Event-Driven Architecture

Introduction to Event-Driven Architecture (EDA)

Event-Driven Architecture is a design paradigm in which the flow of the program is determined by events—these can be user actions, sensor outputs, or messages from other programs. Instead of a central controller calling services, components react to events asynchronously, promoting decoupling, flexibility, and responsiveness.

Event

An event is a factual, immutable record that captures something that has already happened in a system. It's not a command or a request; it's a declaration of a state change or a significant occurrence that can be reacted to.

Structure of an Event

An event is typically structured as a message with the following components:

Field	Description	Example
eventType	The type of the event	"OrderPlaced"
eventId	A unique identifier for this event	"abc123-order-placed"
timestamp	When the event occurred (in UTC or epoch)	"2025-06-05T12:34:56Z"
source	Who or what generated the event	"checkout-service"
data (payload)	The actual data associated with the event	{ "userId": "u789", "orderId": "o456", "amount": 99.95 }
metadata (optional)	Headers, tracing info, schema version, etc.	traceId, version, etc.

Why Are Events Immutable?

Events must not be altered once published. This ensures:

- **Traceability**: We can reconstruct the past accurately.
- **Auditability**: Good for compliance and debugging.
- **Reprocessing**: If we replay an event stream, the outcome will be consistent.

Think of events like entries in a journal you append, but never erase.

Types of Events

Domain Events

- Describe something meaningful that happened within the business domain.
- Examples:
    - "InvoiceGenerated"
    - "FlightCancelled"
    - "CustomerUpgradedToPremium"

Integration Events

- Used to communicate between services, often in a microservices environment.
- Examples:
    - "OrderShipped" sent from logistics service to billing service.

System Events

- Indicate changes or alerts from infrastructure or external systems.
- Examples:
    - "DiskSpaceLow"
    - "DatabaseFailoverTriggered"

Semantics of Events: What Do They Represent?

Each event is a fact, meaning:

- It is a conclusion, not a question or command.
- The event does not expect a response.
- Multiple systems can react independently (fan-out).

Examples in Context

Let’s take "UserSignedUp":

```
{
  "eventType": "UserSignedUp",
  "eventId": "evt-001234",
  "timestamp": "2025-06-05T10:20:00Z",
  "source": "auth-service",
  "data": {
    "userId": "user123",
    "email": "test@example.com",
    "signupMethod": "Google"
  },
  "metadata": {
    "traceId": "abc-xyz-123",
    "schemaVersion": "v1"
  }
}
```

Possible consumers:

- Email Service - sends welcome email
- Analytics Service - logs signup event
- Referral System - checks for invited-by code
- Notification System - sends push notification

State Transition Example

Consider a shopping cart:

User Action	Event Emitted	New State
Adds item	ItemAddedToCart	Cart has 1 item
Removes item	ItemRemovedFromCart	Cart is empty
Checkout	OrderPlaced	Cart is cleared, Order created

Each of these events is a fact in the timeline of the user's shopping session.

Event vs Command vs Query

Concept	Role	Initiator	Response Expected?	Mutable?
Command	Do this action	Caller	Yes	Yes
Query	Give me info	Caller	Yes	No

Concept	Role	Initiator	Response Expected?	Mutable?
Event	This happened	System	No	No

Why Events Matter in Modern Systems

- They enable reactive behavior (e.g., notify users, trigger workflows).
- They support asynchronous communication for scalability.
- They capture a log of truth for analytics and recovery.
- They reduce tight coupling between services.

Event Producer

The component that emits or publishes events. Producers are unaware of who will consume the events.

Examples:

- A user interface submitting a form
- An IoT sensor sending data
- A service detecting a change in the database

Event Consumer

The component that listens to or subscribes to events and acts upon them. A single event can trigger multiple consumers.

Examples:

- A notification service
- An analytics engine
- A billing service

Event Bus / Broker

A middleware system responsible for transporting events between producers and consumers.

Common technologies:

- Apache Kafka
- RabbitMQ

- NATS
- Azure Event Grid
- Amazon EventBridge

Asynchronous Communication

Events are often processed asynchronously, meaning the producer does not wait for the consumer to finish processing. This promotes scalability and responsiveness.

Event Flow Styles

Publish/Subscribe

- Multiple consumers can subscribe to the same event.
- All subscribers are notified whenever an event is published.

Event Notification

- The event contains only the minimal information needed to notify that something happened.
- Consumers then fetch additional data if needed.

Event-Carried State Transfer

- The event itself contains all necessary state/data, allowing consumers to operate without querying back.

Event Sourcing

- All changes to application state are stored as a sequence of events.
- State can be rebuilt by replaying events.

Key Design Patterns in EDA

Choreography

- Each service emits and listens to events, driving behavior collectively without a central orchestrator.
- Promotes autonomy and loose coupling.

Orchestration (less common in pure EDA)

- A central component issues commands based on events.
- More control, but tighter coupling.

CQRS (Command Query Responsibility Segregation)

- Separation of commands (which change state) and queries (which read state).
- Often combined with event sourcing.

Benefits of Event-Driven Architecture

Loose Coupling	Components communicate via events, not direct calls
Scalability	Consumers can be scaled independently based on workload
Resilience	Failure of one consumer doesn’t affect others
Extensibility	New consumers can be added without changing producers
Auditability	Events form a clear history of what happened
Responsiveness	Enables near real-time responses to system changes

Challenges in EDA

Event Ordering	In distributed systems, ensuring strict order of events is hard
Duplication & Idempotency	Events may be delivered more than once, consumers must handle safely
Event Schema Evolution	Changing event structure without breaking consumers is non-trivial
Debugging Complexity	Tracing issues across asynchronous flows is more difficult than in monoliths
Data Consistency	Eventual consistency vs. strong consistency trade-offs

Advanced Concepts

Idempotency

The ability for a consumer to process an event multiple times without changing the result beyond the first application.

Example: Processing a PaymentProcessed event more than once shouldn't charge the customer twice.

Eventual Consistency

In distributed EDA systems, components may not be immediately consistent. They converge over time as events are processed.

Event Time vs. Processing Time

- **Event Time**: When the event actually happened.
- **Processing Time**: When the event was received/processed. Used in stream processing engines (like Apache Flink) to handle out-of-order events.

Event Schema Management

Tools like Avro, Protobuf, or JSON Schema help manage and evolve event formats safely.

Schema Registries (e.g., Confluent Schema Registry) help ensure:

- Forward compatibility
- Backward compatibility
- Validation at publish time

Real-World Applications of EDA

Industry	**Example**
E-commerce	Order placement → Inventory update → Payment service → Shipping notification
IoT	Sensor data stream → Threshold exceeded event → Trigger alert or actuator
Finance	Transaction event → Fraud detection → Ledger update
Media Streaming	Play event → Ad tracking → Analytics dashboard update
Healthcare	Patient check-in → Prescription system → Pharmacy service → Billing

Best Practices

- Use semantic event names (UserRegistered, not Event1)
- Ensure idempotency and deduplication logic on the consumer side
- Maintain observability with tools like OpenTelemetry and centralized logging
- Set clear contracts for event formats and publish them
- Design for resilience: consumers should gracefully handle failures