Kafka Consumers: Complete Developer Guide

A comprehensive refresher on Apache Kafka Consumers, designed for both beginners and experienced developers. This README covers consumer basics, consumer groups, offset management, and real-world applications with detailed Java examples.

Table of Contents

Consumer API

KafkaConsumer Basics

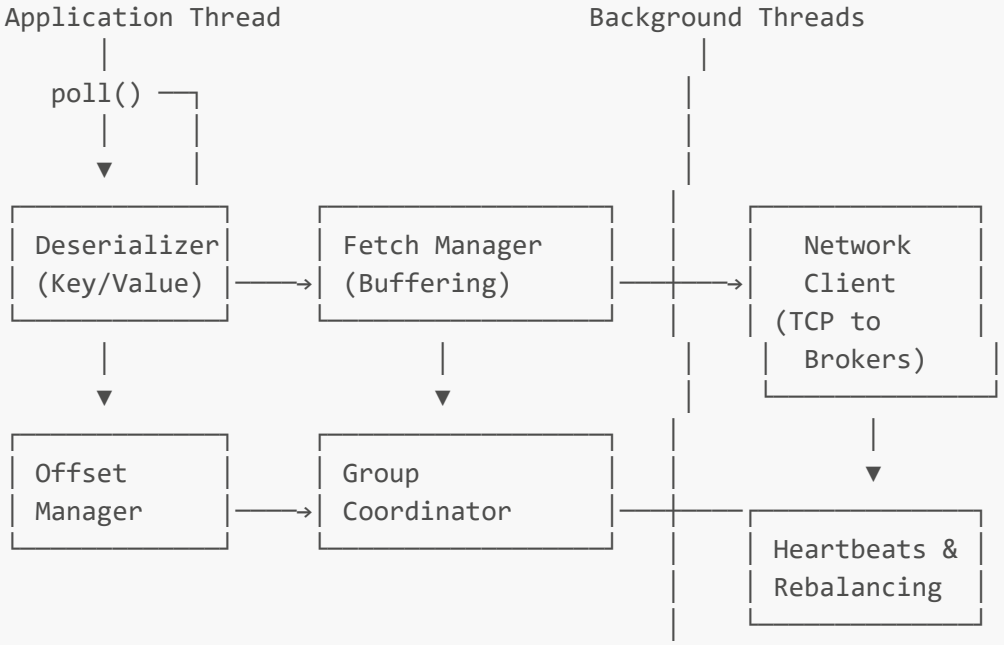
Simple Explanation

The KafkaConsumer is the client API that applications use to read records from Kafka topics. It provides high-level abstractions for subscribing to topics, managing offsets, and handling rebalancing in consumer groups.

Problem It Solves

- **Scalable Data Consumption**: Multiple consumers can process data in parallel
- **Fault Tolerance**: Automatic failover and recovery
- **Order Preservation**: Maintains order within partitions
- **Offset Management**: Tracks what messages have been processed

Internal Architecture



Basic Consumer Configuration

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

public class BasicConsumerConfig {

    public static Properties createBasicConsumerConfig() {
        Properties props = new Properties();

        // Connection settings
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "my-consumer-group");
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, "my-consumer-client");

        // Deserialization
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            StringDeserializer.class.getName());

        // Offset behavior
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 1000);

        // Performance settings
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024);          // Min
1KB per fetch
```

```
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 500);          // Max
wait 500ms
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);           // Max
records per poll

        return props;
    }
}
```

Poll Loop Implementation

Basic Poll Loop Pattern

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

public class BasicConsumerExample {

    private static final String TOPIC = "user-events";
    private volatile boolean running = true;

    public static void main(String[] args) {
        BasicConsumerExample consumer = new BasicConsumerExample();
        consumer.consume();
    }

    public void consume() {
        Properties props = BasicConsumerConfig.createBasicConsumerConfig();

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
{

            // Subscribe to topic(s)
            consumer.subscribe(Arrays.asList(TOPIC));

            // Main consumption loop
            while (running) {
                try {
                    // Poll for records with timeout
                    ConsumerRecords<String, String> records =
                        consumer.poll(Duration.ofMillis(1000));

                    // Process each record
                    for (ConsumerRecord<String, String> record : records) {
                        processRecord(record);
                    }

                    // Commit offsets (if manual commit is enabled)
```

```
                    // consumer.commitSync();

                } catch (Exception e) {
                    System.err.println("Error during consumption: " +
e.getMessage());
                    // Handle error - might want to continue or break based on
error type
                }
            }

        } catch (Exception e) {
            System.err.println("Consumer failed: " + e.getMessage());
        }
    }

    private void processRecord(ConsumerRecord<String, String> record) {
        System.out.printf("Consumed record: topic=%s, partition=%d, offset=%d, " +
            "key=%s, value=%s, timestamp=%d%n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value(), record.timestamp());

        // Your business logic here
        try {
            // Simulate processing time
            Thread.sleep(10);

            // Process the record
            handleUserEvent(record.key(), record.value());

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            running = false;
        } catch (Exception e) {
            System.err.println("Failed to process record: " + e.getMessage());
            // Decide whether to skip, retry, or send to DLQ
        }
    }

    private void handleUserEvent(String userId, String eventData) {
        // Business logic implementation
        System.out.println("Processing event for user: " + userId);
    }

    public void shutdown() {
        running = false;
    }
}
```

Advanced Poll Loop with Error Handling

```
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.errors.*;
import java.time.Duration;
import java.util.Arrays;
import java.util.concurrent.atomic.AtomicBoolean;

public class RobustConsumerExample {

    private final AtomicBoolean running = new AtomicBoolean(true);
    private final String topic;
    private final Properties config;

    public RobustConsumerExample(String topic, Properties config) {
        this.topic = topic;
        this.config = config;
    }

    public void consume() {
        while (running.get()) {
            try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>
(config)) {

                consumer.subscribe(Arrays.asList(topic), new RebalanceListener());

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

                        if (records.isEmpty()) {
                            continue;
                        }

                        processRecords(records);
                        commitOffsets(consumer, records);

                    } catch (WakeupException e) {
                        // Shutdown signal
                        System.out.println("Consumer wakeup called");
                        break;

                    } catch (CommitFailedException e) {
                        // Commit failed - usually during rebalance
                        System.err.println("Commit failed: " + e.getMessage());
                        // Continue - offsets will be reset by rebalance

                    } catch (AuthorizationException e) {
                        // Not authorized - fatal error
                        System.err.println("Not authorized: " + e.getMessage());
                        break;

                    } catch (InvalidOffsetException e) {
                        // Invalid offset - reset
```

```
                        System.err.println("Invalid offset: " + e.getMessage());
                        consumer.seekToBeginning(e.partitions());

                    } catch (RetriableException e) {
                        // Retriable error - log and continue
                        System.err.println("Retriable error: " + e.getMessage());

                    } catch (Exception e) {
                        // Unexpected error
                        System.err.println("Unexpected error: " + e.getMessage());
                        e.printStackTrace();
                        break;
                    }
                }

            } catch (Exception e) {
                System.err.println("Consumer initialization failed: " +
e.getMessage());

                if (running.get()) {
                    // Wait before retry
                    try {
                        Thread.sleep(5000);
                    } catch (InterruptedException ie) {
                        Thread.currentThread().interrupt();
                        break;
                    }
                }
            }
        }

        System.out.println("Consumer shut down");
    }

    private void processRecords(ConsumerRecords<String, String> records) {
        for (ConsumerRecord<String, String> record : records) {
            try {
                // Process individual record
                processRecord(record);

            } catch (Exception e) {
                System.err.printf("Failed to process record at offset %d: %s%n",
                    record.offset(), e.getMessage());

                // Decide on error handling strategy
                handleProcessingError(record, e);
            }
        }
    }

    private void processRecord(ConsumerRecord<String, String> record) {
        // Your business logic here
        System.out.printf("Processing: %s:%s%n", record.key(), record.value());
    }
```

```
    private void handleProcessingError(ConsumerRecord<String, String> record,
Exception e) {
        // Error handling strategies:

        if (e instanceof IllegalArgumentException) {
            // Invalid data - skip record
            System.out.println("Skipping invalid record");

        } else if (e instanceof java.net.ConnectException) {
            // External service unavailable - could retry or send to DLQ
            System.out.println("External service unavailable - sending to DLQ");
            sendToDeadLetterQueue(record);

        } else {
            // Unknown error - could fail fast or continue
            System.err.println("Unknown processing error - continuing");
        }
    }

    private void sendToDeadLetterQueue(ConsumerRecord<String, String> record) {
        // Implementation to send failed records to DLQ
        System.out.println("Sending to DLQ: " + record.key());
    }

    private void commitOffsets(KafkaConsumer<String, String> consumer,
                               ConsumerRecords<String, String> records) {
        try {
            consumer.commitSync();
        } catch (CommitFailedException e) {
            System.err.println("Commit failed: " + e.getMessage());
        }
    }

    public void shutdown() {
        running.set(false);
    }

    // Rebalance listener for cleanup
    private class RebalanceListener implements ConsumerRebalanceListener {
        @Override
        public void
onPartitionsRevoked(java.util.Collection<org.apache.kafka.common.TopicPartition>
partitions) {
            System.out.println("Partitions revoked: " + partitions);
            // Commit current offsets before losing partitions
        }

        @Override
        public void
onPartitionsAssigned(java.util.Collection<org.apache.kafka.common.TopicPartition>
partitions) {
            System.out.println("Partitions assigned: " + partitions);
            // Initialize any partition-specific state
```

```
        }
    }
}
```

Auto vs Manual Commit

Auto Commit (Default)

```
public class AutoCommitConsumerExample {

    public static Properties createAutoCommitConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "auto-commit-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Auto commit settings (default)
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true);
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 5000); // Every 5
seconds

        return props;
    }

    public static void main(String[] args) {
        Properties props = createAutoCommitConfig();

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

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

                for (ConsumerRecord<String, String> record : records) {
                    // Process record
                    processOrder(record.value());

                    // Offsets committed automatically every 5 seconds
                    // No manual commit needed
                }

                // Note: If processing fails after auto-commit,
                // messages might be lost on restart
            }
        }
    }
```

```
    private static void processOrder(String orderData) {
        System.out.println("Processing order: " + orderData);
        // Business logic here
    }
}
```

Auto Commit Trade-offs:

- ☑ **Simplicity**: No manual offset management
- ☑ **Performance**: No commit overhead in processing loop
- ❌ **At-most-once**: Messages can be lost if processing fails after commit
- ❌ **Less Control**: Cannot commit based on processing success

Manual Commit (Recommended for Critical Data)

```
public class ManualCommitConsumerExample {

    public static Properties createManualCommitConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "manual-commit-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Manual commit settings
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);  // Disable
auto commit

        // Reduce fetch size for better control
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);

        return props;
    }

    public static void main(String[] args) {
        Properties props = createManualCommitConfig();

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
{
            consumer.subscribe(Arrays.asList("financial-transactions"));

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

                if (records.isEmpty()) {
                    continue;
                }
```

```
                // Process all records in batch
                boolean allProcessedSuccessfully = true;
                for (ConsumerRecord<String, String> record : records) {
                    try {
                        processFinancialTransaction(record);
                    } catch (Exception e) {
                        System.err.println("Processing failed for record: " +
record.offset());
                        allProcessedSuccessfully = false;
                        break; // Stop processing this batch
                    }
                }

                // Only commit if all records processed successfully
                if (allProcessedSuccessfully) {
                    try {
                        // Synchronous commit - blocks until complete
                        consumer.commitSync();
                        System.out.println("Batch committed successfully");

                    } catch (CommitFailedException e) {
                        System.err.println("Commit failed: " + e.getMessage());
                        // Handle commit failure - maybe reprocess
                    }
                } else {
                    System.out.println("Skipping commit due to processing
failures");
                    // Could implement retry logic here
                }
            }
        }
    }

    private static void processFinancialTransaction(ConsumerRecord<String, String>
record) {
        System.out.printf("Processing transaction: %s at offset %d%n",
            record.value(), record.offset());

        // Simulate processing that could fail
        if (record.value().contains("invalid")) {
            throw new IllegalArgumentException("Invalid transaction data");
        }

        // Actual business logic here
        // Save to database, call external service, etc.
    }
}
```

Asynchronous Manual Commit

```
public class AsyncCommitConsumerExample {

    public static void main(String[] args) {
        Properties props = ManualCommitConsumerExample.createManualCommitConfig();

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
{
            consumer.subscribe(Arrays.asList("user-events"));

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

                for (ConsumerRecord<String, String> record : records) {
                    processUserEvent(record);
                }

                // Asynchronous commit - doesn't block
                consumer.commitAsync(new OffsetCommitCallback() {
                    @Override
                    public void
onComplete(java.util.Map<org.apache.kafka.common.TopicPartition,

org.apache.kafka.clients.consumer.OffsetAndMetadata> offsets,
                                           Exception exception) {
                        if (exception != null) {
                            System.err.println("Commit failed: " +
exception.getMessage());
                            // Could implement retry logic
                        } else {
                            System.out.println("Async commit successful");
                        }
                    }
                });
            }
        }
    }

    private static void processUserEvent(ConsumerRecord<String, String> record) {
        System.out.println("Processing user event: " + record.value());
    }
}
```

Fine-grained Offset Control

```
import java.util.HashMap;
import java.util.Map;

public class FinegrainedOffsetConsumerExample {
```

```
    public static void main(String[] args) {
        Properties props = ManualCommitConsumerExample.createManualCommitConfig();

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

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

                // Track offsets per partition
                Map<TopicPartition, OffsetAndMetadata> currentOffsets = new
HashMap<>();

                for (ConsumerRecord<String, String> record : records) {
                    try {
                        processOrder(record);

                        // Track the offset for this partition
                        currentOffsets.put(
                            new TopicPartition(record.topic(),
record.partition()),
                            new OffsetAndMetadata(record.offset() + 1, "Processed
successfully")
                        );

                    } catch (Exception e) {
                        System.err.printf("Failed to process record at offset %d:
%s%n",
                            record.offset(), e.getMessage());

                        // Don't update offset for failed record
                        break;
                    }
                }

                // Commit specific offsets
                if (!currentOffsets.isEmpty()) {
                    try {
                        consumer.commitSync(currentOffsets);
                        System.out.println("Committed offsets: " +
currentOffsets);

                    } catch (CommitFailedException e) {
                        System.err.println("Offset commit failed: " +
e.getMessage());
                    }
                }
            }
        }
    }

    private static void processOrder(ConsumerRecord<String, String> record) {
```

```
            // Business logic
            System.out.printf("Processing order %s at offset %d%n",
                record.key(), record.offset());
        }
    }
```

👥 Consumer Groups

Rebalancing Strategies

Range Assignor (Default)

```
public class RangeAssignorExample {

    public static Properties createRangeAssignorConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "range-assignor-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Range assignor (default)
        props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            "org.apache.kafka.clients.consumer.RangeAssignor");

        return props;
    }

    /*
     * Range Assignor Distribution:
     * Topic: orders (6 partitions: 0,1,2,3,4,5)
     * 3 consumers: C1, C2, C3
     *
     * Assignment:
     * C1: partitions 0,1  (6/3 = 2)
     * C2: partitions 2,3  (6/3 = 2)
     * C3: partitions 4,5  (6/3 = 2)
     *
     * If 4 consumers:
     * C1: partitions 0,1  (first 6%4=2 consumers get extra)
     * C2: partitions 2,3
     * C3: partition 4
     * C4: partition 5
     */
}
```

Round Robin Assignor

```
public class RoundRobinAssignorExample {

    public static Properties createRoundRobinAssignorConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "roundrobin-assignor-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Round robin assignor
        props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            "org.apache.kafka.clients.consumer.RoundRobinAssignor");

        return props;
    }

    /*
     * Round Robin Assignor Distribution:
     * Topic: orders (6 partitions: 0,1,2,3,4,5)
     * 3 consumers: C1, C2, C3
     *
     * Assignment (round-robin):
     * C1: partitions 0,3
     * C2: partitions 1,4
     * C3: partitions 2,5
     *
     * Better distribution across multiple topics
     */
}
```

Sticky Assignor (Recommended)

```
public class StickyAssignorExample {

    public static Properties createStickyAssignorConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "sticky-assignor-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Sticky assignor - minimizes partition movement
        props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            "org.apache.kafka.clients.consumer.StickyAssignor");

        // Reduce rebalance impact
```

```
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 3000);

        return props;
    }

    /*
     * Sticky Assignor Benefits:
     * 1. Minimizes partition reassignment during rebalance
     * 2. Maintains balanced distribution
     * 3. Reduces consumer startup time after rebalance
     *
     * Example:
     * Initial: C1[0,1], C2[2,3], C3[4,5]
     * C2 leaves: C1[0,1,2], C3[4,5,3]  (C1 and C3 keep their partitions)
     * vs Round Robin: C1[0,2,4], C3[1,3,5]  (all partitions reassigned)
     */
}
```

Cooperative Sticky Assignor (Kafka 2.4+)

```
public class CooperativeStickyAssignorExample {

    public static Properties createCooperativeStickyConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "cooperative-sticky-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Cooperative sticky assignor - incremental rebalancing
        props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            "org.apache.kafka.clients.consumer.CooperativeStickyAssignor");

        // Optimize for cooperative rebalancing
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 3000);
        props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000);

        return props;
    }

    public static void main(String[] args) {
        Properties props = createCooperativeStickyConfig();

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
{

            // Rebalance listener to observe cooperative rebalancing
```

```
            consumer.subscribe(Arrays.asList("orders"), new
ConsumerRebalanceListener() {
                @Override
                public void
onPartitionsRevoked(java.util.Collection<TopicPartition> partitions) {
                    System.out.println("Partitions revoked (cooperative): " +
partitions);
                    // With cooperative rebalancing, this may be called multiple
times
                    // Only revoked partitions are stopped, others continue
processing
                }

                @Override
                public void
onPartitionsAssigned(java.util.Collection<TopicPartition> partitions) {
                    System.out.println("Partitions assigned (cooperative): " +
partitions);
                    // New partitions can start processing while others continue
                }

                @Override
                public void onPartitionsLost(java.util.Collection<TopicPartition>
partitions) {
                    System.out.println("Partitions lost: " + partitions);
                    // Handle partition loss scenario
                }
            });

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

                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("Processing record from partition %d: %s%n",
                        record.partition(), record.value());
                }

                consumer.commitAsync();
            }
        }
    }

    /*
     * Cooperative Rebalancing Benefits:
     * 1. No "stop-the-world" rebalancing
     * 2. Unaffected consumers continue processing
     * 3. Faster rebalancing overall
     * 4. Better availability during membership changes
     *
     * Process:
     * 1. First rebalance: Identify partitions to be revoked
     * 2. Only affected consumers stop processing revoked partitions
     * 3. Second rebalance: Reassign revoked partitions
```

```
     * 4. Processing resumes with new assignment
     */
}
```

Static Membership

Static Consumer Configuration

```
public class StaticMembershipExample {

    public static Properties createStaticMemberConfig(String memberId) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "static-membership-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Static membership configuration
        props.put(ConsumerConfig.GROUP_INSTANCE_ID_CONFIG, memberId); // Unique
static ID

        // Longer session timeout for static members
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 60000);   // 60
seconds
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 6000); // 6 seconds

        // Cooperative rebalancing works well with static membership
        props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            "org.apache.kafka.clients.consumer.CooperativeStickyAssignor");

        return props;
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java StaticMembershipExample <member-id>");
            System.exit(1);
        }

        String memberId = args[0]; // e.g., "consumer-1", "consumer-2"
        Properties props = createStaticMemberConfig(memberId);

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
{

            consumer.subscribe(Arrays.asList("orders"), new
ConsumerRebalanceListener() {
                @Override
                public void
```

```
onPartitionsRevoked(java.util.Collection<TopicPartition> partitions) {
                    System.out.printf("[%s] Partitions revoked: %s%n", memberId,
partitions);
                }

                @Override
                public void
onPartitionsAssigned(java.util.Collection<TopicPartition> partitions) {
                    System.out.printf("[%s] Partitions assigned: %s%n", memberId,
partitions);
                }
            });

            System.out.printf("Started static member: %s%n", memberId);

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

                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("[%s] Processing from partition %d: %s%n",
                        memberId, record.partition(), record.value());
                }

                if (!records.isEmpty()) {
                    consumer.commitSync();
                }
            }

        } catch (Exception e) {
            System.err.printf("[%s] Consumer failed: %s%n", memberId,
e.getMessage());
        }
    }

    /*
     * Static Membership Benefits:
     *
     * 1. Avoid unnecessary rebalances:
     *    - Temporary disconnects don't trigger rebalance
     *    - Restart with same group.instance.id keeps same partitions
     *
     * 2. Faster recovery:
     *    - No partition reassignment on temporary failures
     *    - Immediate resumption when consumer reconnects
     *
     * 3. Better for stateful processing:
     *    - Maintains partition-to-consumer affinity
     *    - Useful for local state/caches tied to partitions
     *
     * Usage scenarios:
     * - Kafka Streams applications
     * - Consumers with local state tied to partitions
     * - Applications sensitive to rebalance overhead
```

```
     */
}
```

Group Coordination

Custom Rebalance Listener with State Management

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class StatefulConsumerExample {

    private final Map<Integer, String> partitionState = new ConcurrentHashMap<>();
    private final String consumerId;

    public StatefulConsumerExample(String consumerId) {
        this.consumerId = consumerId;
    }

    public void consume() {
        Properties props = createConsumerConfig();

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
{

            consumer.subscribe(Arrays.asList("user-events"), new
StateAwareRebalanceListener());

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

                for (ConsumerRecord<String, String> record : records) {
                    processRecordWithState(record);
                }

                consumer.commitSync();
            }
        }
    }

    private Properties createConsumerConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "stateful-consumer-group");
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, consumerId);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
```

```
        // Static membership for better state management
        props.put(ConsumerConfig.GROUP_INSTANCE_ID_CONFIG, consumerId);
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 45000);

        return props;
    }

    private void processRecordWithState(ConsumerRecord<String, String> record) {
        int partition = record.partition();
        String value = record.value();

        // Use partition-local state
        String currentState = partitionState.get(partition);
        String newState = updateState(currentState, value);
        partitionState.put(partition, newState);

        System.out.printf("[%s] Partition %d: %s -> %s%n",
            consumerId, partition, currentState, newState);
    }

    private String updateState(String currentState, String newValue) {
        // State update logic based on new message
        return currentState == null ? newValue : currentState + "," + newValue;
    }

    private class StateAwareRebalanceListener implements ConsumerRebalanceListener
{

        @Override
        public void onPartitionsRevoked(java.util.Collection<TopicPartition>
partitions) {
            System.out.printf("[%s] Partitions revoked: %s%n", consumerId,
partitions);

            // Save partition state before losing partitions
            for (TopicPartition partition : partitions) {
                String state = partitionState.remove(partition.partition());
                if (state != null) {
                    savePartitionState(partition, state);
                }
            }

            // Commit offsets before rebalance
            try {
                // Note: consumer reference not available here in this example
                // In real implementation, you'd pass consumer reference
                System.out.println("Committing offsets before rebalance");
            } catch (Exception e) {
                System.err.println("Failed to commit offsets: " + e.getMessage());
            }
        }

        @Override
```

```
        public void onPartitionsAssigned(java.util.Collection<TopicPartition>
partitions) {
            System.out.printf("[%s] Partitions assigned: %s%n", consumerId,
partitions);

            // Restore state for newly assigned partitions
            for (TopicPartition partition : partitions) {
                String state = loadPartitionState(partition);
                if (state != null) {
                    partitionState.put(partition.partition(), state);
                }
            }
        }

        @Override
        public void onPartitionsLost(java.util.Collection<TopicPartition>
partitions) {
            System.out.printf("[%s] Partitions lost: %s%n", consumerId,
partitions);

            // Clean up state for lost partitions
            for (TopicPartition partition : partitions) {
                partitionState.remove(partition.partition());
            }
        }
    }

    private void savePartitionState(TopicPartition partition, String state) {
        // Save state to external store (database, file, etc.)
        System.out.printf("Saving state for %s: %s%n", partition, state);
    }

    private String loadPartitionState(TopicPartition partition) {
        // Load state from external store
        System.out.printf("Loading state for %s%n", partition);
        return null; // Return actual state from storage
    }

    public static void main(String[] args) {
        String consumerId = args.length > 0 ? args[0] : "consumer-1";
        new StatefulConsumerExample(consumerId).consume();
    }
}
```

Offset Management

__consumer_offsets Topic

Understanding the Internal Topic

```
public class ConsumerOffsetsExplorer {

    public static void main(String[] args) {
        // Consumer offsets are stored in __consumer_offsets topic
        System.out.println("Consumer Offsets Topic Structure:");
        System.out.println("Topic: __consumer_offsets");
        System.out.println("Partitions: 50 (default, configurable with
offsets.topic.num.partitions)");
        System.out.println("Replication Factor: 3 (default, configurable with
offsets.topic.replication.factor)");
        System.out.println("Cleanup Policy: compact (to keep latest offsets
only)");

        exploreOffsetStorage();
    }

    private static void exploreOffsetStorage() {
        /*
         * Offset Storage Format:
         *
         * Key: [group.id, topic, partition]
         * Value: [offset, metadata, commit_timestamp, expire_timestamp]
         *
         * Example:
         * Key: ["my-consumer-group", "orders", 0]
         * Value: [1234, "processed successfully", 1693875600000, 1693962000000]
         *
         * Partition Assignment:
         * hash(group.id) % offsets.topic.num.partitions
         */

        String groupId = "my-consumer-group";
        int offsetsTopicPartitions = 50; // default

        int partitionForGroup = Math.abs(groupId.hashCode()) %
offsetsTopicPartitions;
        System.out.printf("Group '%s' offsets stored in __consumer_offsets
partition %d%n",
            groupId, partitionForGroup);
    }

    // Method to programmatically access offset information
    public static void monitorConsumerOffsets(String groupId) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "offset-monitor-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.ByteArrayDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.ByteArrayDeserializer");
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        try (KafkaConsumer<byte[], byte[]> consumer = new KafkaConsumer<>(props))
```

```
{
            // Subscribe to __consumer_offsets topic
            consumer.subscribe(Arrays.asList("__consumer_offsets"));

            System.out.println("Monitoring consumer offsets...");

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

                for (ConsumerRecord<byte[], byte[]> record : records) {
                    // Decode offset commit messages
                    // Note: This is complex as it requires understanding Kafka's
internal format
                    // In practice, use AdminClient or JMX metrics instead

                    System.out.printf("Offset commit in partition %d at offset
%d%n",
                        record.partition(), record.offset());
                }
            }
        }
    }
}
```

Programmatic Offset Management

```
import org.apache.kafka.clients.admin.*;
import org.apache.kafka.common.TopicPartition;
import java.util.Arrays;
import java.util.Map;
import java.util.concurrent.ExecutionException;

public class OffsetManagementExample {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        Properties adminProps = new Properties();
        adminProps.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

        try (AdminClient adminClient = AdminClient.create(adminProps)) {

            String groupId = "my-consumer-group";

            // List consumer groups
            listConsumerGroups(adminClient);

            // Get consumer group details
            describeConsumerGroup(adminClient, groupId);
```

```
            // List consumer group offsets
            listConsumerGroupOffsets(adminClient, groupId);

            // Reset consumer group offsets
            // resetConsumerGroupOffsets(adminClient, groupId);
        }
    }

    private static void listConsumerGroups(AdminClient adminClient)
            throws ExecutionException, InterruptedException {

        ListConsumerGroupsResult result = adminClient.listConsumerGroups();

        System.out.println("Consumer Groups:");
        result.all().get().forEach(group -> {
            System.out.printf("  Group ID: %s, State: %s, Protocol: %s%n",
                group.groupId(), group.state().orElse("UNKNOWN"),
                group.protocolType().orElse("UNKNOWN"));
        });
    }

    private static void describeConsumerGroup(AdminClient adminClient, String
groupId)
            throws ExecutionException, InterruptedException {

        DescribeConsumerGroupsResult result =
            adminClient.describeConsumerGroups(Arrays.asList(groupId));

        ConsumerGroupDescription description = result.all().get().get(groupId);

        System.out.printf("Consumer Group: %s%n", groupId);
        System.out.printf("  State: %s%n", description.state());
        System.out.printf("  Protocol Type: %s%n", description.protocolType());
        System.out.printf("  Protocol: %s%n", description.protocolData());
        System.out.printf("  Coordinator: %s%n", description.coordinator());

        System.out.println("  Members:");
        description.members().forEach(member -> {
            System.out.printf("    Member ID: %s%n", member.memberId());
            System.out.printf("    Client ID: %s%n", member.clientId());
            System.out.printf("    Host: %s%n", member.host());
            System.out.printf("    Assigned Partitions: %s%n",
member.assignment().topicPartitions());
        });
    }

    private static void listConsumerGroupOffsets(AdminClient adminClient, String
groupId)
            throws ExecutionException, InterruptedException {

        ListConsumerGroupOffsetsResult result =
            adminClient.listConsumerGroupOffsets(groupId);

        Map<TopicPartition, OffsetAndMetadata> offsets =
```

```
result.partitionsToOffsetAndMetadata().get();

        System.out.printf("Offsets for Consumer Group: %s%n", groupId);
        offsets.forEach((partition, offset) -> {
            System.out.printf("  %s: offset=%d, metadata=%s%n",
                partition, offset.offset(), offset.metadata());
        });
    }

    private static void resetConsumerGroupOffsets(AdminClient adminClient, String
groupId)
            throws ExecutionException, InterruptedException {

        // Reset to earliest offsets for all partitions of a topic
        TopicPartition partition = new TopicPartition("orders", 0);

        Map<TopicPartition, OffsetAndMetadata> resetOffsets = Map.of(
            partition, new OffsetAndMetadata(0L, "Reset to beginning")
        );

        AlterConsumerGroupOffsetsResult result =
            adminClient.alterConsumerGroupOffsets(groupId, resetOffsets);

        result.all().get();

        System.out.printf("Reset offsets for group %s%n", groupId);
    }
}
```

Reset Policies

Auto Offset Reset Configuration

```
public class OffsetResetPoliciesExample {

    // Reset to earliest available offset
    public static Properties createEarliestResetConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "earliest-reset-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        return props;
    }

    // Reset to latest offset (skip existing messages)
```

```
    public static Properties createLatestResetConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "latest-reset-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");

        return props;
    }

    // Throw exception on missing offset
    public static Properties createNoneResetConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "none-reset-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "none");

        return props;
    }

    public static void demonstrateResetPolicies() {

        // Scenario 1: New consumer group with earliest
        System.out.println("=== EARLIEST Reset Policy ===");
        demonstrateEarliestReset();

        // Scenario 2: New consumer group with latest
        System.out.println("\n=== LATEST Reset Policy ===");
        demonstrateLatestReset();

        // Scenario 3: Error handling with none
        System.out.println("\n=== NONE Reset Policy ===");
        demonstrateNoneReset();
    }

    private static void demonstrateEarliestReset() {
        Properties props = createEarliestResetConfig();

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

            // This will start reading from the earliest available offset
            // for new consumer groups or when committed offset is invalid
```

```
            ConsumerRecords<String, String> records =
                consumer.poll(Duration.ofMillis(5000));

            System.out.printf("Records fetched with earliest reset: %d%n",
records.count());

        } catch (Exception e) {
            System.err.println("Error with earliest reset: " + e.getMessage());
        }
    }

    private static void demonstrateLatestReset() {
        Properties props = createLatestResetConfig();

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

            // This will start reading from the latest offset
            // (skipping all existing messages)

            ConsumerRecords<String, String> records =
                consumer.poll(Duration.ofMillis(5000));

            System.out.printf("Records fetched with latest reset: %d%n",
records.count());

        } catch (Exception e) {
            System.err.println("Error with latest reset: " + e.getMessage());
        }
    }

    private static void demonstrateNoneReset() {
        Properties props = createNoneResetConfig();

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

            // This will throw an exception if no committed offset exists
            ConsumerRecords<String, String> records =
                consumer.poll(Duration.ofMillis(5000));

            System.out.printf("Records fetched with none reset: %d%n",
records.count());

        } catch (org.apache.kafka.clients.consumer.NoOffsetForPartitionException
e) {
            System.err.println("Expected exception with none reset: " +
e.getMessage());
        } catch (Exception e) {
            System.err.println("Unexpected error: " + e.getMessage());
        }
    }
```

```
    public static void main(String[] args) {
        demonstrateResetPolicies();
    }
}
```

Manual Offset Reset

```
public class ManualOffsetResetExample {

    public static void main(String[] args) {
        Properties props = createManualOffsetConfig();

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
{

            // Manual partition assignment (not using consumer groups)
            TopicPartition partition = new TopicPartition("orders", 0);
            consumer.assign(Arrays.asList(partition));

            // Different manual reset strategies
            demonstrateManualResets(consumer, partition);

        } catch (Exception e) {
            System.err.println("Manual offset reset failed: " + e.getMessage());
        }
    }

    private static Properties createManualOffsetConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        // No group.id for manual assignment
        return props;
    }

    private static void demonstrateManualResets(KafkaConsumer<String, String>
consumer,
                                                TopicPartition partition) {

        // 1. Seek to beginning
        System.out.println("=== Seek to Beginning ===");
        consumer.seekToBeginning(Arrays.asList(partition));
        pollAndPrint(consumer, "Beginning");

        // 2. Seek to end
```

```
        System.out.println("\n=== Seek to End ===");
        consumer.seekToEnd(Arrays.asList(partition));
        pollAndPrint(consumer, "End");

        // 3. Seek to specific offset
        System.out.println("\n=== Seek to Specific Offset ===");
        consumer.seek(partition, 10L);
        pollAndPrint(consumer, "Offset 10");

        // 4. Seek by timestamp
        System.out.println("\n=== Seek by Timestamp ===");
        seekByTimestamp(consumer, partition);

        // 5. Get offset information
        System.out.println("\n=== Offset Information ===");
        printOffsetInformation(consumer, partition);
    }

    private static void pollAndPrint(KafkaConsumer<String, String> consumer,
String context) {
        ConsumerRecords<String, String> records =
            consumer.poll(Duration.ofMillis(2000));

        System.out.printf("%s - Records: %d%n", context, records.count());

        records.forEach(record -> {
            System.out.printf("  Offset: %d, Key: %s, Value: %s%n",
                record.offset(), record.key(), record.value());
        });
    }

    private static void seekByTimestamp(KafkaConsumer<String, String> consumer,
                                        TopicPartition partition) {

        // Seek to 1 hour ago
        long oneHourAgo = System.currentTimeMillis() - (60 * 60 * 1000);

        Map<TopicPartition, Long> timestampsToSearch = Map.of(partition,
oneHourAgo);

        Map<TopicPartition, OffsetAndTimestamp> offsetsForTimes =
            consumer.offsetsForTimes(timestampsToSearch);

        OffsetAndTimestamp offsetAndTimestamp = offsetsForTimes.get(partition);

        if (offsetAndTimestamp != null) {
            consumer.seek(partition, offsetAndTimestamp.offset());
            System.out.printf("Seeking to timestamp %d, offset %d%n",
                oneHourAgo, offsetAndTimestamp.offset());
            pollAndPrint(consumer, "Timestamp seek");
        } else {
            System.out.println("No offset found for timestamp");
        }
    }
```

```
    private static void printOffsetInformation(KafkaConsumer<String, String>
consumer,
                                               TopicPartition partition) {

        // Beginning offsets
        Map<TopicPartition, Long> beginningOffsets =
            consumer.beginningOffsets(Arrays.asList(partition));

        // End offsets
        Map<TopicPartition, Long> endOffsets =
            consumer.endOffsets(Arrays.asList(partition));

        // Current position
        long currentPosition = consumer.position(partition);

        System.out.printf("Partition: %s%n", partition);
        System.out.printf("  Beginning offset: %d%n",
beginningOffsets.get(partition));
        System.out.printf("  End offset: %d%n", endOffsets.get(partition));
        System.out.printf("  Current position: %d%n", currentPosition);
        System.out.printf("  Available records: %d%n",
            endOffsets.get(partition) - currentPosition);
    }
}
```

Lag Monitoring

JMX-based Lag Monitoring

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;
import java.util.Set;

public class ConsumerLagMonitoring {

    public static void main(String[] args) {
        // Start a consumer to generate metrics
        Thread consumerThread = new Thread(() -> runConsumer());
        consumerThread.start();

        // Monitor consumer lag via JMX
        Thread monitorThread = new Thread(() -> monitorLagViaMx());
        monitorThread.start();
    }

    private static void runConsumer() {
        Properties props = createMonitoredConsumerConfig();

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
```

```
{
            consumer.subscribe(Arrays.asList("orders"));

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

                // Simulate processing time to create lag
                for (ConsumerRecord<String, String> record : records) {
                    try {
                        Thread.sleep(100); // Simulate slow processing
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        return;
                    }
                }

                consumer.commitSync();
            }
        }
    }

    private static Properties createMonitoredConsumerConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "monitored-consumer-group");
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, "lag-monitored-consumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        // Enable JMX metrics
        props.put("metric.reporters",
"org.apache.kafka.common.metrics.JmxReporter");

        return props;
    }

    private static void monitorLagViaMx() {
        MBeanServer server = ManagementFactory.getPlatformMBeanServer();

        try {
            while (true) {
                Thread.sleep(10000); // Check every 10 seconds

                // Query consumer lag metrics
                Set<ObjectName> consumerMetrics = server.queryNames(
                    new ObjectName("kafka.consumer:type=consumer-fetch-manager-
metrics,client-id=*"),
                    null
                );
```

```
                System.out.println("\n=== Consumer Lag Metrics ===");

                for (ObjectName objName : consumerMetrics) {
                    try {
                        // Records lag max
                        Object recordsLagMax = server.getAttribute(objName,
"records-lag-max");

                        // Records consumed rate
                        Object recordsConsumedRate = server.getAttribute(objName,
"records-consumed-rate");

                        // Fetch latency average
                        Object fetchLatencyAvg = server.getAttribute(objName,
"fetch-latency-avg");

                        String clientId = objName.getKeyProperty("client-id");

                        System.out.printf("Client: %s%n", clientId);
                        System.out.printf("  Records Lag Max: %s%n",
recordsLagMax);
                        System.out.printf("  Records Consumed Rate: %.2f/sec%n",
recordsConsumedRate);
                        System.out.printf("  Fetch Latency Avg: %.2f ms%n",
fetchLatencyAvg);

                    } catch (Exception e) {
                        System.err.println("Error reading metric: " +
e.getMessage());
                    }
                }

                // Query coordinator metrics
                monitorCoordinatorMetrics(server);

            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private static void monitorCoordinatorMetrics(MBeanServer server) {
        try {
            Set<ObjectName> coordinatorMetrics = server.queryNames(
                new ObjectName("kafka.consumer:type=consumer-coordinator-
metrics,client-id=*"),
                null
            );

            System.out.println("\n=== Coordinator Metrics ===");

            for (ObjectName objName : coordinatorMetrics) {
                try {
                    // Heartbeat rate
```

```
                    Object heartbeatRate = server.getAttribute(objName,
"heartbeat-rate");

                    // Join rate
                    Object joinRate = server.getAttribute(objName, "join-rate");

                    // Sync rate
                    Object syncRate = server.getAttribute(objName, "sync-rate");

                    String clientId = objName.getKeyProperty("client-id");

                    System.out.printf("Client: %s%n", clientId);
                    System.out.printf("  Heartbeat Rate: %.2f/sec%n",
heartbeatRate);
                    System.out.printf("  Join Rate: %.2f/sec%n", joinRate);
                    System.out.printf("  Sync Rate: %.2f/sec%n", syncRate);

                } catch (Exception e) {
                    // Metric might not be available
                }
            }

        } catch (Exception e) {
            System.err.println("Error reading coordinator metrics: " +
e.getMessage());
        }
    }
}
```

Application-Level Lag Monitoring

```
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ApplicationLevelLagMonitor {

    private final AdminClient adminClient;
    private final ScheduledExecutorService scheduler;
    private final String groupId;

    public ApplicationLevelLagMonitor(String bootstrapServers, String groupId) {
        Properties adminProps = new Properties();
        adminProps.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);

        this.adminClient = AdminClient.create(adminProps);
        this.scheduler = Executors.newScheduledThreadPool(1);
        this.groupId = groupId;
    }
```

```
    public void startMonitoring() {
        scheduler.scheduleAtFixedRate(this::checkLag, 0, 30, TimeUnit.SECONDS);
    }

    private void checkLag() {
        try {
            // Get consumer group offsets
            ListConsumerGroupOffsetsResult offsetsResult =
                adminClient.listConsumerGroupOffsets(groupId);

            Map<TopicPartition, OffsetAndMetadata> consumerOffsets =
                offsetsResult.partitionsToOffsetAndMetadata().get();

            if (consumerOffsets.isEmpty()) {
                System.out.println("No consumer offsets found for group: " +
groupId);
                return;
            }

            // Get latest offsets for topics
            Set<TopicPartition> partitions = consumerOffsets.keySet();
            Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo>
latestOffsets =
                adminClient.listOffsets(
                    partitions.stream().collect(java.util.stream.Collectors.toMap(
                        tp -> tp,
                        tp -> OffsetSpec.latest()
                    ))
                ).all().get();

            System.out.printf("\n=== Consumer Lag Report (%s) ===%n",
                java.time.LocalDateTime.now());
            System.out.printf("Group: %s%n", groupId);

            long totalLag = 0;

            for (TopicPartition partition : partitions) {
                OffsetAndMetadata consumerOffset = consumerOffsets.get(partition);
                ListOffsetsResult.ListOffsetsResultInfo latestOffset =
latestOffsets.get(partition);

                if (consumerOffset != null && latestOffset != null) {
                    long lag = latestOffset.offset() - consumerOffset.offset();
                    totalLag += lag;

                    System.out.printf("  %s: consumer=%d, latest=%d, lag=%d%n",
                        partition, consumerOffset.offset(), latestOffset.offset(),
lag);

                    // Alert on high lag
                    if (lag > 1000) {
                        alertHighLag(partition, lag);
                    }
```

```
                }
            }

            System.out.printf("Total Lag: %d messages%n", totalLag);

            // Alert on total high lag
            if (totalLag > 5000) {
                alertHighTotalLag(totalLag);
            }

        } catch (Exception e) {
            System.err.println("Error checking consumer lag: " + e.getMessage());
        }
    }

    private void alertHighLag(TopicPartition partition, long lag) {
        System.err.printf("ALERT: High lag detected for %s: %d messages%n",
            partition, lag);

        // In real implementation:
        // - Send notification to monitoring system
        // - Trigger alert to ops team
        // - Log to alerting service
    }

    private void alertHighTotalLag(long totalLag) {
        System.err.printf("ALERT: High total lag detected for group %s: %d
messages%n",
            groupId, totalLag);

        // In real implementation:
        // - Check consumer health
        // - Consider scaling up consumers
        // - Alert operations team
    }

    public void shutdown() {
        scheduler.shutdown();
        adminClient.close();
    }

    public static void main(String[] args) {
        ApplicationLevelLagMonitor monitor =
            new ApplicationLevelLagMonitor("localhost:9092", "my-consumer-group");

        monitor.startMonitoring();

        // Shutdown hook
        Runtime.getRuntime().addShutdownHook(new Thread(monitor::shutdown));

        // Keep main thread alive
        try {
            Thread.sleep(Long.MAX_VALUE);
        } catch (InterruptedException e) {
```

```
                Thread.currentThread().interrupt();
            }
        }
    }
```

Comprehensive Java Examples

Production-Ready Consumer Application

```
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.errors.*;
import org.apache.kafka.common.TopicPartition;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.time.Duration;
import java.util.*;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicLong;

/**
 * Production-ready Kafka Consumer with comprehensive error handling,
 * monitoring, and resilience features.
 */
public class ProductionKafkaConsumerExample {

    private final Properties config;
    private final String topicName;
    private final AtomicBoolean running = new AtomicBoolean(true);
    private final AtomicLong processedMessages = new AtomicLong(0);
    private final AtomicLong failedMessages = new AtomicLong(0);
    private final ObjectMapper objectMapper = new ObjectMapper();

    public ProductionKafkaConsumerExample(String topicName, Properties config) {
        this.topicName = topicName;
        this.config = config;
    }

    public static void main(String[] args) {
        Properties config = createProductionConsumerConfig();
        ProductionKafkaConsumerExample consumerApp =
            new ProductionKafkaConsumerExample("order-events", config);

        // Setup shutdown hook
        Runtime.getRuntime().addShutdownHook(new Thread(consumerApp::shutdown));

        // Start consuming
        consumerApp.consume();
    }

    private static Properties createProductionConsumerConfig() {
```

```
        Properties props = new Properties();

        // Connection settings
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "broker1:9092,broker2:9092,broker3:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "order-processing-service");
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, "order-processor-" +
UUID.randomUUID());

        // Serialization
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");

        // Reliability settings
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false); // Manual
commit
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // Performance settings
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 50000);       // 50KB min
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 1000);      // 1s max
wait
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);        // 100
records max
        props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000); // 5 min
processing

        // Group coordination
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);    // 30s
session
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 3000);  // 3s
heartbeat

        // Rebalancing
        props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
            "org.apache.kafka.clients.consumer.CooperativeStickyAssignor");

        // Static membership for stability
        props.put(ConsumerConfig.GROUP_INSTANCE_ID_CONFIG,
            "order-processor-" + System.getenv().getOrDefault("HOSTNAME",
"localhost"));

        return props;
    }

    public void consume() {
        while (running.get()) {
            try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>
(config)) {

                consumer.subscribe(Arrays.asList(topicName), new
RebalanceHandler());
```

```
                System.out.println("Consumer started, waiting for messages...");

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

                        if (records.isEmpty()) {
                            continue;
                        }

                        processRecordsBatch(records);
                        commitOffsets(consumer);

                        // Print statistics
                        printStatistics();

                    } catch (WakeupException e) {
                        System.out.println("Consumer wakeup called");
                        break;

                    } catch (CommitFailedException e) {
                        System.err.println("Commit failed during rebalance: " +
e.getMessage());
                        // Continue processing - rebalance will reset offsets

                    } catch (InvalidOffsetException e) {
                        System.err.println("Invalid offset, seeking to beginning:
" + e.getMessage());
                        consumer.seekToBeginning(e.partitions());

                    } catch (AuthorizationException e) {
                        System.err.println("Authorization failed: " +
e.getMessage());
                        break; // Fatal error

                    } catch (InterruptException e) {
                        System.err.println("Consumer interrupted: " +
e.getMessage());
                        Thread.currentThread().interrupt();
                        break;

                    } catch (Exception e) {
                        System.err.println("Unexpected consumer error: " +
e.getMessage());
                        e.printStackTrace();

                        // Decide whether to continue or stop
                        if (isFatalError(e)) {
                            break;
                        }

                        // Wait before retry
```

```
                        try {
                            Thread.sleep(5000);
                        } catch (InterruptedException ie) {
                            Thread.currentThread().interrupt();
                            break;
                        }
                    }
                }

            } catch (Exception e) {
                System.err.println("Consumer initialization failed: " +
e.getMessage());

                if (running.get()) {
                    // Wait before retry
                    try {
                        Thread.sleep(10000);
                    } catch (InterruptedException ie) {
                        Thread.currentThread().interrupt();
                        break;
                    }
                }
            }
        }

        System.out.println("Consumer stopped");
    }

    private void processRecordsBatch(ConsumerRecords<String, String> records) {
        System.out.printf("Processing batch of %d records%n", records.count());

        for (ConsumerRecord<String, String> record : records) {
            try {
                processMessage(record);
                processedMessages.incrementAndGet();

            } catch (Exception e) {
                failedMessages.incrementAndGet();
                handleProcessingError(record, e);
            }
        }
    }

    private void processMessage(ConsumerRecord<String, String> record) throws
Exception {
        // Deserialize message
        OrderEvent orderEvent = objectMapper.readValue(record.value(),
OrderEvent.class);

        // Business logic processing
        System.out.printf("Processing order: %s for user: %s, amount: $%.2f%n",
            orderEvent.getOrderId(), orderEvent.getUserId(),
orderEvent.getAmount());
```

```
            // Simulate processing time
            Thread.sleep(50);

            // Validate order
            if (orderEvent.getAmount() <= 0) {
                throw new IllegalArgumentException("Invalid order amount");
            }

            // Process order (save to database, call external services, etc.)
            processOrder(orderEvent);

            // Update metrics
            updateProcessingMetrics(orderEvent);
        }

        private void processOrder(OrderEvent orderEvent) {
            // Simulate order processing
            // In real implementation:
            // - Save to database
            // - Call payment service
            // - Update inventory
            // - Send confirmation

            System.out.printf("Order %s processed successfully%n",
    orderEvent.getOrderId());
        }

        private void updateProcessingMetrics(OrderEvent orderEvent) {
            // Update application metrics
            // In real implementation:
            // - Update Micrometer/Prometheus metrics
            // - Send to monitoring system
            // - Update business KPIs
        }

        private void handleProcessingError(ConsumerRecord<String, String> record,
    Exception e) {
            System.err.printf("Failed to process record at offset %d: %s%n",
                record.offset(), e.getMessage());

            if (e instanceof IllegalArgumentException) {
                // Invalid data - skip and continue
                System.err.println("Skipping invalid record");
                logFailedRecord(record, e, "INVALID_DATA");

            } else if (e instanceof java.net.ConnectException) {
                // External service unavailable
                System.err.println("External service unavailable - sending to retry
    queue");
                sendToRetryQueue(record, e);

            } else if (e instanceof java.sql.SQLException) {
                // Database error - might be transient
                System.err.println("Database error - sending to retry queue");
```

```
                sendToRetryQueue(record, e);

            } else {
                // Unknown error - send to DLQ
                System.err.println("Unknown error - sending to dead letter queue");
                sendToDeadLetterQueue(record, e);
            }
        }

        private void sendToRetryQueue(ConsumerRecord<String, String> record, Exception
e) {
            // Implementation to send to retry topic
            System.out.printf("Sending to retry queue: key=%s, partition=%d,
offset=%d%n",
                record.key(), record.partition(), record.offset());
        }

        private void sendToDeadLetterQueue(ConsumerRecord<String, String> record,
Exception e) {
            // Implementation to send to DLQ topic
            System.out.printf("Sending to DLQ: key=%s, partition=%d, offset=%d%n",
                record.key(), record.partition(), record.offset());
        }

        private void logFailedRecord(ConsumerRecord<String, String> record, Exception
e, String reason) {
            System.err.printf("Failed record logged: reason=%s, key=%s, offset=%d,
error=%s%n",
                reason, record.key(), record.offset(), e.getMessage());
        }

        private void commitOffsets(KafkaConsumer<String, String> consumer) {
            try {
                consumer.commitSync();
            } catch (CommitFailedException e) {
                System.err.println("Offset commit failed: " + e.getMessage());
                // Could implement retry logic or continue
            }
        }

        private void printStatistics() {
            long processed = processedMessages.get();
            long failed = failedMessages.get();

            if (processed % 100 == 0 && processed > 0) {
                System.out.printf("Statistics: Processed=%d, Failed=%d, Success
Rate=%.2f%%%n",
                    processed, failed,
                    (processed * 100.0) / (processed + failed));
            }
        }

        private boolean isFatalError(Exception e) {
            return e instanceof AuthorizationException ||
```

```
                e instanceof SecurityException ||
                e instanceof OutOfMemoryError;
    }

    public void shutdown() {
        System.out.println("Shutting down consumer...");
        running.set(false);
    }

    // Rebalance handler for proper cleanup
    private class RebalanceHandler implements ConsumerRebalanceListener {

        @Override
        public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
            System.out.println("Partitions revoked: " + partitions);

            // Cleanup before losing partitions
            // - Flush any pending work
            // - Save partition-specific state
            // - Commit current offsets handled by main loop
        }

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

            // Initialize for new partitions
            // - Load partition-specific state
            // - Reset processing metrics
            // - Log partition assignment for monitoring
        }

        @Override
        public void onPartitionsLost(Collection<TopicPartition> partitions) {
            System.err.println("Partitions lost: " + partitions);

            // Handle partition loss
            // - Clean up partition-specific resources
            // - Log error for monitoring
            // - Alert operations if needed
        }
    }

    // Data class for order events
    public static class OrderEvent {
        private String orderId;
        private String userId;
        private double amount;
        private long timestamp;

        // Constructors, getters, setters
        public OrderEvent() {}

        public OrderEvent(String orderId, String userId, double amount, long
```

```
timestamp) {
            this.orderId = orderId;
            this.userId = userId;
            this.amount = amount;
            this.timestamp = timestamp;
        }

        // Getters
        public String getOrderId() { return orderId; }
        public String getUserId() { return userId; }
        public double getAmount() { return amount; }
        public long getTimestamp() { return timestamp; }

        // Setters
        public void setOrderId(String orderId) { this.orderId = orderId; }
        public void setUserId(String userId) { this.userId = userId; }
        public void setAmount(double amount) { this.amount = amount; }
        public void setTimestamp(long timestamp) { this.timestamp = timestamp; }
    }
}
```

⚖️ Comparisons & Trade-offs

Consumer Configuration Trade-offs

Parameter	Higher Value	Lower Value	Recommendation
`max.poll.records`	↑ Throughput, ↑ Memory, ↑ Processing time	↓ Memory, ↓ Latency	100-1000 based on message size
`fetch.min.bytes`	↑ Throughput, ↑ Latency	↓ Latency, ↓ Throughput	1KB-100KB based on traffic
`session.timeout.ms`	↑ Tolerance to GC pauses, ↓ Failure detection	↑ Failure detection, ↓ GC tolerance	30s for production
`heartbeat.interval.ms`	↓ Network overhead	↑ Failure detection speed	1/3 of session timeout

Commit Strategy Comparison

Strategy	Data Guarantees	Performance	Use Case
Auto Commit	At-most-once (potential loss)	★★★ High	Metrics, logs, non-critical
Sync Commit	At-least-once	★ Lower	Financial, critical data
Async Commit	At-least-once	★★ Good	High-volume, some duplicates OK

Strategy	Data Guarantees	Performance	Use Case
Manual per Record	At-least-once	⭐ Lowest	Critical processing, fine control

Rebalancing Strategy Comparison

Strategy	Distribution	Partition Movement	Use Case
Range	Can be uneven with multiple topics	Minimal	Single topic, ordered processing
Round Robin	Even distribution	Higher movement	Multiple topics, balanced load
Sticky	Even + sticky	Minimal movement	Stateful consumers
Cooperative Sticky	Even + sticky + incremental	Minimal + no downtime	Production systems

🚨 Common Pitfalls & Best Practices

1. Poll Loop Mistakes

❌ Blocking Operations in Poll Loop

```
// DON'T - Blocking operations in poll loop
while (true) {
    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));

    for (ConsumerRecord<String, String> record : records) {
        // This blocks the consumer thread!
        callSlowExternalService(record.value()); // 5 seconds per call
    }

    consumer.commitSync();
}
```

```
// DO - Use separate threads for heavy processing
ExecutorService executor = Executors.newFixedThreadPool(10);

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

    List<Future<Void>> futures = new ArrayList<>();
```

```
    for (ConsumerRecord<String, String> record : records) {
        Future<Void> future = executor.submit(() -> {
            processRecord(record);
            return null;
        });
        futures.add(future);
    }

    // Wait for all processing to complete
    for (Future<Void> future : futures) {
        try {
            future.get(30, TimeUnit.SECONDS); // Timeout to prevent hanging
        } catch (Exception e) {
            System.err.println("Processing failed: " + e.getMessage());
        }
    }

    consumer.commitSync();
}
```

❌ Infinite Poll Timeout

```
// DON'T - Infinite timeout blocks shutdown
while (running) {
    ConsumerRecords<String, String> records =
        consumer.poll(Duration.ofMillis(Long.MAX_VALUE)); // Never returns!

    // Processing...
}
```

```
// DO - Use reasonable timeout for responsive shutdown
while (running.get()) {
    ConsumerRecords<String, String> records =
        consumer.poll(Duration.ofMillis(1000)); // 1 second timeout

    if (records.isEmpty()) {
        continue; // Check running flag frequently
    }

    // Process records...
}
```

2. Offset Management Issues

❌ Committing Before Processing

```
// DON'T - Commit before processing (at-most-once, potential loss)
ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));

// Committing first - dangerous!
consumer.commitSync();

for (ConsumerRecord<String, String> record : records) {
    processRecord(record); // If this fails, message is lost forever
}
```

```
// DO - Commit after successful processing
ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));

boolean allProcessedSuccessfully = true;
for (ConsumerRecord<String, String> record : records) {
    try {
        processRecord(record);
    } catch (Exception e) {
        System.err.println("Processing failed: " + e.getMessage());
        allProcessedSuccessfully = false;
        break; // Stop processing batch
    }
}

// Only commit if all records processed successfully
if (allProcessedSuccessfully) {
    consumer.commitSync();
}
```

❌ Not Handling Commit Failures

```
// DON'T - Ignore commit failures
consumer.commitSync(); // What if this fails?
```

```
// DO - Handle commit failures appropriately
try {
    consumer.commitSync();
} catch (CommitFailedException e) {
    System.err.println("Commit failed: " + e.getMessage());

    // Strategy options:
    // 1. Continue (rebalance will reset offsets)
    // 2. Retry commit
    // 3. Alert monitoring system
```

```
    alertCommitFailure(e);
}
```

3. Rebalancing Issues

❌ Long Processing Blocking Rebalance

```
// DON'T - Long processing in rebalance listener
class BadRebalanceListener implements ConsumerRebalanceListener {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        // This blocks rebalancing!
        saveMassiveStateToDatabase(); // 30 seconds
        consumer.commitSync(); // Another potential delay
    }
}
```

```
// DO - Quick cleanup in rebalance listener
class GoodRebalanceListener implements ConsumerRebalanceListener {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        System.out.println("Partitions revoked: " + partitions);

        // Quick operations only
        try {
            consumer.commitSync(Duration.ofSeconds(5)); // Timeout commit
        } catch (Exception e) {
            System.err.println("Quick commit failed: " + e.getMessage());
        }

        // Async cleanup for heavy operations
        CompletableFuture.runAsync(() -> {
            saveMassiveStateToDatabase();
        });
    }
}
```

4. Memory Management Problems

❌ Unbounded Record Processing

```
// DON'T - Process unlimited records without backpressure
props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, Integer.MAX_VALUE);
props.put(ConsumerConfig.FETCH_MAX_BYTES_CONFIG, Integer.MAX_VALUE);

// This can cause OutOfMemoryError!
```

```
ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
List<String> allData = new ArrayList<>();
for (ConsumerRecord<String, String> record : records) {
    allData.add(record.value()); // Unbounded memory growth
}
```

```
// DO - Set reasonable limits and implement backpressure
props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
props.put(ConsumerConfig.FETCH_MAX_BYTES_CONFIG, 52428800); // 50MB

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

// Process in batches to control memory
int batchSize = 100;
List<ConsumerRecord<String, String>> batch = new ArrayList<>();

for (ConsumerRecord<String, String> record : records) {
    batch.add(record);

    if (batch.size() >= batchSize) {
        processBatch(batch);
        batch.clear(); // Free memory
    }
}

// Process remaining records
if (!batch.isEmpty()) {
    processBatch(batch);
}
```

Best Practices Summary

☑ Consumer Configuration Best Practices

1. **Use manual commits** for critical data (`enable.auto.commit=false`)
2. **Set appropriate timeouts** (`session.timeout.ms=30000`, `heartbeat.interval.ms=3000`)
3. **Limit poll size** (`max.poll.records=500` based on processing speed)
4. **Use cooperative sticky assignor** for better rebalancing
5. **Enable static membership** for stable applications

☑ Processing Best Practices

1. **Keep poll loop lightweight** - offload heavy processing to threads
2. **Implement proper error handling** - distinguish retriable vs fatal errors
3. **Commit after processing** - ensure at-least-once semantics
4. **Handle rebalances gracefully** - quick cleanup only
5. **Monitor consumer lag** - set up alerting for high lag

☑ Operational Best Practices

1. **Monitor key metrics** - lag, commit rate, rebalance frequency
2. **Set up proper logging** - include partition, offset, timestamp info
3. **Implement circuit breakers** - handle downstream failures
4. **Use dead letter queues** - for failed messages
5. **Test failure scenarios** - consumer restarts, network partitions

---

🌍 Real-World Use Cases

1. Order Processing System

```
@Service
public class OrderProcessingConsumer {

    private final OrderService orderService;
    private final InventoryService inventoryService;
    private final PaymentService paymentService;
    private final NotificationService notificationService;

    @EventListener
    public void consumeOrderEvents() {
        Properties props = createOrderConsumerConfig();

        try (KafkaConsumer<String, OrderEvent> consumer = new KafkaConsumer<>
(props)) {
            consumer.subscribe(Arrays.asList("order-events"), new
OrderRebalanceListener());

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

                // Process orders in transaction
                processOrdersInTransaction(records);
                consumer.commitSync();
            }
        }
    }

    @Transactional
    private void processOrdersInTransaction(ConsumerRecords<String, OrderEvent>
records) {
        for (ConsumerRecord<String, OrderEvent> record : records) {
            OrderEvent event = record.value();

            try {
                // Process order atomically
                Order order = orderService.processOrder(event);
                inventoryService.reserveItems(order);
                paymentService.processPayment(order);
```

```
                notificationService.sendConfirmation(order);

                logger.info("Order processed: {}", event.getOrderId());

            } catch (InsufficientInventoryException e) {
                // Business logic error - send notification
                notificationService.sendInventoryAlert(event);

            } catch (PaymentFailedException e) {
                // Payment failed - update order status
                orderService.markPaymentFailed(event.getOrderId());

            } catch (Exception e) {
                // Unexpected error - send to DLQ
                sendToDeadLetterQueue(record, e);
                throw e; // Fail transaction
            }
        }
    }
}
```

2. Real-time Analytics Consumer

```
@Component
public class AnalyticsConsumer {

    private final MetricsCollector metricsCollector;
    private final TimeWindowAggregator aggregator;

    public void consumeUserEvents() {
        Properties props = createAnalyticsConsumerConfig();

        try (KafkaConsumer<String, UserEvent> consumer = new KafkaConsumer<>
(props)) {
            consumer.subscribe(Arrays.asList("user-events"));

            // Use time-window processing
            Map<String, UserSession> activeSessions = new HashMap<>();

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

                for (ConsumerRecord<String, UserEvent> record : records) {
                    UserEvent event = record.value();

                    // Update user session
                    String userId = event.getUserId();
                    UserSession session = activeSessions.computeIfAbsent(userId,
                        k -> new UserSession(userId));
```

```
                    session.addEvent(event);

                    // Collect real-time metrics
                    metricsCollector.recordEvent(event.getEventType());

                    // Update aggregations
                    aggregator.updateTimeWindow(event);

                    // Expire old sessions
                    expireOldSessions(activeSessions);
                }

                // Publish aggregated metrics
                publishAggregatedMetrics();
                consumer.commitAsync();
            }
        }
    }

    private Properties createAnalyticsConsumerConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "analytics-consumer-group");

        // Optimize for throughput
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 100000);  // 100KB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 500);   // 500ms
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1000);   // 1000 records

        // Use async commits for better performance
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        return props;
    }
}
```

3. Event Sourcing Consumer

```
@Service
public class EventSourcingConsumer {

    private final EventStore eventStore;
    private final ProjectionUpdater projectionUpdater;

    public void consumeEventStream() {
        Properties props = createEventSourcingConfig();

        try (KafkaConsumer<String, DomainEvent> consumer = new KafkaConsumer<>
(props)) {

            // Manual partition assignment for event sourcing
```

```
            TopicPartition partition = new TopicPartition("domain-events", 0);
            consumer.assign(Arrays.asList(partition));

            // Start from last processed offset
            long lastProcessedOffset = eventStore.getLastProcessedOffset();
            consumer.seek(partition, lastProcessedOffset);

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

                for (ConsumerRecord<String, DomainEvent> record : records) {
                    DomainEvent event = record.value();

                    // Ensure exactly-once processing with idempotency
                    if (eventStore.isEventProcessed(event.getEventId())) {
                        continue; // Skip already processed events
                    }

                    // Apply event atomically
                    applyEventAtomically(event, record.offset());
                }

                // Commit only after successful processing
                consumer.commitSync();
            }
        }
    }

    @Transactional
    private void applyEventAtomically(DomainEvent event, long offset) {
        // Store event
        eventStore.appendEvent(event, offset);

        // Update projections
        projectionUpdater.updateProjections(event);

        // Update last processed offset
        eventStore.updateLastProcessedOffset(offset);
    }
}
```

4. CDC (Change Data Capture) Consumer

```
@Service
public class CDCConsumer {

    private final DatabaseSyncService syncService;
    private final SearchIndexService searchService;
    private final CacheService cacheService;
```

```
    public void consumeDBChanges() {
        Properties props = createCDCConsumerConfig();

        try (KafkaConsumer<String, DatabaseChangeEvent> consumer = new
KafkaConsumer<>(props)) {
            consumer.subscribe(Arrays.asList("db-changes"));

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

                // Group changes by table for efficient processing
                Map<String, List<DatabaseChangeEvent>> changesByTable =
                    groupChangesByTable(records);

                // Process changes by table
                for (Map.Entry<String, List<DatabaseChangeEvent>> entry :
changesByTable.entrySet()) {
                    String tableName = entry.getKey();
                    List<DatabaseChangeEvent> changes = entry.getValue();

                    processTableChanges(tableName, changes);
                }

                consumer.commitSync();
            }
        }
    }

    private void processTableChanges(String tableName, List<DatabaseChangeEvent>
changes) {
        for (DatabaseChangeEvent change : changes) {
            try {
                switch (change.getOperation()) {
                    case INSERT:
                        handleInsert(tableName, change);
                        break;
                    case UPDATE:
                        handleUpdate(tableName, change);
                        break;
                    case DELETE:
                        handleDelete(tableName, change);
                        break;
                }

            } catch (Exception e) {
                logger.error("Failed to process change for table {}: {}",
                    tableName, e.getMessage());
                // Could implement retry logic or send to DLQ
            }
        }
    }

    private void handleUpdate(String tableName, DatabaseChangeEvent change) {
```

```
            // Update secondary database
            syncService.updateRecord(tableName, change.getAfter());

            // Update search index
            searchService.updateDocument(change.getRecordId(), change.getAfter());

            // Invalidate cache
            cacheService.invalidate(tableName, change.getRecordId());
        }
    }
```

📈 Version Highlights

Kafka 4.0 (September 2025) - Current Latest

- ✨ **New Consumer Protocol (KIP-848)**: Faster rebalancing, better partition assignment
- ✨ **Enhanced Cooperative Rebalancing**: Further reduced downtime during rebalances
- ✨ **Improved Lag Monitoring**: Better JMX metrics and monitoring capabilities
- ✨ **Java 17+ Required**: For brokers (Java 11+ for clients)

Kafka 3.x Series Consumer Features

- **3.6** (Oct 2023): Consumer metrics improvements, better error reporting
- **3.5** (Jun 2023): Enhanced static membership stability
- **3.4** (Feb 2023): Improved consumer group rebalancing performance
- **3.3** (Oct 2022): Consumer optimizations for KRaft mode
- **3.2** (May 2022): Better consumer coordinator design
- **3.1** (Jan 2022): Improved incremental cooperative rebalancing
- **3.0** (Sep 2021): Consumer performance optimizations

Key Consumer Evolution

Version	Consumer Feature	Impact
4.0	New consumer protocol (KIP-848)	50% faster rebalancing
2.4	Cooperative sticky assignor	Incremental rebalancing
2.3	Static membership (KIP-345)	Reduced rebalances
2.0	Improved rebalancing protocol	Better failure detection
1.1	Headers support in consumer	Message metadata access
0.11	Exactly-once consumption support	Transactional reading
0.10	Timestamp support	Time-based seeking
0.9	Consumer rewrite	Much better performance

Current Recommendations (2025)

```
// Modern Kafka 4.0 consumer configuration
public static Properties modernConsumerConfig() {
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "modern-consumer-group");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

    // Kafka 4.0 best practices
    props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
        "org.apache.kafka.clients.consumer.CooperativeStickyAssignor");

    // Static membership for stability
    props.put(ConsumerConfig.GROUP_INSTANCE_ID_CONFIG,
        "consumer-" + System.getenv("HOSTNAME"));

    // Optimized settings
    props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
    props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 3000);
    props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
    props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 50000);

    // Manual offset management
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

    return props;
}
```

🔗 Additional Resources

📚 Official Documentation

- Kafka Consumer API Documentation
- Consumer Configuration Reference
- Consumer Group Protocol

🎓 Learning Resources

- Confluent Consumer Tutorial
- Apache Kafka Consumer Deep Dive
- Consumer Best Practices

🔧 Tools & Monitoring

- Consumer Lag Monitoring
- JMX Metrics for Consumers
- kafka-consumer-groups.sh CLI

Troubleshooting

- Consumer Rebalancing Issues
- Common Consumer Problems
- Performance Tuning Guide

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

> **Pro Tip**: Start with cooperative sticky assignor and static membership for production systems. Monitor consumer lag closely and implement proper error handling with retry queues and dead letter topics.