Kafka Integration with Java: Complete Developer Guide

A comprehensive refresher on Apache Kafka's Java integration, covering modern Java features, client APIs, serialization, and best practices for Java developers.

Table of Contents

🔌 Kafka Java Clients

Simple Explanation

Kafka Java clients are the official libraries that allow Java applications to interact with Kafka clusters. They provide high-level APIs for producing, consuming, streaming, and administering Kafka resources.

Problem It Solves

- **Language Integration**: Native Java integration with type safety and modern language features
- **Performance**: Optimized for JVM with efficient memory management and threading
- **Ecosystem**: Seamless integration with Java frameworks like Spring, Quarkus, and enterprise systems
- **Enterprise Features**: Built-in support for security, monitoring, and distributed processing

Internal Architecture

```
Kafka Java Client Architecture:

Application Layer:
```

```
┌─────────────────────────────────────────────────────────────┐
│ Producer API     │ Consumer API     │ Streams API      │ Admin │
│ - Send records   │ - Poll records   │ - Process        │ - Mgmt │
│ - Async/Sync     │ - Auto commit    │ - Transform      │ - Meta │
└─────────────────────────────────────────────────────────────┘
                              ↓
Client Library Layer:
┌─────────────────────────────────────────────────────────────┐
│ Serialization    │ Network I/O      │ Metadata         │ Config │
│ - SerDes         │ - TCP/SSL        │ - Topic info     │ - Props │
│ - Schema Reg     │ - Compression    │ - Partition      │ - Auth  │
└─────────────────────────────────────────────────────────────┘
                              ↓
Kafka Protocol Layer:
┌─────────────────────────────────────────────────────────────┐
│ Binary Protocol │ Request/Response │ Connection Pool │ Retry │
└─────────────────────────────────────────────────────────────┘
```

Producer & Consumer APIs

Modern Producer Implementation

```java
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;
import java.util.Properties;
import java.util.concurrent.Future;

/**
 * Modern Kafka Producer with comprehensive error handling and metrics
 */
public class ModernKafkaProducer {

    private final KafkaProducer<String, String> producer;
    private final String topicName;

    public ModernKafkaProducer(String topicName) {
        this.topicName = topicName;
        this.producer = new KafkaProducer<>(createProducerProps());
    }

    /**
     * Optimized producer configuration for modern applications
     */
    private Properties createProducerProps() {
        Properties props = new Properties();

        // Connection settings
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "modern-producer-" +
System.currentTimeMillis());
```

```
        // Serialization
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

        // Performance optimization
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536); // 64KB
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67108864); // 64MB

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

        // Timeout configuration
        props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120000); // 2 minutes
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000); // 30 seconds

        return props;
    }

    /**
     * Send message asynchronously with callback
     */
    public Future<RecordMetadata> sendAsync(String key, String value) {
        ProducerRecord<String, String> record = new ProducerRecord<>(topicName,
key, value);

        return producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata metadata, Exception exception)
{
                if (exception != null) {
                    System.err.printf("Failed to send message: key=%s,
error=%s%n",
                        key, exception.getMessage());
                } else {
                    System.out.printf("Sent message: key=%s, partition=%d,
offset=%d%n",
                        key, metadata.partition(), metadata.offset());
                }
            }
        });
    }

    /**
     * Send message synchronously
     */
    public RecordMetadata sendSync(String key, String value) {
        try {
            ProducerRecord<String, String> record = new ProducerRecord<>
(topicName, key, value);
            return producer.send(record).get();
        } catch (Exception e) {
            throw new RuntimeException("Failed to send message synchronously", e);
        }
    }

    /**
     * Send with custom headers
     */
    public void sendWithHeaders(String key, String value, Map<String, String>
headers) {
        ProducerRecord<String, String> record = new ProducerRecord<>(topicName,
key, value);

        // Add headers
        headers.forEach((headerKey, headerValue) ->
            record.headers().add(headerKey, headerValue.getBytes()));

        producer.send(record);
    }

    /**
     * Batch send for high throughput
     */
    public void sendBatch(List<KeyValue<String, String>> messages) {
        for (KeyValue<String, String> message : messages) {
            sendAsync(message.key(), message.value());
        }

        // Ensure all messages are sent
        producer.flush();
    }

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

    // Utility class for key-value pairs
    public static class KeyValue<K, V> {
        private final K key;
        private final V value;

        public KeyValue(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public K key() { return key; }
        public V value() { return value; }
    }
}
```

High-Performance Consumer Implementation

```java
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;
import java.time.Duration;
import java.util.*;
import java.util.concurrent.atomic.AtomicBoolean;

/**
 * High-performance Kafka Consumer with manual offset management
 */
public class HighPerformanceConsumer {

    private final KafkaConsumer<String, String> consumer;
    private final AtomicBoolean running = new AtomicBoolean(true);
    private final String groupId;

    public HighPerformanceConsumer(String groupId) {
        this.groupId = groupId;
        this.consumer = new KafkaConsumer<>(createConsumerProps());
    }

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

        // Connection settings
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
        props.put(ConsumerConfig.CLIENT_ID_CONFIG, "high-perf-consumer-" +
System.currentTimeMillis());

        // Serialization
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        // Performance optimization
        props.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, 65536); // 64KB
        props.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, 100);
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1000);
        props.put(ConsumerConfig.MAX_PARTITION_FETCH_BYTES_CONFIG, 1048576); //
1MB

        // Offset management
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        // Session management
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
        props.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 10000);
        props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000); // 5
minutes

        return props;
    }

    /**
     * Start consuming messages with batch processing
     */
    public void startConsuming(List<String> topics, MessageProcessor processor) {
        consumer.subscribe(topics, new RebalanceListener());

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

                if (!records.isEmpty()) {
                    processBatch(records, processor);
                    commitOffsets(records);
                }

            } catch (Exception e) {
                System.err.println("Error during message consumption: " +
e.getMessage());
                // In production, implement proper error handling and retry logic
            }
        }

        consumer.close();
    }

    /**
     * Process messages in batches for better performance
     */
    private void processBatch(ConsumerRecords<String, String> records,
MessageProcessor processor) {
        Map<TopicPartition, List<ConsumerRecord<String, String>>> partitionRecords
=
            new HashMap<>();

        // Group records by partition to maintain ordering
        for (ConsumerRecord<String, String> record : records) {
            TopicPartition tp = new TopicPartition(record.topic(),
record.partition());
            partitionRecords.computeIfAbsent(tp, k -> new ArrayList<>
()).add(record);
        }

        // Process each partition's records in order
        for (Map.Entry<TopicPartition, List<ConsumerRecord<String, String>>> entry
:
             partitionRecords.entrySet()) {
```

```
            for (ConsumerRecord<String, String> record : entry.getValue()) {
                try {
                    processor.process(record);
                } catch (Exception e) {
                    System.err.printf("Failed to process record: topic=%s,
partition=%d, offset=%d, error=%s%n",
                        record.topic(), record.partition(), record.offset(),
e.getMessage());
                    // Implement dead letter queue or retry logic here
                }
            }
        }
    }

    /**
     * Commit offsets manually for better control
     */
    private void commitOffsets(ConsumerRecords<String, String> records) {
        Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();

        for (ConsumerRecord<String, String> record : records) {
            TopicPartition tp = new TopicPartition(record.topic(),
record.partition());
            offsets.put(tp, new OffsetAndMetadata(record.offset() + 1));
        }

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

    /**
     * Rebalance listener for handling partition assignments
     */
    private class RebalanceListener implements ConsumerRebalanceListener {
        @Override
        public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
            System.out.println("Partitions revoked: " + partitions);
            // Commit any pending offsets before rebalance
            try {
                consumer.commitSync();
            } catch (Exception e) {
                System.err.println("Failed to commit during rebalance: " +
e.getMessage());
            }
        }

        @Override
        public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
            System.out.println("Partitions assigned: " + partitions);
        }
    }

    public void stop() {
        running.set(false);
    }

    // Functional interface for message processing
    @FunctionalInterface
    public interface MessageProcessor {
        void process(ConsumerRecord<String, String> record) throws Exception;
    }
}
```

Streams DSL & Processor API

Modern Kafka Streams Implementation

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

/**
 * Modern Kafka Streams application with advanced processing patterns
 */
public class ModernStreamsApplication {

    private final Properties streamsProps;
    private KafkaStreams streams;

    public ModernStreamsApplication(String applicationId) {
        this.streamsProps = createStreamsProps(applicationId);
    }

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

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

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

        // Performance tuning
        props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 4);
        props.put(StreamsConfig.BUFFER_CONFIG, 32 * 1024 * 1024); // 32MB
        props.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 1000);
        props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 64 * 1024 *
1024); // 64MB

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

        // Error handling

props.put(StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,
            LogAndContinueExceptionHandler.class);

        return props;
    }

    /**
     * Build stream topology with modern patterns
     */
    public Topology buildTopology() {
        StreamsBuilder builder = new StreamsBuilder();

        // Source stream
        KStream<String, String> inputStream = builder.stream("input-topic");

        // Complex processing pipeline
        KStream<String, String> processedStream = inputStream
            // Filter invalid messages
            .filter((key, value) -> value != null && !value.isEmpty())

            // Transform messages
            .mapValues(this::transformMessage)

            // Branch into different streams based on content
            .split(Named.as("branch-"))
            .branch((key, value) -> value.contains("error"),
Branched.as("errors"))
            .branch((key, value) -> value.contains("warning"),
Branched.as("warnings"))
            .defaultBranch(Branched.as("normal"));

        // Process error stream
        processedStream.get("branch-errors")
            .peek((key, value) -> System.out.println("Processing error: " +
value))
            .to("error-topic");

        // Process warning stream with windowed aggregation
        processedStream.get("branch-warnings")
            .groupByKey()
            .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(5)))
            .count()
            .toStream()
            .map((windowedKey, count) -> KeyValue.pair(
                windowedKey.key(),
                String.format("Warning count for %s in window: %d",
windowedKey.key(), count)
            ))
            .to("warning-summary-topic");

        // Process normal stream with stateful processing
        processedStream.get("branch-normal")
            .transform(() -> new StatefulTransformer(), "state-store")
            .to("output-topic");

        // Add state store
        builder.addStateStore(Stores.keyValueStoreBuilder(
            Stores.persistentKeyValueStore("state-store"),
            Serdes.String(),
            Serdes.String()
        ));

        return builder.build();
    }

    /**
     * Custom message transformer
     */
    private String transformMessage(String value) {
        // Implement your transformation logic here
        return value.toUpperCase() + "_PROCESSED";
    }

    /**
     * Stateful transformer using processor API
     */
    private static class StatefulTransformer implements Transformer<String,
String, KeyValue<String, String>> {
        private ProcessorContext context;
        private KeyValueStore<String, String> stateStore;

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

        @Override
        public KeyValue<String, String> transform(String key, String value) {
            // Get previous state
            String previousValue = stateStore.get(key);

            // Update state
            stateStore.put(key, value);

            // Transform based on state
            String transformedValue = previousValue != null ?
```

```
                    value + "_UPDATED" : value + "_NEW";

                return KeyValue.pair(key, transformedValue);
            }

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

        /**
         * Start the streams application
         */
        public void start() {
            Topology topology = buildTopology();
            streams = new KafkaStreams(topology, streamsProps);

            // Add shutdown hook
            Runtime.getRuntime().addShutdownHook(new Thread(() -> {
                System.out.println("Shutting down streams application...");
                streams.close(Duration.ofSeconds(10));
            }));

            // Add state change listener
            streams.setStateListener((newState, oldState) -> {
                System.out.printf("State changed from %s to %s%n", oldState,
newState);
            });

            // Add uncaught exception handler
            streams.setUncaughtExceptionHandler((thread, exception) -> {
                System.err.printf("Uncaught exception in thread %s: %s%n",
                    thread.getName(), exception.getMessage());
                return
StreamsUncaughtExceptionHandler.StreamThreadExceptionResponse.SHUTDOWN_CLIENT;
            });

            streams.start();
            System.out.println("Streams application started");
        }

        public void stop() {
            if (streams != null) {
                streams.close(Duration.ofSeconds(10));
            }
        }

        /**
         * Get stream metrics
         */
        public void printMetrics() {
            if (streams != null) {
                streams.metrics().forEach((metricName, metric) -> {
```

```
                System.out.printf("Metric: %s = %s%n", metricName.name(),
metric.metricValue());
            });
        }
    }
}
```

AdminClient API

Comprehensive Admin Operations

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

/**
 * Comprehensive Kafka AdminClient for cluster management
 */
public class KafkaAdminManager {

    private final AdminClient adminClient;

    public KafkaAdminManager() {
        Properties props = new Properties();
        props.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(AdminClientConfig.CLIENT_ID_CONFIG, "admin-client");
        props.put(AdminClientConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);

        this.adminClient = AdminClient.create(props);
    }

    /**
     * Create topic with detailed configuration
     */
    public void createTopic(String topicName, int partitions, short
replicationFactor) {
        try {
            NewTopic newTopic = new NewTopic(topicName, partitions,
replicationFactor);

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

            newTopic.configs(configs);

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

            System.out.printf("Topic '%s' created successfully with %d
partitions%n",
                topicName, partitions);

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

    /**
     * List all topics with details
     */
    public void listTopics() {
        try {
            ListTopicsResult result = adminClient.listTopics(new
ListTopicsOptions().listInternal(true));
            Set<String> topics = result.names().get();

            System.out.println("Available topics:");
            for (String topic : topics) {
                System.out.println("  - " + topic);
            }

            // Get topic descriptions
            DescribeTopicsResult describeResult =
adminClient.describeTopics(topics);
            Map<String, TopicDescription> descriptions =
describeResult.all().get();

            System.out.println("\nTopic details:");
            for (Map.Entry<String, TopicDescription> entry :
descriptions.entrySet()) {
                TopicDescription desc = entry.getValue();
                System.out.printf("Topic: %s, Partitions: %d, Internal: %s%n",
                    desc.name(), desc.partitions().size(), desc.isInternal());
            }

        } catch (ExecutionException | InterruptedException e) {
            System.err.println("Failed to list topics: " + e.getMessage());
        }
    }

    /**
     * Modify topic configuration
     */
    public void modifyTopicConfig(String topicName, Map<String, String> configs) {
        try {
            ConfigResource resource = new
ConfigResource(ConfigResource.Type.TOPIC, topicName);
```

```
            Map<ConfigResource, Collection<ConfigEntry>> configUpdates = new
HashMap<>();
            Collection<ConfigEntry> entries = new ArrayList<>();

            for (Map.Entry<String, String> config : configs.entrySet()) {
                entries.add(new ConfigEntry(config.getKey(), config.getValue()));
            }

            configUpdates.put(resource, entries);

            IncrementalAlterConfigsResult result =
adminClient.incrementalAlterConfigs(configUpdates);
            result.all().get();

            System.out.printf("Topic '%s' configuration updated%n", topicName);

        } catch (ExecutionException | InterruptedException e) {
            System.err.printf("Failed to modify topic config: %s%n",
e.getMessage());
        }
    }

    /**
     * Monitor consumer groups
     */
    public void monitorConsumerGroups() {
        try {
            ListConsumerGroupsResult groupsResult =
adminClient.listConsumerGroups();
            Collection<ConsumerGroupListing> groups = groupsResult.all().get();

            System.out.println("Consumer Groups:");
            for (ConsumerGroupListing group : groups) {
                System.out.printf("  Group: %s, State: %s%n",
                    group.groupId(), group.state().orElse("UNKNOWN"));

                // Get group details
                monitorSingleConsumerGroup(group.groupId());
            }

        } catch (ExecutionException | InterruptedException e) {
            System.err.println("Failed to monitor consumer groups: " +
e.getMessage());
        }
    }

    /**
     * Monitor single consumer group
     */
    private void monitorSingleConsumerGroup(String groupId) {
        try {
            DescribeConsumerGroupsResult describeResult =
                adminClient.describeConsumerGroups(Arrays.asList(groupId));
            ConsumerGroupDescription description =
describeResult.all().get().get(groupId);

            System.out.printf("    Members: %d, Coordinator: %s%n",
                description.members().size(), description.coordinator().id());

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

            if (!offsets.isEmpty()) {
                System.out.println("    Current Offsets:");
                offsets.forEach((tp, offset) ->
                    System.out.printf("      %s-%d: %d%n",
                        tp.topic(), tp.partition(), offset.offset()));
            }

        } catch (ExecutionException | InterruptedException e) {
            System.err.printf("Failed to get details for group '%s': %s%n",
groupId, e.getMessage());
        }
    }

    /**
     * Delete topic
     */
    public void deleteTopic(String topicName) {
        try {
            DeleteTopicsResult result =
adminClient.deleteTopics(Arrays.asList(topicName));
            result.all().get();

            System.out.printf("Topic '%s' deleted successfully%n", topicName);

        } catch (ExecutionException | InterruptedException e) {
            System.err.printf("Failed to delete topic '%s': %s%n", topicName,
e.getMessage());
        }
    }

    /**
     * Get cluster information
     */
    public void getClusterInfo() {
        try {
            DescribeClusterResult result = adminClient.describeCluster();

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

            Collection<Node> nodes = result.nodes().get();
            System.out.printf("  Brokers (%d):%n", nodes.size());
            for (Node node : nodes) {
                System.out.printf("    Broker %d: %s:%d%n",
                    node.id(), node.host(), node.port());
            }

        } catch (ExecutionException | InterruptedException e) {
            System.err.println("Failed to get cluster info: " + e.getMessage());
        }
    }

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

📦 Serialization

Simple Explanation

Serialization in Kafka converts Java objects to bytes for storage and transmission, while deserialization converts bytes back to objects. Modern serialization formats like Avro and Protobuf provide schema evolution and cross-language compatibility.

Problem It Solves

- **Data Format Consistency**: Ensures all producers and consumers use compatible data formats
- **Schema Evolution**: Allows data structure changes without breaking existing consumers
- **Type Safety**: Provides compile-time type checking and runtime validation
- **Performance**: Efficient binary formats reduce storage and network overhead

JSON, Avro, Protobuf

Comprehensive Serialization Examples

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import io.confluent.kafka.serializers.*;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.Serializer;
import java.util.Map;
import java.util.Properties;

/**
 * Comprehensive serialization examples for different formats
 */
public class SerializationExamples {
    /**
     * JSON Serialization with Jackson
     */
    public static class JsonSerializer<T> implements Serializer<T> {
        private final ObjectMapper objectMapper = new ObjectMapper();

        @Override
        public void configure(Map<String, ?> configs, boolean isKey) {
            // Configuration if needed
        }

        @Override
        public byte[] serialize(String topic, T data) {
            try {
                return objectMapper.writeValueAsBytes(data);
            } catch (Exception e) {
                throw new RuntimeException("Failed to serialize JSON", e);
            }
        }

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

    /**
     * Example DTO for serialization
     */
    public static class UserEvent {
        private String userId;
        private String eventType;
        private long timestamp;
        private Map<String, Object> properties;

        // Constructors, getters, setters
        public UserEvent() {}

        public UserEvent(String userId, String eventType, long timestamp,
Map<String, Object> properties) {
            this.userId = userId;
            this.eventType = eventType;
            this.timestamp = timestamp;
            this.properties = properties;
        }

        // Getters and setters
        public String getUserId() { return userId; }
        public void setUserId(String userId) { this.userId = userId; }

        public String getEventType() { return eventType; }
        public void setEventType(String eventType) { this.eventType = eventType; }
        public long getTimestamp() { return timestamp; }
        public void setTimestamp(long timestamp) { this.timestamp = timestamp; }

        public Map<String, Object> getProperties() { return properties; }
        public void setProperties(Map<String, Object> properties) {
this.properties = properties; }
    }

    /**
     * Avro Producer Configuration
     */
    public static Properties getAvroProducerProps() {
        Properties props = new Properties();

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

        // Schema Registry configuration
        props.put("schema.registry.url", "http://localhost:8081");
        props.put("auto.register.schemas", "true");
        props.put("use.latest.version", "true");

        return props;
    }

    /**
     * Avro serialization example
     */
    public static GenericRecord createAvroRecord() {
        String schemaString = """
            {
              "type": "record",
              "name": "UserEvent",
              "namespace": "com.example.kafka",
              "fields": [
                {"name": "userId", "type": "string"},
                {"name": "eventType", "type": "string"},
                {"name": "timestamp", "type": "long"},
                {"name": "properties", "type": {"type": "map", "values":
"string"}}
              ]
            }
            """;

        Schema schema = new Schema.Parser().parse(schemaString);
        GenericRecord record = new GenericData.Record(schema);

        record.put("userId", "user123");
        record.put("eventType", "click");
        record.put("timestamp", System.currentTimeMillis());
        record.put("properties", Map.of("page", "home", "button", "signup"));
        return record;
    }

    /**
     * Protobuf Producer Configuration
     */
    public static Properties getProtobufProducerProps() {
        Properties props = new Properties();

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

        // Schema Registry configuration
        props.put("schema.registry.url", "http://localhost:8081");
        props.put("auto.register.schemas", "true");

        return props;
    }
}
```

Schema Registry Integration

Schema Registry Management

```
import io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;
import
io.confluent.kafka.schemaregistry.client.rest.exceptions.RestClientException;
import org.apache.avro.Schema;
import java.io.IOException;
import java.util.List;

/**
 * Schema Registry integration and management
 */
public class SchemaRegistryManager {

    private final SchemaRegistryClient schemaRegistryClient;

    public SchemaRegistryManager(String schemaRegistryUrl) {
        this.schemaRegistryClient = new
CachedSchemaRegistryClient(schemaRegistryUrl, 100);
    }

    /**
     * Register a new schema
     */
    public int registerSchema(String subject, String schemaString) {
        try {
            Schema schema = new Schema.Parser().parse(schemaString);
            return schemaRegistryClient.register(subject, schema);
        } catch (IOException | RestClientException e) {
            throw new RuntimeException("Failed to register schema for subject: " +
subject, e);
        }
    }

    /**
     * Get latest schema for subject
     */
    public Schema getLatestSchema(String subject) {
        try {
            return
schemaRegistryClient.getLatestSchemaMetadata(subject).getSchema();
        } catch (IOException | RestClientException e) {
            throw new RuntimeException("Failed to get latest schema for subject: "
+ subject, e);
        }
    }

    /**
     * Check schema compatibility
     */
    public boolean isCompatible(String subject, String schemaString) {
        try {
            Schema schema = new Schema.Parser().parse(schemaString);
            return schemaRegistryClient.testCompatibility(subject, schema);
        } catch (IOException | RestClientException e) {
            System.err.println("Failed to check compatibility: " +
e.getMessage());
            return false;
        }
    }

    /**
     * List all subjects
     */
    public List<String> getAllSubjects() {
        try {
            return schemaRegistryClient.getAllSubjects();
        } catch (IOException | RestClientException e) {
            throw new RuntimeException("Failed to get all subjects", e);
        }
    }

    /**
     * Get schema evolution history
     */
    public void printSchemaEvolution(String subject) {
        try {
            List<Integer> versions = schemaRegistryClient.getAllVersions(subject);
            System.out.printf("Schema evolution for subject '%s':%n", subject);
            for (Integer version : versions) {
                Schema schema = schemaRegistryClient.getByVersion(subject,
version, false);
                System.out.printf("  Version %d: %s%n", version,
schema.getName());
            }

        } catch (IOException | RestClientException e) {
            System.err.println("Failed to get schema evolution: " +
e.getMessage());
        }
    }

    /**
     * Example: Schema evolution scenarios
     */
    public void demonstrateSchemaEvolution() {
        String subject = "user-event-value";

        // Version 1: Initial schema
        String schemaV1 = """
            {
              "type": "record",
              "name": "UserEvent",
              "namespace": "com.example.kafka",
              "fields": [
                {"name": "userId", "type": "string"},
                {"name": "eventType", "type": "string"},
                {"name": "timestamp", "type": "long"}
              ]
            }
            """;

        // Version 2: Add optional field (backward compatible)
        String schemaV2 = """
            {
              "type": "record",
              "name": "UserEvent",
              "namespace": "com.example.kafka",
              "fields": [
                {"name": "userId", "type": "string"},
                {"name": "eventType", "type": "string"},
                {"name": "timestamp", "type": "long"},
                {"name": "sessionId", "type": ["null", "string"], "default": null}
              ]
            }
            """;

        // Version 3: Add field with default (backward compatible)
        String schemaV3 = """
            {
              "type": "record",
              "name": "UserEvent",
              "namespace": "com.example.kafka",
              "fields": [
                {"name": "userId", "type": "string"},
                {"name": "eventType", "type": "string"},
                {"name": "timestamp", "type": "long"},
                {"name": "sessionId", "type": ["null", "string"], "default":
null},
                {"name": "deviceType", "type": "string", "default": "unknown"}
              ]
            }
            """;

        System.out.println("Demonstrating schema evolution:");

        System.out.println("Registering V1...");
        int v1Id = registerSchema(subject, schemaV1);
        System.out.println("V1 registered with ID: " + v1Id);

        System.out.println("Checking V2 compatibility...");
        boolean v2Compatible = isCompatible(subject, schemaV2);
        System.out.println("V2 compatible: " + v2Compatible);

        if (v2Compatible) {
            int v2Id = registerSchema(subject, schemaV2);
            System.out.println("V2 registered with ID: " + v2Id);
        }

        System.out.println("Checking V3 compatibility...");
        boolean v3Compatible = isCompatible(subject, schemaV3);
        System.out.println("V3 compatible: " + v3Compatible);

        if (v3Compatible) {
            int v3Id = registerSchema(subject, schemaV3);
            System.out.println("V3 registered with ID: " + v3Id);
        }

        printSchemaEvolution(subject);
    }
}
```

SerDes in Streams

Custom SerDes for Kafka Streams

```
import org.apache.kafka.common.serialization.Deserializer;
import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serializer;
import org.apache.kafka.streams.kstream.Consumed;
import org.apache.kafka.streams.kstream.Produced;
import com.fasterxml.jackson.databind.ObjectMapper;
```

```java
import java.util.Map;

/**
 * Custom SerDes for Kafka Streams
 */
public class CustomSerDes {

    /**
     * Generic JSON Serde
     */
    public static class JsonSerde<T> implements Serde<T> {
        private final Class<T> targetType;
        private final ObjectMapper objectMapper;

        public JsonSerde(Class<T> targetType) {
            this.targetType = targetType;
            this.objectMapper = new ObjectMapper();
        }

        @Override
        public Serializer<T> serializer() {
            return new JsonSerializer<>(objectMapper);
        }

        @Override
        public Deserializer<T> deserializer() {
            return new JsonDeserializer<>(targetType, objectMapper);
        }

        @Override
        public void configure(Map<String, ?> configs, boolean isKey) {
            // Configuration if needed
        }

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

    /**
     * JSON Serializer
     */
    private static class JsonSerializer<T> implements Serializer<T> {
        private final ObjectMapper objectMapper;

        JsonSerializer(ObjectMapper objectMapper) {
            this.objectMapper = objectMapper;
        }

        @Override
        public byte[] serialize(String topic, T data) {
            try {
                return objectMapper.writeValueAsBytes(data);
            } catch (Exception e) {
                throw new RuntimeException("Failed to serialize JSON", e);
            }
        }
    }

    /**
     * JSON Deserializer
     */
    private static class JsonDeserializer<T> implements Deserializer<T> {
        private final Class<T> targetType;
        private final ObjectMapper objectMapper;

        JsonDeserializer(Class<T> targetType, ObjectMapper objectMapper) {
            this.targetType = targetType;
            this.objectMapper = objectMapper;
        }

        @Override
        public T deserialize(String topic, byte[] data) {
            try {
                return objectMapper.readValue(data, targetType);
            } catch (Exception e) {
                throw new RuntimeException("Failed to deserialize JSON", e);
            }
        }
    }

    /**
     * Streams application using custom SerDes
     */
    public static class StreamsWithCustomSerDes {

        public Topology buildTopology() {
            StreamsBuilder builder = new StreamsBuilder();

            // Create custom SerDes
            Serde<UserEvent> userEventSerde = new JsonSerde<>(UserEvent.class);
            Serde<EventSummary> eventSummarySerde = new JsonSerde<>
(EventSummary.class);

            // Input stream with custom SerDes
            KStream<String, UserEvent> userEvents = builder.stream("user-events",
                Consumed.with(Serdes.String(), userEventSerde));

            // Process and aggregate
            KTable<String, EventSummary> eventSummaries = userEvents
                .groupBy((key, event) -> event.getEventType())
                .aggregate(
                    EventSummary::new,
                    (key, event, summary) -> summary.addEvent(event),
                    Materialized.with(Serdes.String(), eventSummarySerde)
                );
            // Output with custom SerDes
            eventSummaries.toStream()
                .to("event-summaries", Produced.with(Serdes.String(),
eventSummarySerde));

            return builder.build();
        }
    }

    /**
     * Event summary DTO
     */
    public static class EventSummary {
        private String eventType;
        private long count;
        private long lastEventTime;

        public EventSummary() {}

        public EventSummary addEvent(UserEvent event) {
            this.eventType = event.getEventType();
            this.count++;
            this.lastEventTime = event.getTimestamp();
            return this;
        }

        // Getters and setters
        public String getEventType() { return eventType; }
        public void setEventType(String eventType) { this.eventType = eventType; }

        public long getCount() { return count; }
        public void setCount(long count) { this.count = count; }

        public long getLastEventTime() { return lastEventTime; }
        public void setLastEventTime(long lastEventTime) { this.lastEventTime =
lastEventTime; }
    }
}
```

🚀 Modern Java Features

Records for DTOs (Java 16+)

Using Records for Kafka Messages

```
import com.fasterxml.jackson.annotation.JsonProperty;
import java.time.Instant;
import java.util.Map;
import java.util.List;
```

```
/**
 * Modern Java Records for Kafka message DTOs
 */
public class KafkaRecords {

    /**
     * Simple user event record
     */
    public record UserEvent(
        @JsonProperty("user_id") String userId,
        @JsonProperty("event_type") String eventType,
        @JsonProperty("timestamp") Instant timestamp,
        @JsonProperty("properties") Map<String, Object> properties
    ) {
        // Compact constructor for validation
        public UserEvent {
            if (userId == null || userId.isBlank()) {
                throw new IllegalArgumentException("User ID cannot be null or
blank");
            }
            if (eventType == null || eventType.isBlank()) {
                throw new IllegalArgumentException("Event type cannot be null or
blank");
            }
            if (timestamp == null) {
                timestamp = Instant.now();
            }
            if (properties == null) {
                properties = Map.of();
            }
        }

        // Custom methods can be added to records
        public boolean isImportantEvent() {
            return "purchase".equals(eventType) || "signup".equals(eventType);
        }

        public String getEventCategory() {
            return switch (eventType) {
                case "click", "view", "scroll" -> "engagement";
                case "purchase", "add_to_cart" -> "transaction";
                case "signup", "login", "logout" -> "authentication";
                default -> "other";
            };
        }
    }

    /**
     * Order event with nested records
     */
    public record OrderEvent(
        String orderId,
        String userId,
        Instant orderTime,
        OrderStatus status,
        List<OrderItem> items,
        PaymentInfo payment
    ) {
        public record OrderItem(
            String productId,
            String productName,
            int quantity,
            double price
        ) {}

        public record PaymentInfo(
            String method,
            double amount,
            String currency
        ) {}

        public enum OrderStatus {
            PENDING, CONFIRMED, SHIPPED, DELIVERED, CANCELLED
        }

        // Computed properties
        public double totalAmount() {
            return items.stream()
                .mapToDouble(item -> item.price * item.quantity)
                .sum();
        }

        public int totalItems() {
            return items.stream()
                .mapToInt(OrderItem::quantity)
                .sum();
        }
    }

    /**
     * Event envelope pattern with records
     */
    public record EventEnvelope<T>(
        String eventId,
        String eventType,
        String source,
        Instant timestamp,
        String schemaVersion,
        T payload
    ) {
        public EventEnvelope(String eventType, String source, T payload) {
            this(
                java.util.UUID.randomUUID().toString(),
                eventType,
                source,
                Instant.now(),
                "1.0",
                payload
            );
        }
    }

    /**
     * Producer using records
     */
    public static class RecordProducer {
        private final KafkaProducer<String, UserEvent> producer;
        private final String topicName;

        public RecordProducer(String topicName) {
            this.topicName = topicName;

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

            this.producer = new KafkaProducer<>(props);
        }

        public void sendUserEvent(UserEvent event) {
            ProducerRecord<String, UserEvent> record =
                new ProducerRecord<>(topicName, event.userId(), event);

            producer.send(record, (metadata, exception) -> {
                if (exception != null) {
                    System.err.printf("Failed to send event: %s%n",
exception.getMessage());
                } else {
                    System.out.printf("Sent event: %s to partition %d, offset
%d%n",
                        event.eventType(), metadata.partition(),
metadata.offset());
                }
            });
        }

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

    /**
     * Example usage of records
     */
    public static void demonstrateRecords() {
        // Creating user events with records
        var clickEvent = new UserEvent(
            "user123",
            "click",
            Instant.now(),
            Map.of("page", "home", "element", "signup-button")
        );

        var purchaseEvent = new UserEvent(
            "user456",
            "purchase",
            Instant.now(),
            Map.of("product_id", "prod789", "amount", "99.99")
        );

        System.out.println("Click event category: " +
clickEvent.getEventCategory());
        System.out.println("Purchase event important: " +
purchaseEvent.isImportantEvent());

        // Creating order events
        var orderEvent = new OrderEvent(
            "order123",
            "user456",
            Instant.now(),
            OrderEvent.OrderStatus.CONFIRMED,
            List.of(
                new OrderEvent.OrderItem("prod1", "Widget", 2, 25.00),
                new OrderEvent.OrderItem("prod2", "Gadget", 1, 49.99)
            ),
            new OrderEvent.PaymentInfo("credit_card", 99.99, "USD")
        );

        System.out.println("Order total: $" + orderEvent.totalAmount());
        System.out.println("Total items: " + orderEvent.totalItems());

        // Event envelope pattern
        var envelope = new EventEnvelope<>("user.click", "web-app", clickEvent);
        System.out.println("Event envelope: " + envelope.eventId());
    }
}
```

Sealed Classes for Event Hierarchies

Event Hierarchy with Sealed Classes

```
/**
 * Sealed classes for type-safe event hierarchies (Java 17+)
 */
public class EventHierarchy {

    /**
     * Base sealed class for all events
     */
    public sealed interface KafkaEvent
```

```
        permits UserEvent, OrderEvent, SystemEvent {

        String eventId();
        Instant timestamp();
        String source();
    }

    /**
     * User-related events
     */
    public sealed interface UserEvent extends KafkaEvent
        permits UserRegistered, UserLoggedIn, UserProfileUpdated {
        String userId();
    }

    public record UserRegistered(
        String eventId,
        Instant timestamp,
        String source,
        String userId,
        String email,
        String registrationMethod
    ) implements UserEvent {}

    public record UserLoggedIn(
        String eventId,
        Instant timestamp,
        String source,
        String userId,
        String sessionId,
        String loginMethod
    ) implements UserEvent {}

    public record UserProfileUpdated(
        String eventId,
        Instant timestamp,
        String source,
        String userId,
        Map<String, Object> updatedFields
    ) implements UserEvent {}

    /**
     * Order-related events
     */
    public sealed interface OrderEvent extends KafkaEvent
        permits OrderCreated, OrderStatusChanged, OrderCancelled {
        String orderId();
        String userId();
    }

    public record OrderCreated(
        String eventId,
        Instant timestamp,
        String source,
        String orderId,
        String userId,
        List<OrderItem> items,
        double totalAmount
    ) implements OrderEvent {

        public record OrderItem(String productId, int quantity, double price) {}
    }

    public record OrderStatusChanged(
        String eventId,
        Instant timestamp,
        String source,
        String orderId,
        String userId,
        String previousStatus,
        String newStatus
    ) implements OrderEvent {}

    public record OrderCancelled(
        String eventId,
        Instant timestamp,
        String source,
        String orderId,
        String userId,
        String reason
    ) implements OrderEvent {}

    /**
     * System events
     */
    public sealed interface SystemEvent extends KafkaEvent
        permits ServiceStarted, ServiceStopped, HealthCheck {
        String serviceName();
    }

    public record ServiceStarted(
        String eventId,
        Instant timestamp,
        String source,
        String serviceName,
        String version
    ) implements SystemEvent {}

    public record ServiceStopped(
        String eventId,
        Instant timestamp,
        String source,
        String serviceName,
        String reason
    ) implements SystemEvent {}

    public record HealthCheck(
        String eventId,
        Instant timestamp,
        String source,
        String serviceName,
        String status,
        Map<String, Object> metrics
    ) implements SystemEvent {}

    /**
     * Event processor using pattern matching and sealed classes
     */
    public static class EventProcessor {

        public void processEvent(KafkaEvent event) {
            // Pattern matching with sealed classes (Java 17+)
            switch (event) {
                case UserEvent userEvent -> processUserEvent(userEvent);
                case OrderEvent orderEvent -> processOrderEvent(orderEvent);
                case SystemEvent systemEvent -> processSystemEvent(systemEvent);
            }
        }

        private void processUserEvent(UserEvent event) {
            switch (event) {
                case UserRegistered registered -> {
                    System.out.printf("New user registered: %s (%s)%n",
                        registered.userId(), registered.email());
                    // Send welcome email, update analytics, etc.
                }
                case UserLoggedIn loggedIn -> {
                    System.out.printf("User logged in: %s with session %s%n",
                        loggedIn.userId(), loggedIn.sessionId());
                    // Update session tracking, security monitoring, etc.
                }
                case UserProfileUpdated updated -> {
                    System.out.printf("User profile updated: %s, fields: %s%n",
                        updated.userId(), updated.updatedFields().keySet());
                    // Sync with CRM, update recommendations, etc.
                }
            }
        }

        private void processOrderEvent(OrderEvent event) {
            switch (event) {
                case OrderCreated created -> {
                    System.out.printf("Order created: %s for user %s, total:
$%.2f%n",
                        created.orderId(), created.userId(),
created.totalAmount());
                    // Inventory reservation, payment processing, etc.
                }
                case OrderStatusChanged statusChanged -> {
                    System.out.printf("Order %s status: %s -> %s%n",
                        statusChanged.orderId(),
                        statusChanged.previousStatus(),
                        statusChanged.newStatus());
                    // Notifications, shipping updates, etc.
                }
                case OrderCancelled cancelled -> {
                    System.out.printf("Order cancelled: %s, reason: %s%n",
                        cancelled.orderId(), cancelled.reason());
                    // Refund processing, inventory release, etc.
                }
            }
        }

        private void processSystemEvent(SystemEvent event) {
            switch (event) {
                case ServiceStarted started -> {
                    System.out.printf("Service started: %s v%s%n",
                        started.serviceName(), started.version());
                    // Service registry update, monitoring setup, etc.
                }
                case ServiceStopped stopped -> {
                    System.out.printf("Service stopped: %s, reason: %s%n",
                        stopped.serviceName(), stopped.reason());
                    // Cleanup, alerting, etc.
                }
                case HealthCheck healthCheck -> {
                    System.out.printf("Health check: %s status %s%n",
                        healthCheck.serviceName(), healthCheck.status());
                    // Monitoring dashboards, alerting, etc.
                }
            }
        }
    }

    /**
     * Event factory for creating events
     */
    public static class EventFactory {

        public static UserRegistered createUserRegistered(String userId, String
email, String method) {
            return new UserRegistered(
                UUID.randomUUID().toString(),
                Instant.now(),
                "user-service",
                userId,
                email,
                method
            );
        }

        public static OrderCreated createOrderCreated(String orderId, String
userId,
                                                    List<OrderCreated.OrderItem>
items) {
            double total = items.stream()
```

```
                .mapToDouble(item -> item.price() * item.quantity())
                .sum();

            return new OrderCreated(
                UUID.randomUUID().toString(),
                Instant.now(),
                "order-service",
                orderId,
                userId,
                items,
                total
            );
        }

        public static HealthCheck createHealthCheck(String serviceName, String 
status,
                                                    Map<String, Object> metrics) {
            return new HealthCheck(
                UUID.randomUUID().toString(),
                Instant.now(),
                serviceName,
                serviceName,
                status,
                metrics
            );
        }
    }

    /**
     * Streams processor for event hierarchy
     */
    public static class EventStreamsProcessor {

        public Topology buildTopology() {
            StreamsBuilder builder = new StreamsBuilder();

            // Input stream of all events
            KStream<String, KafkaEvent> eventStream = builder.stream("all-
events");

            // Branch events by type using sealed classes
            Map<String, KStream<String, KafkaEvent>> branches = eventStream
                .split(Named.as("event-type-"))
                .branch((key, event) -> event instanceof UserEvent, 
Branched.as("user"))
                .branch((key, event) -> event instanceof OrderEvent, 
Branched.as("order"))
                .branch((key, event) -> event instanceof SystemEvent, 
Branched.as("system"))
                .noDefaultBranch();

            // Process user events
            branches.get("event-type-user")
                .mapValues(event -> (UserEvent) event)
```

```
                    .foreach((key, event) -> processUserEventForStreams(event));

                // Process order events
                branches.get("event-type-order")
                    .mapValues(event -> (OrderEvent) event)
                    .foreach((key, event) -> processOrderEventForStreams(event));

                // Process system events
                branches.get("event-type-system")
                    .mapValues(event -> (SystemEvent) event)
                    .foreach((key, event) -> processSystemEventForStreams(event));

                return builder.build();
            }

            private void processUserEventForStreams(UserEvent event) {
                // Process user events in streams context
            }

            private void processOrderEventForStreams(OrderEvent event) {
                // Process order events in streams context
            }

            private void processSystemEventForStreams(SystemEvent event) {
                // Process system events in streams context
            }
        }
    }
```

Virtual Threads (Java 21/25)

Virtual Threads for Kafka Processing

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.time.Duration;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.atomic.AtomicLong;

/**
 * Virtual Threads implementation for high-concurrency Kafka processing
 */
public class VirtualThreadsKafka {

    /**
     * Virtual Thread-based Kafka Consumer
     */
    public static class VirtualThreadConsumer {

        private final KafkaConsumer<String, String> consumer;
        private final ExecutorService virtualThreadExecutor;
        private final AtomicLong processedCount = new AtomicLong(0);
        private volatile boolean running = true;

        public VirtualThreadConsumer(String groupId) {
            this.consumer = new KafkaConsumer<>(createConsumerProps(groupId));

            // Create virtual thread executor (Java 21+)
            this.virtualThreadExecutor =
Executors.newVirtualThreadPerTaskExecutor();
        }

        private Properties createConsumerProps(String groupId) {
            Properties props = new Properties();
            props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
            props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
            props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
            props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1000);
            props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
            return props;
        }

        /**
         * Start consuming with virtual threads for each message
         */
        public void startConsuming(List<String> topics) {
            consumer.subscribe(topics);

            // Main consumer loop runs on platform thread
            Thread consumerThread = Thread.ofPlatform()
                .name("consumer-main")
                .start(() -> {
                    while (running) {
                        try {
                            ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));

                            if (!records.isEmpty()) {
                                processRecordsWithVirtualThreads(records);
                            }

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

            // Add shutdown hook
            Runtime.getRuntime().addShutdownHook(new Thread(() -> {
                running = false;
                try {
                    consumerThread.join(5000);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
                virtualThreadExecutor.shutdown();
                consumer.close();
            }));
        }

        /**
         * Process each record on its own virtual thread
         */
        private void processRecordsWithVirtualThreads(ConsumerRecords<String,
String> records) {
            List<CompletableFuture<Void>> futures = new ArrayList<>();

            for (ConsumerRecord<String, String> record : records) {
                CompletableFuture<Void> future = CompletableFuture.runAsync(() ->
{
                    processRecord(record);
                }, virtualThreadExecutor);

                futures.add(future);
            }

            // Wait for all processing to complete before committing
            CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
                .thenRun(() -> {
                    try {
                        consumer.commitSync();
                        System.out.printf("Committed batch of %d records%n",
records.count());
                    } catch (Exception e) {
                        System.err.println("Failed to commit offsets: " +
e.getMessage());
                    }
                })
                .join();
        }

        /**
         * Process individual record (potentially blocking I/O operations)
         */
        private void processRecord(ConsumerRecord<String, String> record) {
            try {
                // Simulate I/O-bound processing that benefits from virtual
threads

                // 1. Database lookup
                Thread.sleep(10); // Simulate DB query

                // 2. External API call
                Thread.sleep(20); // Simulate HTTP request
                // 3. Business logic processing
                String processedValue = processBusinessLogic(record.value());

                // 4. Store result
                Thread.sleep(5); // Simulate cache/DB write

                long count = processedCount.incrementAndGet();
                if (count % 1000 == 0) {
                    System.out.printf("Processed %d messages%n", count);
                }

            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Processing interrupted for record: " +
record.offset());
            } catch (Exception e) {
                System.err.printf("Failed to process record %d: %s%n",
record.offset(), e.getMessage());
            }
        }

        private String processBusinessLogic(String value) {
            // Simulate complex business logic
            return value.toUpperCase() + "_PROCESSED_" +
System.currentTimeMillis();
        }

        public long getProcessedCount() {
            return processedCount.get();
        }
    }

    /**
     * Virtual Thread-based Kafka Producer for high-throughput scenarios
     */
    public static class VirtualThreadProducer {

        private final KafkaProducer<String, String> producer;
        private final ExecutorService virtualThreadExecutor;
        private final AtomicLong sentCount = new AtomicLong(0);

        public VirtualThreadProducer() {
            this.producer = new KafkaProducer<>(createProducerProps());
            this.virtualThreadExecutor =
Executors.newVirtualThreadPerTaskExecutor();
        }

        private Properties createProducerProps() {
            Properties props = new Properties();
            props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
            props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
```

```
            props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536);
            props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
            props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");
            return props;
        }

        /**
         * Send messages using virtual threads for data preparation
         */
        public CompletableFuture<Void> sendBatchAsync(String topic, List<String>
keys) {
            List<CompletableFuture<RecordMetadata>> futures = new ArrayList<>();

            for (String key : keys) {
                CompletableFuture<RecordMetadata> future = CompletableFuture
                    .supplyAsync(() -> prepareMessageData(key),
virtualThreadExecutor)
                    .thenCompose(messageData -> sendMessage(topic, key,
messageData));

                futures.add(future);
            }

            return CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0]))
                .thenRun(() -> {
                    long count = sentCount.addAndGet(keys.size());
                    System.out.printf("Sent batch of %d messages (total: %d)%n",
keys.size(), count);
                });
        }

        /**
         * Prepare message data (potentially involving I/O operations)
         */
        private String prepareMessageData(String key) {
            try {
                // Simulate data enrichment from external sources
                Thread.sleep(5); // Database lookup
                Thread.sleep(10); // External API call
                Thread.sleep(2); // Data transformation

                return String.format("
{'key':'%s','timestamp':%d,'data':'processed'}",
                    key, System.currentTimeMillis());

            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException("Data preparation interrupted", e);
            }
        }

        /**
         * Send individual message
         */
        private CompletableFuture<RecordMetadata> sendMessage(String topic, String
key, String value) {
            ProducerRecord<String, String> record = new ProducerRecord<>(topic,
key, value);

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

            producer.send(record, (metadata, exception) -> {
                if (exception != null) {
                    future.completeExceptionally(exception);
                } else {
                    future.complete(metadata);
                }
            });

            return future;
        }

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

        public long getSentCount() {
            return sentCount.get();
        }
    }

    /**
     * Performance comparison: Platform threads vs Virtual threads
     */
    public static class PerformanceComparison {

        public static void compareThreadingModels() {
            System.out.println("=== Threading Model Performance Comparison ===");

            // Test with platform threads
            long platformThreadTime = measureProcessingTime(() -> {
                try (ExecutorService executor = Executors.newFixedThreadPool(200))
{
                    processWithExecutor(executor, 10000);
                }
            });

            // Test with virtual threads
            long virtualThreadTime = measureProcessingTime(() -> {
                try (ExecutorService executor =
Executors.newVirtualThreadPerTaskExecutor()) {
                    processWithExecutor(executor, 10000);
                }
            });

            System.out.printf("Platform threads: %d ms%n", platformThreadTime);
            System.out.printf("Virtual threads: %d ms%n", virtualThreadTime);
            System.out.printf("Virtual threads are %.2fx faster%n",
                (double) platformThreadTime / virtualThreadTime);
        }

        private static void processWithExecutor(ExecutorService executor, int
taskCount) {
            List<CompletableFuture<Void>> futures = new ArrayList<>();

            for (int i = 0; i < taskCount; i++) {
                CompletableFuture<Void> future = CompletableFuture.runAsync(() ->
{
                    try {
                        // Simulate I/O-bound work
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }, executor);

                futures.add(future);
            }

            CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0])).join();
        }

        private static long measureProcessingTime(Runnable task) {
            long startTime = System.currentTimeMillis();
            task.run();
            return System.currentTimeMillis() - startTime;
        }
    }

    /**
     * Example usage
     */
    public static void main(String[] args) {
        // Virtual thread consumer example
        VirtualThreadConsumer consumer = new VirtualThreadConsumer("virtual-
thread-group");
        consumer.startConsuming(List.of("test-topic"));

        // Virtual thread producer example
        VirtualThreadProducer producer = new VirtualThreadProducer();
        List<String> keys = IntStream.range(0, 1000)
            .mapToObj(i -> "key-" + i)
            .collect(Collectors.toList());

        producer.sendBatchAsync("test-topic", keys)
            .thenRun(() -> System.out.println("Batch sending completed"));

        // Performance comparison
        PerformanceComparison.compareThreadingModels();
        }
    }
```

⚖️ Comparisons & Trade-offs

Client API Comparison

Feature	Producer API	Consumer API	Streams API	AdminClient
Use Case	Send messages	Receive messages	Stream processing	Cluster management
Complexity	Low	Medium	High	Medium
State Management	Stateless	Offset tracking	Stateful	Stateless
Throughput	Very High	High	Medium-High	Low
Latency	Very Low	Low	Medium	N/A
Scalability	Horizontal	Horizontal	Horizontal	Single instance

Serialization Format Comparison

Format	Size	Performance	Schema Evolution	Cross-Language	Ecosystem
JSON	Large	Slow	Manual	Excellent	Excellent
Avro	Small	Fast	Excellent	Good	Good
Protobuf	Small	Very Fast	Good	Excellent	Excellent
Java Serialization	Large	Slow	Poor	Poor	Java Only

Threading Model Comparison

Model	Concurrency	Memory Usage	Context Switching	I/O Blocking	Best For
Single Thread	Low	Low	None	Blocks all	Simple processing
Platform Threads	Medium	High	Expensive	Blocks thread	CPU-intensive
Virtual Threads	Very High	Low	Cheap	Non-blocking	I/O-intensive
Async/Reactive	High	Medium	Complex	Non-blocking	High throughput

---

🚨 Common Pitfalls & Best Practices

Producer Pitfalls

❌ Common Mistakes

```java
// DON'T - Blocking the main thread
public void badProducerPattern() {
    KafkaProducer<String, String> producer = new KafkaProducer<>(props);

    for (int i = 0; i < 1000; i++) {
        try {
            // This blocks the main thread for each send
            producer.send(record).get(); // DON'T DO THIS
        } catch (Exception e) {
            // Handle error
        }
    }
}

// DON'T - Not handling serialization errors
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "InvalidSerializer");
// Will fail at runtime with unclear error messages
```

✅ Best Practices

```java
// DO - Use async sends with callbacks
public void goodProducerPattern() {
    KafkaProducer<String, String> producer = new KafkaProducer<>(props);

    for (int i = 0; i < 1000; i++) {
        producer.send(record, (metadata, exception) -> {
            if (exception != null) {
                // Handle error asynchronously
                handleError(exception);
            } else {
                // Handle success
                handleSuccess(metadata);
            }
        });
    }

    // Ensure all messages are sent before closing
    producer.flush();
    producer.close();
}
```

Consumer Pitfalls

❌ Common Mistakes

```java
// DON'T - Long processing without commit control
public void badConsumerPattern() {
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

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

        for (ConsumerRecord<String, String> record : records) {
            // Long processing without offset management
            processForLongTime(record); // May cause rebalancing issues
        }
        // Auto-commit may commit before processing is complete
    }
}
```

☑ Best Practices

```java
// DO - Manual offset management with proper error handling
public void goodConsumerPattern() {
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

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

            for (ConsumerRecord<String, String> record : records) {
                try {
                    processRecord(record);
                    // Commit only after successful processing
                    consumer.commitSync(Collections.singletonMap(
                        new TopicPartition(record.topic(), record.partition()),
                        new OffsetAndMetadata(record.offset() + 1)
                    ));
                } catch (Exception e) {
                    // Handle processing error, maybe send to DLQ
                    handleProcessingError(record, e);
                }
            }
        } catch (WakeupException e) {
            // Shutdown signal received
            break;
        }
    }
}
```

Serialization Pitfalls

❌ Common Mistakes

```
// DON'T - Using Java serialization in production
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.ObjectSerializer");
// Breaks with schema changes, not cross-language compatible

// DON'T - Not handling schema evolution
// Old version: {"name": "string"}
// New version: {"name": "string", "age": "int"}
// Will break without proper schema evolution strategy
```

✅ Best Practices

```
// DO - Use Schema Registry with proper evolution strategy
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", "http://localhost:8081");
props.put("auto.register.schemas", "false"); // Control schema registration
props.put("use.latest.version", "true");

// DO - Design schemas for evolution
String evolvedSchema = """
    {
      "type": "record",
      "name": "User",
      "fields": [
        {"name": "name", "type": "string"},
        {"name": "age", "type": ["null", "int"], "default": null}
      ]
    }
    """;
```

Best Practices Summary

✅ Producer Best Practices

1. **Use async sends** with proper callback handling
2. **Enable idempotency** for exactly-once semantics
3. **Configure batching** for optimal throughput
4. **Handle serialization errors** gracefully
5. **Monitor producer metrics** for performance tuning

✅ Consumer Best Practices

1. **Use manual offset management** for critical applications
2. **Handle rebalancing** properly with listeners

3. **Implement proper error handling** and DLQ patterns
4. **Monitor consumer lag** continuously
5. **Use appropriate fetch sizes** for your workload

✅ Streams Best Practices

1. **Design stateless transformations** when possible
2. **Use appropriate SerDes** for your data
3. **Handle exceptions** at the right level
4. **Monitor stream metrics** and state stores
5. **Test with TopologyTestDriver** before deployment

✅ Modern Java Best Practices

1. **Use Records** for immutable DTOs
2. **Leverage Sealed Classes** for type-safe hierarchies
3. **Consider Virtual Threads** for I/O-heavy workloads
4. **Use pattern matching** with sealed classes
5. **Keep up with Java evolution** for new features

🌍 Real-World Use Cases

E-commerce Platform

```java
/**
 * E-commerce platform using modern Java features
 */
public class ECommerceKafkaIntegration {

    // Event hierarchy with sealed classes
    public sealed interface ECommerceEvent permits OrderEvent, InventoryEvent,
UserEvent {
        String eventId();
        Instant timestamp();
    }

    // Records for different event types
    public record OrderEvent(
        String eventId,
        Instant timestamp,
        String orderId,
        String userId,
        OrderStatus status,
        List<OrderItem> items
    ) implements ECommerceEvent {

        public record OrderItem(String productId, int quantity, BigDecimal price)
{}
        public enum OrderStatus { CREATED, PAID, SHIPPED, DELIVERED, CANCELLED }
    }

    // Virtual threads for high-concurrency order processing
    public static class OrderProcessor {
        private final ExecutorService virtualThreadExecutor =
            Executors.newVirtualThreadPerTaskExecutor();

        public void processOrdersWithVirtualThreads(List<OrderEvent> orders) {
            List<CompletableFuture<Void>> futures = orders.stream()
                .map(order -> CompletableFuture.runAsync(() -> {
                    // I/O-intensive operations benefit from virtual threads
                    validatePayment(order);
                    checkInventory(order);
                    updateOrderStatus(order);
                }, virtualThreadExecutor))
                .toList();

            CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0])).join();
        }

        private void validatePayment(OrderEvent order) {
            // Simulate payment validation API call
            try { Thread.sleep(50); } catch (InterruptedException e) { /* handle
*/ }
        }

        private void checkInventory(OrderEvent order) {
            // Simulate inventory check
            try { Thread.sleep(30); } catch (InterruptedException e) { /* handle
*/ }
        }

        private void updateOrderStatus(OrderEvent order) {
            // Simulate database update
            try { Thread.sleep(20); } catch (InterruptedException e) { /* handle
*/ }
        }
    }
}
```

Financial Trading System

```
/**
 * High-frequency trading system with ultra-low latency requirements
 */
public class TradingSystemKafka {

    // Records for trading events
    public record TradeEvent(
        String tradeId,
        String symbol,
        TradeType type,
        BigDecimal price,
        long quantity,
        Instant timestamp
    ) {
        public enum TradeType { BUY, SELL }

        public BigDecimal getValue() {
            return price.multiply(BigDecimal.valueOf(quantity));
        }
    }

    // Ultra-low latency producer
    public static class LowLatencyTradeProducer {
        private final KafkaProducer<String, TradeEvent> producer;

        public LowLatencyTradeProducer() {
            Properties props = new Properties();
            props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
            props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

            // Ultra-low latency settings
            props.put(ProducerConfig.BATCH_SIZE_CONFIG, 0); // No batching
            props.put(ProducerConfig.LINGER_MS_CONFIG, 0);
            props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "none");
            props.put(ProducerConfig.ACKS_CONFIG, "1"); // Leader only

            this.producer = new KafkaProducer<>(props);
        }

        public void sendTradeSync(TradeEvent trade) {
            try {
                ProducerRecord<String, TradeEvent> record =
                    new ProducerRecord<>("trades", trade.symbol(), trade);
                producer.send(record).get(); // Synchronous for ultra-low latency
            } catch (Exception e) {
                throw new RuntimeException("Failed to send trade", e);
            }
        }
    }
}
```

IoT Data Processing

```
/**
 * IoT sensor data processing with Kafka Streams
 */
```

```
public class IoTDataProcessing {

    // Sensor data record
    public record SensorReading(
        String deviceId,
        String sensorType,
        double value,
        String unit,
        Instant timestamp,
        GeoLocation location
    ) {
        public record GeoLocation(double latitude, double longitude) {}

        public boolean isAnomaly() {
            return switch (sensorType) {
                case "temperature" -> value < -40 || value > 80;
                case "humidity" -> value < 0 || value > 100;
                case "pressure" -> value < 800 || value > 1200;
                default -> false;
            };
        }
    }

    // Streams processing with modern Java features
    public static class SensorDataStreams {

        public Topology buildTopology() {
            StreamsBuilder builder = new StreamsBuilder();

            KStream<String, SensorReading> sensorStream = builder.stream("sensor-
data");

            // Use pattern matching for processing different sensor types
            sensorStream
                .filter((key, reading) -> reading != null)
                .foreach((key, reading) -> {
                    switch (reading.sensorType()) {
                        case "temperature" -> processTemperature(reading);
                        case "humidity" -> processHumidity(reading);
                        case "pressure" -> processPressure(reading);
                        default -> System.out.println("Unknown sensor type: " +
reading.sensorType());
                    }
                });

            // Anomaly detection
            sensorStream
                .filter((key, reading) -> reading.isAnomaly())
                .to("sensor-anomalies");

            // Windowed aggregations
            sensorStream
                .groupBy((key, reading) -> reading.deviceId())
                .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(5)))
                .aggregate(
                    SensorAggregation::new,
                    (key, reading, aggregation) ->
aggregation.addReading(reading),
                    Materialized.with(Serdes.String(), new JsonSerde<>
(SensorAggregation.class))
                )
                .toStream()
                .to("sensor-aggregations");

            return builder.build();
        }

        private void processTemperature(SensorReading reading) {
            // Temperature-specific processing
        }

        private void processHumidity(SensorReading reading) {
            // Humidity-specific processing
        }

        private void processPressure(SensorReading reading) {
            // Pressure-specific processing
        }
    }

    public static class SensorAggregation {
        private int count;
        private double sum;
        private double min = Double.MAX_VALUE;
        private double max = Double.MIN_VALUE;

        public SensorAggregation addReading(SensorReading reading) {
            count++;
            sum += reading.value();
            min = Math.min(min, reading.value());
            max = Math.max(max, reading.value());
            return this;
        }

        public double getAverage() {
            return count > 0 ? sum / count : 0;
        }

        // Getters and setters for JSON serialization
        public int getCount() { return count; }
        public void setCount(int count) { this.count = count; }

        public double getSum() { return sum; }
        public void setSum(double sum) { this.sum = sum; }

        public double getMin() { return min; }
        public void setMin(double min) { this.min = min; }
```

```
            public double getMax() { return max; }
            public void setMax(double max) { this.max = max; }
        }
    }
```

📈 Version Highlights

Java Integration Evolution

Kafka Version	Java Version	Key Features
4.0	Java 17+	Virtual threads support, enhanced client APIs
3.8	Java 11+	Improved Streams API, better error handling
3.6	Java 11+	Enhanced serialization, Schema Registry improvements
3.0	Java 8+	**Exactly-once semantics improvements**
2.8	Java 8+	**Streams improvements, better error handling**
2.4	Java 8+	**Foreign key joins in Streams**
2.0	Java 8+	**Headers support, Streams improvements**
1.0	Java 8+	**Exactly-once semantics, Streams GA**
0.11	Java 8+	**Exactly-once semantics preview**
0.10	Java 7+	**Kafka Streams introduction**

Modern Java Features Support

Java Version	Release	Kafka Features
Java 25	2025	Enhanced virtual threads, pattern matching improvements
Java 21	2023	**Virtual threads, pattern matching for switch**
Java 17	2021	**Sealed classes, pattern matching (preview)**
Java 16	2021	**Records (GA)**
Java 14	2020	**Records (preview), pattern matching (preview)**
Java 11	2018	**LTS, HTTP client, var improvements**
Java 8	2014	**Lambdas, Streams, CompletableFuture**

Serialization Evolution

Version	Serialization Features
Kafka 4.0	Enhanced Schema Registry integration, better error handling

Version	Serialization Features
Kafka 3.6	**Protobuf improvements, JSON Schema support**
Kafka 3.0	**Improved Avro serialization, better Schema Registry client**
Kafka 2.8	**Enhanced serialization error handling**
Kafka 2.4	**Schema references support**
Kafka 2.0	**Headers support in all serializers**

🔗 Additional Resources

📖 Official Documentation

- Kafka Java Client Documentation
- Kafka Streams Documentation
- Schema Registry Documentation

🎓 Learning Resources

- Kafka Streams Examples
- Modern Java Features Guide
- Virtual Threads Documentation

🛠️ Tools and Libraries

- Kafka Clients (Maven)
- Confluent Schema Registry Client
- Jackson JSON Library

📊 Performance Resources

- Kafka Performance Tuning
- Virtual Threads Performance Guide
- Serialization Performance Comparison

Last Updated: September 2025
Kafka Version: 4.0.0
Java Compatibility: Java 8+ (Java 21+ recommended for virtual threads)

> 💡 **Pro Tip**: Modern Kafka applications should leverage Java 21+ features like virtual threads for I/O-intensive workloads, records for immutable DTOs, and sealed classes for type-safe event hierarchies. The combination of Kafka's high-performance APIs with modern Java features creates robust, maintainable, and performant data streaming applications.

[397]