instead of retaining records for a fixed time or size. This is useful for:

- Event sourcing and changelog topics
- Database change-data-capture (CDC) snapshots
- Topics used as KTables in Kafka Streams

How it works:

- A background cleaner thread periodically scans dirty log segments
- For each key, only the record with the highest offset is retained
- A null value (tombstone) for a key marks it for eventual deletion

```
// Enable log compaction for a topic
Map<String, String> config = new HashMap<>();
config.put("cleanup.policy", "compact"); // or "compact,delete" for both
adminClient.alterConfigs(Collections.singletonMap(
    new ConfigResource(ConfigResource.Type.TOPIC, "user-profiles"),
    new Config(config.entrySet().stream()
        .map(e -> new ConfigEntry(e.getKey(), e.getValue()))
        .collect(Collectors.toList()))).all().get();
```

## ⭐ FREQUENTLY ASKED Q25. What is KRaft mode in Kafka? Why is ZooKeeper being removed?

**Answer:**

KRaft (Kafka Raft Metadata mode) is the self-managed metadata quorum that replaces ZooKeeper in Kafka 3.x and later. Reasons for removal:

- **Operational complexity:** ZooKeeper requires a separate cluster with separate monitoring and ops
- **Scalability limits:** ZooKeeper becomes a bottleneck with very large numbers of partitions
- **Startup time:** With ZooKeeper, controller failover can take 30+ seconds; KRaft makes it milliseconds
- **Unified security model:** Single security config instead of managing two separate systems

KRaft uses a Raft-based consensus protocol within Kafka brokers themselves to manage cluster metadata. Kafka 3.3+ supports KRaft for production use.

## Q26. Explain Kafka delivery semantics: at-most-once, at-least-once, exactly-once.

**Answer:**

| Semantics | Description | Configuration | Risk |
|---|---|---|---|
| At-most-once | Message delivered 0 or 1 times; may be lost | acks=0 or 1, no retries | Data loss possible |

| | | | |
|---|---|---|---|
| At-least-once | Message delivered 1+ times; may be duplicated | acks=all, retries>0, no idempotence | Duplicates possible |
| Exactly-once | Message delivered exactly once; no loss, no duplication | enable.idempotence=true + transactions, read_committed isolation | Most complex/overhead |

## Q27. How do you handle schema evolution in Kafka?

**Answer:**

Schema evolution is managing changes to message formats over time. Approaches:

- **Schema Registry (Confluent):** Centralizes Avro/JSON/Protobuf schemas; enforces compatibility rules; producers register schemas, consumers fetch them by ID
- **Backward compatibility:** New schema can read data written by old schema (add optional fields with defaults)
- **Forward compatibility:** Old schema can read data written by new schema (remove optional fields)
- **Full compatibility:** Both backward and forward compatible

```
// Avro serializer with Schema Registry
props.put("value.serializer",
"io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", "http://schema-registry:8081");
```

# 10. Kafka Security & Monitoring

## Q28. What security mechanisms does Kafka support?

**Answer:**

- **Authentication:** SASL/PLAIN, SASL/SCRAM, SASL/GSSAPI (Kerberos), mTLS (mutual TLS)
- **Encryption in transit:** TLS/SSL for all broker-client and broker-broker communication
- **Authorization:** ACLs (Access Control Lists) using AclAuthorizer; can also use OPA or custom authorizers
- **Encryption at rest:** Not natively supported; use filesystem/volume encryption

```
// Producer with SSL + SASL
props.put("security.protocol", "SASL_SSL");
props.put("sasl.mechanism", "SCRAM-SHA-256");
props.put("ssl.truststore.location", "/path/to/truststore.jks");
props.put("ssl.truststore.password", "changeit");
```

## Q29. What are important Kafka metrics to monitor?

**Answer:**

### Broker Metrics

- kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec — message ingestion rate
- kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions — must be 0
- kafka.controller:type=KafkaController,name=ActiveControllerCount — must be exactly 1
- kafka.server:type=ReplicaManager,name=IsrShrinksPerSec — ISR shrinks indicate issues

### Consumer Metrics

- records-lag-max — maximum consumer lag; critical for monitoring backlog
- fetch-rate — number of fetch requests per second
- bytes-consumed-rate — throughput consumed

### Producer Metrics

- record-send-rate — records sent per second
- record-error-rate — failed sends; should be 0
- request-latency-avg — average request latency to broker

 **Note:** *Tools commonly used: Prometheus + Grafana with JMX Exporter, Confluent Control Center, Burrow (consumer lag monitoring), and LinkedIn's Cruise Control for rebalancing.*

# 11. Patterns, Best Practices & Common Pitfalls

## Q30. What are common Kafka design patterns?

**Answer:**

- **Event Sourcing:** All state changes are stored as a sequence of events in Kafka; current state derived by replaying events
- **CQRS:** Command Query Responsibility Segregation — commands write to Kafka; queries read from materialized views (KTable)
- **Saga Pattern:** Distributed transactions using choreography (events) or orchestration (commands) across microservices
- **Outbox Pattern:** Write to local DB + outbox table atomically; a Kafka connector reads outbox and publishes events (avoids dual-write)
- **Fan-out:** Multiple consumer groups independently read the same topic for different processing pipelines
- **Dead Letter Queue (DLQ):** Failed messages are written to a separate DLQ topic for manual inspection/retry

## Q31. What are common Kafka pitfalls and how to avoid them?

**Answer:**

- **Uncontrolled rebalancing:** Use static group membership (group.instance.id) and tune session.timeout.ms and max.poll.interval.ms
- **Consumer lag buildup:** Monitor lag metrics; increase parallelism (more partitions + more consumers); optimize processing logic
- **Duplicate processing:** Use idempotent consumers (check record ID before processing) or enable exactly-once semantics
- **Large messages:** Kafka is optimized for small-medium messages (<1MB); store large payloads in S3/blob store and send reference in Kafka
- **Hot partitions:** Poor key choice (e.g., all messages with same key) overloads one partition; use high-cardinality keys
- **Not monitoring consumer lag:** Always monitor records-lag-max; set alerts when lag grows beyond threshold
- **Incorrect offset management:** Committing offsets BEFORE processing causes data loss; process THEN commit

## Q32. How do you implement a Dead Letter Queue (DLQ) pattern in Kafka?

**Answer:**

A DLQ pattern routes messages that fail processing to a separate topic for later inspection or reprocessing.

```
while (true) {
    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        try {
            processRecord(record);
```

```
        } catch (Exception e) {
            // Send to Dead Letter Queue
            ProducerRecord<String, String> dlqRecord = new ProducerRecord<>(
                "orders-dlq", record.key(), record.value());
            dlqRecord.headers().add("error-message",
e.getMessage().getBytes());
            dlqRecord.headers().add("original-topic", "orders".getBytes());
            dlqRecord.headers().add("original-offset",
                Long.toString(record.offset()).getBytes());
            producer.send(dlqRecord);
        }
    }
    consumer.commitSync();
}
```

# 12. Scenario-Based Interview Questions

### ⭐ FREQUENTLY ASKED Q33. How would you design a Kafka-based order processing system?

**Answer:**

High-level design:

- **Topics:** orders (raw), orders-validated, orders-fulfilled, orders-dlq
- **Partitioning:** Partition by order-id or customer-id for locality; use 12–24 partitions for scalability
- **Producers:** Order service publishes to orders topic with acks=all, idempotence enabled
- **Validation consumer:** Consumer group validates orders; writes to orders-validated or orders-dlq
- **Fulfillment consumer:** Separate consumer group reads orders-validated; triggers warehouse/payment
- **Monitoring:** Alert on orders-dlq lag and consumer lag on each stage
- **Schema:** Avro schemas with Schema Registry for type safety and evolution

### Q34. How would you handle message ordering in Kafka across partitions?

**Answer:**

Kafka only guarantees order within a single partition. Strategies to handle ordering:

- **Single partition (simplest):** Use one partition — total ordering guaranteed but zero parallelism
- **Key-based routing:** Ensure all related messages (same order ID) go to the same partition by using the order ID as the message key
- **Application-level sequencing:** Embed a sequence number in the message and reorder in the consumer
- **Kafka Streams windowing:** Use time-based windows to collect and sort events before processing

 **Note:** *For most systems, per-entity ordering (same customer or order) is sufficient. Use the entity ID as the Kafka message key to guarantee ordering for all events related to that entity.*

### Q35. How would you ensure zero data loss in Kafka?

**Answer:**

Configuration checklist for zero data loss:

- **Producer:** acks=all, enable.idempotence=true, retries=MAX_INT, max.in.flight.requests.per.connection=5
- **Broker:** replication.factor>=3, min.insync.replicas=2, unclean.leader.election.enable=false
- **Consumer:** enable.auto.commit=false, process-then-commit pattern
- **Topic:** retention.bytes and retention.ms configured adequately
- **Monitoring:** Alert on under-replicated partitions and broker failures immediately

# CHEAT SHEET: Quick Reference

## Key Configuration Cheat Sheet

### Producer Configs

| Property | Key Values | Use Case |
|---|---|---|
| acks | 0 / 1 / all | 0=fire&forget, 1=leader, all=durability |
| enable.idempotence | true/false | Prevent duplicate sends on retry |
| transactional.id | unique string | Enable exactly-once across topics |
| linger.ms | 0–100ms | Latency vs throughput trade-off |
| batch.size | 16384–131072 | Larger = better throughput |
| compression.type | none/gzip/snappy/lz4/zstd | LZ4 for speed, ZSTD for best ratio |
| retries | 0–2147483647 | MAX_INT with idempotence |
| max.in.flight.requests | 1–5 | 1 for strict ordering, 5 with idempotence |

### Consumer Configs

| Property | Key Values | Use Case |
|---|---|---|
| group.id | string | Identifies the consumer group |
| auto.offset.reset | earliest/latest/none | Where to start reading (no committed offset) |
| enable.auto.commit | true/false | false = manual offset control |
| max.poll.records | 1–500 | Lower for slow processing |
| max.poll.interval.ms | 300000ms | Increase for long processing tasks |
| session.timeout.ms | 45000ms | Increase to avoid false rebalances |
| isolation.level | read_uncommitted/read_committed | read_committed for EOS consumers |
| fetch.min.bytes | 1–65536 | Increase for throughput |

### Broker / Topic Configs

| Property | Recommended (Prod) | Description |
|---|---|---|
| replication.factor | 3 | Data durability across broker failures |
| min.insync.replicas | 2 | Minimum ISR for writes (with acks=all) |
| unclean.leader.election | false | Prevent data loss on leader election |
| retention.ms | 604800000 (7d) | How long to keep messages |
| cleanup.policy | delete / compact | delete=time-based, compact=key-based |
| num.partitions | topic-specific | Parallelism unit; cannot decrease |
| log.segment.bytes | 1073741824 (1GB) | Segment size before rollover |

## Kafka CLI Quick Reference

```
# List topics
kafka-topics.sh --bootstrap-server localhost:9092 --list
```

```
# Create topic
kafka-topics.sh --bootstrap-server localhost:9092 --create --topic my-topic
--partitions 3 --replication-factor 2
```

```
# Describe topic
kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic my-
topic
```

```
# Produce messages
kafka-console-producer.sh --bootstrap-server localhost:9092 --topic my-topic
```

```
# Consume messages (from beginning)
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my-topic
--from-beginning
```

```
# Check consumer group lag
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe
--group my-group
```

```
# Reset consumer offset to beginning
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group my-group
--topic my-topic --reset-offsets --to-earliest --execute
```

## Frequently Asked Questions Summary

| # | Question (Frequently Asked) | Key Answer |
|---|---|---|
| Q 1 | What is Kafka? | Distributed event streaming platform — durable, scalable, high-throughput pub/sub log |
| Q 2 | Core components? | Broker, Topic, Partition, Offset, Producer, Consumer, Consumer Group, ZooKeeper/KRaft |
| Q 4 | How partitioning works? | Key-based (murmur2 hash) or round-robin; ordering guaranteed per partition only |
| Q 5 | What is an offset? | Sequential ID per record within a partition; consumer tracks position |
| Q 7 | How producer works? | Serialize → Partition → Accumulate batch → Background Sender → Ack |
| Q 8 | acks settings? | 0=no ack, 1=leader ack, all=all ISR ack (best durability) |
| Q 9 | Exactly-once? | enable.idempotence=true + transactions + consumer isolation=read_committed |
| Q 10 | Consumer group? | Each partition assigned to ONE consumer in group; multiple groups = broadcast |
| Q 11 | Rebalancing? | Triggered by join/leave/timeout; stop-the-world; use CooperativeStickyAssignor |
| Q 14 | Replication? | Leader handles R/W; followers copy; ISR = in-sync replicas |
| Q 15 | min.insync.replicas? | Min ISR that must ack (with acks=all); use RF=3, minISR=2 in prod |
| Q 20 | Kafka Streams? | Client library for stateful stream processing; KStream vs KTable |
| Q 25 | KRaft mode? | Replaces ZooKeeper with Raft-based metadata within Kafka itself (Kafka 3.3+) |

| | | |
|---|---|---|
| Q 26 | Delivery semantics? | At-most-once / At-least-once / Exactly-once |
| Q 33 | Design order system? | Multiple topics per stage, partition by entity key, monitor lag & DLQ |

## Kafka Delivery Guarantees at a Glance

| Goal | Producer Config | Broker Config | Consumer Config |
|---|---|---|---|
| No data loss | acks=all, retries=MAX_INT, enable.idempotence=true | RF>=3, min.insync.replicas=2, unclean=false | enable.auto.commit =false, process-then-commit |
| High throughput | acks=1, linger.ms=50, batch.size=65536, compression=lz4 | Adequate partitions | max.poll.records=1000, fetch.min.bytes=1024 |
| Exactly-once | enable.idempotence=true, transactional.id set | RF>=3, min.insync.replicas=2 | isolation.level=read_committed |
| Low latency | linger.ms=0, batch.size=1, acks=1 | Low network latency | fetch.min.bytes=1, max.poll.records=1 |

**Good luck with your Kafka interview! □ □**

*Remember: Understand the WHY behind each concept, not just the WHAT.*