Meta description - Event-Driven architecture plays an important role in the software industry. See what it is, how it works, and what benefits it brings to software development.

Title tag - Event-Driven Architecture | Complete Guide

Event-Driven Architecture: What It Is and How to Use It Effectively

Enterprises built on complex architectures, such as those in the financial sector, telecoms and media, face a number of challenges as their systems continue to grow in complexity. Scalability is more difficult to achieve, change is more complex and businesses miss out on opportunities to generate value from their data.

Event-driven architecture (EDA) is an architecture approach used by enterprises to tackle those challenges. EDA helps businesses unlock value from their data by enabling real-time data generation and utilisation, which can result in better, faster decision-making.

In this blog, we're going to give an introduction to EDA, its benefits and whether you should be using it within your organisation.

Before we go into what exactly EDA is, let's take a look at what exactly an event is.

What Are Events?

Events are any change of the state of a system and can either be the change itself (e.g. user actions, business transactions or APIs being called) or the identifier (e.g. a notification of a transaction).

Using an ecommerce platform as an example, events might include a range of end-user actions such as visiting the site, logging in, browsing products, adding a product to the cart or checking out. Additionally, backend or system actions, from inventory updates, shipping invoice creation, sending marketing emails to more technical tasks such as system cache clearance and system log creation, can also be considered events.

The value of an event can degrade if not acted upon in a timely manner. Each interested recipient would need to take action at different speeds depending on the nature of the transaction, event or regulations. Let's examine a card payment transaction done by a bank customer. Once the transaction is completed by a customer at a retail shop, the core banking system must be informed immediately to deduct the amount from the customer's account, and the notification service must inform the customer immediately via SMS or an in-app notification.

However, the statement generation service only needs to know about the event at the time of statement generation for a given period. EDA caters for broadcasting events in real-time or near real-time where events could be used by recipients at their own pace based on their needs.

What Is Meant By Event-Driven Architecture?

EDA is a software design pattern that focuses on broadcasting events to all interested applications, systems or microservices. It's considered an asynchronous architecture as it facilitates asynchronous communication between the sender and the recipient. Traditional request-based architectures require services to wait for a reply before moving on to a new task but the EDA model eliminates this need, removing the constraint of systems depending on a single message flow or awaiting additional processing.

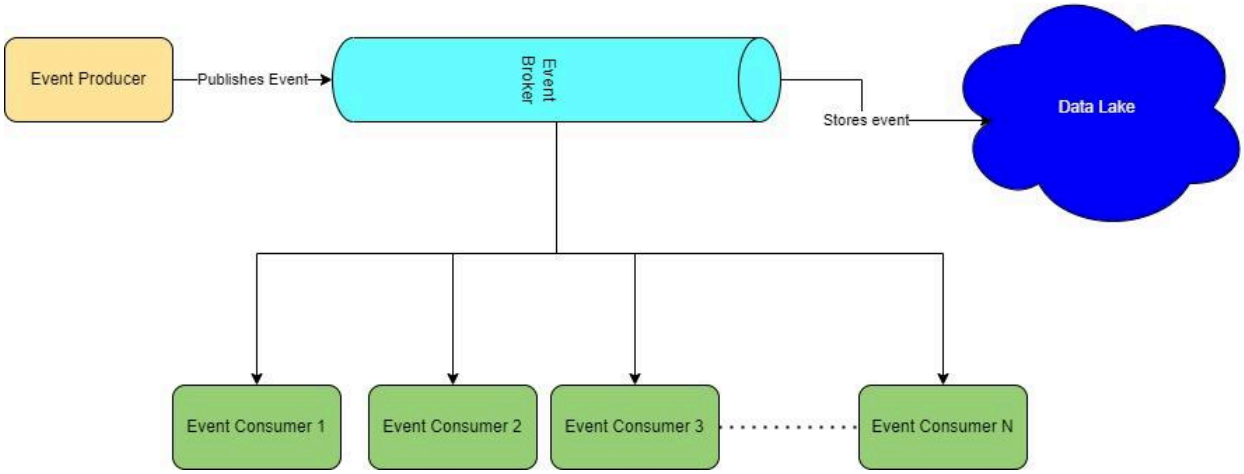

Diagram 1: Event-Driven Architecture High Level View

In an EDA-based system, the event producer will simply publish the event within the broker. It doesn't need to know who is subscribed or what actions they will do with the event. The event consumer doesn't need to know who published the event or what the other consumers will do. The consumer will start its actions once it's informed of the event.

Event Flow in Event-Driven Architecture

Now we have an understanding of EDA and events, we can look at the components in a typical event flow. These components are:

- Event Producers or Publishers
- Event Brokers or Event Hub
- Event Consumers or Subscribers

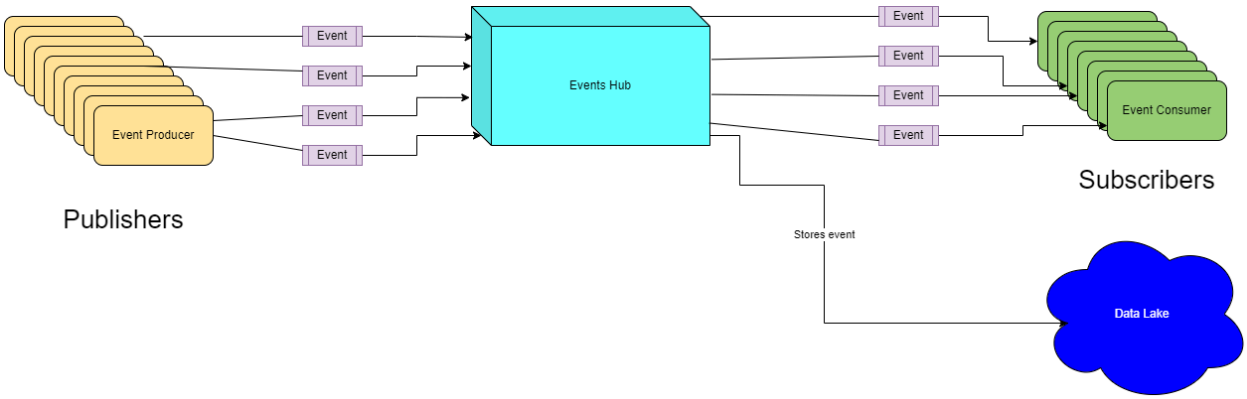

Diagram 2: Scaled EDA system

Event Producer

This is the first stage of the EDA event flow and is the source of an event i.e. where the event is created. An event producer can be any entity that can create an event from web applications, monitoring events, physical sensors, etc. This can be a system of record such as a core banking application, a billing system or an insurance platform. It can also be a system of engagement such as an ecommerce website, a customer care mobile application or a gaming application.

Event Broker

This is the second stage of the flow that acts as the middleware between the event producer and the event consumer. It collects the events created by the producer and routes the event to the appropriate consumer. Event brokers are essential for publish/subscribe messaging system (pub/sub) patterns. However, they can be ignored in simpler EDA implementations where event producers can directly talk to the consumers.

The broker can act asynchronously, accepting and processing multiple events and then sending them to all the appropriate consumers. Brokers can also queue events so that they can be consumed later.

Event Consumer

This is the end of a typical event flow, where the system logic resides and the required actions are carried out depending on the event. It can range from sending an email for a login alert or utilising an analytical engine to process the data sent by an event to even triggering a new event for further processing. A single EDA application can have multiple consumers waiting for different kinds of events targeted at specific actions. Consumers can be systems of records

such as a billing system, a microservice or a core banking system, and they can also be a system of engagement such as a marketing platform or a notification engine.

Loosely Coupled and Decoupled Nature of Event-Driven Architectures

EDA allows developers to easily create applications with loosely coupled services and sometimes entirely decoupled components. The application itself, devices, or in EDA terms, the event consumer does not need to know where the information is coming from. It leads to the loosely coupled nature of different components of the application. Even though these components require each other to perform the necessary actions within the system, they are not strictly bound together. Thus a failure on one service will not cause issues in others.

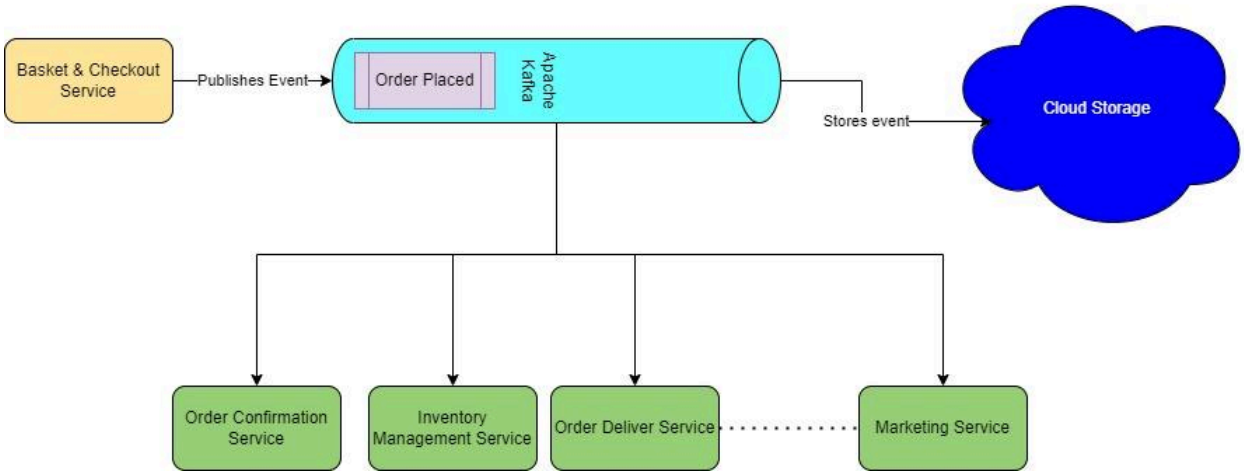

Diagram 3: EDA Example - An Ecommerce Order

In the ecommerce example shown in Diagram 3, the Basket and Checkout service publishes the event—order placement—to all interested applications. All services in this example are unaware of the existence of the other services; they could be part of the same system or components of different systems. For example, the organisation can easily change its marketing messaging according to order history without it having an impact on the event producer—Basket and Checkout Service—or any of the other services, with the caveat that the new platform will use the same produced event.

In the same way, if the organisation decides to add a new service to the checkout flow such as sending a post-sale survey to the customer, this can be developed and subscribed to the event. This will be seamless to the existing components of the system.

This decoupled nature can vastly improve the agility and scalability of a system. EDA provides a good architectural approach for a modular system as it prevents the need to depend on a reply or an acknowledgement for the communication between components. This enables the continuous change and agility needed in a competitive market.

Event-Driven Architecture Messaging Models

EDA relies on events communicated between different components of a system. The two main event models are publish/subscribe and event streaming.

Publish/Subscribe Model

The pub/sub model, which is also called the event messaging model, is based on a subscription pattern. An event consumer will subscribe to single or multiple topics that capture messages published by event producers. When an event producer publishes an event, it will be delivered to all event consumers who have subscribed to the relevant topic.

Event brokers receive the published message, apply transformations if necessary, queue and store them, and then send them to the subscribers. These messages are then deleted from the broker once they're consumed by the event consumers.

Amazon's Simple Notification Service SNS

SNS is a fully managed pub/sub service that sends notifications two ways, A2A and A2P. A2A provides high-throughput, push-based, many-to-many messaging between distributed systems, microservices, and event-driven serverless applications. These applications include Amazon Simple Queue Service (SQS), Amazon Kinesis Data Firehose, AWS Lambda, and other HTTPS endpoints. A2P functionality lets you send messages to your customers with SMS texts, push notifications, and email.

SNS is appropriate for organisations that have workloads on AWS and want to utilise SaaS based tools to free up their developers and operations to focus on business innovation.

Amazon's Simple Queuing Service SQS

Is a fully managed message queueing for microservices, distributed systems and serverless applications. It provides a reliable and highly scalable message distribution. Unlike SNS, it supports a first in first out (FIFO) feature that guarantees the order of messages. It is more suitable for systems that need to receive messages in the exact order and only once. It supports different encryption options like SNS.

Azure Web Pub/Sub

Is a fully managed service that supports native and serverless WebSockets. It offers a real-time pub/sub messaging for web application development through native and serverless WebSocket

support. It is highly scalable and can scale to millions of connections. It is a good option for organisations who have their workloads in Azure and want to utilise a SaaS product to keep their focus on functionality and business logic and innovation.

GCP's Pub/Sub

Is a fully managed highly available message distribution platform. Its synchronous, cross-zone message replication and per-message receipt tracking ensures reliable delivery at any scale. It natively integrates with BigQuery, DataFlow and other operational DBs, as well as providing third-party integrations with Splunk and Datadog for logs along with Striim and Informatica for data integration. This is a great option for organisations who have their workloads in GCP and who want to focus their efforts on business innovation by choosing a SaaS platform.

Apache Kafka

Is an open-source distributed pub/sub platform that offers high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. Like the other Apache products, Kafka is used by thousands of organisations across different industries. It's free and has an active community of developers. This is suitable for organisations who are looking for more control and are not afraid to venture with open source products.

Confluent

Is an off-the-shelf solution that is based on Apache Kafka. It is a cloud native SaaS platform that offers all features of Apache Kafka and other enriched features. It also comes with enterprise grade support.

Event Streaming Model

In this model, event producers stream event messages to the broker, and event consumers are subscribed to these streams. Here, event consumers can consume only the event they require from the event stream at any point. This is in contrast to the pub/sub model, where subscribers receive all the published messages. The messages are persistent which means that published messages are available even after subscribers have consumed them. The streaming model offers two distinct advantages compared to the pub/sub model:

- **Event Persistence:** This model stores the streaming events to allow subscribers to consume events at any time. The period these events are kept is configurable depending on the requirements. This persistent nature allows consumers to process historical data as well as real-time data.

- **Advanced Event Processing:** While simple event processing enables each published event to get delivered to a consumer that carries out a specific functionality. Advanced Event Processing exceeds those capabilities and powers advanced scenarios such as

batch event processing, where event consumers can process a series of events and act upon the results of the entire event series.

AWS EventBridge

Is an event hub SaaS based serverless tool that filters, transforms, routes, and delivers events. It supports multiple event buses, with a rich set of rules to be applied to the events with reply and archiving features.

AWS Kinesis

A data streaming as a service tool that collects, processes, and analyses real-time, streaming data. It is scalable and flexible ingesting different data types such as video, audio, application logs, website clickstreams, and IoT telemetry data for machine learning, analytics, and other applications. Amazon Kinesis enables the processing and analysis of data as it arrives and responding instantly instead of having to wait until all your data is collected before the processing can begin.

Azure Stream Analytics

Azure's data streaming SaaS tool that allows the streaming of large scale data in realtime. It is highly scalable (millions of events), performant (sub-second) and reliable with financially backed SLAs. It is a low code platform that provide an end-to-end analytics pipeline with SQL syntax and extensible with JavaScript and C# custom code.

GCP Dataflow

Is a fully managed highly available message data streaming tool. It Automates the provisioning and management of processing resources. It horizontally autoscales worker resources to maximise resource utilisation. It offers a cheaper processing option for data streams that can tolerate being batch processed within a 6 hour window, as well as powerful AI capabilities to process large volumes of data with near-human intelligence.

Apache Beam

Is an open-source distributed data streaming platform that offers a single programming model for both batch processing and real-time streaming.

How It Works: Example Architecture

Now we have an understanding of event-driven architecture, let's look at two examples.

Simple Monitoring Example

Assume you are managing an array of IoT devices in a manufacturing plant. Each device is configured to provide different metrics to ensure the smooth operation of the production line. This monitoring can range from detecting machine malfunctions and temperature changes to product defects.

In an event-driven configuration with pub/sub method, the broker will receive events from all the devices in real-time and pass those events to all the relevant subscribers. These subscribers may be automated systems that will take action according to the alerts. For instance, increase cooling for the component if a temperature increase is detected or simply alert a maintenance person about that anomaly.

Multi-Tiered Event Streaming Example

Think of a stock analytics organisation that needs to analyse the market movements both in real-time as well as on historical data. In this scenario, different sets of trading platforms can act as the event producers constantly feeding stock changes from different sources to an event broker.

This event broker will forward the necessary events to the subscribed event consumers while storing all the event data. Moreover, it will allow different components to access different sets of data. Components targeted at processing real-time events can subscribe directly to the broker and get the events. Meanwhile, components that require access to historical data can query them directly from the broker.

These components can process these events either individually or as a series of events to analyse the market and predict future market conditions. Furthermore, both real-time and historical data can be processed in parallel to gain even more insights. With these results, the system components can create new events that trigger different functions, such as automated trade executions or informing clients of their analysis.

Should You Use Event-Driven Architecture?

While EDA can be applicable for most modern application developments, there are scenarios where they are extremely beneficial.

Resource State Monitoring And Alerting

EDA is tailor-made for resource state monitoring and alerting. Any change to the state can be instantaneously captured and alerted. It is useful at both the application and infrastructure levels as software can be developed to capture the states of virtually any resource. When this

near-unlimited scalability is coupled with the asynchronous message processing, any number of events from any amount of resources can be captured.

These events can then be easily filtered and distributed to relevant subscribers to take necessary actions. Users can identify faulty patterns in a streaming configuration and easily troubleshoot or audit systems using historical data by storing these data.

Fanout And Parallel Processing

Another pillar of EDA is its ability to process messages asynchronously, allowing to fan out workloads and enabling parallel processing. Any number of events can be received by a broker from different event producers and delivered to different components for processing. In order to reduce the workload of a specific application component, it can be scaled out by provisioning multiples of the same component and distributing the events between them to fan out the work. The same can be done to enable parallel processing where events can be processed in parallel without waiting until a previous event is completed as with traditional response-driven architecture.

A scalable microservices and modular architecture

EDA is a good choice in a complex microservices architecture. With the increase in the number of microservices, synchronous communication between those services will become increasingly complex to orchestrate, increase latency within the system and increase the network utilisation. EDA will give the flexibility and agility to add, remove or modify microservices within the system without impacting the other services or the event producer or main system.

Benefits Of Event-Driven Architecture

While EDA has many benefits, the main benefits will be functionality and flexibility, especially when compared to response-driven architectural patterns. EDA provides the following benefits:

- **Real-Time Data Processing**
  EDA is geared at processing events as soon as they occur without having to wait for a response. Asynchronous messaging allows different components to communicate with various events asynchronously. It allows systems to react to events in real-time. Implementing an event stream will further expand this functionality by allowing users to keep track of both historical and real-time data, which will be crucial for analytics.

- **Responsiveness**
  Real-time event processing helps this architecture increase the responsiveness of a system naturally without requiring additional configurations. It not only increases the responsiveness of the application itself but also in monitoring and troubleshooting scenarios. As events are instantly delivered, any components within the system can act

near instantaneously. Any faults or issues within the system can then be quickly detected and resolved without cascading to more complex issues.

- **Increased Scalability and Fault Tolerance**
  EDA allows users to create loosely coupled or decoupled components. Thus each component of the application does not rely on the availability of each other. Even if a single component fails, others can function without issues, increasing the fault tolerance of the overall system. Furthermore, each component can be independently developed, tested, and deployed without interrupting the entire system. Additionally, each component can be scaled up or down depending on their workload. It will help to manage resources effectively and address bottlenecks in the system efficiently.

- **Extensibility**
  The decoupled approach combined with message-based communication enables developers to easily extend the system's functionality without worrying about conflicts within the system. Simply develop an independent component, stream the events and perform the necessary functions and, if required, create new events as outputs that will trigger other components within the system or supported third-party applications or platforms. Additionally, it allows developers to deploy components in the various platforms from on-premise to cloud services without worrying about communications, as event mesh will manage the distribution of events.

In Summary

With more and more applications becoming cloud-based and utilising multiple services and platforms, EDA enables developers to create simple yet flexible component-based applications without tightly coupling components. It also provides the flexibility to deploy components without being locked into a specific platform.

EDA is relatively easy to understand and implement compared to other architectures. With a simple event flow from event producer to event consumer via a broker, applications can easily be broken down into specific components that fit into this flow. Even with this simplicity, EDA provides unparalleled flexibility to build scalable and fault-tolerant applications that can process large volumes of data efficiently. Besides, the event-driven nature is well suited to deal with real-time data facilitating near-instantaneous reactions to any event. With the rapid growth of adapting EDA in the real world, utilise an event-driven architectural pattern to power your next application.