Event-Driven Architecture Cheatsheet

A quick reference guide to Event-Driven Architecture (EDA) principles, patterns, and technologies. Covers key concepts, benefits, and practical implementation details.

Core Concepts

Fundamental Principles

Events: Significant state changes or occurrences within a system.

Producers: Services that emit events. They don't need to know who consumes them.

Consumers: Services that subscribe to and process events. They are decoupled from producers.

Event Router/Broker: An intermediary that receives events from producers and routes them to appropriate consumers (e.g., Kafka, RabbitMQ).

Asynchronous Communication: Producers and consumers operate independently and don't wait for direct responses.

Key Benefits

Decoupling	Services operate independently, reducing dependencies and improving resilience.
Scalability	Individual services can be scaled independently based on their event processing needs.
Flexibility	New services can be added to consume existing events without modifying producers.
Real-time Processing	Enables immediate reaction to events, supporting real-time analytics and decision-making.

Event Types

Event Notification: Simple notification about a state change. Consumers typically fetch related data.
Example: `OrderCreated`

Event-Carried State Transfer: Event contains the data needed by consumers.
Example: `OrderCreated` event includes order details.

Event-Carried Change Notification: Event contains the changed data.
Example: `OrderUpdated` event includes only updated fields.

Common Patterns

Event Sourcing

Capturing all changes to an application's state as a sequence of events. The current state can be reconstructed by replaying the events.

Benefits: Auditability, temporal queries, easier debugging.

Considerations: Event storage, replay mechanisms, eventual consistency.

CQRS (Command Query Responsibility Segregation)

Separating read and write operations. Write operations (Commands) result in events that update read models (Queries).

Benefits: Optimized read and write performance, simplified data models.

Considerations: Eventual consistency, complexity in managing separate models.

Saga Pattern

Managing distributed transactions by breaking them into a sequence of local transactions. Each local transaction publishes an event to trigger the next transaction in the saga.

Compensation Transactions: If one transaction fails, a series of compensating transactions are executed to undo the previous transactions.

Types: Choreography-based (implicit coordination) and Orchestration-based (explicit coordination).

Technology Stack

Message Brokers

Apache Kafka	High-throughput, fault-tolerant, distributed streaming platform. Suitable for large-scale event processing and data pipelines.
RabbitMQ	Versatile message broker that supports multiple messaging protocols. Good for complex routing and guaranteed delivery.
Amazon SNS/SQS	Cloud-based messaging services. SNS for pub/sub and SQS for message queues. Highly scalable and managed.

Event Processing Frameworks

Apache Flink	Distributed stream processing engine for stateful computations over unbounded data streams. Suitable for real-time analytics and complex event processing.
Apache Spark Streaming	Extension of Spark for processing real-time data streams. Supports micro-batching approach.
Spring Cloud Stream	Framework for building message-driven microservices. Provides abstractions for connecting to different message brokers.

Data Storage

Event Store: Database optimized for storing event streams. Examples: EventStoreDB, AxonDB.

NoSQL Databases: MongoDB, Cassandra, etc. Suitable for storing denormalized read models in CQRS.

Relational Databases: PostgreSQL, MySQL, etc. Can be used for read models, but may require careful optimization.

Implementation Considerations

Consistency

Eventual Consistency: Data may not be immediately consistent across all services. Requires careful handling of race conditions and conflicts.

Idempotency: Consumers should be able to process the same event multiple times without side effects.

Exactly-Once Semantics: Ensuring that each event is processed exactly once. Difficult to achieve in distributed systems. Often approximated with at-least-once delivery and idempotency.

Error Handling

Dead Letter Queues (DLQ): Events that cannot be processed are sent to a DLQ for further investigation.

Retry Mechanisms: Implement retry policies for transient errors. Use exponential backoff to avoid overwhelming the system.

Circuit Breakers: Prevent cascading failures by temporarily stopping event processing when a service is unavailable.

Monitoring and Observability

Event Tracking: Monitor event flow and processing latency.

Correlation IDs: Include a correlation ID in each event to track it across different services.

Metrics and Logging: Collect metrics about event processing and log errors and warnings.