Optimizing Kafka-Based Supply Chain Architectures:

by Simran Ahuja

Submission date: 22-Apr-2025 11:29AM (UTC+0530)
Submission ID: 2653236763
File name: Optimizing_Kafka.pdf (661.89K)
Word count: 6114
Character count: 35708

Optimizing Kafka-Based Supply Chain Architectures: A Comprehensive Performance Analysis

Nupur Giri
HOD, Computer Engineering
V.E.S.I.T.
nupur.giri@ves.ac.in

Sejal Datir
Computer Engineering
V.E.S.I.T.
2021.sejal.datir@ves.ac.in

Simran Ahuja
Computer Engineering
V.E.S.I.T.
2021.simran.ahuja@ves.ac.in

Sania Khan
Computer Engineering
V.E.S.I.T.
2021.sania.khan@ves.ac.in

Jesica Bijju
Computer Engineering
V.E.S.I.T.
2021.jesica.bijju@ves.ac.in

***Abstract*— This paper examines the impact of Apache Kafka parameter tuning on the performance and scalability of a microservices-based supply chain management system. The study evaluates the influence of key parameters including batch size, compression type, partition count, and replication factor—on critical performance metrics such as throughput, latency, and fault tolerance. Experimental results demonstrate that fine-tuning Kafka configurations significantly reduces processing delays, improves throughput, and enhances system resilience under varying workloads. Notably, the trade-off between replication overhead and performance efficiency becomes evident at high message volumes, where lower replication factors yield better processing efficiency. The findings provide actionable insights and practical guidelines for optimizing Kafka performance in large-scale, event-driven supply chain systems.**

I. Introduction

Supply Chain Management (SCM) is a critical component of modern business operations, ensuring the seamless flow of goods from manufacturers to consumers. Traditionally, SCM systems relied on monolithic architectures, where all core functionalities such as order processing, inventory management, logistics, and tracking were tightly coupled into a single application. While effective in earlier systems, this architecture is increasingly inefficient in handling the complexities and scale of modern supply chains. Monolithic SCM systems suffer from performance bottlenecks when processing large transaction volumes, often resulting in degraded efficiency [1]. They are also prone to system-wide failures, where a single fault can disrupt the entire supply chain workflow [2]. Furthermore, they lack scalability, as the entire application must be scaled even if only one module experiences high demand, leading to inefficient resource utilization [3]. Additionally, they are inflexible to business changes, making it difficult to introduce new features or integrate third-party services without significant rework and redeployment.

To overcome these limitations, modern SCM systems are shifting towards microservices architecture, deployed using Docker containers. The approach presented in this paper consists of five independent, loosely coupled services - Manufacturer, Hospital, Transport, Monitor, and Notification. These services operate independently but communicate through Apache Kafka, a real-time distributed event streaming platform that facilitates fast, reliable, and scalable message exchange between microservices [4]. Docker containerization ensures consistent deployment environments across different infrastructure setups, simplifies service orchestration, and enhances portability. By running each service in an isolated container, the architecture achieves improved fault isolation, independent scaling, and simplified maintenance[5]. While Kafka significantly enhances microservices efficiency, its performance heavily depends on parameter optimization. Improper configurations can lead to processing delays, inefficient resource usage, and lower fault tolerance. This paper investigates the impact of tuning key Kafka parameters, including partition count, batch size, and replication factor, on throughput, latency, and overall system performance. Partitioning enhances scalability by enabling parallel processing across multiple brokers. Replication improves fault tolerance by maintaining redundant copies of messages across brokers, ensuring data durability in case of node failures. Batch size optimizations improve throughput by reducing the overhead of individual message transmission [6].

By fine-tuning these Kafka parameters and deploying services using Docker, the proposed system achieves faster event processing, improved reliability, and better scalability. The results demonstrate how parameter optimization and containerized deployment significantly enhance the resilience and performance of microservices-based supply chain systems [7].

II. Literature Survey

Several studies have explored Apache Kafka's role in real-time messaging, data processing, and system optimization. In [8], Kafka's application in microservices architecture is examined, highlighting its asynchronous messaging capabilities and advantages over traditional RESTful APIs in terms of scalability and fault tolerance. Similarly, [9] investigates

Kafka's integration into inventory management systems, demonstrating how real-time data streaming enhances inventory tracking, demand forecasting, and overall supply chain efficiency. Expanding on Kafka's role in microservices, [10] evaluates the scalability of five modern stream processing frameworks on Kubernetes clusters. The benchmarking reveals that all frameworks exhibit near-linear scalability with sufficient resources but differ in efficiency, concluding that no single framework is universally superior, as performance varies by use case.

Focusing on Kafka in supply chains, [11] proposes an integrated Kafka-Akka architecture for concurrent and real-time processing in supply chain management. The study demonstrates how this architecture enhances inventory optimization, order fulfillment, and operational agility. Meanwhile, [12] covers best practices for building and optimizing scalable Kafka clusters, including architecture design, data partitioning, replication strategies, and configuration tuning. The authors emphasize that efficient cluster architecture and fault tolerance mechanisms are essential for achieving high availability, scalability, and low-latency data streaming at large scales.

Partitioning strategies for performance optimization are analyzed in [13]. The study explains how Kafka partitions enable parallelism and higher throughput but introduces trade-offs. It provides guidelines for determining the optimal number of partitions based on throughput requirements, considering factors such as producer and consumer performance, file handle limits, and memory usage. In a comparative study, [14] evaluates Kafka against other messaging frameworks, such as RabbitMQ and ActiveMQ, concluding that Kafka offers superior performance for high-throughput, low-latency applications. Further refining Kafka's scalability, [15] explores heuristic algorithms for optimizing topic partitioning, improving load balancing, fault tolerance, and replication efficiency.

Kafka's real-world applications across industries are presented in [16], where the authors highlight its adaptability in finance, healthcare, IoT, and fraud detection. The study showcases Kafka's role in predictive analytics and event-driven architectures. Similarly, [17] evaluates Kafka as a distributed messaging system based on the publish-subscribe model, emphasizing its high throughput, fault tolerance, and widespread adoption by Fortune 500 companies. Through multiple test scenarios, the system's efficiency is assessed based on QoS parameters.

Performance benchmarking studies further emphasize the importance of configuration tuning. [18] presents a performance analysis of message brokers, including Kafka, under various data ingestion scenarios. The benchmarking demonstrates Kafka's peak ingestion rate of approximately 420,000 messages per second, reinforcing the need for thorough testing and tuning of Kafka configurations for optimal performance. Similarly, [19] addresses Kafka's latency-sensitive processing, identifying high CPU consumption in KafkaProducer as a bottleneck. The study proposes buffering techniques and API modifications, reducing KafkaProducer's CPU load by 75% while maintaining low latency. Additionally, [20] offers a comprehensive performance tuning guide, covering partitioning strategies, broker configurations, garbage collection, and ZooKeeper optimization, ensuring high-throughput and low-latency Kafka deployments.

III. System Performance and Design Considerations

Optimizing Kafka for a microservices-based supply chain management system requires careful consideration of scalability, fault tolerance, and latency. While Kafka provides an event-driven architecture, its default configurations may not always be optimal for large-scale, real-time supply chains. Factors such as inefficient partitioning, improper replication strategies, and inadequate broker scaling can lead to bottlenecks that impact message throughput and system reliability. To address these challenges, this study explores different Kafka configurations to analyze their effects on system performance.

A key design consideration is partitioning strategy, which directly influences message distribution and processing efficiency. Default round-robin partitioning spreads messages evenly but does not account for data locality, potentially increasing retrieval times. In contrast, key-based partitioning groups related events together, improving access efficiency at the cost of potential load imbalance. Similarly, replication enhances fault tolerance by maintaining redundant copies of data, but higher replication factors introduce additional overhead, affecting overall performance. The trade-off between these factors necessitates a thorough evaluation to determine the most effective configuration for supply chain applications.

This section establishes the foundation for the methodology by outlining the rationale behind the Kafka parameter optimizations explored in this study. The following section presents an empirical evaluation of these configurations, detailing their impact on system performance through experimental analysis and benchmarking.

IV. Methodology

A. System Architecture and Experimental Setup

The Kafka-based drug supply chain system follows a microservices architecture, as illustrated in Fig. 1. It consists of five loosely coupled services - Hospital, Manufacturer, Transport, Monitor, and Notification - communicating through Kafka topics for real-time event streaming. This architecture enables independent scaling, fault isolation, and efficient parallel processing, making the system highly resilient and adaptable to fluctuating workloads. To evaluate the system's performance, a Dockerized Kafka cluster was deployed as the core messaging backbone. The cluster was configured with varying broker counts across three configurations to simulate different levels of scalability and fault tolerance. A single Zookeeper instance was used to manage broker metadata, track broker health, and coordinate leader election. All components were containerized using Docker, running

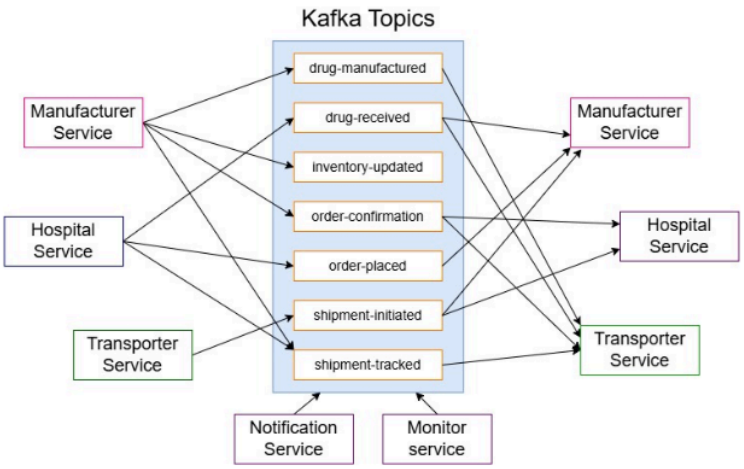

Fig. 1. Kafka architecture diagram

within an isolated bridge network to enable efficient and secure inter-service communication.

The experiments were conducted on a host machine equipped with a MacBook Pro powered by an M1 chip, 16 GB of RAM, and 256 GB of storage. This setup provided sufficient processing power and memory capacity to simulate large-scale workloads without hardware-induced performance bottlenecks.

B. Kafka-Based Microservices Architecture and Communication Flow

The system comprises five microservices, each handling distinct operations within the drug supply chain, as illustrated in Fig. 1. This event-driven architecture enables seamless communication through Kafka topics, ensuring efficient order processing, inventory management, and delivery tracking.

The Hospital Service initiates the workflow by publishing messages to the `order-placed` topic when drug orders are created. It also consumes messages from the `order-confirmation` topic to track the status of placed orders, ensuring hospitals receive real-time updates. The Manufacturer Service consumes messages from the `order-placed` topic to initiate drug production. After verifying inventory availability, it produces updates to the `drug-manufactured`, `inventory-updated`, and `order-confirmation` topics to reflect production progress, ensuring real-time manufacturing and inventory tracking.

The Transport Service handles delivery operations by consuming messages from the `drug-manufactured` topic and producing updates to the `shipment-status` topic, reflecting the shipment's progress. This ensures real-time order tracking and improves transparency. The Monitor Service tracks system performance and Kafka metrics, such as throughput, latency, and broker health. It publishes anomaly alerts to the monitor-alerts topic when irregularities are detected, enhancing observability and fault detection. The Notification Service consumes messages from multiple topics, such as `order-confirmation` and `shipment-status`, broadcasting real-time updates to connected clients to improve supply chain visibility.

Kafka topics serve as communication channels between the microservices, enabling asynchronous, event-driven processing. The `order-placed` topic triggers drug production, while the drug-manufactured topic initiates the delivery workflow. The `shipment-status` topic ensures `real-time` shipment tracking, and the `order-confirmation` topic keeps hospitals informed of order status. The `monitor-alerts` topic enhances system reliability by broadcasting anomalies. This structured topic-based communication ensures reliable event propagation and consistent data flow across the microservices.

C. Test Workload

To simulate real-world usage patterns, the system was subjected to varying message loads, categorized into three distinct levels: light, medium, and heavy. These load levels were designed to assess Kafka's performance under different traffic conditions, evaluating metrics such as throughput, latency, and fault tolerance.

A Python-based simulation script was used to generate the workload, dynamically producing Kafka messages to the `order-placed` topic. Each message represented a drug order, encapsulated in a JSON payload containing details such as drug names, quantities, and timestamps. The timestamps were appended at the time of message production, serving as the basis for calculating end-to-end latency as the message traversed through the microservices. As the system workflow progressed, the subscriber services consumed these messages, triggering subsequent events, including inventory updates, shipment initiation, and notifications.

The workload was classified into the following categories:

- **Light Load**: This level includes message volumes ranging from 100 to 5,000 messages, simulating low-demand scenarios with minimal strain on the system.
- **Medium Load**: This level encompasses message volumes between 7,500 and 20,000 messages, representing stable and consistent order traffic.
- **Heavy Load**: This level simulated peak demand conditions with message volumes ranging from 25,000 to 50,000 messages. It evaluates the system's performance under high-stress scenarios, testing Kafka's capacity to handle large-scale message loads efficiently.

D. Parameter Configuration Strategy

To evaluate Kafka's performance under varying conditions, three distinct configurations were designed, differing in broker count, replication factor, number of partitions, and partitioning strategy. The selection of these configurations aims to capture the impact of scaling Kafka's architecture on message distribution, fault tolerance, and processing efficiency.

- **Configuration 1 (C_1)** represents a minimal setup with a single broker, a replication factor of one, and a single partition. This configuration was chosen to establish a performance baseline, reflecting how Kafka behaves in a basic, non-redundant environment. It highlights the raw throughput and latency characteristics without the overhead of replication or multi-broker coordination.
- **Configuration 2 (C_2)** introduces an additional broker, increases the replication factor to two, and expands the number of partitions to three. This setup was selected to observe the effect of moderate scaling, specifically how Kafka handles fault tolerance and parallel processing. The replication factor improves reliability, while multiple partitions enable better message distribution and concurrency.
- **Configuration 3** (C_3) employs three brokers, a replication factor of three, and six partitions, using key-based partitioning. This configuration was designed to simulate a larger-scale, production-like environment. The higher replication factor ensures message availability during broker failures, while key-based partitioning—using `order-id` as the key—optimizes data locality and preserves message ordering, which is critical for consistent supply chain operations.

Parameter	C_1	C_2	C_3
Brokers	1	2	3
Replication Factor	1	2	3
Partitions	1	3	6
Partitioning Strategy	Round-Robin	Round-Robin	Key-Based

TABLE I
KAFKA CLUSTER CONFIGURATIONS

Table I summarizes the three Kafka configurations used in the experiment, highlighting the key parameters. By comparing these configurations, the experiment aims to analyze how scaling Kafka's architecture impacts key performance metrics, providing insights into its efficiency under varying workloads.

E. Kafka Parameters

The Kafka cluster's performance was evaluated based on four key parameters, each influencing the system's efficiency, scalability, and fault tolerance.

1) **Brokers:** Kafka brokers are responsible for storing and managing message data, facilitating communication between producers and consumers. Deploying multiple brokers enhances fault tolerance, prevents bottlenecks, and ensures high availability, which is critical in large-scale supply chain systems.
2) **Replication Factor:** This parameter determines how many copies of a message exist across brokers, preventing data loss in case of failures. A higher replication factor improves resilience but increases storage overhead and synchronization complexity. The optimal configuration of three ensures redundancy and minimizes downtime.
3) **Partitions:** Partitions divide Kafka topics into smaller segments, enabling parallelism. More partitions allow multiple consumers to process messages concurrently, enhancing throughput. However, excessive partitions can lead to synchronization overhead. The experiment balances partition count to optimize performance.
4) **Partitioning Strategy:** This defines how messages are distributed across partitions. Two strategies were used:
   - **Round-Robin (RR):** Distributes messages evenly across partitions, optimizing load balancing but sacrificing message order consistency.
   - **Key-Based (KB):** Ensures messages with the same key are routed to the same partition, preserving ordering and improving data locality, which is beneficial for event-driven workflows.

F. Performance Metrics

The system's performance was evaluated using four key metrics, measuring Kafka's efficiency under different configurations and workloads.

1) **Average Latency (ms):** Latency measures the time taken for a message to be produced, transmitted, and consumed by Kafka clients. It reflects the overall responsiveness of the system. In supply chain applications, low latency ensures faster event stream processing (e.g., order placements, inventory updates), enabling real-time decision-making. Higher latency indicates delays in data availability, which can impact operational efficiency. Monitoring latency helps optimize Kafka configurations to maintain efficient message processing under varying loads.
2) **Kafka Request Time (ms):** Kafka request time represents the end-to-end processing time of a Kafka request, including network overhead, broker processing, and replication delays. It offers insights into the efficiency of Kafka's internal operations, such as message routing and acknowledgment. Lower request times

indicate faster broker responsiveness, while higher values reveal potential bottlenecks. Reducing Kafka request time is essential for maintaining consistent performance in distributed systems under fluctuating workloads.

3) **Average Jitter (ms):** Jitter quantifies the variation in message delivery time between subsequent messages, caused by network congestion, broker load, or processing delays. In event-driven systems like supply chains, high jitter can lead to out-of-order message delivery, impacting data consistency. Lower jitter ensures stable and predictable message delivery, which is critical for maintaining data integrity. Monitoring jitter helps fine-tune Kafka configurations, reducing fluctuations, especially under high-load conditions.
4) **Throughput (msgs/sec):** Throughput measures the rate at which Kafka processes messages, represented in messages per second (msgs/sec). Higher throughput indicates better system efficiency and the ability to handle larger message volumes. In supply chain systems, maintaining high throughput ensures continuous event stream processing, even during load spikes. Throughput is directly influenced by broker count, partitioning, and replication factor, making it a vital metric for assessing Kafka's scalability and overall performance.

V. Inferences

A detailed analysis of the experimental results is obtained from evaluating Kafka's performance under varying message loads and architectural configurations and presented in Table II.

The inferences are drawn based on key performance metrics such as jitter, Kafka request time, latency, and throughput. Each metric is analyzed across three distinct Kafka setups to assess their scalability and efficiency. The following subsections provide graphical representations and corresponding observations based on the values shown in Table II, highlighting how Kafka's behavior evolves with increasing message volume. These insights offer a comprehensive understanding of Kafka's performance characteristics and help identify potential bottlenecks under different load conditions.

A. Latency

In this study, we evaluated the average latency under three configurations across three load scenarios: Low, Medium and High Load.

1) **Low Load**
Fig. 2 shows the latency trend under low message volumes reveals that all configurations handle the workload efficiently, but C_3 consistently outperforms the others, exhibiting the lowest latency. This is attributed to its higher broker count, increased partitioning, and key-based partitioning strategy, which collectively enhance parallelism and reduce processing time. At 100 messages, C_3 achieves a latency of 13.2 ms, which is 16.5% lower than C_2 (15.8 ms) and 38.3% lower

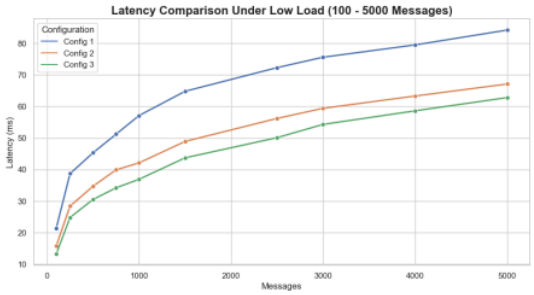

Fig. 2. Latency under Low Load

than C_1 (21.4 ms). This trend persists at 250 messages, where C_3 maintains a latency of 24.8 ms, compared to 28.4 ms for C_2 and 38.7 ms for C_1, highlighting a 35.9% improvement over the default configuration. The superior low-load performance of C_3 stems from its efficient partition parallelism and key-based partitioning, which ensures that related messages are routed to the same partition. This improves data locality and reduces retrieval time. Additionally, lower batch sizes and reduced linger time in C_3 minimize broker-to-broker synchronization delays, enhancing responsiveness.

The improved load distribution across multiple brokers reduces context switching and disk I/O overhead, resulting in faster message retrieval. Furthermore, optimized network round-trip times and reduced Garbage Collection (GC) pauses in C_3 further enhance its efficiency. As a result, C_3 proves highly effective for low-load, real-time streaming scenarios, delivering faster message processing and lower latencies, making it ideal for lightweight supply chain event processing.

2) **Medium Load**

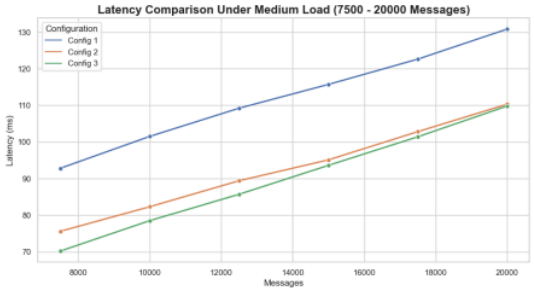

Fig. 3. Latency under Medium Load

Based on Fig. 3, we can derive that as the message volume increases, latency grows for all configurations due to higher processing demands. C_3 maintains the lowest latency across most of the medium load range,

showcasing its superior parallelism and distribution efficiency. However, towards the upper end of the range (near 20,000 messages), C_2 starts to overlap with C_3. The overlapping performance between C_2 and C_3 at higher message counts can be attributed to replication overhead in C_3. While C_3's higher replication factor offers better fault tolerance, it introduces replication-related delays under increased load. Thus, C_2 demonstrates more stable latency performance at medium loads due to fewer replication operations.

3) **High Load**

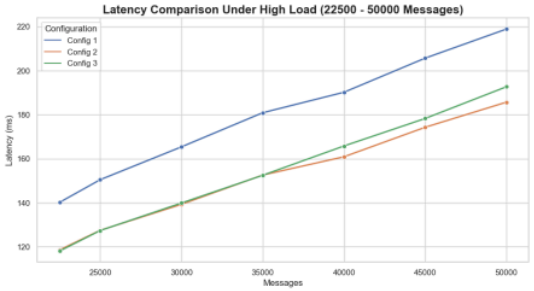

Fig. 4. Latency under High Load

At high message volumes, as depicted in Fig. 4, the latency difference between C_2 and C_3 becomes more noticeable. Although C_3 still offers better latency initially, the performance gap narrows and eventually reverses as the message load increases. C_2 achieves lower latency at the highest message volumes ($\geq$ 35,000 messages), while C_3's performance deteriorates slightly due to the replication overhead. Under high load, the benefits of increased replication in C_3 diminish due to replication-induced latency overhead. The lower replication factor in C_2 makes it more efficient for handling large-scale message loads, resulting in lower latency. This highlights a trade-off between fault tolerance and performance C_3 offers better fault tolerance but slightly higher latency at scale.

B. Kafka Request Time

The Kafka request time trend depicted in Fig. 5 reveals a consistent increase across all configurations as the message volume rises, but the magnitude of increase varies significantly across the three setups.

At low message volumes, C_3 exhibits the lowest request time due to its higher broker count, increased partitioning, and key-based partitioning strategy, which improve load distribution and reduce message processing overhead. For 100 messages, C_3 achieves a request time of 4.1 ms, which is 21.2% lower than C_2 (5.2 ms) and 52.3% lower than C_1 (8.6 ms). This efficiency persists at 250 messages, where C_3 records 3.7 ms, compared to 4.5 ms in C_2 and 6.1 ms in

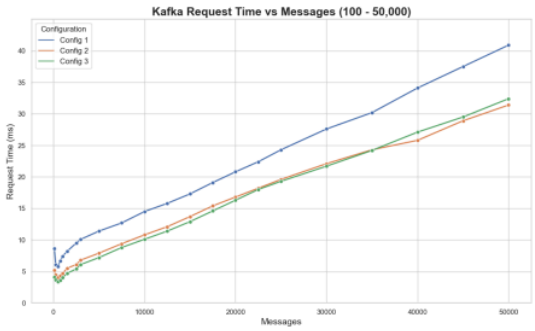

Fig. 5. Kafka Request Time vs Messages

C_1, demonstrating a 39.3% improvement over the baseline configuration.

As the message volume increases, the gap between the configurations widens, highlighting C_3's superior handling of higher loads. At 10,000 messages, C_3 maintains a request time of 10.1 ms, while C_2 records 10.8 ms and C_1 reaches 14.5 ms, resulting in a 30.3% improvement over the default configuration.

At low to moderate loads, C_3 consistently achieves the lowest request times, making it highly efficient for low-latency real-time operations. However, at extremely high volumes ($\geq$ 40,000 messages), its performance slightly deteriorates relative to C_2 due to increased overhead from key-based partitioning, which introduces routing complexity. For instance, at 40,000 messages, C_3 records 27.1 ms, slightly higher than C_2 at 25.8 ms. In contrast, C_2 benefits from its simpler round-robin strategy, which efficiently distributes large-scale workloads with lower partitioning overhead, making it more suitable for massive-scale batch processing.

C. Jitter

Fig. 6. Jitter vs Messages

The jitter trend shown in Fig. 6 presents a steady increase across all configurations as message volume rises, but with significant performance differences between the setups. Jitter measures the variability in message delivery times, and lower

Messages	Latency (ms)			Kafka Request Time (ms)			Jitter (ms)			Throughput (msgs/sec)		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
100	21.4	15.8	13.2	8.6	5.2	4.1	6.0	4.1	3.5	93.5	125.7	140.8
250	38.7	28.4	24.8	6.1	4.5	3.7	4.8	3.8	3.2	108.7	140.3	158.3
500	45.3	34.7	30.5	5.8	4.0	3.4	4.2	3.2	2.8	113.2	162.8	182.6
750	51.2	39.9	34.2	6.7	4.3	3.6	4.9	3.5	3.0	145.3	185.4	210.4
1000	57.1	42.1	36.9	7.4	4.7	4.0	5.1	3.6	3.1	180.4	235.1	260.4
1500	64.8	48.9	43.7	8.2	5.5	4.7	6.3	4.1	3.8	182.5	247.3	275.8
2500	72.3	56.2	50.1	9.5	6.1	5.4	7.1	4.9	4.3	206.6	280.6	310.9
3000	75.6	59.4	54.3	10.1	6.8	6.1	7.8	5.3	4.7	210.2	295.4	328.5
5000	84.2	67.1	62.8	11.4	7.9	7.2	8.3	6.1	5.5	212.7	312.9	345.1
7500	92.8	75.6	70.2	12.7	9.4	8.8	9.1	7.3	6.7	213.9	317.4	355.7
10000	101.5	82.3	78.5	14.5	10.8	10.1	10.6	8.2	7.5	214.3	320.1	360.8
12500	109.2	89.4	85.7	15.8	12.1	11.4	11.9	9.5	8.9	215.7	325.7	366.4
15000	115.7	95.1	93.6	17.3	13.7	12.9	13.2	10.9	10.2	215.2	324.9	365.1
17500	122.6	102.8	101.4	19.1	15.4	14.6	14.7	12.4	11.7	214.5	322.7	363.2
20000	130.8	110.3	109.8	20.8	16.8	16.3	15.9	13.6	13.1	213.1	320.4	360.9
22500	140.2	118.6	118.1	22.4	18.2	18.0	17.5	14.9	14.6	210.8	318.1	358.5
25000	150.5	127.5	127.4	24.3	19.6	19.3	18.7	15.7	15.4	208.3	316.4	355.2
30000	165.4	139.3	139.9	27.6	22.1	21.7	20.1	18.4	18.1	205.9	312.8	350.6
35000	180.9	152.6	152.5	30.2	24.3	24.2	22.5	20.9	20.6	200.7	308.7	345.3
40000	190.2	160.9	165.8	34.1	25.8	27.1	25.3	22.7	23.4	195.4	298.1	340.1
45000	205.7	174.3	178.3	37.5	28.9	29.5	28.1	25.4	25.8	180.2	282.6	325.4
50000	218.9	185.7	192.7	40.9	31.4	32.4	30.4	28.6	28.5	165.8	265.2	310.2

TABLE II
KAFKA PERFORMANCE METRICS ACROSS CONFIGURATIONS

jitter indicates more consistent and predictable message propagation.

At low message volumes, C_3 achieves the lowest jitter, highlighting its superior consistency due to its higher broker count, increased partitioning, and key-based strategy, which enhance data locality and reduce message queuing delays. For 100 messages, C_3 records a jitter of 3.5 ms, which is 14.6% lower than C_2 (4.1 ms) and 41.7% lower than C_1 (6.0 ms). Similarly, at 250 messages, C_3 achieves 3.2 ms, while C_2 reaches 3.8 ms and C_1 shows 4.8 ms, indicating a 33.3% improvement over the baseline.

As the message volume increases, C_3 continues to demonstrate more stable jitter compared to the other configurations. At 5,000 messages, C_3 records 5.5 ms, while C_2 shows 6.1 ms and C_1 spikes to 8.3 ms, resulting in a 33.7% improvement over the baseline. This is due to C_3's key-based partitioning, which ensures related messages are routed to the same partition, reducing rebalancing overhead and improving consistency.

At higher loads ($\geq$ 40,000 messages), the performance gap narrows, with C_2 and C_3 exhibiting almost identical jitter. For instance, at 50,000 messages, C_3 records 28.5 ms, just 0.1 ms lower than C_2 (28.6 ms). This convergence is due to the network saturation and processing bottlenecks that arise at massive volumes, limiting the advantage of key-based routing in C_3. Meanwhile, C_2 benefits from its simpler round-robin distribution, which reduces complexity during large-scale parallel processing, resulting in almost equivalent jitter values.

Overall, C_3 consistently maintains the lowest jitter at low and moderate loads, making it ideal for time-sensitive, real-time applications where consistent message delivery is essential. However, at extremely high volumes, C_2 achieves near-parity with C_3, making it equally viable for large-scale batch processing scenarios.

D. Throughput

Figs. 7, 8, & 9 illustrate the throughput performance of three Kafka configurations (C_1, C_2, C_3) under Low, Medium and High conditions respectively.

1) Low Load

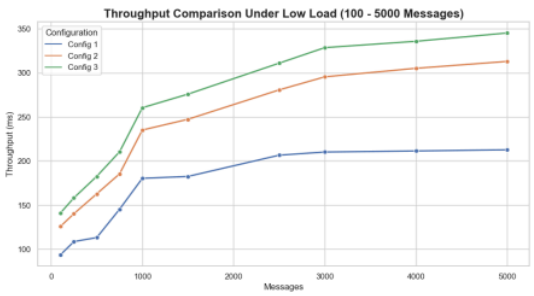

Fig. 7. Throughput under Low Load

At low message volumes, C_3 consistently achieves the

highest throughput, followed by C_2, with C_1 trailing behind, as depicted in Fig. 7. This performance trend underscores the effectiveness of C_3's increased parallelism, which is driven by a higher broker count, expanded partitioning, and efficient key-based message distribution. These architectural optimizations enable parallel consumers to process messages concurrently, reducing contention and significantly enhancing overall throughput. For instance, at 100 messages, C_3 achieves 140.8 msgs/sec, while C_2 processes 125.7 msgs/sec, and C_1 manages only 93.5 msgs/sec. As the message volume increases, the throughput gap widens. By the time the system reaches 5000 messages, C_3's throughput rises to 345.1 msgs/sec, whereas C_2 achieves 312.9 msgs/sec, and C_1 lags behind at 212.7 msgs/sec. This significant performance gap highlights how the enhanced partitioning and broker distribution in C_3 allows it to handle larger message loads more efficiently.

The throughput advantage of C_3 at low loads underscores the importance of parallelism and optimized resource allocation in Kafka. The use of multiple brokers and increased partitioning reduces the burden on individual nodes, enabling superior message distribution. Conversely, C_1 struggles due to its single broker bottleneck and limited partitioning capabilities, resulting in lower throughput, causing its performance to degrade quickly as the load increases, making it unsuitable for even moderately scaled Kafka operations.

2) **Medium Load**

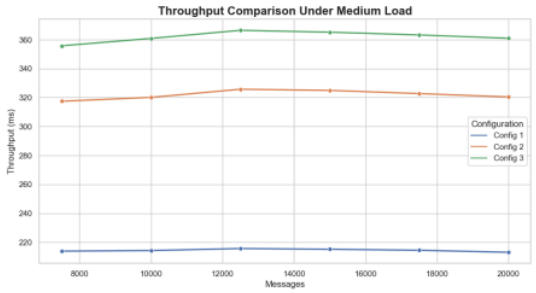

Fig. 8. Throughput under Medium Load

At medium message volumes, the throughput performance across the configurations stabilizes and exhibits a plateauing trend, highlighting the diminishing returns of scaling Kafka infrastructure. C_3 consistently outperforms C_2, although the gap between them narrows as the load increases. This convergence is driven by Kafka's internal overhead, including replication synchronization, disk I/O, and inter-broker coordination, which cap the performance gains from additional brokers and partitions. As the graph in Fig. 8 illustrates, both C_3 and C_2 reach a near-saturation point, where throughput improvements become marginal despite the rising message volume. This plateauing behavior reveals that Kafka's resource utilization becomes less efficient at medium loads, as the added replication and coordination costs counteract the benefits of parallelism. C_1, with its single-broker architecture, falls significantly behind, reflecting its limited scalability and increased bottleneck effect under moderate traffic conditions.

3) **High Load**

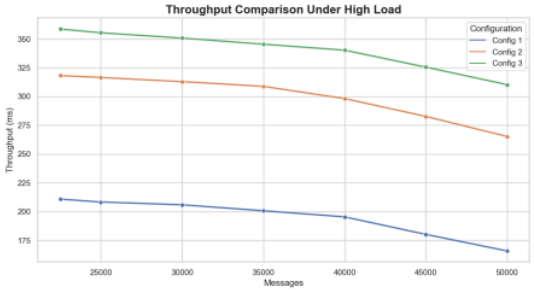

Fig. 9. Throughput under High Load

At high message volumes, the throughput across all configurations begins to decline, indicating the onset of resource saturation and Kafka's diminishing processing efficiency. While C_3 continues to lead, the performance gap between C_3 and C_2 narrows even further. Both configurations plateau and eventually exhibit a downward trend as the system reaches its capacity limits. The throughput dip is caused by intensified replication overhead, increased disk I/O contention, and inter-broker communication delays, which collectively reduce Kafka's ability to sustain higher message rates. The graph in Fig. 9 reveals that C_3 initially maintains the highest throughput, but its performance deteriorates faster due to its higher replication factor, which amplifies coordination overhead under heavy load. In contrast, C_2, with its lower replication burden, sustains throughput more effectively, closing the gap with C_3. This convergence highlights how reducing replication overhead enhances throughput efficiency during large-scale message processing. C_1, which is already constrained by its single-broker architecture, experiences a steeper throughput decline, as its limited capacity and lack of parallelism make it highly susceptible to saturation. The bottlenecked architecture struggles to manage the growing message load, resulting in a sharper drop in throughput.

VI. Conclusion and Key Findings

This study comprehensively evaluated the performance of a Kafka-based supply chain architecture under three configurations (C_1, C_2, and C_3) across varying load levels (low,

medium, and high). The results highlight significant performance differences in terms of latency, throughput, request time, and jitter, revealing key insights into the efficiency and scalability of Kafka-based event streaming systems.

1) **Superior Performance of C_3 at Low and Medium Loads**
   - Throughout the low and medium load scenarios, C_3 consistently achieved the lowest latency and jitter due to its higher broker count, increased partitioning, and key-based message distribution.
   - At low loads (1,000 to 10,000 messages), C_3 exhibited:
     - 35.3% lower latency and 41.7% lower jitter compared to C_1.
     - 17.0% higher throughput than C_2, making it the most efficient configuration for real-time event streaming.
   - The reduced request times and stable jitter values indicate superior consistency and reliability in C_3 at small-to-medium message volumes.
2) **Convergence of C_2 and C_3 at Higher Loads**
   - At higher message volumes (30,000 to 50,000), the performance gap between C_2 and C_3 narrows due to replication-induced overhead in C_3.
   - C_2 outperforms C_3 slightly in terms of latency and jitter at extremely high loads, making it better suited for batch processing scenarios.
   - C_2's simpler round-robin distribution and lower replication factor reduce overhead, enhancing its efficiency for large-scale workloads.
3) **Kafka Saturation at Extremely High Loads**
   - As the message volume increases beyond 40,000, all configurations experience diminishing throughput and rising latency due to Kafka's disk I/O contention, inter-broker coordination, and replication overhead.
   - The throughput decline highlights Kafka's processing saturation point at massive loads, requiring further optimizations such as broker scaling, dynamic partitioning, and consumer rebalancing to sustain efficiency.
4) **Real-World Implications:**
   - For low-latency, real-time supply chain event processing, C_3 is the most effective configuration, offering superior responsiveness and stability.
   - For large-scale, high-volume batch processing, C_2 offers better stability with lower replication overhead, making it more suitable for throughput-centric workloads.
   - C_1, with its single-broker setup, is inefficient for all but the smallest workloads, making it unsuitable for production-scale Kafka operations.

Overall, the study demonstrates that C_3 offers the most consistent and scalable performance for real-time streaming at small-to-medium loads, while C_2 becomes more efficient for large-scale batch processing. These findings provide practical insights for designing and optimizing Kafka-based supply chain architectures, enabling high throughput, low-latency, and fault-tolerant event streaming.

REFERENCES

[1] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: yesterday, today, and tomorrow*. 01 2017.

[2] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybylek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:1–1, 01 2022.

[3] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[4] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. 2015.

[5] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[6] Jay Kreps, Neha Narkhede, and Jun Rao. Benchmarking apache kafka: 2 million writes per second on three cheap machines, 2016. LinkedIn Engineering Blog.

[7] Jay Kreps. Kafka : a distributed messaging system for log processing. 2011.

[8] V Ganesh Tejas and Dr. Hemavathy R. Microservices and its intercommunication using kafka. *International Research Journal of Engineering and Technology (IRJET)*, 07(04), April 2020.

[9] Kiran Peddireddy. Enhancing inventory management systems through kafka data stream integration. *International Advanced Research Journal in Science, Engineering and Technology*, 8(9), September 2021.

[10] Sören Henning and Wilhelm Hasselbring. Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. *Journal of Systems and Software*, 208:111879, 2024.

[11] Suwarna Shukla and Prabhneet Singh. Revolutionizing supply chain management: Real-time data processing and concurrency management using kafka and akka. *International Journal of Innovative Science and Research Technology*, 9(5), May 2024.

[12] Chandrakanth Lekkala. Designing high-performance, scalable kafka clusters for real-time data streaming. 01 2021.

[13] J. Rao. How to choose the number of topics/partitions in a kafka cluster?, March 2015. [Posted 12-March-2015, Accessed 06-April-2022].

[14] Satish Krishnamurthy, Ashvini Byri, Ashish Kumar, Dr. Satendra Pal Singh, Om Goel, and Prof. Dr. Punit Goel. Utilizing kafka and real-time messaging frameworks for high-volume data processing. *(IJPREMS)*, 02(02), February 2022.

[15] Theofanis P. Raptis and Andrea Passarella. On efficiently partitioning a topic in apache kafka. May 2022.

[16] Sameer Shukla. Exploring the power of apache kafka: A comprehensive study of use cases. *International Journal of Latest Engineering and Management Research (IJLEMR)*, 08(03), March 2023.

[17] Tejas V and Dr V. Development of kafka messaging system and its performance test framework using prometheus. *International Journal of Recent Technology and Engineering (IJRTE)*, 9:1622–1626, 05 2020.

[18] Guenter Hesse, Christoph Matthies, and Matthias Uflacker. How fast can we insert? an empirical performance evaluation of apache kafka. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, page 641–648. IEEE, December 2020.

[19] Roman Wiatr, Renata Słota, and Jacek Kitowski. Optimising kafka for stream processing in latency sensitive systems. In *7th International Young Scientist Conference on Computational Science*, 2018.

[20] Cloudera Inc. *Tuning Apache Kafka Performance*, 2024. Accessed: 2025-02-27.

Optimizing Kafka-Based Supply Chain Architectures:

ORIGINALITY REPORT

0%	0%	0%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

Exclude quotes	On	Exclude matches	< 1%
Exclude bibliography	On		