Interview Questions - Event driven architecture

Fundamentals

1. What is Event-Driven Architecture (EDA), and how does it differ from traditional request-response architectures?

Event-Driven Architecture (EDA) is a software architecture pattern where system components communicate through events rather than direct synchronous calls. Instead of services invoking each other directly, they publish events when something happens, and other services listen for and react to those events asynchronously.

Differences from Request-Response Architectures:

- **Decoupling:** In a request-response system (like REST APIs), services depend on each other directly, whereas in EDA, services are decoupled and interact only through events.
- **Scalability:** EDA scales better as new services can consume events without modifying the existing ones.
- **Asynchronous Processing:** Traditional architectures require waiting for responses, but EDA enables non-blocking workflows.

---

2. Explain the difference between Pub-Sub and Event Streaming models.

Both **Publish-Subscribe (Pub-Sub)** and **Event Streaming** are used in event-driven systems but serve different purposes:

Feature	Pub-Sub	Event Streaming
Communication	One-to-many event distribution	Events stored & replayed
Persistence	Events are transient (once consumed, gone)	Events persist for later processing
Ordering	No strict ordering guarantee	Maintains strict event order
Examples	RabbitMQ, AWS SNS, Redis Pub/Sub	Apache Kafka, AWS Kinesis

- **Pub-Sub** is best for real-time notifications where consumers need only the latest

event.

- **Event Streaming** is ideal for processing historical data or event replay scenarios.

---

3. What are the key components of an event-driven system?

An event-driven system typically consists of:

1. **Event Producers** – Emit events when something happens (e.g., a user clicks a button).
2. **Event Brokers** – Middleware that routes events (e.g., Kafka, RabbitMQ, AWS EventBridge).
3. **Event Consumers** – Services that process events asynchronously.
4. **Event Store (optional)** – A log of all events for auditing or event replay.

Scalability & Fault Tolerance

4. What are some challenges of Event-Driven Architecture, and how do you handle eventual consistency?

Challenges:

- **Eventual Consistency:** Since events propagate asynchronously, data might not be updated instantly across all services.
- **Event Ordering:** Ensuring the correct order of events, especially in distributed systems.
- **Debugging Complexity:** Events flow across many services, making it harder to trace issues.
- **Handling Failures:** Consumers may fail while processing events.

Handling Eventual Consistency:

- **Use idempotent operations** to avoid duplicate updates.

- **Implement Sagas** (orchestration/choreography patterns) to ensure business logic correctness.
- **Leverage Event Sourcing** to reconstruct system state from past events.

---

5. How can you ensure event ordering in distributed event processing?

Ensuring event order is critical in systems where sequence matters. Solutions include:

- **Partitioning:** Kafka and similar brokers use partitions, ensuring order within each partition.
- **Event Versioning:** Maintain a version number for events and process them sequentially.
- **Global Ordering Service:** Use a dedicated ordering service that assigns sequence numbers to events.
- **Deduplication:** Implement event IDs to discard out-of-order duplicates.

---

6. What are dead-letter queues (DLQs), and why are they important?

A **Dead-Letter Queue (DLQ)** is a separate queue where failed or unprocessable messages are sent.

Importance of DLQs:

- **Prevents infinite retries:** Avoids messages being retried indefinitely.
- **Facilitates debugging:** Developers can inspect failed events to identify errors.
- **Ensures reliability:** Prevents faulty messages from blocking the main queue.

Example: In Kafka, a dedicated topic can store dead-lettered messages for later processing.

Implementation & Technologies

7. What are the differences between Kafka, RabbitMQ, and AWS EventBridge?

Feature	Apache Kafka	RabbitMQ	AWS EventBridge

Type	Event Streaming	Pub-Sub	Managed Event Bus
Persistence	Stores events for replay	Transient messages	No persistence
Ordering	Guaranteed within partitions	Not guaranteed	No strict order
Scalability	Highly scalable with partitions	Scales horizontally	Scales with AWS infrastructure
Use Case	Large-scale event processing	Real-time messaging	Cloud-native event integration

- **Kafka** is best for event streaming & analytics.
- **RabbitMQ** is ideal for Pub-Sub messaging and task queues.
- **AWS EventBridge** is great for integrating AWS services with event-driven workflows.

---

8. How do you make event processing idempotent to avoid duplicate execution?

Idempotency ensures that processing the same event multiple times does not produce unintended side effects.

Techniques:

- **Use Unique Event IDs:** Track processed event IDs to prevent re-processing.
- **Store Processed States:** Maintain a state in a database (e.g., "processed" flag).
- **Ensure Business Logic is Idempotent:** Avoid updating counters or timestamps without checking past values.
- **Leverage Deduplication in Brokers:** Kafka allows deduplication using log compaction.

Use Cases & Real-World Applications

9. Can you give a real-world example where Event-Driven Architecture is a better choice than a traditional architecture?

🚀 **E-commerce Order Processing System**

- When a customer places an order, multiple actions need to happen:
    - Order Service records the order.
    - Payment Service processes the payment.
    - Inventory Service updates stock.
    - Notification Service sends confirmation.
- Using EDA, each service listens for order events asynchronously, avoiding direct dependencies.
- Benefits: **Scalability, fault isolation, and flexibility to add new services (e.g., analytics, fraud detection) without modifying existing code.**

---

10. What strategies can you use to handle schema evolution in an event-driven system?

When events change over time (e.g., adding/removing fields), breaking changes can occur.

Best Practices for Schema Evolution:

- **Backward Compatibility:** Ensure new consumers can process older events.
- **Versioning:** Include a `schema_version` field in events and maintain different versions.
- **Schema Registry:** Use tools like **Kafka Schema Registry** (Avro, Protobuf) to validate schema changes.
- **Field Deprecation:** Instead of removing fields, mark them as deprecated and stop producing them gradually.
- **Event Transformation Layer:** Introduce an intermediary service that translates older events to the new schema.

Example: In **Kafka**, a Schema Registry ensures that event consumers always receive compatible data formats.