Spring Kafka Monitoring & Observability: Complete Developer Guide

A comprehensive guide covering all aspects of Spring Kafka monitoring and observability, including Micrometer metrics, health checks with Spring Boot Actuator, and distributed tracing with Sleuth and OpenTelemetry, with extensive Java examples and production patterns.

Table of Contents

- Micrometer Metrics
  - Producer Metrics
  - Consumer Metrics
- Health Checks (Spring Boot Actuator)
- Distributed Tracing (Sleuth, OpenTelemetry)
- Comparisons & Trade-offs
- Common Pitfalls & Best Practices
- Version Highlights

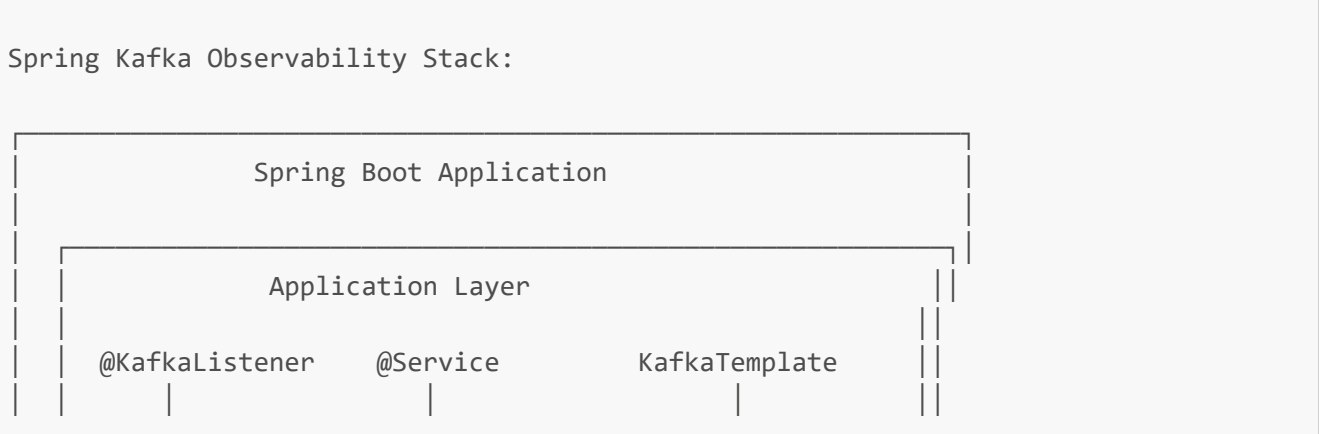
What is Spring Kafka Monitoring & Observability?

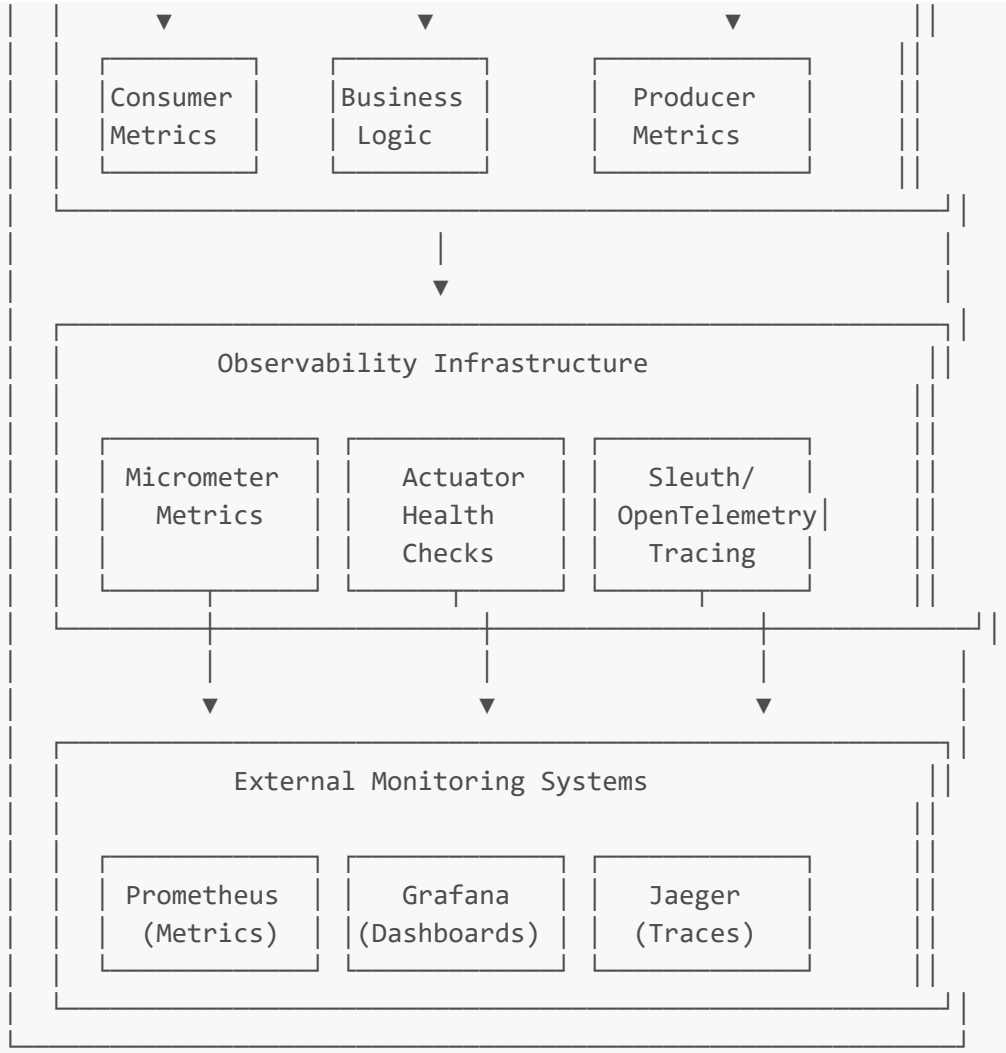
Simple Explanation: Spring Kafka Monitoring & Observability provides comprehensive visibility into Kafka producer and consumer behavior through metrics collection (Micrometer), health monitoring (Actuator), and distributed tracing (Sleuth/OpenTelemetry). This enables proactive monitoring, performance optimization, and troubleshooting of Kafka-based applications in production environments.

Why Monitoring & Observability is Critical:

- **Performance Monitoring**: Track throughput, latency, and error rates
- **Health Assurance**: Monitor connectivity and system health
- **Troubleshooting**: Identify bottlenecks and failure points
- **SLA Compliance**: Ensure service level agreements are met
- **Capacity Planning**: Understand usage patterns and scaling needs
- **Distributed Debugging**: Trace messages across microservices

Spring Kafka Observability Architecture:






```
Observability Data Flow:
1. Kafka operations generate telemetry data
2. Micrometer collects and formats metrics
3. Actuator exposes health check endpoints
4. Sleuth/OpenTelemetry creates distributed traces
5. External systems store and visualize data
6. Alerts and dashboards provide insights
```

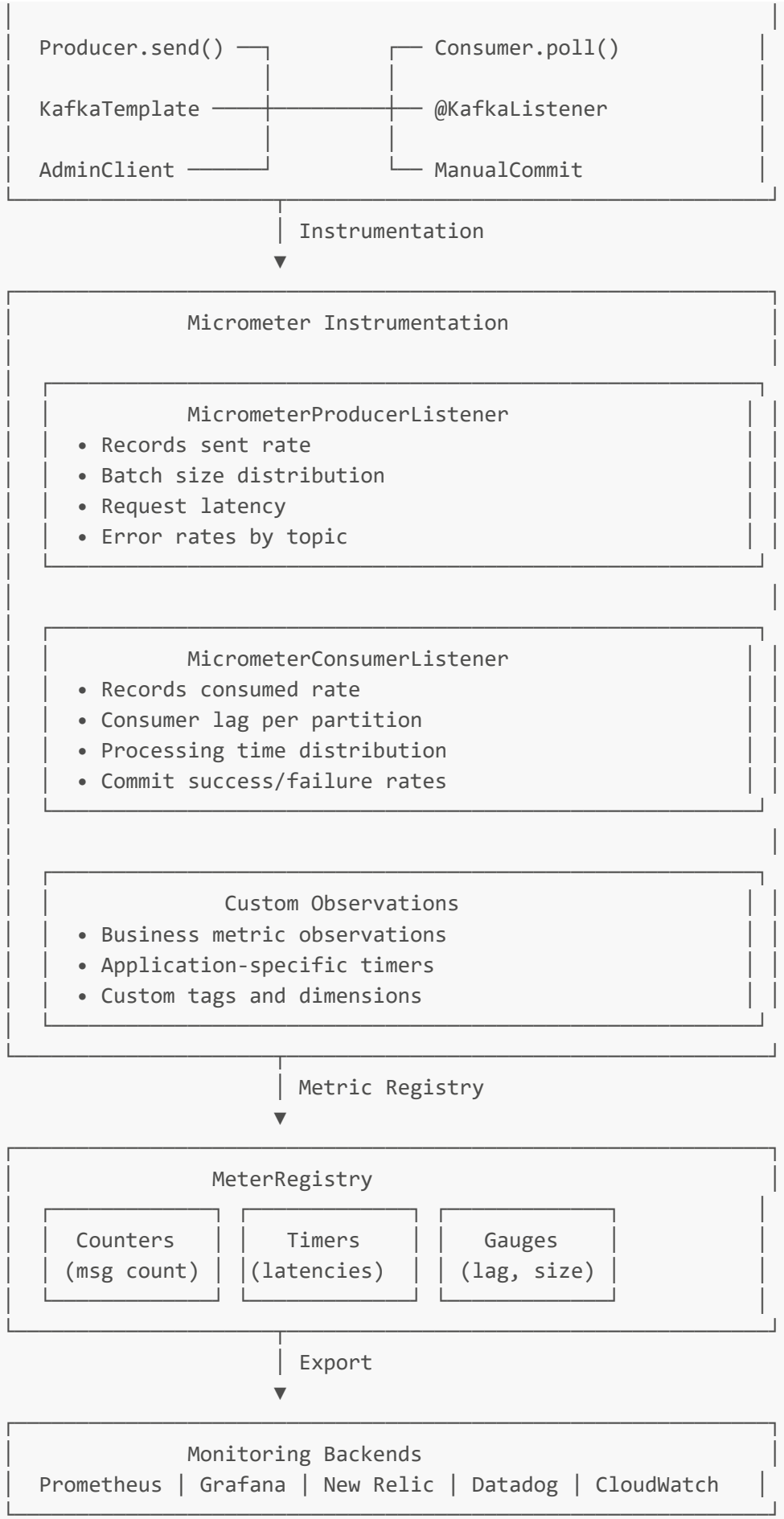
🔍 Micrometer Metrics

Simple Explanation: Micrometer provides a vendor-neutral metrics facade that integrates with Spring Kafka to automatically collect and expose producer/consumer metrics. It captures detailed performance data like message rates, latency distributions, error counts, and throughput metrics that can be exported to monitoring systems like Prometheus, Grafana, or New Relic.

Micrometer Metrics Architecture:

```
Micrometer Metrics Collection Flow:

┌─────────────────────────────────────────────────────────────┐
│                Kafka Operations                             │
```



Producer Metrics

Producer metrics track the performance and behavior of Kafka message producers, including throughput, latency, error rates, and resource utilization.

Complete Producer Metrics Configuration

```
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.binder.kafka.KafkaClientMetrics;
import org.springframework.kafka.core.MicrometerProducerListener;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;

/**
 * Comprehensive producer metrics configuration with Micrometer
 */
@Configuration
@EnableConfigurationProperties(KafkaProperties.class)
@lombok.extern.slf4j.Slf4j
public class KafkaProducerMetricsConfiguration {

    @Autowired
    private MeterRegistry meterRegistry;

    /**
     * Producer factory with comprehensive Micrometer instrumentation
     */
    @Bean
    public ProducerFactory<String, Object>
instrumentedProducerFactory(KafkaProperties properties) {

        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
properties.getBootstrapServers());
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        // Performance configurations that affect metrics
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);

        // Client ID for metrics identification
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "instrumented-producer");

        DefaultKafkaProducerFactory<String, Object> factory = new
DefaultKafkaProducerFactory<>(props);

        // Add Micrometer producer listener for metrics collection
```

```
        factory.addListener(new MicrometerProducerListener<>(meterRegistry,
            Arrays.asList(
                Tag.of("application", "kafka-metrics-demo"),
                Tag.of("environment", "production"),
                Tag.of("component", "kafka-producer")
            )));

        log.info("Configured instrumented producer factory with Micrometer
metrics");

        return factory;
    }

    /**
     * KafkaTemplate with observation capabilities
     */
    @Bean
    public KafkaTemplate<String, Object>
instrumentedKafkaTemplate(ProducerFactory<String, Object> producerFactory) {

        KafkaTemplate<String, Object> template = new KafkaTemplate<>
(producerFactory);

        // Enable observations for detailed metrics
        template.setObservationEnabled(true);

        // Set custom observation convention
        template.setObservationConvention(new
CustomKafkaTemplateObservationConvention());

        // Set producer interceptors for additional metrics
        template.setProducerInterceptors(Arrays.asList(new
MetricsProducerInterceptor()));

        log.info("Configured KafkaTemplate with observations and custom metrics");

        return template;
    }

    /**
     * Custom observation convention for detailed producer metrics
     */
    public static class CustomKafkaTemplateObservationConvention implements
KafkaTemplateObservationConvention {

        @Override
        public KeyValues getLowCardinalityKeyValues(KafkaRecordSenderContext
context) {
            return KeyValues.of(
                "topic", context.getDestination(),
                "partition", String.valueOf(context.getRecord().partition() !=
null ?
                    context.getRecord().partition() : "unassigned"),
                "message.type", determineMessageType(context.getRecord().value()),
```

```
                "client.id", context.getRecord().headers().lastHeader("client-id")
!= null ?
                    new String(context.getRecord().headers().lastHeader("client-
id").value()) : "unknown"
            );
        }

        @Override
        public KeyValues getHighCardinalityKeyValues(KafkaRecordSenderContext
context) {
            return KeyValues.of(
                "record.key", String.valueOf(context.getRecord().key()),
                "record.size",
String.valueOf(estimateRecordSize(context.getRecord())),
                "timestamp", String.valueOf(context.getRecord().timestamp())
            );
        }

        private String determineMessageType(Object value) {
            if (value == null) return "null";
            return value.getClass().getSimpleName();
        }

        private long estimateRecordSize(ProducerRecord<String, Object> record) {
            // Simplified size estimation
            long size = 0;
            if (record.key() != null) size += record.key().length();
            if (record.value() != null) size +=
record.value().toString().length();
            return size;
        }
    }

    /**
     * Custom producer interceptor for additional metrics
     */
    public static class MetricsProducerInterceptor implements
ProducerInterceptor<String, Object> {

        private Counter messagesSent;
        private Counter messagesSuccess;
        private Counter messagesFailure;
        private Timer sendTimer;
        private DistributionSummary recordSize;

        @Override
        public void configure(Map<String, ?> configs) {
            // Initialize metrics (would typically get MeterRegistry from context)
            log.debug("Configuring MetricsProducerInterceptor");
        }

        @Override
        public ProducerRecord<String, Object> onSend(ProducerRecord<String,
Object> record) {
```

```
            // Increment messages sent counter
            if (messagesSent != null) messagesSent.increment();

            // Record message size
            if (recordSize != null) {
                long size = estimateSize(record);
                recordSize.record(size);
            }

            // Add timing information to headers
            record.headers().add("send-timestamp",
                String.valueOf(System.currentTimeMillis()).getBytes());

            return record;
        }

        @Override
        public void onAcknowledgement(RecordMetadata metadata, Exception
exception) {

            if (exception == null) {
                // Success metrics
                if (messagesSuccess != null) messagesSuccess.increment();

                // Record send duration if available
                if (sendTimer != null) {
                    // Implementation would calculate duration from headers
                }

            } else {
                // Failure metrics
                if (messagesFailure != null) messagesFailure.increment();

                log.warn("Producer message failed: partition={}, offset={}, error=
{}",
                    metadata != null ? metadata.partition() : -1,
                    metadata != null ? metadata.offset() : -1,
                    exception.getMessage());
            }
        }

        @Override
        public void close() {
            log.info("Closing MetricsProducerInterceptor");
        }

        private long estimateSize(ProducerRecord<String, Object> record) {
            // Simplified size calculation
            long size = 0;
            if (record.key() != null) size += record.key().getBytes().length;
            if (record.value() != null) size +=
record.value().toString().getBytes().length;
            return size;
```

```
        }
    }
}

/**
 * Producer service with comprehensive metrics
 */
@Service
@lombok.extern.slf4j.Slf4j
public class InstrumentedKafkaProducer {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private MeterRegistry meterRegistry;

    // Custom metrics
    private final Counter businessMessagesProduced;
    private final Timer businessProcessingTimer;
    private final DistributionSummary businessMessageSize;
    private final Gauge activeProducers;

    private final AtomicInteger currentActiveProducers = new AtomicInteger(0);

    public InstrumentedKafkaProducer(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;

        // Initialize custom business metrics
        this.businessMessagesProduced =
Counter.builder("business.messages.produced")
            .description("Number of business messages produced")
            .tag("service", "kafka-producer")
            .register(meterRegistry);

        this.businessProcessingTimer =
Timer.builder("business.message.processing.time")
            .description("Time taken to process and send business messages")
            .register(meterRegistry);

        this.businessMessageSize =
DistributionSummary.builder("business.message.size")
            .description("Size distribution of business messages")
            .baseUnit("bytes")
            .register(meterRegistry);

        this.activeProducers = Gauge.builder("business.active.producers")
            .description("Number of currently active producers")
            .register(meterRegistry, currentActiveProducers, AtomicInteger::get);
    }

    /**
     * Send message with comprehensive metrics tracking
     */
```

```
    public void sendBusinessMessage(String topic, String key, Object message) {

        currentActiveProducers.incrementAndGet();
        Timer.Sample sample = Timer.start(meterRegistry);

        try {
            log.info("Sending business message: topic={}, key={}", topic, key);

            // Record message size
            long messageSize = estimateMessageSize(message);
            businessMessageSize.record(messageSize);

            // Send message with observation
            Observation.createNotStarted("kafka.producer.business.send",
meterRegistry)
                .lowCardinalityKeyValue("topic", topic)
                .lowCardinalityKeyValue("message.type",
message.getClass().getSimpleName())
                .highCardinalityKeyValue("message.key", key)
                .observe(() -> {
                    ListenableFuture<SendResult<String, Object>> future =
                        kafkaTemplate.send(topic, key, message);

                    // Add callback for success/failure handling
                    future.addCallback(
                        result -> handleSendSuccess(result, topic, key),
                        failure -> handleSendFailure(failure, topic, key)
                    );

                    return future;
                });

            // Increment business messages counter
            businessMessagesProduced.increment(
                Tags.of(
                    "topic", topic,
                    "message.type", message.getClass().getSimpleName(),
                    "status", "sent"
                )
            );

        } catch (Exception e) {
            log.error("Error sending business message: topic={}, key={}", topic,
key, e);

            // Record error metrics
            meterRegistry.counter("business.messages.error",
                Tags.of(
                    "topic", topic,
                    "error.type", e.getClass().getSimpleName()
                )).increment();

            throw e;
```

```
        } finally {
            // Record processing time
            sample.stop(businessProcessingTimer);
            currentActiveProducers.decrementAndGet();
        }
    }

    /**
     * Batch send with metrics
     */
    public void sendBusinessMessageBatch(String topic, List<KeyValueMessage>
messages) {

        Timer.Sample batchSample = Timer.start(meterRegistry);

        try {
            log.info("Sending business message batch: topic={}, count={}", topic,
messages.size());

            List<ListenableFuture<SendResult<String, Object>>> futures = new
ArrayList<>();

            for (KeyValueMessage kvMessage : messages) {

                ListenableFuture<SendResult<String, Object>> future =
                    kafkaTemplate.send(topic, kvMessage.getKey(),
kvMessage.getValue());

                futures.add(future);

                // Record individual message metrics

businessMessageSize.record(estimateMessageSize(kvMessage.getValue()));
            }

            // Wait for all messages to complete
            CompletableFuture<Void> allFutures = CompletableFuture.allOf(
                futures.stream()
                    .map(f -> f.completable())
                    .toArray(CompletableFuture[]::new)
            );

            allFutures.whenComplete((result, throwable) -> {
                if (throwable == null) {

                    // Record batch success metrics
                    meterRegistry.counter("business.batch.messages.success",
                        Tags.of("topic", topic, "batch.size",
String.valueOf(messages.size())))
                        .increment(messages.size());

                    log.info("Business message batch sent successfully: topic={},
count={}",
                        topic, messages.size());
```

```
                } else {

                    // Record batch failure metrics
                    meterRegistry.counter("business.batch.messages.failure",
                        Tags.of("topic", topic, "error.type",
throwable.getClass().getSimpleName()))
                        .increment();

                    log.error("Business message batch failed: topic={}, count={}",
                        topic, messages.size(), throwable);
                }
            });

        } finally {
            batchSample.stop(Timer.builder("business.batch.processing.time")
                .tag("topic", topic)
                .register(meterRegistry));
        }
    }

    /**
     * Send message with transaction metrics
     */
    @Transactional("kafkaTransactionManager")
    public void sendTransactionalMessage(String topic, String key, Object message)
{

        Timer.Sample transactionSample = Timer.start(meterRegistry);

        try {
            log.info("Sending transactional business message: topic={}, key={}",
topic, key);

            // Send within transaction
            kafkaTemplate.send(topic, key, message);

            // Record transactional message metrics
            meterRegistry.counter("business.messages.transactional",
                Tags.of(
                    "topic", topic,
                    "transaction.status", "committed"
                )).increment();

            log.info("Transactional message sent successfully: topic={}, key={}",
topic, key);

        } catch (Exception e) {

            // Record transaction failure
            meterRegistry.counter("business.messages.transactional",
                Tags.of(
                    "topic", topic,
                    "transaction.status", "rolled-back",
```

```
                    "error.type", e.getClass().getSimpleName()
                )).increment();

            log.error("Transactional message failed: topic={}, key={}", topic,
key, e);
            throw e;

        } finally {

transactionSample.stop(Timer.builder("business.transaction.processing.time")
                .tag("topic", topic)
                .register(meterRegistry));
        }
    }

    // Helper methods
    private void handleSendSuccess(SendResult<String, Object> result, String
topic, String key) {

        RecordMetadata metadata = result.getRecordMetadata();

        log.debug("Message sent successfully: topic={}, key={}, partition={},
offset={}",
            topic, key, metadata.partition(), metadata.offset());

        // Record success metrics with detailed tags
        meterRegistry.counter("business.messages.send.success",
            Tags.of(
                "topic", topic,
                "partition", String.valueOf(metadata.partition())
            )).increment();

        // Record latency if timestamp is available
        if (metadata.hasTimestamp()) {
            long latency = System.currentTimeMillis() - metadata.timestamp();
            meterRegistry.timer("business.messages.send.latency",
                Tags.of("topic", topic))
                .record(latency, TimeUnit.MILLISECONDS);
        }
    }

    private void handleSendFailure(Throwable failure, String topic, String key) {

        log.error("Message send failed: topic={}, key={}", topic, key, failure);

        // Record failure metrics with error classification
        meterRegistry.counter("business.messages.send.failure",
            Tags.of(
                "topic", topic,
                "error.type", failure.getClass().getSimpleName(),
                "error.retryable", String.valueOf(isRetryableError(failure))
            )).increment();
    }
```

```
    private long estimateMessageSize(Object message) {
        // Simplified size estimation - in production use actual serialization
        if (message == null) return 0;
        return message.toString().getBytes().length;
    }

    private boolean isRetryableError(Throwable error) {
        // Classify errors as retryable or not
        return !(error instanceof SerializationException ||
                 error instanceof RecordTooLargeException ||
                 error instanceof InvalidTopicException);
    }
}

/**
 * Custom producer metrics collector
 */
@Component
@lombok.extern.slf4j.Slf4j
public class ProducerMetricsCollector {

    @Autowired
    private MeterRegistry meterRegistry;

    @Autowired
    private ProducerFactory<String, Object> producerFactory;

    @PostConstruct
    public void initializeKafkaClientMetrics() {

        log.info("Initializing Kafka client metrics collection");

        // Create a producer instance for metrics collection
        try (Producer<String, Object> producer = producerFactory.createProducer())
{

            // Bind Kafka client metrics to Micrometer
            KafkaClientMetrics kafkaClientMetrics = new
KafkaClientMetrics(producer);
            kafkaClientMetrics.bindTo(meterRegistry);

            log.info("Kafka client metrics bound to MeterRegistry");
        }
    }

    /**
     * Collect custom JMX-based metrics
     */
    @Scheduled(fixedRate = 30000) // Every 30 seconds
    public void collectCustomProducerMetrics() {

        try {
            // Collect JMX metrics directly if needed
            MBeanServer server = ManagementFactory.getPlatformMBeanServer();
```

```
            ObjectName producerMetrics = new
ObjectName("kafka.producer:type=producer-metrics,client-id=*");
            Set<ObjectName> objectNames = server.queryNames(producerMetrics,
null);

            for (ObjectName objectName : objectNames) {

                // Extract client ID
                String clientId = objectName.getKeyProperty("client-id");

                // Get record send rate
                Double recordSendRate = (Double) server.getAttribute(objectName,
"record-send-rate");
                if (recordSendRate != null) {
                    Gauge.builder("kafka.producer.record.send.rate")
                        .tag("client.id", clientId)
                        .register(meterRegistry, () -> recordSendRate);
                }

                // Get batch size average
                Double batchSizeAvg = (Double) server.getAttribute(objectName,
"batch-size-avg");
                if (batchSizeAvg != null) {
                    Gauge.builder("kafka.producer.batch.size.avg")
                        .tag("client.id", clientId)
                        .register(meterRegistry, () -> batchSizeAvg);
                }
            }

        } catch (Exception e) {
            log.error("Error collecting custom producer metrics", e);
        }
    }
}

// Supporting data classes
@lombok.Data
@lombok.AllArgsConstructor
@lombok.NoArgsConstructor
class KeyValueMessage {
    private String key;
    private Object value;
}
```

Consumer Metrics

Consumer metrics monitor the performance and health of Kafka message consumers, including consumption rates, processing latency, consumer lag, and error handling.

Complete Consumer Metrics Configuration

```
import io.micrometer.core.instrument.*;
import io.micrometer.core.instrument.binder.kafka.KafkaClientMetrics;
import org.springframework.kafka.core.MicrometerConsumerListener;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.ConsumerSeekAware;

/**
 * Comprehensive consumer metrics configuration with Micrometer
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class KafkaConsumerMetricsConfiguration {

    @Autowired
    private MeterRegistry meterRegistry;

    /**
     * Consumer factory with comprehensive Micrometer instrumentation
     */
    @Bean
    public ConsumerFactory<String, Object>
instrumentedConsumerFactory(KafkaProperties properties) {

        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
properties.getBootstrapServers());
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "instrumented-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);

        // Consumer configurations that affect metrics
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000);

        // Client ID for metrics identification
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, "instrumented-consumer");

        DefaultKafkaConsumerFactory<String, Object> factory = new
DefaultKafkaConsumerFactory<>(props);

        // Add Micrometer consumer listener for metrics collection
        factory.addListener(new MicrometerConsumerListener<>(meterRegistry,
            Arrays.asList(
                Tag.of("application", "kafka-metrics-demo"),
                Tag.of("environment", "production"),
                Tag.of("component", "kafka-consumer")
            )));
```

```
        log.info("Configured instrumented consumer factory with Micrometer
metrics");

        return factory;
    }

    /**
     * Listener container factory with metrics and observation
     */
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
instrumentedKafkaListenerContainerFactory(
            ConsumerFactory<String, Object> consumerFactory) {

        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory);
        factory.setConcurrency(3);

        // Enable observations for detailed metrics
        factory.getContainerProperties().setObservationEnabled(true);

        // Set custom observation convention
        factory.setContainerCustomizer(container -> {
            container.getContainerProperties().setObservationConvention(
                new CustomKafkaListenerObservationConvention());
        });

        // Container properties for metrics
        ContainerProperties containerProps = factory.getContainerProperties();
        containerProps.setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
        containerProps.setSyncCommits(true);
        containerProps.setCommitLogLevel(LogIfLevelEnabled.Level.DEBUG);

        // Add consumer interceptors for additional metrics
        factory.setConsumerInterceptors(Arrays.asList(new
MetricsConsumerInterceptor()));

        // Error handler with metrics
        factory.setCommonErrorHandler(new MetricsErrorHandler());

        log.info("Configured KafkaListenerContainerFactory with observations and
metrics");

        return factory;
    }

    /**
     * Custom observation convention for detailed consumer metrics
     */
    public static class CustomKafkaListenerObservationConvention implements
KafkaListenerObservationConvention {
```

```
        @Override
        public KeyValues getLowCardinalityKeyValues(KafkaRecordReceiverContext
context) {
            return KeyValues.of(
                "topic", context.getSource(),
                "partition", String.valueOf(context.getRecord().partition()),
                "consumer.group", context.getGroupId() != null ?
context.getGroupId() : "unknown",
                "message.type", determineMessageType(context.getRecord().value())
            );
        }

        @Override
        public KeyValues getHighCardinalityKeyValues(KafkaRecordReceiverContext
context) {
            return KeyValues.of(
                "record.key", String.valueOf(context.getRecord().key()),
                "record.offset", String.valueOf(context.getRecord().offset()),
                "record.timestamp",
String.valueOf(context.getRecord().timestamp()),
                "record.size",
String.valueOf(estimateRecordSize(context.getRecord()))
            );
        }

        private String determineMessageType(Object value) {
            if (value == null) return "null";
            return value.getClass().getSimpleName();
        }

        private long estimateRecordSize(ConsumerRecord<String, Object> record) {
            long size = 0;
            if (record.key() != null) size += record.key().length();
            if (record.value() != null) size +=
record.value().toString().length();
            return size;
        }
    }

    /**
     * Custom consumer interceptor for additional metrics
     */
    public static class MetricsConsumerInterceptor implements
ConsumerInterceptor<String, Object> {

        private Counter messagesReceived;
        private Timer processingTime;
        private DistributionSummary recordSize;
        private DistributionSummary consumerLag;

        @Override
        public void configure(Map<String, ?> configs) {
            log.debug("Configuring MetricsConsumerInterceptor");
```

```
        }

        @Override
        public ConsumerRecords<String, Object> onConsume(ConsumerRecords<String,
Object> records) {

            // Record number of messages received
            if (messagesReceived != null) {
                messagesReceived.increment(records.count());
            }

            // Record individual record metrics
            for (ConsumerRecord<String, Object> record : records) {

                // Record size metrics
                if (recordSize != null) {
                    long size = estimateSize(record);
                    recordSize.record(size);
                }

                // Calculate and record consumer lag
                if (consumerLag != null) {
                    long lag = System.currentTimeMillis() - record.timestamp();
                    consumerLag.record(lag);
                }

                // Add processing timestamp to track duration
                record.headers().add("processing-start-timestamp",
                    String.valueOf(System.currentTimeMillis()).getBytes());
            }

            return records;
        }

        @Override
        public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets) {

            log.debug("Consumer interceptor - committed offsets: {}",
offsets.size());

            // Could record commit-related metrics here
            offsets.forEach((tp, offsetMetadata) -> {
                log.debug("Committed: topic={}, partition={}, offset={}",
                    tp.topic(), tp.partition(), offsetMetadata.offset());
            });
        }

        @Override
        public void close() {
            log.info("Closing MetricsConsumerInterceptor");
        }

        private long estimateSize(ConsumerRecord<String, Object> record) {
            long size = 0;
```

```
            if (record.key() != null) size += record.key().getBytes().length;
            if (record.value() != null) size +=
record.value().toString().getBytes().length;
            return size;
        }
    }

    /**
     * Error handler with metrics collection
     */
    public static class MetricsErrorHandler implements CommonErrorHandler {

        @Autowired
        private MeterRegistry meterRegistry;

        @Override
        public boolean handleOne(Exception thrownException, ConsumerRecord<?, ?>
record,
                                 Consumer<?, ?> consumer, MessageListenerContainer
container) {

            // Record error metrics
            if (meterRegistry != null) {
                meterRegistry.counter("kafka.consumer.errors",
                    Tags.of(
                        "topic", record.topic(),
                        "partition", String.valueOf(record.partition()),
                        "error.type", thrownException.getClass().getSimpleName()
                    )).increment();
            }

            log.error("Consumer error handling record: topic={}, partition={},
offset={}, error={}",
                record.topic(), record.partition(), record.offset(),
thrownException.getMessage(), thrownException);

            // Return false to continue with default error handling
            return false;
        }

        @Override
        public void handleOtherException(Exception thrownException, Consumer<?, ?>
consumer,
                                         MessageListenerContainer container, boolean
batchListener) {

            // Record general consumer errors
            if (meterRegistry != null) {
                meterRegistry.counter("kafka.consumer.general.errors",
                    Tags.of(
                        "error.type", thrownException.getClass().getSimpleName(),
                        "batch.listener", String.valueOf(batchListener)
                    )).increment();
            }
```

```
            log.error("General consumer error", thrownException);
        }
    }
}

/**
 * Instrumented Kafka consumer service with comprehensive metrics
 */
@Service
@lombok.extern.slf4j.Slf4j
public class InstrumentedKafkaConsumer implements ConsumerSeekAware {

    @Autowired
    private MeterRegistry meterRegistry;

    // Custom business metrics
    private final Counter businessMessagesConsumed;
    private final Timer businessProcessingTimer;
    private final DistributionSummary businessMessageSize;
    private final Gauge activeConsumers;
    private final DistributionSummary consumerLag;
    private final Counter processingErrors;

    private final AtomicInteger currentActiveConsumers = new AtomicInteger(0);
    private final Map<TopicPartition, Long> lastProcessedOffsets = new
ConcurrentHashMap<>();

    public InstrumentedKafkaConsumer(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;

        // Initialize custom business metrics
        this.businessMessagesConsumed =
Counter.builder("business.messages.consumed")
            .description("Number of business messages consumed")
            .tag("service", "kafka-consumer")
            .register(meterRegistry);

        this.businessProcessingTimer =
Timer.builder("business.message.processing.time")
            .description("Time taken to process business messages")
            .register(meterRegistry);

        this.businessMessageSize =
DistributionSummary.builder("business.consumed.message.size")
            .description("Size distribution of consumed business messages")
            .baseUnit("bytes")
            .register(meterRegistry);

        this.activeConsumers = Gauge.builder("business.active.consumers")
            .description("Number of currently active consumers")
            .register(meterRegistry, currentActiveConsumers, AtomicInteger::get);

        this.consumerLag = DistributionSummary.builder("business.consumer.lag")
```

```
            .description("Consumer lag in milliseconds")
            .baseUnit("milliseconds")
            .register(meterRegistry);

        this.processingErrors = Counter.builder("business.processing.errors")
            .description("Number of message processing errors")
            .register(meterRegistry);
    }

    /**
     * Consume user events with comprehensive metrics
     */
    @KafkaListener(
        topics = "user-events",
        groupId = "instrumented-user-processor",
        containerFactory = "instrumentedKafkaListenerContainerFactory"
    )
    public void consumeUserEvents(@Payload UserEvent userEvent,
                                  @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
                                  @Header(KafkaHeaders.RECEIVED_PARTITION) int
partition,
                                  @Header(KafkaHeaders.OFFSET) long offset,
                                  @Header(KafkaHeaders.RECEIVED_TIMESTAMP) long
timestamp,
                                  Acknowledgment ack) {

        currentActiveConsumers.incrementAndGet();
        Timer.Sample sample = Timer.start(meterRegistry);

        try {
            log.info("Processing user event: userId={}, topic={}, partition={},
offset={}",
                userEvent.getUserId(), topic, partition, offset);

            // Calculate and record consumer lag
            long lag = System.currentTimeMillis() - timestamp;
            consumerLag.record(lag, Tags.of("topic", topic, "partition",
String.valueOf(partition)));

            // Record message size
            long messageSize = estimateMessageSize(userEvent);
            businessMessageSize.record(messageSize, Tags.of("topic", topic,
"message.type", "UserEvent"));

            // Process the message with observation
            Observation.createNotStarted("kafka.consumer.business.process",
meterRegistry)
                .lowCardinalityKeyValue("topic", topic)
                .lowCardinalityKeyValue("message.type", "UserEvent")
                .lowCardinalityKeyValue("partition", String.valueOf(partition))
                .highCardinalityKeyValue("user.id", userEvent.getUserId())
                .highCardinalityKeyValue("offset", String.valueOf(offset))
                .observe(() -> {
                    processUserEvent(userEvent);
```

```
                    return null;
                });

            // Record successful processing
            businessMessagesConsumed.increment(Tags.of(
                "topic", topic,
                "partition", String.valueOf(partition),
                "message.type", "UserEvent",
                "status", "success"
            ));

            // Update last processed offset
            TopicPartition tp = new TopicPartition(topic, partition);
            lastProcessedOffsets.put(tp, offset);

            // Manual acknowledgment
            ack.acknowledge();

            log.debug("User event processed successfully: userId={}, offset={}",
                userEvent.getUserId(), offset);

        } catch (Exception e) {

            log.error("Error processing user event: userId={}, offset={}",
                userEvent.getUserId(), offset, e);

            // Record error metrics
            processingErrors.increment(Tags.of(
                "topic", topic,
                "partition", String.valueOf(partition),
                "error.type", e.getClass().getSimpleName(),
                "message.type", "UserEvent"
            ));

            // Don't acknowledge on error - message will be reprocessed
            throw e;

        } finally {
            sample.stop(businessProcessingTimer.withTags(
                "topic", topic,
                "message.type", "UserEvent"
            ));
            currentActiveConsumers.decrementAndGet();
        }
    }

    /**
     * Batch consumer with metrics
     */
    @KafkaListener(
        topics = "order-events",
        groupId = "instrumented-order-batch-processor",
        containerFactory = "instrumentedKafkaListenerContainerFactory"
    )
```

```
    public void consumeOrderEventsBatch(@Payload List<OrderEvent> orderEvents,
                                        @Header(KafkaHeaders.RECEIVED_TOPIC)
List<String> topics,
                                        @Header(KafkaHeaders.RECEIVED_PARTITION)
List<Integer> partitions,
                                        @Header(KafkaHeaders.OFFSET) List<Long>
offsets,
                                        @Header(KafkaHeaders.RECEIVED_TIMESTAMP)
List<Long> timestamps,
                                        Acknowledgment ack) {

        currentActiveConsumers.incrementAndGet();
        Timer.Sample batchSample = Timer.start(meterRegistry);

        try {
            log.info("Processing order events batch: size={}, topics={}",
                orderEvents.size(),
topics.stream().distinct().collect(Collectors.toList()));

            // Process each message in the batch
            for (int i = 0; i < orderEvents.size(); i++) {
                OrderEvent orderEvent = orderEvents.get(i);
                String topic = topics.get(i);
                Integer partition = partitions.get(i);
                Long offset = offsets.get(i);
                Long timestamp = timestamps.get(i);

                Timer.Sample messagesample = Timer.start(meterRegistry);

                try {
                    // Calculate consumer lag for this message
                    long lag = System.currentTimeMillis() - timestamp;
                    consumerLag.record(lag, Tags.of("topic", topic, "partition",
String.valueOf(partition)));

                    // Record message size
                    long messageSize = estimateMessageSize(orderEvent);
                    businessMessageSize.record(messageSize,
                        Tags.of("topic", topic, "message.type", "OrderEvent"));

                    // Process the order event
                    processOrderEvent(orderEvent);

                    // Record successful processing
                    businessMessagesConsumed.increment(Tags.of(
                        "topic", topic,
                        "partition", String.valueOf(partition),
                        "message.type", "OrderEvent",
                        "processing.mode", "batch",
                        "status", "success"
                    ));

                } catch (Exception e) {
```

```
                    log.error("Error processing order event in batch: orderId={},
offset={}",
                        orderEvent.getOrderId(), offset, e);

                    // Record error for this specific message
                    processingErrors.increment(Tags.of(
                        "topic", topic,
                        "partition", String.valueOf(partition),
                        "error.type", e.getClass().getSimpleName(),
                        "message.type", "OrderEvent",
                        "processing.mode", "batch"
                    ));

                    // Continue processing other messages in batch

                } finally {
                    messageSession.stop(businessProcessingTimer.withTags(
                        "topic", topic,
                        "message.type", "OrderEvent",
                        "processing.mode", "batch"
                    ));
                }
            }

            // Record batch metrics
            meterRegistry.counter("business.batch.processing.success",
                Tags.of(
                    "batch.size", String.valueOf(orderEvents.size()),
                    "topics", String.join(",",
topics.stream().distinct().collect(Collectors.toList()))
                )).increment();

            // Acknowledge entire batch
            ack.acknowledge();

            log.info("Order events batch processed successfully: size={}",
orderEvents.size());

        } catch (Exception e) {

            log.error("Error processing order events batch", e);

            // Record batch failure
            meterRegistry.counter("business.batch.processing.failure",
                Tags.of(
                    "batch.size", String.valueOf(orderEvents.size()),
                    "error.type", e.getClass().getSimpleName()
                )).increment();

            throw e;

        } finally {
            batchSample.stop(Timer.builder("business.batch.processing.time")
                .tag("batch.size", String.valueOf(orderEvents.size()))
```

```
                .register(meterRegistry));
            currentActiveConsumers.decrementAndGet();
        }
    }

    /**
     * Error handling consumer with metrics
     */
    @KafkaListener(
        topics = "error-events",
        groupId = "instrumented-error-processor",
        containerFactory = "instrumentedKafkaListenerContainerFactory"
    )
    public void handleErrorEvents(@Payload String errorMessage,
                                  @Header(KafkaHeaders.RECEIVED_TOPIC) String topic,
                                  @Header(KafkaHeaders.OFFSET) long offset,
                                  @Header(name = "kafka_exception-message", required
= false) String exceptionMessage,
                                  Acknowledgment ack) {

        Timer.Sample sample = Timer.start(meterRegistry);

        try {
            log.info("Processing error event: topic={}, offset={}, error={}",
                topic, offset, exceptionMessage);

            // Process error recovery logic
            processErrorRecovery(errorMessage, exceptionMessage);

            // Record error processing metrics
            meterRegistry.counter("business.error.recovery.success",
                Tags.of("topic", topic)).increment();

            ack.acknowledge();

        } catch (Exception e) {

            log.error("Error processing error event: offset={}", offset, e);

            // Record error processing failure
            meterRegistry.counter("business.error.recovery.failure",
                Tags.of(
                    "topic", topic,
                    "error.type", e.getClass().getSimpleName()
                )).increment();

            throw e;

        } finally {
            sample.stop(Timer.builder("business.error.processing.time")
                .tag("topic", topic)
                .register(meterRegistry));
        }
    }
```

```
    /**
     * ConsumerSeekAware implementation for partition management metrics
     */
    @Override
    public void onPartitionsAssigned(Map<TopicPartition, Long> assignments,
ConsumerSeekCallback callback) {

        log.info("Partitions assigned: {}", assignments);

        // Record partition assignment metrics
        assignments.forEach((tp, offset) -> {
            meterRegistry.gauge("kafka.consumer.assigned.partitions",
                Tags.of(
                    "topic", tp.topic(),
                    "partition", String.valueOf(tp.partition())
                ), 1.0);
        });
    }

    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {

        log.info("Partitions revoked: {}", partitions);

        // Remove partition assignment metrics
        partitions.forEach(tp -> {

meterRegistry.remove(Meter.Id.builder("kafka.consumer.assigned.partitions")
                .tag("topic", tp.topic())
                .tag("partition", String.valueOf(tp.partition()))
                .build());
        });
    }

    // Helper methods
    private void processUserEvent(UserEvent userEvent) {

        log.debug("Processing user event business logic: userId={}",
userEvent.getUserId());

        // Simulate processing time
        try {
            Thread.sleep(50 + (long) (Math.random() * 100)); // 50-150ms
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void processOrderEvent(OrderEvent orderEvent) {

        log.debug("Processing order event business logic: orderId={}",
orderEvent.getOrderId());
```

```
        // Simulate processing time
        try {
            Thread.sleep(30 + (long) (Math.random() * 70)); // 30-100ms
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void processErrorRecovery(String errorMessage, String
exceptionMessage) {

        log.debug("Processing error recovery: error={}", errorMessage);

        // Simulate error recovery processing
        try {
            Thread.sleep(100 + (long) (Math.random() * 200)); // 100-300ms
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private long estimateMessageSize(Object message) {
        if (message == null) return 0;
        return message.toString().getBytes().length;
    }
}

/**
 * Consumer lag monitoring service
 */
@Component
@lombok.extern.slf4j.Slf4j
public class ConsumerLagMonitor {

    @Autowired
    private AdminClient adminClient;

    @Autowired
    private MeterRegistry meterRegistry;

    private final Map<String, Map<TopicPartition, Long>> consumerGroupOffsets =
new ConcurrentHashMap<>();

    @Scheduled(fixedRate = 30000) // Every 30 seconds
    public void monitorConsumerLag() {

        try {
            // Get all consumer groups
            ListConsumerGroupsResult consumerGroupsResult =
adminClient.listConsumerGroups();
            Collection<ConsumerGroupListing> consumerGroups =
                consumerGroupsResult.all().get(10, TimeUnit.SECONDS);

            for (ConsumerGroupListing group : consumerGroups) {
```

```
                monitorConsumerGroupLag(group.groupId());
            }

        } catch (Exception e) {
            log.error("Error monitoring consumer lag", e);
        }
    }

    private void monitorConsumerGroupLag(String groupId) {

        try {
            // Get consumer group offsets
            ListConsumerGroupOffsetsResult offsetsResult =
                adminClient.listConsumerGroupOffsets(groupId);
            Map<TopicPartition, OffsetAndMetadata> offsets =
                offsetsResult.partitionsToOffsetAndMetadata().get(10,
TimeUnit.SECONDS);

            if (offsets.isEmpty()) return;

            // Get latest offsets for the topics
            Set<TopicPartition> topicPartitions = offsets.keySet();
            DescribeTopicsResult topicsResult = adminClient.describeTopics(

topicPartitions.stream().map(TopicPartition::topic).collect(Collectors.toSet()));

            // Calculate and record lag for each partition
            offsets.forEach((tp, offsetMetadata) -> {
                try {
                    // Get high water mark (latest offset)
                    // This is a simplified approach - in production use
ListOffsetsResult
                    long currentOffset = offsetMetadata.offset();

                    // Record consumer lag metric
                    meterRegistry.gauge("kafka.consumer.lag",
                        Tags.of(
                            "group", groupId,
                            "topic", tp.topic(),
                            "partition", String.valueOf(tp.partition())
                        ), currentOffset); // Simplified - should be
(highWaterMark - currentOffset)

                } catch (Exception e) {
                    log.error("Error calculating lag for partition: {}", tp, e);
                }
            });

        } catch (Exception e) {
            log.error("Error monitoring consumer group lag: {}", groupId, e);
        }
    }
}
```

This completes the Producer and Consumer Metrics sections of the Spring Kafka Monitoring & Observability guide. The guide continues with Health Checks and Distributed Tracing in the next part.