UNIT I- INTRODUCTION TO REAL-TIME DATA STREAMING AND MODERN STREAMING ARCHITECTURES

Introduction to Streaming Data – Batch Processing vs Stream Processing – Characteristics of Streaming Data – Event-Driven Architecture – Lambda Architecture – Kappa Architecture – Modern Streaming Architectures – Data Sources (IoT, Sensors, Social Media, Financial Transactions, Video Streams, Edge Devices) – Event Streams – Real-Time Programming Models – Cloud-Native Streaming – Edge Analytics – Serverless Event Processing – Architecture Design Principles – Scalability, Fault Tolerance and High Availability.

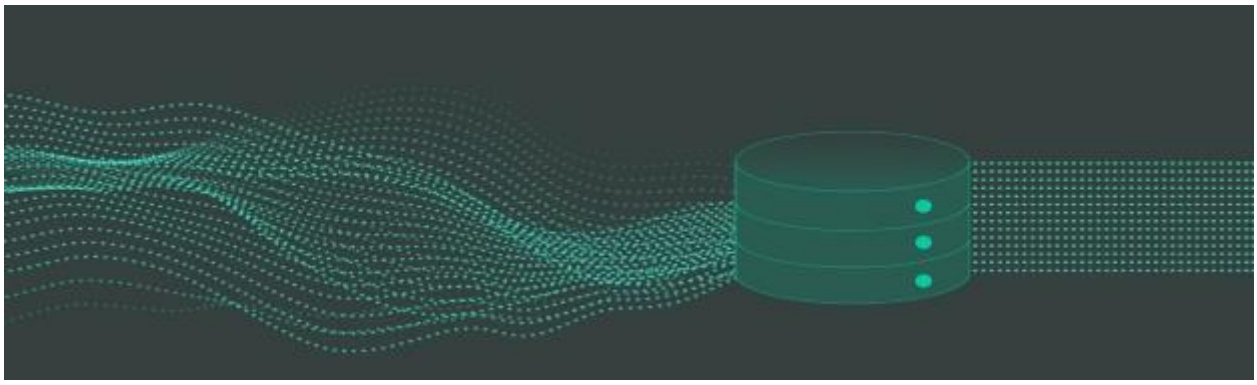
1. Introduction to Streaming Data

What is data stream?

- Data streaming is a modern approach to processing and analyzing data in real-time, as opposed to batch processing methods.
- A data stream is a continuous flow of data elements that are ordered in a sequence and processed as they are generated.
- Data stream is different from traditional batch processing methods in that they are continuous, unbounded, and potentially high-velocity with high variability.

Unlike traditional data processing, where data is collected and processed in batches, data streams are continuously collecting data, making it possible to process data as soon as they are created. This provides businesses with the ability to monitor and succeed in day-to-day operations.

Key features of data streams include their continuous flow, infinite length, unbounded nature, high velocity, and potentially high variability. They are often used in stream processing systems like Apache Spark Streaming.



common examples of streaming data

- Financial market data from stock exchanges and trading platforms
- IoT and sensor readings from equipment and smart devices
- Social media activity streams from platforms like X, Instagram and TikTok
- Website clickstream data from ecommerce and media sites
- Application log and telemetry data from cloud platforms and microservices
- GPS and location tracking data from delivery fleets and rideshare services

Importance of data streams in modern data processing

Data streams play a critical role in modern data processing, enabling real-time insights and automated actions. Businesses can analyze data as it's generated and make decisions based on up-to-the-minute information. This is in contrast to batch processing, which can take hours or even days to complete.

Data streams allow for continuous flow of data, providing insights in real-time and enabling timely data analysis. This is particularly important for industries such as ***finance***, where real-time insights can help detect fraud and other security breaches. It's also important for home security systems, where real-time event data can alert homeowners to potential threats.

Social media feeds generate a continuous stream of data, making it essential for analyzing user behavior and preferences. Retailers can use data streams to monitor market trends and pricing data, while logistics companies can use them to optimize inventory management and troubleshoot systems in real-time.

In the healthcare ***industry***, data streams enable continuous monitoring of patient data, allowing for early detection and intervention in the event of critical health issues. Data streaming can also be used in machine learning algorithms to derive insights from continuous data and improve predictive analytics.

So evidently, data streams are essential for modern data processing and decision-making, enabling businesses to derive valuable insights from the continuous stream of real-time data generated by their internal IT systems and external data sources

How streaming data works

At a high level, streaming data works by continuously capturing, processing and analyzing real-time data flows from various sources. This process consists of four key stages:

- Data ingestion
- Stream processing
- Data analysis
- Data storage

Data ingestion

The first stage involves capturing incoming data streams from diverse sources. Modern data ingestion tools such as Apache Kafka buffer and standardize these streams as they arrive, which helps ensure both scalability and data consistency.

Organizations typically integrate data ingestion tools with other components to create unified workflows. Data integration tools can also further align disparate data types into a standardized format for processing to help ensure that data from multiple sources can be combined and analyzed effectively.

Stream processing

In the processing stage, stream processing frameworks such as Apache Flink analyze and transform data while it is in motion. These frameworks enable organizations to:

- Process complex events in real-time
- Perform data aggregation at scale, such as calculating averages, counting events or adding up transaction values
- Apply transformations—such as filtering, enriching or formatting data—as data flows through the data pipeline

Data analysis and visualization

At this stage, organizations derive actionable business insights from streaming data flows through data visualization and other analytical tools.

Key applications include:

- Real-time dashboards that deliver critical metrics and KPIs
- Operational applications that automate workflows and optimize processes
- Machine learning models that analyze patterns to predict outcomes

Data storage

When storing streaming data, organizations must balance the need to quickly access data for real-time use with long-term data storage, cost-efficiency and data compliance concerns.

Many organizations use data lakes and data lakehouses to store streaming data because these solutions offer low-cost, flexible storage environments for large amounts of data. After streaming data is captured, it might be sent to a data warehouse, where it can be cleaned and prepared for use.

Organizations often implement multiple data storage solutions together in a unified data fabric. For example, financial institutions might use data lakes to store raw transaction streams while using warehouses for analytics and reporting.

Types of streaming data

Organizations can use many types of streaming data to support real-time analytics and decision-making. Some of the most common streaming data flows include:



Event streams

Event streams capture system actions or changes as they occur, such as application programming interface (API) calls, website clicks or app log entries. Event streams are commonly used to track real-time activities across systems, enabling instant responses to user interactions or system events.



Real-time transaction data

Real-time transaction data captures continuous flows of business transactions, such as digital payments or e-commerce purchases. Real-time transaction data supports applications such as fraud detection and instant decision-making.



IoT and sensor data

IoT and sensor data includes information about environmental conditions, equipment performance and physical processes. These data streams often support real-time equipment monitoring and process automation.

Streaming data tools and technologies

With the help of both open source and commercial streaming data solutions, organizations can build scalable data pipelines that are fault-tolerant, meaning they can recover from failures without data loss or downtime.

Two key types of technologies underpin most streaming data implementations: stream processing frameworks and streaming data platforms.

Stream processing frameworks

Stream processing frameworks provide the foundation for handling continuous data flows. These frameworks help organizations build high-performance data pipelines that consistently process large volumes of data quickly and reliably.

Three open source frameworks dominate the streaming landscape:

- **Apache Kafka:** A leading streaming platform, Kafka can handle massive data volumes with millisecond latency. Organizations often use Kafka to build pipelines for activity tracking, operational monitoring and log aggregation.

- **Apache Flink:** Apache Flink specializes in complex event processing and stateful computations. It's valuable for real-time analytics, fraud detection and predictive maintenance, where understanding the context of events over time is critical.

- **Apache Spark:** Known for its unified analytics functions, Spark can handle both batch and stream processing using the same execution engine and APIs. This ability makes it useful in scenarios where organizations need to analyze historical datasets alongside real-time data streams.

- **Spark Structured Streaming:** An extension of the core Spark API built on the Spark SQL engine that supports scalable, fault-tolerant stream processing. It uses micro-batch and continuous processing models to process streaming data incrementally.

Streaming data platforms and services

Streaming data platforms provide tools that support the full lifecycle of real-time data, from ingestion and processing to storage and integration.

Major cloud providers offer managed streaming services that simplify the deployment and operation of high-volume data streaming applications. Examples include Amazon Kinesis from Amazon Web Services (AWS), Microsoft Azure Stream Analytics, Google Cloud Dataflow and IBM Event Streams. These services provide ready-to-use capabilities that help organizations avoid building complex streaming infrastructure from scratch.

Many organizations also adopt hybrid streaming architectures that combine cloud-native services with on-premises systems to meet performance, scalability and data residency requirements.

In addition, platforms such as Confluent provide enterprise-grade streaming capabilities for building, managing and scaling real-time data pipelines across diverse IT environments. Confluent is widely recognized for extending Apache Kafka with advanced features for governance, security, observability and cross-environment data streaming.

2. Batch Processing vs Stream Processing

Batch processing and stream processing are two different methods of handling data. Batch processing is processing vast volumes of data at once and at scheduled intervals, while stream processing is constantly processing data in real-time as it arrives.

What Is Batch Processing

Batch processing is when computers handle high-volume and repetitive tasks by grouping data into batches and processing. This method doesn't require immediate results, saves time, and reduces computing power.

For instance, let's look at your payroll system. In this system, you gather all employee data over a pay period and calculate salaries for everyone at once through batch processing. Similarly, when you run system backups, you consolidate and store data in bulk at specific intervals.

Batch processing generates detailed reports when analyzing large datasets, and it's ideal for achieving consistency and resource efficiency.

However, it is not always suitable for tasks **requiring instant feedback or action.**

What Is Stream Processing

Stream processing continuously ingests and analyzes data. Instead of waiting for the data to accumulate, you can process it instantly. Therefore, you can respond immediately to changes, which is critical for tasks depending on fast decisions.

For example, stream processing powers fraud detection systems. As transactions occur, you can spot and block fraudulent activity before it's too late. Similarly, if you're running an e-commerce site, you can personalize customer experiences straightaway by analyzing their behavior.

You'll also find stream processing in action with IoT devices, like smart thermostats or fitness trackers, which react instantly to changes in their environment.

Stream processing provides the speed for modern applications, but it can be complex to implement and requires advanced infrastructure.

Batch Processing vs. Stream Processing – Key Differences

Batch processing and stream processing are different ways of processing data.

Batch and stream processing are suited to different data operations. You can typically use batch processing for large-scale, infrequent data jobs that don't require immediate results. It processes data in enormous chunks, making it perfect for tasks where a delay is acceptable.

Whereas, you can use stream processing for continuous, real-time data processing where immediate insights and actions are vital. It handles data as it flows in by ensuring minimal latency and real-time data analysis and response.

Criteria	Batch Processing	Stream Processing
The Nature of the Data	Processed gradually in batches.	Processed continuously in a stream.
Processing Time	On a set schedule.	Constant processing.
Complexity	Simple, as it deals with finite and predetermined data chunks.	Complex, as the data flow is constant and may lead to consistency anomalies.
Hardware Requirements	Varies but can be performed by lower-end systems as well as high-end systems.	Demanding while also requiring that the system be operational at all times.
Throughput	High. Batch processing is intended for large amounts of data, and as such, it is optimized with that goal in mind.	Varies depending on the task at hand.
Application	Email campaigns, billing, Invoicing, scientific research, image processing, video processing, etc.	Social media monitoring, fraud detection, healthcare monitoring, network monitoring, etc.

Consistency & Completeness of Data	Data consistency and completeness are usually uncompromised upon processing.	Higher potential for corrupted data or out-of-order data.
Error Recognition & Resolution	Errors can only be recognized and resolved after the processing is finished.	Errors can be recognized and resolved in real-time.
Input Requirements	In batch processing, inputs are static and preset.	In stream processing, inputs are dynamic.
Available Tools	Apache Hive, Apache Spark, Apache Hadoop.	Apache Kafka, Apache Storm, Apache Fink.
Latency	High latency, as insight becomes available only after the processing of the batch finishes.	Low Latency, with insights being available instantaneously.

Advantages and Disadvantages of Batch Processing

The Pros of Batch Processing

- **Efficiency for large datasets**: Batch processing excels at handling large volumes of data in one go, optimizing computational resources, and minimizing processing overhead.
- **Cost-effective resource usage**: Batch jobs are often scheduled during off-peak times, reducing costs associated with high-demand periods.
- **Streamlined workflows**: The sequential nature of batch processing simplifies system design and is ideal for operations that don't require immediate results.
- **Error handling**: Errors can be identified and addressed before processing the next batch to preserve data accuracy and integrity.
- **Scalability**: Batch systems can handle growing data volumes effectively, making them suitable for businesses planning for long-term growth.

The Cons of Batch Processing

- **Delayed outcomes**: Results are not available until the entire batch is processed, which is a drawback for time-sensitive tasks.
- **Resource spikes**: Running batch jobs can temporarily strain system resources, potentially causing slowdowns during processing.
- **Inflexibility**: Adjusting batch workflows often requires significant changes, making them less adaptable to real-time needs.
- **Error propagation**: If errors occur during batch processing, they may affect the entire batch, requiring reprocessing.
- **Higher upfront costs**: Initially, designing and setting up batch processing systems can be resource-intensive

Advantages and Disadvantages of Stream Processing

The Pros of Stream Processing

- **Real-time insights**: Stream processing enables instant data analysis, which is critical for applications like fraud detection, live monitoring, and dynamic decision-making.
- **Continuous data handling**: Data is processed as it arrives, allowing uninterrupted workflows and timely responses.
- **Enhanced agility**: Your company can react quickly to real-time events by providing a competitive edge in fast-paced industries.
- **Event-driven operations**: It's ideal for systems relying on triggers, such as IoT devices, online transactions, and sensor networks.
- **Dynamic scalability**: Stream processing systems can adapt to fluctuating data loads by ensuring consistent performance.

The Cons of Stream Processing

- **Increased complexity**: Implementing stream processing requires sophisticated architecture, specialized tools, and expertise.
- **Higher operational costs**: Real-time systems often demand significant computing resources, which boosts overall expenses.
- **Data accuracy challenges**: Ensuring consistency and handling out-of-order events can be complicated.
- **Monitoring and maintenance**: Stream processing systems need continuous monitoring to address issues immediately.
- **Limited historical context**: Stream processing focuses on current data. As a result, it's less suitable for applications requiring detailed historical analysis.

3.characteristics of streaming Data

1. Chronological Significance

Streaming data retains its value based on the order of events. Timeliness is important for making accurate decisions, as the sequence in which data arrives can impact the resulting analysis. For example, in monitoring systems, the sequence of error logs can determine the cause and effect relationships between different events.

Ensuring chronological accuracy in data streams often involves time-stamping data as it is generated. This allows systems to process events in the correct order, preserving the temporal integrity of the data.

2. Continuous Flow

Data streaming involves a constant flow of data rather than periodic updates. This continuous nature supports real-time analytics, where data is immediately processed as it arrives. Streaming applications can handle high-velocity data streams, enabling timely reactions to events and trends.

Applications like financial trading systems or live social media feeds rely on continuous data flow. These systems must process vast amounts of incoming data without delays to provide accurate, up-to-date information.

3. Uniqueness

Streaming data often features unique, non-repetitive events. Each data point may represent a distinct incident, such as a sensor reading or a user action. This uniqueness requires systems to handle each incoming data piece individually, ensuring no critical information is overlooked.

Systems must be designed to accommodate this uniqueness, processing data points effectively while maintaining overall performance and accuracy. Data deduplication strategies may be necessary to handle occasional repetitive data, but the primary focus remains on treating each piece of streamed data as an individual event.

4. Nonhomogeneity

Streaming data typically comes in various formats and from different sources, challenging systems to process diverse data types efficiently. This nonhomogeneity demands robust, flexible data processing architectures capable of handling mixed data streams without compromising performance.

Processing nonhomogeneous data often involves normalizing and integrating data from multiple sources to provide a cohesive, comprehensive view. Systems must adapt to

different data structures, ensuring seamless integration and consistent processing across various data types, crucial for accurate and meaningful analytics.

5. Imperfection

Streaming data is often imperfect, containing errors, missing values, or inconsistencies. Systems that handle streaming data must implement mechanisms for real-time data validation, cleansing, and correction to ensure data quality and reliability.

Addressing data imperfections in real-time can involve filtering out erroneous data, filling in gaps, and reconciling conflicting information. Effective handling of imperfect data ensures that insights derived from real-time analytics are dependable and actionable.

3. Event-Driven Architecture(EDA)

Event-Driven Architecture

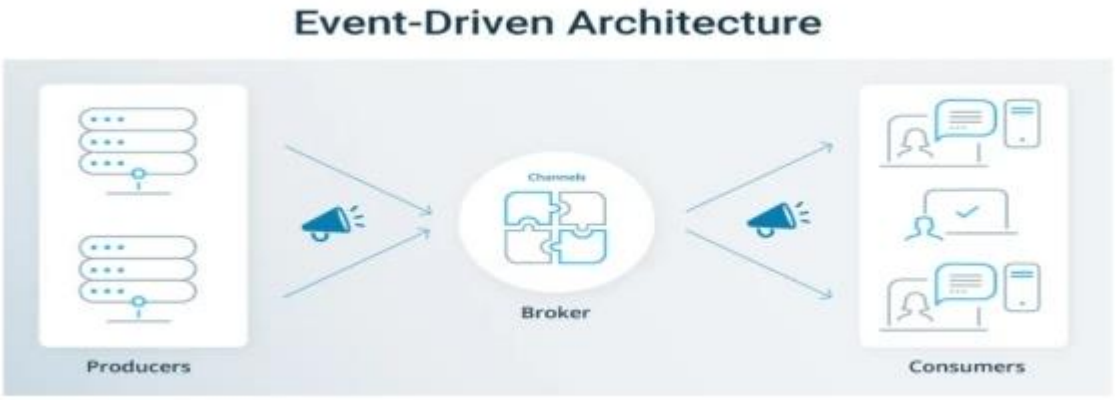


Fig 1. Event-Driven Architecture

What is Event-Driven Architecture?

Event-driven architecture is a software design pattern that allows decoupled applications to asynchronously publish and subscribe to events through an event broker (modern messaging-oriented middleware).

Event-driven architecture is a method of developing enterprise IT systems that allows information to flow in real time between applications, microservices, and connected devices as events occur throughout the business.

The event-driven architecture enables loose coupling of applications by introducing a middleman known as an event broker. This means that applications and devices do not need to know where they are sending information or where the information they are consuming comes from.

What does the term "event" mean in EDA?

A change in state is called an event. In an event-driven architecture, everything that happens within and to your enterprise is an event — customer requests, inventory updates, sensor readings, and so on.

Why should you use an event-driven architecture?

The value of knowing about a given event and being able to react to it degrades over time. The more quickly you can get information about events where they need to be, the more effectively your business can react to opportunities to delight a customer, shift production, and reallocate resources.

That’s why event-driven architecture, which pushes information as events happen, is a better architectural approach than waiting for systems to periodically poll for updates, as is the case with the API-led approach most companies take today.

Event-driven architecture ensures that when an event occurs, information about that event is sent to all of the systems and people that need it. It’s a simple concept, but these events have had quite the journey. They need to efficiently move through a multitude of applications written in many different languages, using different APIs, leveraging different protocols, and arriving at many endpoints such as applications, analytics engines, and user interfaces.

The Advantages of Event-Driven Architecture

The main advantages of event-driven architecture are improved responsiveness, scalability, and agility in business. The ability to respond to real-time information and quickly and easily add new services and analytics significantly improves business processes and the customer experience. Despite the fact that modernizing IT infrastructure can be costly,

With event-driven architecture, you can do the following:

- Everything happens as soon as possible, and nothing is waiting on anything else.
- You don’t have to consider what’s happening downstream, so you can add service instances to scale.
- Topic routing and filtering can divide up services quickly and easily, as in command query responsibility segregation.
- To add another service, you can just have it subscribe to an event and have it generate new events of its own; there’s no impact on existing services.

Common Event-Driven Architecture Concepts

There are eight key architectural concepts that need to be understood for event-driven architecture to be successful:

1. Event broker
2. Event portal
3. Topics
4. Event mesh
5. Deferred execution
6. Eventual consistency
7. Choreography

8. Command Query: Responsibility Segregation

1. Event Broker

An **event Broker** is a middleware (which can be software, an appliance, or SaaS) that routes events between systems using the publish-subscribe messaging pattern. All applications connect to the event broker, which is responsible for accepting events from senders and delivering them to all systems subscribed to receive them.

It takes good system design and governance to ensure that events end up where they are needed and effective communication between those sending events and those who need to respond. This is where tooling—such as an **event portal**—can help capture, communicate, document, and govern event-driven architecture.

2. Event Portal

As organizations look to adopt an event-driven architecture, many are finding it difficult to document the design process and understand the impacts of changes to the system. **Event portals** let people design, create, discover, catalog, share, visualize, secure, and manage events and event-driven applications. Event portals serve three primary audiences:

1. Architects use an event portal to define, discuss, and review events, data definitions, and application relationships.

2. Developers use an event portal to discover, understand, and reuse events across applications, lines of business, and between external organizations.

3. Data scientists use an event portal to understand event-driven data and discover new insights by combining events.

3. Topics

Events are tagged with metadata that describes the event, called a "topic." A topic is a hierarchical text string that describes what's in the event. Publishers just need to know what topic to send an event to, and the event broker takes care of delivery to systems that need it. Application users can register their interest in events related to a given topic by *subscribing* to that topic. They can use wildcards to subscribe to a group of topics that have similar topic strings. By using the correct topic taxonomy and subscriptions, you can fulfill two rules of event-driven architecture:

1. A subscriber should subscribe only to the events it needs. The subscription should do the filtering, not the business logic.

2. A publisher should only send an event once, to one topic, and the event broker should distribute it to any number of recipients.

4. Event Mesh

An event mesh is created and enabled through a network of interconnected event brokers. It's a configurable and dynamic infrastructure layer for distributing events among decoupled applications, cloud services, and devices by dynamically routing events to any application, no matter where these applications are deployed in the world, in any cloud, on-premises, or IoT environment. Technically speaking, an event mesh is a network of interconnected event brokers that share consumer topic subscription information and route messages amongst themselves so they can be passed along to subscribers.

5. Deferred Execution

If you're used to REST-based APIs, the concept of deferred execution can be tricky to comprehend. The essence of event-driven architecture is that when you publish an event, you don't wait for a response. The event broker "holds" (persists) the event until all interested consumers accept or receive it, which may be sometime later. Acting on the original event may then cause other events to be emitted that are similarly persistent.

So event-driven architecture leads to cascades of events that are temporally and functionally independent of each other but occur in a sequence. All we know is that **event A** will at some point cause something to happen. The execution of the logic-consuming **event A** isn't necessarily instant; its execution is deferred.

6. Eventual Consistency

Following on from this idea of deferred execution, where you expect something to happen later but don't wait for it, is the idea of **eventual consistency**. Since you don't know when an event will be consumed and you're not waiting for confirmation, you can't say with certainty that a given database has fully caught up with everything that needs to happen to it and doesn't know when that *will* be the case. If you have multiple stateful entities (database, MDM, ERP), you can't say they will have exactly the same state; you can't assume they are consistent. However, for a given object, we know that it will become consistent *eventually.*

7. Choreography

Deferred execution and eventual consistency lead us to the concept of choreography. To coordinate a sequence of actions being taken by different services, you could choose to introduce a master service dedicated to keeping track of all the other services and taking action if there's an error. This approach, called *orchestration,* offers a single point of reference when tracing a problem, but also a single point of failure and a bottleneck.

With event-driven architecture, services are relied upon to understand what to do with an incoming event, frequently generating new events. This leads to a "dance" of individual services doing their own things but, when combined, producing an implicitly coordinated response, hence the term choreography.

8. CQRS: Command Query Responsibility Segregation

A common way of scaling microservices is to separate the service responsible for doing something (a command) from the service responsible for answering queries. Typically, you have to answer many more queries than for an update or insert, so separating responsibilities this way makes scaling the query service easier.

Using event-driven architecture makes this easy since the topic should contain the verb, so you simply create more instances of the query service and have it listen to the topics with the query verb.

The 6 Principles of Event-Driven Architecture

There we have it, some main principles of event-driven architecture:

1. Use a network of event brokers to make sure the right "things" get the right events.
2. Use topics to make sure you only send once and only receive what you need.
3. Use an event portal to design, document, and govern event-driven architecture across internal and external teams.
4. Use event broker persistence to allow consumers to process events when they're ready (deferred execution).
5. Remember, this means not everything is up-to-date (eventual consistency).

6. Use topics again to separate out different parts of a service (command query responsibility segregation).

Examples of Event-Driven Architecture

The value of event-driven architecture transcends industries and can be applied to small businesses as well as large multinational corporations. Retailers and banks can use it to aggregate data from point-of-sale systems and across distribution networks to execute promotions, optimize inventory, and offer excellent customer service.

Retail and eCommerce: An Example of Event-Driven Architecture

This is an example of event-driven architecture from a retail perspective. Notice that no systems (inventory, finance, or customer support) are polling to ask if there are any new events; they are simply filtered and routed in real-time to the services and applications that have registered their interest.

