Reference Architecture Brief: Stream Data Processing Architecture

Published 30 November 2023 - ID G00802493 - 26 min read

By Analyst(s): Prasad Pore

Initiatives: Data Management Solutions for Technical Professionals

Stream data processing architecture is used to process in-motion data to meet organizational needs for continuous intelligence and real-time analytics. This research helps technical professionals implement stream data processing architectures for data integration, event processing and analytics.

Architecture Brief

Digital transformation is driving the need for analysis of what is happening now, not what happened yesterday or last month. With the increase in streaming data, streaming sources (e.g., clickstreams, IoT devices) and analytical use cases, the importance of streaming data processing has grown enormously in recent years because it provides a competitive advantage by reducing the time gap between data capture and its analysis. Some insights are more valuable immediately after an event occurs, and their value diminishes very rapidly with time. The ability to quickly make decisions and act on data in real time is becoming a key element of SLAs across multiple domains, from the Internet of Things (IoT), finance and cybersecurity to retail.

Compared with the batch-mode, “data at rest” practices of traditional systems, the capability of streaming data applications to process and analyze data in motion has become a key differentiator for organizations. Enterprises are therefore embracing streaming architectures coupled with modern data processing engines and frameworks to create streaming data applications. This approach is referred to by many names: real-time analytics, streaming analytics, complex-event processing (CEP), real-time streaming analytics, and event processing. Although these terms historically had differences, modern frameworks have converged under the term “stream processing.”

Fundamentally, stream data processing architecture is a type of event-driven architecture (EDA) focused on processing and analyzing in-motion data in native streaming mode (processing events one at a time) or microbatch mode (processing events in small batches). It allows for collection, integration and analysis of data in real time, as data is produced, without requiring data to be stored prior to analysis.

The basic attributes of stream processing are the following:

- Streams are unbounded, nonending flows of events — for example, from sensors, logs and financial trading.
- Stream processing supports faster decision making than what is possible with traditional batch-based data processing or ad hoc queries.
- Streaming systems flip the model of data processing from “store before processing” to “process before storing.”
- Stream processing provides businesses with a method for extracting strategic value from data in motion similar to the way traditional analytics tools operate on data at rest, such as in fraud detection.
- Stream processing enables action based on an analysis of a series of events that have happened within a predefined window of time, such as the number of unique visitors to a website.

Architecture Use Cases

Stream data processing architecture is an optimal fit for:

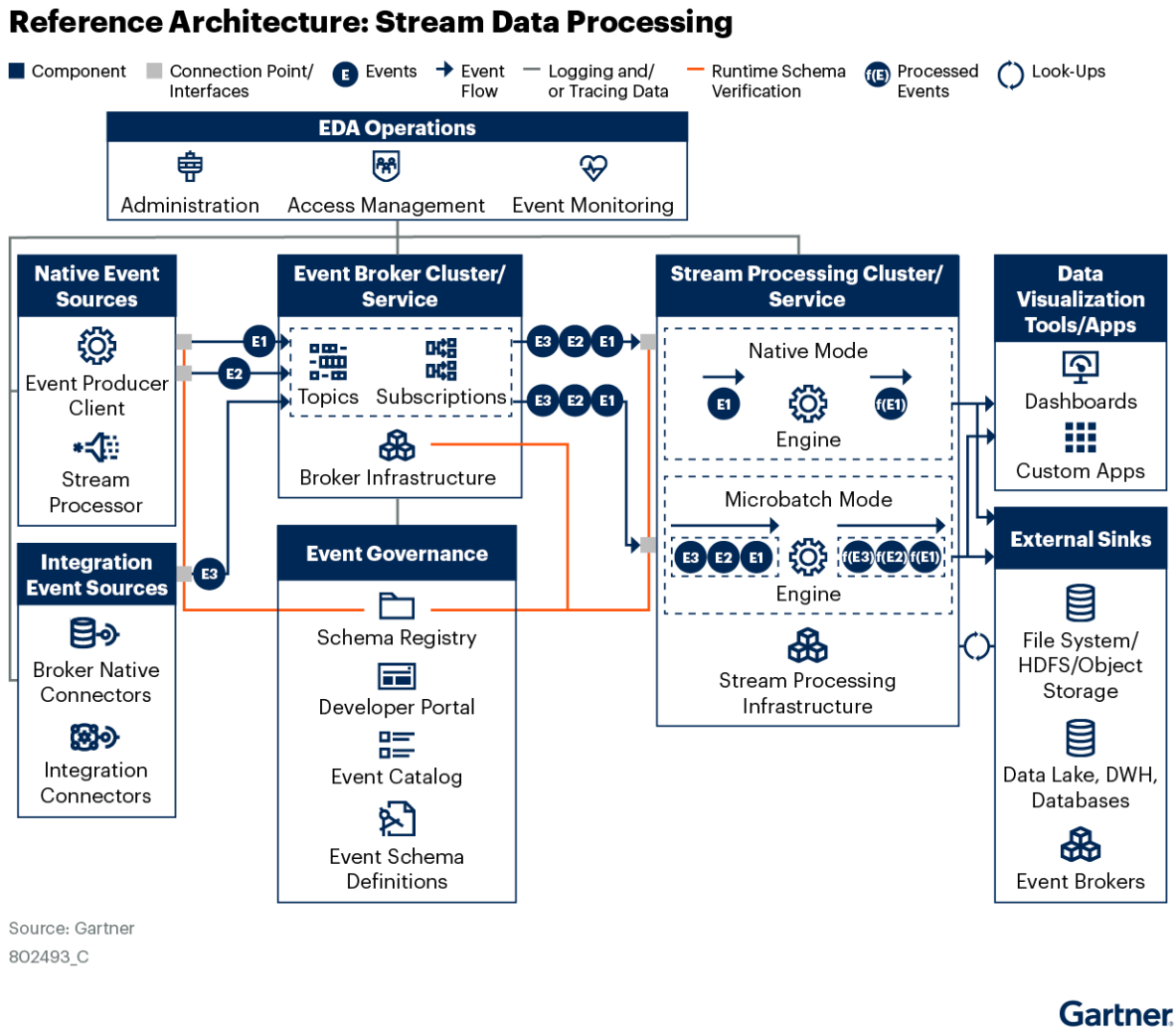
- High-frequency event streams that need to be processed within very strict SLAs — for example, in fraud detection, network monitoring and operational intelligence
- Performing complex analytics or complex event processing (CEP) on each event independently or on groups of events within a window to detect temporal patterns
- Performing continuous data ingestion, filtering, transformation, linking, correlation, enrichment, online feature creation, text data processing for sentiment analytics and massive aggregation to reduce data volumes
- Replicating data at high volume and high velocity, in real time, from transactional databases to analytical databases, data lakes or lakehouses using a change-data-capture (CDC) mechanism

- Edge computing application architectures that require in-stream analytic processing and inferencing

Architecture Diagram

Stream data processing architecture contains multiple components that communicate using messages that represent events. These components include event sources (publisher, producer), event broker, stream processor, visualization tools and external sinks. These components have integrated operations and governance capabilities, as shown in .

Figure 1: Reference Architecture: Stream Data Processing



The reference architecture for stream data processing focuses on the infrastructure to support scalable and managed publish-subscribe communications and stream processing. The architecture includes common classes of event sources, visualization tools and external sinks used in an application context. The stream data processing architecture shares many components with, and extends, the event-driven application architecture defined in Reference Architecture Brief: Event-Driven Application Architecture.

The reference architecture shows three types of connectivity:

- **Event flow:** The flow of events published by event sources to topics managed by the event broker and delivered to stream processing cluster/service or external sinks directly.
- **Runtime schema verification:** The connectivity of source and sink clients (and, optionally, event brokers) to a schema registry to enable runtime verification of event message format against defined and published message schemas.
- **Logging and/or tracing data:** The connectivity between components of the architecture to collect data to support observation and management of event-driven applications.

The architecture is made up of eight core components:

- **Native event sources:** The software and services that can publish events directly to your event broker of choice
- **Integration event sources:** The configurable connectors for collecting or ingesting events from existing software, services and data sources
- **Event broker cluster/service:** The core event-driven middleware component responsible for receiving published events from sources and delivering them (via topics and/or subscription configuration) to subscribed event topics
- **Stream processing cluster/service:** The core software and service that can subscribe directly to the event broker of choice to consume and process streams of events, in native mode and microbatch mode
- **Data visualization tools/apps:** The software applications and services that can visualize streaming data on a dashboard or a website in real time

- **External sinks:** The data storage technologies/DBMS to receive real-time data from the stream data processing service for long-term persistence, or event brokers to receive events from the stream data processing service for downstream real-time applications such as machine learning (ML) inferencing
- **EDA operations:** The control plane capabilities needed to configure, secure and monitor your event-driven architecture
- **Event governance:** The tools for storing, distributing and managing the life cycle of metadata about your EDA, including event schemas, published (shared) topics and documentation

Architecture Capabilities and Components

The following subsections provide an overview of each component of the reference architecture.

Native Event Sources

Native event sources are software and services that can consume events directly from your event broker of choice. This means that they have been designed, built and can be configured to connect directly to the event broker using a supported protocol or its API. Consuming an event triggers some processing that uses information from the event message to do things like effect state changes, perform business logic, notify users or even generate additional events. This component includes two categories of subcomponent:

- An **event consumer** client is an application or service that consumes events from the event broker using a broker-specific software development kit (SDK)/library or a third-party SDK/library that supports the messaging protocol of the broker.
- A **stream processor** is an application or service that uses a stream data processing framework to receive events from one or more topics managed by the event broker. It implements stream-oriented logic to filter, join, and aggregate and analyze events in the context of one or more ordered streams. The output from a stream processor is often a stream of “complex” or derived events that are published to an event broker topic (output stream). However, depending on the implementation framework or platform, the output may be delivered to other data repositories.

Example Technologies

- Event producer clients:
  - Apache Kafka client (Java)
  - Apache Qpid
  - Java Message Service (JMS)
  - RabbitMQ Java Client
  - Solace PubSub+ Messaging APIs
  - Spring messaging
- Stream processors:
  - Akka Streams
  - Apache Flink
  - Apache Kafka Streams API
  - Apache Spark
  - Esper (Java)/NEsper (.NET)
  - Spring Cloud Stream

Key Characteristics

Native event sources have the following characteristics:

- Applications or services that connect directly to the event broker to consume events from a topic
- Should use the protocols and APIs of the broker, often via an SDK or library
- Typically software developed and deployed by the team delivering an event-driven application
- An application or component that is a native event source; can also be a native event source (subscriber) and have other functions

Related Research

- Essential Patterns for Event-Driven and Streaming Architectures

Integration Event Sources

Integration event sources use configurable connectors to collect or ingest events from existing software, services and data sources and publish them to topics managed by an event broker. Integration source connectors also provide data transformation features to allow source data structures to be mapped to event payload schemas. This component is focused on integrating existing systems into your event-driven application architecture as an event source.

This component includes two categories of subcomponent:

- **Broker native connectors** are provided by and specific to the event broker used in your architecture. They may operate within the broker infrastructure or be deployed independently within a hosting environment that allows configured connector instances to scale independently of event broker capacity and be deployed to isolate failure from the broker and other connectors.
- **Integration connectors** are components provided by a discrete integration or automation platform that allows events to be captured from other systems (processes or data sources) and be published to topics managed by the event broker. These connectors are typically either broker-specific or protocol-specific. Broker-specific connectors abstract and encapsulate the proprietary protocols or SDKs to communicate with the broker. Protocol-specific connectors implement, abstract and encapsulate the implementation of open protocols, allowing them to be configured to communicate with any broker that implements that protocol.

Both categories of connectors can be used in conjunction with other technologies to detect and capture events. For example, they may be used in combination with database change data capture (CDC) or application-specific connectors that can subscribe to webhooks or other event-based APIs provided by SaaS applications.

Example Technologies

- Broker native connectors:
  - Amazon EventBridge integrations
  - Apache Kafka Connect
  - Apache Pulsar IO
  - Azure Event Grid Partner Events
  - Solace PubSub+ Connector
- Integration connectors:
  - Boomi — connectors include RabbitMQ connector, Azure Service Bus connector, Apache Kafka Connect, JMS Connector, Solace PubSub+ connector
  - IBM App Connect — Google Cloud Pub/Sub connector, IBM WebSphere MQ connector, Kafka connector, IBM Event Streams connector
  - Microsoft Azure Logic Apps — Azure Service Bus connector, Azure Event Grid connector, Azure Event Hubs connector
  - MuleSoft — Apache Kafka Connect, AMQP Connector, JMS Connector
  - Workato — Apache Kafka Connect, MuleSoft JMS connector

Key Characteristics

Integration event sources have the following characteristics:

- Connect directly to the event broker to publish events captured from an external application, service or data store
- Use the protocols and APIs of the broker, often via an SDK or library
- Typically software developed and deployed by the team delivering an event-driven application
- An application or component that is a native event source; can also be a native event sink (subscriber) and have other functions

Related Research

- Essential Patterns for Event-Driven and Streaming Architectures

Event Broker Cluster/Service

An event broker is the core middleware component in event-driven architecture. It is responsible for receiving published events from sources and delivering them (via topics and/or subscription configuration) to subscribed event sinks. In an event-driven application architecture, reliability of the application is dependent upon a reliable event broker deployment. Therefore, this component is deployed as a resilient cluster of broker instances when self-management and resilience are provided by the service provider.

Event brokers come in many forms. The three most common are:

- **Log-oriented brokers** process topics as append-only logs.
- **Queue-oriented brokers** use a queue-based architecture internally and are typically capable of supporting queue-based communication patterns in addition to publish-subscribe.
- **Subscription-oriented brokers are** typically a cloud-native service and distribute events based on subscription rules.

Event brokers expose one or more APIs or protocols to allow client processes to publish events and/or subscribe to receive them, and can include features to support resilient delivery, message batching, message or batch acknowledgment, and event replay. Some brokers use proprietary protocols, while others use open protocols that can be supported by multiple broker implementations. The open protocols most commonly supported include Advanced Message Queuing Protocol (AMQP; version 0.9.1 or version 1.0), Message Queuing Telemetry Transport (MQTT; version 3.1.1 or version 5.0) and Apache Kafka protocol. Webhooks and websockets can also be used to support lightweight publish-subscribe communication in web-based architectures.

Events published to the broker may be persisted to durable storage and/or distributed across multiple broker nodes in the cluster to provide resilience and recoverability in the event of a failure. The event broker may use an internal storage mechanism, or delegate some or all message persistence to an external store to balance event throughput, latency, reliability and cost of storage.

Authentication and authorization of client connections from event sources and sinks are typically handled by the broker under the configuration and control of the EDA operations component. Most brokers support topic-level authorization to control access for publication of events or creation of subscriptions. Some brokers, including most subscription-oriented brokers, use an outbound “push-based” communication model to deliver events based on subscription rules. This means that the broker must be authorized to connect to the subscribed (sink) resource.

Example Technologies

Examples of event broker cluster/service example include:

- **Log-oriented brokers:**
  - Amazon Kinesis
  - Apache Kafka
  - Azure Event Hubs
- **Queue-oriented brokers:**
  - Apache ActiveMQ Artemis
  - Azure Service Bus
  - Google Cloud Pub/Sub
  - IBM MQ
  - RabbitMQ
  - Solace PubSub+ Platform
- **Subscription-oriented brokers:**
  - Amazon EventBridge
  - Azure Event Grid
  - Google Cloud Eventarc

Key Characteristics

The event broker cluster/service component has the following characteristics:

- It must support the publish-subscribe communication pattern to support minimally coupled relationships between application services (broker clients).
- It must be deployed in a resilient and recoverable configuration to meet application requirements.
- Event (message) durability, ordering and quality of service (QoS) must be consistent with your intended application requirements.
- Client connectivity is dictated by the protocol and API support of the event broker cluster/service component. Options include both proprietary and open-network protocols.
- Event broker clusters/services for production workloads must support upgrade and patching processes that maintain high availability and predictable performance.

Related Research

- Choosing Event Brokers: The Foundation of Your Event-Driven Architecture
- Select the Right Event Broker Technologies for Your Event-Driven Architecture Use Cases

Stream Data Processing Cluster/Service

Stream data processing uses software or services, generally based on a stream processing framework, to receive events from one or more streaming sources via event broker. The framework enables complex event processing and delivers results to downstream data visualization tools, or to external sinks for the persistence. Stream data processors are designed to be decoupled from their data sources and messaging components. The stream data processor handles continuous processing of the data streams to reliably prepare, enrich and process the stream data for downstream consumption. Via connectors or APIs, stream data processors can also read data from databases, data lakes or flat files, which is very helpful for lookup while implementing enrichment transformation on an event.

The output from a stream data processor is often a stream of complex, aggregated or derived events that are pushed for visualizing or persisting in long-term storage. In some cases, it is also pushed to event broker topics for downstream applications such as real-time ML inference or push notification for end-user devices.

The streaming engine is the most complex layer in a streaming application. It supports procedural or declarative language (e.g., C/C++, Java, Python or KSQL) for transforming, aggregating, filtering, joining and retrieving information from single or multiple streams of data, and provides stateful processing for long-running or continuous computations. It provides fault tolerance and recoverability through checkpointing and "savepointing" capabilities.

This component includes two modes of stream processing:

- **Native streaming mode:** Events/records are processed as soon as they arrive in unbounded fashion. The advantage of native streaming is its expressiveness at the event granularity. There is no artificial abstraction on top of an event or record. In this architecture, records are processed on arrival, so they have lower latency than the microbatch mode. Some vendors provide microbatching capabilities on top on native streaming with the help of dataset or table APIs.
- **Microbatch mode:** Small batches are created from incoming records/events according to a predefined time constant or number of records or sessions. Compared to native mode, this boundedness of batch interval inevitably limits system expressiveness and adds latency, but provides higher throughput.

Some of the native stream processing technologies, like Apache Flink, support both native streaming and microbatching. Unified batch and streaming SDKs like Apache Beam delegate workloads to separate runtimes depending on the requirement of bounded and unbounded stream processing.

Example Technologies

Stream processor examples include:

- **Stream processing engines and frameworks:**
  - Apache Apex, Apache Flink, Apache Samza, Apache Spark Structured Streaming, Apache Storm
  - MillWheel

- **Stream processing libraries:**
  - Akka Streams (Scala)
  - Apache Kafka Streams
  - IBM Streams (Python, Java, Scala)
  - Java Streams
  - Spring Cloud Data Flow
- **Stream processing managed services:**
  - Amazon Managed Service for Apache Flink
  - Azure Stream Analytics, Azure Data Explorer
  - Confluent Cloud
  - Google Dataflow

Key Characteristics

Stream processing services have the following characteristics:

- Process data in real time or near real time to allow support for mission-critical real-time business use cases.
- Low latency, scalability and fault tolerance.
- Should support event time processing to handle out-of-order events and ensure accurate results for real-time applications.
- Should support windowing, session and state management capabilities to analyze data in specific time windows or user sessions. To ensure that, watermarking and checkpointing mechanisms are required in stream processing services.
- Able to handle back pressure and schema evolution.

Related Research

- Stream Processing: The New Data Processing Paradigm

- Streaming Analytics in the Cloud: A Comparative Analysis of Amazon, Microsoft and Google

Data Visualization Tools/Apps

Data visualization tools present information in a schematic form (using charts, plots, maps, interactive real-time dashboards and other mediums) to represent statistics, facts or figures. These tools are used widely for business intelligence (BI), exploratory data analysis and monitoring use cases. While early products focused on presenting static data and static visuals, many tools can now dynamically visualize streaming/in-motion data and provide an interactive user experience for slicing, dicing, drill down, roll up, filtering and more operations.

Generally, visualization tools connect to one or more data stores, query the data, and display the results on the dashboard using cached or persisted datasets mapped to source table schema. In the context of streaming analytics, these tools are integrated with the stream processing service using a push mechanism. The stream processing service pushes the data (generally in JSON format) using HTTP post request, to the REST API provided by visualization tools.

These tools also use the concept of the streaming dataset, which has its own schema mapped to data received from a stream processing service. Once received, the streaming data is merged into a streaming dataset, which these tools either cache in memory or store in a database. The data can then be displayed in the format designed by BI developers. The same mechanism can be used to create a custom webpage, web service or app to display real-time streaming data. Some stream processing services, like Azure Data Explorer, have inbuilt capabilities to display basic visualizations.

Example Technologies

Visualization tool/app examples include:

- **Dashboard tools:**
  - Grafana
  - Looker
  - Microsoft PowerBI
  - QlikView
  - Tableau
- **Custom apps:**
  - Any software or web application that can integrate with a stream processing service and visualize the data; for example, websites with charts using D3.js.

Key Characteristics

Visualization tools/apps have the following characteristics:

- Present data visually in various forms for analysis purposes
- May provide capabilities to implement tasks such as collecting data, transforming data, creating visuals with graphs/charts, composing dashboards and stories from individual visuals, analyzing data, and sharing insights
- May provide APIs/web services to embed visualizations in other apps

Related Research

- Magic Quadrant for Analytics and Business Intelligence Platforms
- Critical Capabilities for Analytics and Business Intelligence Platforms
- Solution Criteria for Analytics and Business Intelligence Platforms
- Delivering Analytics Insights Through Practical Dashboard Design

External Sinks

External sinks are software and services that can persist processed stream data for long-term or temporary purposes. They receive data pushed by stream processors using available interfaces of technologies used as a sink. These sinks are built to harvest or stage the data for downstream applications where it can be remodeled as per requirements. Streaming data applications can have one or more external sinks depending on the use cases.This component includes three main categories:

- A **software-defined distributed storage service,** such as cloud object storage or Hadoop Distributed File System (HDFS), that provides a basic mechanism to store/persist the data. Generally, it leverages low-cost commodity hardware and provides simple APIs or web services to store large amounts of structured, semistructured and unstructured data. Generally, object storage or HDFS are immutable in nature and require logic to update/delete the data. Unlike advanced databases, these services are basic storage services, users need to take care of ACID transactions, metadata, data governance and security. Simple or distributed **file systems** can also be used as external sinks.
- A **data lake, data warehouse or database** is an application or service used for analytical or operational purposes to store and manage the data. Data lakes, data warehouses and lakehouses are examples of analytical data stores; relational DBMS (RDBMS) or NoSQL databases are examples of operational data stores. There are also other special-purpose databases, such as time series databases, graph databases, in-memory databases and elastic search databases, which can be used as external sinks depending on the requirements.
- An **event or message broker** can be used as an external sink for further integration of systems or downstream applications, such as real-time ML inferencing service and push notifications. It is not necessary to create a separate instance or cluster of brokers; users can share the same broker to create new topics and subscriptions. (For details, please refer to the component Event Broker Cluster/Service.)

Example Technologies

Examples of external sink technologies include:

- Cloud object storage, HDFS or file systems:
  - Amazon Web Services (AWS) Amazon S3
  - Apache Hadoop HDFS
  - Azure Data Lake Storage Gen2, Azure Blob Storage
  - Google object storage
  - IBM Cloud Object Storage
  - MinIO
  - OpenIO
  - Solid-state drive (SSD)/hard-disk drive (HDD) with basic file systems, Network File System (NFS)

- Data lakes, data warehouses and databases:
  - Amazon Redshift, Amazon Relational Database Service (RDS), Amazon DynamoDB, Amazon Neptune and more
  - Apache Hadoop-based data lake
  - Azure Synapse, Microsoft SQL Server, Azure CosmosDB and more
  - Databricks
  - Elasticsearch
  - Google BigQuery, Google Cloud Spanner, AlloyDB and more
  - IBM Netezza, IBM Db2
  - MongoDB
  - MySQL
  - Neo4j
  - Oracle Database
  - PostgreSQL
  - Redis
  - Snowflake

Key Characteristics

External sinks have the following characteristics:

- Designed for temporary or long-term storage in a robust and fault-tolerant manner
- Provide interfaces, protocols or web services to ingest high-speed streaming data using query languages such as SQL or GraphQL, or programming languages such as Python, Java, Scala or R
- Provide data backup, replication and recovery capabilities
- Should provide robust security, access control and governance capabilities

Related Research

- Comparison of Data Stores to Support Modern Use Cases
- Decision Point for Selecting the Right DBMS

EDA Operations

The EDA operations component provides control plane capabilities to configure, secure and monitor event-driven architecture and its components. EDA operations capabilities support the needs of two user personas:

- **Product teams** operating event-driven applications
- **Platform teams** operating the event brokers and related platform components

The EDA operations component has three subcomponents:

- **Administration** provides tools and services for managing resources within the environment. These resources include:
  - Tenants or namespaces to group and isolate configurations for different teams or applications
  - Topics provisioned with appropriate replication and configuration to meet application requirements
  - Broker native connector instances and configuration
  - Event replication and broker bridging configuration
  - Client connections and active subscriptions
  - Stream processing jobs operating within the broker platform
- **Access management** is used to administer access control lists (ACLs) or role-based access control (RBAC) configuration for the various identities that interact with the event-driven architecture. The identities include product teams, member identities, and service accounts for event sources and sinks. These identities must then be granted appropriate access to resources and platform capabilities including resource creation, modification and administration rights. Access management includes granting topic-level access rights for publication of events and creation of subscriptions.

- **Event monitoring** provides product teams with visibility of the state of their topics, subscriptions, integration process and other resources. This also includes historical trend data to support diagnostic activities.

Example Technologies

Examples of EDA operations component technologies include:

- AKHQ
- Avada Software Infrared360
- Axual Self-Service for Apache Kafka
- Conduktor
- Confluent Cloud Console, Confluent Control Center
- Lenses
- meshIQ
- Solace PubSub+ Cloud Console
- Redpanda Console

Key Characteristics

The EDA operations component has the following characteristics:

- Provides web and/or command line interface (CLI) access to administration of the event broker infrastructure and related resources to enable efficient platform operation and application operations
- Supports a tenancy and authorization model that supports appropriate separation of duties and safe coexistence of multiple applications when sharing platform infrastructure
- Captures and presents operational data to enable product owners and platform teams to judge and improve the health of applications and platform workloads

Related Research

- Essential Patterns for Event-Driven and Streaming Architectures

Event Governance

The event governance component includes tools for storing, publishing and managing the life cycle of metadata about your EDA. This includes event-based API specifications, event schema definitions, published (shared) topics, and documentation and tools to support design time and runtime discovery.

The event governance component has four subcomponents:

- **Schema registry:** A repository of schema definitions and their version dependencies that exposes an API that can be used by event producer clients, event consumer clients and event brokers to validate the format of events as they are published or consumed. The schema registry should also include features for explicitly declaring the compatibility mode for each schema (for example, forward- or backward-compatible) such that updates to a schema can be validated (and rejected if necessary). This helps developers of event producers and consumers determine whether update cycles to handle schema changes should be producer-first or consumer-first.
- **Event catalog:** A repository of additional metadata about the event-driven architecture, participating applications or services and their relationships. This may be an extension of — or integrated with — the schema registry to support runtime discovery, but is most commonly used as the metadata store to support a developer portal.

- **Developer portal:** A user interface supporting design time and ad hoc discovery of artifacts within the event-driven architecture. The portal may provide self-service features to support authorization requests and collaboration on change management; however, the developer portal should provide access to some or all of the following resources:
  - Event schemas
  - Active topics and associated metadata
  - Schema-to-topic relationships
  - Topic relationships or topology
  - Data lineage or origin
  - Event producers
  - Active event consumers
  - Documentation describing the use of and purpose of each asset
- **Event schema definitions:** Define the structure of event messages or records published in the environment. Event schemas allow minimally coupled event publishers and consumers to communicate an interface contract that formally describes the content of each event message.

Example Technologies

Examples of event governance technologies include:

- Apache Avro
- Apicurio Registry
- AsyncAPI
- AWS Glue Schema Registry
- Axual Governance
- Azure Schema Registry
- Bump.sh

- Confluent Cloud Stream Governance
- EventCatalog
- Gravitee
- IBM Event Endpoint Management
- JSON Schema
- Karapace
- Klaw
- Lenses
- meshIQ
- Protocol Buffers
- Solace PubSub+ Event Portal

Key Characteristics

The event governance component has the following characteristics:

- Provides web and/or CLI tools to define, discover, modify and view metadata associated with the structure of the event-driven application, including event schema definitions, topics and topic dependencies, documentation of event semantics
- Enables application owners to publish design and publish metadata that define the event-based interfaces exposed by the components of the application
- Supports metadata formats that enable explicit definition of event payloads (schemas) and allow life cycle management and versioning of those payloads
- Integrates with the event broker and/or source and sink client libraries to provide runtime validation of event message schema definitions and enforcement of additional policies, such as throttling throughput

Related Research

- Essential Patterns for Event-Driven and Streaming Architectures
- Choosing an API Format: REST Using OAS, GraphQL, gRPC or AsyncAPI

Architecture Principles and Patterns

The following architecture principles and patterns apply to stream data processing architecture:

- **Low latency:** In real-time analytics, time value of data decreases with increase in latency. Stream processing architecture provides native stream processing mode for critical applications where time value is very high, and microbatch mode where overall throughput has high value.
- **Stream life cycle management:** Stream data processing architecture supports the end-to-end life cycle of event streams. It enables collection, processing and delivery of events based not only on arrival time, but also original event time. It also handles out-of-order events. It supports effective event state management and fault tolerance, which is critical for complex event processing and analytics use cases. It provides a windowing mechanism to support time-window-based aggregations or analytical processing. Also, it supports schema evolution management, which is critical in business change management.
- **"Exactly once" semantics:** In stream processing, data consistency is very important which is underpinned by "exactly once" semantics of stream processors. Exactly once processing means that each event is processed and stored in the sink exactly once despite pipeline failure. This approach ensures that data is not duplicated or lost.
- **Unified batch and streaming model:** Streaming is not a replacement for canonical batch jobs. In the practical world, many times users need to reprocess all the data because of failures or to accommodate change requests from the business. Traditionally, this has been done using Lambda architecture, which causes overhead of managing separate workflows and code for batch and streaming pipelines (even though business logic is the same), and later merges and deduplicates the data. A unified batch and streaming model reduces architectural complexity and operational overhead by treating batch as a stream and providing a common semantic for batch and streaming workloads.
- **Observability and governance:** Stream data processing is complementary architecture to overall D&A architecture. It is important to implement observability for event brokers, streams, workflow and stream processors, and establish stream data governance and security in accordance with overall D&A governance and security.

Key Architecture Recommendations

- Leverage data partitioning capabilities of the broker to distribute data and scale data processing across multiple nodes and clusters for achieving a high degree of parallelism and performance. Avoid using brokers as long-term storage; even though some of them support storing large amounts of data, they are not a replacement for DBMS or modern data stores.
- Select the appropriate mode of stream processing — native streaming or microbatch — based on careful analysis of use cases and assessing performance, costs and benefits with respect to SLAs.
- Evaluate stream processing frameworks across several capabilities including checkpointing, back-pressure handling, windowing and watermarking, and assess them against your latency, throughput and use case requirements.
- Implement state management mechanisms to maintain intermediate states of data while doing complex data processing.
- Avoid using stream processing systems as data-serving endpoints (except for critical use cases such as monitoring dashboards or real-time analytics). Store the data to another system that can provide very low-latency access to data (e.g., a key/value NoSQL database or in-memory cache) or consumer use-case-specific optimized access to data (e.g., data lake, data warehouse or elastic search database).
- Implement robust security and governance of the architecture including event sources, event streams, sinks and consumer applications, using appropriate encryption, authentication and authorization.
- Leverage managed services and serverless offerings of event brokers as well as stream processors vendors, as deploying and managing broker and stream processor clusters is time-consuming and becomes operational overhead.

Related Architecture Research

Reference Architecture Brief: Cloud-Native Application Architecture

Reference Architecture Brief: Event-Driven Application Architecture

Recommended by the Author

Some documents may not be available as part of your current Gartner subscription.

Essential Patterns for Event-Driven and Streaming Architectures

Choosing Event Brokers: The Foundation of Your Event-Driven Architecture

Select the Right Event Broker Technologies for Your Event-Driven Architecture Use Cases

Succeed With Event-Driven Implementations by Correcting Common Misconceptions

Market Guide for Event Stream Processing

Essential Patterns for Data-, Event- and Application-Centric Integration and Composition

Stream Processing: The New Data Processing Paradigm

Streaming Analytics in the Cloud: A Comparative Analysis of Amazon, Microsoft and Google

Streaming Architectures With Kafka

Maturity Model For Event Driven Architecture
