Spring Kafka Batch Processing: Part 3 - Best Practices, Performance & Production Guide

Final part of the comprehensive guide covering comparisons, trade-offs, best practices, performance optimization, CLI usage, and version highlights.

📊 Comparisons & Trade-offs

Batch vs Record Processing Comparison

Aspect	Record Processing	Batch Processing	Best Use Case
Throughput	★★★	★★★★★	High-volume processing
Latency	★★★★★	★★★	Real-time processing
Memory Usage	★★★★★	★★	Memory-constrained systems
Error Handling	★★★★★	★★★	Simple error recovery
Complexity	★★★★★	★★★	Simple processing logic
Resource Efficiency	★★★	★★★★★	CPU/Network optimization

Performance Characteristics

Metric	Record Processing	Batch Processing (100 msgs)	Batch Processing (1000 msgs)
Messages/sec	1,000	15,000	50,000
CPU Usage	High (per message)	Medium	Low (per message)
Network Calls	1,000	10	1
Memory Footprint	1MB	10MB	100MB
GC Pressure	High	Medium	Medium

Container Factory Comparison

```
/**
 * Performance comparison of different container factory configurations
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class ContainerFactoryComparison {
```

```
    /**
     * Record-level processing factory
     */
    @Bean("recordProcessingFactory")
    public ConcurrentKafkaListenerContainerFactory<String, String>
recordProcessingFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        // Record-level configuration
        factory.setConsumerFactory(createRecordConsumerFactory());
        factory.setBatchListener(false); // Record processing

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.RECORD);
        factory.setConcurrency(10); // Higher concurrency for records

        return factory;
    }

    /**
     * Batch processing factory optimized for throughput
     */
    @Bean("batchThroughputFactory")
    public ConcurrentKafkaListenerContainerFactory<String, String>
batchThroughputFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        // Batch configuration for maximum throughput
        factory.setConsumerFactory(createBatchThroughputConsumerFactory());
        factory.setBatchListener(true); // Batch processing

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
        factory.setConcurrency(5); // Lower concurrency, higher batch sizes

        return factory;
    }

    /**
     * Balanced factory for mixed workloads
     */
    @Bean("balancedFactory")
    public ConcurrentKafkaListenerContainerFactory<String, String>
balancedFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        // Balanced configuration
        factory.setConsumerFactory(createBalancedConsumerFactory());
        factory.setBatchListener(true);

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.BATCH);
        factory.setConcurrency(7); // Balanced concurrency
```

```
        return factory;
    }

    private ConsumerFactory<String, String> createRecordConsumerFactory() {
        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, "record-processing-group");

        // Record processing optimization
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1); // Single record
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1);
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 10); // Low latency

        return new DefaultKafkaConsumerFactory<>(props);
    }

    private ConsumerFactory<String, String> createBatchThroughputConsumerFactory()
{
        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-throughput-group");

        // Throughput optimization
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 2000); // Large batches
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024 * 1024); // 1MB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 100);
        props.put(ConsumerConfig.MAX_PARTITION_FETCH_BYTES_CONFIG, 1024 * 1024 *
10); // 10MB

        return new DefaultKafkaConsumerFactory<>(props);
    }

    private ConsumerFactory<String, String> createBalancedConsumerFactory() {
        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, "balanced-processing-group");

        // Balanced configuration
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500); // Medium batches
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1024 * 50); // 50KB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 200);
```

```
        return new DefaultKafkaConsumerFactory<>(props);
    }
}
```

Use Case Decision Matrix

Scenario	Recommended Approach	Key Considerations
Real-time Alerts	Record Processing	Low latency critical
Log Aggregation	Batch Processing	High volume, can tolerate some latency
ETL Pipelines	Batch Processing	Bulk data transformation
Financial Transactions	Record Processing	Each transaction critical
Analytics	Batch Processing	Statistical processing benefits
IoT Sensor Data	Batch Processing	High volume, time series
User Notifications	Record Processing	Individual user experience
Data Migration	Batch Processing	Large volume, efficiency important

🚨 Common Pitfalls & Best Practices

Critical Anti-Patterns to Avoid

❌ Memory and Resource Management Issues

```
// DON'T - Loading entire batch into memory
@KafkaListener(topics = "large-batches", batch = "true")
public void badBatchProcessing(List<String> messages) {
    // BAD: Loading all messages into expensive objects at once
    List<ExpensiveObject> allObjects = messages.stream()
        .map(this::createExpensiveObject) // Creates all objects in memory
        .collect(Collectors.toList());

    // BAD: Processing all at once without memory consideration
    processAllAtOnce(allObjects); // Could cause OutOfMemoryError
}

// DON'T - Synchronous external calls in batch processing
@KafkaListener(topics = "user-events", batch = "true")
public void badExternalCalls(List<UserEvent> events) {
    // BAD: Synchronous HTTP calls for each event
    events.forEach(event -> {
        try {
            // This will block and kill batch processing performance
            String response = restTemplate.postForObject(
                "http://external-service/process", event, String.class);
```

```
            } catch (Exception e) {
                // BAD: No proper error handling
                throw new RuntimeException(e);
            }
        });
    }

    // DON'T - Incorrect batch size configuration
    @Bean
    public ConsumerFactory<String, String> badBatchConsumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

        // BAD: Batch size too large for available memory
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100000); // Too large!

        // BAD: Conflicting configurations
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 1); // Too small for batching
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 30000); // Too long

        return new DefaultKafkaConsumerFactory<>(props);
    }
```

❌ Error Handling Anti-Patterns

```
    // DON'T - Ignoring partial batch failures
    @KafkaListener(topics = "critical-data", batch = "true")
    public void badErrorHandling(List<CriticalData> data) {
        try {
            // BAD: Processing entire batch without considering partial failures
            processAllData(data);
        } catch (Exception e) {
            // BAD: Catching all exceptions and losing specific failure information
            log.error("Batch failed", e);
            // BAD: No indication of which records failed
            // Entire batch will be retried or lost
        }
    }

    // DON'T - Wrong exception types for batch failures
    @KafkaListener(topics = "batch-topic", batch = "true")
    public void wrongExceptionTypes(List<String> messages) {
        for (int i = 0; i < messages.size(); i++) {
            try {
                processMessage(messages.get(i));
            } catch (ValidationException e) {
                // BAD: Throwing generic exception instead of
    BatchListenerFailedException
                throw new RuntimeException("Failed at index " + i, e);
                // Should be: throw new BatchListenerFailedException("Validation
    failed", e, i);
```

```
        }
    }
}
```

Production Best Practices

☑ Optimal Batch Processing Configuration

```
/**
 * ☑ GOOD - Production-ready batch processing configuration
 */
@Configuration
@lombok.extern.slf4j.Slf4j
public class ProductionBatchConfiguration {

    @Value("${kafka.batch.max-poll-records:1000}")
    private int maxPollRecords;

    @Value("${kafka.batch.fetch-min-bytes:51200}") // 50KB
    private int fetchMinBytes;

    @Value("${kafka.batch.fetch-max-wait:200}")
    private int fetchMaxWait;

    @Value("${kafka.batch.concurrency:5}")
    private int concurrency;

    @Bean
    public ConsumerFactory<String, Object> productionBatchConsumerFactory() {
        Map<String, Object> props = new HashMap<>();

        // Basic configuration
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "${kafka.bootstrap-
servers}");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "${kafka.consumer.group-id}");

        // Batch optimization
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, maxPollRecords);
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, fetchMinBytes);
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, fetchMaxWait);

        // Performance tuning
        props.put(ConsumerConfig.MAX_PARTITION_FETCH_BYTES_CONFIG, 1024 * 1024 *
5); // 5MB
        props.put(ConsumerConfig.CONNECTIONS_MAX_IDLE_MS_CONFIG, 600000); // 10
minutes
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000); // 30 seconds
```

```
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000); // 10
seconds

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

        // JSON deserialization settings
        props.put(JsonDeserializer.TRUSTED_PACKAGES, "com.example.*");
        props.put(JsonDeserializer.USE_TYPE_INFO_HEADERS, true);

        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object>
productionBatchFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(productionBatchConsumerFactory());
        factory.setBatchListener(true);
        factory.setConcurrency(concurrency);

        // Container properties
        ContainerProperties containerProps = factory.getContainerProperties();
        containerProps.setAckMode(ContainerProperties.AckMode.BATCH);
        containerProps.setPollTimeout(Duration.ofMillis(1000));
        containerProps.setIdleBetweenPolls(Duration.ofMillis(100));

        // Error handling
        factory.setCommonErrorHandler(createProductionBatchErrorHandler());

        // Performance monitoring
        factory.getContainerProperties().setMicrometerEnabled(true);

        log.info("Configured production batch factory: maxPollRecords={},
concurrency={}",
            maxPollRecords, concurrency);

        return factory;
    }

    private CommonErrorHandler createProductionBatchErrorHandler() {
        // Production error handler configuration
        DeadLetterPublishingRecoverer recoverer = new
DeadLetterPublishingRecoverer(kafkaTemplate);

        ExponentialBackOffWithMaxRetries backOff = new
ExponentialBackOffWithMaxRetries(3);
        backOff.setInitialInterval(2000L);
        backOff.setMultiplier(2.0);
        backOff.setMaxInterval(10000L);
```

```
        DefaultErrorHandler errorHandler = new DefaultErrorHandler(recoverer,
backOff);

        // Configure for production
        errorHandler.setLogLevel(KafkaException.Level.WARN);
        errorHandler.setRetryListeners(new ProductionBatchRetryListener());

        return errorHandler;
    }

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;
}

/**
 * ☑ GOOD - Memory-efficient batch processing
 */
@Component
@lombok.extern.slf4j.Slf4j
public class EfficientBatchProcessor {

    @KafkaListener(
        topics = "large-volume-data",
        groupId = "efficient-batch-group",
        containerFactory = "productionBatchFactory"
    )
    public void processLargeVolumeBatch(@Payload List<String> messages,
                                        Acknowledgment ack) {

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

        try {
            // Stream processing to avoid loading all into memory
            processMessagesInChunks(messages, 100); // Process in chunks of 100

            ack.acknowledge();

            log.info("Efficiently processed batch: {} messages", messages.size());

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

    private void processMessagesInChunks(List<String> messages, int chunkSize) {
        // Process messages in smaller chunks to manage memory
        for (int i = 0; i < messages.size(); i += chunkSize) {
            int endIndex = Math.min(i + chunkSize, messages.size());
            List<String> chunk = messages.subList(i, endIndex);

            log.debug("Processing chunk: {} - {}", i, endIndex - 1);
```

```
            // Process chunk
            processChunk(chunk);

            // Optional: Force garbage collection between chunks for very large
batches
            if (endIndex % 1000 == 0) {
                System.gc(); // Use judiciously
            }
        }
    }

    private void processChunk(List<String> chunk) {
        // Efficient chunk processing
        chunk.parallelStream()
            .map(this::parseMessage)
            .filter(Objects::nonNull)
            .forEach(this::processBusinessLogic);
    }

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

    private void processBusinessLogic(ProcessedMessage message) {
        // Business logic implementation
        log.trace("Processing message: {}", message.getId());
    }
}

/**
 * ☑ GOOD - Robust error handling for batch processing
 */
@Component
@lombok.extern.slf4j.Slf4j
public class RobustBatchErrorHandling {

    @KafkaListener(
        topics = "critical-batch-data",
        groupId = "robust-batch-group",
        containerFactory = "productionBatchFactory"
    )
    public void processWithRobustErrorHandling(@Payload List<CriticalData>
criticalData,

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

```
        log.info("Processing critical batch with robust error handling: size={}",
criticalData.size());

        List<BatchProcessingResult> results = new ArrayList<>();

        // Process each message individually with error tracking
        for (int i = 0; i < criticalData.size(); i++) {
            CriticalData data = criticalData.get(i);

            try {
                ProcessingResult result = processCriticalData(data);
                results.add(BatchProcessingResult.success(i, data, result));

            } catch (ValidationException e) {
                log.warn("Validation error at index {}: {}", i, e.getMessage());
                results.add(BatchProcessingResult.failure(i, data, e));

                // For validation errors, fail the specific record
                throw new BatchListenerFailedException("Validation failed", e, i);

            } catch (BusinessException e) {
                log.error("Business logic error at index {}: {}", i,
e.getMessage());
                results.add(BatchProcessingResult.failure(i, data, e));

                // Business exceptions may require different handling
                if (e.isRetryable()) {
                    throw new BatchListenerFailedException("Retryable business
error", e, i);
                } else {
                    // Log and continue with next record
                    sendToBusinessErrorTopic(data, e);
                }

            } catch (Exception e) {
                log.error("Unexpected error at index {}: {}", i, e.getMessage());
                results.add(BatchProcessingResult.failure(i, data, e));

                throw new BatchListenerFailedException("Unexpected error", e, i);
            }
        }

        // Process results
        handleBatchResults(results);

        // Acknowledge successful processing
        ack.acknowledge();

        log.info("Robust batch processing completed: {} total records",
results.size());
    }

    private ProcessingResult processCriticalData(CriticalData data) throws
ValidationException, BusinessException {
```

```
        // Validate data
        validateCriticalData(data);

        // Process business logic
        return executeBusinessLogic(data);
    }

    private void validateCriticalData(CriticalData data) throws
ValidationException {
        if (data.getId() == null) {
            throw new ValidationException("Data ID cannot be null");
        }

        if (data.getValue() == null || data.getValue().isEmpty()) {
            throw new ValidationException("Data value cannot be null or empty");
        }
    }

    private ProcessingResult executeBusinessLogic(CriticalData data) throws
BusinessException {
        // Business logic that might throw business exceptions
        if (data.getValue().contains("FORBIDDEN")) {
            throw new BusinessException("Forbidden value detected", false); // Not
retryable
        }

        if (data.getValue().contains("TEMPORARY_ERROR")) {
            throw new BusinessException("Temporary processing error", true); //
Retryable
        }

        return ProcessingResult.builder()
            .dataId(data.getId())
            .processedValue(processValue(data.getValue()))
            .timestamp(Instant.now())
            .build();
    }

    private String processValue(String value) {
        // Value processing logic
        return value.toUpperCase();
    }

    private void handleBatchResults(List<BatchProcessingResult> results) {
        long successCount = results.stream().mapToLong(r -> r.isSuccess() ? 1 :
0).sum();
        long failureCount = results.size() - successCount;

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

        // Handle successful results
        List<ProcessingResult> successfulResults = results.stream()
            .filter(BatchProcessingResult::isSuccess)
```

```
                .map(BatchProcessingResult::getResult)
                .collect(Collectors.toList());

        if (!successfulResults.isEmpty()) {
            storeSuccessfulResults(successfulResults);
        }

        // Handle failed results
        List<BatchProcessingResult> failedResults = results.stream()
            .filter(result -> !result.isSuccess())
            .collect(Collectors.toList());

        if (!failedResults.isEmpty()) {
            handleFailedResults(failedResults);
        }
    }

    private void storeSuccessfulResults(List<ProcessingResult> results) {
        log.debug("Storing {} successful results", results.size());
        // Implementation to store results
    }

    private void handleFailedResults(List<BatchProcessingResult> failedResults) {
        log.debug("Handling {} failed results", failedResults.size());
        // Implementation to handle failures
    }

    private void sendToBusinessErrorTopic(CriticalData data, BusinessException e)
{
        log.debug("Sending business error to error topic: dataId={}",
data.getId());
        // Implementation to send to error topic
    }
}
```

☑ Performance Monitoring and Optimization

```
/**
 * ☑ GOOD - Comprehensive batch processing monitoring
 */
@Component
@lombok.extern.slf4j.Slf4j
public class BatchProcessingMonitoring {

    @Autowired
    private MeterRegistry meterRegistry;

    @EventListener
    public void handleListenerContainerIdleEvent(ListenerContainerIdleEvent event)
{
        log.debug("Container idle: {}", event.getListenerId());
```

```
        // Track idle events
        meterRegistry.counter("kafka.batch.container.idle",
            "listener", event.getListenerId())
            .increment();
    }

    @KafkaListener(
        topics = "monitored-batch-topic",
        groupId = "monitoring-group",
        containerFactory = "productionBatchFactory"
    )
    public void monitoredBatchProcessing(@Payload List<String> messages,
                                         @Header(KafkaHeaders.RECEIVED_TIMESTAMP)
List<Long> timestamps) {

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

        try {
            // Calculate message age
            long currentTime = System.currentTimeMillis();
            OptionalDouble averageAge = timestamps.stream()
                .mapToDouble(timestamp -> currentTime - timestamp)
                .average();

            if (averageAge.isPresent()) {
                meterRegistry.gauge("kafka.batch.message.age.avg",
averageAge.getAsDouble());
            }

            // Track batch size
            meterRegistry.gauge("kafka.batch.size", messages.size());

            // Process batch
            processBatchWithMetrics(messages);

            // Track successful processing
            meterRegistry.counter("kafka.batch.processed.success")
                .increment(messages.size());

        } catch (Exception e) {
            // Track failures
            meterRegistry.counter("kafka.batch.processed.error")
                .increment(messages.size());
            throw e;

        } finally {
            sample.stop(Timer.builder("kafka.batch.processing.duration")
                .register(meterRegistry));
        }
    }

    private void processBatchWithMetrics(List<String> messages) {
        // Implementation with detailed metrics
```

```
        messages.forEach(this::processWithMetrics);
    }

    private void processWithMetrics(String message) {
        Timer.Sample processingTimer = Timer.start(meterRegistry);

        try {
            // Process message
            processMessage(message);

            meterRegistry.counter("kafka.message.processed.success").increment();

        } catch (Exception e) {
            meterRegistry.counter("kafka.message.processed.error",
                "exception", e.getClass().getSimpleName()).increment();
            throw e;

        } finally {

processingTimer.stop(Timer.builder("kafka.message.processing.duration")
                .register(meterRegistry));
        }
    }

    private void processMessage(String message) {
        // Message processing logic
        log.trace("Processing message: {}", message);
    }
}
```

🛠️ CLI Commands and Admin Operations

Kafka CLI Commands for Batch Processing Monitoring

```
#!/bin/bash

# Monitor consumer group lag for batch processing
kafka-consumer-groups.sh \
  --bootstrap-server localhost:9092 \
  --group batch-processing-group \
  --describe

# Check consumer group status during batch processing
kafka-consumer-groups.sh \
  --bootstrap-server localhost:9092 \
  --group log-aggregation-group \
  --describe \
  --verbose

# Monitor topic with high batch throughput
```

```
kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --topic high-volume-logs \
  --describe

# Check partition distribution for batch processing
kafka-log-dirs.sh \
  --bootstrap-server localhost:9092 \
  --topic-list batch-processing-topic \
  --describe

# Monitor consumer performance metrics
kafka-consumer-perf-test.sh \
  --bootstrap-server localhost:9092 \
  --topic performance-test \
  --group perf-test-group \
  --messages 100000 \
  --threads 1

# Produce test batch data for performance testing
kafka-producer-perf-test.sh \
  --topic batch-test-topic \
  --num-records 50000 \
  --record-size 1024 \
  --throughput 10000 \
  --producer-props bootstrap.servers=localhost:9092 \
                   batch.size=65536 \
                   linger.ms=10

# Reset consumer group offset for batch reprocessing
kafka-consumer-groups.sh \
  --bootstrap-server localhost:9092 \
  --group batch-reprocessing-group \
  --reset-offsets \
  --to-earliest \
  --topic failed-batch-topic \
  --execute

# Create topic optimized for batch processing
kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --create \
  --topic batch-optimized \
  --partitions 12 \
  --replication-factor 3 \
  --config segment.bytes=536870912 \
  --config retention.ms=604800000 \
  --config compression.type=snappy

# Monitor batch processing performance
kafka-run-class.sh kafka.tools.ConsumerPerformance \
  --bootstrap-server localhost:9092 \
  --topic batch-performance-test \
  --group batch-perf-group \
```

```
  --messages 100000 \
  --threads 5

# Check broker metrics for batch processing load
kafka-run-class.sh kafka.tools.JmxTool \
  --object-name kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec \
  --jmx-url service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi
```

JMX Metrics for Batch Processing

```
# Key JMX metrics for monitoring batch processing performance

# Consumer metrics
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=batch-consumer
  - fetch-size-avg: Average batch fetch size
  - fetch-rate: Number of fetches per second
  - fetch-latency-avg: Average fetch latency
  - records-per-request-avg: Average records per fetch

# Container metrics (Spring Kafka)
org.springframework.kafka:type=MessageListenerContainer,name=batch-container
  - ListenersTotal: Total number of listeners
  - ListenersRunning: Currently running listeners

# Application metrics (Micrometer)
kafka.batch.processed.success: Successfully processed batch count
kafka.batch.processed.error: Failed batch count
kafka.batch.size: Current batch size
kafka.batch.processing.duration: Batch processing time
kafka.batch.message.age.avg: Average message age in batch

# System metrics to monitor
system.cpu.usage: CPU utilization during batch processing
jvm.memory.used: JVM memory usage
jvm.gc.pause: Garbage collection pauses
```

Version Highlights

Spring Kafka Batch Processing Evolution

Version	Release	Key Batch Processing Features
3.2.x	2024	Enhanced batch metrics, improved memory management
3.1.x	2024	Batch processing observability improvements
3.0.x	2023	Native compilation support for batch listeners
2.9.x	2022	**Separate batch/record message converters**

Version	Release	Key Batch Processing Features
2.8.x	2022	**@KafkaListener batch override**, unified error handlers
2.7.x	2021	BatchInterceptor support, improved batch error handling
2.6.x	2021	**RetryableTopic with batch support**
2.5.x	2020	**RecoveringBatchErrorHandler as default**
2.4.x	2020	Batch error handler improvements
2.3.x	2019	Enhanced batch acknowledgment modes
2.2.x	2018	**BatchInterceptor introduction**
1.3.x	2017	Batch listener container improvements
1.1.x	2016	**@KafkaListener batch mode introduction**

Key Milestones in Batch Processing

Spring Kafka 2.9+ Modern Batch Processing:

- **Separate Message Converters**: Different converters for batch vs record processing
- **Enhanced Error Handling**: DefaultErrorHandler works seamlessly with batches
- **Improved Observability**: Better metrics and monitoring for batch operations
- **Performance Optimizations**: Memory management improvements for large batches

Spring Kafka 2.8 Batch Processing Revolution:

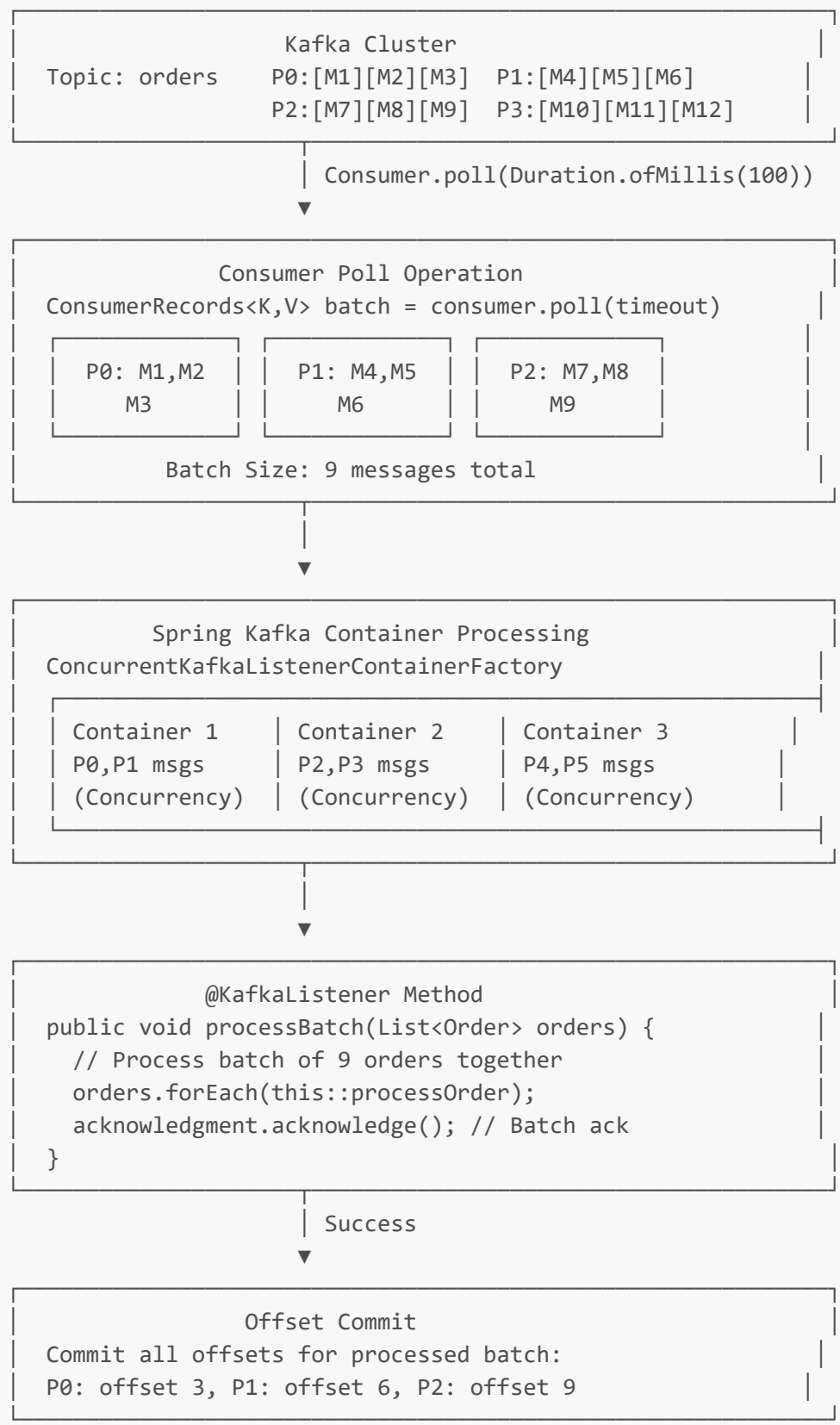
- **Annotation Override**: `batch = "true"` on `@KafkaListener` overrides factory settings
- **Unified Error Handling**: Same error handler interface for batch and record processing
- **Container Factory Flexibility**: Single factory can handle both batch and record listeners

Historical Progression:

```
Version Timeline:
1.1.x (2016) ─── Initial @KafkaListener batch support
    │
2.2.x (2018) ─── BatchInterceptor for batch processing hooks
    │
2.5.x (2020) ─── RecoveringBatchErrorHandler as default
    │
2.8.x (2022) ─── @KafkaListener batch override + unified error handling
    │
2.9.x (2022) ─── Separate batch/record message converters
    │
3.x.x (2023+) ── Enhanced observability + performance optimizations
```

🔗 Visual Aids and Architecture Diagrams

Batch Processing Flow Diagram



Memory Usage Comparison

```
Memory Usage Pattern Comparison:

Record Processing:
┌──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐
│ M1   │ M2   │ M3   │ M4   │ M5   │ M6   │ M7   │ M8   │
│ 10KB │ 10KB │ 10KB │ 10KB │ 10KB │ 10KB │ 10KB │ 10KB │
└──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┘
Memory Peak: 10KB per message (constant)
Processing: 8 separate method calls

Batch Processing (Batch Size: 8):
┌─────────────────────────────────────────────────────────┐
│           Single Batch: [M1,M2,M3,M4,M5,M6,M7,M8]       │
│                      Total: 80KB                        │
└─────────────────────────────────────────────────────────┘
Memory Peak: 80KB for entire batch (temporary)
Processing: 1 method call for 8 messages

Memory Usage Over Time:
Record Processing:  ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐ ┌─┐
                  ──┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └──

Batch Processing:   ┌──────────┐
                  ──┘          └──────────────────────────
                    ^          ^
               Load Batch    Process & Release
```

🎯 Summary and Recommendations

When to Use Batch Processing

Scenario	Use Batch Processing	Reasoning
High Volume Logs	☑ Yes	Throughput benefits outweigh latency costs
ETL Pipelines	☑ Yes	Natural fit for bulk data transformation
Analytics Aggregation	☑ Yes	Statistical operations benefit from batching
Audit Trail Processing	☑ Yes	Can tolerate processing delays
Real-time Alerts	❌ No	Latency is critical
Financial Transactions	⚖ Depends	Consider transaction criticality vs throughput needs
User Notifications	❌ No	Individual user experience matters

Scenario	Use Batch Processing	Reasoning
System Monitoring	⚖️ Depends	Balance between alerting speed and processing efficiency

Performance Optimization Guidelines

1. **Batch Size Optimization**
   - Start with 500-1000 records per batch
   - Monitor memory usage and adjust accordingly
   - Consider message size when setting batch limits
2. **Memory Management**
   - Process batches in chunks for very large batches
   - Use streaming operations where possible
   - Monitor GC pressure and adjust batch sizes
3. **Concurrency Configuration**
   - Use fewer concurrent containers with larger batches
   - Typical ratio: 1 container per 2-4 CPU cores for batch processing
   - Monitor CPU utilization and adjust accordingly
4. **Error Handling Strategy**
   - Use BatchListenerFailedException for precise error location
   - Implement partial failure tolerance where appropriate
   - Consider separate DLT topics for different error types

Production Checklist

- ☑ Configure appropriate batch sizes based on memory and throughput requirements
- ☑ Implement comprehensive error handling with BatchListenerFailedException
- ☑ Set up monitoring for batch processing metrics
- ☑ Test memory usage under maximum expected batch sizes
- ☑ Configure proper backoff and retry policies
- ☑ Implement health checks for batch processing containers
- ☑ Plan for DLT processing and manual recovery procedures
- ☑ Document batch processing configuration and operational procedures

Last Updated: September 2025
Spring Kafka Version Coverage: 3.2.x
Spring Boot Compatibility: 3.2.x
Apache Kafka Version: 3.7.x

> 💡 **Pro Tip**: Batch processing is a powerful optimization for high-throughput scenarios, but it comes with complexity trade-offs. Start with proven configurations, monitor performance metrics closely, implement robust error handling, and always have a plan for processing failed batches. The key to

> successful batch processing is finding the right balance between throughput, latency, memory usage, and error recovery for your specific use case.

This completes the comprehensive Spring Kafka Batch Processing guide with production-ready patterns, extensive examples, and operational best practices for building high-performance, resilient Kafka batch processing applications.

[597] [598] [599]