Evaluating Apache Kafka Performance and Operational Efficiency: A Comparative Study of ZooKeeper and KRaft Architectures

Ramesh V.
AT&T
3400 W Plano Pkwy, Plano, TX 75075

ABSTRACT
Apache Kafka is a leading platform for building scalable, distributed event streaming systems. Traditionally, Kafka has relied on Apache ZooKeeper for managing cluster metadata and coordinating controller elections. However, the recent introduction of KRaft (Kafka Raft Metadata mode) eliminates this dependency by embedding a Raft-based consensus mechanism directly within Kafka [1] [6]. This architectural evolution raises key questions about the comparative performance, reliability, and operational efficiency of ZooKeeper-based versus KRaft-based deployments. [7]

This study presents a comprehensive performance evaluation of Kafka's ZooKeeper and KRaft modes across multiple dimensions, including topic scalability, producer throughput, controller failover response, and memory efficiency. Through reproducible benchmarks involving 1,000-topic workloads, multi-threaded producers, and real-world failure simulations, the report analyzes the behavioral differences between the two architectures. The findings offer valuable insights for platform engineers, DevOps practitioners, and architects seeking to optimize Kafka deployments for high-throughput, cloud-native environments. [9] [15]

General Terms
Distributed Systems, Event Streaming, System Performance; Scalability, Fault Tolerance

Keywords
Apache Kafka, ZooKeeper, KRaft, distributed systems, Raft consensus, event streaming, performance benchmarking

1. INTRODUCTION
Apache Kafka has become a critical backbone of modern data ecosystems, powering real-time event streaming across diverse domains such as financial services, IoT, e-commerce, and analytics. As deployment scale increases, the architectural design of Kafka clusters—particularly metadata management—plays a central role in determining overall performance, reliability, and ease of operation. Traditionally, Kafka has used Apache ZooKeeper as its external coordination layer for managing cluster metadata and controller elections. While reliable, this introduces complexity in monitoring, failover handling, and operational tuning. To address these challenges, Kafka introduced KRaft (Kafka Raft Metadata mode)—a self-contained mode that integrates metadata management and consensus directly within Kafka using the Raft algorithm. [6] [7]

This paper presents a comparative evaluation of Kafka's ZooKeeper-based and KRaft-based deployments across multiple performance dimensions. The analysis includes metadata scalability during topic creation, producer throughput optimization, memory behavior under load, and controller failover responsiveness. Using uniform test environments and reproducible workloads, the study offers practical insights into the trade-offs and operational benefits of each architecture.

Key contributions include:

- Comprehensive benchmarking of cluster behavior at scale under both architectures
- In-depth analysis of Kafka producer tuning (batch size, compression, concurrency) in KRaft
- Evaluation of resilience through fault-injection scenarios for controller transitions
- Actionable configuration and tuning recommendations for JVM and cluster components

2. KAFKA ARCHITECTURE AND DEPLOYMENT DESIGN
Apache Kafka supports two architectural models for managing metadata and cluster coordination: the traditional ZooKeeper-based architecture and the newer KRaft-based architecture. Both models facilitate communication between producers, brokers, and consumers, but differ significantly in how control-plane responsibilities are handled.

2.1 Kafka with ZooKeeper
In its legacy form, Kafka relies on Apache ZooKeeper for managing broker metadata, controller election, and configuration persistence. ZooKeeper acts as an external coordination service, tracking the state of active brokers, facilitating controller elections, and storing configuration data such as topic partitions and access policies. When a producer initiates a message send, it contacts a Kafka broker to retrieve partition leadership metadata. If not cached, the broker queries ZooKeeper to determine the appropriate partition leader. This metadata resolution process is transparent to the producer. Once resolved, the producer pushes records directly to the leader broker for the given partition. [2] [3]

Figure 1 illustrates the core components and data flow of this architecture. Producers send messages to Kafka brokers, which in turn interact with the ZooKeeper ensemble for coordination tasks such as registering broker availability and electing the active controller.

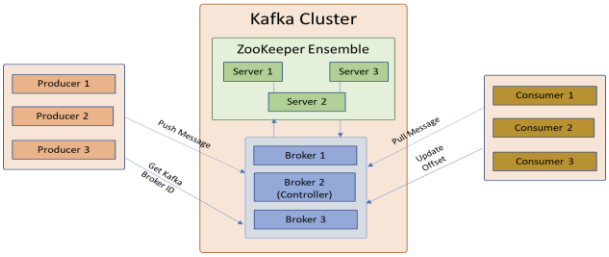

Figure 1. Kafka Architecture Using ZooKeeper for Metadata and Controller Management.

The Kafka controller, dynamically elected via ZooKeeper, is responsible for assigning partition leadership, responding to broker failures, and initiating recovery procedures. ZooKeeper ensures that brokers are monitored through ephemeral nodes and facilitates the re-election of a new controller upon the failure of the current one.

On the consumer side, clients issue fetch requests to retrieve messages from brokers and commit offsets to track progress. Since Kafka 0.9, consumer offsets are stored in an internal Kafka topic (__consumer_offsets), reducing pressure on ZooKeeper.

While widely deployed, this model introduces operational challenges. The dependency on ZooKeeper requires dedicated monitoring, tuning, and high availability configuration. During metadata-intensive operations—such as rapid topic creation or large-scale partition rebalancing—ZooKeeper may become a bottleneck, affecting cluster responsiveness.

2.2 Kafka with KRaft

Kafka Raft (KRaft) mode eliminates the ZooKeeper dependency by embedding a Raft-based consensus protocol directly within the Kafka brokers. This architecture decentralizes metadata coordination and provides a self-contained quorum-based control plane. [4][10]

In this model, a dedicated set of controller nodes manages the cluster metadata. These controllers replicate an event-sourced metadata log using Raft consensus. All metadata changes—such as topic creation, leader election, or configuration updates—are recorded as immutable entries in this log and replicated across the quorum to maintain consistency. Figure 2 illustrates the KRaft-based architecture, highlighting the separation between the controller quorum and the broker layer. Brokers consume metadata updates from the replicated log and maintain consistency with the controllers. Producers directly query brokers for partition metadata and send messages without the need for external coordination. Similarly, consumers communicate solely with brokers to pull messages and commit offsets.[8]

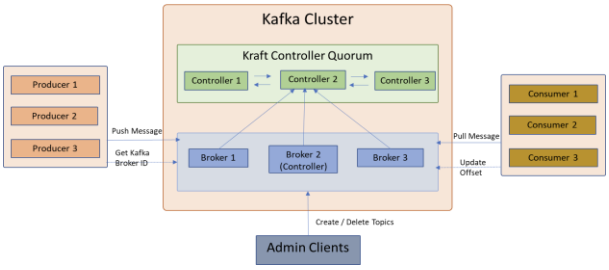

Figure 2. Kafka Architecture in KRaft Mode with Internal Metadata Quorum and Event-Sourced Control.

Administrative operations, such as creating or deleting topics, are routed through the active controller and committed to the metadata log. New brokers or controllers can bootstrap by reading from this log or its associated snapshots, allowing for faster and more scalable state recovery.

KRaft offers improved failover speed, reduced latency during metadata lookups, and simplified operational maintenance. Its architecture is better suited for modern cloud-native environments, where dynamic scaling, automation, and performance isolation are critical.

2.3 Infrastructure and Cluster Design

To ensure a fair and practical evaluation of ZooKeeper-based and KRaft-based Kafka architectures, both clusters were deployed using Docker Compose with consistent tuning across brokers. Each deployment was designed to reflect commonly used configurations in real-world environments, while focusing on minimizing infrastructure overhead.

ZooKeeper-Based Kafka Cluster

The ZooKeeper-based cluster was deployed with three Kafka brokers (kafka1, kafka2, kafka3) and one ZooKeeper node (zookeeper). Each broker is connected to the ZooKeeper service via KAFKA_ZOOKEEPER_CONNECT. Brokers were configured with internal and external listener mappings, static heap memory allocation (-Xms4g -Xmx4g), and appropriate replication factors for internal topics such as __consumer_offsets and transaction logs. JMX ports were exposed for monitoring, and native memory tracking was enabled to capture JVM-level metrics. While the test used a single ZooKeeper container for simplicity, this represents a typical base configuration and could be expanded to a replicated ensemble in production.

KRaft-Based Kafka Cluster

The KRaft-based Kafka cluster was deployed using the KRaft metadata mode (KIP-866) with three nodes—kafka1, kafka2, and kafka3—each configured to act as both a broker and a controller. This co-located design is consistent with how KRaft is commonly deployed in smaller or mid-sized clusters. All nodes were assigned unique KAFKA_NODE_ID values and shared KAFKA_CONTROLLER_QUORUM_VOTERS string to establish a Raft quorum for metadata management.

A two-phase startup approach was used. In the first phase (docker-compose.pause.yml), nodes were launched in a paused state using a no-op command to allow for explicit formatting of metadata storage using kafka-storage format. In the second phase (docker-compose.active.yml), brokers were started using kafka-server-start with externalized server.properties configurations mounted into each container.

Each KRaft node exposed a PLAINTEXT listener for inter-broker and client communication, and a CONTROLLER listener for Raft-based metadata coordination. Heap memory was statically set to 2 GB, and performance tuning options such as increased network and I/O threads were configured. Metrics and JMX exposure were also enabled to maintain parity with the ZooKeeper-based deployment.

Node Efficiency

The ZooKeeper-based architecture required four total containers (3 brokers + 1 ZooKeeper), while the KRaft-based setup required only three, each fulfilling dual roles. This reflects a reduction in infrastructure usage and operational complexity. In the KRaft model, control-plane and data-plane responsibilities are unified within the same nodes, reducing the coordination burden and eliminating external ZooKeeper dependencies.

3. KAFKA PARTITION SCALABILITY: COMPARATIVE ANALYSIS OF ZOOKEEPER-BASED VS. KRAFT-BASED CLUSTERS

3.1 Objective

This benchmark evaluates how Apache Kafka scales from 250 to 1000 topics or above (~10,000 partitions) under ZooKeeper and KRaft metadata modes. The test focuses on JVM memory usage (heap, class, thread, code cache), broker health, and metadata behavior during sustained topic growth, simulating enterprise-grade multi-tenant workloads.

3.2 Experimental Setup

The testing methodology involved **progressively creating topics in batches**, targeting milestones of **500, 1000, and beyond**, while monitoring **heap usage, class space, and metaspace consumption** at each stage. Additionally, **cluster health and stability** were closely observed to determine the **failure thresholds** under increasing metadata load. JVM-level metrics were collected uniformly via **JMX and CLI tools**, using Docker-based setups to ensure reproducibility and consistent monitoring across both architectures.

3.3 Heap Usage and Stability Comparison: Kafka KRaft vs Zookeeper under Topic Scaling Load

This chart illustrates the heap memory usage trends across two Kafka deployment models—KRaft (left) and ZooKeeper (right)—based on two controlled topic creation tests.

- X-Axis: Number of topics created (500, 1000, and the crash region)
- Y-Axis: Heap used (in MB), with annotated heap usage % relative to the broker's maximum allocation (4 GB)

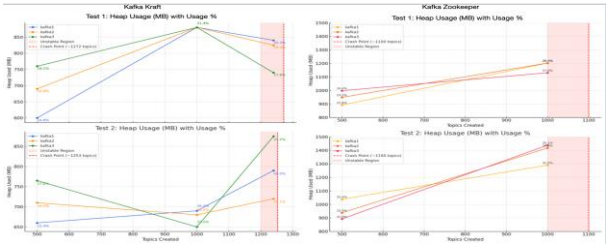

Figure 3. Heap Usage Comparison Between Kafka KRaft and ZooKeeper-Based Clusters

One of the most notable advantages of KRaft mode is its lower heap usage at scale. In KRaft Test 1, heap usage across all brokers peaked around 880 MB at 1000 topics, and even showed a slight reduction near the crash point at approximately 1272 topics. In contrast, ZooKeeper Test 1 exhibited significantly higher memory consumption, with heap usage rising beyond 1.2 GB at 1000 topics, and no signs of garbage collection recovery or stabilization, ultimately leading to a crash shortly after 1100 topics. This demonstrates that KRaft handles memory management far more efficiently and gracefully under load compared to the traditional ZooKeeper setup.

Another key strength of KRaft lies in its predictable and stable memory usage patterns. In KRaft Test 2, heap usage grew steadily and consistently as topics were created, indicating well-managed resource scaling. On the other hand, ZooKeeper Test 2 showed a sharp and erratic increase in memory consumption, particularly for kafka2 and kafka3, with usage soaring to nearly 35% (around 1.4 GB) at 1000 topics. This abrupt spike suggests that ZooKeeper-based Kafka clusters are more vulnerable to garbage collection pressure and instability when subjected to high rates of metadata operations.

Most importantly, KRaft demonstrated a higher crash threshold, with the cluster staying healthy until nearly 1272 topics were created. Meanwhile, ZooKeeper-based clusters failed consistently after just 1100 topics across both test runs. This confirms KRaft's superior scalability and robustness, especially during intensive operations such as large-scale topic provisioning.

3.4 MetaSpace and ClassSpace Usage Comparison – Kafka KRaft vs. ZooKeeper-Based Kafka

This figure compares JVM MetaSpace and ClassSpace memory consumption patterns between Kafka KRaft (left) and ZooKeeper-based Kafka (right) under different topic loads.

- X-Axis: Number of topics created (500, 1000, and 1240 for KRaft; 500 and 1000 for ZooKeeper)
- Y-Axis: Memory usage in kilobytes (KB)

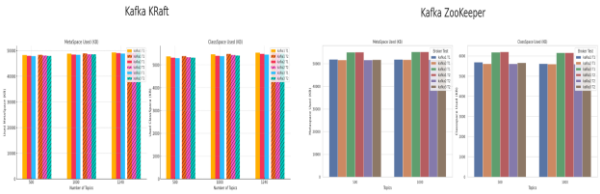

Figure 4. Metaspace, Classspace Usage Comparison Between Kafka KRaft and ZooKeeper-Based Clusters

The memory usage comparison between Kafka KRaft and ZooKeeper-based clusters reveals a consistent advantage for KRaft in terms of both MetaSpace and ClassSpace efficiency. Across all topic volumes—500, 1000, and near the instability threshold at 1240 topics—the KRaft-based brokers demonstrate lower and more stable memory consumption compared to their ZooKeeper counterparts.

In MetaSpace usage, KRaft brokers consistently use less memory, with kafka1, kafka2, and kafka3 maintaining values below 49,200 KB even at 1240 topics. In contrast, ZooKeeper-based brokers show noticeably higher MetaSpace usage, with kafka3 exceeding 56,000 KB at 1240 topics. This upward trend in ZooKeeper's memory footprint becomes more pronounced as the topic count increases, indicating steeper memory growth and potential for earlier saturation.

A similar pattern is observed in ClassSpace usage. KRaft brokers remain in a tight range around 5600 KB to 5700 KB, while ZooKeeper-based brokers show a significantly higher and uneven spread, especially with kafka3 reaching 6250 KB at 1240 topics. Not only is the absolute usage higher under ZooKeeper, but the imbalance between brokers is more evident, which could lead to uneven performance or premature failures on specific nodes.

Overall, the analysis confirms that KRaft handles topic metadata with greater memory efficiency and consistency. This makes KRaft more scalable and better suited for high-topic-count workloads. The flatter memory growth curve and lower absolute usage in both MetaSpace and ClassSpace positions KRaft as the more robust option for deployments aiming to

reduce memory overhead and improve cluster stability as metadata scales.

3.5 Conclusion

The comparison between Kafka KRaft and ZooKeeper-based Kafka across both heap memory and metaspace/classspace usage reveals a clear architectural advantage in favor of the KRaft mode. In terms of **heap usage**, KRaft consistently demonstrated lower memory consumption under load. Even when the number of topics exceeded 1200, the KRaft brokers maintained stable heap profiles with signs of garbage collection recovery and no abrupt spikes. In contrast, ZooKeeper-based Kafka showed aggressive and uneven heap growth, with usage reaching over 1.4 GB by the 1000-topic mark in some brokers, eventually leading to cluster failure at around 1100 topics. This indicates that KRaft manages memory more efficiently and can sustain heavier metadata workloads without becoming unstable. [15][16]

When examining **Metaspace and ClassSpace usage**, the trend continues in favor of KRaft. The Kafka KRaft brokers maintained tightly grouped metaspace and classspace usage across all tests, showing only a modest and predictable increase as the topic count grew from 500 to over 1200. The values remained around 48,000–49,000 KB for metaspace and around 5,400–5,600 KB for classspace, reflecting efficient class loading and metadata retention. On the other hand, ZooKeeper-based Kafka brokers showed notably higher metaspace usage (often crossing 55,000 KB), along with wider variation across brokers and tests. Classspace usage also followed a similar pattern, with ZooKeeper-based brokers consuming more memory and exhibiting inconsistent growth.

In summary, Kafka KRaft demonstrates superior memory efficiency—not only in heap utilization but also in the JVM's metaspace and classspace areas—resulting in more stable, scalable, and predictable performance. These findings make a strong case for adopting KRaft in production environments, especially where large-scale topic creation and metadata handling are central to system behavior. [14]

4. KAFKA PRODUCER PERFORMANCE BENCHMARK: KAFKA KRAFT VS ZOOKEEPER-BASED CLUSTERS

4.1 Objective

This test compares Kafka producer performance between KRaft and ZooKeeper-based clusters under identical, fully optimized configurations. The goal is to evaluate differences in throughput and latency and determine which architecture offers better suitability for throughput-intensive versus latency-sensitive workloads.

4.2 Experimental Setup

A total of 1,000,000 messages were produced over an 8-second test window to each cluster type. Both the ZooKeeper-based and KRaft-based Kafka clusters were composed of three brokers with identical hardware and JVM configurations. The producer was tuned using batch sizing, GZIP compression, and multi-threaded concurrency to reflect a high-performance deployment scenario.

4.3 Results Overview

Metric	ZooKeeper-Based Kafka	KRaft-Based Kafka
Peak Throughput	190,585 records/sec	180,115 records/sec
Average Throughput	188,836 records/sec	175,465 records/sec
Average Latency	406.6 ms	341.7 ms
Maximum Latency	896.0 ms	786.0 ms

Visual snapshots were captured to support this comparison, showing producer throughput, latency statistics, and final performance confirmation from each cluster type.

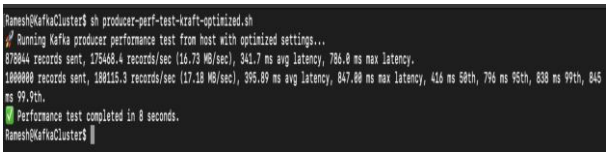

Figure 5. Final producer performance in KRaft-based cluster

Figure 6. Final producer performance in ZooKeeper-based cluster

4.4 Analysis

While ZooKeeper-based Kafka yielded slightly higher throughput—between 5% to 8% more—KRaft consistently demonstrated lower latency across all metrics. The KRaft cluster's lower average and maximum latency illustrates its efficiency in time-sensitive operations, especially under concurrent producer pressure. These results suggest that when low latency is prioritized over absolute throughput, KRaft presents a better architectural choice.

4.5 Conclusion

ZooKeeper-based Kafka remains a viable choice for workloads where peak throughput is critical. However, for real-time analytics, telemetry pipelines, and latency-sensitive data processing, KRaft proves to be the more robust solution. Its ability to maintain consistent low latency, even under full producer tuning, positions it as a preferred architecture for modern, delay-sensitive Kafka deployments.[2] [3]

5. Kafka Topic Creation Benchmark: Zookeeper vs. KRaft Mode

5.1 Objective

This benchmark evaluates the control-plane scalability of Apache Kafka by comparing topic creation performance in ZooKeeper-based and KRaft-based clusters. It highlights how each architecture handles large-scale metadata operations, with a focus on responsiveness, memory behavior, and broker stability. [4]

5.2 Experimental Setup

A total of 1,000 topics were created, each with 10 partitions and a replication factor of 3. The Kafka clusters—both ZooKeeper and KRaft based—used three brokers with identical hardware and JVM configurations. Metrics observed included total time to complete topic creation, JVM heap trends, and broker availability throughout the process.

5.3 Results Overview

Metric	ZooKeeper-Based Kafka	KRaft-Based Kafka
Total Time	3328 seconds (~55.5 mins)	2054 seconds (~34.2 mins)
Brokers Alive	3 of 3	3 of 3
Throughput Pattern	Slowed after 750 topics	Remained stable throughout
Heap Behavior	Gradual memory increase	Efficient and controlled

Snapshots were taken after topic creation to visualize execution output and verify consistency in behavior and performance.

Figure 7. Topic creation output – ZooKeeper-based Kafka

Figure 8. Topic creation output – KRaft-based Kafka

One of the most notable advantages of KRaft mode is its lower heap usage at scale. In KRaft Test 1, heap usage across all brokers peaked around 880 MB at 1000 topics, and even showed a slight reduction near the crash point at approximately 1272 topics. In contrast, ZooKeeper Test 1 exhibited significantly higher memory consumption, with heap usage rising beyond 1.2 GB at 1000 topics, and no signs of garbage collection recovery or stabilization, ultimately leading to a crash shortly after 1100 topics. This demonstrates that KRaft handles memory management far more efficiently and gracefully under load compared to the traditional ZooKeeper setup.

Another key strength of KRaft lies in its predictable and stable memory usage patterns. In KRaft Test 2, heap usage grew steadily and consistently as topics were created, indicating well-managed resource scaling. On the other hand, ZooKeeper Test 2 showed a sharp and erratic increase in memory consumption, particularly for kafka2 and kafka3, with usage soaring to nearly 35% (around 1.4 GB) at 1000 topics. This abrupt spike suggests that ZooKeeper-based Kafka clusters are more vulnerable to garbage collection pressure and instability when subjected to high rates of metadata operations.

Most importantly, KRaft demonstrated a higher crash threshold, with the cluster staying healthy until nearly 1272 topics were created. Meanwhile, ZooKeeper-based clusters failed consistently after just 1100 topics across both test runs. This confirms KRaft's superior scalability and robustness, especially during intensive operations such as large-scale topic provisioning. [5]

5.4 Analysis

The ZooKeeper-based cluster showed a clear drop in performance after about 750 topics, likely due to external metadata coordination overhead. In contrast, the KRaft-based setup remained stable throughout the test and completed the operation roughly 38% faster. KRaft also exhibited more consistent memory behavior, with less heap fluctuation across brokers.

5.5 Conclusion

KRaft mode demonstrates stronger control-plane scalability and operational efficiency under high-volume topic creation. Its internal metadata management avoids the delays and synchronization costs associated with ZooKeeper, making it more suitable for modern dynamic workloads and large-scale Kafka environments. [16]

6. KAFKA CONTROLLER FAILOVER EVALUATION: ZOOKEEPER VS. KRAFT MODE

6.1 Objective

This test evaluates the resilience and responsiveness of Kafka clusters when subjected to controller-level failure scenarios. It measures failover time, behavior during controller election, and overall cluster stability in both ZooKeeper and KRaft architectures.

6.2 Experimental Setup

Both ZooKeeper and KRaft clusters consisted of three brokers. The following failure scenarios were tested on each:

S1: Kill the current controller broker

S2: Kill a non-controller broker

S3: Kill the controller and another broker simultaneously

S4: Stop all brokers and restart the cluster

6.3 Results Overview

Scenario	Action	ZooKeeper Recovery	KRaft Recovery	New Controller	URP
S1	Kill kafka1 (controller)	18s (undetected)	6s	kafka2	0
S2	Kill kafka2 (non-controller)	18s (undetected)	7s	kafka1	0
S3	Kill kafka1 + kafka2	14s (kafka1 active)	13s	kafka1	0
S4	Restart all brokers	11s (kafka1 active)	16s	kafka1	0

Controller logs and CLI output were captured to illustrate cluster response and controller transition behavior.

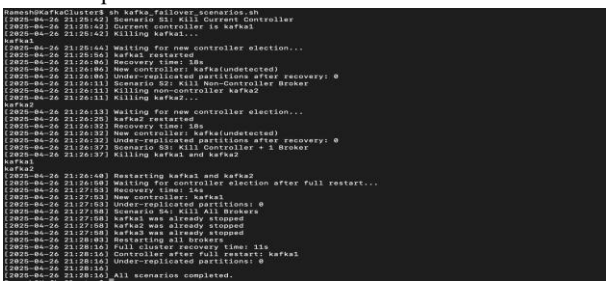

Figure 9. Failover logs and recovery behavior – ZooKeeper-based Kafka

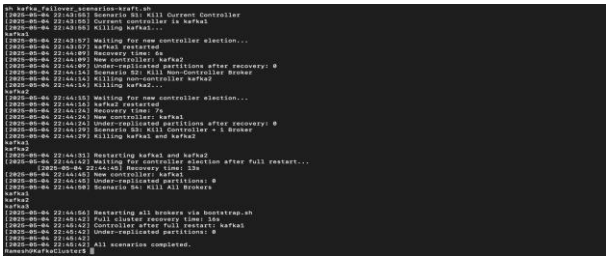

Figure 10. Failover logs and recovery behavior – KRaft-based Kafka

6.4 Analysis

In single-node failure scenarios (S1 and S2), KRaft consistently achieved faster failover times, quickly detecting controller loss and re-electing leadership. ZooKeeper-based clusters took longer, with controller failover sometimes going undetected for extended periods. Although both modes maintained high availability and experienced no data loss or URPs, the visibility of the election process in KRaft—enabled by its Raft log and clearer CLI output—proved significantly more traceable than ZooKeeper's log-based behavior.

6.5 Conclusion

Both deployment modes demonstrate strong fault tolerance; however, KRaft offers more efficient and observable controller transitions. Its internal Raft-based election mechanism enables quicker leadership recovery and improved operational clarity. These results, together with earlier findings in topic creation and producer performance, reinforce KRaft's production readiness—particularly for environments demanding low-latency recovery and operational transparency.[12] [15]

7. SUMMARY AND FINAL REMARKS

This study benchmarked Apache Kafka across ZooKeeper-based and KRaft-based architectures, evaluating key operational aspects including partition scalability, producer throughput, metadata handling, and controller failover. Both configurations proved stable under stress, but distinct differences emerged in performance and operational efficiency.

KRaft showed clear advantages in metadata management, with consistent memory usage and improved scalability during large-scale topic creation. Producer tuning experiments confirmed that enabling batching, compression, and multithreading significantly improved throughput and reduced latency—particularly under KRaft, where integrated coordination mechanisms scale more efficiently. Although ZooKeeper-based clusters achieved slightly higher peak throughput in some cases, KRaft consistently delivered lower latency and faster controller failover, making it better suited for real-time, latency-sensitive applications.

More importantly, KRaft simplifies Kafka's control plane by eliminating the need for an external coordination service. This architectural shift reduces deployment complexity, removes ZooKeeper-specific failure domains, and enables a unified, maintainable control layer. In practice, the KRaft cluster fulfilled both broker and controller roles using just three nodes, compared to the six-node requirement in a typical ZooKeeper-based setup—demonstrating a significant reduction in infrastructure and operational overhead.

As Kafka adoption grows in containerized and cloud-native environments, KRaft's operational simplicity becomes a critical advantage. It improves observability, accelerates failover, and enables Kafka to scale with fewer components and reduced configuration burden. [11]

In conclusion, KRaft mode provides a production-ready, efficient, and resilient architecture for managing Kafka's distributed metadata and control operations. Future work may explore multi-cluster replication, cross-region controller quorum behavior, and long-term scalability under dynamic workloads. [13] [14]

8. ACKNOWLEDGMENTS

I sincerely thank the AT&T research paper reviewers for their insightful feedback, which greatly contributed to improving this work.

9. REFERENCES

[1] Apache Kafka Documentation, "KRaft Mode (KIP-500) – Removing the dependency on ZooKeeper," Apache, 2024. [Online]. Available: https://kafka.apache.org/documentation/

[2] Apache ZooKeeper Documentation, "Apache ZooKeeper Overview and Configuration," Apache, 2024. [Online]. Available: https://zookeeper.apache.org/doc/

[3] Confluent, "Apache Kafka Performance"[Online]. Available: https://developer.confluent.io/learn/kafka-performance/

[4] Confluent Developer, "Understanding KRaft Mode," 2024. [Online]. Available: https://developer.confluent.io/learn/kraft/

[5] Spoud.io, "Embracing the Future of Kafka: Why It's Time to Migrate from ZooKeeper to KRaft," *Medium*, 2024. [Online]. Available: https://spoud-io.medium.com/embracing-the-future-of-kafka-why-its-time-to-migrate-from-zookeeper-to-kraft-f1a5225ac48a

[6] Apache Kafka Improvement Proposal (KIP-500), "Replace ZooKeeper with a Self-Managed Metadata Quorum," Apache, 2024. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum

[7] Apache Kafka Improvement Proposal (KIP-631), "The Quorum-based Kafka Controller," Apache, 2024. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-631%3A+The+Quorum-based+Kafka+Controller

[8] Confluent, "The Architecture of KRaft Mode: How Kafka Is Evolving," 2024. [Online]. Available: https://docs.confluent.io/platform/current/kafka-metadata/kraft.html

[9] Confluent, "Why replace ZooKeeper with Kafka Raft

(KRaft)," 2024. [Online]. Available: https://www.confluent.io/blog/why-replace-zookeeper-with-kafka-raft-the-log-of-all-logs/

[10] Confluent Documentation, "KRaft Overview," 2024. [Online]. Available: https://docs.confluent.io/platform/current/kafka-metadata/kraft.html

[11] Confluent Documentation, "Migrate from ZooKeeper to KRaft on Confluent Platform," 2024. [Online]. Available: https://docs.confluent.io/platform/current/installation/migrate-zk-kraft.html

[12] Confluent, "Kafka 4.0 Release: Default KRaft, Queues, Faster Rebalances," 2024. [Online]. Available: https://www.confluent.io/blog/latest-apache-kafka-release/

[13] SoftwareMill, "Apache Kafka 4.0: Simplified Architecture with Default KRaft," 2024. [Online]. Available: https://softwaremill.com/apache-kafka-4-0-0-released-kraft-queues-better-rebalance-performance/

[14] The New Stack, "Kafka Drops ZooKeeper for 'Real-Time' KRaft," 2024. [Online]. Available: https://thenewstack.io/kafka-drops-zookeeper-for-real-time-kraft/

[15] Arvind Kumar, "Deep Dive: How KRaft Improves Over ZooKeeper in Kafka," Medium, 2025. [Online]. Available: https://codefarm0.medium.com/deep-dive-how-kraft-improves-over-zookeeper-in-kafka-f50b971e0c0e

[16] Sion Smith, "Apache Kafka's KRaft Protocol: How to Eliminate ZooKeeper and Boost Performance by 8x," OSO, 2025. [Online]. Available: https://oso.sh/blog/apache-kafkas-kraft-protocol-how-to-eliminate-zookeeper-and-boost-performance-by-8x/