CLOUD COMPUTING IA 2 REPORT: A Comprehensive Performance Analysis of Stream Processing with Kafka in Cloud Native Deployments for IoT Use-Cases

CHINMAY ACHARYA 16010122002

MANAS BHAMBHRA 16010122017

PARTH BHANUSHALI 16010122019

## **1. Introduction**

The increasing number of Internet of Things (IoT) devices has led to an explosion of real-time data that demands efficient, scalable, and low-latency processing. Traditional data processing architectures often fall short in meeting these demands due to their monolithic and inflexible nature. As a result, the industry has shifted towards stream processing frameworks that support real-time analytics in cloud-native environments.

Cloud-native platforms, characterized by containerized microservices and orchestrated deployments, enable high availability, elasticity, and resilience. Apache Kafka, a distributed messaging system, has emerged as a leading solution for ingesting and distributing real-time data streams at scale. Combined with Kafka Streams—a native stream processing library—Kafka enables developers to perform real-time transformations, aggregations, and enrichments directly within the data pipeline.

This report presents a comprehensive performance analysis of Kafka and Kafka Streams in Kubernetes-managed cloud-native deployments. The aim is to measure key performance metrics such as latency, throughput, CPU usage, and memory footprint under different processing scenarios relevant to IoT workloads. Through this study, we aim to provide insights into optimizing the deployment and resource allocation of Kafka-based stream processing systems for real-world applications.

## **2. Research Objectives**

The primary aim of this study is to evaluate the performance characteristics of Apache Kafka and Kafka Streams in handling real-time IoT data within cloud-native environments. Our first objective is to examine how these technologies respond to varying workloads by measuring key indicators such as throughput, latency, CPU, and memory utilization during different types of stream processing tasks.

We then explore the effects of deploying Kafka and Kafka Streams within Kubernetes clusters, assessing their behavior under both stateless and stateful operations. This includes evaluating the impact of encryption (TLS) and different hardware configurations to understand performance bottlenecks and scalability.

The final objective involves implementing a complete streaming pipeline using Kafka and Kafka Streams to simulate real-world IoT use-cases. This implementation is used to measure system behavior under controlled conditions and validate the feasibility of using these tools for scalable, low-latency data processing in elastic, containerized cloud-native infrastructures.

## **3. Stream Processing in IoT Applications**

The Internet of Things (IoT) ecosystem generates vast volumes of continuous data from interconnected sensors and devices. Real-time analytics on this data is essential for enabling responsive, intelligent systems in domains such as smart cities, healthcare, agriculture, and industrial automation. Stream processing platforms allow for immediate insight extraction and reactive decision-making.

Apache Kafka has emerged as a robust solution for delivering and ingesting IoT data streams. It supports distributed, fault-tolerant, and horizontally scalable data pipelines. Kafka's architecture enables high-throughput and low-latency handling of real-time data, making it ideal for IoT scenarios where timely actions are critical.

Kafka Streams extends Kafka's capabilities by enabling in-place stream processing directly on Kafka topics. It uses a processing topology model comprising source, processor, and sink nodes. This model allows users to implement both stateless transformations (e.g., filter, map) and stateful operations (e.g., windowed aggregations, joins).

Key features that support IoT analytics include:

* Stream-Table Duality: Kafka Streams treats a stream as a changelog and supports dynamic table views of data.
* Scalability: Tasks are distributed across threads and instances, with partition-based parallelism.
* Fault Tolerance: Internal state is backed up using Kafka topics, enabling recovery from failures.
* Lightweight Deployment: Kafka Streams apps run as simple Java applications, containerized via Docker and orchestrated through Kubernetes.

By integrating with cloud-native tools like Strimzi, Prometheus, and Grafana, Kafka enables real-time observability and auto-scaling. These properties make it a powerful engine for stream processing in modern IoT infrastructures.

## **4. Kafka Architecture and Cloud-Native Deployment**

Apache Kafka is a distributed, fault-tolerant, high-throughput messaging system designed for real-time data stream processing. It operates with a cluster of Kafka brokers, coordinated via ZooKeeper. Kafka topics are partitioned and replicated across brokers to ensure scalability and fault tolerance. Data producers write events to Kafka topics, which are then consumed by downstream applications such as Kafka Streams.

Kafka Streams is a Java-based stream processing library that allows applications to read, process, and write data back into Kafka topics. It operates through a processor topology, which defines a directed graph of nodes (processing tasks) that ingest, transform, and forward data.

Kafka Streams supports stateless operations (e.g., filtering, mapping) and stateful operations (e.g., aggregations, joins). These operations are executed as parallel tasks, each mapped to specific partitions. Tasks run in separate threads or instances, enabling high concurrency without shared state, improving scalability and fault isolation.

In a cloud-native deployment, Kafka is containerized using Docker and orchestrated through Kubernetes. The Strimzi operator automates Kafka cluster provisioning, including broker configuration, TLS encryption, and ZooKeeper coordination. It also manages topic creation and facilitates seamless upgrades and recovery.

To monitor Kafka and Kafka Streams, the environment integrates Prometheus for metrics collection and Grafana for visualization. Additionally, Netdata is used to track pod-level CPU and memory consumption. This observability stack enables real-time performance tracking, crucial for adaptive scaling and troubleshooting.

Kafka’s compatibility with Kubernetes allows it to leverage cloud-native benefits like horizontal scaling, self-healing, rolling updates, and isolated networking. This makes it highly suitable for modern IoT and event-driven architectures where data streams are continuous, large in volume, and require near real-time processing.

## **5. Measurement Framework**

To evaluate the performance of Kafka and Kafka Streams in a cloud-native environment, we deployed a complete stream processing pipeline within a Kubernetes cluster using containerized components. The architecture consists of four core elements: a Kafka Producer, a Kafka Cluster, a Kafka Streams Processor, and an optional Kafka Consumer for result validation. All components were packaged as Docker containers and orchestrated using Kubernetes, with the Strimzi operator managing Kafka deployments.

The Kafka Producer is a Java-based application configured to randomly select from 8,760 real-world JSON event files and publish them to a Kafka input topic (Tin). It supports adjustable throughput rates, batching, TLS configuration, and failure detection for unachievable output rates.

The Kafka Streams Processor consumes events from Tin, processes them using stateless or stateful transformations, and writes the results to an output topic (Tout). This component is also Java-based and containerized, with support for multi-threaded processing and configurable commit intervals.

The Kafka Cluster includes ZooKeeper nodes and multiple Kafka brokers, each running in isolated pods. Kafka topics are configured with multiple partitions and a replication factor of 3 to ensure reliability and balanced load distribution. All internal communications were secured using TLS certificates generated by Strimzi.

For observability, Prometheus was used to scrape metrics exposed by both Kafka (via the Strimzi Kafka Exporter) and the Kafka Streams Processor (via JMX Exporter). Grafana was used to visualize performance indicators, while Netdata monitored CPU and memory usage at the pod level.

To assess maximum throughput, the producer’s event rate was incrementally increased until system limits were reached. Measurements were collected for:

* CPU and memory usage of Kafka and Kafka Streams
* Processing latency (time from message production to completion)
* Throughput (events/sec successfully processed)

All experiments were conducted in a Kubernetes cluster provisioned through the ELKH cloud, with performance validated across VM and bare-metal environments.

A diagram of a software system

AI-generated content may be incorrect.

## **6. Stream Processing Scenarios**

In order to evaluate the performance of Kafka and Kafka Streams under varying workloads, three distinct categories of stream processing tasks were implemented: two stateless operations and one stateful aggregation operation. These scenarios simulate common data transformation patterns in IoT analytics pipelines.

A. Filtering (Stateless Operation)

This scenario focuses on filtering incoming IoT event streams based on specific ID field criteria. The Kafka Streams processor applies a filter operation that retains a configurable subset of messages: 0.05%, 1%, and 10% of the input stream.

* Implementation: Kafka Streams filter() method was applied to match target IDs.
* Output: Filtered events were serialized and written back to an output topic.
* Resource Behavior: Minimal memory usage with moderate CPU load. Throughput decreased as more data was retained.
* Latency: Remained under 0.4 seconds in all configurations.

B. Anonymization (Stateless Operation)

In this task, specific fields in the JSON messages were anonymized to protect sensitive information. Values in certain fields were replaced with masked characters (e.g., "X"), affecting less than 1% of total fields per event.

* Implementation: Kafka Streams map() method was used to transform incoming records.
* Output: Entire events, including modified fields, were sent to the output topic.
* Resource Behavior: CPU load was moderate; however, memory usage was higher due to full message reserialization.
* Throughput: Capped around 4.5k events/sec.
* Latency: Also maintained below 0.4 seconds.

C. Aggregation (Stateful Operation)

This scenario evaluates performance under stateful processing using Kafka Streams windowed aggregations. A 1-second tumbling window was used for all subcases.

Subcases:

1. Basic Block Aggregation:
   * Counted occurrences of 11 unique keys in the event stream.
   * Output: A single key-value pair containing 11 fields.
2. Grouped Block Aggregation:
   * Counted events grouped by specific field values (e.g., event codes).
   * Output: Multiple key-value pairs mapping event codes to their frequency.
3. Averaging Block Aggregation:
   * Calculated average duration between start and end timestamps in all events.
   * Output: A single float value representing the average time difference.

* Implementation: Kafka Streams groupByKey(), windowedBy(), and aggregate() APIs were used.
* Resource Behavior: High memory consumption, especially in the basic aggregation subcase. CPU usage was relatively uniform across subcases.
* Throughput: Significantly lower compared to stateless cases, peaking at ~1k events/sec.
* Latency: Averaged around 0.6 seconds.

## **7. Performance Results and Observations**

**A. Baseline (Kafka Only)**

To understand Kafka’s raw performance, an initial test was run without the Kafka Streams Processor. With 12 Kafka brokers configured, the system achieved a peak throughput of 32,000 events/second.

* Optimal resource allocation per broker: 2 vCPUs and 4GB RAM.
* These settings were used in all subsequent tests to maintain consistency.
* Varying the Kafka retention time (15 min vs. 1 hr) and measurement durations (30 min vs. 1 hr) showed no observable impact on performance.

Kafka brokers, due to their JVM-based architecture, utilized up to 90% of allocated memory under load, and retained this usage pattern even after traffic reduced. Hence, Kafka memory metrics were considered constant across configurations.

**B. Filtering Scenario (Stateless)**

Filtering operations (0.05%, 1%, 10% data retained) revealed:

* No major difference in max throughput between 0.05% and 1% filtering.
* 10% filtering led to ~5k events/sec drop in throughput.
* CPU usage (Kafka & Streams) scaled proportionally with retained event volume.
* Memory usage was consistent across all subcases.
* Latency remained < 0.4 seconds in all cases.

A graph of different types of data

AI-generated content may be incorrect.

**C. Anonymization Scenario (Stateless)**

Anonymization applied field-level replacements:

* Max throughput reduced to ~4.5k events/sec, indicating heavier processing.
* Kafka Streams CPU usage capped at ~2.75 vCPUs, lower than filtering.
* Kafka became the bottleneck due to increased output writes to Tout.
* Memory usage was significantly higher due to processing complexity.
* Latency stayed under 0.4 seconds.

A graph of different types of threads

AI-generated content may be incorrect.

**D. Aggregation Scenario (Stateful)**

Three subcases: basic key count, grouped code count, and average time diff:

* All aggregation tasks showed lowest performance, peaking just above 1k events/sec.
* Kafka Streams memory usage was highest in basic aggregation.
* Grouped and averaging subcases showed reduced memory consumption.
* Kafka CPU scaling plateaued despite higher thread/broker counts.
* Max latency observed: ~0.6 seconds.

A graph of different types of data

AI-generated content may be incorrect.

**E. TLS Encryption Impact**

TLS was enabled between Producer–Kafka and Kafka–Processor:

* No observable drop in throughput or latency.
* Kafka CPU usage increased by 6–9%, Streams by 7–10%.
* Memory usage remained stable.
* Filter operation CPU impact summarized:

A table with numbers and words

AI-generated content may be incorrect.

**F. Deployment Environment Impact**

Tests were run on:

1. ELKH Cloud (OpenStack VMs)
2. Bare-metal server with 40 cores & 188GB RAM
3. Bare-metal outperformed VM cluster in both network throughput and resource utilization.
4. Kubernetes networking on VMs allowed only ~30% of the throughput compared to inter-VM transfers, limiting Kafka ingestion rates.
5. Performance differences mirrored CPU benchmark ratios, confirming that hardware and deployment architecture play a critical role.

## 

## **8. Conclusion**

This study presented a detailed performance analysis of Apache Kafka and Kafka Streams in a Kubernetes-managed cloud-native environment, specifically targeting Internet of Things (IoT) data processing scenarios. By evaluating key performance metrics such as throughput, CPU and memory usage, and processing latency, we demonstrated how Kafka-based stream processing systems behave under varying workloads and configurations.

Our experimental results indicate that both Kafka and Kafka Streams offer low-latency and scalable solutions for real-time data pipelines. Stateless operations such as filtering and anonymization showed better performance and efficient resource utilization, while stateful operations like aggregation incurred higher memory and CPU consumption, with relatively lower throughput. TLS encryption introduced minimal overhead, increasing CPU usage slightly but having no significant impact on latency or data rate.

Moreover, our study found that deployment environment plays a significant role in system performance. Virtualized Kubernetes clusters, especially when provisioned through cloud infrastructure like OpenStack, may introduce limitations in terms of network throughput and resource scheduling. These environmental factors must be considered when designing and deploying scalable, high-performance IoT data pipelines.

In summary, Kafka and Kafka Streams are well-suited for modern, scalable, cloud-native IoT applications. Their modularity, resilience, and integration with monitoring tools make them effective for production-grade deployments. Future enhancements could include multi-topic stream joins, advanced orchestration strategies using Kubernetes, and container-level security hardening for enterprise readiness.

REFERENCES

[1] M. H´ eder et al., “The past, present and future of the ELKH cloud,”

Inform´ aci´ os T´ arsadalom, vol. 22, p. 128, aug 2022.

[2] T. J. Saleem and M. A. Chishti, “Data analytics in the internet of things:

a survey,” SCPE, vol. 20, no. 4, pp. 607–630, 2019.

[3] E. Siow, T. Tiropanis, and W. Hall, “Analytics for the internet of things:

A survey,” ACM computing surveys (CSUR), vol. 51, no. 4, 2018.

[4] J. Kreps, N. Narkhede, J. Rao, et al., “Kafka: A distributed messaging

system for log processing,” in Proceedings of the NetDB, vol. 11, 2011.

[5] H. Isah et al., “A survey of distributed data stream processing frame-

works,” IEEE Access, vol. 7, pp. 154300–154316, 2019.

[6] The Kubernetes Authors, “Kubernetes: Production-grade container or-

chestration.” Available: https://kubernetes.io/, 2022. [Online] Accessed:

2022-12-09.

[7] S. Vyas, R. K. Tyagi, C. Jain, and S. Sahu, “Literature review: A

comparative study of real time streaming technologies and apache

kafka,” in 4th CCICT, pp. 146–153, 2021.

[8] S. Henning and W. Hasselbring, “Theodolite: Scalability benchmarking

of distributed stream processing engines in microservice architectures,”

Big Data Research, vol. 25, p. 100209, 2021.

[9] J. Karimov et al., “Benchmarking distributed stream data processing

systems,” in 34th IEEE ICDE, pp. 1507–1518, 2018.

[10] Sanket Chintapalli et al, “Benchmarking streaming computation engines:

Storm, flink and spark streaming,” in IPDPSW, pp. 1789–1792, 2016.

[11] G. van Dongen and D. Van den Poel, “Evaluation of stream processing

frameworks,” IEEE TPDS, vol. 31, no. 8, pp. 1845–1858, 2020.

[12] H. Nasiri, S. Nasehi, and M. Goudarzi, “Evaluation of distributed stream

processing frameworks for iot applications in smart cities,” Journal of

Big Data, vol. 6, no. 1, pp. 1–24, 2019.

[13] E. Shahverdi, A. Awad, and S. Sakr, “Big stream processing systems:

An experimental evaluation,” in 35th IEEE ICDEW, pp. 53–60, 2019.

[14] H. Rajab and T. Cinkelr, “Iot based smart cities,” in ISNCC 2018, 2018.

[15] A. Charulatha and R. Sujatha, “Smart healthcare use cases and appli-

cations,” in Internet of Things Use Cases for the Healthcare Industry,

pp. 185–203, Springer, 2020.

[16] A. F. Ahmad et al., “A review on iot with big data analytics,” in ICoICT,

pp. 160–164, IEEE, 2021.

[17] H. Hejazi, H. Rajab, T. Cinkler, and L. Lengyel, “Survey of platforms

for massive iot,” in Future IoT, pp. 1–8, 2018.

[18] J. Perez, “Using Apache Kafka for Stream Process-

ing: Common Use Cases | OpenLogic by Perforce.”

https://www.openlogic.com/blog/kafka-stream-processing. [Online]

Accessed: 2023-01-14.

[19] A. Chawla et al., “Intelligent monitoring of iot devices using neural

networks,” in 24th ICIN, pp. 137–139, IEEE, 2021.

[20] K. Ogawa et al., “Iot device virtualization for efficient resource utiliza-

tion in smart city iot platform,” in PerCom Workshops, IEEE, 2019.

[21] G. M. D’silva, A. Khan, S. Bari, et al., “Real-time processing of iot

events with historic data using apache kafka and apache spark with

dashing framework,” in 2nd RTEICT, pp. 1804–1809, IEEE, 2017.

[22] P. Hunt et al., “ZooKeeper: Wait-free Coordination for Internet-scale

Systems,” in USENIX ATC 10, 2010.

[23] Apache Software Foundation, “Apache kafka documentation.”

https://kafka.apache.org/documentation, 2022. [Online] Accessed:

2022-12-09.

[24] D. Merkel, “Docker: lightweight linux containers for consistent devel-

opment and deployment,” Linux journal, vol. 2014, no. 239, p. 2, 2014.

[25] Strimzi Authors, “Strimzi overview guide.”

https://strimzi.io/docs/operators/0.30.0/overview.html, 2023. [Online]

Accessed: 2023-01-13.

[26] B. Rabenstein and J. Volz, “Prometheus: A next-generation monitoring

system (talk),” (Dublin), USENIX Association, May 2015.

[27] Grafana Labs, “Grafana: The open observability platform.”

https://grafana.com/, 2023. [Online] Accessed: 2023-01-13.

[28] Netdata Inc., “Netdata: Monitoring and troubleshooting transformed.”

https://www.netdata.cloud/, 2022. [Online] Accessed: 2023-01-13.

[29] Various authors, “JMX Exporter.” https://github.com/prometheus/jmx ex

porter, 2022. [Online] Accessed: 2023-01-13.