Kafka Reliability, Performance & Monitoring: Complete Developer Guide

A comprehensive refresher on Apache Kafka's reliability mechanisms, performance optimization strategies, and monitoring best practices. This README covers replication, performance tuning, and monitoring with detailed Java examples and production-ready implementations.

Table of Contents

Reliability

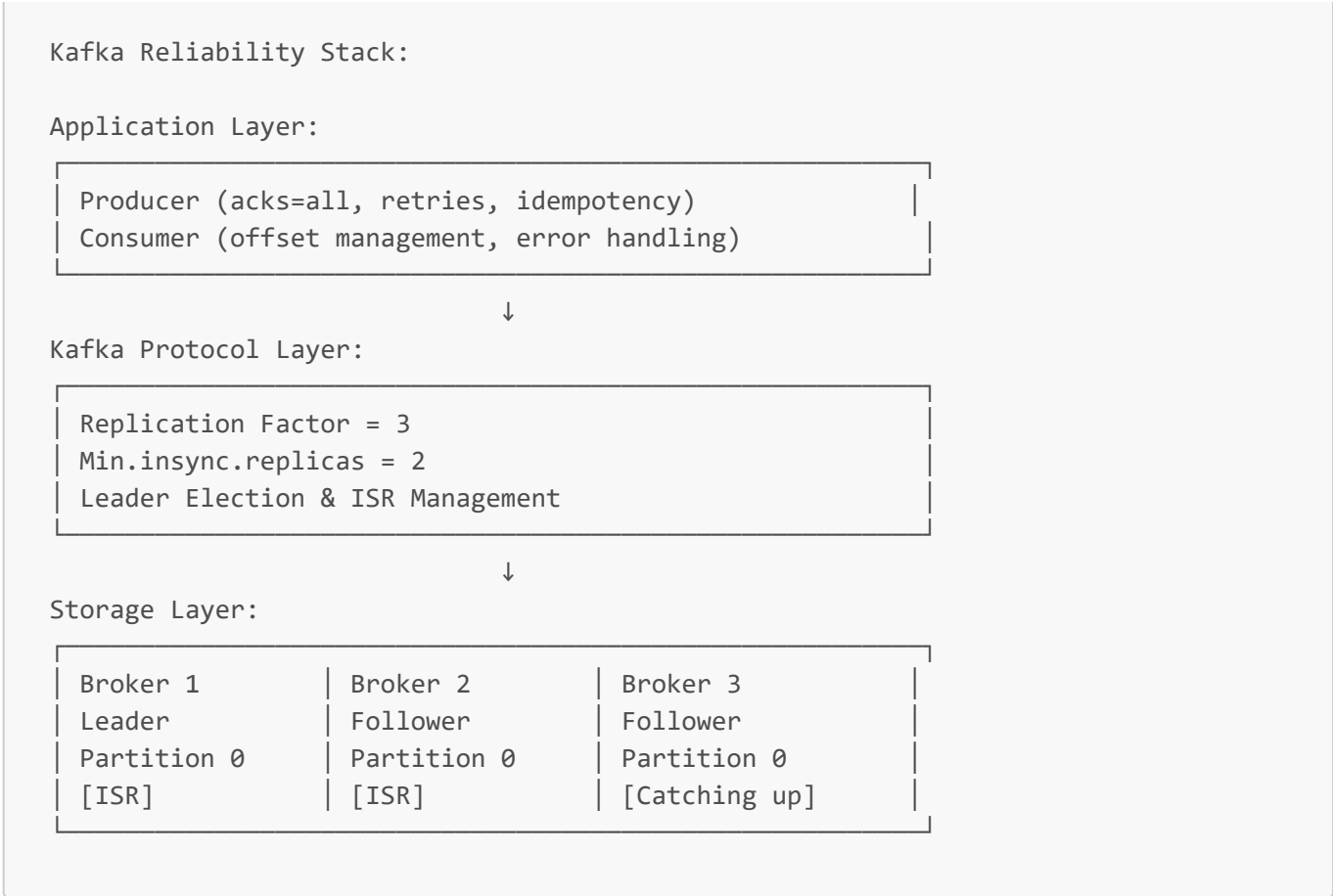
Simple Explanation

Kafka reliability ensures that data is never lost and remains available even during failures. It achieves this through replication, acknowledgment mechanisms, and retry policies that guarantee message delivery and data durability.

Problem It Solves

- **Data loss prevention**: Messages survive broker failures
- **High availability**: Service continues during outages
- **Consistency guarantees**: All consumers see the same data
- **Operational confidence**: Systems can depend on Kafka for critical data

Reliability Architecture



Replication Factor

Simple Explanation

Replication factor determines how many copies of each partition are maintained across different brokers. A replication factor of 3 means each partition has one leader and two followers, providing fault tolerance.

Problem It Solves

- **Broker failure tolerance**: Data survives individual broker failures
- **Load distribution**: Multiple brokers can serve read requests
- **Planned maintenance**: Brokers can be taken offline without data loss
- **Disaster recovery**: Multiple copies provide backup options

Internal Mechanisms

```
import org.apache.kafka.clients.admin.*;
import org.apache.kafka.common.config.ConfigResource;

/**
 * Comprehensive replication factor management and monitoring
 */
public class ReplicationFactorManager {

    private final AdminClient adminClient;
```

```
    public ReplicationFactorManager(Properties adminProps) {
        this.adminClient = AdminClient.create(adminProps);
    }

    /**
     * Optimal replication factor configuration for different environments
     */
    public static Properties getReplicationConfig(Environment env) {
        Properties config = new Properties();

        switch (env) {
            case DEVELOPMENT:
                // Lower replication for resource efficiency
                config.put("default.replication.factor", "1");
                config.put("min.insync.replicas", "1");
                config.put("offsets.topic.replication.factor", "1");
                config.put("transaction.state.log.replication.factor", "1");
                break;

            case TESTING:
                // Moderate replication for testing failure scenarios
                config.put("default.replication.factor", "2");
                config.put("min.insync.replicas", "1");
                config.put("offsets.topic.replication.factor", "2");
                config.put("transaction.state.log.replication.factor", "2");
                break;

            case STAGING:
                // Production-like configuration
                config.put("default.replication.factor", "3");
                config.put("min.insync.replicas", "2");
                config.put("offsets.topic.replication.factor", "3");
                config.put("transaction.state.log.replication.factor", "3");
                break;

            case PRODUCTION:
                // Maximum reliability
                config.put("default.replication.factor", "3");
                config.put("min.insync.replicas", "2");
                config.put("offsets.topic.replication.factor", "5"); // Higher for
critical metadata
                config.put("transaction.state.log.replication.factor", "5");

                // Additional safety measures
                config.put("unclean.leader.election.enable", "false");
                config.put("auto.leader.rebalance.enable", "true");
                config.put("leader.imbalance.per.broker.percentage", "5");
                break;
        }

        return config;
    }

    /**
```

```
     * Create topic with specific replication factor
     */
    public void createTopicWithReplication(String topicName, int partitions,
                                           short replicationFactor) {
        NewTopic newTopic = new NewTopic(topicName, partitions,
replicationFactor);

        // Configure topic-level settings
        Map<String, String> topicConfigs = new HashMap<>();
        topicConfigs.put("min.insync.replicas", String.valueOf(Math.max(1,
replicationFactor - 1)));
        topicConfigs.put("unclean.leader.election.enable", "false");
        topicConfigs.put("cleanup.policy", "delete");
        topicConfigs.put("retention.ms", "604800000"); // 7 days

        newTopic.configs(topicConfigs);

        try {
            CreateTopicsResult result =
adminClient.createTopics(Arrays.asList(newTopic));
            result.all().get(); // Wait for completion

            System.out.printf("Created topic '%s' with %d partitions and
replication factor %d%n",
                topicName, partitions, replicationFactor);

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

    /**
     * Monitor replication status across topics
     */
    public void monitorReplicationStatus() {
        try {
            // Get topic descriptions
            ListTopicsResult listTopicsResult = adminClient.listTopics();
            Set<String> topicNames = listTopicsResult.names().get();

            DescribeTopicsResult describeResult =
adminClient.describeTopics(topicNames);
            Map<String, TopicDescription> topicDescriptions =
describeResult.all().get();

            System.out.println("\n=== Replication Status Report ===");

            for (Map.Entry<String, TopicDescription> entry :
topicDescriptions.entrySet()) {
                String topicName = entry.getKey();
                TopicDescription description = entry.getValue();

                System.out.printf("\nTopic: %s%n", topicName);
```

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

                for (TopicPartitionInfo partition : description.partitions()) {
                    System.out.printf("  Partition %d:%n", partition.partition());
                    System.out.printf("    Leader: %s%n",
partition.leader().id());
                    System.out.printf("    Replicas: %s%n",
                        partition.replicas().stream()
                            .map(node -> String.valueOf(node.id()))
                            .collect(Collectors.joining(", ")));
                    System.out.printf("    ISR: %s%n",
                        partition.isr().stream()
                            .map(node -> String.valueOf(node.id()))
                            .collect(Collectors.joining(", ")));

                    // Check if under-replicated
                    if (partition.isr().size() < partition.replicas().size()) {
                        System.out.printf("    ⚠  UNDER-REPLICATED! (%d/%d in
ISR)%n",
                            partition.isr().size(), partition.replicas().size());
                    }
                }
            }

        } catch (Exception e) {
            System.err.println("Failed to monitor replication status: " +
e.getMessage());
        }
    }

    /**
     * Calculate cluster fault tolerance
     */
    public ClusterFaultTolerance calculateFaultTolerance() {
        try {
            DescribeClusterResult clusterResult = adminClient.describeCluster();
            Collection<Node> nodes = clusterResult.nodes().get();
            int totalBrokers = nodes.size();

            // Analyze topics for minimum replication factor
            ListTopicsResult listTopicsResult = adminClient.listTopics();
            Set<String> topicNames = listTopicsResult.names().get();

            DescribeTopicsResult describeResult =
adminClient.describeTopics(topicNames);
            Map<String, TopicDescription> topicDescriptions =
describeResult.all().get();

            int minReplicationFactor = Integer.MAX_VALUE;
            int minISRSize = Integer.MAX_VALUE;

            for (TopicDescription description : topicDescriptions.values()) {
                for (TopicPartitionInfo partition : description.partitions()) {
```

```
                    minReplicationFactor = Math.min(minReplicationFactor,
partition.replicas().size());
                    minISRSize = Math.min(minISRSize, partition.isr().size());
                }
            }

            int maxBrokerFailures = Math.min(
                minReplicationFactor - 1,  // Based on replication
                minISRSize - 1             // Based on current ISR
            );

            return new ClusterFaultTolerance(
                totalBrokers,
                minReplicationFactor,
                minISRSize,
                maxBrokerFailures
            );

        } catch (Exception e) {
            System.err.println("Failed to calculate fault tolerance: " +
e.getMessage());
            return new ClusterFaultTolerance(0, 0, 0, 0);
        }
    }

    public enum Environment {
        DEVELOPMENT, TESTING, STAGING, PRODUCTION
    }

    public static class ClusterFaultTolerance {
        final int totalBrokers;
        final int minReplicationFactor;
        final int currentMinISR;
        final int maxBrokerFailures;

        ClusterFaultTolerance(int totalBrokers, int minReplicationFactor,
                              int currentMinISR, int maxBrokerFailures) {
            this.totalBrokers = totalBrokers;
            this.minReplicationFactor = minReplicationFactor;
            this.currentMinISR = currentMinISR;
            this.maxBrokerFailures = maxBrokerFailures;
        }

        @Override
        public String toString() {
            return String.format(
                "Cluster Fault Tolerance:%n" +
                "  Total Brokers: %d%n" +
                "  Min Replication Factor: %d%n" +
                "  Current Min ISR: %d%n" +
                "  Max Broker Failures Tolerated: %d%n",
                totalBrokers, minReplicationFactor, currentMinISR,
maxBrokerFailures
            );
```

```
        }
    }

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

Min.insync.replicas

Simple Explanation

`min.insync.replicas` specifies the minimum number of replicas that must acknowledge a write before it's considered successful. It provides a balance between availability and consistency.

Problem It Solves

- **Consistency guarantees**: Ensures data is written to enough replicas
- **Availability trade-offs**: Prevents accepting writes when too few replicas are available
- **Data durability**: Reduces risk of data loss during failures
- **Split-brain prevention**: Avoids inconsistent states during network partitions

Configuration and Implementation

```
/**
 * Min.insync.replicas configuration and monitoring
 */
public class MinInsyncReplicasManager {

    /**
     * Optimal min.insync.replicas configurations for different scenarios
     */
    public static Properties getMinISRConfig(ReliabilityLevel level) {
        Properties config = new Properties();

        switch (level) {
            case MAXIMUM_AVAILABILITY:
                // Prioritize availability over consistency
                config.put("min.insync.replicas", "1");
                config.put("acks", "1"); // Only leader acknowledgment
                config.put("enable.idempotence", "false");
                break;

            case BALANCED:
                // Balance availability and consistency
                config.put("min.insync.replicas", "2");
                config.put("acks", "all");
                config.put("enable.idempotence", "true");
                config.put("retries", "10");
                break;
```

```
            case MAXIMUM_CONSISTENCY:
                // Prioritize consistency over availability
                config.put("min.insync.replicas", "3");
                config.put("acks", "all");
                config.put("enable.idempotence", "true");
                config.put("retries", "Integer.MAX_VALUE");
                break;

            case FINANCIAL_GRADE:
                // Zero tolerance for data loss
                config.put("min.insync.replicas", "3");
                config.put("acks", "all");
                config.put("enable.idempotence", "true");
                config.put("retries", "Integer.MAX_VALUE");
                config.put("max.in.flight.requests.per.connection", "1");
                config.put("delivery.timeout.ms", "120000"); // 2 minutes
                break;
        }

        return config;
    }

    /**
     * Demonstrates different acknowledgment behaviors
     */
    public static class AcknowledgmentDemo {

        public static void demonstrateAcknowledgmentModes() {
            System.out.println("=== Acknowledgment Mode Comparison ===");

            // Scenario: RF=3, Min ISR=2, Current ISR=2
            AckScenario scenario = new AckScenario(3, 2, 2);

            System.out.println("Scenario: RF=3, Min ISR=2, Current ISR=2");
            System.out.println();

            // Test different acks settings
            testAckMode(scenario, "0", "Fire and forget");
            testAckMode(scenario, "1", "Leader acknowledgment only");
            testAckMode(scenario, "all", "All in-sync replicas");

            System.out.println();

            // Scenario: RF=3, Min ISR=2, Current ISR=1 (under-replicated)
            AckScenario underReplicated = new AckScenario(3, 2, 1);

            System.out.println("Scenario: RF=3, Min ISR=2, Current ISR=1 (UNDER-
REPLICATED)");
            System.out.println();

            testAckMode(underReplicated, "0", "Fire and forget");
            testAckMode(underReplicated, "1", "Leader acknowledgment only");
            testAckMode(underReplicated, "all", "All in-sync replicas");
```

```
        }

        private static void testAckMode(AckScenario scenario, String acksMode,
String description) {
            AckResult result = scenario.testAcknowledgment(acksMode);

            System.out.printf("acks=%s (%s):%n", acksMode, description);
            System.out.printf("  Success: %s%n", result.success ? "✅ YES" : "❌
NO");
            System.out.printf("  Reason: %s%n", result.reason);
            System.out.printf("  Latency: %s%n", result.latencyDescription);
            System.out.println();
        }

        private static class AckScenario {
            final int replicationFactor;
            final int minISR;
            final int currentISRSize;

            AckScenario(int replicationFactor, int minISR, int currentISRSize) {
                this.replicationFactor = replicationFactor;
                this.minISR = minISR;
                this.currentISRSize = currentISRSize;
            }

            AckResult testAcknowledgment(String acksMode) {
                switch (acksMode) {
                    case "0":
                        return new AckResult(true, "No acknowledgment required",
"Immediate");

                    case "1":
                        if (currentISRSize >= 1) {
                            return new AckResult(true, "Leader acknowledged", "Low
(~1-5ms)");
                        } else {
                            return new AckResult(false, "No leader available",
"N/A");
                        }

                    case "all":
                        if (currentISRSize >= minISR) {
                            return new AckResult(true,
                                String.format("Acknowledged by %d replicas (min:
%d)", currentISRSize, minISR),
                                "Medium (~5-50ms)");
                        } else {
                            return new AckResult(false,
                                String.format("Insufficient replicas in ISR (%d <
%d)", currentISRSize, minISR),
                                "N/A - Request will fail");
                        }

                    default:
```

```
                        return new AckResult(false, "Invalid acks mode", "N/A");
                    }
                }
            }

            private static class AckResult {
                final boolean success;
                final String reason;
                final String latencyDescription;

                AckResult(boolean success, String reason, String latencyDescription) {
                    this.success = success;
                    this.reason = reason;
                    this.latencyDescription = latencyDescription;
                }
            }
        }

        /**
         * Monitor min.insync.replicas violations
         */
        public static class MinISRMonitor {

            private final AdminClient adminClient;
            private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(1);

            public MinISRMonitor(AdminClient adminClient) {
                this.adminClient = adminClient;
            }

            public void startMonitoring() {
                scheduler.scheduleAtFixedRate(this::checkMinISRViolations, 0, 30,
TimeUnit.SECONDS);
                System.out.println("Started min.insync.replicas monitoring");
            }

            private void checkMinISRViolations() {
                try {
                    System.out.println("\n=== Min ISR Violation Check ===");

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

                    // Describe topics to get partition info
                    Map<String, TopicDescription> topicDescriptions =
                        adminClient.describeTopics(topicNames).all().get();

                    // Get topic configurations for min.insync.replicas
                    List<ConfigResource> configResources = topicNames.stream()
                        .map(name -> new ConfigResource(ConfigResource.Type.TOPIC,
name))
                        .collect(Collectors.toList());
```

```
                Map<ConfigResource, Config> configs =
                    adminClient.describeConfigs(configResources).all().get();

                boolean foundViolations = false;

                for (Map.Entry<String, TopicDescription> entry :
topicDescriptions.entrySet()) {
                    String topicName = entry.getKey();
                    TopicDescription description = entry.getValue();

                    // Get min.insync.replicas for this topic
                    ConfigResource resource = new
ConfigResource(ConfigResource.Type.TOPIC, topicName);
                    Config config = configs.get(resource);

                    ConfigEntry minISREntry = config.get("min.insync.replicas");
                    int minISR = minISREntry != null ?
Integer.parseInt(minISREntry.value()) : 1;

                    // Check each partition
                    for (TopicPartitionInfo partition : description.partitions())
{
                        int currentISRSize = partition.isr().size();

                        if (currentISRSize < minISR) {
                            foundViolations = true;
                            System.out.printf("🚨 VIOLATION: %s-%d ISR=%d <
min.insync.replicas=%d%n",
                                topicName, partition.partition(), currentISRSize,
minISR);
                            System.out.printf("   Replicas: %s%n",
                                partition.replicas().stream()
                                    .map(node -> String.valueOf(node.id()))
                                    .collect(Collectors.joining(", ")));
                            System.out.printf("   ISR: %s%n",
                                partition.isr().stream()
                                    .map(node -> String.valueOf(node.id()))
                                    .collect(Collectors.joining(", ")));
                        }
                    }
                }

                if (!foundViolations) {
                    System.out.println("✅ No min.insync.replicas violations
found");
                }

            } catch (Exception e) {
                System.err.println("Error checking min ISR violations: " +
e.getMessage());
            }
        }

        public void stop() {
```

```
            scheduler.shutdown();
        }
    }

    public enum ReliabilityLevel {
        MAXIMUM_AVAILABILITY,
        BALANCED,
        MAXIMUM_CONSISTENCY,
        FINANCIAL_GRADE
    }
}
```

Producer Retries & Idempotency

Simple Explanation

Producer retries automatically resend failed messages, while idempotency ensures that retries don't create duplicate messages. Together, they provide reliable message delivery with exactly-once semantics.

Problem It Solves

- **Network failures**: Automatic retry on transient failures
- **Duplicate prevention**: Idempotency prevents duplicate messages
- **Ordering guarantees**: Maintains message order during retries
- **Operational simplicity**: Applications don't need custom retry logic

Implementation

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

/**
 * Comprehensive producer reliability configuration and monitoring
 */
public class ReliableProducerManager {

    /**
     * Producer configurations for different reliability requirements
     */
    public static Properties getReliableProducerConfig(ReliabilityProfile profile)
{
        Properties props = new Properties();

        // Basic connection settings
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
```

```
        switch (profile) {
            case HIGH_THROUGHPUT:
                // Optimize for throughput with some reliability trade-offs
                props.put(ProducerConfig.ACKS_CONFIG, "1");
                props.put(ProducerConfig.RETRIES_CONFIG, "5");
                props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "false");
                props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
"5");
                props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, "30000"); //
30 seconds
                break;

            case BALANCED:
                // Balance between performance and reliability
                props.put(ProducerConfig.ACKS_CONFIG, "all");
                props.put(ProducerConfig.RETRIES_CONFIG, "10");
                props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
                props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
"5");
                props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, "60000"); //
1 minute
                break;

            case MAXIMUM_RELIABILITY:
                // Maximum reliability with ordered delivery
                props.put(ProducerConfig.ACKS_CONFIG, "all");
                props.put(ProducerConfig.RETRIES_CONFIG,
String.valueOf(Integer.MAX_VALUE));
                props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
                props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
"1"); // Strict ordering
                props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, "120000"); //
2 minutes
                break;

            case FINANCIAL_GRADE:
                // Zero tolerance for data loss or duplication
                props.put(ProducerConfig.ACKS_CONFIG, "all");
                props.put(ProducerConfig.RETRIES_CONFIG,
String.valueOf(Integer.MAX_VALUE));
                props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
                props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
"1");
                props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, "300000"); //
5 minutes

                // Additional safety measures
                props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, "60000");
                props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, "1000");
                break;
        }

        // Common retry configuration
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, "100");
```

```
        props.put(ProducerConfig.RECONNECT_BACKOFF_MS_CONFIG, "50");
        props.put(ProducerConfig.RECONNECT_BACKOFF_MAX_MS_CONFIG, "1000");

        return props;
    }

    /**
     * Demonstrates idempotent producer behavior
     */
    public static class IdempotentProducerDemo {

        public static void demonstrateIdempotency() {
            System.out.println("=== Idempotent Producer Demo ===");

            // Create producer with idempotency enabled
            Properties props =
getReliableProducerConfig(ReliabilityProfile.BALANCED);

            try (KafkaProducer<String, String> producer = new KafkaProducer<>
(props)) {

                String topic = "idempotency-demo";
                String key = "test-key";
                String value = "test-message-" + System.currentTimeMillis();

                System.out.printf("Sending message: key=%s, value=%s%n", key,
value);

                // Send the same message multiple times to demonstrate idempotency
                for (int i = 0; i < 3; i++) {
                    ProducerRecord<String, String> record =
                        new ProducerRecord<>(topic, key, value);

                    Future<RecordMetadata> future = producer.send(record, new
Callback() {
                        @Override
                        public void onCompletion(RecordMetadata metadata,
Exception exception) {
                            if (exception != null) {
                                System.err.printf("Send failed: %s%n",
exception.getMessage());
                            } else {
                                System.out.printf("Sent to partition %d, offset
%d%n",
                                    metadata.partition(), metadata.offset());
                            }
                        }
                    });

                    // Wait for completion
                    try {
                        RecordMetadata metadata = future.get();
                        System.out.printf("Attempt %d: Sent to partition %d,
offset %d%n",
```

```
                            i + 1, metadata.partition(), metadata.offset());
                    } catch (Exception e) {
                        System.err.printf("Attempt %d failed: %s%n", i + 1,
e.getMessage());
                    }
                }

                System.out.println("Note: With idempotency enabled, only one
message should be stored");

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

    /**
     * Reliable producer with comprehensive error handling
     */
    public static class ReliableProducer {

        private final KafkaProducer<String, String> producer;
        private final AtomicLong sentMessages = new AtomicLong(0);
        private final AtomicLong failedMessages = new AtomicLong(0);
        private final AtomicLong retriedMessages = new AtomicLong(0);

        public ReliableProducer(ReliabilityProfile profile) {
            Properties props = getReliableProducerConfig(profile);
            this.producer = new KafkaProducer<>(props);

            System.out.printf("Created reliable producer with profile: %s%n",
profile);
        }

        public void sendReliably(String topic, String key, String value) {
            ProducerRecord<String, String> record = new ProducerRecord<>(topic,
key, value);

            producer.send(record, new ReliableCallback(key, value));
            sentMessages.incrementAndGet();
        }

        private class ReliableCallback implements Callback {
            private final String key;
            private final String value;
            private int attemptNumber = 1;

            ReliableCallback(String key, String value) {
                this.key = key;
                this.value = value;
            }

            @Override
            public void onCompletion(RecordMetadata metadata, Exception exception)
```

```
{
                if (exception == null) {
                    // Success
                    System.out.printf("✅ Sent: key=%s, partition=%d, offset=%d,
attempt=%d%n",
                        key, metadata.partition(), metadata.offset(),
attemptNumber);
                } else {
                    // Failure
                    failedMessages.incrementAndGet();

                    if (isRetriableException(exception)) {
                        retriedMessages.incrementAndGet();
                        System.out.printf("🔄 Retrying: key=%s, attempt=%d,
error=%s%n",
                            key, attemptNumber, exception.getMessage());
                        attemptNumber++;
                    } else {
                        System.err.printf("❌ Failed permanently: key=%s,
error=%s%n",
                            key, exception.getMessage());
                    }
                }
            }

            private boolean isRetriableException(Exception exception) {
                // Kafka producer will automatically retry retriable exceptions
                // This is just for logging purposes
                return exception instanceof
org.apache.kafka.common.errors.RetriableException ||
                       exception instanceof
org.apache.kafka.common.errors.TimeoutException ||
                       exception instanceof
org.apache.kafka.common.errors.NotLeaderOrFollowerException;
            }
        }

        public void printStatistics() {
            System.out.println("\n=== Producer Statistics ===");
            System.out.printf("Total messages sent: %d%n", sentMessages.get());
            System.out.printf("Failed messages: %d%n", failedMessages.get());
            System.out.printf("Retried messages: %d%n", retriedMessages.get());
            System.out.printf("Success rate: %.2f%%%n",
                (1.0 - (double) failedMessages.get() / sentMessages.get()) * 100);
        }

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

    /**
     * Test different failure scenarios
     */
```

```
    public static class FailureScenarioTester {

        public static void testRetryBehavior() {
            System.out.println("\n=== Retry Behavior Test ===");

            // Test with different configurations
            testConfiguration("High Throughput",
ReliabilityProfile.HIGH_THROUGHPUT);
            testConfiguration("Balanced", ReliabilityProfile.BALANCED);
            testConfiguration("Maximum Reliability",
ReliabilityProfile.MAXIMUM_RELIABILITY);
        }

        private static void testConfiguration(String name, ReliabilityProfile
profile) {
            System.out.printf("\nTesting %s configuration:%n", name);

            Properties props = getReliableProducerConfig(profile);

            System.out.printf("  acks: %s%n",
props.getProperty(ProducerConfig.ACKS_CONFIG));
            System.out.printf("  retries: %s%n",
props.getProperty(ProducerConfig.RETRIES_CONFIG));
            System.out.printf("  idempotence: %s%n",
props.getProperty(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG));
            System.out.printf("  max.in.flight: %s%n",
props.getProperty(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION));
            System.out.printf("  delivery.timeout: %s ms%n",
props.getProperty(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG));

            // Analyze trade-offs
            analyzeBehavior(props);
        }

        private static void analyzeBehavior(Properties props) {
            String acks = props.getProperty(ProducerConfig.ACKS_CONFIG);
            boolean idempotent =
Boolean.parseBoolean(props.getProperty(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG));
            int maxInFlight =
Integer.parseInt(props.getProperty(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNE
CTION));

            System.out.println("  Behavior analysis:");

            // Durability
            if ("all".equals(acks)) {
                System.out.println("    ☑ High durability (waits for ISR
acknowledgment)");
            } else if ("1".equals(acks)) {
                System.out.println("    ⚠  Medium durability (leader only)");
            } else {
                System.out.println("    ✗ Low durability (no acknowledgment)");
            }
```

```
            // Duplication protection
            if (idempotent) {
                System.out.println("    ☑ Protected against duplicates");
            } else {
                System.out.println("    ⚠ May create duplicates on retry");
            }

            // Ordering
            if (maxInFlight == 1) {
                System.out.println("    ☑ Strict ordering guaranteed");
            } else if (idempotent && maxInFlight <= 5) {
                System.out.println("    ☑ Ordering guaranteed with
idempotency");
            } else {
                System.out.println("    ⚠ Ordering may be affected");
            }
        }
    }

    public enum ReliabilityProfile {
        HIGH_THROUGHPUT,
        BALANCED,
        MAXIMUM_RELIABILITY,
        FINANCIAL_GRADE
    }

    // Example usage
    public static void main(String[] args) {
        // Demonstrate idempotency
        IdempotentProducerDemo.demonstrateIdempotency();

        // Test retry behavior
        FailureScenarioTester.testRetryBehavior();

        // Create reliable producer
        ReliableProducer producer = new
ReliableProducer(ReliabilityProfile.BALANCED);

        // Send some messages
        for (int i = 0; i < 10; i++) {
            producer.sendReliably("reliable-topic", "key-" + i, "message-" + i);
        }

        // Print statistics
        producer.printStatistics();
        producer.close();
    }
}
```

⚡ Performance

Simple Explanation

Kafka performance optimization involves tuning producers, consumers, and the underlying system to achieve maximum throughput and minimum latency. This includes batching strategies, compression, and leveraging the operating system's page cache.

Problem It Solves

- **Throughput bottlenecks**: Maximize messages processed per second
- **Latency issues**: Minimize end-to-end message delivery time
- **Resource utilization**: Efficiently use CPU, memory, and network
- **Cost optimization**: Better performance means lower infrastructure costs

Performance Architecture

```
Kafka Performance Stack:

Application Layer:
┌─────────────────────────────────────────────────────────┐
│ Producer Batching (batch.size, linger.ms)               │
│ Consumer Batching (fetch.min.bytes, max.poll.records)   │
│ Compression (lz4, zstd, snappy)                         │
└─────────────────────────────────────────────────────────┘
                              ↓
Network Layer:
┌─────────────────────────────────────────────────────────┐
│ Socket Buffers (send/receive buffer sizes)              │
│ Connection Pooling (max.in.flight.requests)             │
│ Protocol Optimization (binary protocol, zero-copy)      │
└─────────────────────────────────────────────────────────┘
                              ↓
Storage Layer:
┌─────────────────────────────────────────────────────────┐
│ Page Cache (OS-level caching)                           │
│ Sequential I/O (append-only logs)                       │
│ Zero-Copy Transfer (sendfile system call)               │
└─────────────────────────────────────────────────────────┘
```

Producer Performance Tuning

Batching and Linger Configuration

```java
/**
 * Comprehensive producer performance tuning
 */
public class ProducerPerformanceTuning {

    /**
     * Producer configurations optimized for different performance goals
```

```
     */
    public static Properties getPerformanceConfig(PerformanceGoal goal) {
        Properties props = new Properties();

        // Base configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

        switch (goal) {
            case ULTRA_LOW_LATENCY:
                // Optimize for latency (< 1ms)
                props.put(ProducerConfig.BATCH_SIZE_CONFIG, "0"); // Disable
batching
                props.put(ProducerConfig.LINGER_MS_CONFIG, "0");
                props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "none");
                props.put(ProducerConfig.ACKS_CONFIG, "1"); // Leader only
                props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, "16777216"); //
16MB
                break;

            case HIGH_THROUGHPUT:
                // Optimize for throughput (>1M messages/sec)
                props.put(ProducerConfig.BATCH_SIZE_CONFIG, "131072"); // 128KB
                props.put(ProducerConfig.LINGER_MS_CONFIG, "20");
                props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");
                props.put(ProducerConfig.ACKS_CONFIG, "1");
                props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, "134217728"); //
128MB
                props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,
"5");
                break;

            case BALANCED:
                // Balance latency and throughput
                props.put(ProducerConfig.BATCH_SIZE_CONFIG, "32768"); // 32KB
                props.put(ProducerConfig.LINGER_MS_CONFIG, "5");
                props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");
                props.put(ProducerConfig.ACKS_CONFIG, "all");
                props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, "67108864"); //
64MB
                break;

            case MAXIMUM_COMPRESSION:
                // Optimize for network and storage efficiency
                props.put(ProducerConfig.BATCH_SIZE_CONFIG, "262144"); // 256KB
                props.put(ProducerConfig.LINGER_MS_CONFIG, "100");
                props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "zstd");
                props.put(ProducerConfig.ACKS_CONFIG, "all");
                props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, "268435456"); //
256MB
                break;
```

```
        }

        // Common optimizations
        props.put(ProducerConfig.SEND_BUFFER_CONFIG, "131072"); // 128KB
        props.put(ProducerConfig.RECEIVE_BUFFER_CONFIG, "65536"); // 64KB
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, "30000");

        return props;
    }

    /**
     * Compression analysis and comparison
     */
    public static class CompressionAnalyzer {

        public static void analyzeCompressionTypes() {
            System.out.println("=== Compression Type Analysis ===");

            String[] compressionTypes = {"none", "gzip", "snappy", "lz4", "zstd"};
            String sampleData = generateSampleData(1000); // 1KB sample

            for (String compressionType : compressionTypes) {
                analyzeCompression(compressionType, sampleData);
            }
        }

        private static void analyzeCompression(String compressionType, String
data) {
            // Simulate compression metrics (in real implementation, use actual
compression)
            CompressionResult result = simulateCompression(compressionType, data);

            System.out.printf("\n%s Compression:%n",
compressionType.toUpperCase());
            System.out.printf("  Original size: %d bytes%n", data.length());
            System.out.printf("  Compressed size: %d bytes%n",
result.compressedSize);
            System.out.printf("  Compression ratio: %.2f%n",
result.compressionRatio);
            System.out.printf("  CPU overhead: %s%n", result.cpuOverhead);
            System.out.printf("  Compress time: %s%n", result.compressTime);
            System.out.printf("  Decompress time: %s%n", result.decompressTime);
            System.out.printf("  Best for: %s%n", result.bestFor);
        }

        private static CompressionResult simulateCompression(String type, String
data) {
            int originalSize = data.length();

            switch (type.toLowerCase()) {
                case "none":
                    return new CompressionResult(originalSize, 1.0, "None", "0ms",
"0ms",
                        "Ultra-low latency scenarios");
```

```
                case "gzip":
                    return new CompressionResult((int)(originalSize * 0.3), 0.3,
"High", "10ms", "3ms",
                            "Maximum compression, slow networks");

                case "snappy":
                    return new CompressionResult((int)(originalSize * 0.5), 0.5,
"Low", "2ms", "1ms",
                            "Balanced performance, general use");

                case "lz4":
                    return new CompressionResult((int)(originalSize * 0.6), 0.6,
"Very Low", "1ms", "0.5ms",
                            "High throughput, low CPU overhead");

                case "zstd":
                    return new CompressionResult((int)(originalSize * 0.35), 0.35,
"Medium", "5ms", "2ms",
                            "Best compression ratio with good speed");

                default:
                    return new CompressionResult(originalSize, 1.0, "Unknown",
"N/A", "N/A", "Unknown");
            }
        }

        private static String generateSampleData(int size) {
            StringBuilder sb = new StringBuilder();
            String pattern = "Lorem ipsum dolor sit amet, consectetur adipiscing
elit. ";

            while (sb.length() < size) {
                sb.append(pattern);
            }

            return sb.substring(0, size);
        }

        private static class CompressionResult {
            final int compressedSize;
            final double compressionRatio;
            final String cpuOverhead;
            final String compressTime;
            final String decompressTime;
            final String bestFor;

            CompressionResult(int compressedSize, double compressionRatio, String
cpuOverhead,
                              String compressTime, String decompressTime, String
bestFor) {
                this.compressedSize = compressedSize;
                this.compressionRatio = compressionRatio;
                this.cpuOverhead = cpuOverhead;
```

```
                this.compressTime = compressTime;
                this.decompressTime = decompressTime;
                this.bestFor = bestFor;
            }
        }
    }

    /**
     * Batching optimization demonstration
     */
    public static class BatchingOptimizer {

        public static void demonstrateBatchingEffects() {
            System.out.println("\n=== Batching Effects Demonstration ===");

            // Test different batch configurations
            testBatchConfiguration("No Batching", 0, 0);
            testBatchConfiguration("Small Batches", 1024, 1); // 1KB, 1ms
            testBatchConfiguration("Medium Batches", 16384, 5); // 16KB, 5ms
            testBatchConfiguration("Large Batches", 131072, 20); // 128KB, 20ms
            testBatchConfiguration("Huge Batches", 1048576, 100); // 1MB, 100ms
        }

        private static void testBatchConfiguration(String name, int batchSize, int
lingerMs) {
            System.out.printf("\n%s (batch.size=%d, linger.ms=%d):%n", name,
batchSize, lingerMs);

            // Simulate performance characteristics
            BatchingMetrics metrics = simulateBatching(batchSize, lingerMs);

            System.out.printf("  Estimated throughput: %,d messages/sec%n",
metrics.throughput);
            System.out.printf("  Estimated latency: %d ms%n", metrics.latency);
            System.out.printf("  Network efficiency: %.1f%%%n",
metrics.networkEfficiency);
            System.out.printf("  CPU utilization: %s%n", metrics.cpuUtilization);
            System.out.printf("  Memory usage: %s%n", metrics.memoryUsage);

            // Trade-off analysis
            if (metrics.latency < 5) {
                System.out.println("  ✅ Good for low-latency applications");
            }
            if (metrics.throughput > 100000) {
                System.out.println("  ✅ Good for high-throughput applications");
            }
            if (metrics.networkEfficiency > 80) {
                System.out.println("  ✅ Efficient network utilization");
            }
        }

        private static BatchingMetrics simulateBatching(int batchSize, int
lingerMs) {
            // Simplified simulation - real values depend on many factors
```

```
            int throughput;
            int latency;
            double networkEfficiency;
            String cpuUtilization;
            String memoryUsage;

            if (batchSize == 0) {
                // No batching
                throughput = 10000;
                latency = 1;
                networkEfficiency = 20.0;
                cpuUtilization = "High";
                memoryUsage = "Low";
            } else if (batchSize <= 1024) {
                // Small batches
                throughput = 50000;
                latency = Math.max(2, lingerMs);
                networkEfficiency = 40.0;
                cpuUtilization = "Medium-High";
                memoryUsage = "Low";
            } else if (batchSize <= 16384) {
                // Medium batches
                throughput = 200000;
                latency = Math.max(3, lingerMs);
                networkEfficiency = 70.0;
                cpuUtilization = "Medium";
                memoryUsage = "Medium";
            } else if (batchSize <= 131072) {
                // Large batches
                throughput = 500000;
                latency = Math.max(5, lingerMs);
                networkEfficiency = 85.0;
                cpuUtilization = "Low-Medium";
                memoryUsage = "Medium";
            } else {
                // Huge batches
                throughput = 800000;
                latency = Math.max(10, lingerMs);
                networkEfficiency = 95.0;
                cpuUtilization = "Low";
                memoryUsage = "High";
            }

            return new BatchingMetrics(throughput, latency, networkEfficiency,
                                       cpuUtilization, memoryUsage);
        }

        private static class BatchingMetrics {
            final int throughput;
            final int latency;
            final double networkEfficiency;
            final String cpuUtilization;
            final String memoryUsage;
```

```
            BatchingMetrics(int throughput, int latency, double networkEfficiency,
                            String cpuUtilization, String memoryUsage) {
                this.throughput = throughput;
                this.latency = latency;
                this.networkEfficiency = networkEfficiency;
                this.cpuUtilization = cpuUtilization;
                this.memoryUsage = memoryUsage;
            }
        }
    }

    /**
     * Performance testing framework
     */
    public static class PerformanceTester {

        public static void runPerformanceTest(PerformanceGoal goal) {
            System.out.printf("\n=== Performance Test: %s ===\n", goal);

            Properties config = getPerformanceConfig(goal);

            System.out.println("Configuration:");
            config.forEach((key, value) -> {
                if (key.toString().contains("batch") ||
key.toString().contains("linger") ||
                    key.toString().contains("compression") ||
key.toString().contains("buffer")) {
                    System.out.printf("  %s: %s%n", key, value);
                }
            });

            // Simulate performance test results
            PerformanceResult result = simulatePerformanceTest(goal);

            System.out.println("\nResults:");
            System.out.printf("  Throughput: %,d messages/sec%n",
result.throughput);
            System.out.printf("  Latency p50: %d ms%n", result.latencyP50);
            System.out.printf("  Latency p99: %d ms%n", result.latencyP99);
            System.out.printf("  CPU usage: %.1f%%%n", result.cpuUsage);
            System.out.printf("  Memory usage: %d MB%n", result.memoryUsage);
            System.out.printf("  Network throughput: %.1f MB/s%n",
result.networkThroughput);
        }

        private static PerformanceResult simulatePerformanceTest(PerformanceGoal
goal) {
            switch (goal) {
                case ULTRA_LOW_LATENCY:
                    return new PerformanceResult(50000, 1, 3, 45.0, 128, 25.0);
                case HIGH_THROUGHPUT:
                    return new PerformanceResult(1000000, 15, 50, 80.0, 512,
200.0);
```

```
                case BALANCED:
                    return new PerformanceResult(300000, 5, 20, 60.0, 256, 75.0);
                case MAXIMUM_COMPRESSION:
                    return new PerformanceResult(200000, 25, 100, 90.0, 1024,
50.0);
                default:
                    return new PerformanceResult(0, 0, 0, 0.0, 0, 0.0);
            }
        }

        private static class PerformanceResult {
            final int throughput;
            final int latencyP50;
            final int latencyP99;
            final double cpuUsage;
            final int memoryUsage;
            final double networkThroughput;

            PerformanceResult(int throughput, int latencyP50, int latencyP99,
                              double cpuUsage, int memoryUsage, double
networkThroughput) {
                this.throughput = throughput;
                this.latencyP50 = latencyP50;
                this.latencyP99 = latencyP99;
                this.cpuUsage = cpuUsage;
                this.memoryUsage = memoryUsage;
                this.networkThroughput = networkThroughput;
            }
        }
    }

    public enum PerformanceGoal {
        ULTRA_LOW_LATENCY,
        HIGH_THROUGHPUT,
        BALANCED,
        MAXIMUM_COMPRESSION
    }
}
```

Consumer Performance Tuning

Fetch Size and Prefetch Optimization

```
/**
 * Consumer performance tuning and optimization
 */
public class ConsumerPerformanceTuning {

    /**
     * Consumer configurations optimized for different workloads
     */
```

```
    public static Properties getConsumerConfig(ConsumerWorkload workload) {
        Properties props = new Properties();

        // Base configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "perf-test-consumer");

        switch (workload) {
            case LOW_LATENCY_STREAMING:
                // Optimize for minimal processing delay
                props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "1");
                props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "1");
                props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "100");
                props.put(ConsumerConfig.RECEIVE_BUFFER_CONFIG, "32768"); // 32KB
                break;

            case HIGH_THROUGHPUT_BATCH:
                // Optimize for maximum throughput
                props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "1048576"); //
1MB
                props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "500");
                props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "10000");
                props.put(ConsumerConfig.RECEIVE_BUFFER_CONFIG, "131072"); //
128KB
                break;

            case BALANCED_PROCESSING:
                // Balance latency and throughput
                props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "65536"); // 64KB
                props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "100");
                props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "1000");
                props.put(ConsumerConfig.RECEIVE_BUFFER_CONFIG, "65536"); // 64KB
                break;

            case MEMORY_CONSTRAINED:
                // Optimize for low memory usage
                props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "16384"); // 16KB
                props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "100");
                props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "200");
                props.put(ConsumerConfig.RECEIVE_BUFFER_CONFIG, "16384"); // 16KB
                break;
        }

        // Common optimizations
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false"); // Manual
commit for control
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");
        props.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");

        return props;
```

```
    }

    /**
     * Demonstrates fetch size optimization effects
     */
    public static class FetchSizeOptimizer {

        public static void demonstrateFetchSizes() {
            System.out.println("=== Fetch Size Optimization ===");

            int[] fetchSizes = {1, 1024, 16384, 65536, 262144, 1048576}; // 1B to
1MB
            int[] waitTimes = {1, 10, 50, 100, 500}; // 1ms to 500ms

            for (int fetchSize : fetchSizes) {
                for (int waitTime : waitTimes) {
                    analyzeFetchConfiguration(fetchSize, waitTime);
                }
            }
        }

        private static void analyzeFetchConfiguration(int fetchMinBytes, int
fetchMaxWait) {
            FetchMetrics metrics = simulateFetchBehavior(fetchMinBytes,
fetchMaxWait);

            System.out.printf("\nfetch.min.bytes=%d, fetch.max.wait.ms=%d:%n",
                fetchMinBytes, fetchMaxWait);
            System.out.printf("  Throughput: %,d messages/sec%n",
metrics.throughput);
            System.out.printf("  Latency: %d ms%n", metrics.latency);
            System.out.printf("  Network efficiency: %.1f%%%n",
metrics.networkEfficiency);
            System.out.printf("  Memory usage: %s%n", metrics.memoryUsage);

            // Recommendations
            if (metrics.latency <= 5 && metrics.throughput >= 50000) {
                System.out.println("  ✅ Excellent for real-time processing");
            } else if (metrics.throughput >= 200000 && metrics.networkEfficiency
>= 80) {
                System.out.println("  ✅ Excellent for batch processing");
            } else if (metrics.latency <= 10) {
                System.out.println("  ✅ Good for low-latency applications");
            }
        }

        private static FetchMetrics simulateFetchBehavior(int fetchMinBytes, int
fetchMaxWait) {
            // Simplified simulation based on configuration

            int throughput;
            int latency;
            double networkEfficiency;
            String memoryUsage;
```

```
            if (fetchMinBytes <= 1024) {
                // Small fetches - low latency, high overhead
                throughput = 30000 - (fetchMaxWait * 100);
                latency = Math.max(1, fetchMaxWait / 10);
                networkEfficiency = 30.0;
                memoryUsage = "Low";
            } else if (fetchMinBytes <= 65536) {
                // Medium fetches - balanced
                throughput = 150000 + (fetchMaxWait * 200);
                latency = Math.max(5, fetchMaxWait / 5);
                networkEfficiency = 60.0;
                memoryUsage = "Medium";
            } else {
                // Large fetches - high throughput, higher latency
                throughput = 300000 + (fetchMaxWait * 500);
                latency = Math.max(10, fetchMaxWait);
                networkEfficiency = 85.0;
                memoryUsage = "High";
            }

            return new FetchMetrics(Math.max(0, throughput), latency,
networkEfficiency, memoryUsage);
        }

        private static class FetchMetrics {
            final int throughput;
            final int latency;
            final double networkEfficiency;
            final String memoryUsage;

            FetchMetrics(int throughput, int latency, double networkEfficiency,
String memoryUsage) {
                this.throughput = throughput;
                this.latency = latency;
                this.networkEfficiency = networkEfficiency;
                this.memoryUsage = memoryUsage;
            }
        }
    }

    /**
     * High-performance consumer implementation
     */
    public static class HighPerformanceConsumer {

        private final KafkaConsumer<String, String> consumer;
        private final AtomicLong messagesProcessed = new AtomicLong(0);
        private final AtomicLong totalBytesProcessed = new AtomicLong(0);
        private final long startTime = System.currentTimeMillis();
        private volatile boolean running = true;

        public HighPerformanceConsumer(ConsumerWorkload workload) {
            Properties props = getConsumerConfig(workload);
```

```
            this.consumer = new KafkaConsumer<>(props);

            System.out.printf("Created high-performance consumer for workload:
%s%n", workload);
        }

        public void consume(String... topics) {
            consumer.subscribe(Arrays.asList(topics));

            System.out.println("Starting high-performance consumption...");

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

                if (!records.isEmpty()) {
                    processRecords(records);
                    commitOffsets();
                }
            }
        }

        private void processRecords(ConsumerRecords<String, String> records) {
            // Process records in batches for efficiency
            List<ProcessingTask> tasks = new ArrayList<>();

            for (ConsumerRecord<String, String> record : records) {
                tasks.add(new ProcessingTask(record));
                totalBytesProcessed.addAndGet(record.serializedValueSize());
            }

            // Batch processing
            processTasksBatch(tasks);
            messagesProcessed.addAndGet(records.count());
        }

        private void processTasksBatch(List<ProcessingTask> tasks) {
            // Simulate efficient batch processing
            // In real implementation, this might involve:
            // - Database batch inserts
            // - Parallel processing
            // - Efficient serialization/deserialization

            for (ProcessingTask task : tasks) {
                // Minimal processing simulation
                task.process();
            }
        }

        private void commitOffsets() {
            try {
                consumer.commitSync();
            } catch (Exception e) {
                System.err.println("Failed to commit offsets: " + e.getMessage());
```

```
            }
        }

        public void printStatistics() {
            long runtime = System.currentTimeMillis() - startTime;
            long messagesPerSecond = messagesProcessed.get() * 1000 / Math.max(1,
runtime);
            long bytesPerSecond = totalBytesProcessed.get() * 1000 / Math.max(1,
runtime);

            System.out.println("\n=== Consumer Performance Statistics ===");
            System.out.printf("Runtime: %.2f seconds%n", runtime / 1000.0);
            System.out.printf("Messages processed: %,d%n",
messagesProcessed.get());
            System.out.printf("Bytes processed: %,d%n",
totalBytesProcessed.get());
            System.out.printf("Messages/second: %,d%n", messagesPerSecond);
            System.out.printf("MB/second: %.2f%n", bytesPerSecond / (1024.0 *
1024.0));
        }

        public void stop() {
            running = false;
            consumer.close();
        }

        private static class ProcessingTask {
            private final ConsumerRecord<String, String> record;

            ProcessingTask(ConsumerRecord<String, String> record) {
                this.record = record;
            }

            void process() {
                // Simulate processing work
                // In real implementation: business logic, transformations, etc.
            }
        }
    }

    public enum ConsumerWorkload {
        LOW_LATENCY_STREAMING,
        HIGH_THROUGHPUT_BATCH,
        BALANCED_PROCESSING,
        MEMORY_CONSTRAINED
    }
}
```

Page Cache & OS Tuning

Operating System Optimization

```
/**
 * Page cache and OS tuning recommendations and monitoring
 */
public class PageCacheOptimizer {

    /**
     * OS configuration recommendations for Kafka
     */
    public static class OSConfiguration {

        public static void printOptimalSettings() {
            System.out.println("=== Optimal OS Settings for Kafka ===");

            System.out.println("\n1. Virtual Memory Settings
(/etc/sysctl.conf):");
            System.out.println("   vm.swappiness=1                    # Minimize
swapping");
            System.out.println("   vm.dirty_ratio=80                  # Delay
writes for better batching");
            System.out.println("   vm.dirty_background_ratio=5        # Background
write threshold");
            System.out.println("   vm.max_map_count=262144           # Memory map
areas for segments");

            System.out.println("\n2. Network Settings:");
            System.out.println("   net.core.rmem_default=262144      # Default
receive buffer");
            System.out.println("   net.core.rmem_max=16777216        # Max receive
buffer");
            System.out.println("   net.core.wmem_default=262144      # Default
send buffer");
            System.out.println("   net.core.wmem_max=16777216        # Max send
buffer");
            System.out.println("   net.core.netdev_max_backlog=5000  # Network
device backlog");

            System.out.println("\n3. File System Settings:");
            System.out.println("   - Use XFS or ext4 file systems");
            System.out.println("   - Mount with 'noatime' option");
            System.out.println("   - Consider separate disks for logs");

            System.out.println("\n4. CPU Settings:");
            System.out.println("   - Set CPU governor to 'performance'");
            System.out.println("   - Disable CPU frequency scaling");
            System.out.println("   - Consider CPU affinity for Kafka processes");
        }

        public static Map<String, String> getRecommendedSysctlSettings() {
            Map<String, String> settings = new HashMap<>();

            // Virtual memory settings
            settings.put("vm.swappiness", "1");
            settings.put("vm.dirty_ratio", "80");
```

```
            settings.put("vm.dirty_background_ratio", "5");
            settings.put("vm.max_map_count", "262144");
            settings.put("vm.overcommit_memory", "1");

            // Network settings
            settings.put("net.core.rmem_default", "262144");
            settings.put("net.core.rmem_max", "16777216");
            settings.put("net.core.wmem_default", "262144");
            settings.put("net.core.wmem_max", "16777216");
            settings.put("net.core.netdev_max_backlog", "5000");

            // TCP settings
            settings.put("net.ipv4.tcp_window_scaling", "1");
            settings.put("net.ipv4.tcp_rmem", "4096 65536 16777216");
            settings.put("net.ipv4.tcp_wmem", "4096 65536 16777216");

            return settings;
        }
    }

    /**
     * Page cache monitoring and analysis
     */
    public static class PageCacheMonitor {

        public static void monitorPageCacheUsage() {
            System.out.println("=== Page Cache Monitoring ===");

            try {
                // Read memory information from /proc/meminfo
                PageCacheStats stats = readPageCacheStats();

                System.out.printf("Total Memory: %s%n",
formatBytes(stats.totalMemory));
                System.out.printf("Free Memory: %s%n",
formatBytes(stats.freeMemory));
                System.out.printf("Page Cache: %s (%.1f%% of total)%n",
                    formatBytes(stats.pageCache),
                    (double) stats.pageCache / stats.totalMemory * 100);
                System.out.printf("Dirty Pages: %s%n",
formatBytes(stats.dirtyPages));
                System.out.printf("Writeback Pages: %s%n",
formatBytes(stats.writebackPages));

                // Analysis
                analyzePageCacheHealth(stats);

            } catch (Exception e) {
                System.err.println("Failed to read page cache stats: " +
e.getMessage());
                // Provide simulated data for demonstration
                demonstratePageCacheAnalysis();
            }
        }
```

```
        private static PageCacheStats readPageCacheStats() throws IOException {
            // In a real implementation, read from /proc/meminfo
            // For demonstration, return simulated values
            return new PageCacheStats(
                32L * 1024 * 1024 * 1024,  // 32GB total
                2L * 1024 * 1024 * 1024,   // 2GB free
                26L * 1024 * 1024 * 1024,  // 26GB page cache
                512L * 1024 * 1024,        // 512MB dirty
                64L * 1024 * 1024          // 64MB writeback
            );
        }

        private static void analyzePageCacheHealth(PageCacheStats stats) {
            System.out.println("\n=== Page Cache Health Analysis ===");

            double cacheRatio = (double) stats.pageCache / stats.totalMemory;
            double dirtyRatio = (double) stats.dirtyPages / stats.pageCache;

            if (cacheRatio > 0.7) {
                System.out.println("✅ Excellent page cache utilization (>70%)");
            } else if (cacheRatio > 0.5) {
                System.out.println("✅ Good page cache utilization (>50%)");
            } else {
                System.out.println("⚠️  Low page cache utilization (<50%)");
                System.out.println("   Consider reducing JVM heap size");
            }

            if (dirtyRatio < 0.1) {
                System.out.println("✅ Healthy dirty page ratio (<10%)");
            } else if (dirtyRatio < 0.2) {
                System.out.println("⚠️  Moderate dirty page ratio (10-20%)");
            } else {
                System.out.println("❌ High dirty page ratio (>20%)");
                System.out.println("   May indicate I/O bottleneck");
            }

            // Kafka-specific recommendations
            System.out.println("\n=== Kafka-Specific Recommendations ===");
            long recommendedHeap = stats.totalMemory / 4; // 25% of total memory
            System.out.printf("Recommended JVM heap: %s%n",
formatBytes(recommendedHeap));
            System.out.printf("Available for page cache: %s%n",
                formatBytes(stats.totalMemory - recommendedHeap));
        }

        private static void demonstratePageCacheAnalysis() {
            System.out.println("=== Page Cache Analysis (Simulated) ===");

            // Demonstrate different scenarios
            analyzeScenario("Well-tuned system", 32L * 1024 * 1024 * 1024, 26L *
1024 * 1024 * 1024);
            analyzeScenario("Over-allocated JVM", 32L * 1024 * 1024 * 1024, 8L *
1024 * 1024 * 1024);
```

```
            analyzeScenario("Memory-constrained", 8L * 1024 * 1024 * 1024, 4L *
1024 * 1024 * 1024);
        }

        private static void analyzeScenario(String scenario, long totalMemory,
long pageCache) {
            System.out.printf("\n%s:%n", scenario);
            System.out.printf("  Total memory: %s%n", formatBytes(totalMemory));
            System.out.printf("  Page cache: %s (%.1f%%)%n",
                formatBytes(pageCache), (double) pageCache / totalMemory * 100);

            if (pageCache > totalMemory * 0.7) {
                System.out.println("  ☑ Optimal for Kafka performance");
            } else if (pageCache > totalMemory * 0.5) {
                System.out.println("  ⚠  Adequate but could be improved");
            } else {
                System.out.println("  ✗ Suboptimal - reduce JVM heap size");
            }
        }

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

        private static class PageCacheStats {
            final long totalMemory;
            final long freeMemory;
            final long pageCache;
            final long dirtyPages;
            final long writebackPages;

            PageCacheStats(long totalMemory, long freeMemory, long pageCache,
                           long dirtyPages, long writebackPages) {
                this.totalMemory = totalMemory;
                this.freeMemory = freeMemory;
                this.pageCache = pageCache;
                this.dirtyPages = dirtyPages;
                this.writebackPages = writebackPages;
            }
        }
    }

    /**
     * File system optimization recommendations
     */
    public static class FileSystemOptimizer {

        public static void printFileSystemRecommendations() {
            System.out.println("=== File System Optimization ===");
```

```
            System.out.println("\n1. File System Choice:");
            System.out.println("   ☑ XFS (recommended) - excellent for large
files and sequential I/O");
            System.out.println("   ☑ ext4 - good general purpose, widely
supported");
            System.out.println("   ✗ ext3 - poor performance for Kafka
workloads");

            System.out.println("\n2. Mount Options:");
            System.out.println("   noatime,nodiratime    # Disable access time
updates");
            System.out.println("   largeio               # Optimize for large I/O
operations");
            System.out.println("   inode64               # Use 64-bit inodes
(XFS)");

            System.out.println("\n3. Block Size:");
            System.out.println("   XFS: 4KB block size (default)");
            System.out.println("   ext4: 4KB block size");

            System.out.println("\n4. Storage Layout:");
            System.out.println("   - Separate disks for OS and Kafka logs");
            System.out.println("   - Use RAID 10 for best performance and
redundancy");
            System.out.println("   - Consider NVMe SSDs for low-latency
workloads");

            System.out.println("\n5. Directory Structure:");
            System.out.println("   /kafka/logs/broker-0/");
            System.out.println("   /kafka/logs/broker-1/  # Multiple log
directories");
            System.out.println("   /kafka/logs/broker-2/  # on separate disks");
        }

        public static Map<String, String> getOptimalMountOptions() {
            Map<String, String> options = new HashMap<>();

            options.put("XFS", "rw,noatime,nodiratime,largeio,inode64");
            options.put("ext4", "rw,noatime,nodiratime,data=writeback");

            return options;
        }
    }

    /**
     * I/O monitoring and optimization
     */
    public static class IOOptimizer {

        public static void monitorIOPerformance() {
            System.out.println("=== I/O Performance Monitoring ===");

            // Simulate I/O statistics
```

```
            IOStats stats = getIOStats();

            System.out.printf("Read IOPS: %,d%n", stats.readIOPS);
            System.out.printf("Write IOPS: %,d%n", stats.writeIOPS);
            System.out.printf("Read throughput: %.1f MB/s%n",
stats.readThroughput);
            System.out.printf("Write throughput: %.1f MB/s%n",
stats.writeThroughput);
            System.out.printf("Average latency: %.2f ms%n", stats.avgLatency);
            System.out.printf("Queue depth: %d%n", stats.queueDepth);

            analyzeIOPerformance(stats);
        }

        private static IOStats getIOStats() {
            // In real implementation, read from /proc/diskstats or use iostat
            return new IOStats(5000, 15000, 150.0, 450.0, 2.5, 8);
        }

        private static void analyzeIOPerformance(IOStats stats) {
            System.out.println("\n=== I/O Performance Analysis ===");

            // Check for optimal Kafka I/O patterns
            if (stats.writeIOPS > stats.readIOPS * 2) {
                System.out.println("✅ Good write-heavy pattern (typical for
Kafka)");
            }

            if (stats.avgLatency < 5.0) {
                System.out.println("✅ Excellent I/O latency (<5ms)");
            } else if (stats.avgLatency < 10.0) {
                System.out.println("✅ Good I/O latency (<10ms)");
            } else {
                System.out.println("⚠️  High I/O latency (>10ms) - check
storage");
            }

            if (stats.queueDepth > 16) {
                System.out.println("⚠️  High queue depth - possible I/O
bottleneck");
            }

            // Recommendations
            System.out.println("\n=== Optimization Recommendations ===");

            if (stats.writeThroughput < 200) {
                System.out.println("Consider faster storage for write
throughput");
            }

            if (stats.avgLatency > 5) {
                System.out.println("Consider NVMe SSDs for lower latency");
            }
```

```
            System.out.println("Monitor with: iostat -x 1");
            System.out.println("Check disk utilization: iotop");
        }

        private static class IOStats {
            final int readIOPS;
            final int writeIOPS;
            final double readThroughput;
            final double writeThroughput;
            final double avgLatency;
            final int queueDepth;

            IOStats(int readIOPS, int writeIOPS, double readThroughput,
                    double writeThroughput, double avgLatency, int queueDepth) {
                this.readIOPS = readIOPS;
                this.writeIOPS = writeIOPS;
                this.readThroughput = readThroughput;
                this.writeThroughput = writeThroughput;
                this.avgLatency = avgLatency;
                this.queueDepth = queueDepth;
            }
        }
    }
}
```

📊 Monitoring

Simple Explanation

Kafka monitoring provides visibility into cluster health, performance, and operational status. It uses JMX metrics, specialized tools, and consumer lag tracking to ensure optimal performance and prevent issues before they impact applications.

Problem It Solves

- **Proactive issue detection**: Identify problems before they cause outages
- **Performance optimization**: Monitor and tune system performance
- **Capacity planning**: Understand resource usage and growth trends
- **Operational visibility**: Track system health and application behavior

Monitoring Architecture

```
Kafka Monitoring Stack:

Data Sources:
┌─────────────────────────────────────────────────────────────┐
│ Kafka Brokers (JMX)  │ Producers (JMX)  │ Consumers (JMX) │
│ - Broker metrics     │ - Send rates     │ - Lag metrics   │
│ - Topic metrics      │ - Error rates    │ - Fetch rates   │
```

```
| - Partition metrics  | - Latency        | - Commit rates  |
└──────────────────────────────────────────────────────────┘
                              ↓
Collection Layer:
┌──────────────────────────────────────────────────────────┐
| JMX Exporters | Burrow | Kafka Manager | Prometheus      |
└──────────────────────────────────────────────────────────┘
                              ↓
Visualization & Alerting:
┌──────────────────────────────────────────────────────────┐
| Grafana Dashboards | Alert Manager | PagerDuty/Slack    |
└──────────────────────────────────────────────────────────┘
```

JMX Metrics

Comprehensive JMX Monitoring

```
import javax.management.*;
import javax.management.remote.*;

/**
 * Comprehensive JMX metrics monitoring for Kafka
 */
public class KafkaJMXMonitor {

    private final MBeanServerConnection connection;
    private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(3);

    public KafkaJMXMonitor(String brokerHost, int jmxPort) throws IOException {
        String serviceURL =
String.format("service:jmx:rmi:///jndi/rmi://%s:%d/jmxrmi",
            brokerHost, jmxPort);

        JMXServiceURL url = new JMXServiceURL(serviceURL);
        JMXConnector connector = JMXConnectorFactory.connect(url);
        this.connection = connector.getMBeanServerConnection();

        System.out.printf("Connected to Kafka JMX at %s:%d%n", brokerHost,
jmxPort);
    }

    /**
     * Start comprehensive monitoring
     */
    public void startMonitoring() {
        // Monitor broker metrics every 30 seconds
        scheduler.scheduleAtFixedRate(this::monitorBrokerMetrics, 0, 30,
TimeUnit.SECONDS);

        // Monitor topic metrics every 60 seconds
```

```
        scheduler.scheduleAtFixedRate(this::monitorTopicMetrics, 0, 60,
TimeUnit.SECONDS);

        // Monitor consumer lag every 30 seconds
        scheduler.scheduleAtFixedRate(this::monitorConsumerLag, 0, 30,
TimeUnit.SECONDS);

        System.out.println("Started comprehensive JMX monitoring");
    }

    /**
     * Monitor critical broker metrics
     */
    private void monitorBrokerMetrics() {
        try {
            System.out.println("\n=== Broker Metrics ===");

            // Message throughput
            double messagesInPerSec = getDoubleAttribute(
                "kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec",
"OneMinuteRate");
            double bytesInPerSec = getDoubleAttribute(
                "kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec",
"OneMinuteRate");
            double bytesOutPerSec = getDoubleAttribute(
                "kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec",
"OneMinuteRate");

            System.out.printf("Messages In: %,.0f msgs/sec%n", messagesInPerSec);
            System.out.printf("Bytes In: %,.0f bytes/sec (%.2f MB/sec)%n",
                bytesInPerSec, bytesInPerSec / (1024 * 1024));
            System.out.printf("Bytes Out: %,.0f bytes/sec (%.2f MB/sec)%n",
                bytesOutPerSec, bytesOutPerSec / (1024 * 1024));

            // Request metrics
            double producerRequestRate = getDoubleAttribute(

"kafka.network:type=RequestMetrics,name=RequestsPerSec,request=Produce",
"OneMinuteRate");
            double fetchRequestRate = getDoubleAttribute(

"kafka.network:type=RequestMetrics,name=RequestsPerSec,request=FetchConsumer",
"OneMinuteRate");

            System.out.printf("Producer Requests: %,.0f req/sec%n",
producerRequestRate);
            System.out.printf("Fetch Requests: %,.0f req/sec%n",
fetchRequestRate);

            // Error rates
            double failedProduceRate = getDoubleAttribute(

"kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec",
"OneMinuteRate");
```

```
            double failedFetchRate = getDoubleAttribute(
"kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestsPerSec",
"OneMinuteRate");

            if (failedProduceRate > 0 || failedFetchRate > 0) {
                System.out.printf("⚠  Failed Produce: %.2f req/sec%n",
failedProduceRate);
                System.out.printf("⚠  Failed Fetch: %.2f req/sec%n",
failedFetchRate);
            }

            // Resource utilization
            long requestQueueSize = getLongAttribute(
                "kafka.network:type=RequestChannel,name=RequestQueueSize",
"Value");
            long responseQueueSize = getLongAttribute(
                "kafka.network:type=RequestChannel,name=ResponseQueueSize",
"Value");

            System.out.printf("Request Queue: %d%n", requestQueueSize);
            System.out.printf("Response Queue: %d%n", responseQueueSize);

            if (requestQueueSize > 500) {
                System.out.println("⚠  High request queue size - possible
performance issue");
            }

        } catch (Exception e) {
            System.err.println("Error monitoring broker metrics: " +
e.getMessage());
        }
    }

    /**
     * Monitor topic-specific metrics
     */
    private void monitorTopicMetrics() {
        try {
            System.out.println("\n=== Topic Metrics ===");

            // Get all topic names
            Set<ObjectName> topicObjects = connection.queryNames(
                new
ObjectName("kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=*"),
null);

            for (ObjectName obj : topicObjects) {
                String topic = obj.getKeyProperty("topic");
                if (topic != null && !topic.startsWith("__")) { // Skip internal
topics
                    monitorSingleTopic(topic);
                }
            }
```

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

    private void monitorSingleTopic(String topic) {
        try {
            double messagesIn = getDoubleAttribute(

String.format("kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=%s
", topic),
                "OneMinuteRate");
            double bytesIn = getDoubleAttribute(

String.format("kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=%s",
topic),
                "OneMinuteRate");

            if (messagesIn > 0) { // Only show active topics
                System.out.printf("Topic: %s - Messages: %,.0f/sec, Bytes:
%,.0f/sec%n",
                    topic, messagesIn, bytesIn);
            }

        } catch (Exception e) {
            // Topic might not have metrics yet
        }
    }

    /**
     * Monitor under-replicated partitions
     */
    private void monitorReplicationHealth() {
        try {
            long underReplicatedPartitions = getLongAttribute(
                "kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions",
"Value");

            if (underReplicatedPartitions > 0) {
                System.out.printf("🚨 Under-replicated partitions: %d%n",
underReplicatedPartitions);
            }

            double isrExpandsPerSec = getDoubleAttribute(
                "kafka.server:type=ReplicaManager,name=IsrExpandsPerSec",
"OneMinuteRate");
            double isrShrinksPerSec = getDoubleAttribute(
                "kafka.server:type=ReplicaManager,name=IsrShrinksPerSec",
"OneMinuteRate");

            if (isrShrinksPerSec > 0) {
                System.out.printf("⚠️  ISR shrinks: %.2f/sec%n",
```

```
isrShrinksPerSec);
            }
            if (isrExpandsPerSec > 0) {
                System.out.printf("✅ ISR expands: %.2f/sec%n",
isrExpandsPerSec);
            }

        } catch (Exception e) {
            System.err.println("Error monitoring replication health: " +
e.getMessage());
        }
    }

    /**
     * Monitor consumer lag (from consumer clients)
     */
    private void monitorConsumerLag() {
        try {
            System.out.println("\n=== Consumer Lag Monitoring ===");

            // Note: Consumer lag metrics are available from consumer clients, not
brokers
            // This demonstrates how to access them if consumers are reporting to
same JMX server

            Set<ObjectName> consumerObjects = connection.queryNames(
                new ObjectName("kafka.consumer:type=consumer-fetch-manager-
metrics,client-id=*"), null);

            for (ObjectName obj : consumerObjects) {
                String clientId = obj.getKeyProperty("client-id");
                monitorConsumerClient(clientId);
            }

            if (consumerObjects.isEmpty()) {
                System.out.println("No consumer metrics found (consumers may be on
different JVMs)");
            }

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

    private void monitorConsumerClient(String clientId) {
        try {
            // Records lag max across all partitions for this consumer
            Double recordsLagMax = getDoubleAttributeOrNull(
                String.format("kafka.consumer:type=consumer-fetch-manager-
metrics,client-id=%s", clientId),
                "records-lag-max");

            if (recordsLagMax != null && recordsLagMax > 0) {
```

```
                System.out.printf("Consumer %s - Max Lag: %.0f records%n",
clientId, recordsLagMax);

                if (recordsLagMax > 10000) {
                    System.out.printf("🚨 High lag detected for %s%n", clientId);
                }
            }

        } catch (Exception e) {
            // Consumer might not be active
        }
    }

    /**
     * Create performance dashboard
     */
    public void printPerformanceDashboard() {
        try {
            System.out.println("\n" + "=".repeat(60));
            System.out.println("           KAFKA PERFORMANCE DASHBOARD");
            System.out.println("=".repeat(60));

            // Throughput section
            double messagesInPerSec = getDoubleAttribute(
                "kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec",
"OneMinuteRate");
            double bytesInPerSec = getDoubleAttribute(
                "kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec",
"OneMinuteRate");

            System.out.println("\n📊 THROUGHPUT");
            System.out.printf("   Messages/sec: %,12.0f%n", messagesInPerSec);
            System.out.printf("   MB/sec:       %,12.2f%n", bytesInPerSec / (1024
* 1024));

            // Latency section (if available)
            System.out.println("\n⏱️  LATENCY");
            try {
                double totalTimeMs = getDoubleAttribute(

"kafka.network:type=RequestMetrics,name=TotalTimeMs,request=Produce", "Mean");
                System.out.printf("   Produce latency: %8.2f ms%n", totalTimeMs);
            } catch (Exception e) {
                System.out.println("   Produce latency: N/A");
            }

            // Health section
            System.out.println("\n🏥 HEALTH");
            long underReplicated = getLongAttribute(
                "kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions",
"Value");

            if (underReplicated == 0) {
                System.out.println("   Replication:     ☑ Healthy");
```

```
            } else {
                System.out.printf("   Replication:     %d under-replicated%n",
underReplicated);
            }

            // Queue health
            long requestQueue = getLongAttribute(
                "kafka.network:type=RequestChannel,name=RequestQueueSize",
"Value");

            if (requestQueue < 100) {
                System.out.println("   Request Queue:   Healthy");
            } else if (requestQueue < 500) {
                System.out.printf("   Request Queue:   Busy (%d)%n",
requestQueue);
            } else {
                System.out.printf("   Request Queue:   Overloaded (%d)%n",
requestQueue);
            }

            System.out.println("=".repeat(60));

        } catch (Exception e) {
            System.err.println("Error creating performance dashboard: " +
e.getMessage());
        }
    }

    // Utility methods for JMX access
    private double getDoubleAttribute(String objectName, String attribute) throws
Exception {
        ObjectName obj = new ObjectName(objectName);
        Object value = connection.getAttribute(obj, attribute);
        return value instanceof Number ? ((Number) value).doubleValue() : 0.0;
    }

    private Double getDoubleAttributeOrNull(String objectName, String attribute) {
        try {
            return getDoubleAttribute(objectName, attribute);
        } catch (Exception e) {
            return null;
        }
    }

    private long getLongAttribute(String objectName, String attribute) throws
Exception {
        ObjectName obj = new ObjectName(objectName);
        Object value = connection.getAttribute(obj, attribute);
        return value instanceof Number ? ((Number) value).longValue() : 0L;
    }

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

```
    // Example usage
    public static void main(String[] args) {
        try {
            KafkaJMXMonitor monitor = new KafkaJMXMonitor("localhost", 9999);
            monitor.startMonitoring();

            // Print dashboard every 60 seconds
            ScheduledExecutorService dashboardScheduler =
Executors.newSingleThreadScheduledExecutor();
            dashboardScheduler.scheduleAtFixedRate(
                monitor::printPerformanceDashboard, 0, 60, TimeUnit.SECONDS);

            // Run for demo purposes
            Thread.sleep(300000); // 5 minutes

            monitor.stop();
            dashboardScheduler.shutdown();

        } catch (Exception e) {
            System.err.println("Failed to start monitoring: " + e.getMessage());
        }
    }
}
```

Consumer Lag Monitoring

Advanced Consumer Lag Tracking

```
/**
 * Advanced consumer lag monitoring and alerting
 */
public class ConsumerLagMonitor {

    private final AdminClient adminClient;
    private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(2);
    private final Map<String, ConsumerGroupLagHistory> lagHistory = new
ConcurrentHashMap<>();

    public ConsumerLagMonitor(Properties adminProps) {
        this.adminClient = AdminClient.create(adminProps);
    }

    /**
     * Start comprehensive consumer lag monitoring
     */
    public void startMonitoring() {
        // Monitor lag every 30 seconds
        scheduler.scheduleAtFixedRate(this::monitorAllConsumerGroups, 0, 30,
TimeUnit.SECONDS);
```

```
        // Generate lag report every 5 minutes
        scheduler.scheduleAtFixedRate(this::generateLagReport, 0, 300,
TimeUnit.SECONDS);

        System.out.println("Started consumer lag monitoring");
    }

    /**
     * Monitor all consumer groups for lag
     */
    private void monitorAllConsumerGroups() {
        try {
            // Get all consumer groups
            ListConsumerGroupsResult groupsResult =
adminClient.listConsumerGroups();
            Collection<ConsumerGroupListing> groups = groupsResult.all().get();

            for (ConsumerGroupListing group : groups) {
                monitorConsumerGroup(group.groupId());
            }

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

    /**
     * Monitor specific consumer group
     */
    private void monitorConsumerGroup(String groupId) {
        try {
            // Get consumer group description
            DescribeConsumerGroupsResult groupResult =
                adminClient.describeConsumerGroups(Arrays.asList(groupId));
            ConsumerGroupDescription description =
groupResult.all().get().get(groupId);

            if (description.state() != ConsumerGroupState.STABLE) {
                System.out.printf("⚠  Group %s is in state: %s%n", groupId,
description.state());
                return;
            }

            // Get consumer group offsets
            ListConsumerGroupOffsetsResult offsetsResult =
                adminClient.listConsumerGroupOffsets(groupId);
            Map<TopicPartition, OffsetAndMetadata> offsets =
offsetsResult.partitionsToOffsetAndMetadata().get();

            if (offsets.isEmpty()) {
                return; // No offsets for this group
            }
```

```
            // Get latest offsets for these partitions
            Map<TopicPartition, OffsetSpec> latestOffsetSpecs =
offsets.keySet().stream()
                .collect(Collectors.toMap(tp -> tp, tp -> OffsetSpec.latest()));

            ListOffsetsResult latestOffsetsResult =
adminClient.listOffsets(latestOffsetSpecs);
            Map<TopicPartition, ListOffsetsResult.ListOffsetsResultInfo>
latestOffsets =
                latestOffsetsResult.all().get();

            // Calculate lag for each partition
            ConsumerGroupLag groupLag = calculateLag(groupId, offsets,
latestOffsets);

            // Update lag history
            updateLagHistory(groupId, groupLag);

            // Check for alerts
            checkLagAlerts(groupId, groupLag);

        } catch (Exception e) {
            System.err.printf("Error monitoring group %s: %s%n", groupId,
e.getMessage());
        }
    }

    /**
     * Calculate lag for consumer group
     */
    private ConsumerGroupLag calculateLag(String groupId,
                                          Map<TopicPartition, OffsetAndMetadata>
consumerOffsets,
                                          Map<TopicPartition,
ListOffsetsResult.ListOffsetsResultInfo> latestOffsets) {

        Map<TopicPartition, Long> partitionLags = new HashMap<>();
        long totalLag = 0;
        long maxLag = 0;
        TopicPartition maxLagPartition = null;

        for (Map.Entry<TopicPartition, OffsetAndMetadata> entry :
consumerOffsets.entrySet()) {
            TopicPartition tp = entry.getKey();
            long consumerOffset = entry.getValue().offset();

            ListOffsetsResult.ListOffsetsResultInfo latestInfo =
latestOffsets.get(tp);
            if (latestInfo != null) {
                long latestOffset = latestInfo.offset();
                long lag = Math.max(0, latestOffset - consumerOffset);

                partitionLags.put(tp, lag);
```

```
                totalLag += lag;

                if (lag > maxLag) {
                    maxLag = lag;
                    maxLagPartition = tp;
                }
            }
        }

        return new ConsumerGroupLag(groupId, System.currentTimeMillis(),
partitionLags,
                                    totalLag, maxLag, maxLagPartition);
    }

    /**
     * Update lag history for trend analysis
     */
    private void updateLagHistory(String groupId, ConsumerGroupLag currentLag) {
        lagHistory.computeIfAbsent(groupId, k -> new ConsumerGroupLagHistory(k))
                  .addLagMeasurement(currentLag);
    }

    /**
     * Check for lag-based alerts
     */
    private void checkLagAlerts(String groupId, ConsumerGroupLag lag) {
        // Alert thresholds
        long maxLagThreshold = 10000;        // 10k messages
        long totalLagThreshold = 50000;      // 50k total messages
        double lagVelocityThreshold = 1000;  // 1k messages/minute increase

        // Max lag alert
        if (lag.maxLag > maxLagThreshold) {
            System.out.printf("🚨 HIGH LAG ALERT: Group %s has partition lag of
%,d on %s%n",
                groupId, lag.maxLag, lag.maxLagPartition);
        }

        // Total lag alert
        if (lag.totalLag > totalLagThreshold) {
            System.out.printf("🚨 TOTAL LAG ALERT: Group %s has total lag of
%,d%n",
                groupId, lag.totalLag);
        }

        // Lag velocity alert (increasing rapidly)
        ConsumerGroupLagHistory history = lagHistory.get(groupId);
        if (history != null) {
            double velocity = history.calculateLagVelocity();
            if (velocity > lagVelocityThreshold) {
                System.out.printf("🚨 LAG VELOCITY ALERT: Group %s lag increasing
at %,.0f msgs/min%n",
                    groupId, velocity);
            }
```

```
            }
        }

        /**
         * Generate comprehensive lag report
         */
        private void generateLagReport() {
            System.out.println("\n" + "=".repeat(80));
            System.out.println("                    CONSUMER LAG REPORT");
            System.out.println("=".repeat(80));

            if (lagHistory.isEmpty()) {
                System.out.println("No consumer groups found");
                return;
            }

            // Sort groups by total lag (highest first)
            List<Map.Entry<String, ConsumerGroupLagHistory>> sortedGroups =
    lagHistory.entrySet()
                .stream()
                .sorted((e1, e2) -> Long.compare(
                    e2.getValue().getLatestLag().totalLag,
                    e1.getValue().getLatestLag().totalLag))
                .collect(Collectors.toList());

            System.out.printf("%-30s %12s %12s %12s %15s%n",
                "Consumer Group", "Total Lag", "Max Lag", "Partitions", "Trend");
            System.out.println("-".repeat(80));

            for (Map.Entry<String, ConsumerGroupLagHistory> entry : sortedGroups) {
                String groupId = entry.getKey();
                ConsumerGroupLagHistory history = entry.getValue();
                ConsumerGroupLag latest = history.getLatestLag();

                String trend = formatTrend(history.calculateLagVelocity());

                System.out.printf("%-30s %,12d %,12d %12d %15s%n",
                    truncate(groupId, 30),
                    latest.totalLag,
                    latest.maxLag,
                    latest.partitionLags.size(),
                    trend);

                // Show top lagging partitions for this group
                if (latest.totalLag > 1000) {
                    showTopLaggingPartitions(latest, 3);
                }
            }

            System.out.println("=".repeat(80));
        }

        /**
         * Show top lagging partitions for a consumer group
```

```
     */
    private void showTopLaggingPartitions(ConsumerGroupLag lag, int topN) {
        List<Map.Entry<TopicPartition, Long>> topPartitions =
lag.partitionLags.entrySet()
            .stream()
            .sorted(Map.Entry.<TopicPartition, Long>comparingByValue().reversed())
            .limit(topN)
            .collect(Collectors.toList());

        for (Map.Entry<TopicPartition, Long> entry : topPartitions) {
            if (entry.getValue() > 0) {
                System.out.printf("    %s-%d: %,d%n",
                    entry.getKey().topic(),
                    entry.getKey().partition(),
                    entry.getValue());
            }
        }
    }

    private String formatTrend(double velocity) {
        if (velocity > 100) {
            return "📈 Increasing";
        } else if (velocity < -100) {
            return "📉 Decreasing";
        } else {
            return "➡️ Stable";
        }
    }

    private String truncate(String str, int maxLength) {
        return str.length() <= maxLength ? str : str.substring(0, maxLength - 3) +
"...";
    }

    /**
     * Data classes for lag tracking
     */
    public static class ConsumerGroupLag {
        final String groupId;
        final long timestamp;
        final Map<TopicPartition, Long> partitionLags;
        final long totalLag;
        final long maxLag;
        final TopicPartition maxLagPartition;

        ConsumerGroupLag(String groupId, long timestamp, Map<TopicPartition, Long>
partitionLags,
                         long totalLag, long maxLag, TopicPartition
maxLagPartition) {
            this.groupId = groupId;
            this.timestamp = timestamp;
            this.partitionLags = partitionLags;
            this.totalLag = totalLag;
            this.maxLag = maxLag;
```

```
            this.maxLagPartition = maxLagPartition;
        }
    }

    public static class ConsumerGroupLagHistory {
        private final String groupId;
        private final LinkedList<ConsumerGroupLag> lagHistory = new LinkedList<>
();
        private final int maxHistorySize = 100; // Keep last 100 measurements

        ConsumerGroupLagHistory(String groupId) {
            this.groupId = groupId;
        }

        void addLagMeasurement(ConsumerGroupLag lag) {
            lagHistory.addLast(lag);
            if (lagHistory.size() > maxHistorySize) {
                lagHistory.removeFirst();
            }
        }

        ConsumerGroupLag getLatestLag() {
            return lagHistory.isEmpty() ? null : lagHistory.getLast();
        }

        /**
         * Calculate lag velocity (change per minute)
         */
        double calculateLagVelocity() {
            if (lagHistory.size() < 2) {
                return 0.0;
            }

            ConsumerGroupLag latest = lagHistory.getLast();
            ConsumerGroupLag previous = lagHistory.get(lagHistory.size() - 2);

            long timeDiffMs = latest.timestamp - previous.timestamp;
            if (timeDiffMs <= 0) {
                return 0.0;
            }

            long lagDiff = latest.totalLag - previous.totalLag;

            // Convert to change per minute
            return (double) lagDiff * 60000.0 / timeDiffMs;
        }
    }

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

Monitoring Tools

Tool Comparison and Integration

```
/**
 * Comprehensive monitoring tools integration and comparison
 */
public class MonitoringToolsManager {

    /**
     * Monitoring tool configurations and capabilities
     */
    public static class MonitoringToolComparison {

        public static void compareMonitoringTools() {
            System.out.println("=== Kafka Monitoring Tools Comparison ===");

            MonitoringTool[] tools = {
                new MonitoringTool("Confluent Control Center",
                    "Commercial", "Excellent", "High", "Native Kafka",
                    "Enterprise features, intuitive UI, comprehensive"),

                new MonitoringTool("Conduktor",
                    "Commercial", "Excellent", "Medium", "Multi-cluster support",
                    "User-friendly, data masking, good visualization"),

                new MonitoringTool("Burrow (LinkedIn)",
                    "Open Source", "Good", "Low", "Consumer lag focus",
                    "Specialized lag monitoring, no thresholds needed"),

                new MonitoringTool("Kafka Manager (Yahoo)",
                    "Open Source", "Good", "Low", "Cluster management",
                    "Simple UI, basic monitoring, easy setup"),

                new MonitoringTool("Prometheus + Grafana",
                    "Open Source", "Excellent", "Medium", "Customizable",
                    "Highly flexible, requires setup, great alerting"),

                new MonitoringTool("Kafka Exporter",
                    "Open Source", "Good", "Low", "Metrics export",
                    "Lightweight, Prometheus integration"),

                new MonitoringTool("AKHQ",
                    "Open Source", "Good", "Low", "Web UI",
                    "Modern UI, topic browsing, consumer groups"),

                new MonitoringTool("Kafdrop",
                    "Open Source", "Fair", "Low", "Simple UI",
                    "Lightweight, basic features, easy deployment")
            };
```

```
            printToolComparison(tools);
            printRecommendations();
        }

        private static void printToolComparison(MonitoringTool[] tools) {
            System.out.printf("%-25s %-12s %-10s %-10s %-20s%n",
                "Tool", "License", "Features", "Complexity", "Strengths");
            System.out.println("-".repeat(80));

            for (MonitoringTool tool : tools) {
                System.out.printf("%-25s %-12s %-10s %-10s %-20s%n",
                    tool.name,
                    tool.license,
                    tool.featureRating,
                    tool.complexity,
                    tool.keyStrength);
            }
        }

        private static void printRecommendations() {
            System.out.println("\n=== Recommendations by Use Case ===");

            System.out.println("\n🏢 Enterprise Production:");
            System.out.println("   ✅ Confluent Control Center - Most
comprehensive");
            System.out.println("   ✅ Conduktor - User-friendly alternative");

            System.out.println("\n🔧 Open Source / Cost-Conscious:");
            System.out.println("   ✅ Prometheus + Grafana - Most flexible");
            System.out.println("   ✅ AKHQ - Modern UI, good features");

            System.out.println("\n📊 Consumer Lag Focus:");
            System.out.println("   ✅ Burrow - Purpose-built for lag
monitoring");
            System.out.println("   ✅ Kafka Lag Exporter - Prometheus
integration");

            System.out.println("\n🚀 Quick Setup / Development:");
            System.out.println("   ✅ Kafdrop - Simple and fast");
            System.out.println("   ✅ Kafka Manager - Basic but functional");
        }

        private static class MonitoringTool {
            final String name;
            final String license;
            final String featureRating;
            final String complexity;
            final String keyStrength;
            final String description;

            MonitoringTool(String name, String license, String featureRating,
String complexity,
                           String keyStrength, String description) {
                this.name = name;
```

```
                this.license = license;
                this.featureRating = featureRating;
                this.complexity = complexity;
                this.keyStrength = keyStrength;
                this.description = description;
            }
        }
    }

    /**
     * Burrow integration for consumer lag monitoring
     */
    public static class BurrowIntegration {

        private final String burrowUrl;
        private final HttpClient httpClient;

        public BurrowIntegration(String burrowUrl) {
            this.burrowUrl = burrowUrl;
            this.httpClient = HttpClient.newHttpClient();
        }

        /**
         * Get consumer groups from Burrow
         */
        public List<String> getConsumerGroups(String cluster) {
            try {
                String url = String.format("%s/v3/kafka/%s/consumer", burrowUrl,
cluster);
                HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(url))
                    .GET()
                    .build();

                HttpResponse<String> response = httpClient.send(request,
                    HttpResponse.BodyHandlers.ofString());

                if (response.statusCode() == 200) {
                    // Parse JSON response (simplified - use proper JSON library
in production)
                    return parseConsumerGroups(response.body());
                }

            } catch (Exception e) {
                System.err.println("Error getting consumer groups from Burrow: " +
e.getMessage());
            }

            return Collections.emptyList();
        }

        /**
         * Get consumer group lag from Burrow
         */
```

```
        public BurrowLagStatus getConsumerGroupLag(String cluster, String group) {
            try {
                String url = String.format("%s/v3/kafka/%s/consumer/%s/lag",
                    burrowUrl, cluster, group);

                HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(url))
                    .GET()
                    .build();

                HttpResponse<String> response = httpClient.send(request,
                    HttpResponse.BodyHandlers.ofString());

                if (response.statusCode() == 200) {
                    return parseLagStatus(response.body());
                }

            } catch (Exception e) {
                System.err.println("Error getting lag from Burrow: " +
e.getMessage());
            }

            return null;
        }

        private List<String> parseConsumerGroups(String json) {
            // Simplified JSON parsing - use proper JSON library in production
            List<String> groups = new ArrayList<>();
            // Parse consumer groups from JSON response
            groups.add("example-consumer-group");
            return groups;
        }

        private BurrowLagStatus parseLagStatus(String json) {
            // Simplified parsing - use proper JSON library in production
            return new BurrowLagStatus("OK", 1250, System.currentTimeMillis());
        }

        public static class BurrowLagStatus {
            final String status;
            final long totalLag;
            final long timestamp;

            BurrowLagStatus(String status, long totalLag, long timestamp) {
                this.status = status;
                this.totalLag = totalLag;
                this.timestamp = timestamp;
            }
        }
    }

    /**
     * Prometheus metrics exporter for Kafka
     */
```

```
    public static class PrometheusExporter {

        private final MBeanServerConnection jmxConnection;

        public PrometheusExporter(MBeanServerConnection jmxConnection) {
            this.jmxConnection = jmxConnection;
        }

        /**
         * Export Kafka metrics in Prometheus format
         */
        public String exportMetrics() {
            StringBuilder metrics = new StringBuilder();

            try {
                // Export broker metrics
                exportBrokerMetrics(metrics);

                // Export topic metrics
                exportTopicMetrics(metrics);

                // Export consumer metrics
                exportConsumerMetrics(metrics);

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

            return metrics.toString();
        }

        private void exportBrokerMetrics(StringBuilder metrics) {
            try {
                // Messages in per second
                double messagesInPerSec = getDoubleAttribute(
                    "kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec",
"OneMinuteRate");
                metrics.append(String.format("kafka_server_messages_in_per_sec
%.2f%n", messagesInPerSec));

                // Bytes in per second
                double bytesInPerSec = getDoubleAttribute(
                    "kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec",
"OneMinuteRate");
                metrics.append(String.format("kafka_server_bytes_in_per_sec
%.2f%n", bytesInPerSec));

                // Under-replicated partitions
                long underReplicated = getLongAttribute(

"kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions", "Value");

metrics.append(String.format("kafka_server_under_replicated_partitions %d%n",
underReplicated));
```

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

        private void exportTopicMetrics(StringBuilder metrics) {
            try {
                // Get all topics
                Set<ObjectName> topicObjects = jmxConnection.queryNames(
                    new
ObjectName("kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=*"),
null);

                for (ObjectName obj : topicObjects) {
                    String topic = obj.getKeyProperty("topic");
                    if (topic != null && !topic.startsWith("__")) {

                        double messagesIn = getDoubleAttribute(obj.toString(),
"OneMinuteRate");
                        metrics.append(String.format(
                            "kafka_topic_messages_in_per_sec{topic=\"%s\"}
%.2f%n", topic, messagesIn));
                    }
                }

            } catch (Exception e) {
                System.err.println("Error exporting topic metrics: " +
e.getMessage());
            }
        }

        private void exportConsumerMetrics(StringBuilder metrics) {
            try {
                // Consumer lag metrics (if available)
                Set<ObjectName> consumerObjects = jmxConnection.queryNames(
                    new ObjectName("kafka.consumer:type=consumer-fetch-manager-
metrics,client-id=*"), null);

                for (ObjectName obj : consumerObjects) {
                    String clientId = obj.getKeyProperty("client-id");

                    try {
                        double recordsLagMax = getDoubleAttribute(obj.toString(),
"records-lag-max");
                        metrics.append(String.format(
                            "kafka_consumer_records_lag_max{client_id=\"%s\"}
%.2f%n", clientId, recordsLagMax));
                    } catch (Exception e) {
                        // Consumer might not have this metric
                    }
                }
```

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

        private double getDoubleAttribute(String objectName, String attribute)
throws Exception {
            ObjectName obj = new ObjectName(objectName);
            Object value = jmxConnection.getAttribute(obj, attribute);
            return value instanceof Number ? ((Number) value).doubleValue() : 0.0;
        }

        private long getLongAttribute(String objectName, String attribute) throws
Exception {
            ObjectName obj = new ObjectName(objectName);
            Object value = jmxConnection.getAttribute(obj, attribute);
            return value instanceof Number ? ((Number) value).longValue() : 0L;
        }
    }

    /**
     * Unified monitoring dashboard
     */
    public static class UnifiedMonitoringDashboard {

        private final KafkaJMXMonitor jmxMonitor;
        private final ConsumerLagMonitor lagMonitor;
        private final BurrowIntegration burrow;

        public UnifiedMonitoringDashboard(KafkaJMXMonitor jmxMonitor,
                                          ConsumerLagMonitor lagMonitor,
                                          BurrowIntegration burrow) {
            this.jmxMonitor = jmxMonitor;
            this.lagMonitor = lagMonitor;
            this.burrow = burrow;
        }

        public void generateComprehensiveReport() {
            System.out.println("\n" + "=".repeat(100));
            System.out.println("                              KAFKA MONITORING
DASHBOARD");
            System.out.println("=".repeat(100));

            // JMX metrics summary
            if (jmxMonitor != null) {
                jmxMonitor.printPerformanceDashboard();
            }

            // Consumer lag summary
            if (lagMonitor != null) {
                System.out.println("\n📊 CONSUMER LAG SUMMARY");
                // lagMonitor.generateLagReport(); // Would be called here
            }
```

```
            // External tool integration
            if (burrow != null) {
                System.out.println("\n🔍 BURROW INTEGRATION");
                List<String> groups = burrow.getConsumerGroups("local");
                System.out.printf("   Monitored groups: %d%n", groups.size());
            }

            System.out.println("=".repeat(100));
        }
    }
}
```

⚖️ Comparisons & Trade-offs

Reliability Configuration Trade-offs

Configuration	Data Safety	Availability	Performance	Operational Complexity
RF=3, min.insync=2, acks=all	☑ High	☑ Good	⚠ Medium	☑ Simple
RF=5, min.insync=3, acks=all	☑ Very High	☑ Good	❌ Lower	⚠ Complex
RF=3, min.insync=1, acks=1	⚠ Medium	☑ High	☑ High	☑ Simple
RF=1, acks=0	❌ Low	❌ Poor	☑ Very High	☑ Simple

Performance Optimization Trade-offs

Optimization	Throughput	Latency	Resource Usage	Complexity
Large Batches	☑ High	❌ High	⚠ Medium	☑ Low
LZ4 Compression	☑ High	☑ Low	⚠ Medium	☑ Low
ZSTD Compression	☑ High	⚠ Medium	❌ High	☑ Low
No Compression	⚠ Medium	☑ Very Low	☑ Low	☑ Very Low
Page Cache Tuning	☑ High	☑ Medium	⚠ Requires tuning	⚠ Medium

Monitoring Tool Comparison

Tool	Cost	Setup Complexity	Feature Completeness	Learning Curve
Confluent Control Center	❌ High	☑ Low	☑ Excellent	☑ Low

Tool	Cost	Setup Complexity	Feature Completeness	Learning Curve
Conduktor	⚠ Medium	☑ Low	☑ Very Good	☑ Low
Prometheus + Grafana	☑ Free	❌ High	☑ Excellent	⚠ Medium
Burrow	☑ Free	⚠ Medium	⚠ Lag Focus	☑ Low
AKHQ	☑ Free	☑ Low	⚠ Good	☑ Low

Common Pitfalls & Best Practices

Reliability Pitfalls

❌ Common Mistakes

```
// DON'T - Insufficient replication
Properties badReliabilityConfig = new Properties();
badReliabilityConfig.put("default.replication.factor", "1"); // Single point of
failure
badReliabilityConfig.put("min.insync.replicas", "1");
badReliabilityConfig.put("acks", "0"); // No acknowledgment

// DON'T - Mismatched configuration
badReliabilityConfig.put("default.replication.factor", "3");
badReliabilityConfig.put("min.insync.replicas", "3"); // Can't tolerate any
failures
```

☑ Best Practices

```
// DO - Proper reliability configuration
Properties goodReliabilityConfig = new Properties();
goodReliabilityConfig.put("default.replication.factor", "3");
goodReliabilityConfig.put("min.insync.replicas", "2"); // Can tolerate 1 failure
goodReliabilityConfig.put("acks", "all");
goodReliabilityConfig.put("enable.idempotence", "true");
goodReliabilityConfig.put("retries", "Integer.MAX_VALUE");
```

Performance Pitfalls

❌ Performance Anti-patterns

```
// DON'T - No batching
Properties badPerformanceConfig = new Properties();
```

```
badPerformanceConfig.put("batch.size", "0"); // No batching
badPerformanceConfig.put("linger.ms", "0");

// DON'T - Wrong compression for use case
badPerformanceConfig.put("compression.type", "gzip"); // Slow for high-throughput

// DON'T - Tiny consumer fetches
badPerformanceConfig.put("fetch.min.bytes", "1");
badPerformanceConfig.put("max.poll.records", "10"); // Too small
```

☑ Performance Best Practices

```
// DO - Optimize for throughput
Properties goodPerformanceConfig = new Properties();
goodPerformanceConfig.put("batch.size", "65536"); // 64KB
goodPerformanceConfig.put("linger.ms", "10");
goodPerformanceConfig.put("compression.type", "lz4");

// DO - Efficient consumer configuration
goodPerformanceConfig.put("fetch.min.bytes", "65536"); // 64KB
goodPerformanceConfig.put("max.poll.records", "1000");
```

Monitoring Pitfalls

✗ Monitoring Mistakes

```
// DON'T - Only monitor basic metrics
public class BasicMonitoring {
    public void monitorOnlyThroughput() {
        // Missing: lag, errors, latency, resource utilization
        System.out.println("Messages/sec: " + getMessageRate());
    }
}
```

☑ Comprehensive Monitoring

```
// DO - Monitor all critical aspects
public class ComprehensiveMonitoring {
    public void monitorEverything() {
        // Throughput
        System.out.println("Messages/sec: " + getMessageRate());

        // Lag
        System.out.println("Consumer lag: " + getConsumerLag());

        // Errors
```

```
        System.out.println("Error rate: " + getErrorRate());

        // Resources
        System.out.println("CPU usage: " + getCpuUsage());
        System.out.println("Memory usage: " + getMemoryUsage());

        // Health
        System.out.println("Under-replicated partitions: " +
getUnderReplicatedPartitions());
    }
}
```

Best Practices Summary

☑ Reliability Best Practices

1. **Use replication factor 3** for production workloads
2. **Set min.insync.replicas = replication factor - 1** for availability
3. **Enable idempotency** for exactly-once semantics
4. **Use acks=all** for data durability
5. **Test failure scenarios** regularly

☑ Performance Best Practices

1. **Tune batching parameters** for your workload
2. **Use appropriate compression** (lz4 for most cases)
3. **Optimize OS page cache** usage
4. **Monitor and tune fetch sizes** for consumers
5. **Use multiple log directories** on separate disks

☑ Monitoring Best Practices

1. **Monitor consumer lag** continuously
2. **Set up alerting** for critical metrics
3. **Use multiple monitoring tools** for comprehensive coverage
4. **Monitor infrastructure metrics** alongside Kafka metrics
5. **Create dashboards** for different audiences (ops, dev, business)

🌍 Real-World Use Cases

Financial Trading Platform

```
public class FinancialTradingReliability {

    public static Properties getFinancialGradeConfig() {
        Properties config = new Properties();
```

```
        // Maximum reliability
        config.put("default.replication.factor", "5");
        config.put("min.insync.replicas", "3");
        config.put("acks", "all");
        config.put("enable.idempotence", "true");
        config.put("retries", "Integer.MAX_VALUE");
        config.put("max.in.flight.requests.per.connection", "1");

        // Performance with safety
        config.put("compression.type", "lz4");
        config.put("batch.size", "16384"); // Smaller batches for lower latency
        config.put("linger.ms", "1");

        return config;
    }
}
```

IoT Data Pipeline

```
public class IoTDataPipelinePerformance {

    public static Properties getHighThroughputConfig() {
        Properties config = new Properties();

        // High throughput optimization
        config.put("batch.size", "131072"); // 128KB
        config.put("linger.ms", "20");
        config.put("compression.type", "lz4");
        config.put("buffer.memory", "134217728"); // 128MB

        // Consumer optimization
        config.put("fetch.min.bytes", "1048576"); // 1MB
        config.put("max.poll.records", "5000");

        return config;
    }
}
```

Real-time Analytics

```
public class RealTimeAnalyticsMonitoring {

    public void setupAnalyticsMonitoring() {
        // Monitor lag closely for real-time requirements
        ConsumerLagMonitor lagMonitor = new ConsumerLagMonitor(getAdminProps());
        lagMonitor.startMonitoring();

        // Alert on lag > 1 second worth of data
        // Implementation would include alerting logic
```

```
        // Monitor end-to-end latency
        // Track from producer timestamp to consumer processing
    }
}
```

Version Highlights

Reliability Features Evolution

Version	Release Date	Reliability Features
4.0	September 2025	Enhanced idempotency, improved ISR management
3.0	September 2021	**Exactly-once semantics improvements**
2.8	April 2021	Enhanced exactly-once semantics
2.5	June 2020	**Incremental cooperative rebalancing**
2.0	July 2018	**Improved idempotency**
0.11	June 2017	**Exactly-once semantics introduced**
0.10	May 2016	**Idempotent producer**

Performance Improvements Timeline

Version	Performance Features
4.0	Better compression algorithms, tiered storage performance
3.6	**Tiered storage introduction**
3.0	Improved batching, better memory management
2.8	Enhanced page cache utilization
2.4	**Incremental cooperative rebalancing**
2.1	Better consumer fetch performance
1.0	**Improved log compaction performance**

Monitoring Evolution

Version	Monitoring Features
4.0	Enhanced JMX metrics, better observability
3.0	Improved consumer lag metrics
2.0	**Enhanced JMX metric structure**

Version	Monitoring Features
1.0	Better producer/consumer metrics
0.9	**JMX metrics introduction**

Current Recommendations (2025)

```
// Modern Kafka reliability and performance configuration (4.0+)
public static Properties modernKafkaConfig() {
    Properties config = new Properties();

    // Reliability (4.0 best practices)
    config.put("default.replication.factor", "3");
    config.put("min.insync.replicas", "2");
    config.put("acks", "all");
    config.put("enable.idempotence", "true");
    config.put("retries", "Integer.MAX_VALUE");

    // Performance (4.0 optimizations)
    config.put("compression.type", "zstd"); // Better compression in 4.0
    config.put("batch.size", "65536"); // Optimized default
    config.put("linger.ms", "10");

    // Monitoring (4.0 enhancements)
    config.put("metric.reporters", "io.prometheus.jmx.JmxCollector");

    return config;
}
```

Additional Resources

Official Documentation

- Kafka Reliability Guide
- Performance Tuning
- Monitoring Guide

Performance Tuning Resources

- Confluent Performance Tuning
- Page Cache Optimization
- JVM Tuning for Kafka

Monitoring Tools

- Burrow GitHub
- Kafka Exporter
- Confluent Control Center

- Conduktor

Best Practices Guides

- Reliability Best Practices
- Performance Best Practices
- Monitoring Best Practices

Last Updated: September 2025
Kafka Version: 4.0.0
Focus: Production-Ready Reliability, Performance & Monitoring

> **Pro Tip**: Modern Kafka deployments should prioritize observability, automate failure recovery, and tune for specific workload characteristics. The combination of proper reliability configuration, performance optimization, and comprehensive monitoring creates resilient, high-performance data platforms.