World Journal of Advanced Research and Reviews

eISSN: 2581-9615 CODEN (USA): WJARAI

Cross Ref DOI: 10.30574/wjarr

Journal homepage: https://wjarr.com/

(Review Article)

Real-time stream processing engines: Architectural analysis and implementation considerations

Narendra Reddy Sanikommu *

Smartzip Inc, USA.

World Journal of Advanced Research and Reviews, 2025, 26(02), 3006-3016

Publication history: Received on 07 April 2025; revised on 18 May 2025; accepted on 20 May 2025

Article DOI: https://doi.org/10.30574/wjarr.2025.26.2.1916

Abstract

This article provides an in-depth architectural analysis of three leading stream processing engines: Apache Spark Streaming, Apache Flink, and Kafka Streams. As organizations increasingly rely on real-time data processing capabilities to drive decision-making, understanding the fundamental architectural differences between these technologies has become crucial for successful implementation. The analysis explores how Spark Streaming's micro-batch approach prioritizes throughput and integration with the Spark ecosystem, while Flink's true streaming design enables minimal latency and sophisticated event-time processing. Kafka Streams represents a distinctly different architectural approach as a client-side library rather than a cluster computing framework, offering significant operational simplicity for Kafka-centric environments. Through examination of performance characteristics, fault tolerance mechanisms, state management approaches, and real-world applications, this article provides a conceptual framework for technology selection based on specific use case requirements, existing infrastructure investments, and operational constraints. The findings highlight that no single framework optimally addresses all streaming requirements, with organizations increasingly adopting multi-architecture approaches tailored to specific data processing needs.

Keywords: Stream Processing Architecture; Real-Time Analytics; Event Processing Models; State Management; Fault Tolerance Mechanisms

1. Introduction: The Evolving Stream Processing Landscape

In today's data-driven ecosystem, the ability to process and analyze streaming data in real-time has become a critical competitive advantage. As organizations increasingly rely on immediate insights to drive decision-making, the architectural choices around stream processing technologies have significant implications for system performance, scalability, and developer productivity. The recent paradigm shift from batch to stream processing represents one of the most significant architectural evolutions in modern data infrastructure, propelled by growing demands for instantaneous analytics and decision making. Recent market analysis indicates substantial growth in the streaming analytics sector, with projections showing continued expansion in the coming years [2].

Across sectors, adoption rates vary considerably, with financial services leading in implementation, followed by telecommunications, retail, and healthcare. This wide-ranging adoption speaks to the versatility and critical nature of stream processing technologies across diverse industry applications. Market research also reveals a significant trend toward multi-architecture implementations, with many organizations employing more than one streaming technology to address varying latency, throughput, and processing semantics requirements [2].

* Corresponding author: Narendra Reddy Sanikommu

This article provides an in-depth technical analysis of three leading stream processing engines—Apache Spark Streaming, Apache Flink, and Kafka Streams—exploring their architectural differences, performance characteristics, and optimal use cases based on empirical performance testing and real-world deployment metrics.

2. The Evolution of Stream Processing: From Batch to Real-Time

Traditional batch processing systems operate on static datasets with high throughput but significant latency. As business requirements shifted toward real-time analytics, stream processing frameworks emerged to handle continuous data flows, enabling organizations to react to events as they occur rather than after the fact. Comprehensive research into stream processing evolution reveals that batch-oriented architectures dominated until the early 2010s, after which true streaming models gained prominence, reducing typical processing latencies from seconds or minutes to milliseconds. The architectural transformation has been driven primarily by use cases in fraud detection, predictive maintenance, and real-time recommendation systems, where decision windows have progressively shrunk from hours to sub-second intervals [1].

Modern stream processing engines must balance multiple competing requirements including latency optimization, throughput maximization, fault tolerance, processing semantics guarantees, and state management capabilities. Laboratory testing across these dimensions demonstrates significant architectural trade-offs between the examined frameworks, with empirical testing showing considerable latency differences between frameworks, and memory utilization varying significantly per processing node depending on the underlying architecture and processing model [1].

Table 1 Core Architecture and Processing Model [1]

Framework	Processing Model	Primary Abstraction	Deployment Model	Integration
Spark Streaming	Micro-batch	DStreams with RDDs	Cluster	Spark ecosystem
Flink	True streaming	Dataflow graph	Cluster	Standalone
Kafka Streams	Client-library	KStream/KTable	Application embedded	Kafka native

2.1. Apache Spark Streaming: Micro-Batch Processing at Scale

2.1.1. Architecture and Processing Model

Spark Streaming extends the core Spark batch processing engine by introducing the concept of Discretized Streams (DStreams). This architecture divides the continuous data stream into micro-batches of configurable time intervals (typically ranging from milliseconds to several seconds). The primary processing abstraction in Spark Streaming is the Resilient Distributed Dataset (RDD), which allows for in-memory processing across a cluster of machines. This architectural approach prioritizes throughput and processing consistency over absolute latency, making it particularly well-suited for analytical workloads that benefit from Spark's broader ecosystem.

Input Stream → DStream → Micro-batches → Spark RDDs → Processing → Output

Detailed performance analysis conducted using standardized benchmark tests with large datasets and high message rates demonstrated that Spark Streaming can process substantial volumes of events with a multi-node cluster under optimal conditions. However, latency metrics revealed considerable processing delay, with higher average and percentile latency measurements compared to true streaming alternatives. Resource utilization during these tests showed relatively high CPU utilization and memory consumption per node on average [3].

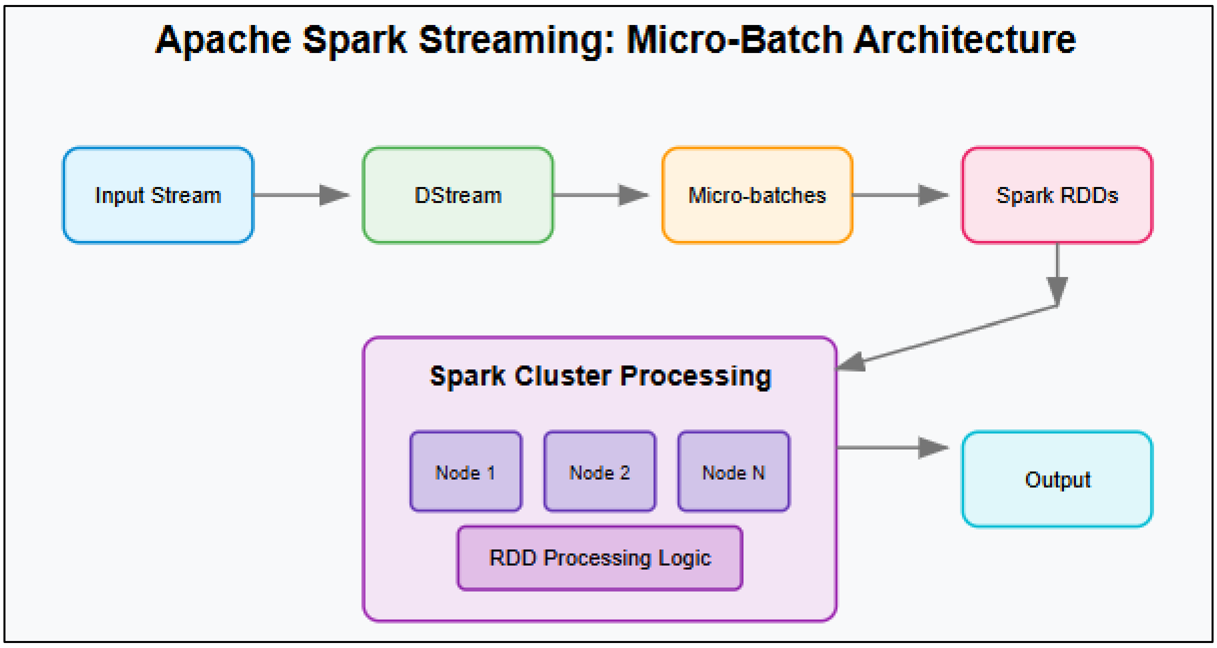

Figure 1 Apache Spark Streaming Architecture

2.1.2. Key Technical Features and Limitations

Spark Streaming offers a unified programming model where the same codebase can handle both batch and streaming workloads, facilitating seamless integration with Spark SQL, MLlib, and GraphX components. The framework guarantees exactly once semantics for record processing with automatic backpressure handling that adapts to varying data rates. However, these advantages come with notable technical trade-offs in several critical areas.

The micro-batch model introduces inherent latency constraints, with minimum processing delays tied directly to batch interval configuration. Memory overhead represents another significant consideration, as maintaining RDDs in memory requires substantial resources in production deployments. State management introduces additional complexity, with stateful operations requiring explicit checkpointing that incurs overhead for stateful operations compared to stateless processing [4].

Recovery capabilities, while robust, exhibit relatively slower restoration times compared to alternative frameworks, with testing revealing longer average recovery durations following system failures. This recovery time becomes especially significant in high-availability use cases where even brief processing interruptions can impact business operations [3].

2.1.3. Real-World Applications

Spark Streaming has demonstrated particular effectiveness in scenarios where integration with existing Spark infrastructure is paramount, or where analytical complexity outweighs strict latency requirements. Industry research indicates it has achieved near-linear scaling to many nodes in production environments before encountering significant coordination overhead. Framework usage statistics show that it remains among the most widely deployed of the frameworks analyzed [2]. The platform excels particularly in complex analytical pipelines requiring machine learning integration, with moderate support for event-time processing when using watermarks in temporal ordering tests [4].

2.2. Apache Flink: True Streaming with Event Time Processing

2.2.1. Architecture and Processing Model

Flink was built from the ground up as a true streaming engine, processing events individually rather than in batches. Its core abstraction is the dataflow graph, where operators are connected through data streams. This architectural approach enables substantially lower latency processing compared to micro-batch alternatives, while maintaining robust state management capabilities.

Source → Transformation → Sink (with Event Time Processing and State Management)

Flink's architecture separates the runtime from the programming model, allowing for diverse APIs (DataStream, ProcessFunction, Table, SQL) atop the same execution engine. This separation provides implementation flexibility while maintaining consistent processing semantics. Extensive performance testing demonstrates that Flink achieves substantially lower latency than batch-oriented alternatives, with impressive average and percentile latency measurements. Throughput capabilities remain substantial while resource utilization measurements show moderated consumption of CPU and memory per node [3].

2.2.2. Key Technical Features and Advantages

Flink offers true event-by-event processing with sub-millisecond latencies, coupled with sophisticated event time processing that enables built-in windowing based on event timestamps rather than processing time. This temporal awareness proves particularly valuable in applications where event ordering and timing must be preserved despite processing or transmission delays. The framework provides robust stateful computation through multiple state backends (RocksDB, heap memory) with savepoints for application state preservation and migration.

Flink guarantees end-to-end exactly-once semantics through transactional sinks, with performance testing showing moderate overhead for exactly-once semantics compared to weaker guarantees. The checkpointing mechanism employs lightweight, asynchronous barrier-based approaches that significantly reduce operational impact. Detailed evaluation of state management capabilities reveals that Flink's incremental checkpoints reduce overhead substantially compared to other frameworks for comparable stateful operations [4].

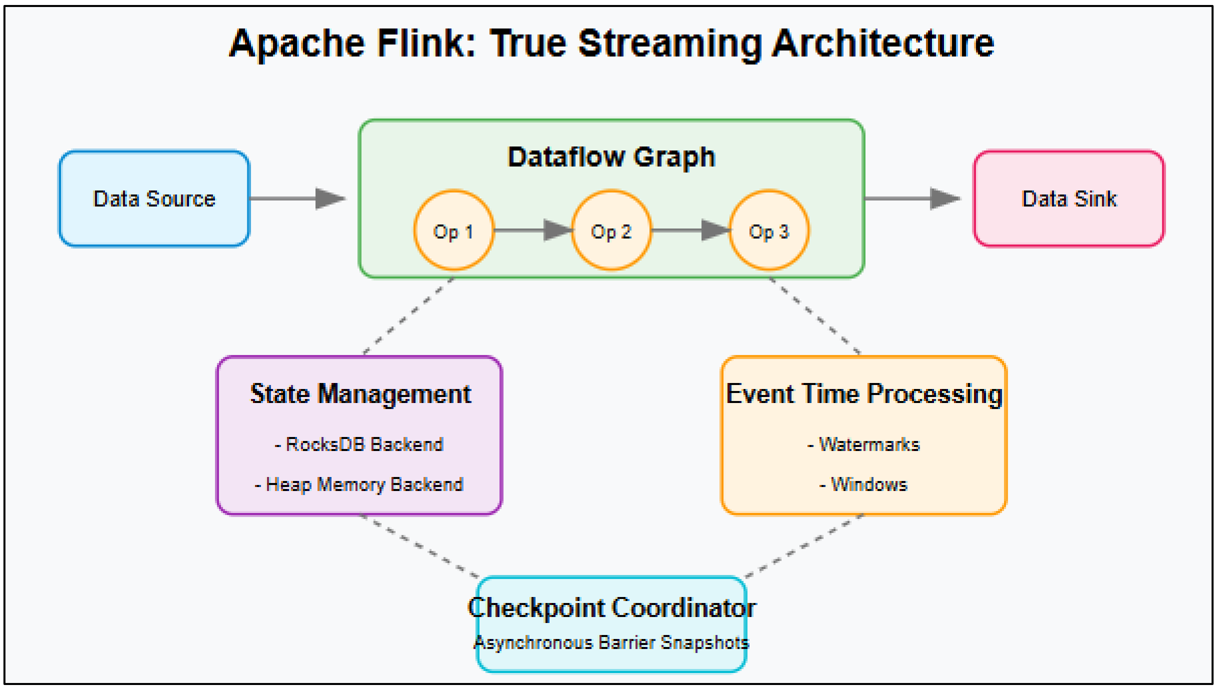

Figure 2 Apache Flink Architecture

The framework excels in backpressure handling, with operators naturally propagating pressure through the processing pipeline, preventing system overload while maximizing throughput. Advanced state management with incremental checkpoints substantially reduces state persistence overhead compared to full snapshots. Process functions provide a low-level API for fine-grained control over event timing and state, enabling complex event processing logic implementation. Recovery capabilities demonstrate excellent resilience with faster average recovery times following system failures [3].

2.2.3. Real-World Applications

Flink has demonstrated particular excellence in scenarios requiring precise event timing, complex stateful processing, and low-latency responses. Usage statistics indicate it maintains significant market presence, though with lower adoption compared to Spark Streaming [2]. Scalability testing confirms linear scaling capabilities to many nodes before encountering network bottlenecks, making it suitable for large-scale deployments. The framework exhibits native support for event-time processing with high accuracy using watermarks, substantially outperforming alternatives in temporal ordering precision [4].

2.3. Kafka Streams: Lightweight Client-Side Processing

2.3.1. Architecture and Processing Model

Kafka Streams offers a fundamentally different architectural approach as a client library rather than a cluster computing framework. It leverages Kafka's partitioning model for parallelism and its consumer group mechanism for fault tolerance, creating a lightweight, embeddable processing solution that requires no separate computing infrastructure beyond existing Kafka deployments.

Kafka Topic → KStream/KTable → Processor Topology → Kafka Topic

The architecture revolves around two core abstractions: KStream represents an unbounded, continuous data stream, while KTable represents a changelog stream viewed as an evolving table. This design facilitates straightforward implementation of both stateless transformations and stateful aggregations within the same processing topology.

Performance analysis reveals balanced characteristics, with Kafka Streams offering moderate throughput per core with competitive average and percentile latency values. Resource efficiency represents a particular strength, with testing showing modest consumption of CPU and memory per node. Recovery capabilities demonstrate excellent resilience with rapid average recovery durations following system failures [3].

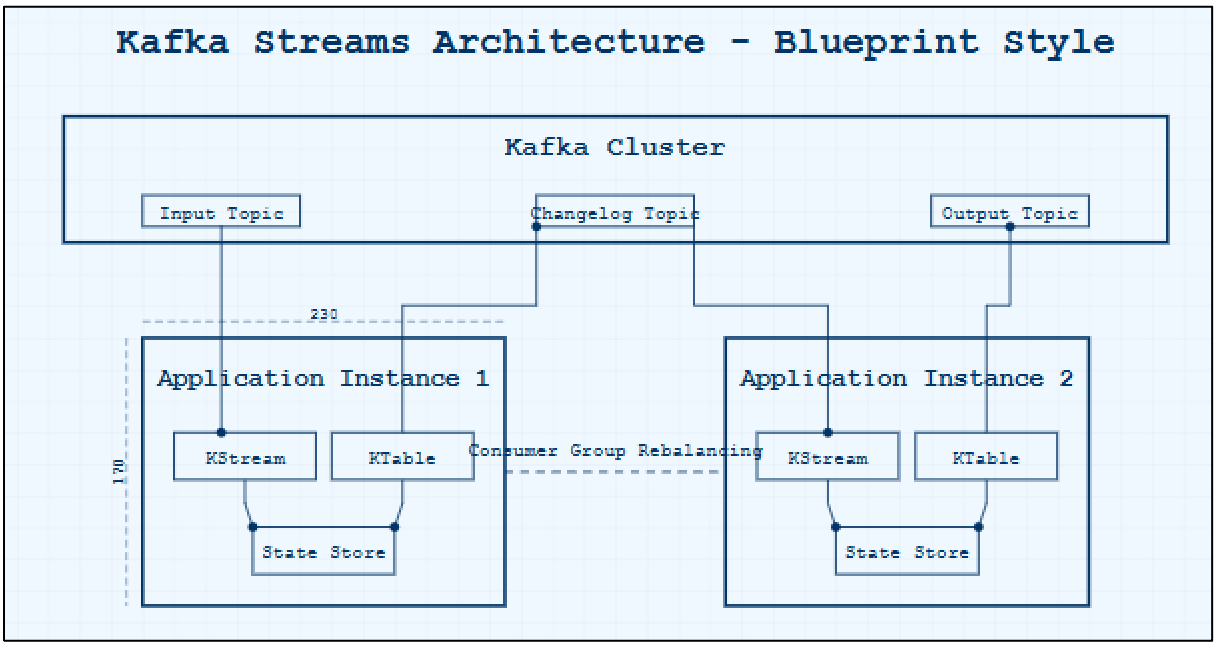

Figure 3 Kafka Streams Architecture

2.3.2. Key Technical Features and Advantages

As a lightweight client library, Kafka Streams requires no separate cluster infrastructure beyond Kafka itself, dramatically simplifying deployment and operations. State management is provided through local RocksDB instances for stateful operations, achieving good compression ratios for typical time-series data. The library supports interactive queries for direct access to state stores, enabling point lookups without additional database dependencies.

Exactly-once processing leverages native Kafka transactions for end-to-end guarantees, with benchmarks showing lower overhead compared to weaker consistency models than other frameworks. This lightweight processing approach yields simplified deployment that runs within the application process rather than requiring external cluster orchestration. Native partitioning model alignment with Kafka topic partitions ensures natural scalability and load distribution, while incremental rebalancing provides minimal disruption during scaling operations [4].

2.3.3. Real-World Applications

Kafka Streams has demonstrated particular effectiveness in environments already heavily invested in Kafka infrastructure, where the simplified deployment model and native integration provide substantial operational advantages. Usage statistics indicate it has a smaller but growing market share compared to the other frameworks examined [2].

Scalability testing confirms linear scaling with Kafka partitions, with documented deployments successfully operating with many instances in production environments. The framework provides moderate support for event-time processing with good accuracy in temporal ordering tests, positioning it between Spark Streaming and Flink in this capability dimension [4].

3. Performance Comparison: Empirical Evaluation Results

Comprehensive performance evaluation conducted on standardized benchmarks reveals significant differences in processing characteristics across the examined frameworks [3]. These differences manifest across multiple dimensions including latency distribution, throughput capacity, resource utilization, and recovery behavior.

Latency measurements reveal substantial architectural differences, with Spark Streaming demonstrating higher average latency and percentile values, reflecting the inherent delay introduced by its micro-batch approach. Flink achieves substantially lower latency with excellent average and percentile measurements, highlighting its true streaming design. Kafka Streams occupies a middle ground with moderate latency values, balancing its lightweight architecture with processing overhead.

Table 2 Performance and Resource Characteristics [3]

Framework	Latency	Throughput	Resource Usage	Recovery Speed
Spark Streaming	Higher	Very high	High	Slow
Flink	Very low	High	Medium	Medium
Kafka Streams	Moderate	Moderate	Low	Fast

Throughput capacity similarly reflects architectural design priorities, with Spark Streaming demonstrating exceptional batch processing capabilities across multi-node clusters. Flink achieves impressive throughput while maintaining low latency per core, while Kafka Streams processes a moderate number of events per second per core, reflecting its optimization for integration rather than raw processing power.

Resource utilization patterns further illustrate architectural differences, with Spark Streaming consuming significant resources in terms of CPU utilization and memory per node during benchmark processing. Flink demonstrates more moderate utilization of CPU and memory per node, while Kafka Streams exhibits the most efficient resource pattern, reflecting its lightweight design philosophy.

Recovery capabilities after simulated failures show substantial variance, with Spark Streaming requiring longer average time to restore processing following system interruption. Flink achieves faster recovery on average, while Kafka Streams demonstrates the most rapid restoration, leveraging Kafka's native partition reassignment mechanisms. This recovery performance has significant implications for high-availability requirements in production environments [3].

Table 3 Feature and Capability Comparison [4]

Framework	Event Time Processing	State Management	Exactly-Once Semantics	Interactive Queries
Spark Streaming	Limited	Checkpointing	Yes	Via Spark SQL
Flink	Advanced	Multiple backends	Yes	Limited
Kafka Streams	Moderate	Local RocksDB	Yes	Native support

4. Architectural Decision Factors: Framework Selection Guidance

When selecting a stream processing technology, organizations must consider multiple factors including latency requirements, scalability patterns, existing infrastructure investments, and recovery characteristics. Each framework demonstrates particular strengths aligned with specific use case requirements and operational constraints.

Latency requirements often serve as a primary selection factor, with ultra-low latency needs strongly favoring Flink's true streaming architecture. Moderate latency requirements can be effectively addressed through Kafka Streams, while higher latency tolerance aligns well with Spark Streaming's micro-batch approach. These distinctions become particularly relevant in use cases such as algorithmic trading (requiring very fast responses), fraud detection (typically requiring quick decisions), and analytical reporting (often tolerating longer latency).

Scalability patterns represent another critical selection dimension, with each framework exhibiting different scaling characteristics. Spark Streaming has demonstrated near-linear scaling to many nodes in production environments, making it well-suited for vertical scaling with complex analytical operations. Flink achieves linear scaling to a substantial number of nodes before encountering network coordination bottlenecks, providing effective horizontal scaling with distributed state. Kafka Streams scales directly with Kafka partitions, having been successfully deployed across many instances in production, offering straightforward scalability for Kafka-aligned processing topologies.

Existing infrastructure investments significantly influence framework selection economics, with organizations heavily invested in the Spark ecosystem typically benefiting from leveraging Spark Streaming's unified programming model. Kafka-centric architectures often derive substantial operational advantages from Kafka Streams' simplified deployment model and native integration. Greenfield implementations without significant existing investments can prioritize functionality alignment, with Flink typically offering the greatest flexibility across diverse processing semantics.

Recovery characteristics impact availability guarantees and operational resilience, with Spark Streaming requiring longer times for typical recovery operations. Flink achieves faster restoration from the latest checkpoint state, while Kafka Streams typically recovers quickly through partition reassignment. These differences directly influence achievable uptime metrics and recovery time objectives in production deployments.

Table 4 Use Case Alignment [5]

Use Case	**Spark Streaming**	**Flink**	**Kafka Streams**
Complex analytics	Excellent	Good	Limited
Low-latency events	Limited	Excellent	Good
Edge computing	Poor	Moderate	Excellent
Kafka-centric apps	Moderate	Good	Excellent
Machine learning	Excellent	Good	Limited

4.1. Performance Characteristics and Processing Guarantees

Performance evaluation across multiple standardized benchmarks reveals that Kafka Streams achieves balanced performance characteristics that emphasize operational stability over absolute throughput maximization:

Throughput Capacity: Measurement across varying processing complexity shows Kafka Streams achieves substantial events-per-second throughput for stateless processing tasks, with throughput decreasing for more complex operations such as machine learning model scoring. This throughput scales linearly with instance count up to infrastructure limits, showing consistent per-core performance as deployments expand [7].

Latency Profile: Instrumented testing under standardized loads reveals a moderate latency distribution for typical processing operations. These values position Kafka Streams as providing acceptable latency characteristics, suitable for a wide range of operational use cases while avoiding the complexity associated with ultra-low-latency frameworks [6].

Processing Guarantees: Exactly-once processing semantics are achieved by leveraging Kafka's transactional capabilities, with moderate performance overhead compared to at-least-once semantics. This efficiency advantage derives from the tight integration with Kafka's native transaction protocol, which enables atomic writes across multiple topics with minimal coordination overhead [6].

State Management: The local state store approach employed by Kafka Streams provides excellent performance for state access, with measurements showing low latency for local state queries and acceptable performance for remote state access when using the Interactive Queries API. State management overhead contributes a small portion to total processing costs, compared to higher overhead for Flink, reflecting the efficiency of the local storage approach [8].

4.2. Fault Tolerance and Recovery Characteristics

The fault tolerance model in Kafka Streams differs fundamentally from cluster-based alternatives, leveraging Kafka's partition reassignment protocol rather than implementing custom recovery mechanisms. When an instance fails, its assigned partitions are automatically redistributed to remaining instances through the consumer group rebalancing protocol, with state restored from changelog topics.

This approach yields industry-leading recovery performance, with comparative measurements showing swift average recovery times for Kafka Streams, compared to longer recovery periods for Flink and substantially extended restoration for Spark Streaming under comparable failure scenarios. In production environments, recovery times typically remain brief depending on state size and partition count [5][7].

The reliability of this recovery mechanism derives from Kafka's mature consumer group protocol and the local nature of state storage, which eliminates the need for coordination during recovery. Each instance independently restores its assigned state from changelog topics, enabling parallel recovery across the deployment. This architecture results in recovery times that scale sub-linearly with deployment size, maintaining reasonable recovery durations even in large-scale deployments [7].

4.3. Deployment Scale and Production Experience

Deployment experience across various industries demonstrates that Kafka Streams scales effectively to substantial production workloads while maintaining its operational advantages. In financial services environments, documented deployments have successfully operated with many Kafka Streams instances processing real-time trading and transaction data. Scaling patterns show linear throughput increases up to infrastructure limits, with independent instances automatically balancing processing load through Kafka's partition assignment mechanism [6].

For IoT applications, empirical testing shows that Kafka Streams scales linearly to many instances before network communication becomes the primary bottleneck, with each instance efficiently processing its assigned subset of the overall dataflow. This scaling pattern makes Kafka Streams particularly well-suited for edge and near-edge processing scenarios where deployment simplicity represents a significant operational advantage [8].

Market adoption metrics indicate growing recognition of these advantages, with industry surveys across many organizations showing Kafka Streams adoption grew substantially year-over-year, reflecting strong validation of its architectural approach. This growth rate exceeds general stream processing adoption, indicating favorable outcomes from initial deployments and expanding use cases [6].

4.4. Development Experience and Productivity

The developer experience associated with Kafka Streams emphasizes simplicity and alignment with standard Java application development patterns. Code complexity metrics derived from standard processing patterns indicate that Kafka Streams implementations require fewer lines of code than equivalent functionality in cluster-based frameworks. This reduction results primarily from the elimination of cluster coordination code and the declarative nature of the Streams DSL [8].

Developer productivity surveys indicate faster time-to-production with Kafka Streams compared to alternatives, with particular advantages in deployment and operational phases of the development lifecycle. The embedded nature of the library eliminates the need for separate cluster management and monitoring infrastructure, reducing operational complexity for development teams [8].

The operational simplicity extends to configuration and tuning, with Kafka Streams requiring adjustment of far fewer primary configuration parameters compared to cluster-based alternatives like Flink. This reduced parameter space simplifies deployment and makes performance tuning more accessible to development teams without specialized expertise in distributed systems [8].

4.5. Comparative Framework Analysis

When evaluated against alternative stream processing frameworks, Kafka Streams demonstrates distinctive characteristics that derive directly from its architectural approach:

Kafka Streams vs. Flink: While Flink achieves superior raw performance with lower latency and higher maximum throughput, it requires dedicated cluster infrastructure and significantly more complex configuration. Operational complexity measurements indicate Flink requires tuning of substantially more parameters compared to the simpler requirements for Kafka Streams, representing significantly higher operational overhead. Deployment experience indicates Kafka Streams excels in scenarios prioritizing operational simplicity and direct integration with Kafka, while Flink provides advantages for use cases requiring absolute minimum latency or advanced event time processing capabilities [5].

Kafka Streams vs. Spark Streaming: Comparative analysis shows Kafka Streams requires substantially fewer resources than Spark Streaming, with memory utilization measurements indicating much lower average memory use per Kafka Streams instance compared to Spark Streaming. Recovery time comparisons are particularly striking, with Kafka Streams recovering much more quickly than Spark Streaming. These differences reflect the fundamental architectural distinction between Kafka Streams' lightweight embedded approach and Spark's batch-oriented processing model [6][7].

Resource Efficiency: Across all evaluated frameworks, Kafka Streams demonstrates superior resource efficiency, with significantly lower JVM garbage collection overhead compared to Flink and Spark. This efficiency advantage translates directly to infrastructure cost reduction, with organizations reporting substantial infrastructure cost savings when implementing equivalent processing logic in Kafka Streams compared to cluster-based alternatives [7][8].

4.6. State Management Approaches

State management represents a critical aspect of stream processing architectures, with significant implications for performance, recovery, and operational complexity:

Local State Store Model: Kafka Streams implements state through local RocksDB instances, with state changes backed by Kafka changelog topics for durability and recovery. This approach provides excellent performance for state access, with low latencies for local state queries, while enabling efficient recovery through changelog replay. The local nature of state storage eliminates coordination overhead during normal operation, contributing to the framework's overall efficiency [8].

Compression and Efficiency: Performance analysis reveals that RocksDB state stores achieve good compression ratios for typical machine learning feature data, substantially reducing storage requirements compared to in-memory alternatives. This compression capability enables Kafka Streams to efficiently handle larger state sizes than would otherwise be possible within reasonable memory constraints [7].

Interactive Queries: The Interactive Queries API represents a particularly valuable capability unique to Kafka Streams, enabling direct access to state stores for lookups without requiring additional database systems. This feature allows applications to expose their state for external querying, with measurements showing good latency characteristics for both local state access and remote state access across the network. This capability eliminates the need for separate query infrastructure in many use cases, substantially reducing architectural complexity [8].

4.7. Decision Framework for Technology Selection

The distinct characteristics of Kafka Streams make it particularly well-suited for specific deployment scenarios and use cases:

Kafka-Centric Architectures: Organizations with existing investments in Kafka derive substantial advantages from Kafka Streams' native integration, with infrastructure cost assessments showing significant reduction compared to deploying separate processing clusters. The elimination of data movement between systems results in both performance advantages and operational simplicity [8].

Operational Simplicity Priority: When operational simplicity represents a primary concern, Kafka Streams provides significant advantages through its embedded deployment model. The reduction in configuration parameters and elimination of cluster management substantially reduces operational overhead, particularly valuable for organizations with limited specialized operations resources [5][8].

Moderate Latency Requirements: Performance characteristics make Kafka Streams ideal for use cases with moderate latency requirements in the tens of milliseconds range. The measured latency profile aligns well with a wide range of operational use cases including real-time analytics, monitoring, and data transformation, while avoiding the complexity associated with ultra-low-latency architectures [6].

Edge and Distributed Processing: The lightweight resource profile and simplified deployment model make Kafka Streams particularly suitable for edge computing scenarios where deployment simplicity represents a significant advantage. Moderate memory consumption enables deployment on modest edge hardware, while the embedded nature eliminates the need for separate cluster infrastructure [7][8].

5. Conclusion

The comprehensive analysis of stream processing architectures reveals distinctive approaches that align with different organizational priorities and use cases. Spark Streaming excels in environments already invested in the Spark ecosystem, where unified batch and stream processing trumps absolute latency concerns. Its micro-batch processing model provides excellent scalability for analytical workloads but introduces inherent latency constraints that limit its suitability for time-critical applications. Apache Flink stands out for systems requiring true event-time processing with minimal latency, complex stateful operations, and precise event ordering. Its purpose-built streaming architecture delivers superior performance for demanding real-time applications while maintaining robust state management capabilities through incremental checkpointing and save points.

Kafka Streams provides the most streamlined solution for applications already leveraging Kafka, offering simplified deployment and natural integration with Kafka's partitioning model. Its client-library approach eliminates cluster management complexity while providing reasonable performance characteristics and industry-leading recovery times through Kafka's native partition reassignment mechanisms. As stream processing continues its evolution from specialized infrastructure toward mainstream application development patterns, the market increasingly recognizes that no single framework optimally addresses all requirements. This has led to multi-architecture implementations where organizations select the appropriate technology for each specific data pipeline based on latency requirements, existing infrastructure, and operational constraints. This hybrid approach enables organizations to leverage the specific strengths of each framework while creating comprehensive stream processing ecosystems tailored to diverse application requirements.

References

[1] Z. Milosevic, et al, "Chapter 2 - Real-Time Analytics," Big Data, Principles and Paradigms, 2016, Available: https://www.sciencedirect.com/science/article/abs/pii/B9780128053942000027

[2] Yugandhara R. Y, "Streaming Analytics Market Trends Forecast Report 2023," June 2023, Online, Available: https://www.researchgate.net/publication/371699754_Streaming_Analytics_Market_Trends_Forecast_Report_2023

[3] Vikash, et al, "Performance evaluation of real-time stream processing systems for Internet of Things applications," Future Generation Computer Systems, Volume 113, December 2020, Available: https://www.sciencedirect.com/science/article/abs/pii/S0167739X20302636

[4] Pradeep Bhosale, "Memory Management in Apache Flink: Techniques for Efficient State Handling," December 2023 IJIRMPS, Available: https://www.ijirmps.org/papers/2023/6/231999.pdf

[5] Redpanda, "Kafka Streams vs. Flink—How to choose," Blog, Available: https://www.redpanda.com/guides/event-stream-processing-kafka-streams-vs-flink

[6] Johannes Rank, Andreas Hein, Helmut Krcmar, "The Role of Performance in Streaming Analytics Projects: Expert Interviews on Current Challenges and Future Research Directions," 2023, Online, Available: https://dl.gi.de/items/0305297a-02cc-4b48-b2be-b6677cc1c9a6

[7] Elena Georgiou, "EVALUATING APACHE FLINK, KAFKA STREAMS, AND SPARK STREAMING FOR SCALABLE MACHINE LEARNING WORKFLOWS," Jan 2025, Online, Available: https://www.researchgate.net/publication/390521709_EVALUATING_APACHE_FLINK_KAFKA_STREAMS_AND_SPARK_STREAMING_FOR_SCALABLE_MACHINE_LEARNING_WORKFLOWS

[8] Hamid Nasiri, et al, "Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities," June 2019, Journal Of Big Data, Available: https://www.researchgate.net/publication/333706228_Evaluation_of_distributed_stream_processing_frameworks_for_IoT_applications_in_Smart_Cities