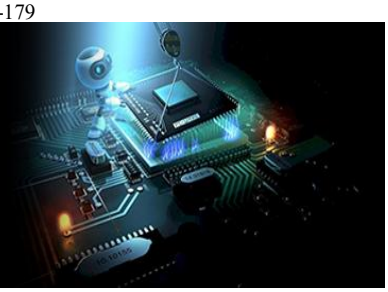
International Journal of Engineering in Computer Science



E-ISSN: 2663-3590
P-ISSN: 2663-3582
Impact Factor (RJIF): 5.52
www.computersciencejournals.com/ijecs
IJECS 2025; 7(2): 176-179
Received: 24-07-2025
Accepted: 27-08-2025

Bogutskii Aleksandr
Bachelor's degree, ITMO University, St Petersburg, Russia

Corresponding Author:
Bogutskii Aleksandr
Bachelor's degree, ITMO University, St Petersburg, Russia

Engineering approaches to building high-performance real-time services: The case of antifraud services

Bogutskii Aleksandr

DOI: https://doi.org/10.33545/26633582.2025.v7.i2b.214

Abstract
In this article, engineering approaches to building high-performance real-time services are considered using the example of antifraud services. It examines the experience of creating distributed high-load solutions applied in industry for processing billions of events in real time with minimal latency. Engineering solutions aimed at detecting suspicious activities are investigated, including data filtering, cleansing, deduplication, and normalization, as well as information aggregation using window functions, hierarchical aggregation, and approximate computing. Architectural patterns and practices for implementing correct processing semantics are described. Approaches to ensuring system stability and performance are also examined.

Keywords: Stream processing, distributed systems, anti-fraud, information aggregation, high-load services, Apache Kafka, auto-scaling

Introduction

Present-day digital services generate and process vast volumes of events in real time. These may include data from online platforms, sensor networks, financial systems, or advertising services, where even minor processing delays can significantly affect user experience and analytical accuracy. The growing scale of such systems are often coupled with the increasing intensity of data streams. They create a demand for new engineering solutions that can set the seal on stable operation under high load.

Conventional approaches to information processing are based on batch computation and are unable to meet these specifications. Their high latency and limited scalability frequently render them unsuitable for tasks involving even millions of events per second. In response to this, architectures oriented exclusively toward stream processing have become apparent, since they enable reduced latency and greater resilience in distributed environments.

Alongside architectural advances, engineering practices for system optimization have also evolved. Modern real-time services require right mechanisms for fraud detection. Equally important are stability factors such as fault tolerance, load control, and predictable behavior under traffic spikes. These challenges have driven the rapid development of streaming analytics technologies and the associated engineering practices in recent years. The purpose of this article is to analyze the engineering approaches applied in the design of high-performance real-time services, with particular emphasis on antifraud services.

Main part. Engineering solutions for antifraud services under high load

Preventing fraud and summarizing user activity are crucial for any high-loaded service with a lot of customers. It is at these stages that decisions are made regarding which data will proceed to subsequent processing and how metrics and analytics will be generated. Well-structured preprocessing can reduce the load on computational nodes several-fold while simultaneously improving the accuracy of final results [1].

The first step of the antifraud process involves filtering and normalization of the incoming stream. This includes removing duplicate events, for example, repeated clicks or sensor signals, in order to prevent distortions in subsequent aggregation. To address this task, systems employ schemes for message identification based on unique keys or combinations of timestamps and hashes, as well as windowed deduplication, which is implemented in modern stream processing engines such as Apache Flink or Apache Beam.

A second key element of anti-fraud is stream cleansing from noise and anomalies. Invalid messages, such as telemetry with corrupted values, are discarded at an early stage. For this purpose, validation rules and lightweight anomaly detection algorithms are applied directly within the streaming pipelines. An equally important component is data format normalization [2]. Experimental studies confirm the effectiveness of this approach. For instance, the SPinDP (Stream Purifier in Distributed Platform) study demonstrated that the introduction of a distributed cleansing layer based on Apache Storm and Kafka, leveraging an RDMA network, increased system throughput by more than 28 times and reduced average processing latency by 2473 times compared to a non-optimized system [3].

Event aggregation holds a central place in antifraud services. In real-time environments, aggregation is typically implemented through windowing functions, which group data into temporal intervals. This enables the computation of metrics such as the number of events occurring within a given period. For more complex scenarios, Complex Event Processing (CEP) is employed to detect patterns in the stream, for example, identifying a sequence of three transaction failures within a ten-second interval.

A hierarchical approach to aggregation has proven effective. Data are first aggregated locally at edge nodes or shards, and then combined into global aggregates. The choice of a specific streaming platform directly determines how antifraud service is implemented.The most commonly adopted solutions include Apache Kafka, AWS Kinesis, and Google Cloud Pub/Sub, each of which demonstrates different approaches to scalability, delivery guarantees, and event ordering (table 1).

Table 1: Event processing platform performance characteristics [4, 5]

Platform feature	Apache Kafka	AWS Kinesis	YTSaurus dynamic tables
Maximum throughput	1M+ messages/second	1MB/shard/second input, 2MB/shard/second output	1M+ messages/second
Latency	Sub-10ms	Milliseconds range	Milliseconds range
Scaling model	Horizontal partitioning	Sharding	Horizontal partitioning
Delivery guarantee	At-least-once	Exactly-once	Exactly-once
Event ordering	Strict (within partition)	Per shard	Strict (within partition)
Processing model	Pull-based	Pull-based	Pull-based

In practice, a variety of architectural patterns are employed for event aggregation. For a long time, the Lambda architecture remained a popular choice. However, it has largely been supplanted by the Kappa architecture, which relies exclusively onstream processing and log replay from Kafka. Another widely adopted pattern is the combination of CQRS (Command Query Responsibility Segregation) and Event Sourcing, where all state changes are recorded as a sequence of events. Aggregates are also built by subscribing to this event stream.

Special attention should be given to the question of correctness and processing semantics. Under heavy load, it is critical for a system to adhere to consistent guarantees. In practice, two primary semantics are distinguished: at-least-once, which is simple and inexpensive but may result in event duplication, and exactly-once, which is more costly but indispensable in financial and advertising services. Flink and Beam implement the latter through checkpoints and transactional sinks. An important engineering practice in this context is the design of idempotent operations, where repeated processing of the same event does not alter the final outcome. Current problems emphasize the necessity of efficient aggregation algorithms. Increasingly, methods of approximate computing, such as HyperLogLog or Count-Min Sketch, are being adopted.

Practice of building high-load real-time event processing services

Modern event-driven architectures and streaming platforms provide high scalability, low-latency capabilities, and system resilience. A prominent technological example of industrial real-time stream processing can be found in Uber's experience. The company has undertaken extensive work to deploy Apache Flink in combination with Apache Kafka and Apache Pinot, building efficient and reliable event-processing pipelines.

Uber leverages Flink for near-real-time advertising data processing: from ingesting events (impressions, clicks), through cleansing, aggregation, and order attribution, to the delivery of summary data for reporting and analytics. The system guarantees exactly-once processing semantics and high reliability, which is critical in the context of advertising events tied to financial transactions. In addition, to further improve resilience and manageability of data streams, Uber developed a Consumer Proxy layer integrated with Kafka (fig. 1).

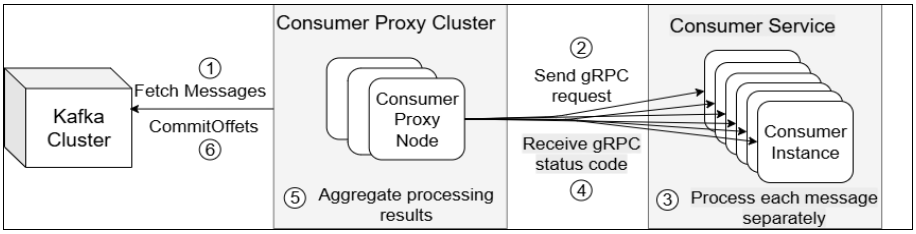


Fig 1: High level consumer proxy architecture [6]

Uber further continues to develop and maintain Flink as the foundation of its streaming infrastructure for petabyte-scale data processing. Main engineering enhancements include the introduction of the FlinkSQL layer, that simplifies the expression of analytics in SQL, as well as mechanisms both for resource estimation and automatic scaling All of them guarantee stable and scalable execution of streaming workloads. One of the core components of Uber's

ecosystem has also become the auto-scaling of Kafka consumers (fig. 2).

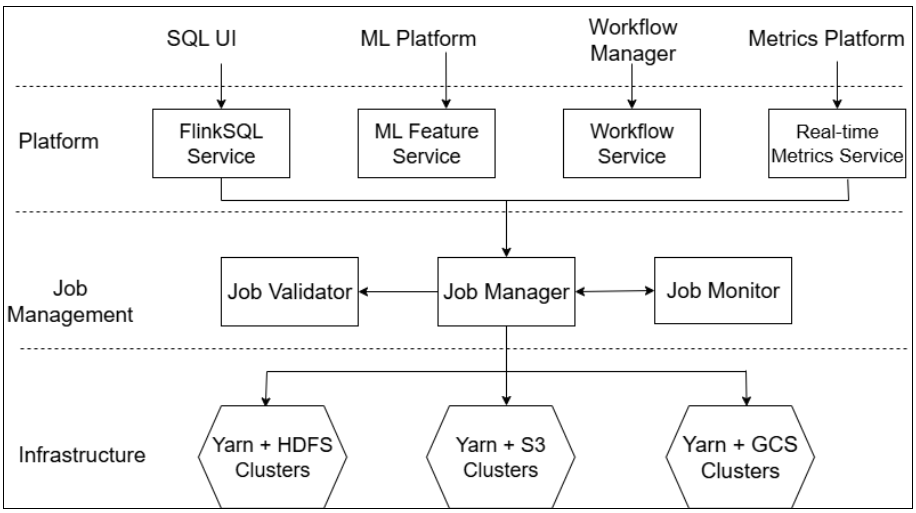


Fig 2: The layers of the Unified Flink architecture at Uber [7]

LinkedIn developed Apache Samza as a solution for stateful stream processing tightly integrated with Kafka. While Samza laid the groundwork for fault-tolerant architectures, there has been a shift towards more universal frameworks such as Apache Beam. This service provides a unified programming model across multiple runners and has been adopted by organizations including Google and Spotify for large-scale real-time analytics pipelines. The company processes millions of events per second on individual tasks, utilizing tens of terabytes of local state [8].

Netflix has built a real-time streaming infrastructure that processed 1 trillion events per day in 2017, scaling this volume twenty-fold by 2021. Its foundation consists of managed versions of Kafka and Flink, the in-house platform Mantis, and data at petabyte scale. This architecture supports both analytical and operational streaming scenarios, delivering low latency, resilience, and organization-wide scalability. As highlighted in official resources, Netflix also employs single-stage Samza jobs to route more than 700 billion events and approximately 1 PB of data per day from fronting Kafka clusters into S3/Hive [9].

The experience of these leading companies illustrates that the successful adoption of real-time stream processing depends on a combination of carefully designed architectural strategies and specialized engineering practices. The cases of Uber, LinkedIn, and Netflix demonstrate that such approaches enable the processing of data at the scale of trillions of events while maintaining resilience and low latency.

Approaches to ensuring stability and performance in high-load systems

Even a well-designed streaming architecture requires dedicated measures to maintain stability and high performance under increasing load. One of the most critical strategies is the design of systems with resilience to failures and overloads. In practice, this is achieved through component isolation and the introduction of protective mechanisms. For example, in microservice-based distributed systems, the Circuit Breaker pattern is commonly employed to prevent cascading failures: when a service becomes overloaded or fails, the automatic "breaker" interrupts the chain of calls, thereby preventing the malfunction from propagating throughout the entire system [10].

Another fundamental principle is load control. Various systems must be able to regulate the rate of incoming requests or events in accordance with their current processing capacity. In case, when the incoming flow exceeds this threshold, mechanisms such as backpressure or input-level load shedding can be used to ward off queue overflow and performance deterioration. Without such shields, IT structures risk entering an unstable operational regime. Research has identified a class of phenomena known as metastable failures, in which surpassing a critical load threshold triggers a self-reinforcing performance collapse [11]. The system falls into a "stall" characterized by runaway queue growth and diminishing efficiency, from which recovery is impossible without external intervention.

Apart from the components of architectural design, the operations procedure is instrumental in ensuring stability. The application of real-time observability and monitoring systems allows immediate identification of anomalies. Accurately calibrated metrics and alerting allow engineers to detect rising latencies or excessive resource consumption before such issues become serious enough to have the potential of causing a total system collapse. Logging and metric storage solutions like Prometheus, Grafana, and ELK stack, along with distributed request tracing, allow deep visibility into activity of a system.

Regular load testing and profiling under conditions approximating real peak traffic are equally necessary. Another key strategy is automatic resource scaling under load. Modern cloud platforms natively support auto-scaling, where the number of service instances dynamically expands or contracts in response to the volume of incoming events (fig. 3).

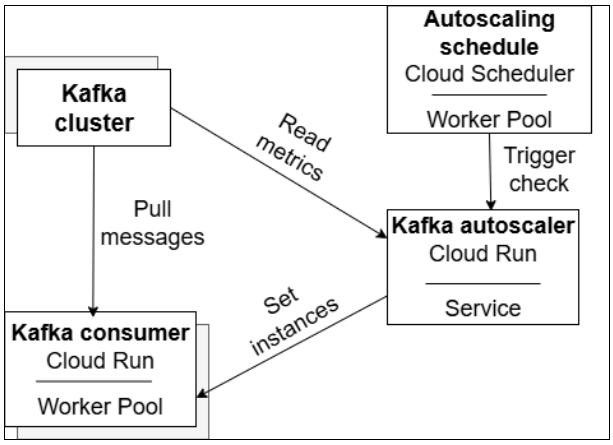


Fig 3: Kafka autoscaling architecture [12]

Optimally tuned auto-scaling mechanisms are instrumental in providing steady service, especially during peak traffic, while at the same time avoiding service degradation and wasteful utilization of resources during off-peak traffic. Additionally, one needs to ensure the integrity of storage of data and the effectiveness of communication mechanisms. Utilization of replicated log-based queuing mechanisms guarantees the persistence and security of messages even with failures of individual nodes.

Taken together, these practices create the operational ground for keeping the stability and performance of large-scale systems. When applied systematically, they provide platforms to remain predictable and resilient, securing uninterrupted service quality even under sudden traffic surges or partial component failures.

Conclusion

The development of real-time distributed systems demonstrates that efficiency and resilience are largely determined by the quality of engineering decisions at the stages of anti-fraud. The application of filtering, cleansing, and deduplication reduces redundant load and improves data reliability, while the use of window functions, hierarchical aggregation, and approximate computing enables a balance between performance and accuracy.

Stability of a system is achieved through the adoption of architectural patterns, load management strategies, and also correct processing semantics. The integration of fault-tolerance mechanisms, monitoring, and automatic scaling ensures that streaming services remain predictable and resilient under heavy load. Industrial experience confirms that the combination of architectural strategies and optimization practices makes it possible to design real-time, high-load services that meet modern requirements for scalability and data processing quality.

References

1. Makhtibekov A. Effectiveness of multichannel marketing in the context of digital transformation. Universum: economics and law: electronic scientific journal. 2025;4(126):15–19.
2. Dudak A. Virtualization and rendering of large data lists. Cold Science. 2024;9:17–25. EDN: QUGQBQ.
3. Gil MS, Moon YS. SPinDP: A High-Speed Distributed Processing Platform for Sampling and Filtering Data Streams. Appl Sci. 2023;13(24):12998. doi:10.3390/app132412998. EDN: NDFWZE.
4. Choudhary SK. Implementing Event-Driven Architecture for Real-Time Data Integration in Cloud Environments. Int J Comput Eng Technol. 2025;16(1):1535–52. doi:10.34218/ijcet_16_01_113. EDN: PNXBGY.
5. Garifullin R. Integration of WebAssembly for high-performance web applications. Int J Latest Eng Manag Res. 2025;10(3):28–31.
6. Uber. Enabling Seamless Kafka Async Queuing with Consumer Proxy [Internet]. Available from: https://www.uber.com/blog/kafka-async-queuing-with-consumer-proxy/ [cited 2025 Sep 14].
7. Fu Y, Soman C. Real-time data infrastructure at Uber. In: Proceedings of the 2021 International Conference on Management of Data. 2021. p. 2503–16. doi:10.1145/3448016.3457552.
8. Apache Beam. Revolutionizing Real-Time Stream Processing: 4 Trillion Events Daily at LinkedIn [Internet]. Available from: https://beam.apache.org/case-studies/linkedin/ [cited 2025 Sep 15].
9. Apache Samza. Powered By [Internet]. Available from: https://samza.apache.org/powered-by/ [cited 2025 Sep 16].
10. Argakoesoemah MD, Candra MZ. Development of Cascading Circuit Breaker System Using Event-Driven Approach in Microservices. In: 2024 IEEE International Conference on Data and Software Engineering (ICoDSE). 2024. p. 102–7. doi:10.1109/ICoDSE63307.2024.10829897.
11. Bronson N, Aghayev A, Charapko A, Zhu T. Metastable failures in distributed systems. In: Proceedings of the Workshop on Hot Topics in Operating Systems. 2021. p. 221–7. doi:10.1145/3458336.3465286.
12. Google Cloud. Autoscale your Kafka consumer workloads [Internet]. Available from: https://cloud.google.com/run/docs/configuring/workerpools/kafka-autoscaler [cited 2025 Sep 22].