Expert Architecture Training Program

Week 1 - Day 1: Theory Guide

Inspire Success, Your Goals & Opportunities

ISYGO Consulting Services

September 19, 2025

Contents

Introduction

Welcome to the **Expert Architecture Training Program**. This guide is designed to provide a **comprehensive understanding of Event-Driven Design (EDD)**, a paradigm where software components communicate through events—significant changes or occurrences in a system, such as "OrderPlaced" or "PaymentProcessed". Drawing from established sources like AWS and Confluent, EDD enables real-time information flow between microservices and devices, replacing polling with reactive, asynchronous notifications. Throughout this day, participants will delve into foundational concepts, advanced technologies, architectural patterns, best practices, and practical examples, building a robust foundation for designing, implementing, and managing event-driven architectures in enterprise-scale applications. As Solace notes, EDD decouples applications via event brokers, fostering agility in modern IT ecosystems.

Objectives of the Day

By the end of this training day, participants will be able to:

1. Define and explain Event-Driven Design (EDD) principles, including its core terminology like events, producers, and consumers, with references to real-world scalability gains.

2. Identify the roles and interactions of producers, consumers, brokers, topics, partitions, and replicas within event-driven systems, exploring fault-tolerance mechanisms like ISR.

3. Compare and contrast technologies such as Apache Kafka, RabbitMQ, and Pulsar, evaluating their trade-offs and use cases through benchmarks and architectural analyses.

4. Apply best practices to ensure fault-tolerant, scalable, and maintainable event-driven systems, incorporating idempotency and schema evolution.

5. Analyze multi-tenancy strategies, such as database-per-tenant or shared-schema approaches, and implement them effectively in hybrid environments.

6. Monitor, test, and log event-driven applications using tools like Prometheus and the ELK stack, with strategies for tracing asynchronous flows.

7. Connect theoretical concepts with real-world use cases, identifying common patterns and potential anti-patterns, from e-commerce inventory to banking fraud detection.

Chapter 1

Event-Driven Design (EDD)

1.1 Definition and Overview

Event-Driven Design (EDD), also known as Event-Driven Architecture (EDA), is an architectural approach where system components communicate asynchronously by producing and consuming **events**, defined as immutable records of significant state changes (e.g., "OrderPlaced", "PaymentProcessed", or "SensorTemperatureUpdated"). As per Confluent's introduction, EDA detects, processes, and reacts to real-time events, enabling loosely coupled systems that scale independently. This paradigm enhances **decoupling** by eliminating direct service calls, improves **scalability** through independent scaling of components, and boosts **responsiveness** by processing events without blocking, making it ideal for distributed, modern systems. AWS emphasizes that EDA encodes business actions into event notifications, supporting serverless and microservices patterns. Unlike request-response models, EDD promotes eventual consistency via patterns like event sourcing and CQRS, where reads and writes are segregated for performance.

1.1.1 Real-World Example

Consider an online e-commerce platform, as illustrated in GeeksforGeeks' system design: - The *Order Service* generates an `OrderPlaced` event when a customer places an order, including details like item ID and quantity. - The *Inventory Service* subscribes to this event to update stock levels in real-time, preventing overselling. - The *Notification Service* uses the same event to send a confirmation email, while a *Recommendation Engine* analyzes it for personalized suggestions. This asynchronous model ensures loose coupling, allowing each service to scale independently and handle failures gracefully. In practice, Netflix employs EDA for real-time recommendations, processing millions of viewing events daily to fan out suggestions across services, as noted in Solace's guide.

1.2 Core Principles

Drawing from AWS's best practices and Solace's complete guide, the core principles of EDD include:

1. **Decoupling:** Producers and consumers operate independently, reducing dependencies and enabling separate evolution and scaling. Publishers send events once to a

topic, with brokers distributing to subscribers, avoiding point-to-point integrations.

2. **Asynchronous Communication:** Events are sent without waiting for immediate consumer action, preventing bottlenecks and enhancing throughput. This supports real-time reactions, as in Heineken's integration of 4,500 applications for logistics and payments.

3. **Event Immutability:** Once published, events cannot be altered, ensuring consistency and enabling audit trails. As Medium's guide highlights, this facilitates replay for recovery without data corruption.

4. **Durable Event Storage:** Events are persisted in brokers or logs, supporting replay for recovery or analytics. Tyk.io stresses using durable queues to handle failures gracefully.

5. **Event Processing Modes:** Systems can handle events in real-time (streaming) or batch modes, tailored to specific business requirements. Additionally, Birlasoft adds idempotent processing to safely handle duplicates, and standardized schemas for evolution.

6. **Loose Coupling and Scalability:** As per DEV Community's 10 principles, use JSON for evolutive payloads and topic hierarchies for filtering, ensuring subscribers receive only relevant events.

Astuce (Trick): To optimize event throughput, pre-calculate partition counts based on expected load (e.g., total events / max consumer capacity) and adjust dynamically with monitoring tools like Prometheus.

These principles, when applied, transform rigid systems into resilient ones—imagine, how might idempotency prevent double-charging in a payment flow?

1.3 Key Terminologies

Event:
A timestamped, immutable record of a state change, such as `InvoiceGenerated`, carrying data like event type, payload, and metadata (e.g., source, timestamp). Confluent defines it as a notification of occurrences for real-time reaction.

Producer:
An application or service that generates and publishes events to a broker, e.g., using Kafka Producer API. Producers should publish once to one topic, per Solace's rules.

Consumer:
A service that subscribes to and processes events, such as a Kafka Consumer API implementation. Consumers filter via subscriptions, not business logic.

Topic:
A logical category or channel for grouping related events, enabling organized parallel processing. Topics use hierarchies (e.g., "ecommerce.orders.placed") for scalability.

Partition:
A subset of a topic that ensures ordered event processing and supports scalability across multiple consumers. Partitions enable horizontal scaling, with ordering per partition.

Broker:
A server or middleware (e.g., Kafka Broker) that stores, routes, and delivers events to consumers. Brokers manage replication for durability.

ISR (In-Sync Replica):
A Kafka-specific mechanism where replicas caught up to the leader form a set eligible for leadership election. As Confluent explains, writes commit only when all ISRs acknowledge, balancing durability and availability.

Schema Registry:
A centralized system (e.g., Confluent Schema Registry) storing event schemas (Avro, JSON, Protobuf) for structural consistency and backward compatibility during evolution.

Pub/Sub Pattern:
A messaging model where producers publish events to topics, and multiple consumers subscribe to receive them. AWS notes this as core to fan-out in microservices.

1.3.1 Example Scenario

In a banking application, as per IBM's overview: - A payment gateway produces `PaymentCompleted` events with details like transaction ID, amount, and account. - The fraud detection service analyzes these events in real-time for anomalies, using machine learning models triggered by the event. - The ledger service updates account balances based on the same events, ensuring atomicity via eventual consistency. This decoupled architecture minimizes inter-service dependencies, enhancing system resilience. ING Bank, for instance, uses EDA for fraud detection and personalized services, processing events across branches and apps for real-time alerts, reducing losses significantly.

Chapter 2

Messaging and Streaming Technologies

2.1 Overview

Event-driven systems depend on messaging platforms that ensure reliable event delivery, persistence, and ordering. Key technologies include:

- **Apache Kafka:** A distributed streaming platform optimized for high throughput (millions of events/sec), low latency, and durable event storage via log-based architecture. Developed at LinkedIn, it's ideal for event sourcing and analytics.
- **RabbitMQ:** A mature broker supporting AMQP, offering complex routing (e.g., exchanges) and guaranteed delivery. It's lightweight for traditional queuing but scales via clustering.
- **Apache Pulsar:** A cloud-native system with multi-tenancy and geo-replication, using BookKeeper for segmented storage. Originating from Yahoo, it separates compute from storage for elastic scaling.

These platforms, as benchmarked by Confluent, vary in architecture: Kafka's log-centric vs. Pulsar's tiered storage vs. RabbitMQ's queue model.

2.2 Comparative Analysis

To clarify trade-offs, consider this refactored single-column table, scaled to fit A4:

Feature	Details
Throughput	Kafka: High (1M+ msgs/sec); RabbitMQ: Medium (10K-100K msgs/sec); Pulsar: High (1M+ msgs/sec)
Latency	Kafka: Low (ms); RabbitMQ: Low-Medium (ms); Pulsar: Low (ms)
Persistence	Kafka: Log-based (durable); RabbitMQ: Optional (queues); Pulsar: Segmented (BookKeeper)
Scaling	Kafka: Horizontal (partitions); RabbitMQ: Clustered nodes; Pulsar: Elastic (separate storage)
Multi-Tenancy	Kafka: Limited (namespaces); RabbitMQ: Limited (vhosts); Pulsar: Native (namespaces)
Protocols	Kafka: Kafka API; RabbitMQ: AMQP, MQTT, STOMP; Pulsar: Pulsar, AMQP, MQTT
Use Case	Kafka: Streaming, analytics; RabbitMQ: Task queuing, RPC; Pulsar: Geo-replication, cloud-native

Astuce (Trick): When configuring topics, set the replication factor to match your fault-tolerance needs (e.g., 3 for 2 failures) and monitor ISR health to avoid leader imbalances.

2.3 Technology Trade-Offs

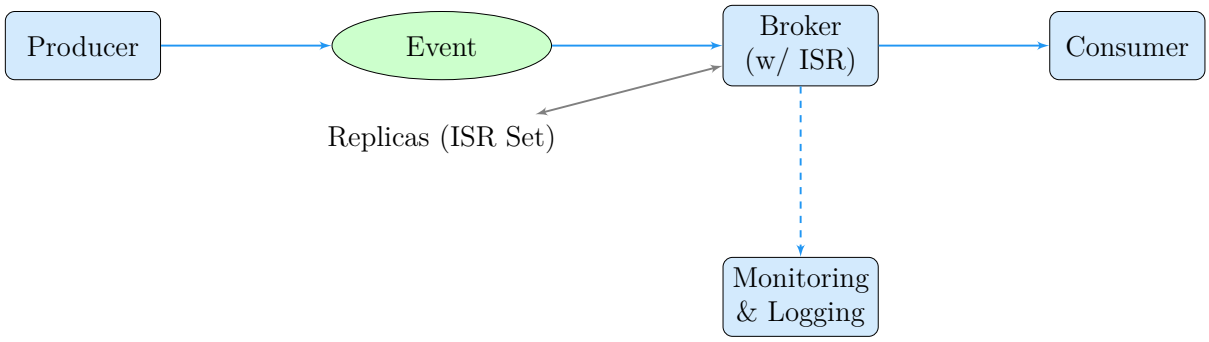
- **Kafka:** Excels in high-volume streaming and analytics (e.g., Netflix's data pipelines) but requires complex setup with ZooKeeper; best for log retention and replay, per Confluent benchmarks showing 2x Pulsar's throughput in some workloads. - **RabbitMQ:** Ideal for enterprise messaging with routing flexibility (e.g., task distribution), less suited for large-scale streaming; lightweight but struggles beyond 100K msgs/sec, as noted in ProjectPro comparisons. - **Pulsar:** Offers multi-tenancy and geo-replication (e.g., Yahoo's global feeds), with a steeper learning curve due to BookKeeper; faster at high throughput than RabbitMQ, per Confluent tests, but ecosystem lags Kafka's connectors.

How might these trade-offs influence your choice for a global e-commerce system?

Chapter 3

Event Flow and Architecture

3.1 Basic Flow Diagram



Technical Note: ISR ensures fault tolerance by maintaining synchronized replicas. Monitor ISR shrinkage (via `kafka-topics -describe`) to detect under-replicated partitions, indicating potential data loss risks.

This diagram, resized to fit the page, illustrates the flow: Producers emit events to brokers, which replicate via ISR for fault tolerance, before consumers process them asynchronously.

Chapter 4

Multitenancy in Event-Driven Systems

4.1 Definition

Multitenancy allows multiple tenants—distinct client groups or organizations—to share infrastructure while maintaining **logical isolation** of data and operations through separate namespaces or schemas. As Microsoft Learn describes, it balances cost-efficiency with isolation, using models like trusted (shared namespaces) vs. hostile (dedicated) for Event Hubs.

4.2 Strategies

- **Database-per-tenant:** Each tenant has a dedicated database, offering maximum isolation but challenging scalability at high tenant volumes; suitable for sensitive data like banking.
- **Schema-per-tenant:** Uses one database with separate schemas per tenant, balancing isolation and manageability; Azure recommends for moderate workloads.
- **Shared-database, shared-schema:** All tenants share a single database and schema, requiring code-level isolation (e.g., tenant IDs in events); cost-effective but risky for noisy neighbors, per Frontegg's pros/cons.
- **Hybrid Approaches:** Combine strategies, e.g., shared brokers with tenant-specific topics in Pulsar, as in SAP BTP's CAP for event mesh.

Trade-offs include security (higher in isolated models) vs. cost (lower in shared), with auditing essential for compliance.

4.3 Case Study: Tenant Document Processing

A tenant uploads 1TB of PDFs daily. The upload service produces `DocumentUploaded` events to a Kafka topic with 10 partitions, calculated as $\lceil 1\,\text{TB}/(100\,\text{MB/s} \times 3600\,\text{s})\rceil \approx 10$, assuming a consumer capacity of 100MB/s over a 1-hour processing window (3600

seconds). The OCR service consumes events in parallel, processing 100K PDFs/hour, with results stored in a data lake for analytics. This setup leverages Kafka's scalability, with partitions distributing load across consumers.kafka2025

Chapter 5

Monitoring, Logging, and Testing

5.1 Monitoring Metrics

As Datadog's best practices outline, holistic EDA monitoring requires:

- **Throughput:** Messages processed per second, tracked via DSM for end-to-end visibility.
- **Latency:** Time from event production to consumption, including broker delays.
- **Consumer Lag:** Unprocessed messages in partitions, alerting on backlogs.
- **Broker Health:** Disk usage, replication status, and ISR shrinkage.
- **Tools:** Prometheus/Grafana for metrics, Datadog DSM for stream tracing, CloudWatch for AWS integrations.

Centralize observability to trace asynchronous flows, correlating metrics with logs.

5.2 Logging

- **Structured Logging:** Includes tenant ID, event context, and trace IDs for traceability in distributed systems.
- **Tools:** ELK stack for aggregation, Loki for lightweight querying, Splunk for advanced analytics.
- **Importance:** Essential for debugging asynchronous pipelines; TestRail stresses comprehensive logging to capture event paths in testing.

5.3 Testing Strategies

- **Unit Tests:** Validate producer/consumer logic with JUnit/Mockito, simulating events.
- **Integration Tests:** Simulate brokers using Docker Compose/Testcontainers; DEV Community recommends event recording/replay for scenarios.

- **Schema Validation:** Ensure event compliance with registries before processing.
- **End-to-End Testing:** Use chaos engineering for fault injection, validating idempotency and ordering, per Tyk.io.

Evaluation Questions

1. How does event immutability, as in AWS EDA, enhance auditing in e-commerce inventory systems?
2. Compare Kafka's ISR with Pulsar's BookKeeper for fault tolerance in banking fraud detection.
3. What trade-offs arise in multi-tenant strategies for a global SaaS CRM, per Microsoft Learn?
4. Design a monitoring setup using Datadog DSM to trace latency in a real-time payment flow.
5. Why might RabbitMQ's routing excel over Kafka in task queuing, but falter in streaming analytics?

Glossary

- **EDD/EDA:** Event-Driven Design/Architecture, using events for asynchronous communication.
- **ISR:** In-Sync Replicas, Kafka's set of synchronized replicas for durability.
- **AMQP:** Advanced Message Queuing Protocol, RabbitMQ's core standard.
- **Pub/Sub:** Publish/Subscribe pattern for event distribution.
- **Multitenancy:** Sharing infrastructure with isolated tenant data.
- **Kafka:** Distributed streaming platform for high-throughput events.
- **RabbitMQ:** Enterprise broker for queuing and routing.
- **Pulsar:** Cloud-native platform with geo-replication.

References

- Apache Kafka Documentationkafka2025
- RabbitMQ Documentation
- Apache Pulsar Documentation
- Martin Fowler - Event-Driven Architecture
- Solace: The Complete Guide to Event-Driven Architecture
- AWS: Best Practices for EDA
- Confluent: EDA Introduction
- Datadog: Monitoring EDA
- GeeksforGeeks: EDA System Design
- Tyk: EDA Best Practices
- DEV Community: 10 EDA Principles
- Birlasoft: EDA Principles and Best Practices
- Confluent: Kafka vs. RabbitMQ vs. Pulsar Benchmark
- Confluent: Kafka vs. Pulsar Comparison
- StreamNative: Pulsar vs. Kafka
- Microsoft Learn: Multitenancy in Event Hubs
- TestRail: Testing EDA
- RisingWave: Real-World EDA Examples
- IBM: What is EDA?