Kafka Advanced & Enterprise Features: Complete Developer Guide

A comprehensive guide covering Kafka's enterprise-grade features including transactions, multi-cluster replication, event-driven architectures, and emerging capabilities through 2025.

Table of Contents

Transactions & Exactly-Once Semantics

Simple Explanation

Kafka transactions ensure atomicity across multiple operations - either all operations succeed or all fail together. This enables exactly-once semantics (EOS) for complex data processing pipelines, preventing data loss and duplication.

Problem It Solves

- **Data Consistency**: Ensures atomic operations across multiple topics and partitions
- **Exactly-Once Processing**: Eliminates duplicate processing in stream applications
- **State Management**: Maintains consistent state in stateful stream processing
- **Multi-Step Operations**: Coordinates complex business transactions across Kafka

Internal Architecture

```
Transaction Architecture:

Producer ⇔ [Transaction Coordinator] ⇔ [Transaction Log Topic]
    ↓              ↓                          ↓
```

```
[Topic A]    [Offset Commits]        [Transaction State]
[Topic B]    [State Store]          [Commit/Abort Records]
```

Idempotency vs Transactions

Feature	Idempotency	Transactions
Scope	Single producer, single partition	Multiple producers, topics, partitions
Guarantee	No duplicates from retries	Atomic multi-operation commits
State	Stateless duplicate detection	Stateful transaction management
Performance	Very low overhead	Moderate overhead
Use Case	Simple publish scenarios	Complex processing pipelines

Comprehensive Transaction Examples

```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import java.time.Duration;
import java.util.*;
import java.util.concurrent.atomic.AtomicLong;

/**
 * Comprehensive Kafka Transactions implementation with EOS
 */
public class KafkaTransactionsExample {

    /**
     * Transactional Producer with complete error handling
     */
    public static class TransactionalProducer {

        private final KafkaProducer<String, String> producer;
        private final String transactionId;
        private final AtomicLong transactionCount = new AtomicLong(0);

        public TransactionalProducer(String transactionId) {
            this.transactionId = transactionId;
            this.producer = new KafkaProducer<>(createTransactionalProps());

            // Initialize transactions
            producer.initTransactions();
        }

        private Properties createTransactionalProps() {
            Properties props = new Properties();

            // Basic configuration
```

```
            props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                "org.apache.kafka.common.serialization.StringSerializer");
            props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                "org.apache.kafka.common.serialization.StringSerializer");

            // Transactional configuration
            props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, transactionId);
            props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);

            // Reliability settings
            props.put(ProducerConfig.ACKS_CONFIG, "all");
            props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
            props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5);

            // Performance tuning
            props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
            props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
            props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");

            // Transaction timeout
            props.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60000); // 60
seconds

            return props;
        }

        /**
         * Execute atomic multi-topic transaction
         */
        public void executeAtomicTransaction(List<TransactionOperation>
operations) {
            long txnId = transactionCount.incrementAndGet();

            try {
                producer.beginTransaction();
                System.out.printf("Started transaction %d%n", txnId);

                // Execute all operations within transaction
                for (TransactionOperation operation : operations) {
                    switch (operation.type()) {
                        case SEND_MESSAGE -> sendMessage(operation);
                        case SEND_TO_MULTIPLE_TOPICS ->
sendToMultipleTopics(operation);
                        case CONDITIONAL_SEND -> conditionalSend(operation);
                    }
                }

                // Commit transaction
                producer.commitTransaction();
                System.out.printf("Committed transaction %d with %d operations%n",
                    txnId, operations.size());

            } catch (Exception e) {
```

```
                try {
                    producer.abortTransaction();
                    System.err.printf("Aborted transaction %d due to error: %s%n",
                        txnId, e.getMessage());
                } catch (Exception abortEx) {
                    System.err.printf("Failed to abort transaction %d: %s%n",
                        txnId, abortEx.getMessage());
                }
                throw new RuntimeException("Transaction failed", e);
            }
        }

        /**
         * Send message with transaction context
         */
        private void sendMessage(TransactionOperation operation) throws Exception
{
            ProducerRecord<String, String> record = new ProducerRecord<>(
                operation.topic(),
                operation.key(),
                operation.value()
            );

            // Add transaction metadata as headers
            record.headers().add("transaction-id", transactionId.getBytes());
            record.headers().add("transaction-timestamp",
                String.valueOf(System.currentTimeMillis()).getBytes());

            RecordMetadata metadata = producer.send(record).get();
            System.out.printf("Sent message to %s-%d at offset %d%n",
                metadata.topic(), metadata.partition(), metadata.offset());
        }

        /**
         * Send to multiple topics atomically
         */
        private void sendToMultipleTopics(TransactionOperation operation) throws
Exception {
            Map<String, String> topicMessages = operation.multiTopicMessages();

            List<Future<RecordMetadata>> futures = new ArrayList<>();

            for (Map.Entry<String, String> entry : topicMessages.entrySet()) {
                ProducerRecord<String, String> record = new ProducerRecord<>(
                    entry.getKey(), operation.key(), entry.getValue());
                futures.add(producer.send(record));
            }

            // Wait for all sends to complete
            for (Future<RecordMetadata> future : futures) {
                future.get();
            }
        }
```

```
        /**
         * Conditional send based on business logic
         */
        private void conditionalSend(TransactionOperation operation) throws
Exception {
            if (operation.condition() != null &&
operation.condition().test(operation.value())) {
                sendMessage(operation);
            }
        }

        /**
         * Process consume-transform-produce pattern with EOS
         */
        public void processWithEOS(String inputTopic, String outputTopic,
                                   String consumerGroup, MessageTransformer
transformer) {

            // Create consumer for reading
            KafkaConsumer<String, String> consumer =
createEOSConsumer(consumerGroup);
            consumer.subscribe(Arrays.asList(inputTopic));

            try {
                while (true) {
                    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));

                    if (!records.isEmpty()) {
                        // Begin transaction for this batch
                        producer.beginTransaction();

                        try {
                            Map<TopicPartition, OffsetAndMetadata> offsets = new
HashMap<>();

                            // Process all records in transaction
                            for (ConsumerRecord<String, String> record : records)
{
                                // Transform message
                                String transformedValue =
transformer.transform(record.value());

                                // Send to output topic
                                ProducerRecord<String, String> outputRecord =
                                    new ProducerRecord<>(outputTopic,
record.key(), transformedValue);
                                producer.send(outputRecord);

                                // Track offset for commit
                                TopicPartition tp = new
TopicPartition(record.topic(), record.partition());
                                offsets.put(tp, new
OffsetAndMetadata(record.offset() + 1));
```

```
                            }

                            // Send offsets to transaction
                            producer.sendOffsetsToTransaction(offsets,
consumerGroup);

                            // Commit transaction
                            producer.commitTransaction();

                            System.out.printf("Processed batch of %d records with
EOS%n", records.count());

                        } catch (Exception e) {
                            producer.abortTransaction();
                            System.err.printf("Aborted transaction for batch
processing: %s%n", e.getMessage());
                            throw e;
                        }
                    }
                }
            } finally {
                consumer.close();
            }
        }

        private KafkaConsumer<String, String> createEOSConsumer(String groupId) {
            Properties props = new Properties();
            props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
            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");

            // EOS consumer configuration
            props.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");
            props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
            props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

            return new KafkaConsumer<>(props);
        }

        public void close() {
            producer.close();
        }
    }

    /**
     * Transaction operation definition
     */
    public record TransactionOperation(
        OperationType type,
        String topic,
        String key,
```

```
            String value,
            Map<String, String> multiTopicMessages,
            java.util.function.Predicate<String> condition
        ) {
            public enum OperationType {
                SEND_MESSAGE,
                SEND_TO_MULTIPLE_TOPICS,
                CONDITIONAL_SEND
            }

            public static TransactionOperation sendMessage(String topic, String key,
String value) {
                return new TransactionOperation(OperationType.SEND_MESSAGE, topic,
key, value, null, null);
            }

            public static TransactionOperation sendToMultiple(String key, Map<String,
String> topicMessages) {
                return new TransactionOperation(OperationType.SEND_TO_MULTIPLE_TOPICS,
null, key, null, topicMessages, null);
            }

            public static TransactionOperation conditionalSend(String topic, String
key, String value,

java.util.function.Predicate<String> condition) {
                return new TransactionOperation(OperationType.CONDITIONAL_SEND, topic,
key, value, null, condition);
            }
        }

        /**
         * Message transformer interface
         */
        @FunctionalInterface
        public interface MessageTransformer {
            String transform(String input);
        }

        /**
         * Financial transaction example using EOS
         */
        public static class FinancialTransactionProcessor {

            private final TransactionalProducer producer;

            public FinancialTransactionProcessor(String transactionId) {
                this.producer = new TransactionalProducer(transactionId);
            }

            /**
             * Process financial transfer with exactly-once semantics
             */
            public void processTransfer(String fromAccount, String toAccount, double
```

```
amount) {
            List<TransactionOperation> operations = Arrays.asList(
                // Debit from source account
                TransactionOperation.sendMessage("account-debits", fromAccount,
                    createDebitMessage(fromAccount, amount)),

                // Credit to destination account
                TransactionOperation.sendMessage("account-credits", toAccount,
                    createCreditMessage(toAccount, amount)),

                // Log transaction
                TransactionOperation.sendMessage("transaction-log",
                    fromAccount + "->" + toAccount,
                    createTransactionLog(fromAccount, toAccount, amount)),

                // Conditional fraud alert (if amount > threshold)
                TransactionOperation.conditionalSend("fraud-alerts",
                    fromAccount, createFraudAlert(fromAccount, toAccount, amount),
                    (value) -> amount > 10000)
            );

            try {
                producer.executeAtomicTransaction(operations);
                System.out.printf("Successfully processed transfer: %s -> %s,
amount: %.2f%n",
                    fromAccount, toAccount, amount);
            } catch (Exception e) {
                System.err.printf("Failed to process transfer: %s%n",
e.getMessage());
                throw e;
            }
        }

        private String createDebitMessage(String account, double amount) {
            return String.format("
{\"account\":\"%s\",\"type\":\"debit\",\"amount\":%.2f,\"timestamp\":%d}",
                account, amount, System.currentTimeMillis());
        }

        private String createCreditMessage(String account, double amount) {
            return String.format("
{\"account\":\"%s\",\"type\":\"credit\",\"amount\":%.2f,\"timestamp\":%d}",
                account, amount, System.currentTimeMillis());
        }

        private String createTransactionLog(String from, String to, double amount)
{
            return String.format("
{\"from\":\"%s\",\"to\":\"%s\",\"amount\":%.2f,\"timestamp\":%d}",
                from, to, amount, System.currentTimeMillis());
        }

        private String createFraudAlert(String from, String to, double amount) {
            return String.format("
```

```
{\"from\":\"%s\",\"to\":\"%s\",\"amount\":%.2f,\"alert\":\"high_amount\",\"timesta
mp\":%d}",
                from, to, amount, System.currentTimeMillis());
        }
    }
}
```

Exactly-Once Processing with Streams

Advanced Streams with EOS

```
import org.apache.kafka.streams.*;
import org.apache.kafka.streams.kstream.*;
import org.apache.kafka.streams.state.*;
import org.apache.kafka.common.serialization.Serdes;
import java.time.Duration;
import java.util.Properties;

/**
 * Exactly-Once Semantics with Kafka Streams
 */
public class StreamsEOSExample {

    /**
     * E-commerce order processing with exactly-once guarantees
     */
    public static class OrderProcessingStreams {

        public static void main(String[] args) {
            Properties streamsProps = createStreamsProps("order-processing-eos");

            StreamsBuilder builder = new StreamsBuilder();
            Topology topology = buildOrderProcessingTopology(builder);

            KafkaStreams streams = new KafkaStreams(topology, streamsProps);

            // Add shutdown hook
            Runtime.getRuntime().addShutdownHook(new Thread(() -> {
                streams.close(Duration.ofSeconds(10));
            }));

            streams.start();
        }

        private static Properties createStreamsProps(String applicationId) {
            Properties props = new Properties();

            // Basic configuration
            props.put(StreamsConfig.APPLICATION_ID_CONFIG, applicationId);
            props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(StreamsConfig.CLIENT_ID_CONFIG, applicationId + "-client");
```

```
            // Exactly-once configuration
            props.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG,
                StreamsConfig.EXACTLY_ONCE_V2);

            // Serialization
            props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
                Serdes.String().getClass().getName());
            props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
                Serdes.String().getClass().getName());

            // Performance and reliability
            props.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 1000);
            props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 64 * 1024 *
1024);
            props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 4);

            // State store configuration
            props.put(StreamsConfig.STATE_DIR_CONFIG, "/tmp/kafka-streams/" +
applicationId);

            return props;
        }

        private static Topology buildOrderProcessingTopology(StreamsBuilder
builder) {

            // Input streams
            KStream<String, String> rawOrders = builder.stream("raw-orders");
            KStream<String, String> paymentEvents = builder.stream("payment-
events");
            KStream<String, String> inventoryUpdates = builder.stream("inventory-
updates");

            // Add state stores
            StoreBuilder<KeyValueStore<String, String>> orderStateStore =
                Stores.keyValueStoreBuilder(
                    Stores.persistentKeyValueStore("order-states"),
                    Serdes.String(),
                    Serdes.String()
                );
            builder.addStateStore(orderStateStore);

            StoreBuilder<KeyValueStore<String, String>> inventoryStore =
                Stores.keyValueStoreBuilder(
                    Stores.persistentKeyValueStore("inventory-levels"),
                    Serdes.String(),
                    Serdes.String()
                );
            builder.addStateStore(inventoryStore);

            // Process orders with exactly-once semantics
            KStream<String, String> processedOrders = rawOrders
                .transform(() -> new OrderValidationTransformer(), "order-states")
```

```
                .filter((key, value) -> value != null);

            // Join with payments
            KStream<String, String> paidOrders = processedOrders
                .join(paymentEvents,
                    (order, payment) -> joinOrderWithPayment(order, payment),
JoinWindows.ofTimeDifferenceWithNoGrace(Duration.ofMinutes(5)));

            // Update inventory atomically
            KStream<String, String> fulfilledOrders = paidOrders
                .transform(() -> new InventoryUpdateTransformer(), "inventory-
levels");

            // Output to multiple topics atomically
            fulfilledOrders.to("fulfilled-orders");

            // Create order summary aggregations
            KTable<String, Long> orderCountsByCustomer = processedOrders
                .groupBy((orderId, orderData) -> extractCustomerId(orderData))
                .count(Materialized.as("customer-order-counts"));

            orderCountsByCustomer.toStream().to("customer-order-stats");

            // Create inventory alerts
            KStream<String, String> lowInventoryAlerts = inventoryUpdates
                .filter((productId, level) -> Integer.parseInt(level) < 10)
                .mapValues(level -> createLowInventoryAlert(level));

            lowInventoryAlerts.to("inventory-alerts");

            return builder.build();
        }

        private static String joinOrderWithPayment(String order, String payment) {
            // Combine order and payment information
            return String.format("
{\"order\":%s,\"payment\":%s,\"status\":\"paid\",\"timestamp\":%d}",
                order, payment, System.currentTimeMillis());
        }

        private static String extractCustomerId(String orderData) {
            // Extract customer ID from order JSON
            // Simplified implementation
            return "customer-" + orderData.hashCode() % 1000;
        }

        private static String createLowInventoryAlert(String level) {
            return String.format("
{\"alert\":\"low_inventory\",\"level\":%s,\"timestamp\":%d}",
                level, System.currentTimeMillis());
        }
    }
```

```
    /**
     * Order validation transformer with state
     */
    private static class OrderValidationTransformer
        implements Transformer<String, String, KeyValue<String, String>> {

        private ProcessorContext context;
        private KeyValueStore<String, String> stateStore;

        @Override
        public void init(ProcessorContext context) {
            this.context = context;
            this.stateStore = context.getStateStore("order-states");
        }

        @Override
        public KeyValue<String, String> transform(String orderId, String
orderData) {
            try {
                // Check for duplicate orders (idempotency)
                String existingOrder = stateStore.get(orderId);
                if (existingOrder != null) {
                    System.out.printf("Duplicate order detected: %s%n", orderId);
                    return null; // Filter out duplicate
                }

                // Validate order data
                if (!isValidOrder(orderData)) {
                    System.out.printf("Invalid order: %s%n", orderId);
                    return null;
                }

                // Store order state
                String validatedOrder = addValidationTimestamp(orderData);
                stateStore.put(orderId, validatedOrder);

                System.out.printf("Validated order: %s%n", orderId);
                return KeyValue.pair(orderId, validatedOrder);

            } catch (Exception e) {
                System.err.printf("Error processing order %s: %s%n", orderId,
e.getMessage());
                return null;
            }
        }

        private boolean isValidOrder(String orderData) {
            // Implement order validation logic
            return orderData != null && orderData.contains("customer") &&
orderData.contains("items");
        }

        private String addValidationTimestamp(String orderData) {
            // Add validation timestamp to order
```

```
            return orderData.replace("}", ",\"validated_at\":" +
System.currentTimeMillis() + "}");
        }

        @Override
        public void close() {
            // Cleanup if needed
        }
    }

    /**
     * Inventory update transformer with atomic updates
     */
    private static class InventoryUpdateTransformer
        implements Transformer<String, String, KeyValue<String, String>> {

        private ProcessorContext context;
        private KeyValueStore<String, String> inventoryStore;

        @Override
        public void init(ProcessorContext context) {
            this.context = context;
            this.inventoryStore = context.getStateStore("inventory-levels");
        }

        @Override
        public KeyValue<String, String> transform(String orderId, String
orderData) {
            try {
                // Extract items from order
                List<String> items = extractItemsFromOrder(orderData);

                // Check and update inventory atomically
                for (String item : items) {
                    String currentLevel = inventoryStore.get(item);
                    int level = currentLevel != null ?
Integer.parseInt(currentLevel) : 0;

                    if (level <= 0) {
                        System.err.printf("Insufficient inventory for item %s in
order %s%n", item, orderId);
                        return null; // Reject order
                    }

                    // Reduce inventory
                    inventoryStore.put(item, String.valueOf(level - 1));
                }

                // Add fulfillment timestamp
                String fulfilledOrder = addFulfillmentTimestamp(orderData);
                System.out.printf("Fulfilled order: %s%n", orderId);

                return KeyValue.pair(orderId, fulfilledOrder);
```

```
            } catch (Exception e) {
                System.err.printf("Error fulfilling order %s: %s%n", orderId,
e.getMessage());
                return null;
            }
        }

        private List<String> extractItemsFromOrder(String orderData) {
            // Simplified item extraction
            // In real implementation, parse JSON and extract item list
            return Arrays.asList("item1", "item2");
        }

        private String addFulfillmentTimestamp(String orderData) {
            return orderData.replace("}", ",\"fulfilled_at\":" +
System.currentTimeMillis() + "}");
        }

        @Override
        public void close() {
            // Cleanup if needed
        }
    }
}
```

🌐 Multi-Cluster & Replication

Simple Explanation

Multi-cluster Kafka deployments enable data replication across geographic regions, cloud providers, or environments for disaster recovery, data locality, compliance, and high availability.

Problem It Solves

- **Disaster Recovery**: Automatic failover to secondary clusters
- **Geographic Distribution**: Low-latency access across regions
- **Data Compliance**: Keep data within specific jurisdictions
- **Load Distribution**: Spread workload across multiple clusters
- **Migration**: Move between Kafka distributions or cloud providers

MirrorMaker 2.0

Advanced MirrorMaker 2.0 Implementation

```
import org.apache.kafka.connect.mirror.*;
import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.AdminClientConfig;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
```

```
import java.util.*;
import java.util.concurrent.ExecutionException;

/**
 * Comprehensive MirrorMaker 2.0 implementation with monitoring
 */
public class MirrorMaker2Implementation {

    /**
     * MirrorMaker 2.0 configuration builder
     */
    public static class MM2ConfigBuilder {

        public static Map<String, String> createActivePassiveDRConfig(
            String sourceCluster, String targetCluster,
            String sourceBootstrap, String targetBootstrap) {

            Map<String, String> config = new HashMap<>();

            // Cluster definitions
            config.put("clusters", sourceCluster + "," + targetCluster);
            config.put(sourceCluster + ".bootstrap.servers", sourceBootstrap);
            config.put(targetCluster + ".bootstrap.servers", targetBootstrap);

            // Replication flows
            config.put("replication.flows", sourceCluster + "->" + targetCluster);

            // Source connector configuration
            config.put(sourceCluster + "->" + targetCluster + ".enabled", "true");
            config.put(sourceCluster + "->" + targetCluster + ".topics",
"orders.*,payments.*,customers.*");
            config.put(sourceCluster + "->" + targetCluster + ".topics.blacklist",
".*internal.*");

            // Consumer group sync
            config.put(sourceCluster + "->" + targetCluster +
".sync.group.offsets.enabled", "true");
            config.put(sourceCluster + "->" + targetCluster +
".sync.group.offsets.interval.seconds", "60");

            // Heartbeat configuration
            config.put(sourceCluster + "->" + targetCluster +
".emit.heartbeats.enabled", "true");
            config.put(sourceCluster + "->" + targetCluster +
".emit.heartbeats.interval.seconds", "30");

            // Performance tuning
            config.put(sourceCluster + "->" + targetCluster +
".producer.batch.size", "65536");
            config.put(sourceCluster + "->" + targetCluster +
".producer.linger.ms", "10");
            config.put(sourceCluster + "->" + targetCluster +
".producer.compression.type", "lz4");
            config.put(sourceCluster + "->" + targetCluster +
```

```
".consumer.max.poll.records", "1000");

            // Error handling and retries
            config.put(sourceCluster + "->" + targetCluster + ".producer.retries",
"2147483647");
            config.put(sourceCluster + "->" + targetCluster +
".producer.enable.idempotence", "true");
            config.put(sourceCluster + "->" + targetCluster + ".task.max", "8");

            // Naming strategy
            config.put("replication.policy.class",
                "org.apache.kafka.connect.mirror.DefaultReplicationPolicy");
            config.put("replication.policy.separator", ".");

            // Security (if needed)
            addSecurityConfig(config, sourceCluster, targetCluster);

            return config;
        }

        /**
         * Create bidirectional active-active configuration
         */
        public static Map<String, String> createActiveActiveConfig(
            String cluster1, String cluster2,
            String bootstrap1, String bootstrap2) {

            Map<String, String> config = new HashMap<>();

            // Cluster definitions
            config.put("clusters", cluster1 + "," + cluster2);
            config.put(cluster1 + ".bootstrap.servers", bootstrap1);
            config.put(cluster2 + ".bootstrap.servers", bootstrap2);

            // Bidirectional replication flows
            config.put("replication.flows",
                cluster1 + "->" + cluster2 + "," + cluster2 + "->" + cluster1);

            // Configure both directions
            configureBidirectionalFlow(config, cluster1, cluster2);
            configureBidirectionalFlow(config, cluster2, cluster1);

            // Loop prevention
            config.put("replication.policy.class",
                "org.apache.kafka.connect.mirror.DefaultReplicationPolicy");
            config.put("checkpoints.topic.replication.factor", "3");
            config.put("heartbeats.topic.replication.factor", "3");
            config.put("offset-syncs.topic.replication.factor", "3");

            return config;
        }

        private static void configureBidirectionalFlow(Map<String, String> config,
                                                       String source, String target)
```

```
{
            String flow = source + "->" + target;

            config.put(flow + ".enabled", "true");
            config.put(flow + ".topics", ".*");
            config.put(flow + ".topics.blacklist",
                ".*\\.internal,.*heartbeats,.*checkpoints,.*offset-syncs");
            config.put(flow + ".sync.group.offsets.enabled", "true");
            config.put(flow + ".emit.heartbeats.enabled", "true");
            config.put(flow + ".task.max", "4");
        }

        private static void addSecurityConfig(Map<String, String> config,
                                              String sourceCluster, String
targetCluster) {
            // SASL/SSL configuration for both clusters
            config.put(sourceCluster + ".security.protocol", "SASL_SSL");
            config.put(sourceCluster + ".sasl.mechanism", "SCRAM-SHA-512");
            config.put(sourceCluster + ".sasl.jaas.config",
                "org.apache.kafka.common.security.scram.ScramLoginModule required
" +
                "username=\"source-user\" password=\"source-pass\";");

            config.put(targetCluster + ".security.protocol", "SASL_SSL");
            config.put(targetCluster + ".sasl.mechanism", "SCRAM-SHA-512");
            config.put(targetCluster + ".sasl.jaas.config",
                "org.apache.kafka.common.security.scram.ScramLoginModule required
" +
                "username=\"target-user\" password=\"target-pass\";");
        }
    }

    /**
     * MirrorMaker 2.0 monitoring and health checker
     */
    public static class MM2Monitor {

        private final AdminClient sourceAdmin;
        private final AdminClient targetAdmin;
        private final String sourceCluster;
        private final String targetCluster;

        public MM2Monitor(String sourceBootstrap, String targetBootstrap,
                          String sourceCluster, String targetCluster) {
            this.sourceCluster = sourceCluster;
            this.targetCluster = targetCluster;

            Properties sourceProps = new Properties();
            sourceProps.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
sourceBootstrap);
            this.sourceAdmin = AdminClient.create(sourceProps);

            Properties targetProps = new Properties();
            targetProps.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
```

```
targetBootstrap);
            this.targetAdmin = AdminClient.create(targetProps);
        }

        /**
         * Check replication lag for all topics
         */
        public Map<String, ReplicationLag> checkReplicationLag() {
            Map<String, ReplicationLag> lagMap = new HashMap<>();

            try {
                // Get source topics
                Set<String> sourceTopics = sourceAdmin.listTopics().names().get();

                for (String sourceTopic : sourceTopics) {
                    if (sourceTopic.startsWith("_")) continue; // Skip internal
topics

                    String mirroredTopic = sourceCluster + "." + sourceTopic;

                    try {
                        ReplicationLag lag = calculateTopicLag(sourceTopic,
mirroredTopic);
                        lagMap.put(sourceTopic, lag);
                    } catch (Exception e) {
                        System.err.printf("Failed to check lag for topic %s:
%s%n",
                            sourceTopic, e.getMessage());
                    }
                }

            } catch (Exception e) {
                System.err.printf("Failed to check replication lag: %s%n",
e.getMessage());
            }

            return lagMap;
        }

        private ReplicationLag calculateTopicLag(String sourceTopic, String
mirroredTopic)
            throws ExecutionException, InterruptedException {

            // Get partition count and offsets from source
            var sourceDesc =
sourceAdmin.describeTopics(Arrays.asList(sourceTopic))
                .values().get(sourceTopic).get();

            // Check if mirrored topic exists
            try {
                var targetDesc =
targetAdmin.describeTopics(Arrays.asList(mirroredTopic))
                    .values().get(mirroredTopic).get();
```

```
                // Calculate lag (simplified - in real implementation, check per-
partition)
                long sourceTotalMessages = getApproximateMessageCount(sourceAdmin,
sourceTopic);
                long targetTotalMessages = getApproximateMessageCount(targetAdmin,
mirroredTopic);

                long lag = sourceTotalMessages - targetTotalMessages;
                long latency = System.currentTimeMillis(); // Simplified

                return new ReplicationLag(sourceTopic, mirroredTopic, lag,
latency,
                    sourceDesc.partitions().size(),
targetDesc.partitions().size());

            } catch (Exception e) {
                // Mirrored topic doesn't exist
                return new ReplicationLag(sourceTopic, mirroredTopic, -1, -1,
                    sourceDesc.partitions().size(), 0);
            }
        }

        private long getApproximateMessageCount(AdminClient admin, String topic) {
            // Simplified implementation - in reality, sum up high water marks
            return System.currentTimeMillis() % 10000; // Mock value
        }

        /**
         * Check heartbeat connectivity
         */
        public boolean checkHeartbeats() {
            try {
                String heartbeatTopic = sourceCluster + ".heartbeats";

                // Check if heartbeat topic exists in target cluster
                Set<String> targetTopics = targetAdmin.listTopics().names().get();

                if (!targetTopics.contains(heartbeatTopic)) {
                    System.err.println("Heartbeat topic not found: " +
heartbeatTopic);
                    return false;
                }

                // In real implementation, check recent heartbeat messages
                System.out.println("Heartbeat connectivity OK");
                return true;

            } catch (Exception e) {
                System.err.printf("Heartbeat check failed: %s%n", e.getMessage());
                return false;
            }
        }

        /**
```

```
         * Generate monitoring report
         */
        public void generateMonitoringReport() {
            System.out.println("=== MirrorMaker 2.0 Monitoring Report ===");
            System.out.printf("Source Cluster: %s%n", sourceCluster);
            System.out.printf("Target Cluster: %s%n", targetCluster);
            System.out.printf("Report Time: %s%n", new Date());
            System.out.println();

            // Check heartbeats
            boolean heartbeatsOK = checkHeartbeats();
            System.out.printf("Heartbeats Status: %s%n", heartbeatsOK ? "OK" :
"FAILED");

            // Check replication lag
            Map<String, ReplicationLag> lagMap = checkReplicationLag();
            System.out.printf("Topics Monitored: %d%n", lagMap.size());

            System.out.println("\nReplication Lag Details:");
            for (Map.Entry<String, ReplicationLag> entry : lagMap.entrySet()) {
                ReplicationLag lag = entry.getValue();
                System.out.printf("  %s: lag=%d messages, partitions=%d->%d%n",
                    lag.sourceTopic, lag.messageLag, lag.sourcePartitions,
lag.targetPartitions);

                if (lag.messageLag > 10000) {
                    System.out.printf("    WARNING: High lag detected for %s%n",
lag.sourceTopic);
                }
                if (lag.sourcePartitions != lag.targetPartitions) {
                    System.out.printf("    WARNING: Partition count mismatch for
%s%n", lag.sourceTopic);
                }
            }
        }

        public void close() {
            sourceAdmin.close();
            targetAdmin.close();
        }
    }

    /**
     * Replication lag information
     */
    public record ReplicationLag(
        String sourceTopic,
        String mirroredTopic,
        long messageLag,
        long latencyMs,
        int sourcePartitions,
        int targetPartitions
    ) {}
```

```
    /**
     * Disaster recovery coordinator
     */
    public static class DisasterRecoveryCoordinator {

        private final String primaryCluster;
        private final String drCluster;
        private final MM2Monitor monitor;

        public DisasterRecoveryCoordinator(String primaryBootstrap, String
drBootstrap,
                                           String primaryCluster, String drCluster)
{
            this.primaryCluster = primaryCluster;
            this.drCluster = drCluster;
            this.monitor = new MM2Monitor(primaryBootstrap, drBootstrap,
                primaryCluster, drCluster);
        }

        /**
         * Execute disaster recovery failover
         */
        public void executeFailover() {
            System.out.println("=== DISASTER RECOVERY FAILOVER INITIATED ===");

            try {
                // 1. Stop accepting new writes to primary (application-level)
                System.out.println("Step 1: Stopping writes to primary
cluster...");
                notifyApplicationsToStopWrites();

                // 2. Wait for replication to catch up
                System.out.println("Step 2: Waiting for replication to catch
up...");
                waitForReplicationCatchup();

                // 3. Verify data consistency
                System.out.println("Step 3: Verifying data consistency...");
                boolean consistent = verifyDataConsistency();
                if (!consistent) {
                    System.err.println("WARNING: Data consistency issues
detected!");
                }

                // 4. Update DNS/load balancer to point to DR cluster
                System.out.println("Step 4: Updating DNS to point to DR
cluster...");
                updateDNSToPointToDR();

                // 5. Start applications on DR cluster
                System.out.println("Step 5: Starting applications on DR
cluster...");
                startApplicationsOnDR();
```

```
                System.out.println("=== FAILOVER COMPLETED ===");

            } catch (Exception e) {
                System.err.printf("FAILOVER FAILED: %s%n", e.getMessage());
                throw new RuntimeException("Disaster recovery failover failed",
e);
            }
        }

        private void notifyApplicationsToStopWrites() {
            // Implementation would notify all producer applications
            System.out.println("  - Notified producer applications to stop");
        }

        private void waitForReplicationCatchup() throws InterruptedException {
            int attempts = 0;
            int maxAttempts = 30; // 5 minutes with 10-second intervals

            while (attempts < maxAttempts) {
                Map<String, ReplicationLag> lagMap =
monitor.checkReplicationLag();

                boolean caughtUp = lagMap.values().stream()
                    .allMatch(lag -> lag.messageLag < 100); // Allow small lag

                if (caughtUp) {
                    System.out.println("  - Replication caught up successfully");
                    return;
                }

                System.out.printf("  - Waiting for catchup, attempt %d/%d%n",
                    attempts + 1, maxAttempts);
                Thread.sleep(10000); // Wait 10 seconds
                attempts++;
            }

            System.err.println("  - WARNING: Replication did not fully catch up");
        }

        private boolean verifyDataConsistency() {
            // In real implementation, verify key business metrics
            Map<String, ReplicationLag> lagMap = monitor.checkReplicationLag();

            for (ReplicationLag lag : lagMap.values()) {
                if (lag.messageLag > 1000) { // Allow some tolerance
                    return false;
                }
                if (lag.sourcePartitions != lag.targetPartitions) {
                    return false;
                }
            }

            System.out.println("  - Data consistency verified");
            return true;
```

```
        }

        private void updateDNSToPointToDR() {
            // Implementation would update DNS records
            System.out.println("  - DNS updated to point to DR cluster");
        }

        private void startApplicationsOnDR() {
            // Implementation would start consumer applications on DR cluster
            System.out.println("  - Applications started on DR cluster");
        }
    }
}
```

Disaster Recovery Patterns

Comprehensive DR Implementation

```
import java.util.*;
import java.util.concurrent.*;
import java.time.Duration;
import java.time.Instant;

/**
 * Comprehensive disaster recovery patterns for Kafka
 */
public class KafkaDisasterRecovery {

    /**
     * RTO/RPO monitoring and alerting
     */
    public static class DRMetricsCollector {

        private final ScheduledExecutorService scheduler =
            Executors.newScheduledThreadPool(2);
        private final List<DRMetric> metrics = new ArrayList<>();

        public void startMonitoring(MM2Monitor monitor) {

            // RTO monitoring (Recovery Time Objective)
            scheduler.scheduleAtFixedRate(() -> {
                try {
                    boolean heartbeatsOK = monitor.checkHeartbeats();
                    if (!heartbeatsOK) {
                        recordOutage(Instant.now());
                    }
                } catch (Exception e) {
                    System.err.printf("RTO monitoring failed: %s%n",
e.getMessage());
                }
            }, 0, 30, TimeUnit.SECONDS);
```

```
            // RPO monitoring (Recovery Point Objective)
            scheduler.scheduleAtFixedRate(() -> {
                try {
                    Map<String, ReplicationLag> lagMap =
monitor.checkReplicationLag();
                    recordLagMetrics(lagMap);
                } catch (Exception e) {
                    System.err.printf("RPO monitoring failed: %s%n",
e.getMessage());
                }
            }, 0, 60, TimeUnit.SECONDS);
        }

        private void recordOutage(Instant outageTime) {
            DRMetric metric = new DRMetric(
                MetricType.OUTAGE,
                outageTime,
                "Primary cluster heartbeat failure",
                Map.of("severity", "high")
            );

            metrics.add(metric);
            triggerAlert(metric);
        }

        private void recordLagMetrics(Map<String, ReplicationLag> lagMap) {
            for (Map.Entry<String, ReplicationLag> entry : lagMap.entrySet()) {
                ReplicationLag lag = entry.getValue();

                if (lag.messageLag > 10000) { // High lag threshold
                    DRMetric metric = new DRMetric(
                        MetricType.HIGH_LAG,
                        Instant.now(),
                        String.format("High replication lag for topic %s",
lag.sourceTopic),
                        Map.of("topic", lag.sourceTopic, "lag",
String.valueOf(lag.messageLag))
                    );

                    metrics.add(metric);
                    triggerAlert(metric);
                }
            }
        }

        private void triggerAlert(DRMetric metric) {
            System.err.printf("DR ALERT: %s - %s%n", metric.type,
metric.description);

            // In real implementation, send to alerting system
            switch (metric.type) {
                case OUTAGE -> sendCriticalAlert(metric);
                case HIGH_LAG -> sendWarningAlert(metric);
```

```
                case DATA_INCONSISTENCY -> sendCriticalAlert(metric);
            }
        }

        private void sendCriticalAlert(DRMetric metric) {
            // Send to PagerDuty, OpsGenie, etc.
            System.out.println("  -> Sent critical alert to on-call team");
        }

        private void sendWarningAlert(DRMetric metric) {
            // Send to Slack, email, etc.
            System.out.println("  -> Sent warning alert to operations team");
        }

        public void generateDRReport() {
            System.out.println("=== DISASTER RECOVERY METRICS REPORT ===");

            // Calculate RTO
            Duration maxOutageDuration = calculateMaxOutageDuration();
            System.out.printf("Maximum Outage Duration (RTO): %s%n",
maxOutageDuration);

            // Calculate RPO
            Duration maxDataLoss = calculateMaxDataLoss();
            System.out.printf("Maximum Potential Data Loss (RPO): %s%n",
maxDataLoss);

            // Availability metrics
            double availability = calculateAvailability();
            System.out.printf("Availability: %.4f%% %n", availability * 100);

            // Alert summary
            Map<MetricType, Long> alertCounts = metrics.stream()
                .collect(Collectors.groupingBy(m -> m.type,
Collectors.counting()));

            System.out.println("\nAlert Summary:");
            alertCounts.forEach((type, count) ->
                System.out.printf("  %s: %d alerts%n", type, count));
        }

        private Duration calculateMaxOutageDuration() {
            // Calculate from outage metrics
            return Duration.ofMinutes(5); // Mock
        }

        private Duration calculateMaxDataLoss() {
            // Calculate from lag metrics
            return Duration.ofSeconds(30); // Mock
        }

        private double calculateAvailability() {
            // Calculate uptime percentage
            return 0.9998; // Mock - 99.98%
```

```
        }
    }

    /**
     * DR metric record
     */
    public record DRMetric(
        MetricType type,
        Instant timestamp,
        String description,
        Map<String, String> attributes
    ) {}

    public enum MetricType {
        OUTAGE, HIGH_LAG, DATA_INCONSISTENCY, FAILOVER_EVENT
    }

    /**
     * Multi-region DR orchestrator
     */
    public static class MultiRegionDROrchestrator {

        private final Map<String, ClusterInfo> clusters;
        private String primaryRegion;
        private final ExecutorService executorService =
            Executors.newFixedThreadPool(5);

        public MultiRegionDROrchestrator(Map<String, ClusterInfo> clusters,
                                         String primaryRegion) {
            this.clusters = clusters;
            this.primaryRegion = primaryRegion;
        }

        /**
         * Execute multi-region failover
         */
        public CompletableFuture<FailoverResult> executeMultiRegionFailover(
            String targetRegion, FailoverReason reason) {

            return CompletableFuture.supplyAsync(() -> {
                System.out.printf("=== MULTI-REGION FAILOVER: %s -> %s ===\n",
                    primaryRegion, targetRegion);

                Instant startTime = Instant.now();

                try {
                    // Pre-flight checks
                    validateTargetRegion(targetRegion);

                    // Coordinate failover across all regions
                    List<CompletableFuture<Void>> regionalFailovers = new
ArrayList<>();

                    for (Map.Entry<String, ClusterInfo> entry :
```

```
clusters.entrySet()) {
                        String region = entry.getKey();
                        if (!region.equals(targetRegion)) {
                            regionalFailovers.add(executeRegionalFailover(region,
targetRegion));
                        }
                    }

                    // Wait for all regional failovers to complete
                    CompletableFuture.allOf(regionalFailovers.toArray(new
CompletableFuture[0]))
                        .get(10, TimeUnit.MINUTES);

                    // Update primary region
                    String oldPrimary = primaryRegion;
                    primaryRegion = targetRegion;

                    // Verify failover success
                    verifyFailoverSuccess(targetRegion);

                    Duration failoverTime = Duration.between(startTime,
Instant.now());

                    return new FailoverResult(
                        true,
                        oldPrimary,
                        targetRegion,
                        failoverTime,
                        reason,
                        "Multi-region failover completed successfully"
                    );

                } catch (Exception e) {
                    Duration failoverTime = Duration.between(startTime,
Instant.now());

                    return new FailoverResult(
                        false,
                        primaryRegion,
                        targetRegion,
                        failoverTime,
                        reason,
                        "Failover failed: " + e.getMessage()
                    );
                }
            }, executorService);
        }

        private void validateTargetRegion(String targetRegion) {
            ClusterInfo targetCluster = clusters.get(targetRegion);
            if (targetCluster == null) {
                throw new IllegalArgumentException("Target region not configured:
" + targetRegion);
            }
```

```
            if (!targetCluster.healthy) {
                throw new IllegalStateException("Target region is not healthy: " +
targetRegion);
            }
        }

        private CompletableFuture<Void> executeRegionalFailover(String region,
String targetRegion) {
            return CompletableFuture.runAsync(() -> {
                System.out.printf("  Updating region %s to point to %s%n", region,
targetRegion);

                // Update application configurations
                updateApplicationConfig(region, targetRegion);

                // Update load balancers
                updateLoadBalancers(region, targetRegion);

                // Restart services if needed
                if (requiresServiceRestart(region)) {
                    restartServices(region);
                }

                System.out.printf("  Region %s failover completed%n", region);

            }, executorService);
        }

        private void updateApplicationConfig(String region, String targetRegion) {
            ClusterInfo targetCluster = clusters.get(targetRegion);
            // Update Kafka bootstrap servers in application configurations
            System.out.printf("    Updated %s apps to use %s%n", region,
targetCluster.bootstrapServers);
        }

        private void updateLoadBalancers(String region, String targetRegion) {
            // Update load balancer configurations
            System.out.printf("    Updated load balancers in %s%n", region);
        }

        private boolean requiresServiceRestart(String region) {
            // Determine if services need restart based on region configuration
            return clusters.get(region).requiresRestart;
        }

        private void restartServices(String region) {
            // Restart application services
            System.out.printf("    Restarted services in %s%n", region);
        }

        private void verifyFailoverSuccess(String targetRegion) {
            // Verify that the target region is accepting traffic
            ClusterInfo targetCluster = clusters.get(targetRegion);
```

```
            // In real implementation, check health endpoints, metrics, etc.
            System.out.printf("  Verified failover success to %s%n",
targetRegion);
        }

        /**
         * Automated DR testing
         */
        public void runDRTest() {
            System.out.println("=== DR TEST EXECUTION ===");

            // Choose a test target region
            String testTarget = clusters.keySet().stream()
                .filter(region -> !region.equals(primaryRegion))
                .findFirst()
                .orElseThrow(() -> new IllegalStateException("No test target
available"));

            try {
                // Execute test failover
                FailoverResult result = executeMultiRegionFailover(testTarget,
                    FailoverReason.DR_TEST).get(15, TimeUnit.MINUTES);

                if (result.success) {
                    System.out.println("✓ DR Test PASSED");

                    // Wait for test duration
                    Thread.sleep(30000); // 30 seconds

                    // Failback to original
                    executeMultiRegionFailover(result.sourceRegion,
                        FailoverReason.FAILBACK).get(15, TimeUnit.MINUTES);

                    System.out.println("✓ DR Failback PASSED");
                } else {
                    System.err.println("✗ DR Test FAILED: " + result.message);
                }

            } catch (Exception e) {
                System.err.printf("✗ DR Test ERROR: %s%n", e.getMessage());
            }
        }
    }

    /**
     * Cluster information
     */
    public static class ClusterInfo {
        public final String bootstrapServers;
        public final boolean healthy;
        public final boolean requiresRestart;
        public final Map<String, String> properties;
```

```
        public ClusterInfo(String bootstrapServers, boolean healthy,
                           boolean requiresRestart, Map<String, String> properties)
{
            this.bootstrapServers = bootstrapServers;
            this.healthy = healthy;
            this.requiresRestart = requiresRestart;
            this.properties = properties;
        }
    }

    /**
     * Failover result
     */
    public record FailoverResult(
        boolean success,
        String sourceRegion,
        String targetRegion,
        Duration executionTime,
        FailoverReason reason,
        String message
    ) {}

    public enum FailoverReason {
        DISASTER, PLANNED_MAINTENANCE, DR_TEST, FAILBACK, PERFORMANCE_ISSUE
    }
}
```

🏗 Event-Driven Architectures

Simple Explanation

Event-driven architecture (EDA) uses Kafka as the central nervous system for applications, where components communicate through events rather than direct calls. This enables loose coupling, scalability, and reactive systems.

Problem It Solves

- **Loose Coupling**: Services don't need to know about each other
- **Scalability**: Components can scale independently
- **Resilience**: Failures in one component don't cascade
- **Real-time Processing**: Immediate response to business events
- **Audit Trail**: Complete history of all business events

Event Sourcing with Kafka

Complete Event Sourcing Implementation

```
import java.util.*;
import java.util.concurrent.CompletableFuture;
```

```
import java.time.Instant;
import java.util.stream.Collectors;

/**
 * Comprehensive Event Sourcing implementation with Kafka
 */
public class EventSourcingWithKafka {

    /**
     * Event store using Kafka as the storage backend
     */
    public static class KafkaEventStore {

        private final KafkaProducer<String, String> producer;
        private final KafkaConsumer<String, String> consumer;
        private final String eventsTopic;

        public KafkaEventStore(String eventsTopic) {
            this.eventsTopic = eventsTopic;
            this.producer = new KafkaProducer<>(createProducerProps());
            this.consumer = new KafkaConsumer<>(createConsumerProps());
        }

        private Properties createProducerProps() {
            Properties props = new Properties();
            props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                "org.apache.kafka.common.serialization.StringSerializer");
            props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                "org.apache.kafka.common.serialization.StringSerializer");

            // Event sourcing requires strong guarantees
            props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
            props.put(ProducerConfig.ACKS_CONFIG, "all");
            props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);

            return props;
        }

        private Properties createConsumerProps() {
            Properties props = new Properties();
            props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            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");
            props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

            return props;
        }

        /**
         * Append event to the event store
         */
```

```
        public CompletableFuture<EventMetadata> appendEvent(DomainEvent event) {
            String eventJson = serializeEvent(event);

            ProducerRecord<String, String> record = new ProducerRecord<>(
                eventsTopic,
                event.aggregateId(),
                eventJson
            );

            // Add event metadata as headers
            record.headers().add("event-type",
event.getClass().getSimpleName().getBytes());
            record.headers().add("event-version", "1.0".getBytes());
            record.headers().add("timestamp",
event.timestamp().toString().getBytes());
            record.headers().add("correlation-id",
event.correlationId().getBytes());

            CompletableFuture<EventMetadata> future = new CompletableFuture<>();

            producer.send(record, (metadata, exception) -> {
                if (exception != null) {
                    future.completeExceptionally(exception);
                } else {
                    EventMetadata eventMeta = new EventMetadata(
                        metadata.topic(),
                        metadata.partition(),
                        metadata.offset(),
                        metadata.timestamp(),
                        event.eventId()
                    );
                    future.complete(eventMeta);
                }
            });

            return future;
        }

        /**
         * Read all events for a specific aggregate
         */
        public List<DomainEvent> getEventsForAggregate(String aggregateId) {
            return getEventsForAggregate(aggregateId, 0);
        }

        /**
         * Read events for aggregate from specific version
         */
        public List<DomainEvent> getEventsForAggregate(String aggregateId, long
fromVersion) {
            List<DomainEvent> events = new ArrayList<>();

            // Set consumer to read from beginning
            consumer.assign(getPartitionsForTopic(eventsTopic));
```

```
            consumer.seekToBeginning(consumer.assignment());

            try {
                boolean foundEvents = true;
                while (foundEvents) {
                    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
                    foundEvents = !records.isEmpty();

                    for (ConsumerRecord<String, String> record : records) {
                        if (aggregateId.equals(record.key())) {
                            DomainEvent event = deserializeEvent(record.value());
                            if (event.version() >= fromVersion) {
                                events.add(event);
                            }
                        }
                    }
                }
            } finally {
                consumer.unsubscribe();
            }

            return events.stream()
                .sorted(Comparator.comparing(DomainEvent::version))
                .collect(Collectors.toList());
        }

        /**
         * Create aggregate snapshot
         */
        public void createSnapshot(String aggregateId, AggregateSnapshot snapshot)
{
            String snapshotTopic = eventsTopic + "-snapshots";
            String snapshotJson = serializeSnapshot(snapshot);

            ProducerRecord<String, String> record = new ProducerRecord<>(
                snapshotTopic,
                aggregateId,
                snapshotJson
            );

            record.headers().add("snapshot-version",
                String.valueOf(snapshot.version()).getBytes());
            record.headers().add("snapshot-timestamp",
                snapshot.timestamp().toString().getBytes());

            producer.send(record);
        }

        private Collection<TopicPartition> getPartitionsForTopic(String topic) {
            // Get all partitions for the topic
            return Arrays.asList(new TopicPartition(topic, 0)); // Simplified
        }
```

```
        private String serializeEvent(DomainEvent event) {
            // JSON serialization of event
            return String.format(
                "
{\"eventId\":\"%s\",\"aggregateId\":\"%s\",\"version\":%d,\"timestamp\":\"%s\",\"c
orrelationId\":\"%s\",\"eventType\":\"%s\",\"eventData\":%s}",
                event.eventId(),
                event.aggregateId(),
                event.version(),
                event.timestamp(),
                event.correlationId(),
                event.getClass().getSimpleName(),
                event.toJson()
            );
        }

        private DomainEvent deserializeEvent(String eventJson) {
            // JSON deserialization to event object
            // Simplified implementation
            return new AccountCreatedEvent("", "", 0, Instant.now(), "", "", 0.0);
        }

        private String serializeSnapshot(AggregateSnapshot snapshot) {
            return snapshot.toJson();
        }
    }

    /**
     * Base domain event interface
     */
    public interface DomainEvent {
        String eventId();
        String aggregateId();
        long version();
        Instant timestamp();
        String correlationId();
        String toJson();
    }

    /**
     * Bank account events for event sourcing example
     */
    public record AccountCreatedEvent(
        String eventId,
        String aggregateId,
        long version,
        Instant timestamp,
        String correlationId,
        String accountHolderName,
        double initialBalance
    ) implements DomainEvent {

        @Override
        public String toJson() {
```

```
            return String.format("
{\"accountHolderName\":\"%s\",\"initialBalance\":%.2f}",
                accountHolderName, initialBalance);
        }
    }

    public record MoneyDepositedEvent(
        String eventId,
        String aggregateId,
        long version,
        Instant timestamp,
        String correlationId,
        double amount,
        String description
    ) implements DomainEvent {

        @Override
        public String toJson() {
            return String.format("{\"amount\":%.2f,\"description\":\"%s\"}",
                amount, description);
        }
    }

    public record MoneyWithdrawnEvent(
        String eventId,
        String aggregateId,
        long version,
        Instant timestamp,
        String correlationId,
        double amount,
        String description
    ) implements DomainEvent {

        @Override
        public String toJson() {
            return String.format("{\"amount\":%.2f,\"description\":\"%s\"}",
                amount, description);
        }
    }

    /**
     * Bank account aggregate using event sourcing
     */
    public static class BankAccount {

        private String accountId;
        private String accountHolderName;
        private double balance;
        private long version;
        private final List<DomainEvent> uncommittedEvents = new ArrayList<>();

        // Private constructor for reconstruction from events
        private BankAccount() {}
```

```
        /**
         * Create new account (command)
         */
        public static BankAccount createAccount(String accountId, String
holderName,
                                                double initialBalance, String
correlationId) {
            if (initialBalance < 0) {
                throw new IllegalArgumentException("Initial balance cannot be
negative");
            }

            BankAccount account = new BankAccount();

            AccountCreatedEvent event = new AccountCreatedEvent(
                UUID.randomUUID().toString(),
                accountId,
                1,
                Instant.now(),
                correlationId,
                holderName,
                initialBalance
            );

            account.applyEvent(event);
            return account;
        }

        /**
         * Deposit money (command)
         */
        public void deposit(double amount, String description, String
correlationId) {
            if (amount <= 0) {
                throw new IllegalArgumentException("Deposit amount must be
positive");
            }

            MoneyDepositedEvent event = new MoneyDepositedEvent(
                UUID.randomUUID().toString(),
                accountId,
                version + 1,
                Instant.now(),
                correlationId,
                amount,
                description
            );

            applyEvent(event);
        }

        /**
         * Withdraw money (command)
         */
```

```
        public void withdraw(double amount, String description, String
correlationId) {
            if (amount <= 0) {
                throw new IllegalArgumentException("Withdrawal amount must be
positive");
            }
            if (balance < amount) {
                throw new IllegalStateException("Insufficient funds");
            }

            MoneyWithdrawnEvent event = new MoneyWithdrawnEvent(
                UUID.randomUUID().toString(),
                accountId,
                version + 1,
                Instant.now(),
                correlationId,
                amount,
                description
            );

            applyEvent(event);
        }

        /**
         * Apply event to aggregate state
         */
        private void applyEvent(DomainEvent event) {
            // Apply the event to the current state
            switch (event) {
                case AccountCreatedEvent created -> {
                    this.accountId = created.aggregateId();
                    this.accountHolderName = created.accountHolderName();
                    this.balance = created.initialBalance();
                    this.version = created.version();
                }
                case MoneyDepositedEvent deposited -> {
                    this.balance += deposited.amount();
                    this.version = deposited.version();
                }
                case MoneyWithdrawnEvent withdrawn -> {
                    this.balance -= withdrawn.amount();
                    this.version = withdrawn.version();
                }
                default -> throw new IllegalArgumentException("Unknown event type:
" + event.getClass());
            }

            // Add to uncommitted events
            uncommittedEvents.add(event);
        }

        /**
         * Reconstruct aggregate from event history
         */
```

```
    public static BankAccount fromHistory(List<DomainEvent> events) {
        BankAccount account = new BankAccount();

        for (DomainEvent event : events) {
            account.applyEventFromHistory(event);
        }

        return account;
    }

    private void applyEventFromHistory(DomainEvent event) {
        // Apply without adding to uncommitted events
        switch (event) {
            case AccountCreatedEvent created -> {
                this.accountId = created.aggregateId();
                this.accountHolderName = created.accountHolderName();
                this.balance = created.initialBalance();
                this.version = created.version();
            }
            case MoneyDepositedEvent deposited -> {
                this.balance += deposited.amount();
                this.version = deposited.version();
            }
            case MoneyWithdrawnEvent withdrawn -> {
                this.balance -= withdrawn.amount();
                this.version = withdrawn.version();
            }
        }
    }

    /**
     * Get uncommitted events for persistence
     */
    public List<DomainEvent> getUncommittedEvents() {
        return new ArrayList<>(uncommittedEvents);
    }

    /**
     * Mark events as committed
     */
    public void markEventsAsCommitted() {
        uncommittedEvents.clear();
    }

    // Getters
    public String getAccountId() { return accountId; }
    public String getAccountHolderName() { return accountHolderName; }
    public double getBalance() { return balance; }
    public long getVersion() { return version; }
}

/**
 * Repository for event-sourced aggregates
 */
```

```
    public static class EventSourcedBankAccountRepository {

        private final KafkaEventStore eventStore;

        public EventSourcedBankAccountRepository(KafkaEventStore eventStore) {
            this.eventStore = eventStore;
        }

        /**
         * Save aggregate by persisting its events
         */
        public CompletableFuture<Void> save(BankAccount account) {
            List<DomainEvent> uncommittedEvents = account.getUncommittedEvents();

            if (uncommittedEvents.isEmpty()) {
                return CompletableFuture.completedFuture(null);
            }

            // Save all uncommitted events
            List<CompletableFuture<EventMetadata>> eventFutures =
uncommittedEvents.stream()
                .map(eventStore::appendEvent)
                .collect(Collectors.toList());

            return CompletableFuture.allOf(eventFutures.toArray(new
CompletableFuture[0]))
                .thenRun(() -> {
                    account.markEventsAsCommitted();
                    System.out.printf("Saved %d events for account %s%n",
                        uncommittedEvents.size(), account.getAccountId());
                });
        }

        /**
         * Load aggregate by replaying events
         */
        public CompletableFuture<BankAccount> findById(String accountId) {
            return CompletableFuture.supplyAsync(() -> {
                List<DomainEvent> events =
eventStore.getEventsForAggregate(accountId);

                if (events.isEmpty()) {
                    throw new IllegalArgumentException("Account not found: " +
accountId);
                }

                BankAccount account = BankAccount.fromHistory(events);
                System.out.printf("Loaded account %s from %d events%n", accountId,
events.size());

                return account;
            });
        }
    }
```

```
    /**
     * Event metadata
     */
    public record EventMetadata(
        String topic,
        int partition,
        long offset,
        long timestamp,
        String eventId
    ) {}

    /**
     * Aggregate snapshot for performance optimization
     */
    public interface AggregateSnapshot {
        String aggregateId();
        long version();
        Instant timestamp();
        String toJson();
    }

    public record BankAccountSnapshot(
        String aggregateId,
        long version,
        Instant timestamp,
        String accountHolderName,
        double balance
    ) implements AggregateSnapshot {

        @Override
        public String toJson() {
            return String.format(
                "{\"accountHolderName\":\"%s\",\"balance\":%.2f,\"version\":%d}",
                accountHolderName, balance, version
            );
        }
    }
}
```

CQRS with Kafka

Complete CQRS Implementation

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicReference;

/**
 * CQRS (Command Query Responsibility Segregation) with Kafka
 */
public class CQRSWithKafka {
```

```
    /**
     * Command side of CQRS - handles writes
     */
    public static class CommandSide {

        private final EventSourcedBankAccountRepository repository;
        private final KafkaEventStore eventStore;

        public CommandSide(KafkaEventStore eventStore) {
            this.eventStore = eventStore;
            this.repository = new EventSourcedBankAccountRepository(eventStore);
        }

        /**
         * Command handler for account creation
         */
        public CompletableFuture<String> handleCreateAccount(CreateAccountCommand
command) {
            return CompletableFuture.supplyAsync(() -> {
                try {
                    // Validate command
                    validateCreateAccountCommand(command);

                    // Create aggregate
                    BankAccount account = BankAccount.createAccount(
                        command.accountId(),
                        command.holderName(),
                        command.initialBalance(),
                        command.correlationId()
                    );

                    // Save events
                    repository.save(account).get();

                    System.out.printf("Account created: %s%n",
command.accountId());
                    return command.accountId();

                } catch (Exception e) {
                    throw new RuntimeException("Failed to create account", e);
                }
            });
        }

        /**
         * Command handler for deposits
         */
        public CompletableFuture<Void> handleDeposit(DepositMoneyCommand command)
{
            return repository.findById(command.accountId())
                .thenCompose(account -> {
                    try {
                        account.deposit(
```

```
                            command.amount(),
                            command.description(),
                            command.correlationId()
                        );

                        return repository.save(account);

                    } catch (Exception e) {
                        CompletableFuture<Void> failed = new CompletableFuture<>
();
                        failed.completeExceptionally(e);
                        return failed;
                    }
                });
        }

        /**
         * Command handler for withdrawals
         */
        public CompletableFuture<Void> handleWithdrawal(WithdrawMoneyCommand
command) {
            return repository.findById(command.accountId())
                .thenCompose(account -> {
                    try {
                        account.withdraw(
                            command.amount(),
                            command.description(),
                            command.correlationId()
                        );

                        return repository.save(account);

                    } catch (Exception e) {
                        CompletableFuture<Void> failed = new CompletableFuture<>
();
                        failed.completeExceptionally(e);
                        return failed;
                    }
                });
        }

        private void validateCreateAccountCommand(CreateAccountCommand command) {
            if (command.accountId() == null || command.accountId().isEmpty()) {
                throw new IllegalArgumentException("Account ID is required");
            }
            if (command.holderName() == null || command.holderName().isEmpty()) {
                throw new IllegalArgumentException("Account holder name is
required");
            }
            if (command.initialBalance() < 0) {
                throw new IllegalArgumentException("Initial balance cannot be
negative");
            }
        }
```

```
    }

    /**
     * Query side of CQRS - handles reads with optimized views
     */
    public static class QuerySide {

        private final Map<String, AccountView> accountViews = new
ConcurrentHashMap<>();
        private final Map<String, List<TransactionView>> transactionHistory = new
ConcurrentHashMap<>();
        private final KafkaConsumer<String, String> eventConsumer;
        private final AtomicReference<Thread> projectionThread = new
AtomicReference<>();
        private volatile boolean running = true;

        public QuerySide() {
            this.eventConsumer = new KafkaConsumer<>(createConsumerProps());
        }

        private Properties createConsumerProps() {
            Properties props = new Properties();
            props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(ConsumerConfig.GROUP_ID_CONFIG, "cqrs-query-side");
            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");
            props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

            return props;
        }

        /**
         * Start projecting events into read models
         */
        public void startProjections(String eventsTopic) {
            Thread thread = new Thread(() -> {
                eventConsumer.subscribe(Arrays.asList(eventsTopic));

                while (running) {
                    try {
                        ConsumerRecords<String, String> records =
                            eventConsumer.poll(Duration.ofMillis(100));

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

                    } catch (Exception e) {
                        System.err.printf("Error in event projection: %s%n",
e.getMessage());
                    }
                }
```

```
            });

            if (projectionThread.compareAndSet(null, thread)) {
                thread.start();
                System.out.println("Started CQRS event projections");
            }
        }

        private void processEventForProjection(ConsumerRecord<String, String>
record) {
            String accountId = record.key();
            DomainEvent event = deserializeEvent(record.value());

            switch (event) {
                case AccountCreatedEvent created ->
                    projectAccountCreated(accountId, created);
                case MoneyDepositedEvent deposited ->
                    projectMoneyDeposited(accountId, deposited);
                case MoneyWithdrawnEvent withdrawn ->
                    projectMoneyWithdrawn(accountId, withdrawn);
            }
        }

        private void projectAccountCreated(String accountId, AccountCreatedEvent
event) {
            AccountView view = new AccountView(
                accountId,
                event.accountHolderName(),
                event.initialBalance(),
                event.timestamp(),
                event.timestamp(),
                1
            );

            accountViews.put(accountId, view);

            // Add to transaction history
            TransactionView transaction = new TransactionView(
                event.eventId(),
                accountId,
                "ACCOUNT_CREATED",
                0,
                event.initialBalance(),
                event.initialBalance(),
                "Account opened",
                event.timestamp()
            );

            transactionHistory.computeIfAbsent(accountId, k -> new ArrayList<>())
                .add(transaction);

            System.out.printf("Projected account creation: %s%n", accountId);
        }
```

```
        private void projectMoneyDeposited(String accountId, MoneyDepositedEvent
event) {
            AccountView currentView = accountViews.get(accountId);
            if (currentView != null) {
                AccountView updatedView = new AccountView(
                    accountId,
                    currentView.holderName(),
                    currentView.balance() + event.amount(),
                    currentView.createdAt(),
                    event.timestamp(),
                    currentView.version() + 1
                );

                accountViews.put(accountId, updatedView);

                // Add transaction
                TransactionView transaction = new TransactionView(
                    event.eventId(),
                    accountId,
                    "DEPOSIT",
                    event.amount(),
                    0,
                    updatedView.balance(),
                    event.description(),
                    event.timestamp()
                );

                transactionHistory.get(accountId).add(transaction);

                System.out.printf("Projected deposit: %s, amount: %.2f%n",
accountId, event.amount());
            }
        }

        private void projectMoneyWithdrawn(String accountId, MoneyWithdrawnEvent
event) {
            AccountView currentView = accountViews.get(accountId);
            if (currentView != null) {
                AccountView updatedView = new AccountView(
                    accountId,
                    currentView.holderName(),
                    currentView.balance() - event.amount(),
                    currentView.createdAt(),
                    event.timestamp(),
                    currentView.version() + 1
                );

                accountViews.put(accountId, updatedView);

                // Add transaction
                TransactionView transaction = new TransactionView(
                    event.eventId(),
                    accountId,
                    "WITHDRAWAL",
```

```
                    0,
                    event.amount(),
                    updatedView.balance(),
                    event.description(),
                    event.timestamp()
                );

                transactionHistory.get(accountId).add(transaction);

                System.out.printf("Projected withdrawal: %s, amount: %.2f%n",
accountId, event.amount());
            }
        }

        /**
         * Query methods - optimized for reads
         */
        public Optional<AccountView> getAccount(String accountId) {
            return Optional.ofNullable(accountViews.get(accountId));
        }

        public List<AccountView> getAllAccounts() {
            return new ArrayList<>(accountViews.values());
        }

        public List<AccountView> getAccountsByHolderName(String holderName) {
            return accountViews.values().stream()
                .filter(account -> account.holderName().contains(holderName))
                .collect(Collectors.toList());
        }

        public List<TransactionView> getTransactionHistory(String accountId) {
            return transactionHistory.getOrDefault(accountId, new ArrayList<>())
                .stream()

.sorted(Comparator.comparing(TransactionView::timestamp).reversed())
                .collect(Collectors.toList());
        }

        public List<TransactionView> getRecentTransactions(String accountId, int
limit) {
            return getTransactionHistory(accountId).stream()
                .limit(limit)
                .collect(Collectors.toList());
        }

        /**
         * Analytics queries
         */
        public double getTotalBalance() {
            return accountViews.values().stream()
                .mapToDouble(AccountView::balance)
                .sum();
        }
```

```
        public Map<String, Double> getBalancesByCustomer() {
            return accountViews.values().stream()
                .collect(Collectors.toMap(
                    AccountView::holderName,
                    AccountView::balance,
                    Double::sum
                ));
        }

        private DomainEvent deserializeEvent(String eventJson) {
            // Simplified deserialization
            return new AccountCreatedEvent("", "", 0, Instant.now(), "", "", 0.0);
        }

        public void stop() {
            running = false;
            if (eventConsumer != null) {
                eventConsumer.close();
            }
        }
    }

    /**
     * Commands for the write side
     */
    public record CreateAccountCommand(
        String accountId,
        String holderName,
        double initialBalance,
        String correlationId
    ) {}

    public record DepositMoneyCommand(
        String accountId,
        double amount,
        String description,
        String correlationId
    ) {}

    public record WithdrawMoneyCommand(
        String accountId,
        double amount,
        String description,
        String correlationId
    ) {}

    /**
     * Views for the read side - optimized for queries
     */
    public record AccountView(
        String accountId,
        String holderName,
        double balance,
```

```
        Instant createdAt,
        Instant lastModified,
        long version
    ) {}

    public record TransactionView(
        String transactionId,
        String accountId,
        String type,
        double creditAmount,
        double debitAmount,
        double balanceAfter,
        String description,
        Instant timestamp
    ) {}

    /**
     * Complete CQRS system coordinator
     */
    public static class CQRSSystem {

        private final CommandSide commandSide;
        private final QuerySide querySide;
        private final KafkaEventStore eventStore;

        public CQRSSystem(String eventsTopic) {
            this.eventStore = new KafkaEventStore(eventsTopic);
            this.commandSide = new CommandSide(eventStore);
            this.querySide = new QuerySide();

            // Start query side projections
            querySide.startProjections(eventsTopic);
        }

        // Command operations
        public CompletableFuture<String> createAccount(String accountId, String
holderName,
                                                       double initialBalance) {
            return commandSide.handleCreateAccount(
                new CreateAccountCommand(accountId, holderName, initialBalance,
                    UUID.randomUUID().toString())
            );
        }

        public CompletableFuture<Void> deposit(String accountId, double amount,
String description) {
            return commandSide.handleDeposit(
                new DepositMoneyCommand(accountId, amount, description,
                    UUID.randomUUID().toString())
            );
        }

        public CompletableFuture<Void> withdraw(String accountId, double amount,
String description) {
```

```
            return commandSide.handleWithdrawal(
                new WithdrawMoneyCommand(accountId, amount, description,
                    UUID.randomUUID().toString())
            );
        }

        // Query operations
        public Optional<AccountView> getAccount(String accountId) {
            return querySide.getAccount(accountId);
        }

        public List<TransactionView> getTransactionHistory(String accountId) {
            return querySide.getTransactionHistory(accountId);
        }

        public double getTotalBalance() {
            return querySide.getTotalBalance();
        }

        public void shutdown() {
            querySide.stop();
        }
    }
}
```

🚀 Emerging Features (2023–2025)

KRaft-Only Deployments

Complete KRaft Implementation

```
import org.apache.kafka.clients.admin.*;
import org.apache.kafka.common.config.ConfigResource;
import java.util.*;
import java.util.concurrent.ExecutionException;

/**
 * KRaft-only deployment management and examples
 */
public class KRaftDeploymentExample {

    /**
     * KRaft cluster manager for modern deployments
     */
    public static class KRaftClusterManager {

        private final AdminClient adminClient;
        private final String clusterId;

        public KRaftClusterManager(String bootstrapServers) {
```

```
            Properties props = new Properties();
            props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);
            props.put(AdminClientConfig.CLIENT_ID_CONFIG, "kraft-cluster-
manager");
            this.adminClient = AdminClient.create(props);
            this.clusterId = getClusterId();
        }

        private String getClusterId() {
            try {
                DescribeClusterResult result = adminClient.describeCluster();
                return result.clusterId().get();
            } catch (Exception e) {
                throw new RuntimeException("Failed to get cluster ID", e);
            }
        }

        /**
         * Get KRaft controller information
         */
        public void displayKRaftControllerInfo() {
            try {
                System.out.println("=== KRaft Controller Information ===");

                DescribeClusterResult clusterResult =
adminClient.describeCluster();

                System.out.printf("Cluster ID: %s%n",
clusterResult.clusterId().get());
                System.out.printf("Controller ID: %s%n",
clusterResult.controller().get().id());

                Collection<Node> nodes = clusterResult.nodes().get();
                System.out.printf("Total Nodes: %d%n", nodes.size());

                System.out.println("\nCluster Nodes:");
                for (Node node : nodes) {
                    System.out.printf("  Node %d: %s:%d%n",
                        node.id(), node.host(), node.port());
                }

                // Get broker configurations to identify controllers vs brokers
                displayNodeRoles();

            } catch (Exception e) {
                System.err.printf("Failed to get controller info: %s%n",
e.getMessage());
            }
        }

        private void displayNodeRoles() {
            try {
                // In KRaft mode, we can identify node roles through
```

```
configurations
                System.out.println("\nNode Roles (KRaft Mode):");

                Collection<Node> nodes =
adminClient.describeCluster().nodes().get();

                for (Node node : nodes) {
                    try {
                        // Query node-specific configurations to determine role
                        ConfigResource brokerResource = new ConfigResource(
                            ConfigResource.Type.BROKER,
String.valueOf(node.id()));

                        DescribeConfigsResult configResult =
adminClient.describeConfigs(
                            Arrays.asList(brokerResource));

                        Config config =
configResult.all().get().get(brokerResource);

                        // Check for process.roles configuration
                        ConfigEntry processRoles = config.get("process.roles");

                        if (processRoles != null && processRoles.value() != null)
{
                            System.out.printf("  Node %d: %s (roles: %s)%n",
                                node.id(), node.host(), processRoles.value());
                        } else {
                            System.out.printf("  Node %d: %s (role: unknown)%n",
                                node.id(), node.host());
                        }

                    } catch (Exception e) {
                        System.out.printf("  Node %d: %s (role: unable to
determine)%n",
                            node.id(), node.host());
                    }
                }

            } catch (Exception e) {
                System.err.printf("Failed to determine node roles: %s%n",
e.getMessage());
            }
        }

        /**
         * Monitor KRaft metadata topic
         */
        public void monitorKRaftMetadata() {
            System.out.println("=== KRaft Metadata Monitoring ===");

            try {
                // List all topics to find KRaft metadata topics
                ListTopicsResult topicsResult = adminClient.listTopics(
```

```
                    new ListTopicsOptions().listInternal(true));

                Set<String> allTopics = topicsResult.names().get();

                // Find KRaft-specific topics
                List<String> kraftTopics = allTopics.stream()
                    .filter(topic -> topic.startsWith("@metadata") ||
                                     topic.startsWith("__cluster_metadata"))
                    .collect(Collectors.toList());

                System.out.printf("KRaft Metadata Topics Found: %d%n",
kraftTopics.size());

                for (String topic : kraftTopics) {
                    System.out.printf("  - %s%n", topic);

                    // Get topic description
                    try {
                        DescribeTopicsResult topicResult =
adminClient.describeTopics(
                            Arrays.asList(topic));
                        TopicDescription description =
topicResult.all().get().get(topic);

                        System.out.printf("    Partitions: %d%n",
description.partitions().size());
                        System.out.printf("    Internal: %s%n",
description.isInternal());

                    } catch (Exception e) {
                        System.out.printf("    Error getting details: %s%n",
e.getMessage());
                    }
                }

            } catch (Exception e) {
                System.err.printf("Failed to monitor KRaft metadata: %s%n",
e.getMessage());
            }
        }

        /**
         * Validate KRaft cluster health
         */
        public KRaftHealthReport validateKRaftHealth() {
            KRaftHealthReport.Builder reportBuilder = new
KRaftHealthReport.Builder();

            try {
                // Check cluster connectivity
                DescribeClusterResult clusterResult =
adminClient.describeCluster();
                Collection<Node> nodes = clusterResult.nodes().get();
```

```
                reportBuilder.totalNodes(nodes.size());
                reportBuilder.controllerId(clusterResult.controller().get().id());

                // Check if we have a valid controller
                boolean hasController = clusterResult.controller().get() != null;
                reportBuilder.hasController(hasController);

                // Check metadata topic health
                boolean metadataHealthy = checkMetadataTopicHealth();
                reportBuilder.metadataTopicHealthy(metadataHealthy);

                // Check leader election stability
                boolean leaderStable = checkLeaderElectionStability();
                reportBuilder.leaderElectionStable(leaderStable);

                reportBuilder.healthy(hasController && metadataHealthy &&
leaderStable);

            } catch (Exception e) {
                reportBuilder.healthy(false);
                reportBuilder.error(e.getMessage());
            }

            return reportBuilder.build();
        }

        private boolean checkMetadataTopicHealth() {
            try {
                // Check if metadata topic exists and is accessible
                ListTopicsResult topicsResult = adminClient.listTopics(
                    new ListTopicsOptions().listInternal(true));

                Set<String> topics = topicsResult.names().get();

                return topics.stream()
                    .anyMatch(topic -> topic.contains("metadata") ||
                                       topic.contains("cluster_metadata"));

            } catch (Exception e) {
                return false;
            }
        }

        private boolean checkLeaderElectionStability() {
            try {
                // Check controller stability over time
                Integer controllerId1 =
adminClient.describeCluster().controller().get().id();
                Thread.sleep(1000);
                Integer controllerId2 =
adminClient.describeCluster().controller().get().id();

                return controllerId1.equals(controllerId2);
```

```
            } catch (Exception e) {
                return false;
            }
        }

        /**
         * Create KRaft-optimized topic configuration
         */
        public void createKRaftOptimizedTopic(String topicName, int partitions,
                                              short replicationFactor) {
            try {
                NewTopic newTopic = new NewTopic(topicName, partitions,
replicationFactor);

                // KRaft-optimized configurations
                Map<String, String> configs = new HashMap<>();
                configs.put("cleanup.policy", "delete");
                configs.put("retention.ms", "604800000"); // 7 days
                configs.put("compression.type", "lz4");
                configs.put("min.insync.replicas", "2");

                // KRaft benefits from these optimizations
                configs.put("unclean.leader.election.enable", "false");
                configs.put("min.insync.replicas", String.valueOf(Math.max(1,
replicationFactor - 1)));

                newTopic.configs(configs);

                CreateTopicsResult result =
adminClient.createTopics(Arrays.asList(newTopic));
                result.all().get();

                System.out.printf("Created KRaft-optimized topic: %s%n",
topicName);

            } catch (Exception e) {
                System.err.printf("Failed to create topic %s: %s%n", topicName,
e.getMessage());
            }
        }

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

    /**
     * KRaft health report
     */
    public static class KRaftHealthReport {
        private final boolean healthy;
        private final int totalNodes;
        private final int controllerId;
        private final boolean hasController;
```

```
        private final boolean metadataTopicHealthy;
        private final boolean leaderElectionStable;
        private final String error;

        private KRaftHealthReport(Builder builder) {
            this.healthy = builder.healthy;
            this.totalNodes = builder.totalNodes;
            this.controllerId = builder.controllerId;
            this.hasController = builder.hasController;
            this.metadataTopicHealthy = builder.metadataTopicHealthy;
            this.leaderElectionStable = builder.leaderElectionStable;
            this.error = builder.error;
        }

        public void printReport() {
            System.out.println("=== KRaft Health Report ===");
            System.out.printf("Overall Health: %s%n", healthy ? "HEALTHY" :
"UNHEALTHY");
            System.out.printf("Total Nodes: %d%n", totalNodes);
            System.out.printf("Controller ID: %d%n", controllerId);
            System.out.printf("Has Controller: %s%n", hasController ? "Yes" :
"No");
            System.out.printf("Metadata Topic Healthy: %s%n", metadataTopicHealthy
? "Yes" : "No");
            System.out.printf("Leader Election Stable: %s%n", leaderElectionStable
? "Yes" : "No");

            if (error != null) {
                System.out.printf("Error: %s%n", error);
            }

            if (!healthy) {
                System.out.println("\n⚠  HEALTH CHECK FAILED - Investigation
Required");
            } else {
                System.out.println("\n✅ KRaft Cluster is Healthy");
            }
        }

        // Getters
        public boolean isHealthy() { return healthy; }
        public int getTotalNodes() { return totalNodes; }
        public int getControllerId() { return controllerId; }
        public boolean hasController() { return hasController; }
        public boolean isMetadataTopicHealthy() { return metadataTopicHealthy; }
        public boolean isLeaderElectionStable() { return leaderElectionStable; }
        public String getError() { return error; }

        public static class Builder {
            private boolean healthy;
            private int totalNodes;
            private int controllerId;
            private boolean hasController;
            private boolean metadataTopicHealthy;
```

```
            private boolean leaderElectionStable;
            private String error;

            public Builder healthy(boolean healthy) { this.healthy = healthy;
return this; }
            public Builder totalNodes(int totalNodes) { this.totalNodes =
totalNodes; return this; }
            public Builder controllerId(int controllerId) { this.controllerId =
controllerId; return this; }
            public Builder hasController(boolean hasController) {
this.hasController = hasController; return this; }
            public Builder metadataTopicHealthy(boolean healthy) {
this.metadataTopicHealthy = healthy; return this; }
            public Builder leaderElectionStable(boolean stable) {
this.leaderElectionStable = stable; return this; }
            public Builder error(String error) { this.error = error; return this;
}

            public KRaftHealthReport build() {
                return new KRaftHealthReport(this);
            }
        }
    }

    /**
     * KRaft migration utilities
     */
    public static class KRaftMigrationUtility {

        /**
         * Validate pre-migration requirements
         */
        public static MigrationReadiness validateMigrationReadiness(String
zkConnect,
                                                                     String
kafkaBootstrap) {
            MigrationReadiness.Builder builder = new MigrationReadiness.Builder();

            try {
                // Check Kafka version compatibility
                boolean versionCompatible =
checkKafkaVersionCompatibility(kafkaBootstrap);
                builder.versionCompatible(versionCompatible);

                // Check ZooKeeper connectivity
                boolean zkConnected = checkZooKeeperConnectivity(zkConnect);
                builder.zooKeeperConnected(zkConnected);

                // Check cluster stability
                boolean clusterStable = checkClusterStability(kafkaBootstrap);
                builder.clusterStable(clusterStable);

                // Check for ongoing operations
                boolean noOngoingOps = checkForOngoingOperations(kafkaBootstrap);
```

```
                builder.noOngoingOperations(noOngoingOps);

                builder.ready(versionCompatible && zkConnected && clusterStable &&
noOngoingOps);

            } catch (Exception e) {
                builder.ready(false);
                builder.error(e.getMessage());
            }

            return builder.build();
        }

        private static boolean checkKafkaVersionCompatibility(String
bootstrapServers) {
            try {
                Properties props = new Properties();
                props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);

                try (AdminClient admin = AdminClient.create(props)) {
                    // Check broker version through describe cluster
                    DescribeClusterResult result = admin.describeCluster();
                    Collection<Node> nodes = result.nodes().get();

                    // In real implementation, check actual Kafka version
                    return !nodes.isEmpty();
                }

            } catch (Exception e) {
                return false;
            }
        }

        private static boolean checkZooKeeperConnectivity(String zkConnect) {
            // In real implementation, test ZooKeeper connection
            return zkConnect != null && !zkConnect.isEmpty();
        }

        private static boolean checkClusterStability(String bootstrapServers) {
            try {
                Properties props = new Properties();
                props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);

                try (AdminClient admin = AdminClient.create(props)) {
                    // Check if all brokers are available
                    DescribeClusterResult result = admin.describeCluster();
                    Collection<Node> nodes = result.nodes().get();

                    return nodes.size() >= 3; // Minimum recommended for
production
                }
```

```
            } catch (Exception e) {
                return false;
            }
        }

        private static boolean checkForOngoingOperations(String bootstrapServers)
{
            // Check for ongoing partition reassignments, preferred replica
elections, etc.
            // Simplified implementation
            return true;
        }
    }

    /**
     * Migration readiness report
     */
    public static class MigrationReadiness {
        private final boolean ready;
        private final boolean versionCompatible;
        private final boolean zooKeeperConnected;
        private final boolean clusterStable;
        private final boolean noOngoingOperations;
        private final String error;

        private MigrationReadiness(Builder builder) {
            this.ready = builder.ready;
            this.versionCompatible = builder.versionCompatible;
            this.zooKeeperConnected = builder.zooKeeperConnected;
            this.clusterStable = builder.clusterStable;
            this.noOngoingOperations = builder.noOngoingOperations;
            this.error = builder.error;
        }

        public void printReport() {
            System.out.println("=== KRaft Migration Readiness Report ===");
            System.out.printf("Ready for Migration: %s%n", ready ? "YES" : "NO");
            System.out.printf("Version Compatible: %s%n", versionCompatible ? "✓"
: "✗");
            System.out.printf("ZooKeeper Connected: %s%n", zooKeeperConnected ?
"✓" : "✗");
            System.out.printf("Cluster Stable: %s%n", clusterStable ? "✓" : "✗");
            System.out.printf("No Ongoing Operations: %s%n", noOngoingOperations ?
"✓" : "✗");

            if (error != null) {
                System.out.printf("Error: %s%n", error);
            }

            if (!ready) {
                System.out.println("\n⚠  NOT READY FOR MIGRATION");
                System.out.println("Please address the failed checks before
proceeding.");
            } else {
```

```
                System.out.println("\n✅ READY FOR KRAFT MIGRATION");
            }
        }

        public boolean isReady() { return ready; }

        public static class Builder {
            private boolean ready;
            private boolean versionCompatible;
            private boolean zooKeeperConnected;
            private boolean clusterStable;
            private boolean noOngoingOperations;
            private String error;

            public Builder ready(boolean ready) { this.ready = ready; return this;
}
            public Builder versionCompatible(boolean compatible) {
this.versionCompatible = compatible; return this; }
            public Builder zooKeeperConnected(boolean connected) {
this.zooKeeperConnected = connected; return this; }
            public Builder clusterStable(boolean stable) { this.clusterStable =
stable; return this; }
            public Builder noOngoingOperations(boolean none) {
this.noOngoingOperations = none; return this; }
            public Builder error(String error) { this.error = error; return this;
}

            public MigrationReadiness build() {
                return new MigrationReadiness(this);
            }
        }
    }
}
```

Tiered Storage Implementation

Advanced Tiered Storage Management

```
import org.apache.kafka.clients.admin.*;
import org.apache.kafka.common.config.ConfigResource;
import java.util.*;
import java.util.concurrent.ExecutionException;

/**
 * Comprehensive Tiered Storage implementation and management
 */
public class TieredStorageImplementation {

    /**
     * Tiered Storage manager for Kafka 3.6+
     */
```

```
    public static class TieredStorageManager {

        private final AdminClient adminClient;

        public TieredStorageManager(String bootstrapServers) {
            Properties props = new Properties();
            props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);
            props.put(AdminClientConfig.CLIENT_ID_CONFIG, "tiered-storage-
manager");
            this.adminClient = AdminClient.create(props);
        }

        /**
         * Create topic with tiered storage enabled
         */
        public void createTieredStorageTopic(String topicName, int partitions,
                                              short replicationFactor,
                                              TieredStorageConfig tsConfig) {
            try {
                NewTopic newTopic = new NewTopic(topicName, partitions,
replicationFactor);

                Map<String, String> configs = new HashMap<>();

                // Enable tiered storage
                configs.put("remote.storage.enable", "true");

                // Local retention configuration (hot tier)
                configs.put("local.retention.ms",
String.valueOf(tsConfig.localRetentionMs()));
                configs.put("local.retention.bytes",
String.valueOf(tsConfig.localRetentionBytes()));

                // Total retention configuration (hot + cold tier)
                configs.put("retention.ms",
String.valueOf(tsConfig.totalRetentionMs()));
                configs.put("retention.bytes",
String.valueOf(tsConfig.totalRetentionBytes()));

                // Segment size affects tiering frequency
                configs.put("segment.ms", String.valueOf(tsConfig.segmentMs()));
                configs.put("segment.bytes",
String.valueOf(tsConfig.segmentBytes()));

                // Performance optimizations
                configs.put("compression.type", tsConfig.compressionType());
                configs.put("cleanup.policy", "delete");
                configs.put("min.insync.replicas", "2");

                newTopic.configs(configs);

                CreateTopicsResult result =
adminClient.createTopics(Arrays.asList(newTopic));
```

```
                result.all().get();

                System.out.printf("Created tiered storage topic: %s%n",
topicName);
                System.out.printf("  Local retention: %d ms%n",
tsConfig.localRetentionMs());
                System.out.printf("  Total retention: %d ms%n",
tsConfig.totalRetentionMs());

            } catch (Exception e) {
                System.err.printf("Failed to create tiered storage topic %s:
%s%n",
                    topicName, e.getMessage());
            }
        }

        /**
         * Enable tiered storage on existing topic
         */
        public void enableTieredStorageOnTopic(String topicName,
TieredStorageConfig tsConfig) {
            try {
                ConfigResource resource = new
ConfigResource(ConfigResource.Type.TOPIC, topicName);

                Map<ConfigResource, Map<String, String>> configs = new HashMap<>
();
                Map<String, String> topicConfigs = new HashMap<>();

                // Enable tiered storage
                topicConfigs.put("remote.storage.enable", "true");
                topicConfigs.put("local.retention.ms",
String.valueOf(tsConfig.localRetentionMs()));
                topicConfigs.put("local.retention.bytes",
String.valueOf(tsConfig.localRetentionBytes()));

                configs.put(resource, topicConfigs);

                AlterConfigsResult result = adminClient.alterConfigs(configs);
                result.all().get();

                System.out.printf("Enabled tiered storage on topic: %s%n",
topicName);

            } catch (Exception e) {
                System.err.printf("Failed to enable tiered storage on topic %s:
%s%n",
                    topicName, e.getMessage());
            }
        }

        /**
         * Monitor tiered storage metrics
         */
```

```
        public TieredStorageMetrics getTieredStorageMetrics(String topicName) {
            try {
                // Get topic description
                DescribeTopicsResult topicResult =
adminClient.describeTopics(Arrays.asList(topicName));
                TopicDescription description =
topicResult.all().get().get(topicName);

                // Get topic configuration
                ConfigResource resource = new
ConfigResource(ConfigResource.Type.TOPIC, topicName);
                DescribeConfigsResult configResult =
adminClient.describeConfigs(Arrays.asList(resource));
                Config config = configResult.all().get().get(resource);

                // Check if tiered storage is enabled
                boolean tieredEnabled = "true".equals(getConfigValue(config,
"remote.storage.enable"));

                if (!tieredEnabled) {
                    return new TieredStorageMetrics(topicName, false, 0, 0, 0, 0,
0, 0, "N/A");
                }

                // In real implementation, these would be actual JMX metrics
                long localDataSize = calculateLocalDataSize(topicName);
                long remoteDataSize = calculateRemoteDataSize(topicName);
                long totalDataSize = localDataSize + remoteDataSize;

                long localSegments = calculateLocalSegments(topicName);
                long remoteSegments = calculateRemoteSegments(topicName);
                long totalSegments = localSegments + remoteSegments;

                double tieredPercentage = totalDataSize > 0 ?
                    (double) remoteDataSize / totalDataSize * 100 : 0;

                String healthStatus = assessTieredStorageHealth(topicName);

                return new TieredStorageMetrics(
                    topicName,
                    true,
                    localDataSize,
                    remoteDataSize,
                    totalDataSize,
                    localSegments,
                    remoteSegments,
                    tieredPercentage,
                    healthStatus
                );

            } catch (Exception e) {
                System.err.printf("Failed to get tiered storage metrics for %s:
%s%n",
                    topicName, e.getMessage());
```

```
                return new TieredStorageMetrics(topicName, false, 0, 0, 0, 0, 0,
0, "ERROR");
            }
        }

        private String getConfigValue(Config config, String key) {
            ConfigEntry entry = config.get(key);
            return entry != null ? entry.value() : null;
        }

        // These would use actual JMX metrics in real implementation
        private long calculateLocalDataSize(String topicName) {
            return 1024 * 1024 * 100; // Mock: 100MB
        }

        private long calculateRemoteDataSize(String topicName) {
            return 1024 * 1024 * 500; // Mock: 500MB
        }

        private long calculateLocalSegments(String topicName) {
            return 10; // Mock
        }

        private long calculateRemoteSegments(String topicName) {
            return 45; // Mock
        }

        private String assessTieredStorageHealth(String topicName) {
            // In real implementation, check various health indicators
            return "HEALTHY";
        }

        /**
         * Generate tiered storage report for all topics
         */
        public void generateTieredStorageReport() {
            try {
                System.out.println("=== Tiered Storage Cluster Report ===");

                // Get all topics
                ListTopicsResult topicsResult = adminClient.listTopics();
                Set<String> allTopics = topicsResult.names().get();

                List<String> userTopics = allTopics.stream()
                    .filter(topic -> !topic.startsWith("_") &&
!topic.startsWith("@"))
                    .collect(Collectors.toList());

                System.out.printf("Total User Topics: %d%n", userTopics.size());

                long totalLocalData = 0;
                long totalRemoteData = 0;
                int tieredEnabledCount = 0;
```

```
                System.out.println("\nPer-Topic Tiered Storage Status:");
                System.out.printf("%-30s %-10s %-15s %-15s %-10s %-10s%n",
                    "Topic", "Enabled", "Local (MB)", "Remote (MB)", "Tiered%",
"Health");
                System.out.println("-".repeat(100));

                for (String topic : userTopics) {
                    TieredStorageMetrics metrics = getTieredStorageMetrics(topic);

                    if (metrics.tieredEnabled()) {
                        tieredEnabledCount++;
                        totalLocalData += metrics.localDataSizeBytes();
                        totalRemoteData += metrics.remoteDataSizeBytes();
                    }

                    System.out.printf("%-30s %-10s %-15s %-15s %-10.1f %-10s%n",
                        truncate(topic, 29),
                        metrics.tieredEnabled() ? "YES" : "NO",
                        formatBytes(metrics.localDataSizeBytes()),
                        formatBytes(metrics.remoteDataSizeBytes()),
                        metrics.tieredPercentage(),
                        metrics.healthStatus()
                    );
                }

                System.out.println("-".repeat(100));
                System.out.printf("Summary: %d/%d topics with tiered storage
enabled%n",
                    tieredEnabledCount, userTopics.size());
                System.out.printf("Total Local Data: %s%n",
formatBytes(totalLocalData));
                System.out.printf("Total Remote Data: %s%n",
formatBytes(totalRemoteData));

                if (totalLocalData + totalRemoteData > 0) {
                    double clusterTieredPercentage =
                        (double) totalRemoteData / (totalLocalData +
totalRemoteData) * 100;
                    System.out.printf("Cluster Tiered Percentage: %.1f%%%n",
clusterTieredPercentage);
                }

            } catch (Exception e) {
                System.err.printf("Failed to generate tiered storage report:
%s%n", e.getMessage());
            }
        }

        /**
         * Cost analysis for tiered storage
         */
        public void performCostAnalysis(Map<String, StorageCostConfig>
costConfigs) {
            System.out.println("=== Tiered Storage Cost Analysis ===");
```

```
            try {
                ListTopicsResult topicsResult = adminClient.listTopics();
                Set<String> allTopics = topicsResult.names().get();

                double totalLocalCost = 0;
                double totalRemoteCost = 0;
                double totalSavings = 0;

                System.out.printf("%-30s %-12s %-12s %-12s %-12s%n",
                    "Topic", "Local Cost", "Remote Cost", "Total Cost",
"Savings");
                System.out.println("-".repeat(80));

                for (String topic : allTopics) {
                    if (topic.startsWith("_") || topic.startsWith("@")) continue;

                    TieredStorageMetrics metrics = getTieredStorageMetrics(topic);
                    StorageCostConfig costConfig =
costConfigs.getOrDefault("default",
                        new StorageCostConfig(0.10, 0.02, 0.08)); // Default costs
per GB/month

                    double localCostGB = (double) metrics.localDataSizeBytes() /
(1024 * 1024 * 1024);
                    double remoteCostGB = (double) metrics.remoteDataSizeBytes() /
(1024 * 1024 * 1024);

                    double localCost = localCostGB *
costConfig.localStorageCostPerGBMonth();
                    double remoteCost = remoteCostGB *
costConfig.remoteStorageCostPerGBMonth();
                    double totalCost = localCost + remoteCost;

                    // Calculate savings compared to all-local storage
                    double allLocalCost = (localCostGB + remoteCostGB) *
costConfig.localStorageCostPerGBMonth();
                    double savings = allLocalCost - totalCost;

                    totalLocalCost += localCost;
                    totalRemoteCost += remoteCost;
                    totalSavings += savings;

                    System.out.printf("%-30s $%-11.2f $%-11.2f $%-11.2f
$%-11.2f%n",
                        truncate(topic, 29), localCost, remoteCost, totalCost,
savings);
                }

                System.out.println("-".repeat(80));
                System.out.printf("Total Monthly Cost: Local=$%.2f, Remote=$%.2f,
Total=$%.2f%n",
                    totalLocalCost, totalRemoteCost, totalLocalCost +
totalRemoteCost);
```

```
                System.out.printf("Monthly Savings from Tiered Storage: $%.2f%n",
totalSavings);
                System.out.printf("Annual Savings: $%.2f%n", totalSavings * 12);

            } catch (Exception e) {
                System.err.printf("Failed to perform cost analysis: %s%n",
e.getMessage());
            }
        }

        private String truncate(String str, int maxLength) {
            return str.length() > maxLength ? str.substring(0, maxLength - 2) +
".." : str;
        }

        private String formatBytes(long bytes) {
            if (bytes < 1024 * 1024) {
                return String.format("%.1f KB", bytes / 1024.0);
            } else if (bytes < 1024 * 1024 * 1024) {
                return String.format("%.1f MB", bytes / (1024.0 * 1024));
            } else {
                return String.format("%.1f GB", bytes / (1024.0 * 1024 * 1024));
            }
        }

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

    /**
     * Tiered storage configuration
     */
    public record TieredStorageConfig(
        long localRetentionMs,      // How long to keep data locally (hot)
        long localRetentionBytes,   // How much data to keep locally
        long totalRetentionMs,      // Total retention across both tiers
        long totalRetentionBytes,   // Total retention bytes across both tiers
        long segmentMs,             // Segment rolling time
        long segmentBytes,          // Segment size
        String compressionType      // Compression for efficiency
    ) {

        // Predefined configurations for common use cases
        public static TieredStorageConfig realTimeAnalytics() {
            return new TieredStorageConfig(
                Duration.ofHours(6).toMillis(),      // 6 hours local
                1024 * 1024 * 1024L,                 // 1GB local
                Duration.ofDays(30).toMillis(),      // 30 days total
                100L * 1024 * 1024 * 1024,           // 100GB total
                Duration.ofMinutes(15).toMillis(),   // 15 min segments
                64 * 1024 * 1024,                    // 64MB segments
                "lz4"
            );
```

```
        }

        public static TieredStorageConfig logAggregation() {
            return new TieredStorageConfig(
                Duration.ofDays(1).toMillis(),       // 1 day local
                5L * 1024 * 1024 * 1024,             // 5GB local
                Duration.ofDays(365).toMillis(),     // 1 year total
                1024L * 1024 * 1024 * 1024,          // 1TB total
                Duration.ofHours(1).toMillis(),      // 1 hour segments
                128 * 1024 * 1024,                   // 128MB segments
                "zstd"
            );
        }

        public static TieredStorageConfig eventSourcing() {
            return new TieredStorageConfig(
                Duration.ofDays(7).toMillis(),       // 1 week local
                10L * 1024 * 1024 * 1024,            // 10GB local
                Long.MAX_VALUE,                      // Infinite retention
                Long.MAX_VALUE,                      // Infinite size
                Duration.ofDays(1).toMillis(),       // Daily segments
                256 * 1024 * 1024,                   // 256MB segments
                "zstd"
            );
        }
    }

    /**
     * Tiered storage metrics
     */
    public record TieredStorageMetrics(
        String topicName,
        boolean tieredEnabled,
        long localDataSizeBytes,
        long remoteDataSizeBytes,
        long totalDataSizeBytes,
        long localSegments,
        long remoteSegments,
        double tieredPercentage,
        String healthStatus
    ) {}

    /**
     * Storage cost configuration
     */
    public record StorageCostConfig(
        double localStorageCostPerGBMonth,   // Cost per GB/month for local SSD
storage
        double remoteStorageCostPerGBMonth,  // Cost per GB/month for remote
object storage
        double datTransferCostPerGB          // Cost per GB for data transfer to
remote storage
    ) {}
```

```
    /**
     * Tiered storage producer with optimization
     */
    public static class TieredStorageOptimizedProducer {

        private final KafkaProducer<String, String> producer;

        public TieredStorageOptimizedProducer() {
            this.producer = new KafkaProducer<>(createOptimizedProps());
        }

        private Properties createOptimizedProps() {
            Properties props = new Properties();

            props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                "org.apache.kafka.common.serialization.StringSerializer");
            props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                "org.apache.kafka.common.serialization.StringSerializer");

            // Optimizations for tiered storage
            props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536);      // Larger
batches
            props.put(ProducerConfig.LINGER_MS_CONFIG, 20);          // Slightly
higher linger
            props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4"); // Fast
compression
            props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 128 * 1024 * 1024); //
128MB buffer

            // Reliability for long-term storage
            props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
            props.put(ProducerConfig.ACKS_CONFIG, "all");
            props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);

            return props;
        }

        /**
         * Send messages optimized for tiered storage
         */
        public void sendOptimizedBatch(String topic, List<KeyValueTimestamp>
messages) {
            try {
                List<Future<RecordMetadata>> futures = new ArrayList<>();

                for (KeyValueTimestamp kvt : messages) {
                    ProducerRecord<String, String> record = new ProducerRecord<>(
                        topic, kvt.key(), kvt.value());

                    // Add timestamp for better tiering decisions
                    if (kvt.timestamp() != null) {
                        record = new ProducerRecord<>(topic, null,
                            kvt.timestamp().toEpochMilli(), kvt.key(),
```

```
kvt.value());
                    }

                    // Add headers for tiering hints
                    record.headers().add("data-class",
getDataClass(kvt.value()).getBytes());
                    record.headers().add("retention-priority",
getRetentionPriority(kvt.value()).getBytes());

                    futures.add(producer.send(record));
                }

                // Wait for all to complete
                for (Future<RecordMetadata> future : futures) {
                    future.get();
                }

                System.out.printf("Sent optimized batch of %d messages to %s%n",
                    messages.size(), topic);

            } catch (Exception e) {
                System.err.printf("Failed to send optimized batch: %s%n",
e.getMessage());
            }
        }

        private String getDataClass(String value) {
            // Classify data for tiering decisions
            if (value.contains("error") || value.contains("exception")) {
                return "diagnostic";
            } else if (value.contains("transaction") || value.contains("payment"))
{
                return "transactional";
            } else {
                return "operational";
            }
        }

        private String getRetentionPriority(String value) {
            // Determine retention priority
            if (value.contains("audit") || value.contains("compliance")) {
                return "high";
            } else if (value.contains("debug") || value.contains("trace")) {
                return "low";
            } else {
                return "medium";
            }
        }

        public void close() {
            producer.close();
        }
    }
```

```
    public record KeyValueTimestamp(String key, String value, Instant timestamp)
{}
}
```

📊 Comprehensive Comparison Tables

Enterprise Features Comparison

Feature	Complexity	Performance Impact	Operational Overhead	Use Cases
Transactions	High	Medium	Medium	Financial systems, EOS processing
Event Sourcing	Very High	Low-Medium	High	Audit trails, CQRS, banking
Tiered Storage	Medium	Low	Low	Cost optimization, long retention
Multi-Cluster	High	Medium	Very High	DR, geographic distribution
KRaft Mode	Low	Low	Low	Modern deployments, simplified ops

Kafka Version Feature Matrix (2023-2025)

Version	KRaft Status	Tiered Storage	Transactions	Key Features
4.0	Default	GA	Enhanced	ZooKeeper removed, Queues
3.9	Production	GA	Stable	Dynamic KRaft quorums
3.8	Production	GA	Stable	Performance improvements
3.7	Production	GA	Stable	Enhanced monitoring
3.6	Production	**GA**	Stable	**Tiered Storage GA**
3.5	Production	Early Access	Stable	KRaft improvements

🚨 Common Pitfalls & Best Practices

Enterprise Deployment Pitfalls

❌ Common Mistakes

```
// DON'T - Enable tiered storage without proper cost analysis
Map<String, String> configs = new HashMap<>();
```

```
configs.put("remote.storage.enable", "true");
// Without understanding local vs remote retention costs

// DON'T - Use transactions without proper error handling
producer.beginTransaction();
producer.send(record); // No exception handling
producer.commitTransaction(); // May fail silently

// DON'T - Deploy KRaft without migration planning
// Directly switching from ZooKeeper to KRaft in production
```

☑ Best Practices

```
// DO - Analyze costs before enabling tiered storage
TieredStorageConfig config = TieredStorageConfig.realTimeAnalytics();
// Cost analysis performed, appropriate retention configured

// DO - Implement comprehensive transaction error handling
try {
    producer.beginTransaction();

    for (ProducerRecord record : batch) {
        producer.send(record);
    }

    producer.sendOffsetsToTransaction(offsets, groupId);
    producer.commitTransaction();

} catch (Exception e) {
    producer.abortTransaction();
    handleTransactionError(e);
}

// DO - Plan KRaft migration carefully
MigrationReadiness readiness = KRaftMigrationUtility
    .validateMigrationReadiness(zkConnect, kafkaBootstrap);
if (readiness.isReady()) {
    // Proceed with migration
} else {
    // Address issues first
}
```

🌍 Real-World Use Cases

Financial Services with Full EOS

```
/**
 * Complete financial trading system with exactly-once semantics
```

```
 */
public class FinancialTradingSystem {

    public void processTradeOrder(TradeOrder order) {
        // Multi-topic transaction with exactly-once guarantees
        List<TransactionOperation> operations = Arrays.asList(
            // Validate funds
            TransactionOperation.sendMessage("account-validations",
                order.accountId(), validateFunds(order)),

            // Reserve funds
            TransactionOperation.sendMessage("fund-reservations",
                order.accountId(), reserveFunds(order)),

            // Execute trade
            TransactionOperation.sendMessage("trade-executions",
                order.tradeId(), executeTrade(order)),

            // Update portfolio
            TransactionOperation.sendMessage("portfolio-updates",
                order.accountId(), updatePortfolio(order))
        );

        transactionalProducer.executeAtomicTransaction(operations);
    }
}
```

Global E-commerce with Multi-Region DR

```
/**
 * Global e-commerce platform with disaster recovery
 */
public class GlobalEcommercePlatform {

    public void setupGlobalDR() {
        Map<String, ClusterInfo> regions = Map.of(
            "us-east", new ClusterInfo("kafka-us-east:9092", true, false,
Map.of()),
            "eu-west", new ClusterInfo("kafka-eu-west:9092", true, false,
Map.of()),
            "ap-southeast", new ClusterInfo("kafka-ap-southeast:9092", true,
false, Map.of())
        );

        MultiRegionDROrchestrator orchestrator =
            new MultiRegionDROrchestrator(regions, "us-east");

        // Automated DR testing
        orchestrator.runDRTest();
    }
}
```

IoT Platform with Tiered Storage

```
/**
 * IoT sensor platform with cost-optimized tiered storage
 */
public class IoTPlatform {

    public void setupIoTTopics() {
        TieredStorageManager manager = new TieredStorageManager("localhost:9092");

        // High-frequency sensor data with aggressive tiering
        TieredStorageConfig sensorConfig = new TieredStorageConfig(
            Duration.ofHours(1).toMillis(),      // 1 hour local (hot analysis)
            512 * 1024 * 1024L,                  // 512MB local
            Duration.ofDays(1095).toMillis(),    // 3 years total retention
            10L * 1024 * 1024 * 1024 * 1024,     // 10TB total
            Duration.ofMinutes(10).toMillis(),   // 10-minute segments
            32 * 1024 * 1024,                    // 32MB segments
            "zstd"                               // High compression
        );

        manager.createTieredStorageTopic("sensor-data", 100, (short) 3,
sensorConfig);

        // Cost analysis
        Map<String, StorageCostConfig> costConfigs = Map.of(
            "default", new StorageCostConfig(0.15, 0.01, 0.02) // AWS pricing
        );

        manager.performCostAnalysis(costConfigs);
    }
}
```

Additional Resources

Official Documentation

- Kafka Transactions
- KRaft Mode
- Tiered Storage
- MirrorMaker 2.0

Advanced Learning

- Event Sourcing Patterns
- CQRS Implementation
- Kafka with Apache Flink

🛠️ Enterprise Tools

- Confluent Platform
- Apache Kafka on Kubernetes
- Monitoring with Prometheus

Last Updated: September 2025
Kafka Version Coverage: 3.6+ through 4.0
Enterprise Features: Production-Ready

> 💡 **Enterprise Tip**: The combination of KRaft mode, tiered storage, and exactly-once semantics represents the future of Kafka deployments. These features work together to provide simplified operations, cost optimization, and strong consistency guarantees that enterprise applications demand.