Kafka Fundamentals: Complete Developer Guide

A comprehensive refresher on Apache Kafka fundamentals, designed for both beginners and experienced developers. This README covers core concepts, internal architecture, configuration, APIs, best practices, and real-world applications.

Table of Contents

📚 Core Concepts

Topics, Partitions & Offsets

Simple Explanation

- **Topic**: A logical channel or category for messages (like a folder for emails)
- **Partition**: A topic is split into ordered sequences called partitions (like pages in a book)
- **Offset**: A unique sequential ID for each message within a partition (like line numbers)

```
Topic: "user-events"
├── Partition 0: [msg0, msg1, msg2, msg3, ...]  ← offsets: 0,1,2,3...
├── Partition 1: [msg0, msg1, msg2, ...]        ← offsets: 0,1,2...
└── Partition 2: [msg0, msg1, msg2, msg3, ...]  ← offsets: 0,1,2,3...
```

Problem It Solves

- **Scalability**: Single topic can handle millions of messages by distributing across partitions
- **Parallel Processing**: Multiple consumers can read from different partitions simultaneously

- **Fault Tolerance**: Messages are replicated across multiple brokers

Internal Architecture

- **Commit Log**: Each partition is an immutable, append-only log
- **Segment Files**: Partitions are split into segments for efficient storage and deletion
- **Index Files**:
  - `.index`: Maps logical offset → byte position in log file
  - `.timeindex`: Maps timestamp → logical offset
  - `.log`: Contains actual message data

Data Structures

```
Partition Directory Structure:
my-topic-0/
├── 00000000000000000000.log       # Messages 0-999
├── 00000000000000000000.index     # Offset → byte position
├── 00000000000000000000.timeindex # Timestamp → offset
├── 00000000000000001000.log       # Messages 1000-1999 (active)
├── 00000000000000001000.index
├── 00000000000000001000.timeindex
└── leader-epoch-checkpoint        # Leader election history
```

Key Configuration Parameters

```
# Topic Creation
num.partitions=3                    # Default partitions (default: 1)
default.replication.factor=3        # Default replication (default: 1)

# Segment Rolling
log.segment.bytes=1073741824        # 1GB per segment (default)
log.roll.ms=604800000              # Roll every 7 days (default)

# Retention
log.retention.hours=168             # Keep data 7 days (default)
log.retention.bytes=-1              # No size limit (default)

# Indexing
log.index.interval.bytes=4096       # Add index entry every 4KB (default)
log.index.size.max.bytes=10485760   # Max index file size 10MB (default)
```

Trade-offs When Tuning

- **More Partitions**: ↑ Parallelism, ↓ End-to-end latency during failures
- **Larger Segments**: ↓ File handles, ↑ Deletion granularity
- **Shorter Retention**: ↓ Storage costs, ↓ Consumer flexibility

Producers, Consumers & Consumer Groups

Simple Explanation

- **Producer**: Application that sends messages to Kafka topics
- **Consumer**: Application that reads messages from Kafka topics
- **Consumer Group**: Group of consumers that work together to consume all partitions

Problem It Solves

- **Load Balancing**: Consumer groups automatically distribute partitions among consumers
- **Fault Tolerance**: If consumer fails, others pick up its partitions
- **Scalability**: Add consumers to process more data in parallel

Internal Architecture

```
Topic with 3 partitions, Consumer Group with 2 consumers:

Partition 0 ──────→ Consumer A
Partition 1 ──────→ Consumer A
Partition 2 ──────→ Consumer B

If Consumer A fails:
Partition 0 ──────→ Consumer B
Partition 1 ──────→ Consumer B
Partition 2 ──────→ Consumer B
```

Key Configuration Parameters

Producer Configs:

```
# Durability
acks=all                           # Wait for all replicas (strongest durability)
retries=2147483647                 # Retry forever (default)
enable.idempotence=true            # Prevent duplicates (default in 3.0+)

# Performance
batch.size=16384                   # Batch size in bytes (default)
linger.ms=0                        # Wait time to fill batch (default)
compression.type=none              # none, gzip, snappy, lz4, zstd

# Networking
request.timeout.ms=30000           # Request timeout (default)
max.in.flight.requests.per.connection=5  # Concurrent requests (default)
```

Consumer Configs:

```
# Consumer Group
group.id=my-consumer-group          # Consumer group ID (required)
group.protocol=consumer             # Use new protocol (Kafka 4.0+)

# Offset Management
enable.auto.commit=true             # Auto-commit offsets (default)
auto.commit.interval.ms=5000        # Commit frequency (default)
auto.offset.reset=latest            # Where to start if no offset (default)

# Fetching
fetch.min.bytes=1                   # Min bytes to fetch (default)
fetch.max.wait.ms=500               # Max wait for min bytes (default)
max.poll.records=500                # Max records per poll (default)
```

Trade-offs When Tuning

- **Producer batching**: ↑ Throughput, ↑ Latency
- **Strong durability (acks=all)**: ↑ Safety, ↓ Throughput
- **Auto-commit offsets**: ↑ Simplicity, ↓ Control over exactly-once processing

Brokers & Clusters

Simple Explanation

- **Broker**: A single Kafka server that stores data and serves client requests
- **Cluster**: Multiple brokers working together for scalability and fault tolerance
- **Leader/Follower**: Each partition has one leader (handles reads/writes) and followers (replicas)

Problem It Solves

- **High Availability**: If broker fails, others continue serving
- **Load Distribution**: Data and requests spread across multiple servers
- **Scalability**: Add brokers to handle more throughput

Internal Architecture

```
Kafka Cluster (3 brokers):

Broker 1 (Controller)     Broker 2                 Broker 3
├─ Topic-A P0 (Leader)   ├─ Topic-A P0 (Follower) ├─ Topic-A P0 (Follower)
├─ Topic-A P1 (Follower) ├─ Topic-A P1 (Leader)   ├─ Topic-A P1 (Follower)
└─ Topic-B P0 (Follower) └─ Topic-B P0 (Follower) └─ Topic-B P0 (Leader)
```

Key Configuration Parameters

```
# Broker Identity
broker.id=1                        # Unique broker ID in cluster

# Networking
listeners=PLAINTEXT://localhost:9092 # Listener endpoints
advertised.listeners=PLAINTEXT://localhost:9092  # Advertised to clients

# Log Directories
log.dirs=/var/kafka-logs           # Data storage directories

# Replication
default.replication.factor=3        # Default replicas per partition
min.insync.replicas=2              # Min replicas for ack=all

# Performance
num.network.threads=8              # Network request handler threads
num.io.threads=8                   # Disk I/O threads
socket.send.buffer.bytes=102400    # Socket send buffer
socket.receive.buffer.bytes=102400 # Socket receive buffer
```

Records (Messages)

Simple Explanation

A Kafka record consists of:

- **Key**: Optional identifier (affects partitioning)
- **Value**: The actual message payload
- **Headers**: Optional metadata (key-value pairs)
- **Timestamp**: When message was created/received

Problem It Solves

- **Flexible Data**: Support various data formats and metadata
- **Partitioning Strategy**: Key determines which partition message goes to
- **Message Routing**: Headers enable advanced routing and filtering

Internal Structure

```
Record Structure:
┌─────────────┬────────────┬─────────────┬──────────┬────────────┐
│ Timestamp   │ Key        │ Value       │ Headers  │ Offset     │
├─────────────┼────────────┼─────────────┼──────────┼────────────┤
│ 1693875600  │ "user-123" │ "login"     │ source:app│ 42         │
└─────────────┴────────────┴─────────────┴──────────┴────────────┘
```

🏗 Setup & Architecture

Kafka Broker Configuration

Essential Broker Settings

```
# Server Basics
broker.id=1
listeners=PLAINTEXT://0.0.0.0:9092
log.dirs=/var/kafka-logs

# Cluster Coordination (KRaft Mode - Kafka 3.3+)
process.roles=broker,controller     # This broker acts as both
node.id=1                          # Unique node ID
controller.quorum.voters=1@localhost:9093,2@localhost:9094,3@localhost:9095
metadata.log.dir=/var/kafka-metadata

# Replication & Durability
default.replication.factor=3
min.insync.replicas=2
unclean.leader.election.enable=false

# Performance Tuning
num.network.threads=8
num.io.threads=16
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
num.replica.fetchers=4

# Memory & GC
-Xms6g -Xmx6g                     # Heap size (typically 6GB)
-XX:+UseG1GC                      # G1 garbage collector
```

Trade-offs

- **More network threads**: ↑ Concurrent client handling, ↑ CPU usage
- **Larger socket buffers**: ↑ Throughput for large messages, ↑ Memory usage
- **Higher min.insync.replicas**: ↑ Durability, ↓ Availability during failures

ZooKeeper vs KRaft

ZooKeeper Mode (Legacy - Deprecated in 4.0)

Problems with ZooKeeper:

- Additional operational complexity (separate ZooKeeper cluster)
- Scalability bottleneck (~200K partitions per cluster)
- Split-brain scenarios during network partitions
- Slower metadata operations

```
# ZooKeeper Configuration (Legacy)
zookeeper.connect=zk1:2181,zk2:2181,zk3:2181
zookeeper.connection.timeout.ms=6000
```

KRaft Mode (Kafka 3.3+, Default in 4.0)

Advantages:

- No external dependencies
- Better scalability (2M+ partitions tested)
- Faster metadata operations
- Simplified operations
- Stronger consistency guarantees

```
# KRaft Configuration
process.roles=broker,controller                # Combined mode
node.id=1                                     # Unique across cluster
controller.quorum.voters=1@broker1:9093,2@broker2:9093,3@broker3:9093
metadata.log.dir=/var/kafka-metadata
```

Migration Process (ZooKeeper → KRaft)

1. **Prepare**: Ensure Kafka 3.4+ and clean shutdown
2. **Generate Metadata**: Use `kafka-storage.sh` to create cluster UUID
3. **Format Storage**: Format KRaft metadata logs
4. **Migrate**: Use dual-write mode during migration
5. **Switch**: Complete migration and remove ZooKeeper

Version Highlights:

- **Kafka 2.8**: KRaft early access
- **Kafka 3.3**: KRaft production ready
- **Kafka 4.0**: KRaft default, ZooKeeper deprecated

Cluster Metadata & Controller Election

Controller Role

The **Controller** is a special broker responsible for:

- Partition leader election
- Replica management
- Metadata distribution to other brokers
- Handling broker failures

KRaft Controller Election

```
KRaft Quorum (3 controllers):

Controller 1 (Leader)    Controller 2 (Follower)    Controller 3 (Follower)
      │                           │                          │
      ├─ Handles all writes ──────┼─ Replicates metadata ────┼─ Replicates
metadata
      ├─ Manages elections        │                          │
      └─ Sends heartbeats ────────┴─ Monitors leader ────────┴─ Monitors leader

If Controller 1 fails:
Controller 2 (New Leader)  Controller 3 (Follower)    Controller 1 (Down)
      │                           │                          │
      ├─ Takes over writes ───────┼─ Continues replication   │ [Disconnected]
      └─ Manages elections ───────┘                          │
```

Key Configuration

```
# KRaft Controller Settings
controller.quorum.voters=1@kafka1:9093,2@kafka2:9093,3@kafka3:9093
controller.quorum.election.timeout.ms=1000         # Election timeout
controller.quorum.fetch.timeout.ms=2000           # Follower fetch timeout
controller.quorum.retry.backoff.ms=20             # Retry interval

# Metadata Replication
metadata.log.segment.bytes=1048576                 # Metadata log segment size
metadata.log.retention.ms=604800000               # Keep metadata 7 days
```

💻 APIs & CLI Usage

Producer/Consumer/Admin APIs

Admin API Operations

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

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

# Describe Topic
kafka-topics.sh --bootstrap-server localhost:9092 \
  --describe --topic user-events

# Delete Topic
```

```
kafka-topics.sh --bootstrap-server localhost:9092 \
  --delete --topic user-events

# Alter Topic (add partitions)
kafka-topics.sh --bootstrap-server localhost:9092 \
  --alter --topic user-events --partitions 6
```

CLI Commands Reference

Consumer Group Management

```
# List Consumer Groups
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

# Describe Consumer Group
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --describe --group my-group

# Reset Offsets to Beginning
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --reset-offsets --group my-group --topic user-events --to-earliest

# Reset Offsets to Specific Offset
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --reset-offsets --group my-group --topic user-events:0 --to-offset 100

# Delete Consumer Group
kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
  --delete --group my-group
```

Console Producer/Consumer

```
# Console Producer
kafka-console-producer.sh --bootstrap-server localhost:9092 \
  --topic user-events --property "key.separator=:" \
  --property "parse.key=true"

# Console Consumer
kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic user-events --from-beginning \
  --property "print.key=true" \
  --property "key.separator=: "

# Consumer with Consumer Group
kafka-console-consumer.sh --bootstrap-server localhost:9092 \
  --topic user-events --group test-group
```

☕ Java Examples

Maven Dependencies

```xml
<dependencies>
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.6.0</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>2.0.9</version>
  </dependency>
</dependencies>
```

Producer Example

```java
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;
import java.util.Properties;

public class KafkaProducerExample {
    private static final String TOPIC = "user-events";
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";

    public static void main(String[] args) {
        // Producer configuration
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, 
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, 
StringSerializer.class);

        // Production settings
        props.put(ProducerConfig.ACKS_CONFIG, "all");              // Wait for all 
replicas
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE); // Retry 
forever
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);  // Prevent 
duplicates

        // Performance settings
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);         // 16KB 
batches
        props.put(ProducerConfig.LINGER_MS_CONFIG, 5);              // Wait 5ms for 
batch
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "gzip");  // Compress
```

```
data

        try (Producer<String, String> producer = new KafkaProducer<>(props)) {
            for (int i = 0; i < 100; i++) {
                String key = "user-" + i;
                String value = "login-event-" + i;

                ProducerRecord<String, String> record =
                    new ProducerRecord<>(TOPIC, key, value);

                // Synchronous send
                RecordMetadata metadata = producer.send(record).get();
                System.out.printf("Sent record to partition %d, offset %d%n",
                    metadata.partition(), metadata.offset());

                Thread.sleep(100);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Consumer Example

```
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.StringDeserializer;
import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

public class KafkaConsumerExample {
    private static final String TOPIC = "user-events";
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final String GROUP_ID = "user-events-consumers";

    public static void main(String[] args) {
        // Consumer configuration
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, GROUP_ID);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Consumer behavior
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  // Start
from beginning
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);      // Manual
offset commits
```

```
        // Performance settings
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024);          // Min 1KB per fetch
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 1000);        // Wait max 1s
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);          // Max 100 records/poll

        try (Consumer<String, String> consumer = new KafkaConsumer<>(props)) {
            consumer.subscribe(Arrays.asList(TOPIC));

            while (true) {
                ConsumerRecords<String, String> records =
                    consumer.poll(Duration.ofMillis(1000));

                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("Consumed record: key=%s, value=%s, " +
                        "partition=%d, offset=%d%n",
                        record.key(), record.value(),
                        record.partition(), record.offset());

                    // Process the record here
                    processRecord(record);
                }

                // Commit offsets after processing
                if (!records.isEmpty()) {
                    consumer.commitSync();
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void processRecord(ConsumerRecord<String, String> record) {
        // Your business logic here
        System.out.println("Processing: " + record.value());
    }
}
```

Kafka Streams Example

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Produced;
```

```java
import java.util.Properties;
import java.util.concurrent.CountDownLatch;

public class KafkaStreamsExample {
    private static final String INPUT_TOPIC = "user-events";
    private static final String OUTPUT_TOPIC = "user-event-counts";

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "user-events-processor");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

        StreamsBuilder builder = new StreamsBuilder();

        // Read stream from input topic
        KStream<String, String> events = builder.stream(INPUT_TOPIC);

        // Count events by user (key)
        KTable<String, Long> eventCounts = events
            .groupByKey()
            .count();

        // Output to result topic
        eventCounts.toStream()
            .to(OUTPUT_TOPIC, Produced.with(Serdes.String(), Serdes.Long()));

        // Start the streams application
        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        CountDownLatch latch = new CountDownLatch(1);

        // Graceful shutdown
        Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}
```

⚖️ Comparisons & Trade-offs

Consumer Groups vs Kafka Streams

Aspect	Consumer Groups	Kafka Streams
Use Case	Simple consume & process	Complex stream processing
State Management	Stateless (typically)	Built-in state stores
Fault Tolerance	Manual checkpoint	Automatic state recovery
Processing Model	Record-at-a-time	Stream operations (map, filter, join)
Scalability	Scale by adding consumers	Scale by adding stream threads
Complexity	Low	Medium to High
Latency	Low	Low to Medium
Exactly-once	Manual implementation	Built-in support

Performance vs Reliability Trade-offs

Configuration	Performance Impact	Reliability Impact
`acks=0`	⭐⭐⭐ Highest throughput	❌ Messages may be lost
`acks=1`	⭐⭐ Good throughput	⚠️ Leader failure may lose data
`acks=all`	⭐ Lower throughput	✅ Highest durability
`retries=0`	⭐⭐⭐ No retry overhead	❌ Transient failures cause loss
`retries=MAX`	⭐ Retry overhead	✅ Handles transient failures
`enable.idempotence=true`	⭐⭐ Slight overhead	✅ Prevents duplicates

Ordering Guarantees

Scenario	Ordering Guarantee
Single partition	✅ Total order within partition
Multiple partitions	❌ No order across partitions
`max.in.flight.requests=1`	✅ Strict ordering per partition
`enable.idempotence=true`	✅ Order preserved with retries
Consumer group with multiple consumers	✅ Order per partition, not across partitions

---

🚨 Common Pitfalls & Best Practices

1. Configuration Mistakes

❌ Setting `request.timeout.ms` Too Low

```
# DON'T
request.timeout.ms=5000  # Too aggressive, may cause cascading failures
```

```
# DO
request.timeout.ms=30000  # Default - allows brokers to handle load
```

Why: Low timeouts can create retry storms during broker load, making problems worse.

❌ Misunderstanding Producer Retries

```
// DON'T - May lose messages
props.put(ProducerConfig.RETRIES_CONFIG, 0);
props.put(ProducerConfig.ACKS_CONFIG, "0");
```

```
// DO - Safe defaults
props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
props.put(ProducerConfig.ACKS_CONFIG, "all");
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
```

❌ Ordering Issues with Retries

```
// DON'T - Can reorder messages
props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);
props.put(ProducerConfig.RETRIES_CONFIG, 10);
```

```
// DO - Maintains order
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true); // Handles ordering
// OR
props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);
```

2. Operational Issues

❌ Going Overboard with Partitions

```
# DON'T - Too many partitions
kafka-topics.sh --create --topic events --partitions 1000 --replication-factor 3
```

Problems:

- Longer failover times
- More memory usage per broker
- "Too many open files" errors

Formula for partition count:

```
partitions >= max(target_throughput/producer_throughput,
target_throughput/consumer_throughput)

Example: 200MB/s target, 50MB/s producer, 25MB/s consumer
partitions >= max(200/50, 200/25) = max(4, 8) = 8 partitions
```

❌ Setting `segment.ms` Too Low

```
# DON'T - Creates too many small files
log.segment.ms=300000  # 5 minutes
```

```
# DO - Default is usually fine
log.segment.ms=604800000  # 7 days (default)
```

3. Monitoring Neglect

Essential JMX Metrics to Monitor

```
# Broker Request Latency
kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Produce
kafka.network:type=RequestMetrics,name=TotalTimeMs,request=FetchConsumer

# Under Replicated Partitions (should be 0)
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions

# Request Queue Size
kafka.network:type=RequestChannel,name=RequestQueueSize

# Log Size Growth
kafka.log:type=LogSize,name=Size,topic=*,partition=*
```

```
# Consumer Lag
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=*
```

4. Consumer Issues

❌ Not Handling Rebalancing Properly

```java
// DON'T - No rebalance handling
consumer.subscribe(Arrays.asList("events"));
while (true) {
    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));
    // Process records...
    consumer.commitSync(); // May fail during rebalance
}
```

```java
// DO - Handle rebalances
consumer.subscribe(Arrays.asList("events"), new ConsumerRebalanceListener() {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        // Commit current offsets before losing partitions
        consumer.commitSync();
        System.out.println("Partitions revoked: " + partitions);
    }

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        System.out.println("Partitions assigned: " + partitions);
    }
});
```

5. Message Size Issues

❌ Large Messages Without Proper Configuration

```properties
# DON'T - Default limits may reject large messages
message.max.bytes=1000012          # ~1MB (default)
replica.fetch.max.bytes=1048576    # 1MB (default)
```

```properties
# DO - Increase limits for large messages
message.max.bytes=10485760         # 10MB
replica.fetch.max.bytes=10485760   # 10MB
fetch.max.bytes=10485760           # 10MB (consumer)
max.request.size=10485760          # 10MB (producer)
```

Best Practices Summary

☑ Producer Best Practices

1. **Enable idempotence** for exactly-once semantics
2. **Use appropriate** `acks` setting for your durability needs
3. **Batch messages** with `linger.ms` for throughput
4. **Handle exceptions** and implement retry logic
5. **Monitor producer metrics** (latency, error rate)

☑ Consumer Best Practices

1. **Use consumer groups** for automatic load balancing
2. **Handle rebalances** gracefully
3. **Commit offsets** after processing messages
4. **Set appropriate** `session.timeout.ms` and `heartbeat.interval.ms`
5. **Monitor consumer lag**

☑ Operational Best Practices

1. **Monitor key broker metrics** (latency, under-replicated partitions)
2. **Plan partition count** based on throughput requirements
3. **Use appropriate replication factor** (typically 3)
4. **Implement proper logging** and alerting
5. **Regular capacity planning** and performance testing

🌍 Real-World Use Cases

1. Log Aggregation

Scenario: Collecting logs from multiple services into centralized system

```java
// Microservice log producer
public class LogProducer {
    private final Producer<String, String> producer;

    public void logEvent(String service, String message) {
        ProducerRecord<String, String> record =
            new ProducerRecord<>("app-logs", service, message);
        producer.send(record);
    }
}

// Log aggregation consumer
public class LogAggregator {
    public void processLogs() {
        consumer.subscribe(Arrays.asList("app-logs"));
```

```
        while (true) {
            ConsumerRecords<String, String> records = 
consumer.poll(Duration.ofMillis(1000));
            for (ConsumerRecord<String, String> record : records) {
                // Send to Elasticsearch, store in database, etc.
                storeLog(record.key(), record.value(), record.timestamp());
            }
        }
    }
}
```

Why Kafka:

- High throughput for log volume
- Multiple consumers (Elasticsearch, monitoring, analytics)
- Retention for historical analysis

2. Change Data Capture (CDC)

Scenario: Capturing database changes for downstream systems

```
// Database change event
public class OrderChangeEvent {
    private String orderId;
    private String operation; // INSERT, UPDATE, DELETE
    private String beforeState;
    private String afterState;
    private long timestamp;
}

// CDC Producer (from database trigger/log)
producer.send(new ProducerRecord<>("order-changes", order.getId(), changeEvent));

// Downstream consumers
// Consumer 1: Update search index
// Consumer 2: Send notifications
// Consumer 3: Update analytics warehouse
```

Why Kafka:

- Preserves order of changes per entity (partition by ID)
- Multiple downstream systems can consume same events
- Replay capability for rebuilding systems

3. Event Sourcing

Scenario: Storing all state changes as events

```java
public class AccountEventSourcing {
    public void processAccountCommand(String accountId, AccountCommand command) {
        // Validate command against current state
        List<AccountEvent> events = validateAndCreateEvents(command);

        // Store events in Kafka
        for (AccountEvent event : events) {
            producer.send(new ProducerRecord<>("account-events", accountId,
event));
        }
    }

    public Account rebuildAccountState(String accountId) {
        // Replay all events for this account
        consumer.assign(Arrays.asList(new TopicPartition("account-events",
getPartition(accountId))));
        consumer.seekToBeginning(consumer.assignment());

        Account account = new Account(accountId);
        ConsumerRecords<String, AccountEvent> records =
consumer.poll(Duration.ofMillis(10000));

        for (ConsumerRecord<String, AccountEvent> record : records) {
            if (record.key().equals(accountId)) {
                account.apply(record.value()); // Apply event to rebuild state
            }
        }
        return account;
    }
}
```

Why Kafka:

- Immutable event log
- Complete audit trail
- Can rebuild state at any point in time

4. Real-time Analytics

Scenario: Processing streaming data for dashboards and alerts

```java
// Using Kafka Streams for real-time aggregation
StreamsBuilder builder = new StreamsBuilder();

KStream<String, OrderEvent> orders = builder.stream("orders");

// Real-time revenue calculation
orders
    .groupBy((key, order) -> order.getProductCategory())
    .windowedBy(TimeWindows.of(Duration.ofMinutes(5)))
```

```
    .aggregate(
        () -> 0.0,
        (key, order, aggregate) -> aggregate + order.getAmount(),
        Materialized.as("revenue-by-category")
    )
    .toStream()
    .to("revenue-alerts");
```

Why Kafka:

- Low latency stream processing
- Stateful operations (windowing, aggregation)
- Scalable processing with Kafka Streams

5. Microservices Communication

Scenario: Async communication between services

```
// Order service publishes events
public class OrderService {
    public void createOrder(Order order) {
        // Save order to database
        orderRepository.save(order);

        // Publish event for other services
        OrderCreatedEvent event = new OrderCreatedEvent(order.getId(),
order.getCustomerId());
        producer.send(new ProducerRecord<>("order-events", order.getId(), event));
    }
}

// Inventory service consumes events
public class InventoryService {
    @EventHandler
    public void handleOrderCreated(OrderCreatedEvent event) {
        // Reserve inventory for the order
        inventoryRepository.reserve(event.getOrderId());
    }
}

// Notification service consumes events
public class NotificationService {
    @EventHandler
    public void handleOrderCreated(OrderCreatedEvent event) {
        // Send confirmation email
        emailService.sendOrderConfirmation(event.getCustomerId());
    }
}
```

Why Kafka:

- Decouples services
- Multiple services can react to same events
- Reliable message delivery

📈 Version Highlights

Kafka 4.0 (September 2025) - Current Latest

- ✨ **KRaft by default** - ZooKeeper removed entirely
- ✨ **New consumer protocol** (KIP-848) - Faster rebalancing
- ✨ **Queues for Kafka** (KIP-932) - Point-to-point messaging support
- ✨ **Java 11 minimum** for clients, Java 17 for brokers
- ✨ **Eligible Leader Replicas** (KIP-966) - Better leader election

Kafka 3.x Series (2021-2024)

- **3.6** (Oct 2023): Tiered storage improvements, KIP-405
- **3.5** (Jun 2023): KRaft production-ready improvements
- **3.4** (Feb 2023): KRaft metadata shell, group protocol improvements
- **3.3** (Oct 2022): KRaft production ready, self-balancing clusters
- **3.2** (May 2022): Kafka Streams improvements, KIP-768
- **3.1** (Jan 2022): Raft improvements, foreign key joins in Streams
- **3.0** (Sep 2021): KRaft early access, Java 8 deprecation

Kafka 2.x Series (2018-2021)

- **2.8** (Jan 2021): KRaft early access mode
- **2.7** (Dec 2020): Incremental cooperative rebalancing
- **2.6** (Aug 2020): TLS 1.3 support
- **2.5** (Apr 2020): Co-groups in Kafka Streams, TLS improvements
- **2.4** (Dec 2019): Foreign key joins, consumer improvements
- **2.3** (Jun 2019): Kafka Streams improvements
- **2.2** (Mar 2019): Incremental cooperative rebalancing
- **2.1** (Nov 2018): Zstandard compression
- **2.0** (Jul 2018): Kafka Streams improvements, security enhancements

Key Features by Version

Version	Key Features
4.0	Default KRaft, Queues, New consumer protocol, Java 11/17
3.3	KRaft production ready
2.8	KRaft early access
2.4	Foreign key joins in Streams
2.1	Zstandard compression

Version	Key Features
1.0	Exactly-once semantics
0.11	Idempotent producer
0.10	Kafka Streams, timestamps
0.9	Security (SSL/SASL), Kafka Connect

🔗 Additional Resources

📖 Official Documentation

- Apache Kafka Documentation
- Confluent Kafka Tutorials
- Kafka Improvement Proposals (KIPs)

🎓 Learning Resources

- Kafka: The Definitive Guide - O'Reilly Book
- Confluent Developer - Free courses
- Apache Kafka on GitHub - Source code

🔧 Tools & Monitoring

- Kafka Manager - Cluster management
- Kafdrop - Web UI for Kafka
- Conduktor - Desktop GUI for Kafka
- JMX Monitoring - Built-in metrics

🏗 Architecture Diagrams

```
Visual Resources:
├── Topic → Partition → Offset hierarchy
├── Producer → Broker → Consumer flow
├── Consumer group rebalancing
├── KRaft controller election process
├── Replication and ISR management
└── Stream processing topologies
```

🐛 Troubleshooting Guides

- Common Issues & Solutions
- Performance Tuning Guide
- Security Configuration

🤝 Contributing

Found an error or want to add more examples? Feel free to contribute:

1. Fork this repository
2. Create a feature branch (`git checkout -b feature/amazing-feature`)
3. Commit your changes (`git commit -m 'Add some amazing feature'`)
4. Push to the branch (`git push origin feature/amazing-feature`)
5. Open a Pull Request

📄 License

This documentation is licensed under Apache License 2.0 - same as Apache Kafka.

Last Updated: September 2025
Kafka Version: 4.0.0
Compatibility: Java 11+ (clients), Java 17+ (brokers)

> 💡 **Pro Tip**: Bookmark this README and keep it handy during your Kafka development journey. The examples and configurations shown here are production-tested patterns used by thousands of organizations worldwide.