Kafka Streams: Complete Developer Guide

A comprehensive refresher on Apache Kafka Streams, designed for both beginners and experienced developers. This README covers stream processing concepts, operations, deployment strategies, and real-world applications with detailed Java examples.

Table of Contents

Core Concepts

KStream, KTable, GlobalKTable

Simple Explanation

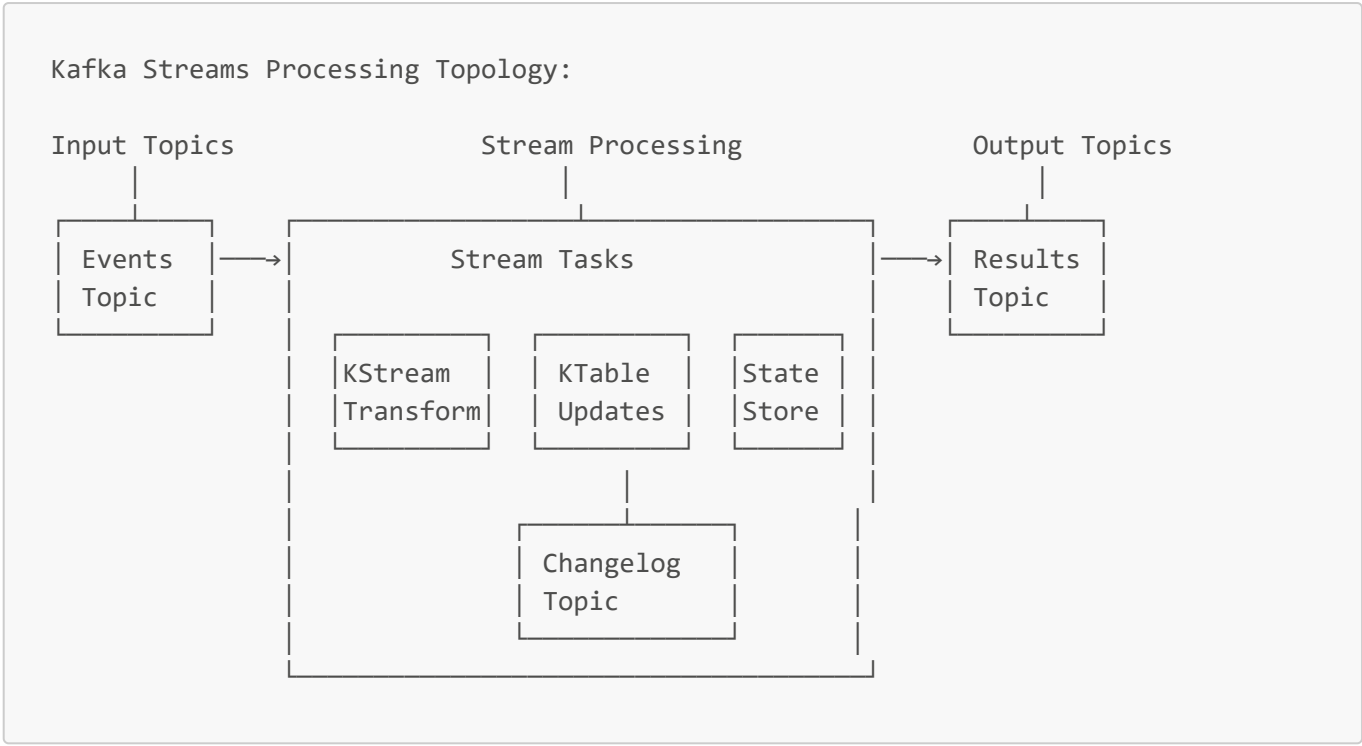
Kafka Streams provides three primary abstractions for working with data:

- **KStream**: Represents an unbounded stream of facts/events (record stream)
- **KTable**: Represents a changelog stream where each record is an update (table)
- **GlobalKTable**: A replicated table available to all stream tasks

Problem They Solve

- **KStream**: Processing infinite streams of events where each record is independent
- **KTable**: Maintaining the latest state for each key, supporting upserts and lookups
- **GlobalKTable**: Providing reference data accessible across all partitions for enrichment

Internal Architecture



KStream - Record Stream

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

public class KStreamExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Create KStream from topic
        KStream<String, String> sourceStream = builder.stream("user-events");

        // Each record is treated as an independent event
        KStream<String, String> processedStream = sourceStream
            .filter((key, value) -> value.contains("purchase"))
            .mapValues(value -> "Processed: " + value)
            .peek((key, value) -> System.out.println("Event: " + key + " -> " +
value));

        // Output to topic
        processedStream.to("processed-events");

        // Build and start topology
        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        // Shutdown hook
```

```
        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    private static Properties createStreamConfig() {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "kstream-example");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        return props;
    }
}
```

KTable - Changelog Stream

```
import org.apache.kafka.streams.state.KeyValueStore;
import org.apache.kafka.common.utils.Bytes;

public class KTableExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Create KTable from compacted topic - maintains latest value per key
        KTable<String, String> userTable = builder.table("user-profiles",
            Materialized.<String, String, KeyValueStore<Bytes, byte[]>>as("user-
store")
                .withKeySerde(Serdes.String())
                .withValueSerde(Serdes.String()));

        // Filter and transform the table
        KTable<String, String> activeUsersTable = userTable
            .filter((key, value) -> value.contains("active"))
            .mapValues(value -> value.toUpperCase());

        // Convert KTable back to KStream to see changes
        KStream<String, String> userChanges = activeUsersTable.toStream();

        userChanges.foreach((key, value) ->
            System.out.println("User update: " + key + " -> " + value));

        // Build and start
        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }
```

```
    private static Properties createStreamConfig() {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "ktable-example");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

        // Enable caching for KTable (default behavior)
        props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 10 * 1024 *
1024L); // 10MB
        props.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 30000); // 30 seconds

        return props;
    }
}
```

GlobalKTable - Replicated Reference Data

```
public class GlobalKTableExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Create GlobalKTable - replicated to all instances
        GlobalKTable<String, String> userProfilesGlobal = builder.globalTable(
            "user-profiles-global",
            Materialized.<String, String, KeyValueStore<Bytes, byte[]>>as("global-
user-store")
                .withKeySerde(Serdes.String())
                .withValueSerde(Serdes.String()));

        // Create stream of events
        KStream<String, String> userEvents = builder.stream("user-events");

        // Join stream with global table (no co-partitioning required)
        KStream<String, String> enrichedEvents = userEvents.join(
            userProfilesGlobal,
            (eventKey, eventValue) -> eventKey, // Extract join key from event
            (eventValue, profileValue) -> {
                return String.format("Event: %s, Profile: %s", eventValue,
profileValue);
            }
        );

        enrichedEvents.to("enriched-user-events");

        // Build and start
        KafkaStreams streams = new KafkaStreams(builder.build(), props);
```

```
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    /*
     * GlobalKTable Benefits:
     * 1. No co-partitioning requirement for joins
     * 2. Lower latency lookups (local copy)
     * 3. Can join on different keys
     *
     * GlobalKTable Trade-offs:
     * 1. Higher memory usage (full replication)
     * 2. Longer startup time (must read all data)
     * 3. Should only be used for small, slow-changing datasets
     */
}
```

Event-time vs Processing-time

Simple Explanation

- **Event Time**: When the event actually occurred (embedded in the record)
- **Processing Time**: When Kafka Streams processes the event (current system time)
- **Stream Time**: The maximum event time seen so far in the stream

Why It Matters

Event time ensures correct windowing and aggregations even when events arrive out of order, while processing time is simpler but can lead to incorrect results with late-arriving data.

TimestampExtractor Implementation

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.streams.processor.TimestampExtractor;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;

public class CustomTimestampExtractor implements TimestampExtractor {

    private final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public long extract(ConsumerRecord<Object, Object> record, long
previousTimestamp) {
        try {
            // Extract timestamp from JSON payload
            JsonNode jsonNode = objectMapper.readTree(record.value().toString());

            if (jsonNode.has("eventTime")) {
```

```
                return jsonNode.get("eventTime").asLong();
            } else if (jsonNode.has("timestamp")) {
                return jsonNode.get("timestamp").asLong();
            }

            // Fallback to record timestamp
            return record.timestamp();

        } catch (Exception e) {
            // Fallback to record timestamp on parsing error
            System.err.println("Failed to extract timestamp: " + e.getMessage());
            return record.timestamp();
        }
    }
}

public class EventTimeProcessingExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Use custom timestamp extractor for event time
        KStream<String, String> eventStream = builder.stream("events",
            Consumed.with(Serdes.String(), Serdes.String())
                .withTimestampExtractor(new CustomTimestampExtractor()));

        // Windowed aggregation using event time
        KTable<Windowed<String>, Long> windowedCounts = eventStream
            .groupByKey()
            .windowedBy(TimeWindows.of(Duration.ofMinutes(5))) // 5-minute windows
            .count();

        // Convert to stream to observe results
        windowedCounts.toStream().foreach((windowedKey, count) -> {
            System.out.printf("Window [%s - %s]: key=%s, count=%d%n",
                new java.util.Date(windowedKey.window().start()),
                new java.util.Date(windowedKey.window().end()),
                windowedKey.key(),
                count);
        });

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    private static Properties createStreamConfig() {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "event-time-example");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
```

```
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

        // Default timestamp extractor (uses record timestamp)
        // props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
        //     FailOnInvalidTimestamp.class);

        return props;
    }
}
```

Stream Time and Late Records

```
public class StreamTimeExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        KStream<String, String> stream = builder.stream("time-events");

        // Transform with timestamp awareness
        KStream<String, String> timestampedStream = stream.transform(
            () -> new Transformer<String, String, KeyValue<String, String>>() {
                private ProcessorContext context;

                @Override
                public void init(ProcessorContext context) {
                    this.context = context;
                }

                @Override
                public KeyValue<String, String> transform(String key, String
value) {
                    long eventTime = context.timestamp();
                    long streamTime = context.currentStreamTimeMs();

                    // Check if record is late
                    if (eventTime < streamTime - Duration.ofMinutes(5).toMillis())
{
                        System.out.printf("Late record detected: event=%d,
stream=%d, delay=%d ms%n",
                            eventTime, streamTime, streamTime - eventTime);
                    }

                    String enrichedValue =
String.format("%s|eventTime=%d|streamTime=%d",
                        value, eventTime, streamTime);

                    return KeyValue.pair(key, enrichedValue);
                }
```

```
                @Override
                public void close() {}
            }
        );

        timestampedStream.to("timestamped-events");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }
}
```

State Stores & RocksDB

Simple Explanation

State stores maintain local state for stateful operations like aggregations and joins. RocksDB is the default persistent key-value store that provides fast read/write access with changelog backup to Kafka topics.

Problem It Solves

- **Local State**: Fast access to intermediate results without network calls
- **Fault Tolerance**: State recovery through changelog topics
- **Scalability**: Distributed state across application instances

RocksDB Configuration

```
import org.apache.kafka.streams.state.RocksDBConfigSetter;
import org.rocksdb.*;
import java.util.Map;

public class CustomRocksDBConfig implements RocksDBConfigSetter {

    @Override
    public void setConfig(String storeName, Options options, Map<String, Object>
configs) {

        // Optimize for write-heavy workloads
        options.setCompactionStyle(CompactionStyle.LEVEL);
        options.setLevelCompactionDynamicLevelBytes(true);

        // Memory settings
        options.setDbWriteBufferSize(64 * 1024 * 1024); // 64MB write buffer
        options.setMaxWriteBufferNumber(3);
        options.setTargetFileSizeBase(64 * 1024 * 1024); // 64MB SST files

        // Block cache for reads (shared across all stores)
```

```
        BlockBasedTableConfig blockConfig = new BlockBasedTableConfig();
        blockConfig.setBlockCacheSize(100 * 1024 * 1024); // 100MB block cache
        blockConfig.setBlockSize(16 * 1024); // 16KB blocks
        blockConfig.setCacheIndexAndFilterBlocks(true);
        blockConfig.setPinL0FilterAndIndexBlocksInCache(true);

        // Bloom filter for faster key lookups
        blockConfig.setFilterPolicy(new BloomFilter(10, false));

        options.setTableFormatConfig(blockConfig);

        // Compression
        options.setCompressionType(CompressionType.LZ4_COMPRESSION);

        // Performance settings
        options.setMaxBackgroundCompactions(4);
        options.setMaxBackgroundFlushes(2);

options.setIncreaseParallelism(Runtime.getRuntime().availableProcessors());

        // Statistics for monitoring
        options.setStatistics(new Statistics());

        System.out.printf("Configured RocksDB for store: %s%n", storeName);
    }

    @Override
    public void close(String storeName, Options options) {
        // Cleanup resources
        System.out.printf("Closing RocksDB store: %s%n", storeName);
    }
}

public class StateStoreExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Create stream with custom state store
        KStream<String, String> userEvents = builder.stream("user-events");

        // Aggregate with custom materialized state store
        KTable<String, Long> userEventCounts = userEvents
            .groupByKey()
            .count(Materialized.<String, Long, KeyValueStore<Bytes,
byte[]>>as("user-counts-store")
                .withKeySerde(Serdes.String())
                .withValueSerde(Serdes.Long())
                .withCachingEnabled() // Enable caching for better performance
                .withLoggingEnabled(Map.of(
                    "cleanup.policy", "compact",
                    "min.cleanable.dirty.ratio", "0.01"
                )));
```

```
        // Access state store for queries (see Interactive Queries section)

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    private static Properties createStreamConfig() {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "state-store-example");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

        // Custom RocksDB configuration
        props.put(StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG,
CustomRocksDBConfig.class);

        // State store settings
        props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 50 * 1024 *
1024); // 50MB cache
        props.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 10000); // 10s commit
interval

        // State directory
        props.put(StreamsConfig.STATE_DIR_CONFIG, "/tmp/kafka-streams-state");

        return props;
    }
}
```

Custom State Store

```
import org.apache.kafka.streams.processor.StateStore;
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;
import org.apache.kafka.streams.processor.Processor;
import org.apache.kafka.streams.processor.ProcessorContext;

public class CustomStateStoreExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        Topology topology = new Topology();

        // Create custom state store
        StoreBuilder<KeyValueStore<String, Long>> storeBuilder = Stores
```

```
            .keyValueStoreBuilder(Stores.persistentKeyValueStore("custom-store"),
                Serdes.String(), Serdes.Long())
            .withCachingEnabled()
            .withLoggingEnabled(Map.of("cleanup.policy", "compact"));

        // Add state store to topology
        topology.addStateStore(storeBuilder);

        // Add source processor
        topology.addSource("source", "input-topic");

        // Add processor that uses the state store
        topology.addProcessor("processor", () -> new CustomProcessor(), "source");
        topology.connectProcessorAndStateStores("processor", "custom-store");

        // Add sink
        topology.addSink("sink", "output-topic", "processor");

        KafkaStreams streams = new KafkaStreams(topology, props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    // Custom processor using state store
    static class CustomProcessor implements Processor<String, String> {
        private ProcessorContext context;
        private KeyValueStore<String, Long> stateStore;

        @Override
        public void init(ProcessorContext context) {
            this.context = context;
            this.stateStore = (KeyValueStore<String, Long>)
context.getStateStore("custom-store");
        }

        @Override
        public void process(String key, String value) {
            // Get current count from state store
            Long currentCount = stateStore.get(key);
            if (currentCount == null) {
                currentCount = 0L;
            }

            // Increment count
            Long newCount = currentCount + 1;
            stateStore.put(key, newCount);

            // Forward result
            context.forward(key, String.valueOf(newCount));

            // Commit periodically
            if (newCount % 100 == 0) {
                context.commit();
```

```
            }
        }

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

⚙ Stream Processing Operations

Transformation Operations

Map, Filter, FlatMap Operations

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.util.Arrays;
import java.util.List;

public class TransformationOperationsExample {

    private static final ObjectMapper objectMapper = new ObjectMapper();

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        KStream<String, String> sourceStream = builder.stream("raw-events");

        // === MAP OPERATIONS ===

        // mapValues - transform only the value
        KStream<String, String> upperCaseStream = sourceStream
            .mapValues(value -> value.toUpperCase());

        // map - transform both key and value
        KStream<String, Integer> lengthStream = sourceStream
            .map((key, value) -> KeyValue.pair(key + "_length", value.length()));

        // selectKey - change the key
        KStream<String, String> rekeyedStream = sourceStream
            .selectKey((key, value) -> extractUserIdFromValue(value));

        // === FILTER OPERATIONS ===

        // filter - keep records matching predicate
        KStream<String, String> filteredStream = sourceStream
            .filter((key, value) -> value.contains("important"));
```

```
        // filterNot - remove records matching predicate
        KStream<String, String> cleanedStream = sourceStream
            .filterNot((key, value) -> value.contains("spam"));

        // === FLATMAP OPERATIONS ===

        // flatMapValues - one-to-many transformation on values
        KStream<String, String> wordsStream = sourceStream
            .flatMapValues(value -> Arrays.asList(value.split("\\s+")));

        // flatMap - one-to-many transformation on key-value pairs
        KStream<String, String> expandedStream = sourceStream
            .flatMap((key, value) -> {
                List<KeyValue<String, String>> result = Arrays.asList(
                    KeyValue.pair(key + "_part1", value.substring(0, Math.min(10,
value.length()))),
                    KeyValue.pair(key + "_part2", value.substring(Math.min(10,
value.length())))
                );
                return result;
            });

        // === PEEK OPERATION ===

        // peek - side effect without changing the stream
        KStream<String, String> debugStream = sourceStream
            .peek((key, value) -> System.out.println("Processing: " + key + " -> "
+ value));

        // === BRANCH OPERATION ===

        // branch - split stream into multiple branches
        KStream<String, String>[] branches = sourceStream.branch(
            (key, value) -> value.startsWith("ERROR"),   // Branch 0: errors
            (key, value) -> value.startsWith("WARN"),    // Branch 1: warnings
            (key, value) -> true                         // Branch 2: everything
else
        );

        branches[0].to("error-events");
        branches[1].to("warning-events");
        branches[2].to("info-events");

        // === COMPLEX TRANSFORMATION EXAMPLE ===

        KStream<String, String> processedEvents = sourceStream
            .filter((key, value) -> isValidJson(value))
            .mapValues(this::enrichEvent)
            .filter((key, value) -> !value.contains("test"))
            .selectKey((key, value) -> extractKeyFromEvent(value))
            .peek((key, value) -> logProcessedEvent(key, value));

        processedEvents.to("processed-events");
```

```
        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    private static String extractUserIdFromValue(String value) {
        try {
            JsonNode json = objectMapper.readTree(value);
            return json.has("userId") ? json.get("userId").asText() : "unknown";
        } catch (Exception e) {
            return "unknown";
        }
    }

    private static boolean isValidJson(String value) {
        try {
            objectMapper.readTree(value);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    private String enrichEvent(String value) {
        try {
            JsonNode json = objectMapper.readTree(value);
            // Add timestamp if not present
            if (!json.has("timestamp")) {
                return objectMapper.writeValueAsString(
                    objectMapper.createObjectNode()
                        .setAll((com.fasterxml.jackson.databind.node.ObjectNode)
json)
                        .put("timestamp", System.currentTimeMillis())
                );
            }
            return value;
        } catch (Exception e) {
            return value;
        }
    }

    private String extractKeyFromEvent(String value) {
        try {
            JsonNode json = objectMapper.readTree(value);
            return json.has("entityId") ? json.get("entityId").asText() :
"default";
        } catch (Exception e) {
            return "default";
        }
    }

    private void logProcessedEvent(String key, String value) {
```

```
            System.out.printf("Processed event: %s -> %s%n", key,
                value.length() > 100 ? value.substring(0, 100) + "..." : value);
        }

        private static Properties createStreamConfig() {
            Properties props = new Properties();
            props.put(StreamsConfig.APPLICATION_ID_CONFIG, "transformation-example");
            props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
    Serdes.String().getClass());
            props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
    Serdes.String().getClass());
            return props;
        }
    }
```

Windowing & Aggregation

Tumbling Windows

```
    public class TumblingWindowExample {

        public static void main(String[] args) {
            Properties props = createStreamConfig();
            StreamsBuilder builder = new StreamsBuilder();

            KStream<String, String> salesEvents = builder.stream("sales-events");

            // Parse sales amount from event
            KStream<String, Double> salesAmounts = salesEvents
                .mapValues(value -> parseSalesAmount(value))
                .filter((key, value) -> value != null);

            // Tumbling window aggregation - non-overlapping 5-minute windows
            TimeWindows tumblingWindow = TimeWindows.of(Duration.ofMinutes(5));

            KTable<Windowed<String>, Double> salesByStore = salesAmounts
                .groupByKey()
                .windowedBy(tumblingWindow)
                .aggregate(
                    () -> 0.0,                                    // Initializer
                    (key, value, aggregate) -> aggregate + value,  // Aggregator
                    Materialized.<String, Double, WindowStore<Bytes,
    byte[]>>as("sales-by-store")
                        .withKeySerde(Serdes.String())
                        .withValueSerde(Serdes.Double()));

            // Convert to stream to output results
            salesByStore.toStream().foreach((windowedKey, totalSales) -> {
                System.out.printf("Store: %s, Window: [%s - %s], Total Sales:
    $%.2f%n",
```

```
                windowedKey.key(),
                formatTime(windowedKey.window().start()),
                formatTime(windowedKey.window().end()),
                totalSales);
        });

        // Suppress intermediate results - only emit when window closes
        KTable<Windowed<String>, Double> finalResults = salesByStore

.suppress(Suppressed.untilWindowCloses(Suppressed.BufferConfig.unbounded()));

        finalResults.toStream()
            .map((windowedKey, value) -> KeyValue.pair(
                windowedKey.key() + "_" + windowedKey.window().start(),
                String.format("%.2f", value)))
            .to("sales-summary");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    private static Double parseSalesAmount(String eventValue) {
        try {
            // Assuming JSON format: {"storeId": "store1", "amount": 123.45, ...}
            JsonNode json = new ObjectMapper().readTree(eventValue);
            return json.has("amount") ? json.get("amount").asDouble() : null;
        } catch (Exception e) {
            return null;
        }
    }

    private static String formatTime(long timestamp) {
        return new java.text.SimpleDateFormat("HH:mm:ss").format(new
java.util.Date(timestamp));
    }
}
```

Hopping Windows

```
public class HoppingWindowExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        KStream<String, String> clickEvents = builder.stream("click-events");

        // Extract user ID and create stream of user clicks
        KStream<String, String> userClicks = clickEvents
```

```
            .selectKey((key, value) -> extractUserId(value))
            .mapValues(value -> "click");

        // Hopping window - 10-minute windows advancing every 2 minutes
        TimeWindows hoppingWindow = TimeWindows
            .of(Duration.ofMinutes(10))      // Window size
            .advanceBy(Duration.ofMinutes(2)); // Advance interval

        KTable<Windowed<String>, Long> clickCounts = userClicks
            .groupByKey()
            .windowedBy(hoppingWindow)
            .count(Materialized.as("user-click-counts"));

        // Monitor for high-activity users (more than 100 clicks in window)
        clickCounts.toStream()
            .filter((windowedKey, count) -> count > 100)
            .foreach((windowedKey, count) -> {
                System.out.printf("HIGH ACTIVITY ALERT: User %s had %d clicks in
window [%s - %s]%n",
                    windowedKey.key(),
                    count,
                    formatTime(windowedKey.window().start()),
                    formatTime(windowedKey.window().end()));
            });

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    private static String extractUserId(String eventValue) {
        try {
            JsonNode json = new ObjectMapper().readTree(eventValue);
            return json.has("userId") ? json.get("userId").asText() : "unknown";
        } catch (Exception e) {
            return "unknown";
        }
    }
}
```

Session Windows

```
public class SessionWindowExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        KStream<String, String> userActivity = builder.stream("user-activity");
```

```
        // Extract user ID from activity events
        KStream<String, String> userSessions = userActivity
            .selectKey((key, value) -> extractUserId(value));

        // Session window - group activities with 30-minute inactivity gap
        SessionWindows sessionWindow = SessionWindows
            .with(Duration.ofMinutes(30))      // Inactivity gap
            .grace(Duration.ofMinutes(5));     // Grace period for late events

        // Count activities per session
        KTable<Windowed<String>, Long> sessionActivityCounts = userSessions
            .groupByKey()
            .windowedBy(sessionWindow)
            .count(Materialized.as("user-sessions"));

        // Aggregate session data
        KTable<Windowed<String>, String> sessionSummary = userSessions
            .groupByKey()
            .windowedBy(sessionWindow)
            .aggregate(
                () -> new SessionData(),  // Initializer
                (key, value, sessionData) -> {
                    sessionData.addActivity(value);
                    return sessionData;
                },
                (key, sessionData1, sessionData2) -> {
                    // Merger for session window merging
                    sessionData1.merge(sessionData2);
                    return sessionData1;
                },
                Materialized.<String, SessionData, SessionStore<Bytes,
byte[]>>as("session-summary")
                    .withKeySerde(Serdes.String())
                    .withValueSerde(new SessionDataSerde()));

        // Output session summaries
        sessionSummary.toStream().foreach((windowedKey, sessionData) -> {
            long sessionDuration = windowedKey.window().end() -
windowedKey.window().start();
            System.out.printf("Session completed - User: %s, Duration: %d minutes,
Activities: %d%n",
                windowedKey.key(),
                sessionDuration / (1000 * 60),
                sessionData.getActivityCount());
        });

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    // Session data aggregator
    static class SessionData {
```

```
        private int activityCount = 0;
        private java.util.Set<String> activityTypes = new java.util.HashSet<>();

        public void addActivity(String activity) {
            activityCount++;
            try {
                JsonNode json = new ObjectMapper().readTree(activity);
                if (json.has("type")) {
                    activityTypes.add(json.get("type").asText());
                }
            } catch (Exception e) {
                // Handle parsing error
            }
        }

        public void merge(SessionData other) {
            this.activityCount += other.activityCount;
            this.activityTypes.addAll(other.activityTypes);
        }

        public int getActivityCount() {
            return activityCount;
        }

        public java.util.Set<String> getActivityTypes() {
            return activityTypes;
        }
    }

    // Custom Serde for SessionData
    static class SessionDataSerde implements Serde<SessionData> {
        @Override
        public Serializer<SessionData> serializer() {
            return new SessionDataSerializer();
        }

        @Override
        public Deserializer<SessionData> deserializer() {
            return new SessionDataDeserializer();
        }
    }

    static class SessionDataSerializer implements Serializer<SessionData> {
        private final ObjectMapper objectMapper = new ObjectMapper();

        @Override
        public byte[] serialize(String topic, SessionData data) {
            try {
                return objectMapper.writeValueAsBytes(data);
            } catch (Exception e) {
                throw new RuntimeException("Error serializing SessionData", e);
            }
        }
    }
```

```
    static class SessionDataDeserializer implements Deserializer<SessionData> {
        private final ObjectMapper objectMapper = new ObjectMapper();

        @Override
        public SessionData deserialize(String topic, byte[] data) {
            try {
                return objectMapper.readValue(data, SessionData.class);
            } catch (Exception e) {
                throw new RuntimeException("Error deserializing SessionData", e);
            }
        }
    }
}
```

Join Operations

Stream-Stream Join

```
public class StreamStreamJoinExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Two input streams that need to be joined
        KStream<String, String> orderEvents = builder.stream("order-events");
        KStream<String, String> paymentEvents = builder.stream("payment-events");

        // Both streams must be co-partitioned (same key and same number of
partitions)

        // Define join window - events must arrive within 10 minutes of each other
        JoinWindows joinWindow = JoinWindows.of(Duration.ofMinutes(10))
            .grace(Duration.ofMinutes(2)); // Grace period for late events

        // Inner join - only emit when both sides have events within window
        KStream<String, String> innerJoinedStream = orderEvents.join(
            paymentEvents,
            (orderValue, paymentValue) -> {
                // Value joiner - combine values from both sides
                return String.format("Order: %s, Payment: %s", orderValue,
paymentValue);
            },
            joinWindow,
            Joined.with(Serdes.String(), Serdes.String(), Serdes.String()));

        // Left join - emit for every order event, with null payment if no match
        KStream<String, String> leftJoinedStream = orderEvents.leftJoin(
            paymentEvents,
            (orderValue, paymentValue) -> {
```

```
                if (paymentValue != null) {
                    return String.format("Paid Order: %s, Payment: %s",
orderValue, paymentValue);
                } else {
                    return String.format("Unpaid Order: %s", orderValue);
                }
            },
            joinWindow,
            Joined.with(Serdes.String(), Serdes.String(), Serdes.String()));

        // Outer join - emit for events from either side
        KStream<String, String> outerJoinedStream = orderEvents.outerJoin(
            paymentEvents,
            (orderValue, paymentValue) -> {
                if (orderValue != null && paymentValue != null) {
                    return String.format("Matched: Order=%s, Payment=%s",
orderValue, paymentValue);
                } else if (orderValue != null) {
                    return String.format("Order without payment: %s", orderValue);
                } else {
                    return String.format("Payment without order: %s",
paymentValue);
                }
            },
            joinWindow,
            Joined.with(Serdes.String(), Serdes.String(), Serdes.String()));

        // Output joined results
        innerJoinedStream.to("order-payment-matched");
        leftJoinedStream.to("order-payment-left-join");
        outerJoinedStream.to("order-payment-outer-join");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    /*
     * Stream-Stream Join Requirements:
     * 1. Both streams must be co-partitioned
     * 2. Same number of partitions
     * 3. Same partitioning strategy (same key)
     * 4. Events must arrive within the join window
     *
     * Use Cases:
     * - Correlating related events (order + payment)
     * - Fraud detection (transaction + location)
     * - IoT sensor data correlation
     */
}
```

Stream-Table Join

```
public class StreamTableJoinExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Stream of user events
        KStream<String, String> userEvents = builder.stream("user-events");

        // Table of user profiles (latest state per user)
        KTable<String, String> userProfiles = builder.table("user-profiles");

        // Stream-Table join - enrich events with user profile data
        KStream<String, String> enrichedEvents = userEvents.join(
            userProfiles,
            (eventValue, profileValue) -> {
                return String.format("Event: %s, Profile: %s", eventValue,
profileValue);
            },
            Joined.with(Serdes.String(), Serdes.String(), Serdes.String()));

        // Left join - include events even if no profile exists
        KStream<String, String> leftJoinedEvents = userEvents.leftJoin(
            userProfiles,
            (eventValue, profileValue) -> {
                if (profileValue != null) {
                    return String.format("Event with profile: %s, Profile: %s",
eventValue, profileValue);
                } else {
                    return String.format("Event without profile: %s", eventValue);
                }
            },
            Joined.with(Serdes.String(), Serdes.String(), Serdes.String()));

        enrichedEvents.to("enriched-user-events");
        leftJoinedEvents.to("user-events-with-profiles");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    /*
     * Stream-Table Join Characteristics:
     * 1. Not windowed - uses latest table value
     * 2. Streams and tables must be co-partitioned
     * 3. Stream events are enriched with table data
     * 4. Table updates don't trigger new join results
     *
```

```
     * Use Cases:
     * - Event enrichment with reference data
     * - Adding user profile to events
     * - Product catalog lookups
     */
}
```

Stream-GlobalKTable Join

```
public class StreamGlobalKTableJoinExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Stream of events (can be from any partition structure)
        KStream<String, String> events = builder.stream("product-events");

        // GlobalKTable - replicated to all instances (no co-partitioning needed)
        GlobalKTable<String, String> productCatalog =
builder.globalTable("product-catalog");

        // Join stream with global table
        KStream<String, String> enrichedEvents = events.join(
            productCatalog,
            (eventKey, eventValue) -> extractProductId(eventValue), // Key mapper
            (eventValue, catalogValue) -> {
                return String.format("Event: %s, Product Info: %s", eventValue,
catalogValue);
            });

        // Left join with global table
        KStream<String, String> leftJoinedEvents = events.leftJoin(
            productCatalog,
            (eventKey, eventValue) -> extractProductId(eventValue),
            (eventValue, catalogValue) -> {
                if (catalogValue != null) {
                    return String.format("Event with catalog: %s, Catalog: %s",
eventValue, catalogValue);
                } else {
                    return String.format("Event without catalog: %s", eventValue);
                }
            });

        enrichedEvents.to("product-events-enriched");
        leftJoinedEvents.to("product-events-with-catalog");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

```
    }

    private static String extractProductId(String eventValue) {
        try {
            JsonNode json = new ObjectMapper().readTree(eventValue);
            return json.has("productId") ? json.get("productId").asText() : null;
        } catch (Exception e) {
            return null;
        }
    }

    /*
     * Stream-GlobalKTable Join Benefits:
     * 1. No co-partitioning requirement
     * 2. Can join on different keys
     * 3. Lower latency (local lookup)
     * 4. Supports non-key joins with key mapper
     *
     * Trade-offs:
     * 1. Higher memory usage (full replication)
     * 2. Longer startup time
     * 3. Should only be used for small reference datasets
     */
}
```

Table-Table Join

```
public class TableTableJoinExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();
        StreamsBuilder builder = new StreamsBuilder();

        // Two tables representing different aspects of user data
        KTable<String, String> userProfiles = builder.table("user-profiles");
        KTable<String, String> userPreferences = builder.table("user-
preferences");

        // Inner join - combine user profile with preferences
        KTable<String, String> userProfilesWithPreferences = userProfiles.join(
            userPreferences,
            (profileValue, preferencesValue) -> {
                return String.format("Profile: %s, Preferences: %s", profileValue,
preferencesValue);
            },
            Materialized.as("user-profiles-with-preferences"));

        // Left join - include all profiles, with null preferences if not
available
        KTable<String, String> allProfilesWithOptionalPreferences =
userProfiles.leftJoin(
```

```
            userPreferences,
            (profileValue, preferencesValue) -> {
                if (preferencesValue != null) {
                    return String.format("Profile: %s, Preferences: %s",
profileValue, preferencesValue);
                } else {
                    return String.format("Profile: %s, No preferences",
profileValue);
                }
            },
            Materialized.as("all-profiles-with-preferences"));

        // Outer join - include data from both tables
        KTable<String, String> outerJoinedProfiles = userProfiles.outerJoin(
            userPreferences,
            (profileValue, preferencesValue) -> {
                String profile = profileValue != null ? profileValue : "No
profile";
                String preferences = preferencesValue != null ? preferencesValue :
"No preferences";
                return String.format("Profile: %s, Preferences: %s", profile,
preferences);
            },
            Materialized.as("outer-joined-profiles"));

        // Convert tables to streams to output changes
        userProfilesWithPreferences.toStream().to("user-complete-profiles");
        allProfilesWithOptionalPreferences.toStream().to("user-profiles-optional-
prefs");
        outerJoinedProfiles.toStream().to("user-profiles-outer-join");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    /*
     * Table-Table Join Characteristics:
     * 1. Result is another table (not a stream)
     * 2. Not windowed - uses current values
     * 3. Updates to either table trigger re-evaluation
     * 4. Tables must be co-partitioned
     * 5. Only emits changes when result changes
     *
     * Use Cases:
     * - Combining related reference data
     * - Creating materialized views
     * - Data denormalization
     */
}
```

🚀 Deployment & Scaling

Parallelism with Partitions

Understanding Kafka Streams Parallelism

```
public class ParallelismExample {

    public static void main(String[] args) {
        // Parallelism in Kafka Streams is determined by:
        // 1. Number of input topic partitions
        // 2. Number of stream threads per instance
        // 3. Number of application instances

        Properties props = createStreamConfig();

        // Configure multiple stream threads
        props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 4);

        StreamsBuilder builder = new StreamsBuilder();

        // Stream from topic with multiple partitions
        KStream<String, String> events = builder.stream("multi-partition-topic");

        // Each stream task processes one partition
        KStream<String, String> processed = events
            .peek((key, value) -> {
                System.out.printf("Thread: %s, Processing: %s -> %s%n",
                    Thread.currentThread().getName(), key, value);
            })
            .mapValues(value -> processEvent(value))
            .filter((key, value) -> value != null);

        processed.to("processed-topic");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);

        // Add state change listener to monitor thread assignment
        streams.setStateListener((newState, oldState) -> {
            System.out.printf("Streams state change: %s -> %s%n", oldState,
newState);
        });

        streams.start();

        // Runtime statistics
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            printStreamMetrics(streams);
            streams.close();
        }));
    }
```

```
    private static String processEvent(String value) {
        // Simulate processing time
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return null;
        }
        return value.toUpperCase();
    }

    private static void printStreamMetrics(KafkaStreams streams) {
        System.out.println("\n=== Stream Metrics ===");
        streams.allMetrics().forEach((metricName, metric) -> {
            if (metricName.name().contains("process-rate") ||
                metricName.name().contains("commit-rate")) {
                System.out.printf("%s: %.2f%n", metricName.name(),
metric.metricValue());
            }
        });
    }

    /*
     * Parallelism Rules:
     *
     * 1. Maximum parallelism = number of input partitions
     * 2. Stream tasks = max(partitions across all input topics)
     * 3. Each stream task assigned to exactly one thread
     * 4. Each partition assigned to exactly one stream task
     *
     * Example:
     * - Topic A: 6 partitions
     * - Topic B: 4 partitions
     * - Stream tasks created: 6 (max of 6, 4)
     * - Can run up to 6 threads across all instances
     *
     * Scaling strategies:
     * - Scale up: Add more instances (up to partition count)
     * - Scale out: Increase partition count (requires repartitioning)
     */
}
```

Dynamic Scaling Example

```
public class DynamicScalingExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();

        // Start with fewer threads, add more dynamically
        props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 2);
```

```
        StreamsBuilder builder = new StreamsBuilder();

        KStream<String, String> events = builder.stream("scaling-test-topic");

        // CPU-intensive processing to demonstrate scaling
        KStream<String, String> processed = events
            .mapValues(value -> performHeavyProcessing(value));

        processed.to("scaling-output-topic");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);
        streams.start();

        // Monitor and scale dynamically
        scheduleScalingMonitor(streams);

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }

    private static void scheduleScalingMonitor(KafkaStreams streams) {
        java.util.concurrent.ScheduledExecutorService scheduler =
            java.util.concurrent.Executors.newScheduledThreadPool(1);

        scheduler.scheduleAtFixedRate(() -> {
            try {
                // Get processing rate
                double processRate = getProcessingRate(streams);
                int currentThreads = streams.allMetadata().size();

                System.out.printf("Current threads: %d, Process rate: %.2f
records/sec%n",
                    currentThreads, processRate);

                // Scale up if processing rate is low and we have available
partitions
                if (processRate < 100 && currentThreads < 6) {
                    System.out.println("Scaling up: Adding stream thread");
                    streams.addStreamThread();
                }

                // Scale down if we have excess capacity
                if (processRate > 500 && currentThreads > 2) {
                    System.out.println("Scaling down: Removing stream thread");
                    streams.removeStreamThread();
                }

            } catch (Exception e) {
                System.err.println("Error in scaling monitor: " + e.getMessage());
            }
        }, 30, 30, java.util.concurrent.TimeUnit.SECONDS);
    }

    private static double getProcessingRate(KafkaStreams streams) {
```

```
        return streams.allMetrics().entrySet().stream()
            .filter(entry -> entry.getKey().name().equals("process-rate"))
            .mapToDouble(entry -> (Double) entry.getValue().metricValue())
            .sum();
    }

    private static String performHeavyProcessing(String value) {
        // Simulate CPU-intensive work
        StringBuilder result = new StringBuilder(value);
        for (int i = 0; i < 1000; i++) {
            result.append("_").append(i);
        }
        return result.substring(0, Math.min(1000, result.length()));
    }
}
```

Fault Tolerance & State Recovery

State Recovery Example

```
public class FaultToleranceExample {

    public static void main(String[] args) {
        Properties props = createStreamConfig();

        // Configure for fault tolerance
        props.put(StreamsConfig.REPLICATION_FACTOR_CONFIG, 3);
        props.put(StreamsConfig.STATE_DIR_CONFIG, "/tmp/kafka-streams-fault-
tolerance");

        // Enable exactly-once processing for stronger guarantees
        props.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG,
StreamsConfig.EXACTLY_ONCE_V2);

        StreamsBuilder builder = new StreamsBuilder();

        KStream<String, String> events = builder.stream("fault-tolerance-events");

        // Stateful operation with fault tolerance
        KTable<String, Long> eventCounts = events
            .groupByKey()
            .count(Materialized.<String, Long, KeyValueStore<Bytes,
byte[]>>as("event-counts")
                .withCachingEnabled()
                .withLoggingEnabled(Map.of(
                    "cleanup.policy", "compact",
                    "segment.ms", "3600000" // 1 hour segments
                )));

        // Simulate failure recovery
        KStream<String, String> enrichedEvents = events.transformValues(
```

```
            () -> new StatefulTransformer(),
            "fault-tolerance-store"
        );

        // Add state store
        StoreBuilder<KeyValueStore<String, String>> storeBuilder = Stores
            .keyValueStoreBuilder(Stores.persistentKeyValueStore("fault-tolerance-
store"),
                Serdes.String(), Serdes.String())
            .withCachingEnabled()
            .withLoggingEnabled(Map.of("cleanup.policy", "compact"));

        builder.addStateStore(storeBuilder);

        enrichedEvents.to("fault-tolerant-output");

        KafkaStreams streams = new KafkaStreams(builder.build(), props);

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

            // Return REPLACE_THREAD to restart the failed thread
            return
StreamsUncaughtExceptionHandler.StreamThreadExceptionResponse.REPLACE_THREAD;
        });

        // Add state change listener for monitoring
        streams.setStateListener((newState, oldState) -> {
            System.out.printf("State transition: %s -> %s%n", oldState, newState);

            if (newState == KafkaStreams.State.REBALANCING) {
                System.out.println("Rebalancing - some tasks may be recovering
state");
            } else if (newState == KafkaStreams.State.RUNNING) {
                System.out.println("All tasks running - state recovery complete");
            }
        });

        streams.start();

        // Simulate graceful shutdown
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("Shutting down streams application...");
            streams.close(Duration.ofSeconds(30));
            System.out.println("Shutdown complete");
        }));
    }

    // Custom transformer with state
    static class StatefulTransformer implements ValueTransformer<String, String> {
        private KeyValueStore<String, String> stateStore;
```

```
        private ProcessorContext context;

        @Override
        public void init(ProcessorContext context) {
            this.context = context;
            this.stateStore = (KeyValueStore<String, String>)
                context.getStateStore("fault-tolerance-store");
        }

        @Override
        public String transform(String value) {
            try {
                // Simulate processing that updates state
                String previousValue = stateStore.get("last-processed");
                stateStore.put("last-processed", value);
                stateStore.put("processed-at-" + System.currentTimeMillis(),
value);

                // Simulate occasional failures for testing
                if (value.contains("error")) {
                    throw new RuntimeException("Simulated processing error");
                }

                return String.format("Processed: %s (previous: %s)", value,
previousValue);

            } catch (Exception e) {
                System.err.println("Processing error: " + e.getMessage());
                // Return original value on error
                return value;
            }
        }

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

    /*
     * Fault Tolerance Mechanisms:
     *
     * 1. Changelog Topics:
     *    - Every state store backed by replicated Kafka topic
     *    - Automatic state recovery on restart
     *    - Configurable replication factor
     *
     * 2. Checkpointing:
     *    - Periodic commits of processing progress
     *    - Recovery from last checkpoint on failure
     *
     * 3. Exception Handling:
     *    - Uncaught exception handlers
     *    - Thread replacement vs shutdown
```

```
     *
     * 4. Exactly-Once Processing:
     *    - Transactional semantics
     *    - No duplicate processing on recovery
     */
}
```

Interactive Queries

Basic Interactive Queries

```
import org.apache.kafka.streams.state.QueryableStoreTypes;
import org.apache.kafka.streams.state.ReadOnlyKeyValueStore;
import org.apache.kafka.streams.StoreQueryParameters;

public class InteractiveQueriesExample {

    private KafkaStreams streams;

    public static void main(String[] args) {
        InteractiveQueriesExample example = new InteractiveQueriesExample();
        example.startStreamsApplication();
        example.startQueryServer();
    }

    private void startStreamsApplication() {
        Properties props = createStreamConfig();

        // Enable interactive queries
        props.put(StreamsConfig.APPLICATION_SERVER_CONFIG, "localhost:8080");

        StreamsBuilder builder = new StreamsBuilder();

        KStream<String, String> userEvents = builder.stream("user-events");

        // Create queryable state store
        KTable<String, Long> userEventCounts = userEvents
            .groupByKey()
            .count(Materialized.<String, Long, KeyValueStore<Bytes,
byte[]>>as("user-event-counts")
                .withKeySerde(Serdes.String())
                .withValueSerde(Serdes.Long()));

        // Create windowed state store
        KTable<Windowed<String>, Long> userEventCountsByWindow = userEvents
            .groupByKey()
            .windowedBy(TimeWindows.of(Duration.ofMinutes(5)))
            .count(Materialized.as("user-event-counts-windowed"));

        streams = new KafkaStreams(builder.build(), props);
        streams.start();
```

```
        // Wait for streams to be ready
        streams.setStateListener((newState, oldState) -> {
            if (newState == KafkaStreams.State.RUNNING) {
                System.out.println("Streams application is running - queries
enabled");
            }
        });
    }

    private void startQueryServer() {
        // Simple HTTP server for interactive queries
        try {
            com.sun.net.httpserver.HttpServer server =
                com.sun.net.httpserver.HttpServer.create(
                    new java.net.InetSocketAddress(8080), 0);

            // Endpoint to query user event count
            server.createContext("/user-count", exchange -> {
                String userId =
extractUserIdFromPath(exchange.getRequestURI().getPath());
                Long count = queryUserEventCount(userId);

                String response = String.format("{\"userId\": \"%s\", \"count\":
%d}",
                    userId, count != null ? count : 0);

                exchange.getResponseHeaders().add("Content-Type",
"application/json");
                exchange.sendResponseHeaders(200, response.length());
                exchange.getResponseBody().write(response.getBytes());
                exchange.close();
            });

            // Endpoint to query all user counts
            server.createContext("/all-counts", exchange -> {
                String response = queryAllUserCounts();

                exchange.getResponseHeaders().add("Content-Type",
"application/json");
                exchange.sendResponseHeaders(200, response.length());
                exchange.getResponseBody().write(response.getBytes());
                exchange.close();
            });

            // Endpoint for range queries
            server.createContext("/user-range", exchange -> {
                String response = queryUserRange("user1", "user9");

                exchange.getResponseHeaders().add("Content-Type",
"application/json");
                exchange.sendResponseHeaders(200, response.length());
                exchange.getResponseBody().write(response.getBytes());
                exchange.close();
```

```
            });

            server.start();
            System.out.println("Interactive query server started on port 8080");

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

    private Long queryUserEventCount(String userId) {
        try {
            ReadOnlyKeyValueStore<String, Long> store = streams.store(
                StoreQueryParameters.fromNameAndType("user-event-counts",
                    QueryableStoreTypes.keyValueStore()));

            return store.get(userId);

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

    private String queryAllUserCounts() {
        try {
            ReadOnlyKeyValueStore<String, Long> store = streams.store(
                StoreQueryParameters.fromNameAndType("user-event-counts",
                    QueryableStoreTypes.keyValueStore()));

            StringBuilder result = new StringBuilder("{\"users\": [");
            boolean first = true;

            try (org.apache.kafka.streams.KeyValue<String, Long> iterator =
                    (org.apache.kafka.streams.KeyValue<String, Long>)
store.all().next()) {

                // Note: This is simplified - actual implementation would use
proper iteration

            } catch (Exception e) {
                // Handle iteration
            }

            result.append("]}");
            return result.toString();

        } catch (Exception e) {
            return "{\"error\": \"" + e.getMessage() + "\"}";
        }
    }

    private String queryUserRange(String from, String to) {
        try {
```

```
            ReadOnlyKeyValueStore<String, Long> store = streams.store(
                StoreQueryParameters.fromNameAndType("user-event-counts",
                    QueryableStoreTypes.keyValueStore()));

            StringBuilder result = new StringBuilder("{\"range\": [");

            // Range query
            var iterator = store.range(from, to);
            boolean first = true;

            while (iterator.hasNext()) {
                var entry = iterator.next();
                if (!first) result.append(",");
                result.append(String.format("{\"key\": \"%s\", \"value\": %d}",
                    entry.key, entry.value));
                first = false;
            }
            iterator.close();

            result.append("]}");
            return result.toString();

        } catch (Exception e) {
            return "{\"error\": \"" + e.getMessage() + "\"}";
        }
    }

    private String extractUserIdFromPath(String path) {
        // Extract user ID from path like "/user-count/user123"
        String[] parts = path.split("/");
        return parts.length > 2 ? parts[2] : "unknown";
    }

    /*
     * Interactive Queries Features:
     *
     * 1. Local Queries:
     *    - Query local state stores directly
     *    - Low latency (no network calls)
     *    - Point queries and range queries
     *
     * 2. Remote Queries:
     *    - Discover which instance has the data
     *    - Route queries to correct instance
     *    - Aggregate results from multiple instances
     *
     * 3. Query Types:
     *    - Key-value store queries
     *    - Window store queries
     *    - Session store queries
     *
     * Use Cases:
     * - Real-time dashboards
     * - Application state inspection
```

```
     * - Debugging and monitoring
     */
}
```

Distributed Interactive Queries

```
import org.apache.kafka.streams.state.HostInfo;

public class DistributedInteractiveQueriesExample {

    private KafkaStreams streams;
    private HostInfo thisHostInfo;

    public void startDistributedQueries() {
        thisHostInfo = new HostInfo("localhost", 8080);

        Properties props = createStreamConfig();
        props.put(StreamsConfig.APPLICATION_SERVER_CONFIG,
            thisHostInfo.host() + ":" + thisHostInfo.port());

        // ... build topology ...

        streams = new KafkaStreams(builder.build(), props);
        streams.start();

        startDistributedQueryServer();
    }

    private void startDistributedQueryServer() {
        try {
            com.sun.net.httpserver.HttpServer server =
                com.sun.net.httpserver.HttpServer.create(
                    new java.net.InetSocketAddress(thisHostInfo.port()), 0);

            // Distributed query endpoint
            server.createContext("/distributed-count", exchange -> {
                String userId =
extractUserIdFromPath(exchange.getRequestURI().getPath());
                String response = queryUserCountDistributed(userId);

                exchange.getResponseHeaders().add("Content-Type",
"application/json");
                exchange.sendResponseHeaders(200, response.length());
                exchange.getResponseBody().write(response.getBytes());
                exchange.close();
            });

            server.start();

        } catch (Exception e) {
            System.err.println("Failed to start distributed query server: " +
```

```
e.getMessage());
        }
    }

    private String queryUserCountDistributed(String userId) {
        try {
            // Find which instance has the data for this key
            HostInfo hostInfo = streams.allMetadataForStore("user-event-counts")
                .stream()
                .filter(metadata -> metadata.stateStoreNames().contains("user-
event-counts"))
                .map(metadata -> metadata.hostInfo())
                .filter(host -> isKeyOnHost(userId, host))
                .findFirst()
                .orElse(null);

            if (hostInfo == null) {
                return "{\"error\": \"No host found for key: " + userId + "\"}";
            }

            // Query local store if data is on this instance
            if (hostInfo.equals(thisHostInfo)) {
                Long count = queryUserEventCount(userId);
                return String.format("{\"userId\": \"%s\", \"count\": %d,
\"host\": \"local\"}",
                    userId, count != null ? count : 0);
            } else {
                // Query remote instance
                return queryRemoteHost(hostInfo, userId);
            }

        } catch (Exception e) {
            return "{\"error\": \"" + e.getMessage() + "\"}";
        }
    }

    private boolean isKeyOnHost(String key, HostInfo host) {
        // In a real implementation, you would check the partition assignment
        // This is simplified for the example
        return streams.allMetadata().stream()
            .anyMatch(metadata -> metadata.hostInfo().equals(host) &&
                metadata.topicPartitions().stream()
                    .anyMatch(tp -> getPartitionForKey(key, tp.topic()) ==
tp.partition()));
    }

    private int getPartitionForKey(String key, String topic) {
        // Get partition for key (simplified)
        return Math.abs(key.hashCode()) % getPartitionCount(topic);
    }

    private int getPartitionCount(String topic) {
        // In real implementation, get from metadata
        return 4; // Assume 4 partitions
```

```
    }

    private String queryRemoteHost(HostInfo hostInfo, String userId) {
        try {
            // Make HTTP request to remote instance
            String url = String.format("http://%s:%d/user-count/%s",
                hostInfo.host(), hostInfo.port(), userId);

            // Use HTTP client to make request (simplified)
            return String.format("{\"userId\": \"%s\", \"redirected_to\":
\"%s:%d\"}",
                userId, hostInfo.host(), hostInfo.port());

        } catch (Exception e) {
            return "{\"error\": \"Failed to query remote host: " + e.getMessage()
+ "\"}";
        }
    }

    /*
     * Distributed Query Pattern:
     *
     * 1. Client sends query to any instance
     * 2. Instance determines which instance has the data
     * 3. If local: query local store directly
     * 4. If remote: forward request to correct instance
     * 5. Return result to client
     *
     * Benefits:
     * - Scalable across multiple instances
     * - Transparent to clients
     * - Fault tolerant (can retry on different instances)
     */
}
```

Comprehensive Java Examples

Complete Kafka Streams Application

```
import org.apache.kafka.streams.*;
import org.apache.kafka.streams.kstream.*;
import org.apache.kafka.streams.state.*;
import org.apache.kafka.common.serialization.*;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.core.JsonProcessingException;

import java.util.Properties;
import java.time.Duration;
import java.util.concurrent.CountDownLatch;
```

```
/**
 * Complete E-commerce Analytics Kafka Streams Application
 *
 * This application processes:
 * 1. User events (clicks, views, purchases)
 * 2. Product catalog updates
 * 3. User profiles
 *
 * And produces:
 * 1. Real-time user analytics
 * 2. Product recommendation updates
 * 3. Alert streams for anomalies
 */
public class EcommerceAnalyticsApplication {

    private static final ObjectMapper objectMapper = new ObjectMapper();

    public static void main(String[] args) {
        Properties props = createStreamConfig();

        Topology topology = buildTopology();
        KafkaStreams streams = new KafkaStreams(topology, props);

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

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

        // Start streams
        CountDownLatch latch = new CountDownLatch(1);
        Runtime.getRuntime().addShutdownHook(new Thread("shutdown-hook") {
            @Override
            public void run() {
                System.out.println("Shutting down streams...");
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.err.println("Application failed: " + e.getMessage());
```

```
            System.exit(1);
        }
    }

    private static Topology buildTopology() {
        StreamsBuilder builder = new StreamsBuilder();

        // === INPUT STREAMS AND TABLES ===

        // User events stream
        KStream<String, String> userEvents = builder.stream("user-events",
            Consumed.with(Serdes.String(), Serdes.String())
                .withTimestampExtractor(new UserEventTimestampExtractor()));

        // Product catalog table
        KTable<String, String> productCatalog = builder.table("product-catalog");

        // User profiles table
        GlobalKTable<String, String> userProfiles = builder.globalTable("user-
profiles");

        // === STREAM PROCESSING ===

        // 1. Parse and validate events
        KStream<String, UserEvent> parsedEvents = userEvents
            .mapValues(EcommerceAnalyticsApplication::parseUserEvent)
            .filter((key, event) -> event != null)
            .selectKey((key, event) -> event.getUserId());

        // 2. Enrich events with user profile data
        KStream<String, EnrichedUserEvent> enrichedEvents = parsedEvents.join(
            userProfiles,
            (userId, event) -> userId,
            (event, profile) -> enrichEventWithProfile(event, profile)
        );

        // 3. Enrich with product data for purchase events
        KStream<String, EnrichedUserEvent> fullyEnrichedEvents = enrichedEvents
            .filter((userId, event) -> "purchase".equals(event.getEventType()))
            .join(productCatalog,
                (userId, event) -> event.getProductId(),
                (event, product) -> enrichEventWithProduct(event, product))
            .merge(enrichedEvents.filter((userId, event) ->
!"purchase".equals(event.getEventType())));

        // === ANALYTICS AND AGGREGATIONS ===

        // 4. User session analytics (session windows)
        KTable<Windowed<String>, UserSessionAnalytics> userSessionAnalytics =
enrichedEvents
            .groupByKey()

.windowedBy(SessionWindows.with(Duration.ofMinutes(30)).grace(Duration.ofMinutes(5
)))
```

```
            .aggregate(
                UserSessionAnalytics::new,
                (userId, event, analytics) -> analytics.addEvent(event),
                (userId, analytics1, analytics2) -> analytics1.merge(analytics2),
                Materialized.<String, UserSessionAnalytics, SessionStore<Bytes,
byte[]>>as("user-sessions")
                    .withKeySerde(Serdes.String())
                    .withValueSerde(new UserSessionAnalyticsSerde()));

        // 5. Product popularity (tumbling windows)
        KTable<Windowed<String>, Long> productPopularity = parsedEvents
            .filter((userId, event) -> event.getProductId() != null)
            .selectKey((userId, event) -> event.getProductId())
            .groupByKey()
            .windowedBy(TimeWindows.of(Duration.ofHours(1)))
            .count(Materialized.as("product-popularity"));

        // 6. Revenue analytics (hopping windows for trending)
        KTable<Windowed<String>, Double> revenueAnalytics = parsedEvents
            .filter((userId, event) -> "purchase".equals(event.getEventType()))
            .mapValues(event -> event.getAmount())
            .groupBy((userId, amount) -> "total", Grouped.with(Serdes.String(),
Serdes.Double()))

.windowedBy(TimeWindows.of(Duration.ofHours(1)).advanceBy(Duration.ofMinutes(15)))
            .reduce(Double::sum, Materialized.as("revenue-analytics"));

        // === ANOMALY DETECTION ===

        // 7. Detect unusual user behavior
        KStream<String, String> anomalyAlerts = enrichedEvents
            .transform(() -> new AnomalyDetectionTransformer(), "user-behavior-
store");

        // Add anomaly detection state store
        builder.addStateStore(Stores
            .keyValueStoreBuilder(Stores.persistentKeyValueStore("user-behavior-
store"),
                Serdes.String(), Serdes.String())
            .withCachingEnabled());

        // === OUTPUT STREAMS ===

        // Output enriched events
        fullyEnrichedEvents
            .mapValues(EcommerceAnalyticsApplication::enrichedEventToJson)
            .to("enriched-user-events");

        // Output session analytics
        userSessionAnalytics.toStream()
            .map((windowedUserId, analytics) -> KeyValue.pair(
                windowedUserId.key(),
                sessionAnalyticsToJson(analytics)))
            .to("user-session-analytics");
```

```
        // Output product trends
        productPopularity.toStream()
            .map((windowedProductId, count) -> KeyValue.pair(
                windowedProductId.key(),
                String.format("
{\"productId\":\"%s\",\"count\":%d,\"window\":\"%s\"}",
                    windowedProductId.key(), count, windowedProductId.window())))
            .to("product-trends");

        // Output revenue analytics
        revenueAnalytics.toStream()
            .map((windowedKey, revenue) -> KeyValue.pair(
                "revenue",
                String.format("{\"revenue\":%.2f,\"window\":\"%s\"}", revenue,
windowedKey.window())))
            .to("revenue-analytics");

        // Output anomaly alerts
        anomalyAlerts.to("anomaly-alerts");

        return builder.build();
    }

    // === HELPER CLASSES AND METHODS ===

    static class UserEvent {
        private String userId;
        private String eventType;
        private String productId;
        private double amount;
        private long timestamp;

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

        public UserEvent(String userId, String eventType, String productId, double
amount, long timestamp) {
            this.userId = userId;
            this.eventType = eventType;
            this.productId = productId;
            this.amount = amount;
            this.timestamp = timestamp;
        }

        // 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 String getProductId() { return productId; }
        public void setProductId(String productId) { this.productId = productId; }
        public double getAmount() { return amount; }
        public void setAmount(double amount) { this.amount = amount; }
```

```
        public long getTimestamp() { return timestamp; }
        public void setTimestamp(long timestamp) { this.timestamp = timestamp; }
    }

    static class EnrichedUserEvent extends UserEvent {
        private String userSegment;
        private String productCategory;

        public EnrichedUserEvent(UserEvent event) {
            super(event.getUserId(), event.getEventType(), event.getProductId(),
                  event.getAmount(), event.getTimestamp());
        }

        public String getUserSegment() { return userSegment; }
        public void setUserSegment(String userSegment) { this.userSegment =
userSegment; }
        public String getProductCategory() { return productCategory; }
        public void setProductCategory(String productCategory) {
this.productCategory = productCategory; }
    }

    static class UserSessionAnalytics {
        private int eventCount = 0;
        private int purchaseCount = 0;
        private double totalSpent = 0.0;
        private java.util.Set<String> viewedProducts = new java.util.HashSet<>();

        public UserSessionAnalytics addEvent(EnrichedUserEvent event) {
            eventCount++;
            if ("purchase".equals(event.getEventType())) {
                purchaseCount++;
                totalSpent += event.getAmount();
            }
            if (event.getProductId() != null) {
                viewedProducts.add(event.getProductId());
            }
            return this;
        }

        public UserSessionAnalytics merge(UserSessionAnalytics other) {
            eventCount += other.eventCount;
            purchaseCount += other.purchaseCount;
            totalSpent += other.totalSpent;
            viewedProducts.addAll(other.viewedProducts);
            return this;
        }

        // Getters
        public int getEventCount() { return eventCount; }
        public int getPurchaseCount() { return purchaseCount; }
        public double getTotalSpent() { return totalSpent; }
        public java.util.Set<String> getViewedProducts() { return viewedProducts;
}
    }
```

```
    // Custom timestamp extractor
    static class UserEventTimestampExtractor implements TimestampExtractor {
        @Override
        public long extract(ConsumerRecord<Object, Object> record, long
previousTimestamp) {
            try {
                JsonNode json = objectMapper.readTree(record.value().toString());
                return json.has("timestamp") ? json.get("timestamp").asLong() :
record.timestamp();
            } catch (Exception e) {
                return record.timestamp();
            }
        }
    }

    // Anomaly detection transformer
    static class AnomalyDetectionTransformer
            implements Transformer<String, EnrichedUserEvent, KeyValue<String,
String>> {

        private KeyValueStore<String, String> behaviorStore;
        private ProcessorContext context;

        @Override
        public void init(ProcessorContext context) {
            this.context = context;
            this.behaviorStore = (KeyValueStore<String, String>)
                context.getStateStore("user-behavior-store");
        }

        @Override
        public KeyValue<String, String> transform(String userId, EnrichedUserEvent
event) {
            try {
                // Get user's recent behavior
                String recentBehavior = behaviorStore.get(userId);
                UserBehaviorProfile profile = recentBehavior != null ?
                    objectMapper.readValue(recentBehavior,
UserBehaviorProfile.class) :
                    new UserBehaviorProfile();

                // Check for anomalies
                String anomaly = detectAnomaly(profile, event);

                // Update behavior profile
                profile.addEvent(event);
                behaviorStore.put(userId,
objectMapper.writeValueAsString(profile));

                // Return anomaly alert if detected
                if (anomaly != null) {
                    return KeyValue.pair(userId, anomaly);
                }
```

```
                return null; // No anomaly

            } catch (Exception e) {
                System.err.println("Anomaly detection error: " + e.getMessage());
                return null;
            }
        }

        private String detectAnomaly(UserBehaviorProfile profile,
EnrichedUserEvent event) {
            // Detect unusual purchase amounts
            if ("purchase".equals(event.getEventType())) {
                double avgPurchase = profile.getAveragePurchaseAmount();
                if (avgPurchase > 0 && event.getAmount() > avgPurchase * 5) {
                    return String.format("Unusual large purchase: $%.2f (avg:
$%.2f)",
                            event.getAmount(), avgPurchase);
                }
            }

            // Detect rapid consecutive purchases
            if ("purchase".equals(event.getEventType()) &&
                profile.getLastPurchaseTime() > 0 &&
                event.getTimestamp() - profile.getLastPurchaseTime() < 60000) { //
1 minute
                return "Rapid consecutive purchases detected";
            }

            return null;
        }

        @Override
        public void close() {}
    }

    static class UserBehaviorProfile {
        private double totalSpent = 0;
        private int purchaseCount = 0;
        private long lastPurchaseTime = 0;

        public void addEvent(EnrichedUserEvent event) {
            if ("purchase".equals(event.getEventType())) {
                totalSpent += event.getAmount();
                purchaseCount++;
                lastPurchaseTime = event.getTimestamp();
            }
        }

        public double getAveragePurchaseAmount() {
            return purchaseCount > 0 ? totalSpent / purchaseCount : 0;
        }

        public long getLastPurchaseTime() { return lastPurchaseTime; }
```

```
    }

    // Helper methods for JSON serialization
    private static UserEvent parseUserEvent(String json) {
        try {
            return objectMapper.readValue(json, UserEvent.class);
        } catch (Exception e) {
            System.err.println("Failed to parse user event: " + e.getMessage());
            return null;
        }
    }

    private static EnrichedUserEvent enrichEventWithProfile(UserEvent event,
String profile) {
        EnrichedUserEvent enriched = new EnrichedUserEvent(event);
        try {
            JsonNode profileJson = objectMapper.readTree(profile);
            enriched.setUserSegment(profileJson.get("segment").asText("unknown"));
        } catch (Exception e) {
            enriched.setUserSegment("unknown");
        }
        return enriched;
    }

    private static EnrichedUserEvent enrichEventWithProduct(EnrichedUserEvent
event, String product) {
        try {
            JsonNode productJson = objectMapper.readTree(product);

event.setProductCategory(productJson.get("category").asText("unknown"));
        } catch (Exception e) {
            event.setProductCategory("unknown");
        }
        return event;
    }

    private static String enrichedEventToJson(EnrichedUserEvent event) {
        try {
            return objectMapper.writeValueAsString(event);
        } catch (Exception e) {
            return "{}";
        }
    }

    private static String sessionAnalyticsToJson(UserSessionAnalytics analytics) {
        try {
            return objectMapper.writeValueAsString(analytics);
        } catch (Exception e) {
            return "{}";
        }
    }

    // Custom Serde for UserSessionAnalytics
    static class UserSessionAnalyticsSerde implements Serde<UserSessionAnalytics>
```

```
{
        @Override
        public Serializer<UserSessionAnalytics> serializer() {
            return (topic, data) -> {
                try {
                    return objectMapper.writeValueAsBytes(data);
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
            };
        }

        @Override
        public Deserializer<UserSessionAnalytics> deserializer() {
            return (topic, data) -> {
                try {
                    return objectMapper.readValue(data,
UserSessionAnalytics.class);
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
            };
        }
    }

    private static Properties createStreamConfig() {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "ecommerce-analytics");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

        // Performance settings
        props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 4);
        props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 100 * 1024 *
1024); // 100MB
        props.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 10000); // 10 seconds

        // Exactly-once processing
        props.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG,
StreamsConfig.EXACTLY_ONCE_V2);

        // State directory
        props.put(StreamsConfig.STATE_DIR_CONFIG, "/tmp/kafka-streams-ecommerce");

        return props;
    }
}
```

⚖️ Comparisons & Trade-offs

Stream Processing Approaches Comparison

Approach	Latency	Throughput	Complexity	State Management	Use Cases
Kafka Streams	Low (ms)	High	Medium	Built-in (RocksDB)	Real-time analytics, microservices
Consumer Groups	Low (ms)	Very High	Low	Manual (external)	Simple transformations, routing
Apache Flink	Very Low (ms)	Very High	High	Advanced	Complex CEP, ML pipelines
Apache Storm	Very Low (ms)	High	High	Manual	Real-time alerts, monitoring
Spark Streaming	Medium (seconds)	Very High	Medium	Built-in	Batch + stream, ML

KStream vs KTable vs GlobalKTable

Feature	KStream	KTable	GlobalKTable
Data Model	Event stream	Changelog (latest per key)	Replicated changelog
Memory Usage	Low	Medium	High (full replication)
Join Requirements	Co-partitioned	Co-partitioned	No co-partitioning
Update Semantics	Append only	Upsert	Upsert (all instances)
Query Latency	N/A	Low (local)	Very Low (local)
Startup Time	Fast	Fast	Slow (full read)
Best For	Event processing	Aggregations	Reference data joins

Windowing Strategies Comparison

Window Type	Overlap	Use Case	Memory Usage	Latency
Tumbling	None	Fixed interval analytics	Low	Medium
Hopping	Yes	Smooth trending, moving averages	High	Medium
Session	Variable	User session analytics	Medium	High
Sliding	Continuous	Real-time monitoring	Very High	Low

🚨 Common Pitfalls & Best Practices

1. State Store Issues

❌ Not Handling Large State

```
// DON'T - Unbounded state growth
KTable<String, Set<String>> userActivities = events
    .groupByKey()
    .aggregate(
        HashSet::new,
        (key, value, set) -> {
            set.add(value); // Grows indefinitely!
            return set;
        },
        Materialized.as("user-activities"));
```

```
// DO - Implement state cleanup or use windowed stores
KTable<Windowed<String>, Set<String>> userActivities = events
    .groupByKey()
    .windowedBy(TimeWindows.of(Duration.ofDays(1))) // Bounded by time
    .aggregate(
        HashSet::new,
        (key, value, set) -> {
            set.add(value);
            return set;
        },
        Materialized.as("user-activities"));

// Or implement custom cleanup
KTable<String, Set<String>> userActivitiesWithCleanup = events
    .groupByKey()
    .aggregate(
        HashSet::new,
        (key, value, set) -> {
            set.add(value);
            // Keep only recent items
            if (set.size() > 1000) {
                Set<String> trimmed = new HashSet<>(
                    set.stream().skip(set.size() -
800).collect(Collectors.toList()));
                return trimmed;
            }
            return set;
        },
        Materialized.as("user-activities-bounded"));
```

2. Serialization Problems

❌ Using Default Java Serialization

```
// DON'T - Non-portable, version-dependent
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.Serdes$StringSerde");
```

```
// DO - Use schema registry or JSON with proper versioning
public class AvroSerdeExample {

    public static Properties createAvroConfig() {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "avro-example");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

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

        // Use Avro Serdes
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
            "io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde");
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
            "io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde");

        return props;
    }
}

// Alternative: JSON with versioning
public class VersionedJsonSerde<T> implements Serde<T> {
    private final Class<T> targetClass;
    private final ObjectMapper objectMapper;

    public VersionedJsonSerde(Class<T> targetClass) {
        this.targetClass = targetClass;
        this.objectMapper = new ObjectMapper();
        // Configure for backward compatibility
        objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false);
    }

    @Override
    public Serializer<T> serializer() {
        return (topic, data) -> {
            try {
                // Add version field
                ObjectNode jsonNode = objectMapper.valueToTree(data);
                jsonNode.put("_version", "1.0");
                return objectMapper.writeValueAsBytes(jsonNode);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        };
```

```
    }

    @Override
    public Deserializer<T> deserializer() {
        return (topic, data) -> {
            try {
                JsonNode jsonNode = objectMapper.readTree(data);
                String version = jsonNode.has("_version") ?
                    jsonNode.get("_version").asText() : "1.0";

                // Handle version-specific deserialization
                return objectMapper.treeToValue(jsonNode, targetClass);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        };
    }
}
```

3. Join Problems

❌ Not Ensuring Co-partitioning

```
// DON'T - Streams not co-partitioned
KStream<String, String> orders = builder.stream("orders"); // 6 partitions
KStream<String, String> payments = builder.stream("payments"); // 4 partitions

// This will fail at runtime!
KStream<String, String> joined = orders.join(payments, /* ... */);
```

```
// DO - Ensure co-partitioning
KStream<String, String> orders = builder.stream("orders");
KStream<String, String> payments = builder.stream("payments");

// Repartition to ensure same partitioning
KStream<String, String> repartitionedPayments = payments
    .selectKey((key, value) -> extractOrderId(value)) // Same key as orders
    .repartition(Repartitioned.<String, String>as("payments-repartitioned")
        .withNumberOfPartitions(6)); // Same partition count as orders

KStream<String, String> joined = orders.join(repartitionedPayments,
    (orderValue, paymentValue) -> orderValue + ":" + paymentValue,
    JoinWindows.of(Duration.ofMinutes(5)));
```

4. Window and Time Issues

❌ Not Handling Late Data

```
// DON'T - No grace period for late events
TimeWindows window = TimeWindows.of(Duration.ofMinutes(5)); // No grace period

KTable<Windowed<String>, Long> counts = stream
    .groupByKey()
    .windowedBy(window)
    .count(); // Late events will be dropped
```

```
// DO - Add appropriate grace period
TimeWindows window = TimeWindows.of(Duration.ofMinutes(5))
    .grace(Duration.ofMinutes(2)); // Allow 2 minutes for late events

KTable<Windowed<String>, Long> counts = stream
    .groupByKey()
    .windowedBy(window)
    .count();

// Also consider suppression for final results
KTable<Windowed<String>, Long> finalCounts = counts
    .suppress(Suppressed.untilWindowCloses(Suppressed.BufferConfig.unbounded()));
```

Best Practices Summary

☑ Configuration Best Practices

1. **Use exactly-once processing** for critical applications
2. **Configure appropriate batch sizes** for your throughput needs
3. **Set reasonable cache sizes** to balance memory and performance
4. **Use static membership** for stable deployments
5. **Configure proper replication factors** for state stores

☑ Development Best Practices

1. **Design for idempotence** - operations should be safe to retry
2. **Handle schema evolution** - use versioned serialization
3. **Implement proper error handling** - distinguish transient vs permanent errors
4. **Use meaningful store names** - for debugging and monitoring
5. **Add comprehensive logging** - include topology and partition info

☑ Operational Best Practices

1. **Monitor lag and processing rates** - set up alerting
2. **Test with realistic data volumes** - performance characteristics change
3. **Plan for scaling** - understand partition limits
4. **Backup state directories** - for disaster recovery
5. **Use interactive queries** - for debugging and monitoring

Real-World Use Cases

1. Fraud Detection System

```
@Service
public class FraudDetectionStreams {

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

        // Transaction stream
        KStream<String, Transaction> transactions = builder
            .stream("transactions", Consumed.with(Serdes.String(), new JsonSerde<>
(Transaction.class)));

        // User profiles
        GlobalKTable<String, UserProfile> userProfiles = builder
            .globalTable("user-profiles", Consumed.with(Serdes.String(), new
JsonSerde<>(UserProfile.class)));

        // Real-time fraud scoring
        KStream<String, FraudAlert> fraudAlerts = transactions
            .join(userProfiles,
                (txnKey, txn) -> txn.getUserId(),
                (txn, profile) -> calculateFraudScore(txn, profile))
            .filter((key, score) -> score.getRiskScore() > 0.8)
            .mapValues(score -> new FraudAlert(score.getTransactionId(),
score.getRiskScore()));

        // Windowed analysis for velocity checks
        KTable<Windowed<String>, TransactionSummary> velocityAnalysis =
transactions
            .groupBy((key, txn) -> txn.getUserId())
            .windowedBy(TimeWindows.of(Duration.ofMinutes(10)))
            .aggregate(
                TransactionSummary::new,
                (userId, txn, summary) -> summary.addTransaction(txn),
                Materialized.as("velocity-analysis"));

        // High-velocity alerts
        velocityAnalysis.toStream()
            .filter((windowedUserId, summary) -> summary.getTransactionCount() >
10 || summary.getTotalAmount() > 10000)
            .map((windowedUserId, summary) -> KeyValue.pair(
                windowedUserId.key(),
                new FraudAlert("VELOCITY_" + windowedUserId.key(), 0.9)))
            .to("fraud-alerts");

        fraudAlerts.to("fraud-alerts");

        return builder.build();
```

```
    }

    private FraudScore calculateFraudScore(Transaction txn, UserProfile profile) {
        double riskScore = 0.0;

        // Amount-based risk
        if (txn.getAmount() > profile.getAverageTransactionAmount() * 5) {
            riskScore += 0.3;
        }

        // Location-based risk
        if (!txn.getLocation().equals(profile.getHomeLocation())) {
            riskScore += 0.2;
        }

        // Time-based risk
        if (isUnusualTime(txn.getTimestamp(), profile.getUsualActivityHours())) {
            riskScore += 0.1;
        }

        return new FraudScore(txn.getId(), riskScore);
    }
}
```

2. IoT Sensor Data Processing

```
@Service
public class IoTDataProcessing {

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

        // Raw sensor data
        KStream<String, SensorReading> sensorData = builder
            .stream("sensor-readings", Consumed.with(Serdes.String(), new
JsonSerde<>(SensorReading.class)));

        // Device configuration
        GlobalKTable<String, DeviceConfig> deviceConfigs = builder
            .globalTable("device-configs");

        // Validate and enrich sensor data
        KStream<String, EnrichedSensorReading> enrichedData = sensorData
            .join(deviceConfigs,
                (sensorId, reading) -> reading.getDeviceId(),
                (reading, config) -> enrichReading(reading, config))
            .filter((sensorId, enrichedReading) -> enrichedReading.isValid());

        // Real-time anomaly detection
        KStream<String, Alert> anomalies = enrichedData
            .transform(() -> new AnomalyDetector(), "sensor-history");
```

```
        // Windowed aggregations for monitoring
        KTable<Windowed<String>, SensorStats> sensorStats = enrichedData
            .groupBy((sensorId, reading) -> reading.getSensorType())
            .windowedBy(TimeWindows.of(Duration.ofMinutes(5)))
            .aggregate(
                SensorStats::new,
                (sensorType, reading, stats) -> stats.addReading(reading),
                Materialized.as("sensor-statistics"));

        // Predictive maintenance alerts
        KStream<String, MaintenanceAlert> maintenanceAlerts = enrichedData
            .filter((sensorId, reading) ->
reading.getSensorType().equals("vibration"))
            .transform(() -> new PredictiveMaintenanceTransformer(), "maintenance-
models");

        // Output streams
        anomalies.to("iot-anomalies");
        sensorStats.toStream().map(this::formatSensorStats).to("sensor-
dashboard");
        maintenanceAlerts.to("maintenance-alerts");

        return builder.build();
    }

    private EnrichedSensorReading enrichReading(SensorReading reading,
DeviceConfig config) {
        EnrichedSensorReading enriched = new EnrichedSensorReading(reading);
        enriched.setLocation(config.getLocation());
        enriched.setThresholds(config.getThresholds());
        enriched.setValid(isWithinThresholds(reading, config));
        return enriched;
    }
}
```

3. Social Media Stream Analytics

```
@Service
public class SocialMediaAnalytics {

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

        // Social media posts
        KStream<String, SocialPost> posts = builder
            .stream("social-posts", Consumed.with(Serdes.String(), new JsonSerde<>
(SocialPost.class)));

        // User profiles
        KTable<String, UserProfile> userProfiles = builder
```

```
            .table("user-profiles");

        // Real-time sentiment analysis
        KStream<String, SentimentAnalysis> sentimentStream = posts
            .mapValues(post -> analyzeSentiment(post))
            .filter((postId, sentiment) -> sentiment != null);

        // Trending topics (session windows for bursty topics)
        KTable<Windowed<String>, Long> trendingTopics = posts
            .flatMapValues(post -> extractHashtags(post.getContent()))
            .groupBy((postId, hashtag) -> hashtag)
            .windowedBy(SessionWindows.with(Duration.ofMinutes(30)))
            .count(Materialized.as("trending-topics"));

        // Influential users (hopping windows for smooth trends)
        KTable<Windowed<String>, UserInfluence> influentialUsers = posts
            .join(userProfiles, (post, profile) -> enrichPostWithProfile(post,
profile))
            .groupBy((postId, enrichedPost) -> enrichedPost.getUserId())

.windowedBy(TimeWindows.of(Duration.ofHours(1)).advanceBy(Duration.ofMinutes(15)))
            .aggregate(
                UserInfluence::new,
                (userId, enrichedPost, influence) ->
influence.addPost(enrichedPost),
                Materialized.as("user-influence"));

        // Real-time brand monitoring
        KStream<String, BrandMention> brandMentions = posts
            .flatMap((postId, post) -> extractBrandMentions(post))
            .join(sentimentStream,
                (mention, sentiment) -> mention.withSentiment(sentiment),
                JoinWindows.of(Duration.ofMinutes(1)));

        // Crisis detection (rapid negative sentiment)
        KTable<Windowed<String>, CrisisAlert> crisisDetection = brandMentions
            .filter((mentionId, mention) -> mention.getSentiment().isNegative())
            .groupBy((mentionId, mention) -> mention.getBrand())
            .windowedBy(TimeWindows.of(Duration.ofMinutes(15)))
            .aggregate(
                CrisisAlert::new,
                (brand, mention, alert) -> alert.addNegativeMention(mention),
                Materialized.as("crisis-detection"))
            .filter((windowedBrand, alert) -> alert.isThresholdExceeded());

        // Output streams
        trendingTopics.toStream()
            .filter((windowedTopic, count) -> count > 100)
            .map(this::formatTrendingTopic)
            .to("trending-dashboard");

        influentialUsers.toStream()
            .filter((windowedUser, influence) -> influence.getInfluenceScore() >
0.8)
```

```
            .map(this::formatInfluentialUser)
            .to("influencer-dashboard");

        brandMentions.to("brand-monitoring");

        crisisDetection.toStream()
            .map(this::formatCrisisAlert)
            .to("crisis-alerts");

        return builder.build();
    }

    private SentimentAnalysis analyzeSentiment(SocialPost post) {
        // Use ML model or sentiment analysis service
        double sentimentScore =
sentimentAnalysisService.analyze(post.getContent());
        return new SentimentAnalysis(post.getId(), sentimentScore);
    }

    private List<String> extractHashtags(String content) {
        return Arrays.stream(content.split("\\s+"))
            .filter(word -> word.startsWith("#"))
            .map(String::toLowerCase)
            .collect(Collectors.toList());
    }
}
```

4. Real-time Recommendation Engine

```
@Service
public class RecommendationEngine {

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

        // User behavior events
        KStream<String, UserEvent> userEvents = builder
            .stream("user-events", Consumed.with(Serdes.String(), new JsonSerde<>
(UserEvent.class)));

        // Product catalog
        GlobalKTable<String, Product> productCatalog = builder
            .globalTable("product-catalog");

        // User profiles with preferences
        KTable<String, UserProfile> userProfiles = builder
            .table("user-profiles", Materialized.as("user-profiles-store"));

        // Real-time user session tracking
        KTable<Windowed<String>, UserSession> userSessions = userEvents
            .groupByKey()
```

```
            .windowedBy(SessionWindows.with(Duration.ofMinutes(30)))
            .aggregate(
                UserSession::new,
                (userId, event, session) -> session.addEvent(event),
                (userId, session1, session2) -> session1.merge(session2),
                Materialized.as("user-sessions"));

        // Product affinity calculation
        KTable<String, ProductAffinity> productAffinities = userEvents
            .filter((userId, event) -> event.getEventType().equals("view") ||
event.getEventType().equals("purchase"))
            .groupByKey()
            .aggregate(
                ProductAffinity::new,
                (userId, event, affinity) -> affinity.addEvent(event),
                Materialized.as("product-affinities"));

        // Generate recommendations in real-time
        KStream<String, Recommendation> recommendations = userEvents
            .filter((userId, event) -> event.getEventType().equals("view"))
            .join(userProfiles, (event, profile) -> new UserContext(event,
profile))
            .transform(() -> new RecommendationGenerator(), "recommendation-
models");

        // Collaborative filtering updates
        KTable<String, CollaborativeModel> collaborativeModels = userEvents
            .filter((userId, event) -> event.getEventType().equals("purchase"))
            .groupBy((userId, event) -> event.getProductId())
            .aggregate(
                CollaborativeModel::new,
                (productId, event, model) -> model.addPurchase(event.getUserId()),
                Materialized.as("collaborative-models"));

        // A/B testing for recommendations
        KStream<String, RecommendationTest> abTestResults = recommendations
            .transform(() -> new ABTestTransformer(), "ab-test-assignments");

        // Output streams
        recommendations.to("user-recommendations");
        abTestResults.to("ab-test-results");

        // Update models based on user feedback
        KStream<String, ModelUpdate> modelUpdates = builder
            .stream("user-feedback")
            .transform(() -> new ModelUpdateTransformer(), "recommendation-
models");

        modelUpdates.to("model-updates");

        return builder.build();
    }

    // Recommendation generator transformer
```

```
    static class RecommendationGenerator implements Transformer<String,
UserContext, KeyValue<String, Recommendation>> {
        private KeyValueStore<String, String> modelStore;
        private ProcessorContext context;

        @Override
        public void init(ProcessorContext context) {
            this.context = context;
            this.modelStore = (KeyValueStore<String, String>)
context.getStateStore("recommendation-models");
        }

        @Override
        public KeyValue<String, Recommendation> transform(String userId,
UserContext userContext) {
            try {
                // Get user's ML model
                String modelJson = modelStore.get(userId);
                UserRecommendationModel model = modelJson != null ?
                    parseModel(modelJson) : new UserRecommendationModel();

                // Generate recommendations based on current context
                List<String> recommendations =
model.generateRecommendations(userContext);

                if (!recommendations.isEmpty()) {
                    Recommendation recommendation = new Recommendation(
                        userId,
                        recommendations,
                        System.currentTimeMillis()
                    );

                    return KeyValue.pair(userId, recommendation);
                }

                return null;

            } catch (Exception e) {
                System.err.println("Recommendation generation failed: " +
e.getMessage());
                return null;
            }
        }

        @Override
        public void close() {}
    }
}
```

Version Highlights

Kafka Streams Evolution Timeline

Version	Release Date	Major Features
4.0	March 2025	New consumer protocol, enhanced cooperative rebalancing
3.0	September 2021	Improved performance, better error handling
2.8	April 2021	KRaft support, improved state stores
2.4	December 2019	Cooperative rebalancing, improved joins
2.1	July 2018	Named topologies, improved suppression
1.1	March 2018	Exactly-once semantics improvements
0.11	June 2017	**Exactly-once processing**, headers support
0.10	May 2016	**Initial Kafka Streams release**, basic DSL

Key Features by Version

Kafka Streams 4.0 (March 2025)

- ✨ **Enhanced Consumer Protocol**: 50% faster rebalancing with new consumer group protocol
- ✨ **Improved State Recovery**: Parallel state restoration for faster startup
- ✨ **Better Resource Management**: Dynamic thread scaling improvements
- ✨ **Java 17+ Required**: Modern JVM features and performance

Kafka Streams 3.x Series

- **3.6** (October 2023): Improved metrics and monitoring capabilities
- **3.5** (June 2023): Enhanced windowing performance
- **3.4** (February 2023): Better state store management
- **3.3** (October 2022): Improved cooperative rebalancing
- **3.2** (May 2022): Interactive queries enhancements
- **3.1** (January 2022): Performance optimizations
- **3.0** (September 2021): **Major refactoring** for better performance

Kafka Streams 2.x Highlights

- **2.8** (April 2021): Early KRaft support, RocksDB optimizations
- **2.7** (December 2020): Improved error handling, better windowing
- **2.6** (August 2020): Enhanced state store APIs
- **2.5** (April 2020): Improved exactly-once semantics (EOS v2)
- **2.4** (December 2019): **Cooperative rebalancing**, sticky partition assignor

Kafka Streams 1.x and 0.x

- **1.1** (March 2018): Exactly-once improvements, better debugging
- **1.0** (July 2017): Production-ready release, API stabilization
- **0.11** (June 2017): **Exactly-once processing**, headers support

- **0.10** (May 2016): **First release**, basic stream processing DSL

Current Best Practices (2025)

```
// Modern Kafka Streams configuration (4.0+)
public static Properties modernStreamsConfig() {
    Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "modern-streams-app");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

    // Kafka 4.0 optimizations
    props.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG,
StreamsConfig.EXACTLY_ONCE_V2);
    props.put(StreamsConfig.REPLICATION_FACTOR_CONFIG, 3);

    // Enhanced performance settings
    props.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 4);
    props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 100 * 1024 * 1024);
// 100MB
    props.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 5000); // 5 seconds

    // State management
    props.put(StreamsConfig.STATE_DIR_CONFIG, "/var/kafka-streams");
    props.put(StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG,
CustomRocksDBConfig.class);

    return props;
}
```

Breaking Changes Timeline

4.0 Breaking Changes

- Minimum Java version: 11 → 17
- Removed deprecated APIs (pre-3.6)
- Changed default rebalancing protocol

3.0 Breaking Changes

- Major internal refactoring
- Some metric names changed
- Deprecated old consumer protocol

2.0 Breaking Changes

- Java 8 minimum requirement

- Updated dependencies
- API cleanup from 1.x

1.0 Breaking Changes

- API stabilization from 0.x
- Package reorganization
- Serde interface changes

🔗 Additional Resources

📚 Official Documentation

- Kafka Streams Developer Guide
- Kafka Streams DSL API
- Processor API

🎓 Learning Resources

- Confluent Kafka Streams Tutorial
- Apache Kafka Streams Examples
- Kafka Streams in Action Book

🔧 Tools & Libraries

- Schema Registry - For Avro/JSON Schema management
- KSQL/ksqlDB - SQL interface for stream processing
- Kafka Streams Test Utils

📊 Monitoring & Operations

- Kafka Streams Metrics
- JMX Monitoring
- Confluent Control Center

🐛 Troubleshooting

- Kafka Streams FAQ
- Performance Tuning Guide
- Common Issues and Solutions

Last Updated: September 2025
Kafka Version: 4.0.0
Java Compatibility: 11+ (clients), 17+ (recommended)

> 💡 **Pro Tip**: Start with Kafka Streams for stateful stream processing needs. Use exactly-once processing for critical applications, and monitor state store sizes carefully. Interactive queries provide powerful real-time access to your application state.