The American Journal of Engineering and Technology
ISSN 2689-0984 Volume 08 - 2026

Architectural Strategies for Reprocessing Historical Data in Real-Time Systems

[1] Rybchanka Aliaksandr
[1] Senior Full-stack Software Engineer Amsterdam, The Netherlands

Received: 18th Nov 2025 | Received Revised Version: 28th Nov 2025 | Accepted: 12th Jan 2026 | Published: 27th Jan 2026

Volume 08 Issue 01 2025 | Crossref DOI: 10.37547/tajet/Volume08Issue01-15

Abstract

This study examines architectural strategies for reprocessing historical data in real-time systems built around the Kappa architecture and Apache Kafka–based microservices. The research addresses the growing need to recompute derived state, machine-learning features and aggregates without interrupting continuous processing or violating correctness guarantees. The work systematises approaches to "time-travel" over event logs, including full-topic replay, snapshot-plus-log reconstruction and isolated backfill pipelines. Special attention is given to the interaction between Kafka, stateful stream processors such as Apache Flink and Kafka Streams, and microservice-oriented designs that rely on local or external state stores. The goal is to formulate practical design guidelines for architecting reprocessing workflows under strict latency, availability and consistency requirements. The article presents an analytical comparison of modern stream-processing platforms and real-world case studies from the financial and fraud detection domains. In conclusion, the study formulates recommendations on choosing between local and external state, structuring replay traffic, and integrating reprocessing pipelines into production Kappa-style systems without global downtime.

Keywords: Kappa architecture, Apache Kafka, real-time stream processing, historical data reprocessing, time-travel replay, Kafka Streams, Apache Flink, stateful microservices, event sourcing, data streaming architecture.

Cite This Article: Aliaksandr, R. (2026). Architectural Strategies for Reprocessing Historical Data in Real-Time Systems. The American Journal of Engineering and Technology, 8(01), 136–144. https://doi.org/10.37547/tajet/Volume08Issue01-15

1. Introduction

Real-time products increasingly depend on derived views built from continuous event streams, such as fraud-scoring features, user behaviour aggregates, materialised timelines, alerts, and monitoring indicators. These views evolve over years of production activity and rest on assumptions encoded in stream-processing topologies. As schemas, business rules and machine-learning models change, teams face recurring pressure to recompute long histories of events while keeping real-time flows responsive. Conventional batch reprocessing over static snapshots conflicts with event-driven architectures that use Kafka as a durable log and favour Kappa-style designs where both historical and live traffic pass through the same streaming pipeline.

Historical reprocessing in such environments presents several practical constraints. Event logs often span multiple years and petabyte-scale topic histories. Systems must respect event-time semantics, ordering within partitions and exactly-once guarantees during replay. Production microservices rely on local state stores or external databases that already encode years of

operational behaviour. Restarting entire clusters or duplicating infrastructure is rarely a viable option. At the same time, regulatory requirements, backfills for new machine-learning features, and corrections of logic errors demand predictable strategies for recomputing historical states.

The goal of this article is to describe architectural strategies that enable the reprocessing of historical data in real-time systems built on Kafka and Kappa architectures, while preserving availability and correctness. The first task is to analyse how stateful stream-processing engines and Kafka-centric microservice architectures support replay, snapshot-based reconstruction and hybrid approaches. The second task is to compare local state stores embedded in stream processors with external operational or analytical databases in the context of large-scale reprocessing and long retention. The third task is to formulate design recommendations on partitioning, topic topology, isolation of backfill pipelines and coordination with downstream services, tailored to organisations that already operate Kafka-centric microservice landscapes.

The novelty of the work lies in its focus on "time-travel" and large-scale replay in Kappa-oriented systems. Existing publications describe performance characteristics of Kafka, Flink and Kafka Streams, or present domain-specific pipelines in finance and fraud detection. The present study consolidates that knowledge into an architecture-level view centred on replay and reconstruction, connecting research on state management, microservices and real-time streaming into a coherent set of patterns directly applicable to modern Kafka-based infrastructures.

2. Materials and Methods

The study relies on recent publications that examine Kafka-centric streaming, microservice architectures and state management in modern stream-processing engines. A. Bozkurt, with co-authors, analyses joint use of Apache Flink and Apache Kafka in a real-time case study, focusing on integration patterns, low-latency event handling and performance characteristics of a combined pipeline [1]. R. S. Dev and J. Usha present a unified fraud-detection pipeline that integrates Kafka, ksqlDB and Apache Flink, describing end-to-end real-time processing of transactional streams in a production-like environment [2]. Y. Mei and collaborators introduce disaggregated state management in Apache Flink 2.0, detailing remote state storage, asynchronous execution and implications for checkpointing, recovery and rescaling of large stateful jobs [3]. S. Pamarthi examines adoption of Kafka and Flink in financial services, outlining streaming architectures for fraud detection, risk monitoring and customer analytics built on an event-driven backbone [4]. I. Pelle and co-authors conduct a quantitative performance analysis of Kafka and Kafka Streams in cloud-native deployments, providing measurements of latency, throughput and resource footprint for stream-processing workloads relevant to IoT and similar scenarios [5]. S. M. Podduturi investigates real-time data processing in microservices architectures, with emphasis on event-driven communication, patterns such as CQRS and event sourcing, and the impact of streaming technologies on scalability and reliability [6]. S. Saket and colleagues describe migration from batch processing to a Kafka- and Flink-based streaming pipeline for real-time event joining in a machine-learning context, addressing challenges around event-time semantics, checkpointing and scalability [7]. V. K. Tambi studies real-time data stream processing using Kafka-driven AI pipelines, illustrating how Kafka topics feed machine-learning models and how model outputs integrate back into streaming systems [8]. B. Tanneru explores the use of Kafka messaging in microservices for real-time data processing, focusing on communication patterns, decoupling of services and operational concerns [9]. G. Wang and an extended team rethink distributed stream processing in Apache Kafka, defining consistency and completeness guarantees and elaborating on exactly-once semantics in Kafka Streams-based applications [10].

The article employs an analytical methodology that synthesises the conceptual framework of the reviewed works. Comparative analysis is used to contrast local versus external state management in the context of replay, taking into account performance measurements, fault-tolerance mechanisms, and operational practices discussed in the literature [3, 5, 10]. The structural analysis of streaming pipelines, as described in case studies from payments, fraud detection, and financial analytics, supports the generalisation of architectural patterns for historical reprocessing [1, 2, 4, 7, 8]. The study further applies typological classification to identify families of replay strategies—full log replay, snapshot-plus-log reconstruction and isolated backfill pipelines—and to relate them to microservice patterns such as event sourcing and CQRS [6,9]. The resulting recommendations emerge from correlating these patterns

with properties of Kappa architecture and Kafka-centric microservice ecosystems rather than from experimental benchmarking.

3. Results

Architectural strategies for reprocessing historical data in real-time systems must respect the properties of Kafka as an append-only log, the semantics of stateful stream processors and the operational constraints of microservice environments. The reviewed literature converges on a picture in which Kafka topics, stream processors and state stores form an integrated substrate for both online processing and replay.

Works that combine Kafka and Flink illustrate how a unified event log underpins both live and recomputation workloads. In the case study by A. Bozkurt and co-authors, Kafka serves as the backbone for ingestion. At the same time, Flink performs low-latency event processing, with business-specific transformations implemented as stream topologies over Kafka topics [1]. Dev and Usha build a similar structure in the fraud-detection pipeline: Kafka captures transactional events, ksqlDB expresses immediate business rules, and Flink implements deeper analytics with complex event processing and machine-learning models [2]. Pamarthi extends this pattern to multiple financial use cases, such as fraud detection and risk monitoring, where Kafka and Flink jointly provide end-to-end streaming for mission-critical workloads [4]. In all three cases, replaying historical data is conceptually achieved by feeding past events back into the same Kafka topics and letting the streaming jobs recompute the derived state, which directly aligns with Kappa architecture principles.

A more detailed view of stateful processing emerges from research on Flink and Kafka Streams. Mei and co-authors describe how Flink 2.0 decouples computation from state by storing operator state in a remote distributed file system, with local disk used as cache [3]. Continuous streaming of state updates into the remote store, combined with an asynchronous execution model, leads to faster checkpointing and recovery. These properties directly influence replay strategies: large historical reprocessing runs benefit from disaggregated state, as checkpoints remain fast and jobs can be rescaled or migrated during long backfills without incurring costly state transfers [3]. Pelle and colleagues evaluate Kafka and Kafka Streams in cloud-native deployments, showing that the combined system sustains sub-second latency and predictable scaling under different workloads, while explicitly highlighting the resource footprint of Kafka Streams' local RocksDB-backed state [5]. Wang and co-authors detail the correctness guarantees of Kafka Streams, including exactly-once semantics in the presence of failures and out-of-order data [10]. For historical reprocessing, this means that once input topics and changelog topics are stored durably, it is possible to rebuild local state stores by replaying logs from the beginning or from a specific offset while preserving transactional guarantees.

Microservice-oriented publications highlight architectural patterns that integrate streaming with service decomposition, emphasising the importance of service decomposition in microservice-oriented architectures. Podduturi describes microservice landscapes where each service owns its data and communicates through event-driven mechanisms, with architectures frequently relying on patterns such as event sourcing, CQRS and saga-style coordination [6]. Tanneru focuses on Kafka as a central messaging layer for microservices. Services publish domain events into Kafka topics and subscribe to relevant topics to maintain their own materialised views and databases [9]. In these settings, historical reprocessing involves replaying domain events into consumers—either by rewinding consumer offsets on existing topics or by copying subsets of events into dedicated "backfill topics" that are consumed by specialised pipelines.

Across the surveyed cases, three main strategies for reprocessing historical data in Kafka-centric real-time systems can be distinguished.

The first strategy is full-topic replay into the existing real-time pipeline. In its simplest form, this involves resetting consumer offsets to an earlier point or creating a new consumer group that reads from the beginning of the topic while the topologies remain identical. This approach suits corrections of business logic that affect pure transformations without external side effects. Studies on Kafka Streams and Flink demonstrate that local state can be reconstructed entirely from the log and changelog topics, provided that retention is sufficient and exactly-once semantics are enabled [3, 5, 10]. Saket and co-authors, in their migration to a streaming pipeline for machine-learning features, explicitly rely on event-time semantics and stateful operators backed by RocksDB and checkpointing to maintain correctness when processing both historical and live data [7]. Full-topic replay remains a straightforward concept, but for high-throughput systems with multi-year retention, it risks

overloading production clusters; therefore, it often requires throttling, separate consumer groups and careful capacity planning grounded in measured throughput figures such as those reported by Pelle et al. [5].

The second strategy combines snapshots with log replay. Here, a consistent snapshot of the state—either a Flink savepoint, a Kafka Streams state store backup, or an external database snapshot—is taken at a known point in the log. Historical reprocessing starts from that snapshot and replays only the tail of the log from the snapshot position onwards. Mei and co-authors demonstrate that Flink 2.0 can handle huge states with rapid checkpointing and recovery thanks to disaggregated storage [3]. This behaviour supports long-running backfills: operators can create new jobs that restore from a past savepoint and then replay a time-bounded segment of Kafka topics, reconstructing state corresponding to a specific historical interval without having to rewind the entire log. Dev and Usha's fraud-detection pipeline implicitly follows this pattern when it uses checkpointing and stateful operators to maintain model-related features; re-training or feature definition changes can be addressed by restoring older checkpoints and replaying segments of transactional history [2]. Similarly, Kafka Streams applications, as described by Wang et al., depend on changelog topics to reconstruct local state stores. Snapshots of changelogs, combined with offsets, yield a practical mechanism for partial replay [10].

The third strategy isolates a dedicated backfill pipeline from the online path. Several works describe architectures where new streaming jobs or microservices read historical data from Kafka or external storage, compute derived results, and publish them into new topics or external stores without disrupting the existing pipeline [1, 4, 6–9]. Bozkurt and colleagues, for example, construct Flink topologies tailored to specific analytical tasks and evaluate their behaviour under different scenarios [1]. Pamarthi's financial streaming architectures feature separate flows for fraud detection, risk monitoring and analytics, suggesting a natural split between live scoring and historical recomputation [4]. Podduturi outlines the use of event sourcing and CQRS, where read models are materialised from an event log by dedicated projection services [6]. In such designs, backfill work is performed by launching additional projector instances that read from earlier offsets or from historical copies of topics and rebuild read models in isolation. Once reprocessing finishes, a controlled cutover shifts traffic to the new projections or state stores.

These strategies can be combined with specific design choices around state location. Local state stores, such as RocksDB in Kafka Streams or Flink's embedded state backends, favour low latency during live processing and replay, as state is co-located with computation [3, 5, 10]. They depend on Kafka changelog topics or remote file systems to persist and restore state. External stores—such as relational databases, key-value stores, or search engines—offer independent scaling, familiar query interfaces, and potentially reduced operational coupling to the streaming cluster, as illustrated in microservice architectures and Kafka-centric integration scenarios [6, 8, 9]. When historical reprocessing is performed, pipelines that rely on local state typically replay events faster, limited mainly by Kafka throughput and state-backend IO. In contrast, pipelines that write into external stores need to consider write amplification, index rebuilds and transactional semantics on the external side.

The reviewed works on AI and domain-specific pipelines add further nuance. Tambi examines Kafka-driven AI models where Kafka topics feed online inference, and reprocessing historical data corresponds to replaying event streams into updated models to regenerate features or predictions [8]. Saket et al. describe a transition from batch to streaming for training data preparation, where Flink streaming joins over Kafka replaces batch joins over static tables [7]. In such settings, historical reprocessing must respect event-time semantics during replay to avoid distorting feature windows and training distributions.

Figure 1 summarises these relationships in the form of an integrated Kappa-style pipeline, adapted from the fraud-detection streaming architecture described by Pamarthi.

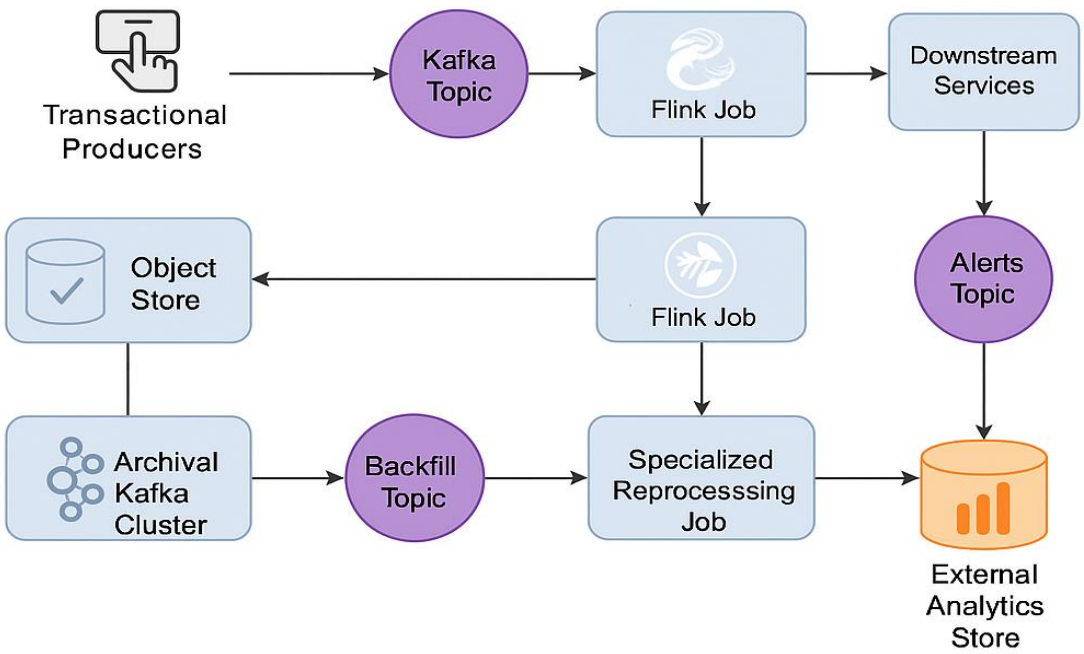

Fig. 1. Kafka–Flink pipeline for real-time and historical reprocessing in a fraud-detection scenario (compiled by the author based on [4])

In the illustrated topology, transactional producers write into a primary Kafka topic. A Flink job consumes this topic, maintains state in a disaggregated backend as described by Mei et al. and emits fraud alerts into a separate topic used by downstream services [3, 4]. Historical data stored in an object store or an archival Kafka cluster can be re-ingested into the same primary topic or into a dedicated backfill topic, which is consumed by a specialised reprocessing job. This job shares the same transformation logic but writes results into a new alerts topic and into an external analytics store. Control over consumer groups, topic selection and output destinations allow operators to replay years of history without pausing real-time fraud detection, illustrating how the architectural strategies discussed above materialise in a concrete system.

Overall, the literature indicates that robust historical reprocessing in real-time systems rests on three pillars: durable event logs with sufficient retention and partitioning, stateful stream processors with strong correctness guarantees and efficient state management, and microservice-level patterns that separate online paths from backfill pipelines. Kafka and its ecosystem, combined with Flink, Kafka Streams, and microservice practices, supply the building blocks. Architectural decisions on state placement, replay strategy, and cutover mechanism determine how effectively historical data can be reprocessed in production.

4. Discussion

The analytical synthesis of the reviewed works enables a structured comparison of design options for historical reprocessing in Kafka-based Kappa architectures. A first-dimension concerns where the state resides during both online processing and replay. A second dimension concerns the organisation of pipelines and topics used for replay and cutover.

Table 1 contrasts local state stores embedded in stream processors with external state stores managed outside the streaming runtime. The comparison draws on empirical measurements of Kafka Streams performance [5], descriptions of Flink's embedded and disaggregated state [3], the correctness properties of Kafka Streams [10], and architectural accounts of microservices using external databases alongside Kafka [6, 8, 9].

Table 1 – Local versus external state stores for historical reprocessing in Kafka-based real-time systems [2–10]

Dimension	Local state (embedded stores)	External state (databases, data lakes)
State location and technology	State co-located with processors, stored in RocksDB or similar backends in Kafka Streams and Flink, with durability achieved through changelogs or remote DFS.	State is stored in independent systems, such as relational databases, key-value stores, search engines, or analytical warehouses, accessed over the network.
Latency during online processing	Low end-to-end latency due to local lookups and writes; empirical studies report sub-second processing under realistic loads when Kafka Streams and Kafka are tuned appropriately.	Latency depends on the external system's response time, transaction model, and network conditions; it may be suitable for use cases where a few extra milliseconds are acceptable.
Behaviour under large-scale replay	High-throughput replay is limited mainly by Kafka I/O, checkpointing, and compaction behaviour in the state backend. Flink 2.0's disaggregated state further reduces checkpoint and recovery overhead for replay workloads.	Replay throughput is often constrained by write amplification and indexing overhead in external stores; long backfills may require throttling, batch writes or temporary relaxation of secondary indexes.
Failure isolation and blast radius	Failures in stream processors directly affect the availability of state; recovery depends on restoring state from changelogs or remote snapshots, which have been optimised in modern Flink deployments.	Failures in the streaming layer and external stores are partially decoupled; external state may remain available even if streaming jobs are restarted, but consistency between log and database must be managed explicitly.
Operational complexity	Operators manage fewer moving parts; the state lifecycle is tied to the streaming cluster, simplifying backup and restore at the price of tighter coupling.	Teams maintain separate operational stacks (Kafka, stream processors, databases), with dedicated backup, upgrade, and scaling procedures. As complexity increases, so does flexibility.
Typical use cases	High-throughput pipelines where most reads and writes stay within streaming jobs and where replay speed and correctness guarantees are priority (fraud detection, time-sensitive analytics).	Microservices that combine streaming with rich query workloads, ad-hoc analytics or transactional updates shared with non-streaming components (customer profiles, configuration stores, search indices).

The table indicates that local state stores, especially when combined with disaggregated architectures such as Flink 2.0, favour scenarios where historical reprocessing requires sustained high throughput and where the main outputs are derived streams rather than shared transactional databases [3, 5]. External state stores are suitable for ecosystems where the same data must serve diverse query patterns and where microservices or analytical tools expect traditional database interfaces [6, 8, 9]. For Kappa-style architectures, a common compromise involves keeping operationally critical state within stream processors while pushing slower-moving

or read-heavy projections into external stores, which are fed by derived Kafka topics.

Table 2 summarises the architectural strategies for historical reprocessing identified in the literature and in the previous section, relating them to typical tooling and scenarios. The classification relies on case studies that utilise Kafka and Flink in conjunction [1, 2, 4, 7], AI pipelines driven by Kafka [8], and microservices that employ Kafka as an event bus [6, 9].

Table 2 – Architectural strategies for reprocessing historical data in Kafka-centric Kappa architectures [1–10]

Strategy	Short description	Typical tooling and topic design	Example scenarios and sources
Full log replay through the online pipeline	Historical events are read from the beginning (or from a chosen offset) of production topics by a consumer group running the same topology as the online path; live traffic continues on the same issues with separate consumer groups.	Kafka topics with sufficient retention; Kafka Streams or Flink jobs using embedded or disaggregated state; separate consumer groups for live and replay traffic; optional throttling.	Correcting pure transformation logic or enriching existing aggregates without external side effects; migrating from batch to streaming as described in real-time event joining and case studies of Flink–Kafka integrations.
Snapshot-plus-log reconstruction	Reprocessing begins from a state snapshot (Flink savepoint, Kafka Streams state backup, or database snapshot) and replays only the subsequent part of the log, reconstructing the state for a chosen historical interval.	Checkpointing or snapshot mechanisms in Flink 2.0 and Kafka Streams, dedicated topics for changelogs, and infrastructure for storing and cataloguing snapshots and their corresponding offsets.	Rolling back to old model versions or business rules, recomputing features for a fixed time window in fraud detection or risk analytics, as reported in streaming pipelines in finance and fraud-detection systems.
Isolated backfill pipelines	Historical data is copied into dedicated “backfill topics” or read from archival storage and processed by specialised streaming jobs or microservices; outputs are written into new topics or external stores, and traffic is switched after verification.	Kafka Connect or custom loaders to re-ingest historical data; parallel pipelines in Flink or Kafka Streams; separate output topics or databases for backfilled projections; controlled cutover via topic re-wiring or configuration changes.	Building new read models in event-sourced microservice landscapes, generating new AI features from old events, or introducing additional analytics streams without disturbing existing consumers.
Hybrid selective replay	Only subsets of partitions, tenants or event types are replayed, often combined with filters or projection services that target specific microservices or analytical products.	Fine-grained topic partitioning by tenant or domain; selective consumers that read only specific partitions; projection services implemented as microservices subscribing to filtered topics.	Tenant-specific backfills in multi-tenant SaaS platforms, selective reconstruction of corrupted projections, or country-specific regulatory recomputation where only part of the event space is relevant.

These strategies reflect a trade-off between simplicity and control. Full log replay utilises existing infrastructure but risks competing with online processing for resources [3, 5, 10]. Snapshot-plus-log reconstruction reduces reprocessing time for long histories, but it depends on reliable snapshot management [2, 3, 10]. Isolated backfill pipelines add complexity in topic management and cutover procedures, yet they align well with microservice practices around independent deployments and gradual rollout [1, 4, 6–9]. Hybrid selective replay demonstrates how careful partitioning and topic design enable the limitation of reprocessing scope and reduction of operational risk [5–7, 9].

For organisations already invested in Kafka-centric Kappa architectures, the literature suggests combining these strategies in a layered design. Operational states that must be continuously accurate—for example, fraud scores or risk indicators—can be handled by local or disaggregated state stores with either full replay or snapshot-plus-log reconstruction [2–4, 7, 10]. Derived projections consumed by external services can be rebuilt through isolated pipelines and cut over when consistency checks pass [1, 4, 6, 8, 9]. Domain events used for both operational and analytical purposes benefit from careful schema evolution and topic retention policies, so that historical replay remains technically feasible for several years of data [5–7].

The choice between local and external state, and between direct replay and isolated backfill, closely interacts with an organisation's operational maturity. Systems described in high-throughput IoT and financial scenarios rely heavily on automation for scaling, checkpoint management and monitoring [3–5]. Microservice landscapes emphasise design patterns and contract-based event schemas to contain the complexity of replay and backfill [6, 8, 9]. In both categories, the success of historical reprocessing depends not only on the raw performance of Kafka and stream processors, but on disciplined topic design, versioning strategies for topologies and state, and well-defined procedures for validating replay results before exposing them to end users.

5. Conclusion

The study demonstrates that the effective reprocessing of historical data in Kafka-centric real-time systems relies on treating the event log as the primary system of record and designing stateful processing around this assumption. When architectures follow Kappa principles, the replay of event logs, combined with strong exactly-once guarantees in Kafka Streams and Apache Flink, becomes the foundation for reconstructing application state without relying on separate batch stacks.

Comparative analysis of local and external state stores indicates that embedded or disaggregated state backends favour high-throughput backfills and tight control over correctness. In contrast, external databases bring flexibility at the cost of more complex replay procedures. Systems that integrate Kafka with microservice patterns such as event sourcing and CQRS benefit from isolating backfill pipelines and using topic-level controls for cutover.

The examined strategies—full log replay, snapshot-plus-log reconstruction, isolated backfill pipelines and hybrid selective replay—provide a practical toolkit for engineering "time-travel" capabilities into existing Kafka-based infrastructures. Combining these patterns with rigorous topic design, disciplined state management and operational safeguards allows organisations to recompute years of history while keeping real-time services responsive, which makes such approaches especially suitable for streaming-heavy microservice ecosystems built on Kafka and Kappa architecture.

References

1. Bozkurt, A., Ekici, F., & Yetiskul, H. (2023). Utilizing Flink and Kafka technologies for real-time data processing: A case study. *The Eurasia Proceedings of Science, Technology, Engineering and Mathematics, 24*, 177–183. https://doi.org/10.55549/epstem.1406274

2. Dev, R. S., & Usha, J. (2025). Real-time processing with Kafka, ksqlDB & Apache Flink: A fraud detection pipeline. *International Journal of Computer Applications, 187*(60), 13–18. https://www.ijcaonline.org/archives/volume187/number60/dev-2025-ijca-925872.pdf

3. Mei, Y., Lan, Z., Huang, L., Lei, Y., Yin, H., Xia, R., Hu, K., Carbone, P., Kalavri, V., & Wang, F. (2025). Disaggregated state management in Apache Flink 2.0. *Proceedings of the VLDB Endowment, 18*(12), 4846–4859. https://doi.org/10.14778/3750601.3750609

4. Pamarthi, S. (2023). *Apache Flink and Apache Kafka in financial services: Real-time streaming for*

data processing and analytics [White paper]. https://www.researchgate.net/publication/397017733_Apache_Flink_and_Apache_Kafka_in_Financial_Services_Real-Time_Streaming_for_Data_Processing_and_Analytics

5. Pelle, I., Szőke, B., Fayad, A., Cinkler, T., & Toka, L. (2023). A comprehensive performance analysis of stream processing with Kafka in cloud native deployments for IoT use-cases. In *NOMS 2023: IEEE/IFIP Network Operations and Management Symposium* (pp. 1–6). https://doi.org/10.1109/NOMS56928.2023.10154377

6. Podduturi, S. M. (2024). Real-time data processing in microservices architectures. *International Journal of Computer Engineering and Technology, 15*(6), 760–773. https://doi.org/10.5281/zenodo.14228620

7. Saket, S., Chandela, V., & Kalim, M. D. (2024). Real-time event joining in practice with Kafka and Flink. *arXiv*. Advance online publication. https://arxiv.org/abs/2410.15533

8. Tambi, V. K. (2023). Real-time data stream processing with Kafka-driven AI models. *International Journal of Current Engineering and Scientific Research*. Advance online publication. https://philpapers.org/archive/VARRDS.pdf

9. Tanneru, B. (2023). Application of Kafka messaging in microservices for real-time data processing. *International Journal of Innovative Research in Engineering & Multidisciplinary Physical Sciences, 11*(5), 1–4. https://doi.org/10.5281/zenodo.14945204

10. Wang, G., Chen, L., Dikshit, A., Gustafson, J., Chen, B., Sax, M. J., Roesler, J., Blee-Goldman, S., Cadonna, B., Mehta, A., Madan, V., & Rao, J. (2021). Consistency and completeness: Rethinking distributed stream processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)* (pp. 2602–2613). https://doi.org/10.1145/3448016.3457556