Deep Science Publishing, 2026
https://doi.org/10.70593/978-93-7185-126-8_3

DeepScience
Open Access Books

Chapter 3: Real-Time Data Ingestion and Streaming Pipelines

3.1. Introduction

The rise of streaming technologies has spurred the emergence of real-time data ingestion and streaming pipelines, which serve as the foundation for building streaming data platforms. Real-time data ingestion focuses on collecting, validating, and transforming data as it becomes available. A streaming pipeline consists of a series of data transformation steps applied to streams of data—effectively a set of streaming ETL processes. Both concepts are wide in scope and can be addressed along various dimensions, including protocol semantics and guarantees; data serialization formats; support for schema evolution and data quality; pipeline architectures; data transformation and enrichment such as windowing, aggregation, and complex event processing; durable storage; and consistent state management. Real-time data processing is acknowledged as a rapidly maturing field of research and engineering practice.

The supporting foundations and concepts, along with an outline of open challenges, offer an overview of real-time data ingestion and streaming pipelines. The latter can be seen as the foundations of streaming data platforms—semantically richer and more fully featured than their streaming processing framework subsystems. While firms such as Confluent, Databricks, and Streamlio provide specialty products in the commercial space, a wide variety of open-source technologies are also available for building real-time data ingestion and streaming data platforms.

3.2. Fundamental Concepts in Real-Time Data Processing

Streaming processing is qualitatively different from traditional batch processing. In contrast to batch jobs, where the entire input data set is assumed to be available before resource scheduling, and jobs are processed in a single pass with specific completion time, in streaming processing input data are continuously generated and consumed. As

such, latency and consistency aspects are the main qualities to address; throughput is mostly an option. Commonly used terms in streaming or real-time processing are latency, throughput, backpressure as well as event-time, processing-time and data generation-time.

Real-time data velocity, variety, veracity and volume represent the main parameters of systems supporting streaming data. Over the last years many new and mature processing systems focused on these properties offering new paradigms for data processing at scale (additional V – can volume be kept constant whilst velocity increases? – have also been suggested). However, latency achieved has lagged behind original ambition.

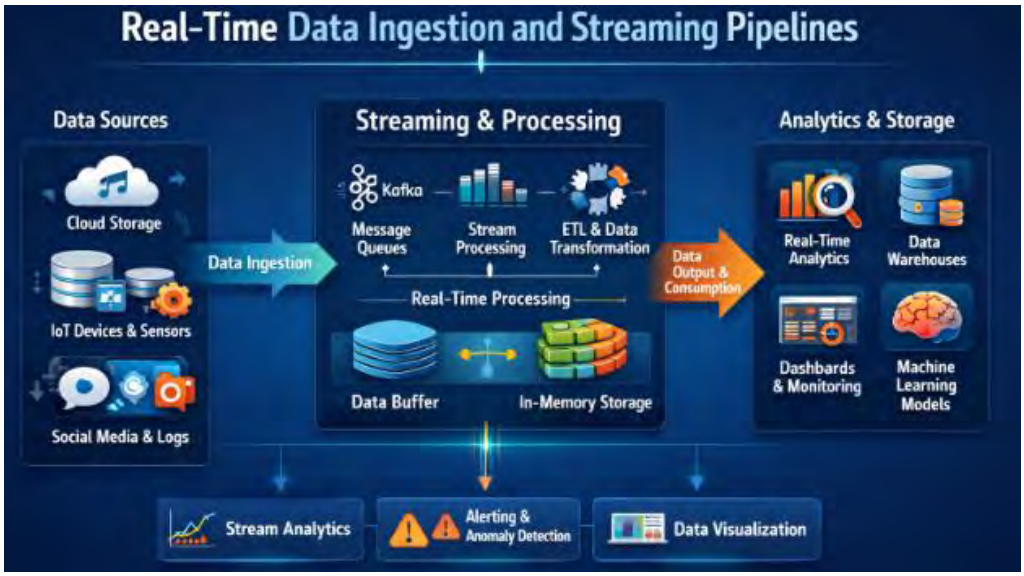

Fig 3.1: Real-time data visualization pipeline overview

3.3. Data Ingestion Methods and Protocols

In a big data ecosystem, data is being generated by different events at a very high speed. Due to the volume and variety of data being generated in a distributed way, there is a requirement of a system that can collect and process data in real-time. Data ingestion is the process of transferring data from various data sources for further processing. As the volume of data being produced every second is enormous, it is difficult to collect all data and analyze it. It should be processed close to the data source.

There can be two types of data ingestion: bulk ingestion and stream ingestion. Bulk ingestion collects data in batches for a specified period of time. It is often a scheduled process that runs at regular time intervals. Stream ingestion collects data as it is generated; hence it introduces low latency. There are different protocols and systems for

ingesting streaming data. Streaming data is transferred from a sending agent to the receiving agent in the form of messages. There should be a communication protocol between the sending and receiving agents, and also a serialization format to represent all messages in a consistent way. A message queue or a streaming platform can be used to transfer the streaming messages between sending and receiving agents. Data ingestion is not only about transferring data; data should be validated, enriched and put in a format understandable to the receiving agent. This requires a schema, and the schema needs to evolve with time. For messages being sent on the wire in the streaming environment, data quality and schema governance must be ensured.

Equation 1: Throughput Rate

Measures how much data your pipeline can process per unit time.

$$Throughput = TotalDataProcessedTime\text{Throughput} = \frac{\text{Total Data Processed}}{\text{Time}} Throughput = TimeTotalDataProcessed$$

Example:
If 500 MB is processed in 10 seconds → Throughput = 50 MB/s

3.3.1. Message Queues and Streaming Platforms

At the lowest level, systems process streams of events, so any such event stream must be ingested somehow. Distributed systems scale naturally by allowing many independent components to operate concurrently. Modern systems must therefore also be able to handle data-velocity and data-variety. Effective natural scaling leads toact with concurrent producers and consumers. Real-time ingestion is usually done using either message queues or dedicated streaming engines; these techniques enable high-throughput paging for data that does not require low latency. The distinction is that message queues use a point-to-point model (exactly one consumer for message) versus a publish/subscribe model (zero or more subscribers). Messages can endure for hours, days, or even longer periods of time, according to how much capacity and cost the deployment can support. Messages can endure from seconds to days but almost no queues provide true buffering capacity since they aim to minimize latency rather than maximize throughput. With sufficient resource provisioning, exactly-once processing and streaming-at-most-once can both be achieved.

Message queues are fundamental and both but do provide some level of durability, ordering, low-latency, or at-least-once processing guarantees. Implementations such as ZeroMQ or NATS aim towards high performance. When no durability is needed, lower-level mechanisms like TCP, Websocket, or MQTT can be used. Publishing messages to

additional sinks, such as a search index, a monitoring system, and long-term storage, is also supported natively by some queues.

3.3.2. Data Serialization Formats

Right before data is ingested into an analytic system, it often undergoes serialization. The chosen serialization format affects the size and speed of the data moving along the pipeline. Furthermore, if the data is written to a persistent or semi-persistent sink, the schema of the data must be set in a way that enables future compatibility. In simple words, if new fields are added or removed from the original schema, it should not break existing consumers who are reading the data in the original format.

JSON is one of the most widely used data representation formats. It is very flexible and easy to understand for humans. Unfortunately, it is also relatively large, since it contains field names for the individual entries. Therefore, it would take a lot of space for high-velocity sensors, for example. The lack of binary format is also seen as a performance issue because it is costly for machines to interpret the format. Schema evolution is difficult as well, since changes (adding or removing fields) are likely to break older consumers unless these consumers are designed to accommodate the changes.

For high-speed streaming use cases, Protocol Buffers and Avro are often preferred. Both solutions are compact binary formats. While Protocol Buffers are designed primarily for ease of use for developers, Avro is specifically designed for compatibility and speed during execution. Avro uses a JSON schema file to describe the data, and when the data is being processed, the actual schema used is extracted from the data and cached. This is done in a way that avoids adding complexity to the consumers, who still see a JSON-like interface. Schema evolution in Avro is very versatile: fields can be deleted, added, or made optional without breaking backward compatibility; and the number of fields can be changed, as long as the backward or forward compatibility rules defined in Avro are respected.

3.3.3. Data Quality and Schema Evolution

Validation, enrichment, and transformation are key steps for ensuring that ingested data meets the quality and completeness requirements of downstream consumers. Data validation, often coupled with simple data enrichment, operates on the incoming data stream itself. More sophisticated transformations are usually performed outside the ingestion process, unless the requirements are very strict and the added complexity warranted.

Data in-flight enrichment is often performed using reference data that is either static (e.g., lookup tables) or periodically refreshed (e.g., GeoIP or pricing data). A governance solution with support for classification and masking can help de-identify sensitive data while improving data quality through the use of reference data.

Many streaming platforms support the notion of a schema registry, where schema evolution for the supported data formats is controlled and schema compatibility is checked before data is written to the stream. Although compatibility tests are usually performed at serialization and deserialization time, additional tests such as triggered or logical validation can be used to ensure data correctness. A schema evolution governance strategy should also consider backward and forward compatibility as well as transformation during reads.

Natively streaming alternatives such as DataHub can facilitate data governance for rapidly changing enterprise data. Their support for multiple data discovery and transformation endpoints allows for declarative transformations and ensures that data in the enterprise is classified, cataloged, known, and governed. Forward transformation through change data capture can be provided via change log topics.

Fig 3.2: Real-time data ingestion and schema evolution

3.4. Streaming Architectures

Streaming architectures have traditionally been categorized as either centralized (a.k.a. monolithic/multi-layer) pipelines or federated ones. The former consists of a unique control mechanism that governs all aspects of the streaming processing. This usually allows for greater throughput: optimizations in resource allocation in the ingestion part of the system optimize the later query stage of the streaming. However, such

centralization comes at a price. Control complexity grows, resembling that of a distributed DBMS. A malfunction of the control unit can adversely affect all real-time applications rather than just one of them; all-encompassing logic can hinder fault isolation; and the overhead of data movement to a placement with lower latency for the user is bigger than if this movement was performed in an ad hoc manner.

In a federated architecture, control is decentralized, with each analytics team developing solutions to its own needs. Real-time and near-real-time queries are executed in a self-service manner. Unlike the centralized approach, where the streaming system itself is characterized by increasing velocity, buffer capacity, and backpressure management, the emphasis from a federated view is put on the streaming applications: velocity, backpressure, and memory buffer limits are defined, and only the efficient operation of each streaming application is controlled. Scalability is achieved by letting independent teams take care of independent applications, avoiding the need to support a single ensemble with increasing velocity. Users typically prefer local data for lower-latency queries; as such, data movement toward the application side is done in a point-to-point manner.

Equation 2: End-to-End Latency

Captures the delay from data generation to final consumption.

$$\begin{aligned} Latency = Tprocessing + Tqueue + Tnetwork\text{Latency} \\ = T_{\text{processing}} + T_{\text{queue}} + T_{\text{network}} Latency \\ = Tprocessing + Tqueue + Tnetwork \end{aligned}$$

Where:

- TprocessingT_{\text{processing}}Tprocessing: time to process data
- TqueueT_{\text{queue}}Tqueue: waiting time in buffers/queues
- TnetworkT_{\text{network}}Tnetwork: transmission delay

3.4.1. Centralized vs. Federated Pipelines

Data pipelines can be categorized as centralized or federated. Control is highly concentrated in the former. A few teams build, maintain, and monitor every pipeline in the platform, enforcing global policies (e.g., naming conventions, licensing requirements), providing testing capabilities to the entire organization, and deciding which features can be used and when. Centralized pipelines facilitate the development of company-wide best practices and principles for data transformation and quality. They also reduce operational complexity by avoiding multiple adaptations to the same data transformation or check.

Centralized pipelines, however, also come with drawbacks. As the company grows, the number of data consumers increases, and the burden on the centralized team grows. When a centralized pipeline grows too large, becoming a monolith, it becomes harder to maintain. To address these problems, organizations may attempt to distribute the pipeline-building responsibilities across multiple teams. Although this often alleviates resource pressure, it usually increases operational complexity, as every team will require a certain level of expertise to build and maintain their own pipelines, all while adhering to the guidelines defined by the centralized team.

3.4.2. Lambda, Kappa, and Beyond

Several approaches combine both processing methods beyond the simple use of a messaging queue. The Lambda architecture proposed by Nathan Marz in 2011 takes a batch processing system and adds a stream processing layer for low-latency needs. The result is a system that can handle ultra-low latency requirements while still offering high throughput for extremely large datasets that might take years to process using batch processing alone. However, Lambda architecture comes with maintenance challenges, since it contains two parallel paths of processing with very different implementations. Implementing both data processing layers normally requires a high level of expertise, and any failure on the stream layer would force a fallback to the batch layer, which is not optimal while offering normal operation. Nevertheless, the Lambda architecture has been instrumental in the adoption of stream processing even for teams that lack streaming expertise.

A simpler architecture is the Kappa architecture proposed by Jay Kreps in 2014, in which ingestion is done through a message queue or streaming platform, and downstream processing is done with streaming processing systems exclusively for all data. The Kappa architecture does not necessarily come without its own issues: while the stream processing implementation is generally much easier to maintain than a Lambda-specific implementation, an additional level of complexity must be assumed when implementing the batch processing, since it has to be user-defined and typically implemented only when needed. Nevertheless, through this architecture, Kreps advocates a unification of both paradigms targeted by Lambda, which should focus on the specific needs of each layer and workflow instead of being hardcoded into the architecture. More recently, other architectures such as Batching Again (for processing of lagged data) and the Delta architecture (for supporting temporal analysis use cases on streaming pipelines) have been introduced, although they currently lack the maturity and mainstream adoption of Lambda and Kappa architectures.

3.5. Data Transformation and Enrichment

Data Transformation and Enrichment

Data transformation is a key step in real-time data ingestion and streaming pipelines. The ingested data may need to be reshaped or enriched before it can be consumed. One of the transformation capabilities is known as windowing and aggregation, which is often required for metrics and business intelligence (BI) dashboards. Data may also need to be correlated as part of complex event processing. Effects and examples are described in the two sections that follow.

Data Enrichment

When incoming data lacks sufficient context for some downstream consumer, real-time data ingestion and streaming pipelines can often fill those gaps with one or more enrichment steps. The enrichment step(s) can access other systems or datasets—such as reference data and lookup tables—where additional information is available. The enriched data has an appropriate schema for the downstream consumers, which may be dashboards, reporting tools, data stores, operational systems, or machine learning models. An online store or database is often used for the reference data because random reads of small datasets require low-latency access. Caching approaches can further reduce datadir delays.

3.5.1. Windowing and Aggregation

Specifying the structure of the real-time processing flow, windows divide the incoming data stream into discrete segments with defined start and end boundaries. These windows are necessary for associating records with the desired time intervals, enabling sequential operations such as aggregation and counting, which generally require a complete data set. The window can be tumbling, sliding, or session-based, and it is equipped with a trigger that specifies when the operation on the windowed records should be performed. Late arrival handling is another important aspect of window processing.

Typically, a tumbling window slides along the datapath, partitioning the incoming stream into non-overlapping segments of fixed size (for example, 1 min records for the period between 10:00 and 10:30). Although all windowed streams are logically independent, the span of a tumbling window operation is the duration of the window, and the triggering policy for an operation within such a window should be set to fire at regular intervals equal to the window span. Unlike batch systems, which can assume input completeness, streaming systems might need to deal with late-arriving data. Common practice is to provide a separate late-arriving stream that is processed after the main windowing function and triggers the relevant downstream functions. An alternative

mechanism augments the windowing function by associating late-arriving data with a separate late-output location.

Equation 3: Backpressure Condition

Defines when the system becomes overloaded.

$$IngestionRate > ProcessingRate \text{Ingestion Rate} > \text{Processing Rate} IngestionRate > ProcessingRate$$

If:

Rin>RoutR_{in} > R_{out}Rin>Rout

Then:

- Queue size grows
- Latency increases
- System may fail without scaling or throttling

3.5.2. Complex Event Processing

Event streams are often more than a sequence of unstructured events; they may reveal patterns or structured compound events. An event—an occurrence that changes a system's state and is worth recording—may be defined as an observable change. When many events are occurring or changes in state can be expressed as mathematical functions, processing can best be accomplished with declarative queries.

A complex event consists of a collection of basic events that belong together, such as people being together, call sequences, or IP packets sent from the same source to the same destination. Event correlation detects relationships between separate, unrelated events; for instance, using a home security event (a door closing) and a motion detection event (motion detected) to determine whether an intruder is in a house. A correlation window is the period during which correlation is computed.

Unlike basic event processing, which is typically stateless or uses only past context, complex event processing often requires the evaluation of state-maintaining patterns over time. A process that computes the set of all sessions of a web server may be naturally implemented using a finite state machine. A stream of session events consists of information captured during an application server session, such as the session identifier, the user identifier, the creation time, the start time of the first request, the end time of the last request, and the total number of requests.

Stateful event processing can be treated as the challenge of making the pattern deterministic for both the input and the output. Stateful processing mechanisms are sometimes explicitly provided by stream processing systems. Applications that fit this paradigm include detecting users logged into the service at the same time and complex alert generation (for example, when a user clicks on a link but does not proceed to the next page within a given time).

3.6. Storage and State Management

Data Storage and State Management

Durable storage serves as the ultimate sink that retains aggregated and/or freshly enriched data, and can also feed analysis and serving systems. Such durable storage may be read from and written to at any time and by any component regardless of their roles in the broader system. As induced, massive amounts of immutable data generated by older applications can interact gracefully with data produced at streaming velocity within new services. Key storage types are summarized using the typical distinction between data lakes and data warehouses. Beyond discussion of these respective systems, the data lakehouse concept is briefly introduced in the context of durability.

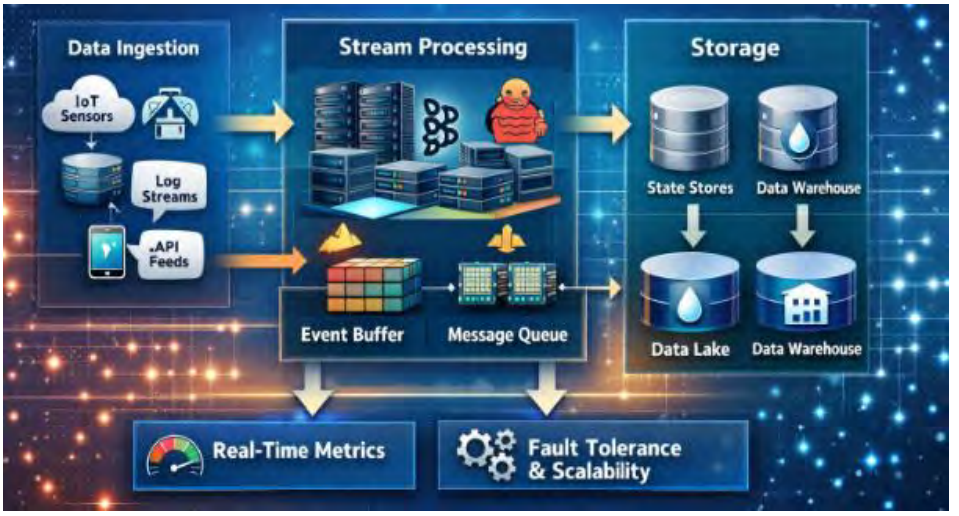

Fig 3.3: Real-time data processing and analytics flow

Completing the storage narrative, an examination of state management follows. Trackable application state may be stored in durable form, as needed to guarantee exactly-once processing semantics. It may also be cached in memory for efficiency in cases of exactly-once, at-least-once, or potentially-lost processing. The snapshotting/storage mechanism called changelog, which fully decouples state storage from the application logic, is detailed along with the recovery procedures that it enables.

Finally, checkpoints and relevant addition mechanisms that confer reliability to streaming applications are also explained.

3.6.1. Durable Storage Options

Different forms of durable storage are required for data flows at various stages. For streaming ingestion, a message queue or streaming system with sufficient replay guarantees suffices. After event consumption and transformation, storage that enables analytical querying becomes necessary. Stream processing engines can ingest the results back into such storage or write directly. Traditional data lakes, data warehouses, and more recent data lakehouses all provide durable storage for analytically processed data, but durability guarantees vary.

A data lake's primary function is ingesting and retaining data quickly without compliance with a particular schema. Ingesting raw data rapidly is crucial for enabling data exploration and producing relaxed-scope short-lived analytics. Read operations are often performed by specialized components that can efficiently scan non-structured data and require little or no data preparation.

3.6.2. State Backends and Exactly-Once Processing

Reliable handling of mutable state is crucial for fault tolerance and correctness of streaming pipelines. Additionally, stateful streaming transformations often necessitate additional middleware to achieve fault tolerance and exactly-once processing guarantees. Stateful streams can be scaled across multiple processors by providing a separate state for each key. Stateful transformations typically require persistent, durable state access to withstand process failures. The memory of a single processor is fragile and must be backed by an external storage, backed by in-memory databases, disk persistence, and distributed databases. To ensure that operations on stateful transformations can be made fault-tolerant, the information flow of stateful operations is often extended with a dedicated changelog stream. In situations with high throughput data distribution, a distributed changelog stream is typically used in conjunction with a distributed key-value store to ensure that both state read and write operations are fault-tolerant.

The changelog serves as an external storage for the mutable state of the respective processor. All write operations are no longer performed only on the memory of the processor, but also appended to the changelog. The changelog captures every change that happens to a key in the state and enables the state to be recovered after a sudden process failure – this opens up the potential for implementing exactly-once processing

semantics. Exactly-once semantics guarantees that the processing of each event will be reflected in the external systems exactly once and without any duplicates even in the case of failures. It is achieved in conjunction with stateful operations that introduce an arbitrary writable state. The application can be viewed conceptually as a transaction system with two phases: the first phase is any kind of preprocessing that writes to intermediate topics and may emit multiple records in case of late events. The second phase reads from the intermediate topics in a consistency-preserving manner and writes the final result to exactly-once sinks.

3.7. Conclusion

Real-time data ingestion and streaming pipelines provide novel approaches to extracting, transforming, and loading data that cater to the growing demands of modern data ecosystems, which require applications to act earlier and make decisions faster. Fundamental aspects of streaming systems, ranging from latency and throughput to backpressure and event-time semantics, affect design choices at different levels, from individual components to the whole data pipeline architecture; fulfilling additional requirements such as reliability, data quality, schema evolution, and data governance introduces further considerations. Features of numerous data ingestion methods—from messaging queues and serialization protocols to data quality practices—serve as the basis for a comparison of centralized and federated streaming pipelines, as well as a discussion of the Lambda and Kappa architectures. Techniques for data transformation and enrichment—such as windowing and complex event processing—are addressed in preparation for the analysis of data storage and state management.

The growing volume and velocity of data, coupled with the need for more varied, multimodal, multimedium data, exceed the capabilities of batch data pipelines and necessitate new architectures and technologies that can address these challenges. Traditional batch-oriented systems are no longer sufficient, as evidenced by increasing requests for more than one release a month on products such as Google Search and Amazon. Indeed, the services offered by these large technology companies—such as Google Search, Netflix, and Amazon Alexa—may serve as a source of inspiration, albeit at a different scale. Real-time data ingestion and streaming pipelines provide an alternative approach to extracting, transforming, and loading data for real-time applications. Nevertheless, many challenges remain unresolved, including support for complex event processing, standardized workflows, differing quality and velocity requirements, and improved interoperability with other components of a streaming environment.

3.7.1. Future Directions

Real-time data ingestion and streaming systems have matured drastically over recent years. Streaming-systems have proliferated and gained traction in cloud-native data platforms; publish-and-subscribe messaging systems have gained widespread acceptance as the canonical way of sending messages across decoupled systems. Despite these advances, there remain multiple opportunities for further improvement.

Firstly, while these architectures can scale data-processing workloads, often dynamically, these scaling capabilities are not easily exploited for data-ingestion workloads. Regarding ingestion, support for decoupled data ingestion is nascent, without clear separation of ingestion responsibility from processing. Operational support for more intuitive asset-based governance and reliability guarantees remains limited. Standard formats such as Apache Arrow can provide more compact message encodings, or ease the challenge of interoperability. However, beyond a common serialization format, storage systems still do not expose clear interoperability contracts that consumers can rely upon. Finally, core infrastructure such as global state, moderation or session coordination services remains kingly implemented within domain-specific pipelines.

Emerging technologies such as knowledge graphs, Spatial Data Management, Reverse ETL, and Freemium data-ingestion services can lay novel foundations for these data-ingestion workloads and enable richer data-settings, models and use-cases..

References

Stonebraker, M., et al. (2023). Data lakes vs warehouses revisited. *Communications of the ACM*.

Kolla, S. H., Inala, R., & Kumar, M. V. K. (2026). Secure RAG Architectures with Small Language Models for Governance-Aligned LLM Deployment in Enterprise Service Management Platforms. International Journal of Economic Practices and Theories, 2026, 166-179.

Abadi, D. (2023). Cloud-native data management systems. *IEEE Data Engineering Bulletin*.

Segireddy, A. R. (2022). Terraform and Ansible in Building Resilient Cloud-Native Payment Architectures. International Journal of Intelligent Systems and Applications in Engineering, 10, 444-455.

Nargesian, F., et al. (2023). Data discovery in modern data platforms. *VLDB Conference*.

Nagabhyru, K. C., & Kumar, M. V. K. (2025). Generative AI Meets Data Engineering: Automating Code, Query Generation, And Data Insights in Large Scale Enterprises. Query Generation, And Data Insights in Large Scale Enterprises (April 23, 2025).

Polyzotis, N., et al. (2023). Data validation for ML pipelines. *CIDR Conference*.

Jagtap, S., Inala, R., Venu, M., & Divya, T. V. (2025, October). Large-Scale Crowd Flow Prediction Using Temporal Convolutional Network with Spatio-Temporal Attention. In 2025

International Conference on Communication, Computer, and Information Technology (IC3IT) (pp. 1-6). IEEE.

Zaharia, M., et al. (2024). Lakehouse architecture for modern data platforms. *Databricks Whitepaper*.

Danghi, P. S., Maniraj, K., Jain, P., Adilakshmi, K., Garapati, R. S., & Jain, S. K. (2025). Artificial Intelligence Based Energy Optimization Framework for Wireless Sensor Networks. In 2025 IEEE 5th International Conference on ICT in Business Industry & Government (ICTBIG) (pp. 1–6). IEEE. https://doi.org/10.1109/ictbig68706.2025.11323860

Holland, S., Hosny, A., Newman, S., Joseph, J., & Chmielinski, K. (2020). The dataset nutrition label. Data Protection and Privacy, 5, 1–26.

Deep Learning-Driven Optimization of ISO 20022 Protocol Stacks for Secure Cross-Border Messaging. (2024). MSW Management Journal, 34(2), 1545-1554.

Sculley, D., Snoek, J., Wiltschko, A., & Rahimi, A. (2018). Winner's curse? On pace, progress, and empirical rigor. In International Conference on Learning Representations Workshop.

Keerthi Amistapuram. (2023). Privacy-Preserving Machine Learning Models for Sensitive Customer Data in Insurance Systems. Educational Administration: Theory and Practice, 29(4), 5950–5958. https://doi.org/10.53555/kuey.v29i4.10965

Schelter, S., Böse, J.-H., Kirschnick, J., Klein, T., & Seufert, S. (2017). Automatically tracking metadata and provenance of machine learning experiments. In Proceedings of Machine Learning Systems Workshop at NIPS.

Lebcir, I., Shah, A., Nagubandi, A. R., Dhoke, S. M., & Mishra, M. K. (2025). FinTech and Financial Inclusion in Emerging Economies: An Empirical Assessment. Advances in Consumer Research, 2(6).

Buneman, P., Khanna, S., & Tan, W.-C. (2001). Why and where: A characterization of data provenance. In Proceedings of the 8th International Conference on Database Theory (pp. 316–330).

Cui, Y., Widom, J., & Wiener, J. L. (2000). Tracing the lineage of view data in a warehousing environment. ACM Transactions on Database Systems, 25(2), 179–227.

Cheney, J., Chiticariu, L., & Tan, W.-C. (2009). Provenance in databases: Why, how, and where. Foundations and Trends in Databases, 1(4), 379–474.

Simmhan, Y. L., Plale, B., & Gannon, D. (2005). A survey of data provenance in e-science. SIGMOD Record, 34(3), 31–36.

Herschel, M., Diestelkämper, R., & Ben Lahmar, H. (2017). A survey on provenance: What for? What form? What from? The VLDB Journal, 26(6), 881–906.

Bandi, V. D. V. K. AI-Based Anomaly Detection Frameworks in Distributed Enterprise Data Systems.

Freire, J., Koop, D., Santos, E., & Silva, C. T. (2008). Provenance for computational tasks: A survey. Computing in Science & Engineering, 10(3), 11–21.

Gadi, A. L. , Gadi, A. L. Kannan, S. , Kannan, S. Nandan, B. P. , Nandan, B. P. Komaragiri, V. B. , & Komaragiri, V. B. (2021). Advanced Computational Technologies in Vehicle Production, Digital Connectivity, and Sustainable Transportation: Innovations in Intelligent Systems, Eco-Friendly Manufacturing, and Financial Optimization. Universal Journal of Finance and Economics, 1(1), 87-100. https://doi.org/10.31586/ujfe.2021.1296

Davidson, S. B., & Freire, J. (2008). Provenance and scientific workflows: Challenges and opportunities. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (pp. 1345–1350).

Davuluri, P. N. (2020). Event-Driven Architectures for Real-Time Regulatory Monitoring in Global Banking.

Moreau, L., Clifford, B., Freire, J., Futrelle, J., et al. (2011). The Open Provenance Model core specification (v1.1). Future Generation Computer Systems, 27(6), 743–756.

Missier, P., Soiland-Reyes, S., Owen, S., Tan, W.-C., et al. (2013). PROV-DM: The PROV data model. W3C Recommendation.

Curino, C., Moon, H. J., Tanca, L., & Zaniolo, C. (2013). Schema evolution in database systems: A survey. Data & Knowledge Engineering, 85, 1–24.

Bernstein, P. A., & Haas, L. M. (2008). Information integration in the enterprise. Communications of the ACM, 51(9), 72–79.

Yandamuri, U. S. (2026). AI-Enabled Workflow Automation and Predictive Analytics for Enterprise Operations Management. Management, 3(1), 15-24.

Lenzerini, M. (2002). Data integration: A theoretical perspective. In Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (pp. 233–246).