Deep Science Publishing, 2026
https://doi.org/10.70593/978-93-7185-592-1_4

DeepScience
Open Access Books

Chapter 4: Streaming Data Platforms and Event-Driven Design

4.1. Introduction

Streaming data platforms allow for the processing of continual data records at large volume, high velocity, and with low latency. Event-driven design applies a programming style where the flow of the program is determined by events, which are changes in the state of the system. Events are captured over time (a time-stream) and then strongly typed into an event schema with the associated data.

Three foundational questions are important. First, what are the core principles behind streaming platforms? Second, how is event-driven design applied to build event-driven systems? Third, how do Cap Theorem trade-offs shape the design of streaming systems? Answers draw from experience building and deploying systems for capturing and analysing data from transactions and demographic flows at Twilio and booking data at the Farelogix airline distribution platform.

4.1.1. Background and Significance

Streaming data platforms and event-driven design merit their own models because they offer distinct principles and patterns that influence architecture in ways that batch and service-oriented design do not. Streaming data platforms process a continuous flow of incoming events in real time and mirror the modern world of fast business and instant analytics. Their core concepts—streams, events, and time—blur the binary distinction between batch and stream data, and their architecture follows ingestion, processing, storage, and serving patterns that often rely on the same technology in the same sequence as batch-oriented systems.

Event-driven design subscribes to event sourcing as the primary form of domain model and a publish—and—subscribe model for communication. It enables stateless microservices with low coupling inside and between each layer of the architecture. It can

tolerate reduced data quality through defined rules and is optimized for horizontal scaling, though it requires idempotence in processing, delivers at least once rather than exactly once guarantees, and necessitates support for out-of-order delivery. Event-driven design can, however, introduce additional complexity through data replay during retries and compensating actions alluded to in the design of materialized views and serving layers in the storage model.

Fig 4.1: Streaming Data Platforms and Event-Driven Design

4.1.2. Research design

This research combines systematic mapping and enterprise architecture methods to produce a coherent view of streaming data platforms and the event-driven design principles that exploit them. First, a set of candidate streaming platform designs and event-driven principles is defined through a systematic mapping of the literature on streaming data. Streaming data platforms are a cohesive architecture that brings together a set of data- and process-centric properties supporting data-driven decision making and the creation of new events from the detection of pattern matches in existing data. Streaming data platforms also support decision-automation through event-driven architectures, which strongly decouple decision following from decision making within the same organisation.

The cognitive and resource cost of actually making decisions is generally much higher than that of simply responding to other people's decisions. The above principles,

proposed in the context of organisation-wide event-driven architectures, have been linked with those of distributed systems. The platform architecture ultimately drives the enterprise architecture, defining how data moves and is processed, both in batch and in real time.

4.2. Foundations of Streaming Data Platforms

Streaming Data Platforms and Event-Driven Design: Principles, Architecture, and Applications

The term streaming data platforms designates a class of data system that extends the batch-centric data lake approach in order to address the requirements of true streaming use cases—or at least make them possible. The shift from traditional batch-centric data systems to new architectures for true streaming is driven by the emerging Digital 4.0 trend; Streaming Data Platforms address the needs of organisations that aim to develop a Data-In-Motion capability that will allow them to differentiate and compete on the basis of real-time, predictive, and prescriptive data-driven insights. Typical Streaming Data Platforms: applications of streaming processing in mainstream sectors, such as business, finance, and telecommunications; centrality of streaming use cases and event-driven application design.

Following these themes, the section discusses the core tenets of Streaming Data Platforms and the guiding principles of event-driven design. Both perspectives are concerned with the very structure of real-time data processing, be it a single application developed with streaming technology, or a complex system made up of many loosely coupled, event-driven components. Local application development – a single, streaming application – needs to consider event sources, routing, stateless vs. stateful processing, and guarantees of accuracy. Distributed architecture requires complementary focus on the principles of materialized views, data storage, and the roles of event producers and consumers.

4.2.1. Core concepts: streams, events, and time

The concepts of stream, event, and time form the foundation of event-driven systems supported by a streaming data platform. A streaming data platform supports two types of streams, which have fundamental differences: event streams and data streams. An event stream is a potentially infinite sequence of events — changes that have occurred in the system. Operations on these streams derive data streams, which represent current or projected system state. Streams receive the data they contain from sources. The most common types of sources are user applications, devices that collect information from the

physical environment, and external external service or system interfaces that produce information or require actions. In a streaming data platform no dedicated sender of information is required; streams do not persist source content. The nodes processing stream content can be heretical; a sending event source can simply delete its own copy of the data.

Despite the similar terminology, the definitions of event and data vary according to the type of processing being considered. A data stream consists of records, the unit of data exchanged as the stream contents flow through the nodes. A record encodes information in key–value format: key, attribute string; value, structure defined by its associated schema. Time is an essential element of both event and data streams. Events are ordered according to event time, the instant at which the change occurred in the system. Data stream records are ordered according to processing time, when the record was ingested into the stream. Operations can be grouped into stateless and stateful processing based on whether the result of the operation depends only on the incoming record.

4.2.2. Architectural patterns: ingestion, processing, storage, and serving

To develop typical event-driven applications using streaming data platforms, the above concept of events and streams must be extended into specific architectural patterns that focus on process and the flow of data through the different processing phases of the system. These processing patterns subdivide an application into four, often asynchronous, stages: ingestion, processing, storage, and serving. Application-level ingestion patterns describe where and how events are produced; the remaining two processing phases describe how events are communicated through the system and how storage is conceptualized within an event-driven architecture.

In streaming data platforms, these phases correspond to the ingestion, processing, and serving components. Event sources write events into the streaming data platform's buffering layer, from where data are consumed and processed. Ingestion patterns outline the specification of event sources: the choice of external system that acts as an event source, what logical data do the sources represent, what transactions are needed, and how they are translated to produce a stream of events.

4.3. Event-Driven Design Principles

An application’s composition, including the interaction and coupling between its separate components or services, can have a large impact on the possibility of incorporating other services and exchanging specific implementations. The idea of decoupling components by employing events for signaling and communication has been

referred to as event-driven design or event-driven architecture. Such decoupling allows services to be added, removed, or modified without breaking existing components that are not directly dependent on them. The ability to add new services is particularly suited for a streaming data platform where continuous data flows into the system that can be processed, analyzed, and made accessible to new services without affecting the core application.

The simplest form of an event is that of a record generated by an event source that is routed to an event-processing function. Depending on the nature of the function, a function may be stateless or stateful; that is, its output may depend only on the current input or on all inputs it has seen so far. For a wide variety of applications, statelessness provides the easiest programming model and the highest throughput, scaled-out or distributed implementations being naturally supported. However, the uniqueness of these sources do not just exhibit or impose the challenges on a streaming data platform; rather they are relevant attributes to all streaming data platform implementations. Important properties and techniques include idempotence, which underpins the ability to scale-out in batch or in those services that write to distributed storage backends, along with the precise or at-least-once processing guarantees of stateful operators and windowed aggregations; windows themselves requiring special semantic models to correctly express the relationship between real-world events and the computation outcomes.

Fig 4.2: Event-Driven Design Principles

4.3.1. Event sources and event routing

Any event-driven application is subject to a continuous stream of inputs from multiple sources. For example, in a digital marketing application, users browsing a website send a stream of events, while clickstream data and advertisement performance data act as a

source of batch input. Such data sources can be classified as either trigger sources or batch sources. Trigger sources continuously create small streams of events in reaction to some stimulus, and often in real time. Batch sources continuously collect information and make it available in a large-batch form at a specific frequency. Trigger sources of all types tend to push their outputs downstream, while batch sources are usually pulled by the batch processing stage.

Trigger events can be made richer and more expressive by adding context about the data and about previous events. This context is usually made available in the form of a static lookup table or as a small stream of batch updates containing the latest changes to the lookup table. These updates need to be routed to the stateful processing step so that the lookup table can be kept up to date. The routing can be implemented as a smart join operation that sends the updates to the correct stateful processing instance based on the join key.

Trigger events can also be used to route the batch data—due to their small size, these trigger events are good candidates for sending out to the cloud—and, in the case of a marketing application, can also be used to trigger the running of a batch processing step that creates advertisement performance data.

4.3.2. Stateless versus stateful processing

Stateless data processing methods do not maintain any information on intermediate results between distinct input events. Therefore, each input event is processed in isolation. For example, the calculation of maximum or minimum values over an event stream do not require state, since the input and output of the processing methods can be stored as part of a (materialized) view at any time. Hence, every incoming event can be handled independently of past events in the stream.

Stateful methods, by contrast, maintain additional information in state variables that capture knowledge about previously processed events, which can then be used to produce an output for every incoming event. Typical examples are orders of magnitude higher than the throughput of the stream: a stream of (partially) gerund events with the fraction of increase shown of now surpassed the latest on the history dimension an event stream from an in-store shopping service can simply be stored and the sizes be updated as new purchases are detected.

Stateful methods generally suspend their operation until state information is completed and stored, for later on-line or batch processing. In many event-driven business applications, however, events occur continually and states are updated constantly, often with time-windows or penalty functions. Examples include real-time detection of Visa-

like cards used by been stolen, or tracing the footsteps of suspects through travel-control camera events.

The extra information held in state variables allow for semantics not available with stateless methods. In the Visa example, the states would capture the list of nicknames (perhaps with associated amounts) and their temporariness.

4.3.3. Idempotence, exactly-once processing, and at-least-once guarantees

One of the biggest challenges with event-driven applications is reliably delivering messages to downstream consumers. When using volatile processing capabilities, this is done through at-least-once processing; the idea is to retry processing as many times as needed until it succeeds. This approach works fine for stateless operations such as writing to a database and is sufficient for many use cases. However, many stateful operations such as aggregations or joins are complex to support in a retryable manner. Nevertheless, they can be built on top of at-least-once processing if the subsequent stages are idempotent.

Idempotence means that performing the same operation multiple times has the same effect as performing it once. In the context of writing to a sink it means that the same message can be safely written over and over again. Idempotent operations have been traditionally used in databases for incrementing counters in a failed-attempt-safe manner. Additionally, many databases provide some type of unique constrait expression to enforce it. This is equivalent to saying that if two consumers try to apply a change, only one will succeed, like different shards doing the same action at the same time.

To make a stream-processing application fault-tolerant while avoiding the complexity of ensuring idempotence, Direct Acyclic Graph (DAG) subtrees can be replicated on multiple processing nodes. The writes to sinks can then be partitioned per node. In cases where there are multiple subscriptions to a topic, such as a user-oriented topic that needs to be presented in different formats, read replicas can be deployed to consistently handle the expected load without data loss or duplication.

4.4. Streaming Processing Paradigms

Batch processing, the traditional dominant paradigm for large-scale data processing, can be applied to streaming scenarios with adequate care, but the inherent latency of batch processing operations disqualifies it as a streaming processing paradigm. The differences between streaming and batch processing can be viewed on a spectrum, where streaming processing offers a sequence of results over time, and batch processing—a single final result. Streaming data platforms support windowed aggregations that group unbounded

streams into finite mini-batches, and queries consuming bounded streams and producing unbounded data. Aggregate functions applied on finite streams can be expressed as windowed aggregations, while higher-level transformations consuming unbounded streams can often be expressed with windowed aggregations interspersed with stateful processing steps. Supports for windowed aggregations, temporal event join processing, and event-time semantics are consequently the key distinctions of the stream-batch processing paradigms.

Batch and streaming processing differ from another viewpoint as well. Batches move through the processing pipeline in distinct waves, commonly described as flows of waves passsing through different water bodies. Events arrive in streams in an almost endless succession, propelling the processing system to a state of flow. Streaming processing systems experimental lose an explicit flow concept even though they often perform natural-wave-appearance operations.

4.4.1. Batch vs. streaming trade-offs

Event-driven design, the practice of connecting event-processing components through an event bus, leverages streaming data platforms that provide a uniform way of ingesting, processing, storing, and serving streaming data. While event-driven design provides clear separation of concerns, it must be applied judiciously to remain effective. Built-in infrastructure capabilities may not align with business needs, leading to costly redundancy. Streaming data platforms and event-driven design offer well-established concepts, knowledge, and experience for domaining teams embarking on new projects and transitioning from batch designs to streaming data processing. Nevertheless, the functionality preserved or gained through a streaming architecture must be weighed against engineering and operational costs.

Traditionally, a batch-processing design, which uses the same compute clusters for both production and consumption within a scheduled framework, is followed. Utilizing infrastructure for event-driven design only when needed helps contain operational costs and reduce engineering overheads. Where shared natural-progression and support-processing pipelines cannot be built, a streaming dedicated-event-processing architecture can help disambiguate. Event-driven design is process-agnostic. Essential adaption is needed only for natural-progression persistence. Rather than dedicated ingestion infrastructure within an event-directed architecture, separate practically-independent and schedules pipelines may be deployed at a fraction of the hosting costs. Implementing such avoidance when possible can significantly reduce engineering, de-risk fit-for-purpose quality, and cut current-affairs support costs by half.

4.4.2. Windowing, aggregations, and time semantics

In many applications, the computational results of processing an unbounded data stream must be computed over finite intervals of time. These intervals, called windows, may be static or dynamic, may be defined based on event timestamps (real-world time) or processing timestamps (the time when an event is ingested into the system), and may introduce new complexities in state management; for example, the managed state may need to retain data for only a subset of the past event stream instead of all past events. In some cases, the managed state does not directly reflect the semantics of the computations. Rather, the system may employ a compensating mechanism based on an underlying durable storage to satisfy the expected semantics.

Windowing is commonly used for aggregation operations such as counting, summing, averaging, and joining, which naturally compute an evolving answer based on past events and are desired at a certain level of granularity. A batch of results is produced at the end of each window interval. Much work has focused on designing progressively efficient algorithms and systems for such aggregate queries. Event-time semantics is essential when an aggregate query runs in a real-world application and the input data is subject to delays, because it alleviates the quality-of-result problems caused by input delays, at the cost of worst-case design. A more relaxed quality-of-result concept is often used in practice — namely, the results for a past window remain correct, while those for a future window are subject to possible change. The precision of the evolving answer can be traded against resource consumption by carefully bounding the size of the retained past state.

Windowing also comes into play when querying observational data. Queries may look for sensor devices exhibiting anomalous behaviour in the past few hours and remain current while new data arrive. Such queries can be processed at a low level of granularity, but there is no trade-off with resource consumption; they require a Compensating Event Processing (CEP) architecture and can thus be done efficiently without any need for a complex state management mechanism.

4.5. Storage and Serving Models for Streaming Data

Future state changes of a domain, such as the current location of train carriages, vehicle traffic on a motorway, or the temperature of a room, change rapidly. By contrast, durable storage cannot change state instantly; it is slowed by the time taken to commit the change and by the latency of reads across a network. Data storage for streaming-processing systems must support replay and may be used to recover from failure. As recent research shows, there are at least three ways to relax the consistency guarantees associated with storage consumed by a streaming-processing application: materialized views, time-

travel support, and compensations. A serving layer is then built on top of the storage model.

The events produced by the newest state changes may be discarded based on policy, or kept in a separate structure for a limited time. Both serve to give only a limited budget to storage. Stream-processing sentences are occasionally used to execute these events by methods associated with change detection or incremental modes of processing.

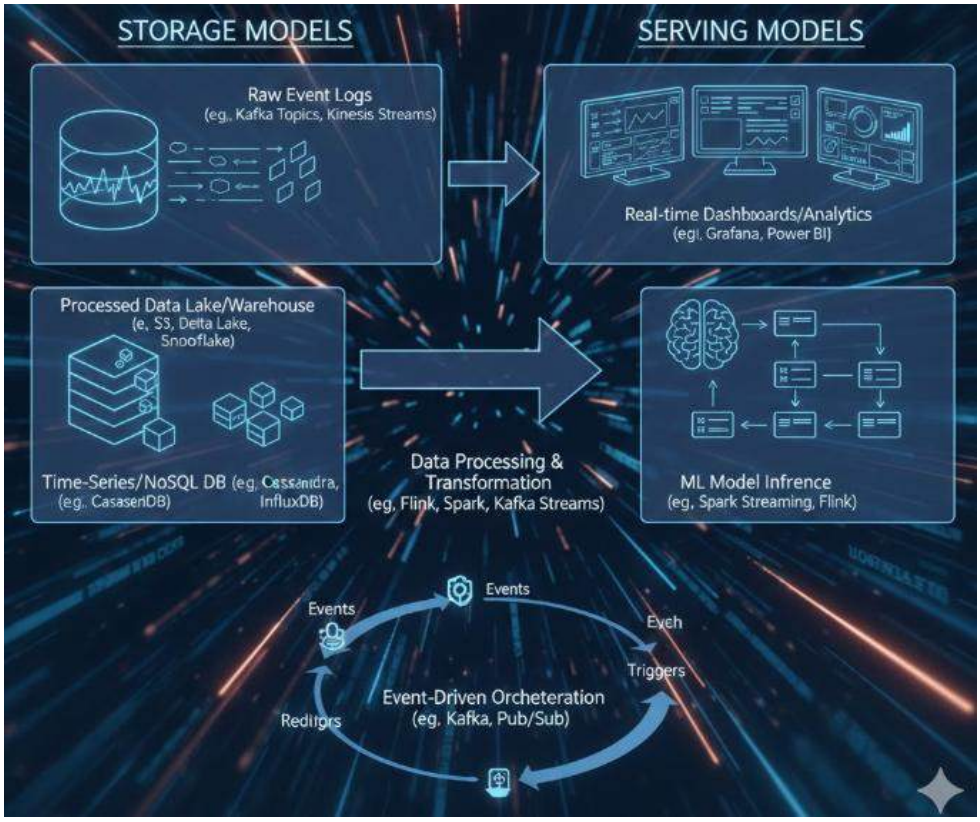

Fig 4.3: Storage and Serving Models for Streaming Data

4.5.1. Durable storage backends and log-structured stores

Durable storage backends provide fault tolerance by guaranteeing the preservation of data with per-event durability while also offering high throughput. As such, they facilitate the replication of events across processing nodes. For cost-effective and performant storage, data is often written in a log-structured manner, where the focus is on minimizing sequential write costs rather than optimizing for random reads. The latter are typically served from in-memory caches or in the form of materialized views backed by the storage system. The log-structured approach maps well to traditional disk and

solid-state drive (SSD) architectures, which are optimized for sequential writes. Further compression techniques (e.g., columnar and Delta encoding) can help reduce the storage foot-print.

Log-structured storage systems use a write-ahead log (WAL) to persist transactions before they are executed; the data flow from databases also has a similar design. Following a crash failure, the read/write operations can be recovered using the WAL. While the parent structure is a B+-tree, the log-structured database can be conceptually viewed as a log-structured storage to which HDFS-like event stores can be consistently replicated using a WAL. Durably-consistent log-structured stores can also store raw events for time-travel queries and can be leveraged to implement reply functionalities such as compensatory actions in microservices.

4.5.2. Materialized views and serving layers

A materialized view is the result of running a query against a streaming data platform, storing the current output of the query for fast lookups. Queries that create materialized views can range from basic operations, such as filtering, to complex aggregations and joins. These views enable real-time lookups via a simple key-based API, while the underlying query continues to run incrementally to maintain the view's accuracy. Materialized views can be automatically maintained by the streaming data platform whenever the data in the source table changes, so that the views reflect the results of the queries, with minimal latency.

Despite their appeal, managing multiple materialized views for an application can introduce a non-negligible maintenance burden. The streaming data platform may store a materialized view for any query deemed expensive to recompute; however, the view-creation query must remain fixed. An alternative is to maintain a stopping-point finger table that supports fast lookups. Instead of explicitly defining a set of materialized views, the data engineer or analyst specifies an order on the set of views and the search key range index. The streaming data platform retains a limited number of views specified by this order and the search-key-index finger table, automatically maintaining the data in the view that receives the next-query lookup with the lowest latency.

Materialized views of using a serving layer naturally fit within decoupled architecture, while otherwise integrating transparently. A serving layer differs from a mere front-facing entry point, as the serving-layer interface is not limited to ingestion. However, it is common for stream endpoints to be cleanly separated from serving layers. Serving layers can also be used to support applications with one or more read paths, where the read workload on the source tables is high relative to the updates. In such situations,

continuous aggregates are maintained over the source tables, with SCOPE ordered indexes on the aggregated data, to provide low-latency response for the read workloads.

4.5.3. Time travel, compensations, and data replay

The tap of a finger into a GPS-enabled personal device awakens a cascade of automated responses that monitor and activate a myriad of systems, resulting in reliable and accurate information. The user's whereabouts and actions at any given point in time are captured in a stream of events for logging, processing, and future recommendations. Yet, the user's physical being does not cease to exist. Data size can be expunged into the abyss of forgetting, returning only when derived from log data or materialized views. And there remains the shortcoming of compensating for earlier decisions that have already changed reality.

The bulk of enterprise data should be treated as a reservoir of knowledge supporting every decision made throughout the life cycle of a corporation: Whether a potential client has been made an offer, whether a prospect has receded into oblivion, whether an anticipated purchase was made at a higher price than previously offered, or whether a technical misjudgment has caused the loss of a client. Even the notion of "client" can change over time and may not protect against price jumps as Black & Scholes realized and debated. Every event represents information and should be preserved. All events are most valuable at the time of occurrence, but may also enable more accurate decisions in the future—when searching for areas not undergoing seasonal effects, spotting potentially ephemeral correlations, or answering a question that had occurred too late for the data to be produced on time. Data permitting exploration and review of earlier decisions should be preserved in any event.

4.6. Conclusion

Streaming data platforms and event-driven design are increasingly relevant to applications such as prosthetics, monitoring, on-line advertising, and ride hailing and sharing. Considerable attention has therefore been given to the principles of these fields and an architecture for streaming data platforms, grounded in the nature of streaming data, its processing requirements, challenges, and application patterns. Event sources, event routing, ingestion pipelines, stateless and stateful processing, message brokers, back-end processing engines, storage, and serving layers have been examined. Open questions include nonstandard data semantics, storage options for nonstripeable, nonseparable queries, and systems that provide the clean separation of concerns established by modern data warehouses, rather than precursor architectures built on relational databases.

Future Directions. The popularity of streaming data platforms and event-driven design will continue as new applications emerge and are commercialized. Current research hot spots include building reliable and secure systems that protect end users, onboarding multiple streams in real time from undermonitored or poorly monitored sensors, and integrating event-driven design into programming languages. In addition, the cost-optimizing closed-loop architecture of streaming data platforms deserves more attention.

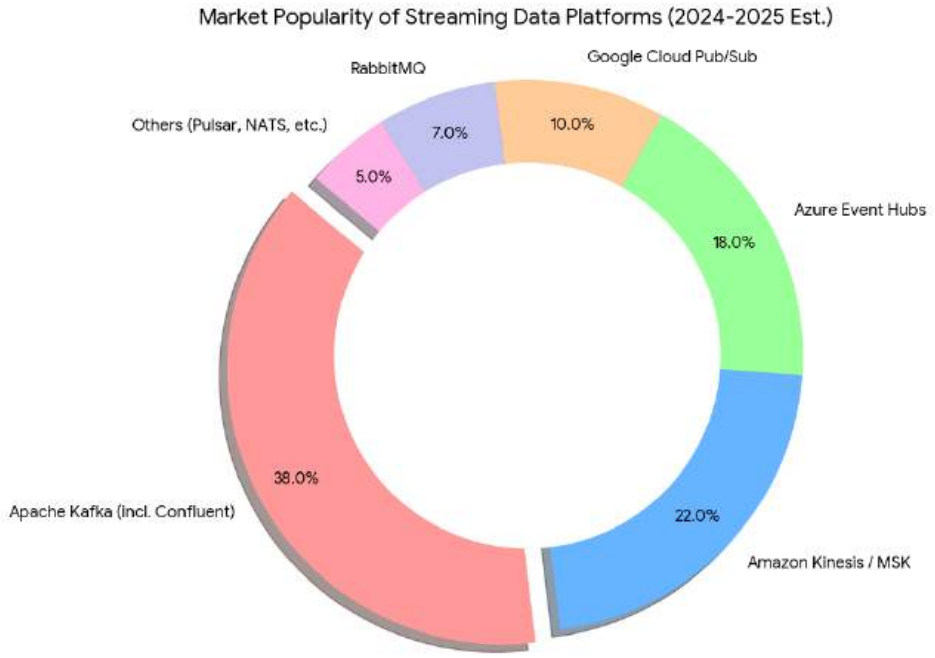

Fig 4.4: Streaming Data Platforms and Event-Driven Design

4.6.1. Future Directions

The next stage in their exploration of data platforms and event-driven software design is to closely examine event-driven design principles across four dimensions, and then consider how event-driven streaming principles apply to the key architectural functions of streaming data platforms—data ingestion, processing and storage—and how they interoperate and fit together. The results provide the foundational knowledge required to answer questions about design and implementation that are specific to an actual streaming data platform. With these aspects established, they then move on to an exploration of practical applications and experience acquired from deploying a real streaming data platform. These complement and build upon the previous theory-driven, platform-centered analysis, expanding their work in a manner that bridges those perspectives to the primary concerns at the initiatory stage of the research endeavor: the

nature of streaming data platforms, event-driven design, and the foundations of streaming processing. The following stages apply these observations and concepts, but from a more applied perspective, examining specific implementations and the lessons learnt in deploying them.

The end goal is to set out principles, patterns, and considerations for implementing a streaming data platform that is suitable for a variety of purposes. These notably include complex processing of heterogeneous sources internal to a large-scale, cross-national service, as well as undertaking resilient, elastic, near-real-time, at-scale, end-to-end processing of vast volumes of distilled user-experience telemetry from a correspondingly vast-scale web service..

References

Dario, A. (2025). Perspectives on Stream Processing. Computer Science Reports. This work explores the semantic gap between traditional relational algebra and modern distributed stream processing graphs (Dario, 2025).

Avinash Pamisetty, Vijaya Rama Raju Gottimukkala. (2024). Agentic AI-Driven Multi-Cloud Big Data Architecture For Predictive Demand, Credit Risk, And Inventory Financing In National Food Service Supply Chains. Metallurgical and Materials Engineering, 30(4), 959–975. Retrieved from https://metall-mater-eng.com/index.php/home/article/view/1933

Gulisano, V. (2021). Motivations and Challenges for Stream Processing in Edge Computing. Companion of the ACM/SPEC International Conference on Performance Engineering, 17–18. https://doi.org/10.1145/3447545.3451899

Keerthi Amistapuram. (2024). Federated Learning for Cross-Carrier Insurance Fraud Detection: Secure Multi-Institutional Collaboration. Journal of Computational Analysis and Applications (JoCAAA), 33(08), 6727–6738. Retrieved from https://www.eudoxuspress.com/index.php/pub/article/view/3934

Hu, F., Yang, C., Jiang, Y., Li, Y., Song, W., Duffy, D. Q., Schnase, J. L., & Lee, T. (2018). A hierarchical indexing strategy for optimizing Apache Spark with HDFS to efficiently query big geospatial raster data. International Journal of Digital Earth, 13(3), 410–428. https://doi.org/10.1080/17538947.2018.1523957

Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media. (Foundational text on event sourcing and stream processing).

Yandamuri, U. S. (2022). Big Data Pipelines for Cross-Domain Decision Support: A Cloud-Centric Approach. International Journal of Scientific Research and Modern Technology, 1(12), 227–237. https://doi.org/10.38124/ijsrmt.v1i12.1111

Stopford, B. (2018). Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka. O'Reilly Media. Focuses on using a central log as a source of truth (Stopford, 2018).

Nagabhyru, K. C., & Babu, A. J. Human In The Loop Generative AI: Redefining Collaborative Data Engineering For High Stakes Industries.

Ajayi, O. (2025). Practical Event-Driven Microservices: A Database-Centric Alternative to Message Brokers. Research Square. This paper analyzes the operational overhead of Kafka compared to database-centric models for moderate-scale systems (Ajayi, 2025).

Siva Hemanth Kolla. (2023). Deep Learning–Driven Retrieval-Augmented Generation for Enterprise ITSM Automation: A Governance-Aligned Large Language Model Architecture . Journal of Computational Analysis and Applications (JoCAAA), 31(4), 2489–2502. Retrieved from https://www.eudoxuspress.com/index.php/pub/article/view/4774

Das, S. S. (2023). Enterprise Event Hub: The Rise of Event Stream-Oriented Systems for Real-Time Business Decisions. Journal of Advance and Future Research, 1(10). Explores the evolution of central "Event Hubs" for real-time orchestration (Das, 2023).

Avinash Reddy Segireddy. (2022). Terraform and Ansible in Building Resilient Cloud-Native Payment Architectures. International Journal of Intelligent Systems and Applications in Engineering, 10(3s), 444–455. Retrieved from https://www.ijisae.org/index.php/IJISAE/article/view/7905

Henning, S., & Hasselbring, W. (2021). The Theodosite tool for scalability benchmarking of distributed stream processing. IEEE/ACM International Conference on Software Engineering. (Benchmarking horizontal scalability in event-driven systems).

GUNTUPALLI, R. (2025). EXPLAINABLE AI IN CLINICAL DECISION SUPPORT: INTERPRETABLE NEURAL MODELS FOR TRUSTWORTHY HEALTHCARE AUTOMATIONEXPLAINABLE AI IN CLINICAL DECISION SUPPORT: INTERPRETABLE NEURAL MODELS FOR TRUSTWORTHY HEALTHCARE AUTOMATION. TPM–Testing, Psychometrics, Methodology in Applied Psychology, 32(S9 (2025): Posted 15 December), 462-471.

Khriji, S., Benbelgacem, Y., Chéour, R., Houssaini, D. E., & Kanoun, O. (2021). Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks. The Journal of Supercomputing, 78, 3374–3401. https://doi.org/10.1007/s11227-021-03955-6

Sudhakar, A. V. V., Inala, R., Verma, A. K., Nag, K., Pandey, V., & Anand, P. S. (2025). Hybrid Rule-Based and Machine Learning Framework for Embedding Anti-Discrimination Law in Automated Decision Systems. In 2025 International Conference on Intelligent Communication Networks and Computational Techniques (ICICNCT) (pp. 1–6). IEEE. 2025 International Conference on Intelligent Communication Networks and Computational Techniques (ICICNCT). https://doi.org/10.1109/icicnct66124.2025.11232861

Laigner, R., et al. (2021). Data Management in Microservices: State of the Practice, Challenges, and Research Directions. Proceedings of the VLDB Endowment. (Examines the complexity of data consistency in event-driven microservices).

Varri, D. B. S. (2022). A Framework for Cloud-Integrated Database Hardening in Hybrid AWS-Azure Environments: Security Posture Automation Through Wiz-Driven Insights. International Journal of Scientific Research and Modern Technology, 1(12), 216-226.

Alsader, M., BarakaBitze, A. A., & Mkwawa, I. (2025). QoE-Driven Adaptive Video Streaming: Architectures, Techniques, and Future Research Challenges Toward 6G Networks. IEEE Access. https://doi.org/10.1109/ACCESS.2025.3597058

Vadisetty, R., Polamarasetti, A., Goyal, M. K., Rongali, S. K., kumar Prajapati, S., & Butani, J. B. (2025, May). Cloud-Based Immersive Learning: The Role of Virtual Reality, Big Data, and Generative AI in Transformative Education Experiences. In 2025 International

Conference on Advancements in Smart, Secure and Intelligent Computing (ASSIC) (pp. 1-6). IEEE.

Jose, J. A. C., et al. (2024). Smart Shelf System for Customer Behavior Tracking in Supermarkets. Sensors, 24(2), 367. https://doi.org/10.3390/s24020367

Lagorce, X., Orchard, G., Galluppi, F., Shi, B. E., & Benosman, R. B. (2017). HOTS: A Hierarchy of Event-Based Time-Surfaces for Pattern Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, 39(7), 1346–1359. https://doi.org/10.1109/tpami.2016.2574707

Aitha, A. R. (2023). CloudBased Microservices Architecture for Seamless Insurance Policy Administration. International Journal of Finance (IJFIN)-ABDC Journal Quality List, 36(6), 607-632.

Wen, Y., Zhang, S., & Zhang, S. (2025). Event-driven architecture and intelligent decision tree facilitated sustainable trade activity monitoring model design. PMC PubMed Central. This study addresses the rate inconsistency between data injection and processing in trade monitoring (Wen et al., 2025).

Yao, J., et al. (2025). Challenges and Opportunities in Big Data Analytics for Industry 4.0: A Systematic Evaluation of Current Architectures. IEEE Xplore. Identifies dominant technological patterns like Apache Kafka for ingestion and Spark for processing (Yao et al., 2025).

Thutari, R. T., Garapati, R. S., BM, M., & RK, S. (2025, October). Adaptive Access Control and Authentication Management for IoT Using Attention-GRU and Reinforcement Learning. In 2025 2nd International Conference on Software, Systems and Information Technology (SSITCON) (pp. 1-6). IEEE.