Spring Kafka Batch Processing: Part 2 - Error Handling & Production Use Cases

Continuation of the comprehensive guide covering batch error handling, production use cases, best practices, and performance optimizations.

🚨 Error Handling in Batches

Simple Explanation: Batch error handling in Spring Kafka is more complex than single record processing because when one message in a batch fails, you need to decide how to handle the entire batch. Spring Kafka provides specialized error handlers for batch processing scenarios.

Why Batch Error Handling is Different:

- **Partial Batch Failures**: Only some records in batch may fail
- **Offset Management**: Need to commit successful records, retry failed ones
- **Batch Atomicity**: Decide whether to process batch as atomic unit or individually
- **Performance Impact**: Error handling shouldn't negate batch processing benefits
- **Recovery Strategies**: Different approaches for different failure types

Batch Error Handling Architecture:

```
Batch Error Handling Flow:

┌─────────────────────────────────────────────────────────────┐
│                    Batch Processing                         │
│  [Msg1✓][Msg2✓][Msg3❌][Msg4][Msg5][Msg6][Msg7][Msg8]    │
│  Success Success FAIL  Pending...                          │
└─────────────────────────────────────────────────────────────┘
                      │
                      │ BatchListenerFailedException
                      ▼
┌─────────────────────────────────────────────────────────────┐
│              DefaultErrorHandler                            │
│  ┌─────────────────────────────────────────────────────────┤
│  │ 1. Commit successful records (Msg1, Msg2)             │
│  │ 2. Seek to failed record position (Msg3)              │
│  │ 3. Create new batch starting from Msg3                │
│  │ 4. Apply retry logic with backoff                     │
│  └─────────────────────────────────────────────────────────┤
└─────────────────────────────────────────────────────────────┘
                      │
                      │ After retries exhausted
                      ▼
┌─────────────────────────────────────────────────────────────┐
│                 Recovery Handler                            │
│  ┌─────────────────┐  ┌─────────────────┐               │
│  │ Send failed     │  │ Skip failed     │               │
│  │ records to DLT  │  │ record & continue│               │
│  └──────────────────┘  └──────────────────┘                  │
└──────────────────────────────────────────────────────────────┘
```

Advanced Batch Error Handling Configuration

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.listener.DefaultErrorHandler;
import org.springframework.kafka.listener.DeadLetterPublishingRecoverer;
import org.springframework.kafka.listener.BatchListenerFailedException;
import org.springframework.kafka.support.ExponentialBackOffWithMaxRetries;

import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.common.TopicPartition;

/**
 * Comprehensive batch error handling configuration
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class BatchErrorHandlerConfiguration {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    private MeterRegistry meterRegistry;

    /**
     * Basic batch error handler with retry and recovery
     */
    @Bean("basicBatchErrorHandler")
    public CommonErrorHandler basicBatchErrorHandler() {

        // DLT recoverer for failed batch records
        DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(kafkaTemplate);

        // Exponential backoff for batch retries
        ExponentialBackOffWithMaxRetries backOff = new
ExponentialBackOffWithMaxRetries(5);
        backOff.setInitialInterval(2000L);  // 2 seconds
        backOff.setMultiplier(2.0);         // Double each time
        backOff.setMaxInterval(30000L);     // Cap at 30 seconds

        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

        // Configure batch-specific exception handling
        configureBatchExceptions(errorHandler);

        // Add retry listeners for batch monitoring
        errorHandler.setRetryListeners(createBatchRetryListener("BasicBatch"));

        log.info("Configured basic batch error handler: maxRetries=5, exponential
backoff");

        return errorHandler;
    }

    /**
     * Advanced batch error handler with partial failure handling
     */
    @Bean("advancedBatchErrorHandler")
    public CommonErrorHandler advancedBatchErrorHandler() {

        // Custom recoverer for batch-aware recovery
        ConsumerRecordRecoverer batchAwareRecoverer = new
BatchAwareRecoverer(kafkaTemplate);

        // Sophisticated backoff strategy
        DefaultErrorHandler errorHandler = new
DefaultErrorHandler(batchAwareRecoverer);

        // Custom backoff function for batch processing
        BiFunction<ConsumerRecord<?, ?>, Exception, BackOff> batchBackoffFunction
=
            (record, exception) -> {

                if (exception instanceof BatchListenerFailedException) {
                    BatchListenerFailedException batchEx =
(BatchListenerFailedException) exception;
                    int failedIndex = batchEx.getIndex();

                    log.debug("Batch failure at index {}, applying targeted
backoff", failedIndex);

                    // Short backoff for early failures (likely transient)
                    if (failedIndex < 10) {
                        return new FixedBackOff(1000L, 3L);
                    } else {
                        return new ExponentialBackOff(2000L, 1.5);
                    }
                } else {
                    // Standard exponential backoff for non-batch exceptions
                    return new ExponentialBackOff(3000L, 2.0);
                }
            };

        errorHandler.setBackOffFunction(batchBackoffFunction);

        configureBatchExceptions(errorHandler);
        errorHandler.setRetryListeners(createBatchRetryListener("AdvancedBatch"));
        log.info("Configured advanced batch error handler with partial failure
handling");

        return errorHandler;
    }

    /**
     * High-throughput batch error handler optimized for performance
     */
    @Bean("highThroughputBatchErrorHandler")
    public CommonErrorHandler highThroughputBatchErrorHandler() {

        // Fast-fail recoverer for high-throughput scenarios
        ConsumerRecordRecoverer fastFailRecoverer = new
FastFailBatchRecoverer(kafkaTemplate);

        // Minimal retry for high throughput
        FixedBackOff minimalBackoff = new FixedBackOff(500L, 2L); // Only 2
retries

        DefaultErrorHandler errorHandler = new
DefaultErrorHandler(fastFailRecoverer, minimalBackoff);

        // Aggressive exception classification for fast processing
        errorHandler.addNotRetryableExceptions(
            // Don't retry parsing errors in high-throughput scenarios
            com.fasterxml.jackson.core.JsonParseException.class,
            IllegalArgumentException.class,
            ValidationException.class
        );


errorHandler.setRetryListeners(createBatchRetryListener("HighThroughput"));

        log.info("Configured high-throughput batch error handler: minimal retries,
fast-fail");

        return errorHandler;
    }

    /**
     * Container factories with different batch error handlers
     */
    @Bean("basicBatchContainerFactory")
    public ConcurrentKafkaListenerContainerFactory<String, String>
basicBatchContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            createBaseBatchFactory();

        factory.setCommonErrorHandler(basicBatchErrorHandler());

        return factory;
    }
    @Bean("advancedBatchContainerFactory")
    public ConcurrentKafkaListenerContainerFactory<String, String>
advancedBatchContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            createBaseBatchFactory();

        factory.setCommonErrorHandler(advancedBatchErrorHandler());

        return factory;
    }

    @Bean("highThroughputBatchContainerFactory")
    public ConcurrentKafkaListenerContainerFactory<String, String>
highThroughputBatchContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            createBaseBatchFactory();

        factory.setCommonErrorHandler(highThroughputBatchErrorHandler());

        return factory;
    }

    // Helper methods
    private ConcurrentKafkaListenerContainerFactory<String, String>
createBaseBatchFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        // Use batch consumer factory
        factory.setConsumerFactory(createBatchConsumerFactory());
        factory.setBatchListener(true);

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);

        return factory;
    }

    private ConsumerFactory<String, String> createBatchConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        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, "batch-error-handler-group");
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1000);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    private void configureBatchExceptions(DefaultErrorHandler errorHandler) {
        // Fatal exceptions for batch processing
        errorHandler.addNotRetryableExceptions(

org.springframework.kafka.support.serializer.DeserializationException.class,

org.springframework.messaging.converter.MessageConversionException.class,
            IllegalArgumentException.class,
            SecurityException.class
        );

        // Retryable exceptions
        errorHandler.addRetryableExceptions(
            BatchProcessingException.class,
            ExternalServiceException.class,
            java.util.concurrent.TimeoutException.class,
            org.springframework.dao.TransientDataAccessException.class
        );
    }

    private RetryListener createBatchRetryListener(String handlerName) {
        return new BatchRetryListener(handlerName, meterRegistry);
    }
}

/**
 * Batch-aware recoverer that handles partial failures intelligently
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BatchAwareRecoverer implements ConsumerRecordRecoverer {

    private final KafkaTemplate<String, Object> kafkaTemplate;
    private final MeterRegistry meterRegistry;

    public BatchAwareRecoverer(KafkaTemplate<String, Object> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
        this.meterRegistry = Metrics.globalRegistry;
    }

    @Override
    public void accept(ConsumerRecord<?, ?> record, Exception exception) {
        log.info("BatchAware recovery for record: topic={}, partition={}, offset=
{}",
            record.topic(), record.partition(), record.offset());

        try {
            if (exception instanceof BatchListenerFailedException) {
                handleBatchFailure(record, (BatchListenerFailedException)
exception);
            } else {
                handleGenericFailure(record, exception);
            }

        } catch (Exception e) {
            log.error("Recovery failed for record: topic={}, offset={}",
                record.topic(), record.offset(), e);

            // Final fallback - send to poison pill topic
            sendToPoisonPillTopic(record, exception);
        }
    }

    private void handleBatchFailure(ConsumerRecord<?, ?> record,
BatchListenerFailedException batchException) {
        int failedIndex = batchException.getIndex();
        Throwable rootCause = batchException.getCause();

        log.info("Handling batch failure: failedIndex={}, rootCause={}",
            failedIndex, rootCause.getClass().getSimpleName());

        // Analyze failure type and route accordingly
        if (rootCause instanceof ValidationException) {
            sendToValidationDlt(record, batchException, failedIndex);

        } else if (rootCause instanceof ExternalServiceException) {
            scheduleRetryWithDelay(record, batchException, Duration.ofMinutes(5));

        } else if (isBusinessLogicFailure(rootCause)) {
            sendToBusinessLogicDlt(record, batchException, failedIndex);

        } else {
            sendToGenericDlt(record, batchException, failedIndex);
        }

        // Update batch failure metrics
        updateBatchFailureMetrics(record.topic(), failedIndex, rootCause);
    }

    private void handleGenericFailure(ConsumerRecord<?, ?> record, Exception
exception) {
        log.info("Handling generic failure: exception={}",
exception.getClass().getSimpleName());

        // Standard DLT routing for non-batch failures
        String dltTopic = record.topic() + ".DLT";

        ProducerRecord<Object, Object> dltRecord = new ProducerRecord<>(
            dltTopic, record.key(), record.value());

        // Add failure metadata
        dltRecord.headers().add("failure-type", "generic".getBytes());
        dltRecord.headers().add("original-exception",
exception.getClass().getName().getBytes());
        dltRecord.headers().add("failure-timestamp",
String.valueOf(System.currentTimeMillis()).getBytes());

        kafkaTemplate.send(dltRecord);
        log.info("Sent failed record to generic DLT: {}", dltTopic);
    }

    private void sendToValidationDlt(ConsumerRecord<?, ?> record,
                                     BatchListenerFailedException batchException,
int failedIndex) {
        String dltTopic = record.topic() + ".validation.DLT";

        ProducerRecord<Object, Object> dltRecord = new ProducerRecord<>(
            dltTopic, record.key(), record.value());

        // Add batch-specific headers
        addBatchFailureHeaders(dltRecord, batchException, failedIndex);
        dltRecord.headers().add("validation-failure", "true".getBytes());

        kafkaTemplate.send(dltRecord);

        log.info("Sent batch validation failure to DLT: topic={}, index={}",
dltTopic, failedIndex);
    }

    private void sendToBusinessLogicDlt(ConsumerRecord<?, ?> record,
                                        BatchListenerFailedException batchException,
int failedIndex) {
        String dltTopic = record.topic() + ".business.DLT";

        ProducerRecord<Object, Object> dltRecord = new ProducerRecord<>(
            dltTopic, record.key(), record.value());

        addBatchFailureHeaders(dltRecord, batchException, failedIndex);
        dltRecord.headers().add("business-logic-failure", "true".getBytes());
        dltRecord.headers().add("requires-manual-review", "true".getBytes());

        kafkaTemplate.send(dltRecord);

        log.info("Sent batch business logic failure to DLT: topic={}, index={}",
dltTopic, failedIndex);
    }

    private void sendToGenericDlt(ConsumerRecord<?, ?> record,
                                  BatchListenerFailedException batchException, int
failedIndex) {
        String dltTopic = record.topic() + ".batch.DLT";

        ProducerRecord<Object, Object> dltRecord = new ProducerRecord<>(
            dltTopic, record.key(), record.value());

        addBatchFailureHeaders(dltRecord, batchException, failedIndex);

        kafkaTemplate.send(dltRecord);

        log.info("Sent batch failure to generic DLT: topic={}, index={}",
dltTopic, failedIndex);
    }
    private void scheduleRetryWithDelay(ConsumerRecord<?, ?> record,
                                        BatchListenerFailedException batchException,
Duration delay) {
        String retryTopic = record.topic() + ".retry";

        ProducerRecord<Object, Object> retryRecord = new ProducerRecord<>(
            retryTopic, record.key(), record.value());

        // Schedule for future processing
        long retryTime = System.currentTimeMillis() + delay.toMillis();
        retryRecord.headers().add("scheduled-retry-time",
String.valueOf(retryTime).getBytes());
        retryRecord.headers().add("retry-reason", "external-service-
failure".getBytes());
        addBatchFailureHeaders(retryRecord, batchException,
batchException.getIndex());

        kafkaTemplate.send(retryRecord);

        log.info("Scheduled batch record for retry: topic={}, delay={}min",
            retryTopic, delay.toMinutes());
    }

    private void sendToPoisonPillTopic(ConsumerRecord<?, ?> record, Exception
exception) {
        String poisonTopic = record.topic() + ".poison";

        ProducerRecord<Object, Object> poisonRecord = new ProducerRecord<>(
            poisonTopic, record.key(), record.value());

        poisonRecord.headers().add("poison-reason", "recovery-failed".getBytes());
        poisonRecord.headers().add("original-exception",
exception.getClass().getName().getBytes());
        poisonRecord.headers().add("requires-investigation", "true".getBytes());

        kafkaTemplate.send(poisonRecord);

        log.error("Sent record to poison pill topic: {}", poisonTopic);
    }

    private void addBatchFailureHeaders(ProducerRecord<Object, Object> record,
                                        BatchListenerFailedException batchException,
int failedIndex) {
        record.headers().add("batch-failure", "true".getBytes());
        record.headers().add("failed-index",
String.valueOf(failedIndex).getBytes());
        record.headers().add("batch-exception",
batchException.getClass().getName().getBytes());
        record.headers().add("root-cause",
batchException.getCause().getClass().getName().getBytes());
        record.headers().add("failure-timestamp",
String.valueOf(System.currentTimeMillis()).getBytes());
    }
    private boolean isBusinessLogicFailure(Throwable exception) {
        return exception instanceof BusinessRuleException ||
               exception instanceof WorkflowException ||
               exception instanceof ValidationException;
    }

    private void updateBatchFailureMetrics(String topic, int failedIndex,
Throwable rootCause) {
        meterRegistry.counter("kafka.batch.failure",
            Tags.of(
                "topic", topic,
                "exception", rootCause.getClass().getSimpleName(),
                "index_range", getIndexRange(failedIndex)
            )).increment();
    }

    private String getIndexRange(int index) {
        if (index < 10) return "0-9";
        else if (index < 100) return "10-99";
        else if (index < 1000) return "100-999";
        else return "1000+";
    }
}

/**
 * Fast-fail recoverer for high-throughput batch processing
 */
@Component
@lombok.extern.slf4j.Slf4j
public class FastFailBatchRecoverer implements ConsumerRecordRecoverer {

    private final KafkaTemplate<String, Object> kafkaTemplate;

    public FastFailBatchRecoverer(KafkaTemplate<String, Object> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    @Override
    public void accept(ConsumerRecord<?, ?> record, Exception exception) {
        // Fast-fail approach - minimal processing overhead
        String dltTopic = record.topic() + ".fast.DLT";

        // Create minimal DLT record
        ProducerRecord<Object, Object> dltRecord = new ProducerRecord<>(
            dltTopic, record.key(), record.value());

        // Add essential headers only
        dltRecord.headers().add("fast-fail", "true".getBytes());
        dltRecord.headers().add("exception",
exception.getClass().getSimpleName().getBytes());

        // Send asynchronously for maximum throughput
        kafkaTemplate.send(dltRecord).whenComplete((result, ex) -> {
            if (ex != null) {
                log.error("Failed to send to fast DLT: topic={}, offset={}",
                    record.topic(), record.offset(), ex);
            }
        });

        log.debug("Fast-fail recovery: topic={}, offset={}", record.topic(),
record.offset());
    }
}

/**
 * Batch retry listener for monitoring and metrics
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BatchRetryListener implements RetryListener {

    private final String handlerName;
    private final MeterRegistry meterRegistry;

    public BatchRetryListener(String handlerName, MeterRegistry meterRegistry) {
        this.handlerName = handlerName;
        this.meterRegistry = meterRegistry;
    }

    @Override
    public void failedDelivery(ConsumerRecord<?, ?> record, Exception ex, int
deliveryAttempt) {
        log.warn("[{}] Batch delivery attempt {} failed: topic={}, partition={},
offset={}, error={}",
            handlerName, deliveryAttempt, record.topic(), record.partition(),
            record.offset(), ex.getMessage());

        // Track batch retry metrics
        String exceptionType = ex instanceof BatchListenerFailedException ?
            "batch_failure" : ex.getClass().getSimpleName();

        meterRegistry.counter("kafka.batch.retry.attempts",
            Tags.of(
                "handler", handlerName,
                "topic", record.topic(),
                "exception", exceptionType,
                "attempt", String.valueOf(deliveryAttempt)
            )).increment();

        // Alert on high retry attempts for batch processing
        if (deliveryAttempt >= 4) {
            alertHighBatchRetries(record, ex, deliveryAttempt);
        }
    }

    @Override
    public void recovered(ConsumerRecord<?, ?> record, Exception ex) {
        log.info("[{}] Batch record recovered after retries: topic={}, partition={}, offset={}",
            handlerName, record.topic(), record.partition(), record.offset());

        meterRegistry.counter("kafka.batch.recovered",
            Tags.of("handler", handlerName, "topic", record.topic()))
            .increment();
    }

    @Override
    public void recoveryFailed(ConsumerRecord<?, ?> record, Exception original, Exception failure) {
        log.error("[{}] Batch recovery failed: topic={}, partition={}, offset={}, original={}, recovery={}",
            handlerName, record.topic(), record.partition(), record.offset(),
            original.getMessage(), failure.getMessage());

        String exceptionType = original instanceof BatchListenerFailedException ?
            "batch_failure" : original.getClass().getSimpleName();

        meterRegistry.counter("kafka.batch.recovery.failed",
            Tags.of(
                "handler", handlerName,
                "topic", record.topic(),
                "exception", exceptionType
            )).increment();

        // Alert on batch recovery failures
        alertBatchRecoveryFailure(record, original, failure);
    }

    private void alertHighBatchRetries(ConsumerRecord<?, ?> record, Exception ex, int attempt) {
        log.error("🚨 HIGH BATCH RETRY COUNT: handler={}, topic={}, attempt={}, error={}",
            handlerName, record.topic(), attempt, ex.getMessage());
    }

    private void alertBatchRecoveryFailure(ConsumerRecord<?, ?> record, Exception original, Exception failure) {
        log.error("🚨 BATCH RECOVERY FAILURE: handler={}, topic={}, originalError={}, recoveryError={}",
            handlerName, record.topic(), original.getMessage(), failure.getMessage());
    }
}

/**
 * Batch processing examples with error handling
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BatchErrorHandlingExamples {
    /**
     * Batch listener with explicit failure handling
     */
    @KafkaListener(
        topics = "error-prone-batch",
        groupId = "batch-error-group",
        containerFactory = "advancedBatchContainerFactory"
    )
    public void processBatchWithErrorHandling(@Payload List<String> messages,

@Header(KafkaHeaders.RECEIVED_PARTITION) List<Integer> partitions,
                                              @Header(KafkaHeaders.OFFSET)
List<Long> offsets,
                                              Acknowledgment ack) {

        log.info("Processing batch with error handling: size={}",
messages.size());

        try {
            // Process each message in batch
            for (int i = 0; i < messages.size(); i++) {
                String message = messages.get(i);

                try {
                    processMessageWithValidation(message);

                } catch (ValidationException e) {
                    log.error("Validation error at index {}: {}", i,
e.getMessage());

                    // Throw BatchListenerFailedException to indicate specific
failure
                    throw new BatchListenerFailedException("Validation failed", e,
i);

                } catch (ExternalServiceException e) {
                    log.error("External service error at index {}: {}", i,
e.getMessage());

                    throw new BatchListenerFailedException("External service
failure", e, i);
                }
            }

            // Acknowledge successful batch processing
            ack.acknowledge();

            log.info("Batch processing completed successfully: {} messages",
messages.size());

        } catch (BatchListenerFailedException e) {
            log.error("Batch processing failed at index {}: {}", e.getIndex(),
e.getMessage());
            throw e; // Re-throw to trigger error handler
        } catch (Exception e) {
            log.error("Unexpected error in batch processing", e);
            throw e;
        }
    }

    /**
     * Batch listener with partial failure tolerance
     */
    @KafkaListener(
        topics = "partially-tolerant-batch",
        groupId = "partial-tolerance-group",
        containerFactory = "basicBatchContainerFactory"
    )
    public void processPartiallyTolerantBatch(@Payload List<String> messages,
                                              Acknowledgment ack) {

        log.info("Processing partially tolerant batch: size={}", messages.size());

        int successCount = 0;
        int failureCount = 0;
        List<BatchProcessingError> errors = new ArrayList<>();

        // Process all messages, collecting failures
        for (int i = 0; i < messages.size(); i++) {
            String message = messages.get(i);

            try {
                processMessageWithPartialTolerance(message);
                successCount++;

            } catch (Exception e) {
                log.warn("Failed to process message at index {}: {}", i,
e.getMessage());

                failureCount++;
                errors.add(new BatchProcessingError(i, message, e));

                // Continue processing other messages
            }
        }

        log.info("Batch processing completed: success={}, failures={}",
successCount, failureCount);

        // Handle failures if any
        if (!errors.isEmpty()) {
            handlePartialFailures(errors, messages.size());

            // Decision: acknowledge batch despite partial failures
            // Failed messages are handled separately
            ack.acknowledge();
        } else {
            ack.acknowledge();
        }
    }

    /**
     * High-throughput batch with minimal error handling
     */
    @KafkaListener(
        topics = "high-throughput-logs",
        groupId = "high-throughput-group",
        containerFactory = "highThroughputBatchContainerFactory"
    )
    public void processHighThroughputBatch(@Payload List<String> logs) {

        log.debug("Processing high-throughput batch: size={}", logs.size());

        // Fast processing with minimal error handling overhead
        logs.parallelStream().forEach(log -> {
            try {
                processLogFast(log);
            } catch (Exception e) {
                // Log and continue - don't let single failure stop batch
                log.debug("Failed to process log: {}", e.getMessage());
            }
        });
    }

    // Business logic methods
    private void processMessageWithValidation(String message) throws
ValidationException {
        if (message == null || message.trim().isEmpty()) {
            throw new ValidationException("Message cannot be null or empty");
        }

        if (message.contains("INVALID")) {
            throw new ValidationException("Message contains invalid content");
        }

        // Process valid message
        log.trace("Processed valid message: {}", message);
    }

    private void processMessageWithPartialTolerance(String message) throws
Exception {
        // Simulate processing that might fail
        if (message.contains("FAIL")) {
            throw new ProcessingException("Simulated processing failure");
        }

        log.trace("Processed message with partial tolerance: {}", message);
    }

    private void processLogFast(String log) {
        // Optimized log processing
        log.trace("Fast processed log: {}", log);
    }

    private void handlePartialFailures(List<BatchProcessingError> errors, int
totalMessages) {
        log.info("Handling {} partial failures out of {} total messages",
errors.size(), totalMessages);

        // Send failed messages to error topic for later processing
        errors.forEach(error -> {
            try {
                sendToErrorTopic(error);
            } catch (Exception e) {
                log.error("Failed to send error to error topic: index={}",
error.getIndex(), e);
            }
        });
    }

    private void sendToErrorTopic(BatchProcessingError error) {
        // Implementation to send failed message to error topic
        log.debug("Sending partial failure to error topic: index={}",
error.getIndex());
    }
}

// Supporting classes
@lombok.Data
@lombok.AllArgsConstructor
class BatchProcessingError {
    private int index;
    private String message;
    private Exception exception;
}

class BatchProcessingException extends Exception {
    public BatchProcessingException(String message) { super(message); }
    public BatchProcessingException(String message, Throwable cause) {
super(message, cause); }
}

class ProcessingException extends Exception {
    public ProcessingException(String message) { super(message); }
}
```

🌍 Use Cases (Log Aggregation, ETL, Analytics)

Real-World Batch Processing Applications

```
/**
 * Log Aggregation System using Batch Processing
 */
@Service
@lombok.extern.slf4j.Slf4j
public class LogAggregationService {

    @Autowired
    private LogAnalyticsRepository logRepository;

    @Autowired
    private AlertingService alertingService;

    /**
     * High-volume log aggregation with batch processing
     */
    @KafkaListener(
        topics = "application-logs",
        groupId = "log-aggregation-group",
        containerFactory = "highThroughputBatchContainerFactory"
    )
    public void aggregateApplicationLogs(
            @Payload List<String> logMessages,
            @Header(KafkaHeaders.RECEIVED_TIMESTAMP) List<Long> timestamps,
            @Header(KafkaHeaders.RECEIVED_PARTITION) List<Integer> partitions) {

        long startTime = System.currentTimeMillis();

        log.info("Aggregating application logs batch: size={}",
logMessages.size());

        try {
            // Parse log messages in batch
            List<LogEntry> logEntries = logMessages.parallelStream()
                .map(this::parseLogMessage)
                .filter(Objects::nonNull)
                .collect(Collectors.toList());

            // Group logs by time windows for aggregation
            Map<String, List<LogEntry>> logsByTimeWindow =
groupLogsByTimeWindow(logEntries);

            // Process each time window
            for (Map.Entry<String, List<LogEntry>> entry :
logsByTimeWindow.entrySet()) {
                String timeWindow = entry.getKey();
                List<LogEntry> windowLogs = entry.getValue();

                processLogTimeWindow(timeWindow, windowLogs);
            }

            // Performance metrics
            long processingTime = System.currentTimeMillis() - startTime;
            double throughput = logMessages.size() / (processingTime / 1000.0);

            log.info("Log aggregation completed: {} logs in {}ms, throughput: 
{:.2f} logs/sec",
                logMessages.size(), processingTime, throughput);

        } catch (Exception e) {
            log.error("Error in log aggregation batch processing", e);
            throw e;
        }
    }

    /**
     * System metrics aggregation from multiple services
     */
    @KafkaListener(
        topics = "system-metrics",
        groupId = "metrics-aggregation-group",
        containerFactory = "analyticsBatchContainerFactory"
    )
    public void aggregateSystemMetrics(@Payload List<String> metricMessages) {

        log.info("Aggregating system metrics batch: size={}", 
metricMessages.size());

        // Parse metrics
        List<MetricData> metrics = metricMessages.stream()
            .map(this::parseMetricMessage)
            .filter(Objects::nonNull)
            .collect(Collectors.toList());

        // Group by service and metric type
        Map<String, Map<String, List<MetricData>>> metricsByServiceAndType = 
metrics.stream()
            .collect(Collectors.groupingBy(
                MetricData::getServiceName,
                Collectors.groupingBy(MetricData::getMetricType)
            ));

        // Aggregate metrics per service
        for (Map.Entry<String, Map<String, List<MetricData>>> serviceEntry : 
metricsByServiceAndType.entrySet()) {
            String serviceName = serviceEntry.getKey();
            Map<String, List<MetricData>> serviceMetrics = 
serviceEntry.getValue();

            processServiceMetrics(serviceName, serviceMetrics);
        }
    }

    /**
     * Security event log processing with real-time alerting
     */
    @KafkaListener(
        topics = "security-logs",
        groupId = "security-analysis-group",
        containerFactory = "advancedBatchContainerFactory"
    )
    public void processSecurityLogs(@Payload List<String> securityMessages,
                                   Acknowledgment ack) {

        log.info("Processing security logs batch: size={}",
securityMessages.size());

        List<SecurityEvent> events = new ArrayList<>();
        List<SecurityAlert> alerts = new ArrayList<>();

        try {
            // Parse and analyze security events
            for (int i = 0; i < securityMessages.size(); i++) {
                String message = securityMessages.get(i);

                try {
                    SecurityEvent event = parseSecurityEvent(message);
                    events.add(event);

                    // Real-time threat detection
                    SecurityAlert alert = analyzeSecurityThreat(event);
                    if (alert != null) {
                        alerts.add(alert);
                    }

                } catch (Exception e) {
                    log.warn("Failed to parse security event at index {}: {}", i,
e.getMessage());
                    throw new BatchListenerFailedException("Security parsing
failed", e, i);
                }
            }

            // Batch store security events
            if (!events.isEmpty()) {
                logRepository.saveSecurityEvents(events);
            }

            // Send real-time alerts
            if (!alerts.isEmpty()) {
                alerts.forEach(alertingService::sendSecurityAlert);
            }

            ack.acknowledge();

            log.info("Security log processing completed: {} events, {} alerts",
                events.size(), alerts.size());

        } catch (Exception e) {
            log.error("Error processing security logs batch", e);
            throw e;
        }
    }

    // Log processing helper methods
    private LogEntry parseLogMessage(String logMessage) {
        try {
            // Parse structured log format (JSON, etc.)
            return LogEntry.fromJson(logMessage);
        } catch (Exception e) {
            log.debug("Failed to parse log message: {}", e.getMessage());
            return null;
        }
    }

    private Map<String, List<LogEntry>> groupLogsByTimeWindow(List<LogEntry>
logEntries) {
        return logEntries.stream()
            .collect(Collectors.groupingBy(entry -> {
                // Create 5-minute time windows
                long timestamp = entry.getTimestamp().toEpochMilli();
                long windowStart = (timestamp / 300000) * 300000;
                return Instant.ofEpochMilli(windowStart).toString();
            }));
    }

    private void processLogTimeWindow(String timeWindow, List<LogEntry> logs) {
        log.debug("Processing log time window {}: {} logs", timeWindow,
logs.size());

        // Aggregate log statistics
        LogWindowStats stats = LogWindowStats.builder()
            .timeWindow(timeWindow)
            .totalLogs(logs.size())
            .errorCount(logs.stream().mapToInt(log -> log.isError() ? 1 :
0).sum())
            .warnCount(logs.stream().mapToInt(log -> log.isWarning() ? 1 :
0).sum())

.uniqueServices(logs.stream().map(LogEntry::getServiceName).collect(Collectors.toS
et()))
            .build();

        // Store aggregated statistics
        logRepository.saveLogWindowStats(stats);

        // Check for anomalies
        if (stats.getErrorCount() > 100) { // Threshold
            alertingService.sendErrorSpike(stats);
        }
    }

    private MetricData parseMetricMessage(String metricMessage) {
        try {
            return MetricData.fromJson(metricMessage);
        } catch (Exception e) {
            log.debug("Failed to parse metric message: {}", e.getMessage());
            return null;
        }
    }

    private void processServiceMetrics(String serviceName, Map<String,
List<MetricData>> serviceMetrics) {
        log.debug("Processing metrics for service {}: {} metric types",
            serviceName, serviceMetrics.size());

        for (Map.Entry<String, List<MetricData>> metricEntry :
serviceMetrics.entrySet()) {
            String metricType = metricEntry.getKey();
            List<MetricData> metricValues = metricEntry.getValue();

            // Calculate aggregations
            MetricAggregation aggregation = calculateMetricAggregation(metricType,
metricValues);

            // Store aggregated metrics
            logRepository.saveMetricAggregation(serviceName, aggregation);

            // Check thresholds
            checkMetricThresholds(serviceName, metricType, aggregation);
        }
    }

    private SecurityEvent parseSecurityEvent(String message) {
        return SecurityEvent.fromJson(message);
    }

    private SecurityAlert analyzeSecurityThreat(SecurityEvent event) {
        // Implement threat detection logic
        if (event.getEventType().equals("LOGIN_FAILURE") &&
event.getFailureCount() > 5) {
            return SecurityAlert.builder()
                .alertType("BRUTE_FORCE_ATTACK")
                .severity("HIGH")
                .sourceEvent(event)
                .build();
        }

        return null; // No alert needed
    }

    private MetricAggregation calculateMetricAggregation(String metricType,
List<MetricData> metrics) {
        DoubleSummaryStatistics stats = metrics.stream()
            .mapToDouble(MetricData::getValue)
            .summaryStatistics();

        return MetricAggregation.builder()
            .metricType(metricType)
```

```
                .count(stats.getCount())
                .min(stats.getMin())
                .max(stats.getMax())
                .average(stats.getAverage())
                .sum(stats.getSum())
                .build();
    }

    private void checkMetricThresholds(String serviceName, String metricType,
MetricAggregation aggregation) {
        // Implement threshold checking logic
        if (metricType.equals("CPU_USAGE") && aggregation.getAverage() > 80.0) {
            alertingService.sendCpuAlert(serviceName, aggregation);
        }
    }
}

/**
 * ETL (Extract, Transform, Load) Processing using Batch Processing
 */
@Service
@lombok.extern.slf4j.Slf4j
public class EtlBatchProcessingService {

    @Autowired
    private DataTransformationService transformationService;

    @Autowired
    private DataWarehouseService dataWarehouseService;

    /**
     * Customer data ETL processing
     */
    @KafkaListener(
        topics = "raw-customer-data",
        groupId = "customer-etl-group",
        containerFactory = "analyticsBatchContainerFactory"
    )
    public void processCustomerDataEtl(@Payload List<String> rawCustomerData,
                                       Acknowledgment ack) {

        log.info("Processing customer data ETL batch: size={}",
rawCustomerData.size());

        try {
            // EXTRACT: Parse raw customer data
            List<RawCustomerRecord> rawRecords = rawCustomerData.stream()
                .map(this::extractCustomerRecord)
                .filter(Objects::nonNull)
                .collect(Collectors.toList());

            log.debug("Extracted {} valid customer records", rawRecords.size());

            // TRANSFORM: Clean and transform data
            List<TransformedCustomerRecord> transformedRecords =
rawRecords.stream()
                .map(transformationService::transformCustomerRecord)
                .filter(Objects::nonNull)
                .collect(Collectors.toList());

            log.debug("Transformed {} customer records",
transformedRecords.size());

            // Group by data category for optimized loading
            Map<String, List<TransformedCustomerRecord>> recordsByCategory =
transformedRecords.stream()

.collect(Collectors.groupingBy(TransformedCustomerRecord::getCategory));

            // LOAD: Batch insert into data warehouse
            for (Map.Entry<String, List<TransformedCustomerRecord>> entry :
recordsByCategory.entrySet()) {
                String category = entry.getKey();
                List<TransformedCustomerRecord> categoryRecords =
entry.getValue();

                dataWarehouseService.batchLoadCustomerRecords(category,
categoryRecords);

                log.debug("Loaded {} records for category: {}",
categoryRecords.size(), category);
            }

            ack.acknowledge();

            log.info("Customer ETL batch completed: {} records processed",
transformedRecords.size());

        } catch (Exception e) {
            log.error("Error in customer ETL batch processing", e);
            throw e;
        }
    }

    /**
     * Sales transaction ETL with data quality checks
     */
    @KafkaListener(
        topics = "sales-transactions",
        groupId = "sales-etl-group",
        containerFactory = "basicBatchContainerFactory"
    )
    public void processSalesTransactionEtl(@Payload List<String> salesData,
                                            Acknowledgment ack) {

        log.info("Processing sales transaction ETL batch: size={}",
salesData.size());
```

```
        List<String> dataQualityErrors = new ArrayList<>();

        try {
            // EXTRACT and validate
            List<SalesTransaction> transactions = new ArrayList<>();

            for (int i = 0; i < salesData.size(); i++) {
                String rawTransaction = salesData.get(i);

                try {
                    SalesTransaction transaction =
extractSalesTransaction(rawTransaction);

                    // Data quality validation
                    List<String> validationErrors =
validateSalesTransaction(transaction);
                    if (!validationErrors.isEmpty()) {
                        dataQualityErrors.addAll(validationErrors);
                        log.warn("Data quality issues at index {}: {}", i,
validationErrors);
                    } else {
                        transactions.add(transaction);
                    }

                } catch (Exception e) {
                    log.warn("Failed to extract transaction at index {}: {}", i,
e.getMessage());
                    dataQualityErrors.add("Extraction failed at index " + i + ": "
+ e.getMessage());
                }
            }

            if (!transactions.isEmpty()) {
                // TRANSFORM: Enrich with additional data
                List<EnrichedSalesTransaction> enrichedTransactions =
transactions.stream()
                    .map(transformationService::enrichSalesTransaction)
                    .collect(Collectors.toList());

                // LOAD: Batch load into analytics database

dataWarehouseService.batchLoadSalesTransactions(enrichedTransactions);

                log.info("Sales ETL batch completed: {} transactions loaded",
enrichedTransactions.size());
            }

            // Handle data quality issues
            if (!dataQualityErrors.isEmpty()) {
                handleDataQualityIssues(dataQualityErrors);
            }

            ack.acknowledge();
        } catch (Exception e) {
            log.error("Error in sales ETL batch processing", e);
            throw e;
        }
    }

    /**
     * Inventory data ETL with real-time updates
     */
    @KafkaListener(
        topics = "inventory-updates",
        groupId = "inventory-etl-group",
        containerFactory = "advancedBatchContainerFactory"
    )
    public void processInventoryEtl(@Payload List<String> inventoryData,
                                   @Header(KafkaHeaders.RECEIVED_TIMESTAMP)
List<Long> timestamps,
                                   Acknowledgment ack) {

        log.info("Processing inventory ETL batch: size={}", inventoryData.size());

        try {
            // Extract inventory updates with timestamps
            List<InventoryUpdate> updates = new ArrayList<>();

            for (int i = 0; i < inventoryData.size(); i++) {
                String rawUpdate = inventoryData.get(i);
                Long timestamp = timestamps.get(i);

                InventoryUpdate update = extractInventoryUpdate(rawUpdate,
timestamp);
                if (update != null) {
                    updates.add(update);
                }
            }

            // Group by product for batch processing
            Map<String, List<InventoryUpdate>> updatesByProduct = updates.stream()
                .collect(Collectors.groupingBy(InventoryUpdate::getProductId));

            // Process each product's updates
            for (Map.Entry<String, List<InventoryUpdate>> entry :
updatesByProduct.entrySet()) {
                String productId = entry.getKey();
                List<InventoryUpdate> productUpdates = entry.getValue();

                // Sort by timestamp to ensure correct order

productUpdates.sort(Comparator.comparing(InventoryUpdate::getTimestamp));

                // Apply updates in sequence
                processProductInventoryUpdates(productId, productUpdates);
            }
            ack.acknowledge();

            log.info("Inventory ETL batch completed: {} products updated", 
updatesByProduct.size());

        } catch (Exception e) {
            log.error("Error in inventory ETL batch processing", e);
            throw e;
        }
    }

    // ETL helper methods
    private RawCustomerRecord extractCustomerRecord(String rawData) {
        try {
            return RawCustomerRecord.fromCsv(rawData);
        } catch (Exception e) {
            log.debug("Failed to extract customer record: {}", e.getMessage());
            return null;
        }
    }

    private SalesTransaction extractSalesTransaction(String rawTransaction) {
        return SalesTransaction.fromJson(rawTransaction);
    }

    private List<String> validateSalesTransaction(SalesTransaction transaction) {
        List<String> errors = new ArrayList<>();

        if (transaction.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
            errors.add("Transaction amount must be positive");
        }

        if (transaction.getCustomerId() == null || 
transaction.getCustomerId().isEmpty()) {
            errors.add("Customer ID is required");
        }

        if (transaction.getTransactionDate().isAfter(Instant.now())) {
            errors.add("Transaction date cannot be in the future");
        }

        return errors;
    }

    private InventoryUpdate extractInventoryUpdate(String rawUpdate, Long 
timestamp) {
        try {
            InventoryUpdate update = InventoryUpdate.fromJson(rawUpdate);
            update.setTimestamp(Instant.ofEpochMilli(timestamp));
            return update;
        } catch (Exception e) {
            log.debug("Failed to extract inventory update: {}", e.getMessage());
            return null;
        }
    }

    private void handleDataQualityIssues(List<String> errors) {
        log.warn("Data quality issues detected: {} errors", errors.size());

        // Send to data quality monitoring system
        errors.forEach(error -> {
            log.debug("Data quality error: {}", error);
            // Could send to monitoring/alerting system
        });
    }

    private void processProductInventoryUpdates(String productId,
List<InventoryUpdate> updates) {
        log.debug("Processing {} inventory updates for product: {}",
updates.size(), productId);

        // Apply updates sequentially to maintain consistency
        for (InventoryUpdate update : updates) {
            dataWarehouseService.applyInventoryUpdate(productId, update);
        }
    }
}

/**
 * Real-time Analytics using Batch Processing
 */
@Service
@lombok.extern.slf4j.Slf4j
public class AnalyticsBatchProcessingService {

    @Autowired
    private AnalyticsRepository analyticsRepository;

    @Autowired
    private RealtimeDashboardService dashboardService;

    /**
     * User behavior analytics processing
     */
    @KafkaListener(
        topics = "user-events",
        groupId = "user-analytics-group",
        containerFactory = "analyticsBatchContainerFactory"
    )
    public void processUserBehaviorAnalytics(@Payload List<String> userEvents) {

        log.info("Processing user behavior analytics batch: size={}",
userEvents.size());

        // Parse user events
        List<UserEvent> events = userEvents.stream()
            .map(this::parseUserEvent)
            .filter(Objects::nonNull)
            .collect(Collectors.toList());

        // Group events by user for session analysis
        Map<String, List<UserEvent>> eventsByUser = events.stream()
            .collect(Collectors.groupingBy(UserEvent::getUserId));

        // Analyze user sessions
        List<UserSession> sessions = new ArrayList<>();
        for (Map.Entry<String, List<UserEvent>> entry : eventsByUser.entrySet()) {
            String userId = entry.getKey();
            List<UserEvent> userEvents1 = entry.getValue();

            UserSession session = analyzeUserSession(userId, userEvents1);
            sessions.add(session);
        }

        // Store session analytics
        analyticsRepository.saveUserSessions(sessions);

        // Update real-time dashboard
        updateUserAnalyticsDashboard(sessions);

        log.info("User behavior analytics completed: {} sessions analyzed",
sessions.size());
    }

    /**
     * Product recommendation analytics
     */
    @KafkaListener(
        topics = "product-interactions",
        groupId = "recommendation-analytics-group",
        containerFactory = "analyticsBatchContainerFactory"
    )
    public void processRecommendationAnalytics(@Payload List<String>
interactionData) {

        log.info("Processing recommendation analytics batch: size={}",
interactionData.size());

        // Parse product interactions
        List<ProductInteraction> interactions = interactionData.stream()
            .map(this::parseProductInteraction)
            .filter(Objects::nonNull)
            .collect(Collectors.toList());

        // Build co-occurrence matrix for product recommendations
        Map<String, Map<String, Integer>> coOccurrenceMatrix =
buildCoOccurrenceMatrix(interactions);

        // Update recommendation scores
        updateRecommendationScores(coOccurrenceMatrix);

        // Generate trending product insights
        List<TrendingProduct> trendingProducts =
identifyTrendingProducts(interactions);

        // Store analytics results
        analyticsRepository.saveRecommendationMatrix(coOccurrenceMatrix);
        analyticsRepository.saveTrendingProducts(trendingProducts);

        log.info("Recommendation analytics completed: {} products analyzed",
coOccurrenceMatrix.size());
    }

    // Analytics helper methods
    private UserEvent parseUserEvent(String eventData) {
        try {
            return UserEvent.fromJson(eventData);
        } catch (Exception e) {
            log.debug("Failed to parse user event: {}", e.getMessage());
            return null;
        }
    }

    private UserSession analyzeUserSession(String userId, List<UserEvent> events)
{
        // Sort events by timestamp
        events.sort(Comparator.comparing(UserEvent::getTimestamp));

        Instant sessionStart = events.get(0).getTimestamp();
        Instant sessionEnd = events.get(events.size() - 1).getTimestamp();
        long sessionDuration = Duration.between(sessionStart,
sessionEnd).toSeconds();

        // Calculate session metrics
        Map<String, Long> eventCounts = events.stream()
            .collect(Collectors.groupingBy(UserEvent::getEventType,
Collectors.counting()));

        return UserSession.builder()
            .userId(userId)
            .sessionStart(sessionStart)
            .sessionEnd(sessionEnd)
            .sessionDuration(sessionDuration)
            .totalEvents(events.size())
            .eventCounts(eventCounts)
            .build();
    }

    private ProductInteraction parseProductInteraction(String interactionData) {
        try {
            return ProductInteraction.fromJson(interactionData);
        } catch (Exception e) {
            log.debug("Failed to parse product interaction: {}", e.getMessage());
            return null;
        }
    }
    private Map<String, Map<String, Integer>>
buildCoOccurrenceMatrix(List<ProductInteraction> interactions) {
        // Group interactions by user session
        Map<String, List<ProductInteraction>> interactionsBySession =
interactions.stream()
            .collect(Collectors.groupingBy(ProductInteraction::getSessionId));

        Map<String, Map<String, Integer>> coOccurrence = new HashMap<>();

        // Build co-occurrence matrix
        for (List<ProductInteraction> sessionInteractions :
interactionsBySession.values()) {
            Set<String> sessionProducts = sessionInteractions.stream()
                .map(ProductInteraction::getProductId)
                .collect(Collectors.toSet());

            // For each pair of products in the session
            for (String product1 : sessionProducts) {
                for (String product2 : sessionProducts) {
                    if (!product1.equals(product2)) {
                        coOccurrence.computeIfAbsent(product1, k -> new HashMap<>
())
                            .merge(product2, 1, Integer::sum);
                    }
                }
            }
        }

        return coOccurrence;
    }

    private void updateRecommendationScores(Map<String, Map<String, Integer>>
coOccurrenceMatrix) {
        // Update recommendation algorithms with new co-occurrence data
        log.debug("Updating recommendation scores for {} products",
coOccurrenceMatrix.size());
    }

    private List<TrendingProduct>
identifyTrendingProducts(List<ProductInteraction> interactions) {
        // Analyze interaction frequency and timing to identify trending products
        Map<String, Long> productCounts = interactions.stream()
            .collect(Collectors.groupingBy(ProductInteraction::getProductId,
Collectors.counting()));

        return productCounts.entrySet().stream()
            .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
            .limit(10) // Top 10 trending
            .map(entry -> TrendingProduct.builder()
                .productId(entry.getKey())
                .interactionCount(entry.getValue())
                .trendScore(calculateTrendScore(entry.getValue()))
                .build())
            .collect(Collectors.toList());
    }

    private double calculateTrendScore(Long interactionCount) {
        // Simple trend score calculation
        return Math.log(interactionCount + 1) * 100;
    }

    private void updateUserAnalyticsDashboard(List<UserSession> sessions) {
        // Update real-time dashboard with session analytics
        UserAnalyticsSummary summary = UserAnalyticsSummary.builder()
            .totalSessions(sessions.size())
            .averageSessionDuration(sessions.stream()
                .mapToLong(UserSession::getSessionDuration)
                .average()
                .orElse(0.0))
            .totalUniqueUsers(sessions.stream()
                .map(UserSession::getUserId)
                .collect(Collectors.toSet())
                .size())
            .build();

        dashboardService.updateUserAnalytics(summary);
    }
}
```

This completes Part 2 of the comprehensive Spring Kafka Batch Processing guide, covering error handling and real-world use cases. Part 3 will cover comparisons, best practices, and production patterns.