International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Optimizing Scalability and Decoupling with Event-Driven Architecture: A Cross-Industry Analysis and A Comparative Perspective

Laxman Vattam

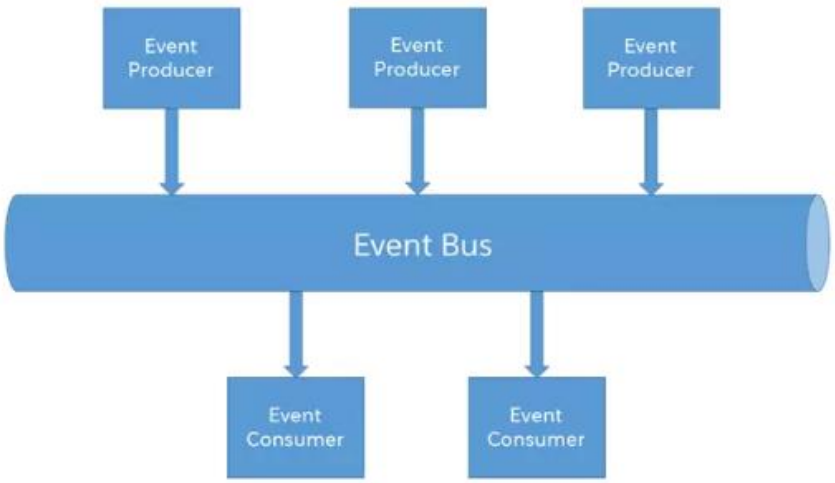
LaxmanVattam@gmail.com
Independent Researcher, Texas, USA

Abstract
Event-driven architecture (EDA) facilitates the efficient generation and processing of events, which signify changes in system states. This architectural paradigm enhances the flexibility of system interactions, streamlines cross-system processes, and enables near real-time updates. In an event-driven model, events are produced irrespective of whether consumers are actively listening, and consumers do not send acknowledgments upon receipt. The key components of EDA include event producers, event consumers, event channels, event messages, and events themselves. EDA is particularly beneficial for large-scale distributed systems as it eliminates direct dependencies between event producers and consumers, simplifying system interactions. This paper presents a comparative analysis of various EDA models, their applications across industries, and their advantages in promoting scalability while reducing system coupling. Although various event-driven architectures are examined, the primary focus of this article is on Salesforce Event-Driven Architecture and its applications.

Keywords: Salesforce, Even Driven, Publish, Subscribe, Change Data Capture, CDC, Platform Events, Apache Kafka, RabbitMQ

1. Introduction

The event-driven communication model follows a publisher-subscriber framework in which an event source broadcasts messages that multiple subscribers can consume. This can be likened to a radio transmission, where a signal is broadcasted and received by listeners tuned to the correct frequency. Events are dispatched asynchronously, and reception occurs without requiring explicit acknowledgment from consumers. This facilitates near real-time communication.



As enterprise systems expand, they become increasingly complex due to the integration of independent services with varied development cycles. This complexity necessitates a transition to asynchronous communication, where EDA provides operational autonomy and flexibility. By decoupling system components, EDA simplifies interactions, enhances scalability, and reduces the need for direct system integrations.

Core Components of Event-Driven Systems

- **Event Message:**
  Contains information about an event, such as a notification regarding an order placement.
- **Event:**
  Represents a meaningful change in state within a business process, such as order fulfillment initiation.
- **Event Channel:**
  A communication medium where producers send and consumers retrieve event messages.
- **Event Producer:**
  The source generating and broadcasting event messages.
- **Event Bus:**
  A distributed system facilitating event storage and delivery in a time-sequenced manner.
- **Event Consumer:**
  A subscriber that processes event messages upon retrieval from the event bus.
- **Event consumer**
  An event consumer is a subscriber to a channel that receives messages from the channel. For example, an order fulfillment app that gets notifications about new orders.

Event-Driven Patterns

EDA supports various architectural patterns to address different business and technical requirements.Use the below table to identify the potential patterns based on your use case:

Pattern	Near Real-Time	Unique Message Copy	Guarantee Delivery	Identify Message Recipients	Reduce Message Size	Transform Data
Publish / Subscribe (Unique Copy)	X	X	X	X		
Fanout	X	X				
Claim Check	X		X	X	X	
Passed Messages	X		X	X	X	X
Streaming	X	X	X	X		
Queueing	X	X	X	X		

Pattern Walkthroughs

Event-driven architecture encompasses several design patterns, each suited to different use cases. Some patterns serve as general-purpose solutions, functioning effectively in purely event-driven scenarios without additional requirements.

1. **Publish-Subscribe Model**
   In this model, publishers and subscribers remain loosely coupled. Publishers transmit messages to an event bus, which subsequently distributes them to any interested subscribers.
   Systems can both publish and subscribe to multiple event types.

2. **Publish-Subscribe with Unique Copies**
   This variation of the publish-subscribe model ensures that each subscriber receives a distinct copy of the message. Publishers send messages through one or more channels in the event bus, and subscribers listen to relevant channels, retrieving new messages upon arrival. This approach guarantees message delivery and allows tracking of which subscribers have received specific messages. Additionally, replay capabilities enable systems to recover past events, enhancing fault tolerance.

3. **Fanout Pattern**
   In the fanout model, messages are disseminated to one or multiple recipients via a shared message queue. Unlike the unique copy model, subscribers retrieve the same message from the queue rather than receiving individual copies. Although this approach improves system performance, it complicates message tracking, making it difficult to confirm whether a subscriber has received a particular message.

4. **Claim Check Pattern**
   Instead of transmitting the complete message payload through the event bus, the claim check pattern involves storing the message body separately. A message header containing a reference to the stored data (known as a claim check) is sent to subscribers. This approach reduces the volume of data transmitted through the event bus, ensuring that messages conform to system size constraints.

5. **Passed Messages Pattern**
   This pattern leverages a streaming message platform to manage high-volume spikes and complex data transformations. The message handling process is divided into multiple layers:
   One layer manages message routing, determining necessary transformations and final destinations.
   Another layer processes different levels of transformation, including field mappings and object relationships.
   The final component outputs the transformed message in its completed form.

6. **Streaming Pattern**
   Unlike traditional event-driven patterns, which focus on single-purpose event publishing, event streaming services generate continuous event streams. Subscribers access these streams and process events in the precise order in which they were received. Each subscriber receives a distinct copy of the message stream, ensuring guaranteed delivery and allowing precise tracking of received events.

7. **Queuing Pattern**
   In the queuing model, producers dispatch messages to message queues, where they remain until retrieved by subscribers. Most queue-based messaging systems adhere to a first-in, first-out (FIFO) structure, deleting messages once they are processed. Since each subscriber has a designated queue, additional setup is required, but this ensures reliable delivery and enables tracking of message recipients.

Salesforce Platform Events: Business Applications
Platform events facilitate real-time, bidirectional integrations by enabling event-driven interactions. Within this framework, Salesforce can function as either an event producer or a consumer, allowing seamless data communication between systems.

Salesforce as an Event Producer: Salesforce generates and transmits notifications to external order fulfillment applications regarding product shipment requests.

Salesforce as an Event Consumer: An external order fulfillment system sends shipment status updates to Salesforce regarding an order's progress.

Applications of Salesforce Platform Events

Salesforce Platform Events are particularly advantageous in scenarios that demand real-time data exchange and automation. Below are some key use cases:

1. **Optimizing Business Operations through Platform Events**
   Organizations can leverage Platform Events to automate intricate workflows efficiently. For instance, upon order placement, Salesforce can trigger an event that initiates inventory updates, payment transactions, and shipment coordination, eliminating the need for manual processing. This approach minimizes human errors and accelerates operational efficiency.

2. **Seamless Real-Time Data Synchronization with External Systems**
   Another significant advantage of Platform Events is their ability to synchronize data across external systems such as Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) solutions. By leveraging event-driven architecture, changes in Salesforce data can be immediately propagated to other platforms, ensuring consistency and real-time information across all integrated systems.

3. **Inbound Event-Driven Architecture in Salesforce**
   The need to push data into Salesforce has led to the widespread adoption of event-driven architecture, commonly implemented using streaming technologies such as Kafka and Kinesis. This approach follows a Publisher-Subscriber model, facilitating real-time data processing across interconnected systems.

   Salesforce provides multiple mechanisms to implement event-driven architecture, including Platform Events, Change Data Capture (CDC), and Push Topics (Streaming API). However, selecting the appropriate method depends on the specific use case and integration context.

Change Data Capture (CDC) for Outbound Integrations
CDC is primarily utilized for outbound integrations within Salesforce. When a data modification occurs within a Salesforce object, CDC captures this change as an event. This event then triggers the necessary processes to inform downstream systems, ensuring that dependent platforms remain synchronized.

Using Platform Events for Near Real-Time Data Ingestion
For scenarios requiring near real-time data ingestion into Salesforce, Platform Events serve as an effective solution. Third-party systems can publish events, such as Event Objects in Salesforce, based on data sourced from streaming platforms like Kafka.

However, integrating external systems with Salesforce using Platform Events involves API-based event publishing. API rate limits impose constraints on how frequently data can be ingested. For instance:

- A single event representing one record published to Kafka translates to one API call to Salesforce.
- If 20 events occur within a five-minute span, this results in 20 API calls to Salesforce.

Understanding the Integration Process
The integration between Salesforce and Kafka involves monitoring Kafka topics for new records. These records are subsequently transformed and published as Platform Events in Salesforce. However, additional transformation and processing logic is required to ensure seamless event propagation.

Given that API consumption scales in direct proportion to the data volume from the source system, organizations must consider API rate limits and organizational capacity when designing event-driven integrations.

Polling-based batch ETL processes exert excessive pressure on source data systems and lead to the ingestion of outdated information into downstream systems. In contrast, event-driven architectural patterns offer a more efficient alternative, eliminating the need for polling-based workflows and enabling real-time data processing.

Methods for Publishing Platform Events
When creating a Platform Event in Salesforce, users have two options for determining how events are published:

- **Immediate Publishing:** This method triggers the event instantly, irrespective of whether the initiating transaction completes successfully. The event is broadcast as soon as it is generated, without dependency on the final transaction state.
    - This publishing approach is particularly useful for scenarios like logging, where capturing the event is essential regardless of the transaction's outcome.
- **Publishing After Commit:** With this option, the platform event is published only if the triggering transaction completes successfully. This guarantees that subscribers receive data only after the transaction is finalized.
    - This publishing method is ideal for use cases requiring data accuracy and consistency, ensuring that events are only triggered upon transaction completion.
- **Custom Platform Events** can be created using Apex, Flow Builder, or external APIs. Events can be subscribed to within the Salesforce platform through Apex triggers, point-and-click tools, or externally via the Pub/Sub API. Upon publication, subscribers execute the associated business logic.

Event Relay Mechanism

The Event Relay system introduces an innovative approach for transmitting Platform Events or Change Data Capture events. Departing from the conventional request-response polling framework, this solution leverages an event-driven architecture to facilitate real-time data streaming directly to Amazon EventBridge. By doing so, it eliminates the dependency on middleware or custom-built code, such as the proprietary solution initially developed by Vacasa. Instead, Event Relay empowers developers to seamlessly connect their Salesforce applications with AWS services through a native, bidirectional, and event-driven integration. In this setup, events are automatically published to the Salesforce Event Bus and subsequently forwarded to Amazon EventBridge, streamlining the integration process and enhancing operational efficiency.

Event-Driven SOA for IoT Services

An Event-driven Service-oriented Architecture (EDSOA) for IoT services operates by utilizing distributed events as the fundamental mechanism through which each IoT service communicates independent and meaningful events, articulates its capabilities and requirements, and maintains loose coupling with other services. However, these distributed events lack sufficient expressiveness to fully encapsulate business logic within a Service-oriented Architecture (SOA), as business activities are inherently interconnected rather than entirely independent. To address this limitation, IoT services can be developed using resource information, executed through a combination of independent and shared events, and coordinated effectively through event sessions.

Replaying Platform Events

- Platform Events created in API version 44.0 or earlier are classified as Legacy Standard-Volume Platform Events. From API version 45.0 onward, new event definitions default to High-Volume Platform Events, which offer enhanced scalability. While Standard-Volume Platform Events remain supported, they are no longer available for new event definitions.
- High-Volume Platform Event messages are retained for 72 hours, allowing subscribers to replay events within this window. In contrast, Legacy Standard-Volume Platform Events are stored for only 24 hours.
- Event replay functionality is accessible exclusively through the Pub/Sub API, enabling subscribers to integrate replay features within their applications as required.
- Communication protocols such as CometD and gRPC facilitate interaction between Platform Events and external systems, enabling real-time synchronization and event-driven processes.
- These protocols provide a replay mechanism, allowing subscribers to navigate back to a specified point in the message stream using the ReplayID of the last received event, thereby retrieving events that occurred between that point and the present.
- For instance, if a client experiences a one-hour disconnection, it can retrieve all events that occurred since the last received event by using the corresponding ReplayID. However, if the disconnection extends beyond the replay window, the events will no longer be available for retrieval.

Comparison Between Platform Events and Change Data Capture (CDC)

While both Platform Events **and** Change Data Capture (CDC) enable data event tracking in Salesforce, their objectives and implementations differ. CDC is specifically designed for tracking modifications in Salesforce records, ensuring that changes are captured and propagated to external systems. Conversely,

Platform Events facilitate broader communication, supporting event-driven interactions between Salesforce and multiple external platforms**.** CDC is best suited for monitoring and recording data changes, whereas Platform Events provide a more flexible solution for integrating multi-system architectures.

Tools/Solutions that are similar to Platform Events that enable Event Driven API management:

The adoption of event-driven architecture has been growing significantly. A study conducted by Solace, involving 840 developers across nine countries, revealed that 85% of businesses had implemented event-driven architecture by 2021. Additionally, 72% of global enterprises reported extensive utilization of this architecture, with 71% acknowledging that its advantages outweigh any drawbacks.

One of the key drivers behind the increasing adoption of event-driven API management is the widespread shift toward microservices. Organizations are prioritizing the development of loosely coupled services and seeking real-time, continuous data exchange, further fueling interest in event-driven API solutions.

The demand for event-driven solutions continues to grow, particularly for asynchronous communication protocols such as MQTT, data streaming, and IoT applications. However, managing event-driven APIs presents unique challenges. As this approach becomes more widespread, an increasing number of specialized tools have emerged, designed specifically to support event-driven architectures.

Key Tools for Event-Driven API Management

1. **Apache Kafka**
   To implement event-driven APIs, a robust data streaming solution is essential. Apache Kafka stands out as one of the most widely adopted open-source platforms for creating data streams. Due to its extensive community support, Kafka offers a variety of libraries and tools that streamline its integration. Additionally, its compatibility with numerous technologies ensures seamless deployment, reducing the time required for troubleshooting and setup.

2. **OntoPop**
   One of the most effective ways to grasp event-driven API architecture is through practical visualization. OntoPop, an open-source tool, leverages event-driven data pipelines and APIs to illustrate various ontology versions. Its GitHub repository offers extensive documentation and hands-on examples, making it an invaluable resource for developers looking to deepen their understanding of event-driven API execution.

3. **RabbitMQ**
   Another widely used open-source tool for event-driven architecture is RabbitMQ. This message broker facilitates asynchronous communication by converting various data types into messages. Given its popularity, RabbitMQ is supported by an array of integration tools that simplify event-driven API implementation. Moreover, its reliance on HTTP-based APIs enhances ease of management and monitoring.

4. **Axway**
   Developers often struggle with redundant tasks, which can lead to inefficiencies and increased error rates. Axway offers a practical solution by allowing users to seamlessly integrate event-driven APIs into their existing API catalogs. Additionally, Axway enables the transformation of traditional APIs into event-driven models, making it a valuable tool for organizations seeking comprehensive API management solutions.

5. **Webhooks**
   A widely used technique for event-driven API integration is the webhook, which allows users to register for real-time updates. Functioning as a publicly accessible HTTP POST endpoint, a webhook enables data reception from designated sources. Unlike conventional APIs that require active querying, webhooks broadcast updates whenever relevant events occur, offering an efficient alternative to continuous polling.

6. **Gravitee**
   While tools like Apache Kafka and RabbitMQ efficiently manage in-house data streams, they lack built-in mechanisms for public-facing event-driven API management. Gravitee addresses this gap with its innovative approach to **event-native API management**. Designed specifically for event-driven systems, Gravitee also supports synchronous API interactions. With built-in connectivity features, support for publish-subscribe (pub/sub) models, and compatibility with asynchronous protocols such as MQTT and AMQP, Gravitee is an excellent choice for organizations requiring continuous public data streams.

7. **WebSockets**
   WebSockets operate similarly to webhooks but support bidirectional communication. Their setup mirrors that of webhooks, making them quick and straightforward to implement. Organizations seeking real-time, two-way communication with users should consider WebSockets as a viable solution.

Key Industry Applications of EDA:

1. **Retail:**

   - Dynamic price adjustments in response to demand variations
   - Automated inventory updates when products are added or sold
   - Instant tracking of orders and delivery status notifications

2. **Finance:**

   - Real-time fraud detection mechanisms
   - Continuous updates on market data
   - Instant alerts for stock price fluctuations

3. **Logistics:**

   - Live tracking of shipments and delivery statuses
   - Instant notifications for completed deliveries
   - Efficient management of warehouse inventory

4. **Internet of Things (IoT):**

   - Processing of sensor-generated data from interconnected devices
   - Real-time monitoring and alert systems for devices
   - Automated home adjustments triggered by sensor readings

5. **Gaming:**

   - Instant updates in multiplayer gaming environments
   - Real-time tracking of player actions and status modifications

6. **Streaming Services:**

   - Live updates on viewership metrics
   - Personalized content recommendations based on user behavior

7. **Social Media:**

   - Instant notifications for new posts, comments, and interactions
   - Real-time updates on user activities and engagements

Real World EDA examples

See below for few real-world use cases for EDA

1. **Uber – One of the Largest Implementations of Kafka**

   Uber relies heavily on Apache Kafka to support a vast array of operations. With over 300 microservices integrated into its architecture, Uber processes petabytes of data daily in near real-time. Kafka plays a crucial role in managing event-driven workflows, including transmitting event data from its rider, driver, and customer service applications, streaming database changelogs to subscribers, and processing location tracking data.

   Given the enormous scale of its data infrastructure, Uber developed Hudi, a specialized platform for building streaming data lakes, which is now an Apache Foundation project. Additionally, Uber's data ecosystem incorporates other advanced tools such as Spark, Parquet, Presto, and Hive to enhance data processing and analytics.

2. **Singapore Airlines – Utilizing ksqlDB and Kafka Streams for Predictive Maintenance**

   Singapore Airlines has adopted ksqlDB, an open-source streaming SQL engine built on Apache Kafka, to power its predictive maintenance pipeline. This system is designed to identify potential malfunctions in aircraft components before they escalate into operational issues.

   Sensor data from aircraft is ingested into Kafka, where ksqlDB filters relevant information based on parameters such as aircraft model and tail number. The filtered data is then processed through Kafka Streams, which applies essential transformations before running a machine learning algorithm to predict potential maintenance issues. The insights generated from this analysis trigger alerts, allowing preemptive action to be taken before any critical failures occur.

   This predictive maintenance framework has significantly improved operational efficiency by detecting issues early, mitigating risks, and preventing flight delays or cancellations. The system is highly scalable, fault-tolerant, and capable of seamlessly integrating new streaming analytics tools.

3. **Lyft – Enabling Ride-Sharing with Kafka**

   Lyft, a leading ride-sharing service, leverages Apache Kafka to facilitate various operations, including real-time ride-matching. When a user requests a ride through the Lyft app, the request is transmitted to a Kafka cluster. A network of microservices processes this request by identifying an available driver. Once a suitable match is found, the driver's details are instantly relayed back to the customer. This rapid process, typically completed within seconds, is made possible by Kafka's efficient event-driven architecture.

2. Conclusion

Numerous applications rely on real-time data processing and continuous monitoring to function effectively. In an era dominated by data-driven decision-making, restricting API architectures to only support querying or polling is increasingly impractical. The proliferation of technologies like the Internet of Things (IoT) is expected to further amplify the demand for event-driven approaches.

The implementation of event-driven API management is essential across various domains where immediate data responsiveness is critical. While event-driven design patterns may initially appear intricate, they are fundamentally built upon object-oriented programming principles that developers inherently aim to follow. The primary challenge lies not in understanding these patterns but in consistently incorporating them into practical applications. When designing a new feature, developers should assess whether leveraging an event-driven pattern could simplify the implementation process while enhancing the overall architectural robustness.

By adopting event-driven architectures, organizations can build scalable, adaptable systems capable of supporting business expansion without encountering performance bottlenecks. Although this architectural approach introduces certain complexities, it enables independent teams to work autonomously while contributing to the development of sophisticated applications. However, adopting an event-driven model extends beyond technological choices—it necessitates a fundamental shift in development methodologies and team collaboration. To maximize its benefits, organizations must empower developers with the flexibility to select appropriate technologies and design paradigms tailored to their specific application components.

References

1. Salesforce, "Event-Driven Architecture"
   Available: https://architect.salesforce.com/decision-guides/event-driven
2. Nordicapis, "9 Tools That Enable Event-Driven API Management,"
   Available: https://nordicapis.com/9-tools-that-enable-event-driven-api-management/
3. Medium, "Achieve Architectural Efficiency with Event Replays,"
   Available:https://medium.com/salesforce-architects/achieve-architectural-efficiency-with-event-relays-c8d4ec7f07af
4. LinkedIn, "Inbound Event Driven Architecture - Salesforce,"
   Available: https://www.linkedin.com/pulse/inbound-event-driven-architecture-salesforce-aditya-gv/
5. Estuary, "10 Event-Driven Archtecture Examples: Real-World Use Cases,"
   Available: https://estuary.dev/blog/event-driven-architecture-examples/

6. Tinybird, “Event-driven architecture best practices for databases and files,” Available:https://www.tinybird.co/blog-posts/event-driven-architecture-best-practices-for-databases-and-files
7. aws, “What is EDQA (Event-Driven Architecture)?,” Available: https://aws.amazon.com/what-is/eda/
8. waytoeasylearn, “Event-driven architecture pattern,” Available: https://waytoeasylearn.com/learn/event-driven-architecture-pattern/#google_vignette
9. geeksforgeeks, “Event-driven architecture Patterns in Cloud Native Applications,” Available:https://www.geeksforgeeks.org/event-driven-architecture-patterns-in-cloud-native-applications/?ref=ml_lbp
10. Yang Zhang, Li Duan, Jun Liang Chen, "Event-Driven SOA for IoT Services", 2014 IEEE International Conference on Services Computing