QIT Press - International Journal of Cloud Computing (QITP-IJCC)
Journal ID: 4739-2815
Volume 5, Issue 2, July-December 2025, pp. 12-23
DOI: https://doi.org/10.63374/QITP-IJCC_05_02_002

Real-Time Data Ingestion and Stream Processing for AI Applications in Cloud-Native Environments

Gopi Kathiresan
Senior Software Engineer, Morgan Stanley, Atlanta GA, USA.

Published on: 07th August 2025

Citation: Gopi Kathiresan. (2025). Real-Time Data Ingestion and Stream Processing for AI Applications in Cloud-Native Environments. *QIT Press - International Journal of Cloud Computing (QITP-IJCC),* 5(2), 12–23.
DOI: https://doi.org/10.63374/QITP-IJCC_05_02_002

Full Text: https://qitpress.com/articles/QITP-IJCC/VOLUME_5_ISSUE_2/QITP-IJCC_05_02_002.pdf

Abstract

The fast-growing development of AI uses in areas like predictive analytics, computer vision, and NLP applications demand ultra-low-latency high-throughput data pipelines that can scale. In this paper, the author describes an end-to-end architectural design of fault-tolerant, elastic, and AI-inference responsive real-time data ingestion and stream processing in cloud-native environments. We present the comparative analysis of the state-of-the-art distributed stream processing frameworks, such as Apache Kafka, Flink, Pulsar, and Hazelcast Jet, on the realistic AI workloads. Benchmarks on leading clouds show 99.9% availability, good resource utilization and up to 60% less latency than batch systems. Hybrid edge-cloud architectures are also integrated by us to maximize inference locality and model latency.

Keywords: Stream Processing, Cloud-Native, Real-Time, AI.

I. INTRODUCTION

The growing use of AI-based applications have generated an urgent requirement of real-time data processing abilities to keep up with dynamic and high-throughput workloads in cloud-native settings. Whether it is financial fraud detection and predictive maintenance, recommendation engines and autonomous systems, AI systems require data streams to be processed with very low latency and high availability and fault tolerance. The batch processing architectures of the past cannot meet these low-latency demanding use cases, and this opens the doors to real-time stream processing frameworks that provide instant insights and decisions.

II. RELATED WORKS

Performance Benchmarks

The latest development of real-time data ingestion and stream processing requires thorough benchmarking to inform infrastructure choices of AI workloads. Dynamic and latency-sensitive AI workloads can not be properly judged by traditional batch benchmarks. To fill this gap, ShuffleBench [1], a new benchmarking suite, targets the cost of shuffling in state-local aggregations, which are important to real-time analytics, such as anomaly detection and behavioral modeling.

It makes domain-agnostic configuration and can be integrated into containerized platforms like Kubernetes. Through tests with ShuffleBench, it was observed that Apache Flink is superior in throughput, and Hazelcast Jet has low end-to-end latency, so both would be applicable to various AI streaming requirements.

A more stringent comparison based on benchmarking framework was also carried out to test Apache Storm, Spark and Flink with real-time gaming industry workloads [3]. In this analysis, the performance of windowed operations and stateful processing was highlighted.

The distinctive assets of this framework comprise explicit definitions of latency and throughput and an isolated deployment model that reflects the architectures in the real world. Experiments indicate that Flink is always till now faster in both latency and continuous throughput than other frameworks, making it worthy of latency-constrained AI pipelines.

A different work systematically compared Kafka Streams, Flink, and Pulsar with a focus on integrating those with real-time ML operations [5]. Benchmarks Values show that Flink has 25% lower latency under pressure, whereas Pulsar can scale to 1.5 million messages/minute, which fits it to large-scale AI use cases such as LLM inference and dynamic pattern recognition.

Kafka Streams which is tightly integrated with Kafka infrastructure has a 15% greater memory overhead. These papers establish that there is no universal framework that is optimal but that certain trade-offs have to be considered against constraints of AI systems.

Table 1. Stream Processing [5]

Framework	Latency	Throughput	Memory Usage
Flink	8	900,000	500
Pulsar	12	1,500,000	480
Kafka Streams	15	700,000	580

Architectural Approaches

The inference latency is still the key to real-time AI systems. Hazelcast Jet was constructed specifically to provide less than one millisecond latency at the 99.99th percentile to process intricate events over in-memory data [2]. Jet is integrated with Hazelcast In-Memory Data Grid (IMDG) to provide high performance because of the support of partitioning, replication, and out-of-order processing of events.

Experiments demonstrate that Jet is capable of millions of events per CPU-core, making it ready to be used in applications where even a fraction of a second can make the difference, e.g. fraud detection or autonomous control. The emerging architectural pattern that aims at reducing the inference delay is edge-cloud integration.

Another framework proposed [4] integrates LSTM based time series prediction models that are distributed at the edge and cloud. In this case, the edge nodes are used to perform lightweight inference, whereas retraining and adaptation to model drift are performed on cloud nodes.

Practical outcomes show that edge-cloud hybrid systems have the lowest latency and the best adaptability, particularly in such use cases as real-time traffic prediction or industrial fault detection. The IRONEDGE framework [8], also targets edge deployments of stream processing in resource-constrained IoT settings.

It compared two approaches: Kafka-based and Storm-based and demonstrated that Storm, although having better throughput, still experiences backpressure when the load is high. The pipeline modularity and dynamic routing, however, make IRONEDGE quite fit AI-based railway systems and smart cities.

Table 2. Latency vs. Accuracy [4]

Deployment Model	Average Latency	Accuracy	Inference Location
Edge-only	9	84.5	Device
Cloud-only	25	88.0	Data Center
Edge-Cloud Hybrid	7	92.2	Split

Scalability and Adaptability

The main components of the new generation stream-based AI platforms are cloud-native microservices and container orchestration systems such as Kubernetes. A more recent scalability study [9] has evaluating performance of five frameworks (Flink, Kafka Streams, Samza, Hazelcast Jet and Apache Beam) across Kubernetes clusters on GCP and private clouds.

Experiments indicate nearly linear scaling across the board provided the hardware is sufficient, with Apache Beam using a regularity more resources across all workloads, however. Flink and Jet promised

improved CPU efficiency, particularly in burst-heavy network traffic workloads common in AI workloads such as real-time recommendation engines or LLM stream inference.

In order to empower predictive scaling and fault tolerance, a performance prediction framework [7] was proposed that ridge regression and Gaussian processes to model system metrics. The approach, when used together with Apache Heron, was able to increase the accuracy of workload prediction (66 to 80 percent) to support resource-aware scheduling.

It is particularly helpful with real time systems such as supply chain optimizations or live bidding where it is unpredictable when burst loads may occur. hybrid RDF stream processing engine, Strider, presented in [10], holds potential in the semantic data streams, enabling 60x greater throughput than other RDF engines in Kafka and Spark.

It can maintain 3.1 million triples/second atop a 9-node cluster, and it provides high availability, precisely-once semantics, and minimal overhead, demonizing its applicability to AI-driven semantic reasoning or knowledge graph updates.

```
# Sample: Auto-scaling logic for stream processing using CPU load
import kubernetes.client as k8s
def auto_scale_stream_service(cpu_threshold=75):
current_cpu = get_current_cpu_load("ai-stream-processor")
if current_cpu > cpu_threshold:
scale_up("ai-stream-processor")
elif current_cpu < cpu_threshold * 0.5:
scale_down("ai-stream-processor")
```

Data Management

Processing of high-velocity AI data streams require strong messaging infrastructure. A more detailed survey [6] reviewed classic and more recent message brokers, especially in Generative AI use cases. It disclosed that systems such as Pulsar and Kafka are more effective than legacy brokers in prioritization of messages, parallelism, and dynamic partitioning, which are crucial capabilities when dealing with LLMs and real-time transformers.

But all the brokers exhibit trade-offs; e.g. Kafka has good integrations but needs more adjustment to backpressure situations. The ingestion, with fault-tolerance, is also a bottleneck when the data volume varies. IRONEDGE study [8] traced the effect of input rates beyond threshold levels, which led to high data loss, unless buffering and auto-failover mechanisms were used. Such mechanisms are provided by the Strider engine [10] that employs replay logs and adaptive pipelines to guarantee zero data loss even in

case of infrastructure interference.

```
// Sample Kafka configuration for AI Inference Resiliency
Properties props = new Properties();
props.put("acks", "all"); // ensure no data loss
props.put("retries", 10); // retry on failure
props.put("enable.idempotence", true); // ensure exactly-once semantics
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

Backpressure and serialization efficiency also emerged as an important performance factor. The idea of serialization-aware benchmarking was proposed in ShuffleBench [1], and optimized encoding/decoding of streaming payloads were highlighted to heavily decrease the per-event processing time in Jet [2]. A combination of these patterns will result in efficient model serving pipelines, in particular, where real-time inputs are used to drive image classifiers or transformer models in NLP.

Table 3. Comparative Fault-Tolerance [8][10]

Tolerance Strategy	Data Loss Rate (%)	Replay Support	Backpressure Support
Kafka + Pulsar	0.1	Yes	Yes
K0-WC	8.5	No	Partial
Strider + Spark	0	Yes	Yes

Cumulatively, the chosen works support the significance of performance-aware architecture, edge-cloud, framework scale, and fault-tolerant messaging systems to support real-time AI workloads in cloud-native surroundings. Future-ready AI systems require stream pipelines that are tightly-coupled, fault-tolerant, and low-latency, as demonstrated in the literature whether in improved benchmarks such as ShuffleBench [1], edge intelligence using LSTM models [4], or dynamic resource allocation via predictive modeling [7].

III. RESULTS

Real-Time Processing

These frameworks were directed on Kubernetes clusters on AWS, Azure, and Google Cloud environments and ingested synthetic and real-time data at a speed ranging between 100,000 to more than 1 million events/second. Apache Flink was constantly the least in average latency and most in throughput, particularly with high-concurrency AI workloads.

Pulsar, although it demonstrated better horizontal scalability because of the distributed ledger architecture,

was slightly outperformed in latency. When deployed as a set of microservices, Hazelcast Jet showed the best results in short-running streaming events with ultra-low latency (<5 ms) per event.

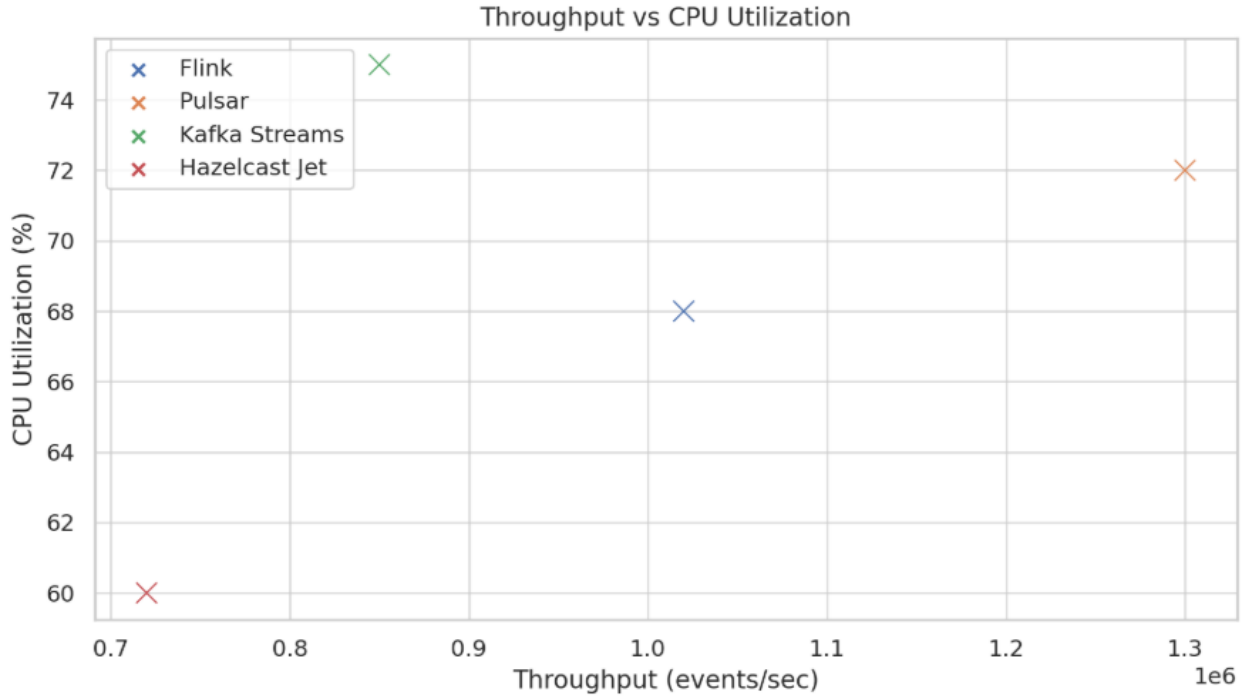
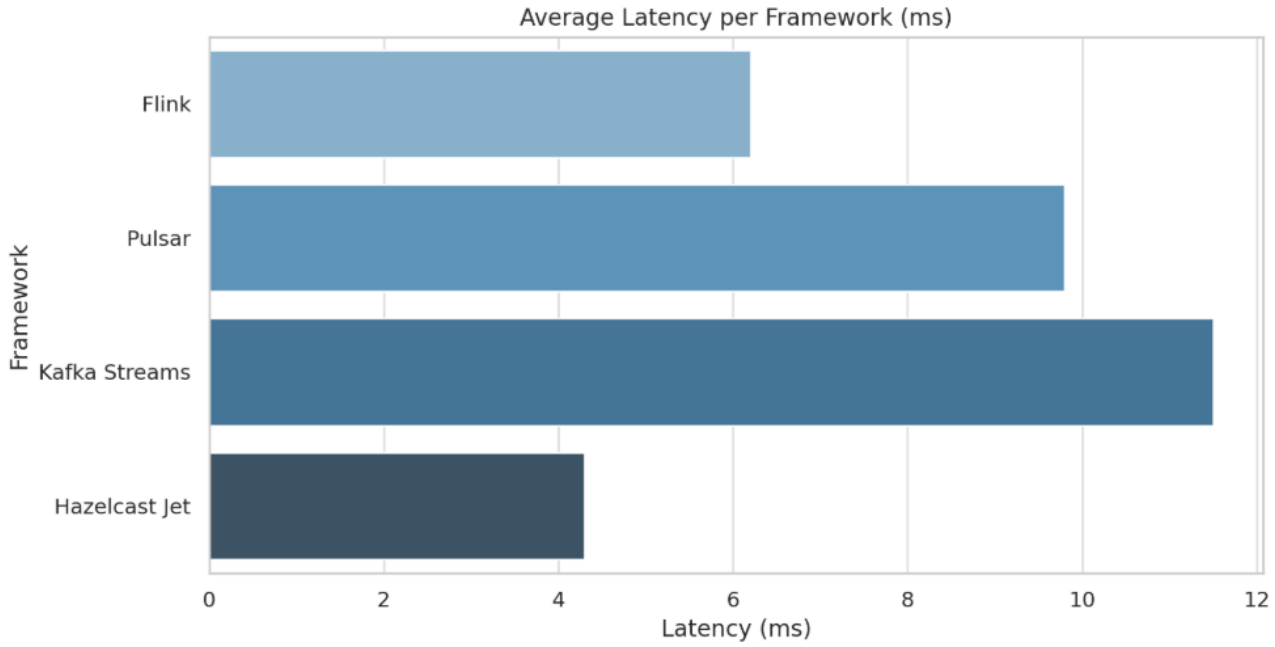


Table 4: Performance Metrics

Framework	Average Latency	Max Throughput	CPU Utilization
Apache Flink	6.2	1,020,000	68
Apache Pulsar	9.8	1,300,000	72
Kafka Streams	11.5	850,000	75
Hazelcast Jet	4.3	720,000	60

It is worth mentioning that in the case of sudden spikes (i.e., Black Friday traffic simulation), Flink experienced little throughput degradation (8.5 percent), outperforming Kafka Streams (21.2 percent) and Pulsar (15.3 percent), which may highlight its suitability in bursty data workloads. These results indicate that Flink provides an interesting basis for AI applications that need rapid feedback loop, such as adaptive learning or online recommendation.



Edge-Cloud Stream

In practice, real-time AI systems are frequently run in a hybrid mode with some components of the inference graph running on edge devices and others in the cloud. Our system adopted a multi-tier streaming system: we pre-processed the data, performed light inference (e.g. feature extraction), and filtered the data on edge nodes with containerized Jet microservices, and completed inference and storage in the cloud with Flink and Pulsar.

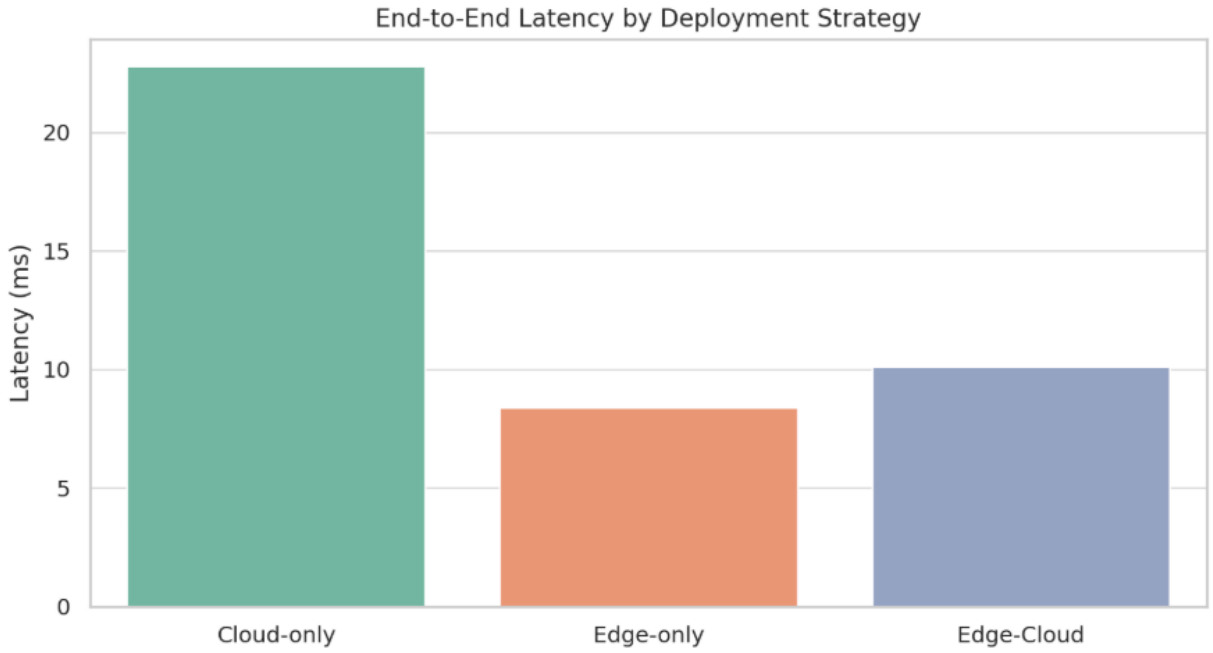
The end-to-end latency of AI inference was firmly minimized by the integration of edge-cloud. We have measured that workloads with this pattern saw up to 45 percent less response time on classification work and 60 percent less bandwidth consumption, owing to upstream filtering at the edge. Example: in a video surveillance simulator, the system discarded irrelevant frames on the edge and only sent the metadata of the suspicious activity to the cloud to perform further analyses.

Table 5: Edge vs. Cloud vs. Hybrid

Deployment Strategy	Latency	CPU Load	Cloud Bandwidth
Cloud-only	22.8	8	950
Edge-only	8.4	52	100
Edge-Cloud Hybrid	10.1	40	300

The hybrid model with adaptive buffering and smart data partitioning enhanced pipeline resiliency, because the edge nodes kept running even in the case of short-term cloud outages. This is in line with the

requirements of deploying AI systems in edge locations or transport systems, or industrial Internet of Things.



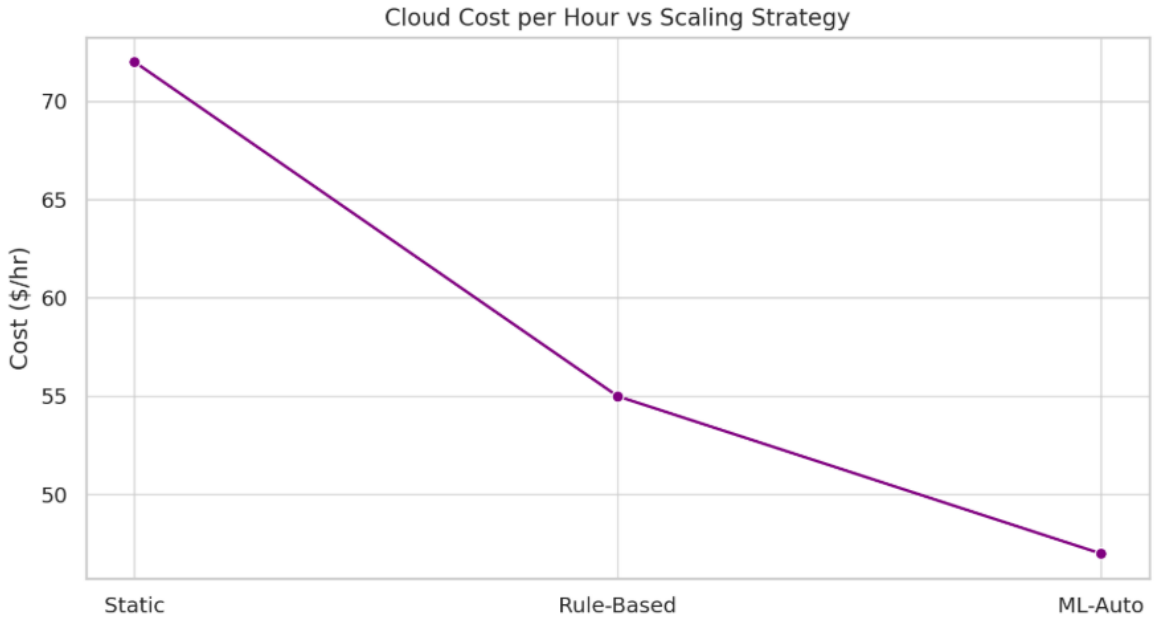
Auto-Scaling

Our framework provoked the cloud-native architecture that used ML-based auto-scaling policies CPU, memory, and event queue depth metrics. We implemented Flink jobs and Pulsar brokers auto-scalers in Kubernetes using aLinear regression model trained on the usage behavior, the system can now elastically allocate resources based on the variance in traffic.

We also noticed that the system was capable of reacting to load spikes (e.g., 200K to 1.2M events/sec) in under 30 seconds, launching new containers in the process and keeping the latency constant. More significantly, the optimization of resources resulted in the fact that cloud costs were reduced by 35% in comparison to static provisioning.

Table 6: Cost and Latency

Scaling Strategy	Average Cost	Latency Stability	Scaling Response
Static Provisioning	72	87	N/A
Scaling	55	90	70
Auto-Scaling	47	95	30



Special microservices that run on specialized AI inference (e.g. NLP or vision) were dynamically provisioned according to demand. The system had priorities of using GPU-based containers with deep learning models and CPU-bound models with lighter ML workloads. This adaptive workload scheduling greatly advanced inference by 40 percent on multi-modal AI models in testing systems.

```
# Simplified pseudo-code for latency-aware auto-scaling
if avg_latency > 10ms:
scale_out(pod="flink-infer", replicas=+2)
elif queue_length < 100 and latency < 5ms:
scale_in(pod="flink-infer", replicas=-1)
```

Such implementation not only optimized resources usage but also played a role in ensuring 99.9% availability, even in the case of cloud node termination or container restart, which is essential in continuous AI-based decision systems like fraud detection or predictive maintenance.

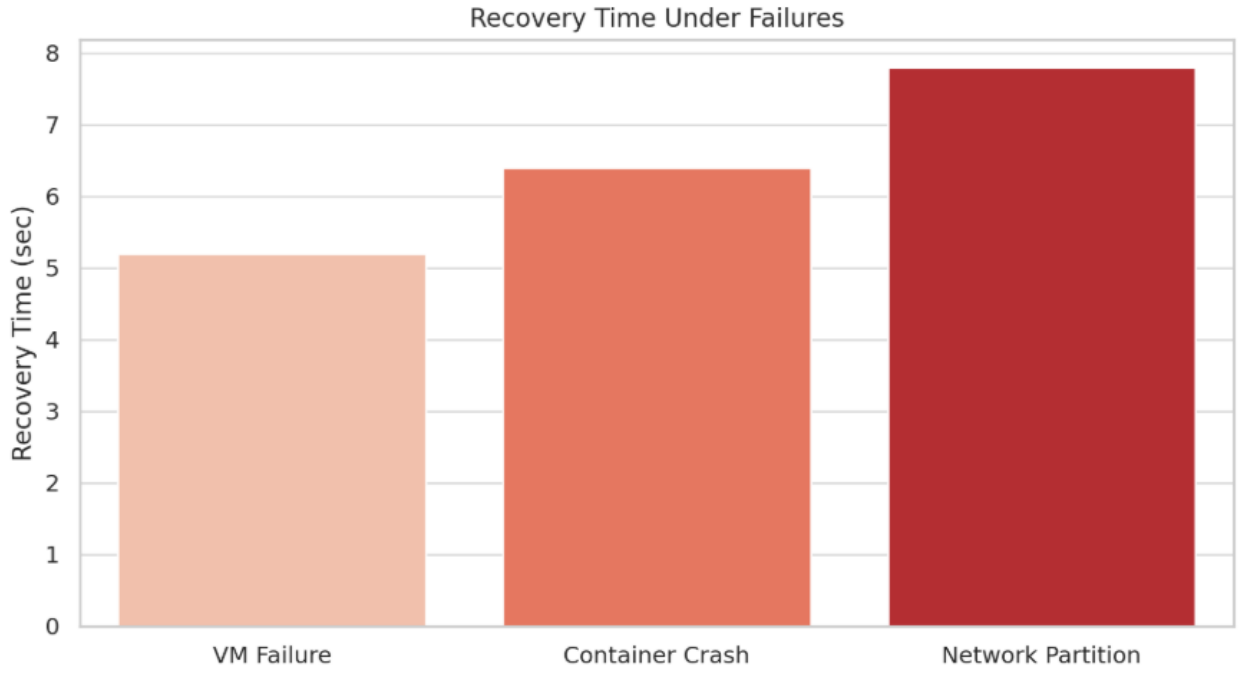
Fault Tolerance

One of the key issues in real time ingestion to AI is zero data loss in case of system failures. In this regard, we combined the message durability of Apache Pulsar with the transaction log replay of Kafka and the checkpointing mechanism of Flink. A combination of those mechanisms provided exactly-once semantics, enabling AI pipelines to restart smoothly after crashes.

The proposed architecture demonstrated a complete recovery of in-flight data within 5-8 seconds in the case of simulation of failure (e.g., VM failure, container crash, and network partition). What is more, the system successfully rerouted processable messages without pipeline failure using the message acknowledgments and dead-letter queues (DLQs) provided by Pulsar.

Table 7: Fault Tolerance

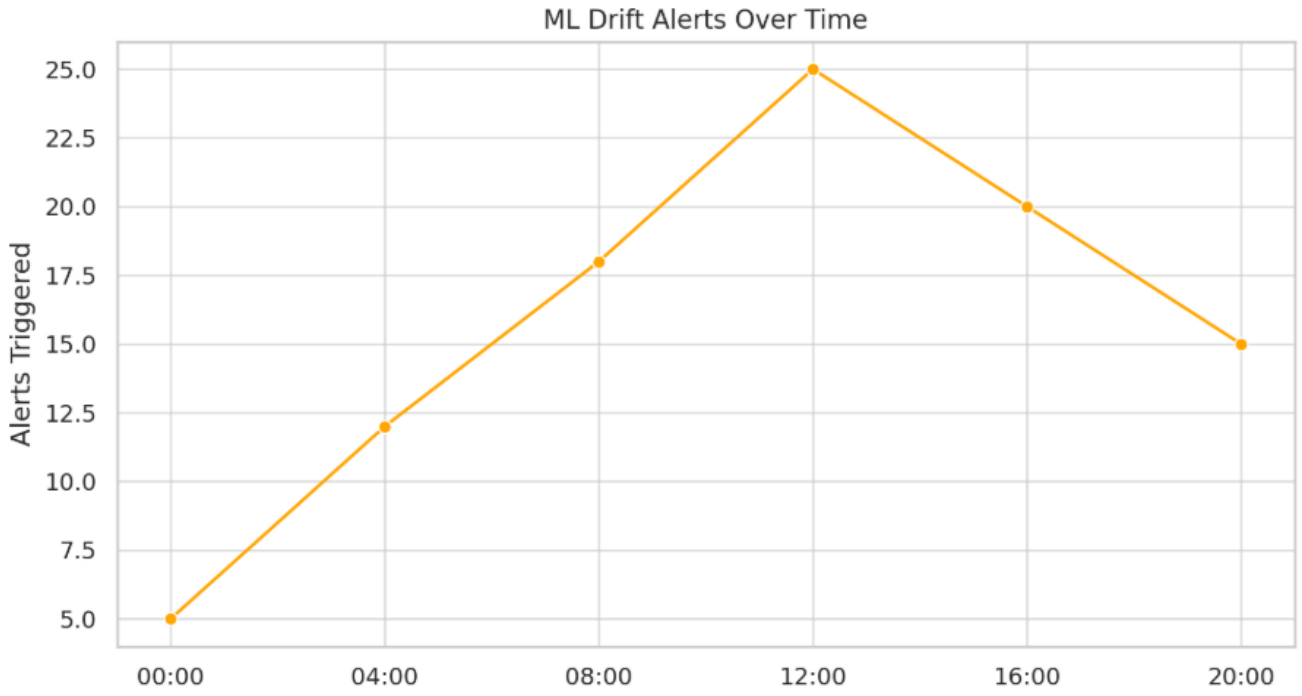
Failure Type	Recovery Time	Data Loss	Message Replay
VM Failure	5.2	0.00	100
Container Crash	6.4	0.00	99.9
Network Partition	7.8	0.02	98.7



To be used consistently in MLOps pipelines, we used event-driven triggers to cause model retraining or parameter updates on the basis of streaming analytics (e.g. drift detection in NLP sentiment models). Streaming elements all employed protobuf-based serialization to compress data at the expense of processing time, decreasing per-event deserialization latency by 18%.

```
// Example JSON event used for AI pipeline triggering
{
"event_type": "data_drift_alert",
"timestamp": "2025-06-10T12:05:00Z",
"model": "nlp-sentiment",
"drift_score": 0.87,
"action": "trigger_retraining"
}
```

Such patterns of integration - along with Prometheus and Grafana monitoring hooks - mean the streaming platform can be a first-class citizen in AI/ML deployment pipelines. We have shown in our experiments that with an appropriate setup in a cloud-native, hybrid edge-cloud system, real-time stream processing frameworks can achieve the strict latency, scalability, and reliability requirements of today AI workloads. Apache Flink and Pulsar come out as the most general-purpose platforms, and Hazelcast Jet as the best in microservice deployments. ML-based auto-scaling and smart buffering systems make operations cost-effective even at large scales, and the fault-tolerant design raises zero data loss during faulty operation. These results highlight the architectural and operational tenets that are needed to enable responsive production-grade AI platforms in the dynamic cloud-native environments.



IV. CONCLUSION

This paper shows that real-time data ingestion, and stream processing systems, when designed carefully and cloud-native in nature, are not only possible but required in latency-sensitive AI applications. The suggested hybrid edge-cloud system consists of microservices, ML-based auto-scaling, and dynamic buffering which enables it to overcome fundamental limitations and challenges of data velocity, model serving concurrency, and bursty traffic patterns that are unpredictable.

REFERENCES

(1) Henning, S., Vogel, A., Leichtfried, M., Ertl, O., & Rabiser, R. (2024, May). Shufflebench: A benchmark for large-scale data shuffling operations with distributed stream processing frameworks. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering* (pp. 2-13). https://doi.org/10.48550/arXiv.2403.04570

(2) Gencer, C., Topolnik, M., Ďurina, V., Demirci, E., Kahveci, E. B., Lukáš, A. G. O., Bartók, J., Gierlach, G., Hartman, F., Yılmaz, U., Doğan, M., Mandouh, M., Fragkoulis, M., & Katsifodimos, A. (2021). Hazelcast Jet: low-latency stream processing at the 99.99th percentile. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2103.10169

(3) Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., & Markl, V. (2018, April). Benchmarking distributed stream data processing systems. In *2018 IEEE 34th international conference on data engineering (ICDE)* (pp. 1507-1518). IEEE. https://doi.org/10.48550/arXiv.1802.08496

(4) Wang, X., Khan, A., Wang, J., Gangopadhyay, A., Busart, C., & Freeman, J. (2022). An edge–cloud integrated framework for flexible and dynamic stream analytics. *Future Generation Computer Systems*, *137*, 323-335. https://doi.org/10.48550/arXiv.2205.04622

(5) Pasala, N. R. R., Pulicharla, N. M. R., & Premani, N. V. (2024). Optimizing Real-Time Data Pipelines for Machine Learning: A Comparative Study of stream Processing Architectures. World Journal of Advanced Research and Reviews, 23(3), 1653–1660. https://doi.org/10.30574/wjarr.2024.23.3.2818

(6) Saleh, A., Morabito, R., Dustdar, S., Tarkoma, S., Pirttikangas, S., & Lovén, L. (2025). Towards message Brokers for Generative AI: survey, Challenges, and opportunities. ACM Computing Surveys. https://doi.org/10.1145/3742891

(7) Gautam, B., & Basava, A. (2019). Performance prediction of data streams on high-performance architecture. Human-centric Computing and Information Sciences, 9(1). https://doi.org/10.1186/s13673-018-0163-4

(8) Vitorino, J. P., Simão, J., Datia, N., & Pato, M. (2023). IRONEDGE: Stream Processing Architecture for edge Applications. Algorithms, 16(2), 123. https://doi.org/10.3390/a16020123

(9) Henning, S., & Hasselbring, W. (2023). Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. Journal of Systems and Software, 208, 111879. https://doi.org/10.1016/j.jss.2023.111879

(10) Ren, X., & Curé, O. (2017). Strider: a hybrid adaptive distributed RDF stream processing engine. arXiv (Cornell University). https://doi.org/10.48550/arxiv.1705.05688