International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 3, Issue 3, PP. 20-27, 2022
ISSN: 3050-9416 | https://doi.org/10.63282/30509416/IJAIBDCMS-V3I3P103

Event-Driven Data Engineering in Microservices Architectures

Dr. Isabella Rossi,
University of Florence, AI & Digital Twin Technologies Institute, Italy.

Abstract: Event-driven data engineering in microservices architectures represents a transformative approach to building responsive and scalable systems. This architecture leverages events changes in state or updates to enable asynchronous communication between decoupled services. By implementing an event-driven architecture (EDA), organizations can achieve real-time data processing, enhancing their ability to react to changes as they occur. For instance, in a retail environment, an event-driven system can instantly update inventory levels and adjust marketing strategies based on sales transactions, fostering agility and responsiveness. The core components of EDA include event producers, which generate events; event routers, responsible for distributing these events; and event consumers, which process the events and execute necessary actions. This decoupling allows for independent scaling and deployment of services, significantly improving system reliability and reducing latency. However, challenges such as ensuring data consistency and managing event duplication must be addressed. Solutions like event sourcing and Command Query Responsibility Segregation (CQRS) can help maintain consistency across services. As organizations increasingly seek real-time insights, adopting event-driven architectures is becoming essential for enhancing business capabilities, enabling rapid decision-making, and fostering innovation in a data-centric world.

Keywords: Event-Driven Architecture, Microservices, Real-Time Data Processing, Asynchronous Communication, Event Producers, Event Consumers, Data Consistency, Scalability.

1. Introduction

1.1. Introduction to Event-Driven Data Engineering

In the rapidly evolving landscape of software development, microservices architectures have emerged as a preferred approach for building scalable and maintainable applications. This architectural style promotes the decomposition of applications into smaller, independent services that can be developed, deployed, and scaled independently. One of the most significant advancements in this domain is the adoption of event-driven data engineering. This approach focuses on the production, detection, consumption, and reaction to events changes in state that occur within a system allowing for more responsive and dynamic applications.

1.2. The Importance of Event-Driven Architectures

Event-driven architectures (EDAs) facilitate asynchronous communication between microservices, enabling them to react to events in real-time. Unlike traditional request-response models, where services depend on synchronous interactions, EDA allows services to operate independently. For instance, when a user places an order, an event is generated that can trigger various downstream processes such as inventory updates, payment processing, and shipment notifications without requiring direct communication between services. This decoupling not only enhances system resilience but also improves scalability, as services can be scaled independently based on their specific load requirements. Moreover, EDAs support real-time data processing, which is crucial for businesses aiming to leverage data-driven insights. In industries like finance or e-commerce, the ability to process and respond to events instantaneously can provide a competitive edge. For example, fraud detection systems can analyze transaction events in real-time to identify suspicious activities and take immediate action.

1.3. Challenges and Considerations

While the benefits of event-driven data engineering are substantial, there are challenges that organizations must navigate. Ensuring data consistency across distributed services can be complex due to the asynchronous nature of event processing. Additionally, managing event duplication and ensuring that events are processed exactly once are critical considerations for maintaining system integrity. To address these challenges, organizations can implement strategies such as event sourcing and Command Query Responsibility Segregation (CQRS). These approaches help maintain a clear separation between read and write operations while ensuring robust data management practices.

2. Event-Driven Data Engineering: Concepts and Principles

2.1. Definition of Event-Driven Architectures (EDA)

Event-Driven Architecture (EDA) is a software design paradigm that emphasizes the production, detection, and reaction to events significant changes in state within a system. In EDA, an event is defined as a notification that signifies a change, such as an item being added to a shopping cart in an e-commerce application or a sensor detecting a temperature change in an IoT device. This architecture allows for asynchronous communication between decoupled services, enabling them to operate independently while still being able to respond to events as they occur.

The core components of EDA include event producers, event routers, and event consumers. Event producers generate events and publish them to an event router or broker, which is responsible for filtering and distributing these events to the appropriate consumers. This decoupling of producers and consumers allows for greater flexibility and scalability, as services can be modified or scaled independently without impacting other components of the system. For instance, if a new consumer is added to process specific types of events, it can be integrated without requiring changes to the existing producers or other consumers.

EDAs are particularly beneficial in modern applications built on microservices architectures. They facilitate real-time data processing and enable systems to react promptly to user actions or external triggers. This responsiveness is crucial for applications that demand immediate feedback, such as financial services, online gaming, and real-time analytics platforms. By leveraging event-driven principles, organizations can create systems that are not only more efficient but also more resilient against failures due to their ability to handle events asynchronously.

2.2. Role of Events in Microservices-Based Data Processing

In microservices architectures, events play a pivotal role in facilitating data processing and communication among independent services. Events serve as the primary means through which microservices interact with one another, allowing them to exchange information without being tightly coupled. This loose coupling is essential for achieving the agility and scalability that microservices promise.

When an event occurs such as a user completing a purchase an event notification is generated that encapsulates relevant information about the change in state. This notification can include details like the items purchased, transaction ID, and user information. The event producer publishes this notification to an event broker or message queue, where it can be consumed by one or more event consumers that are interested in that specific type of event. For example, in an e-commerce application, different services may respond to the purchase event: one service may update inventory levels, another may initiate payment processing, and yet another may trigger shipment notifications.

Events also enable real-time data processing by allowing services to react immediately as changes occur. This capability is crucial for applications that require timely insights or actions based on user interactions or system states. For instance, in a financial trading application, market data events must be processed in real-time to inform trading decisions and maintain competitive advantage.

Moreover, events contribute to improved fault tolerance within microservices architectures. By logging events in durable storage systems, organizations can maintain an audit trail of actions taken within the system. In case of failure or error, these logged events can be replayed to restore the system to its previous state or correct inconsistencies.

2.3. Data Consistency, Event Sourcing, and CQRS (Command Query Responsibility Segregation)

Data consistency poses significant challenges in distributed systems like those built on microservices architectures. As services operate independently and communicate through asynchronous events, ensuring that all components reflect the same state at any given time can be complex. To address these challenges effectively, two key strategies event sourcing and Command Query Responsibility Segregation (CQRS) are often employed.

Event sourcing is a pattern where state changes are stored as a sequence of events rather than just storing the current state itself. Each event represents a change that has occurred within the system (e.g., "Item X was added to cart"). By maintaining a log of all these events, it becomes possible to reconstruct the current state by replaying them in order. This approach not only provides an audit trail but also allows for greater flexibility in handling changes over time since past states can be revisited or analyzed without needing complex migrations or transformations.

On the other hand, CQRS separates the responsibilities for reading data from those for writing data. In this model, commands (which modify state) are distinct from queries (which retrieve data). This separation allows each side to be

optimized independently; for instance, write operations can focus on consistency while read operations can be optimized for performance and scalability. CQRS works well with event sourcing because it enables systems to react quickly to commands by updating their state based on incoming events while providing efficient read models tailored for various consumer needs.

Together, event sourcing and CQRS enhance data consistency by ensuring that all changes are captured as discrete events and allowing different models for reading and writing data based on specific use cases. These strategies empower organizations to build robust microservices architectures capable of maintaining consistency across distributed components while facilitating real-time processing and responsiveness.

3. Architectural Patterns for Event-Driven Data Engineering

3.1. Message-Driven Microservices

Event-Driven Architecture (EDA) for Microservices, a modern approach to handling data flow and communication between loosely coupled services. In this model, services do not communicate directly but instead rely on event-based interactions to trigger operations asynchronously. This enables greater scalability, resilience, and real-time data processing.

At the starting point, events are generated by an event producer, which could be a microservice responsible for tracking user actions, database changes, or any significant system updates. These events are then placed into an event queue, a crucial component that acts as a temporary storage buffer. Message brokers like Apache Kafka, RabbitMQ, or AWS SQS are commonly used for this purpose. The queue ensures that events are not lost and can be processed in the correct order, even if consumers are temporarily unavailable.

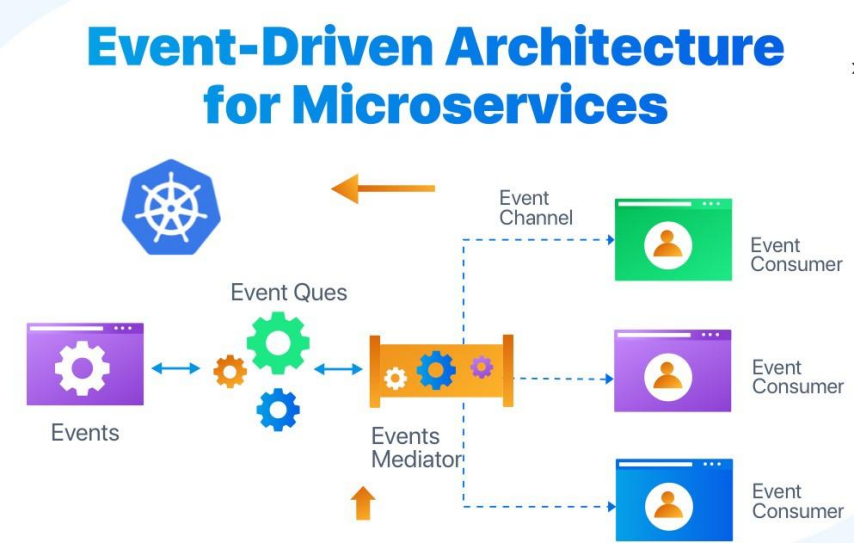

Figure 1: Event-Driven Microservices Data Flow

An event mediator is positioned between the event producer and consumers. This component ensures that events are routed appropriately based on predefined rules. It could be implemented using an event-driven framework like Kafka Streams, Apache Flink, or Google Cloud Pub/Sub, which enables advanced event processing, filtering, and transformation before dispatching them to multiple consumers.

On the right side of the diagram, multiple event consumers are shown. These microservices subscribe to the event channel and process the received events independently. This allows different services to react to the same event in different ways—for example, one service might update a database, another might trigger a notification, and yet another might start a machine learning pipeline. This decoupling ensures that system performance remains high and that failures in one service do not directly impact others.

Finally, the presence of Kubernetes (represented by the Kubernetes logo) indicates that this architecture is designed for containerized environments. Kubernetes provides orchestration, auto-scaling, and fault tolerance, making it easier to deploy and manage microservices-based applications. The entire event-driven system thus enhances agility, scalability, and real-time responsiveness, making it ideal for modern cloud-native applications.

3.2. Event Sourcing and Change Data Capture (CDC)

3.2.1. Capturing Database Changes as Events

Change Data Capture (CDC) is a technique that identifies and captures changes in a database, such as inserts, updates, and deletes, and replicates these changes in real-time to other systems. In an event-driven architecture, CDC transforms these database changes into events that can be consumed by microservices. The process begins with a log-mining component that monitors the database's transaction logs for any modifications. Once a change is detected, it is formatted as an event (e.g., "OrderCreated" or "CustomerUpdated") and published to an event bus or messaging system.

This method of capturing changes ensures that all microservices have access to the latest data without tightly coupling them to the database. By treating database changes as events, CDC allows for asynchronous communication among services, enabling them to react to changes in real-time. For instance, when a new order is placed, the order service can publish an event that triggers inventory updates and initiates payment processing across different services without direct dependencies.

3.2.2. Benefits for Microservices Data Consistency

CDC provides significant benefits for maintaining data consistency across microservices architectures. First, it establishes a clear source of truth by externalizing the state of the database into a stream of events. This allows each microservice to subscribe to relevant events and update its own state accordingly. As a result, even if services are distributed across different locations or environments, they can remain synchronized with the latest data.

Moreover, CDC enhances fault tolerance and resilience in microservices. By maintaining an event log of all changes, organizations can recover from failures more effectively. If a service fails or goes offline, it can replay the events from the log to restore its state without losing any critical data. This capability is particularly valuable in scenarios where real-time data processing is essential.

3.3. Stream Processing and Real-Time Analytics

3.3.1. Tools for Stream Processing

Stream processing is essential for handling real-time data flows in microservices architectures. Several powerful tools facilitate this process, including Apache Flink, Kafka Streams, and Spark Streaming.

Apache Flink is known for its ability to handle complex event processing with low latency and high throughput. It supports stateful computations and offers advanced features like event time processing and windowing operations.
Kafka Streams, part of the Apache Kafka ecosystem, provides a lightweight library for building applications that process streams of data. It integrates seamlessly with Kafka's messaging system, allowing developers to build robust stream processing applications using familiar Kafka concepts. Spark Streaming extends Apache Spark's capabilities to handle real-time data streams. It allows developers to process live data streams using Spark's powerful batch processing capabilities, making it suitable for scenarios where both batch and stream processing are required.

3.3.2. Use Cases and Benefits in Microservices

The integration of stream processing tools within microservices architectures offers numerous benefits and use cases. One prominent application is real-time analytics, where businesses can gain immediate insights from data as it flows through their systems. For example, e-commerce platforms can analyze customer behavior in real-time to optimize marketing strategies or adjust inventory levels dynamically based on demand. Another significant use case is fraud detection in financial services. By continuously monitoring transaction streams using tools like Apache Flink or Kafka Streams, organizations can identify suspicious activities instantaneously and take action to mitigate risks. Stream processing also enhances operational efficiency by enabling event-driven workflows across microservices. For instance, when an event occurs such as a user signing up for a service stream processing can trigger automated workflows that update user profiles, send welcome emails, and initiate onboarding processes without manual intervention.

4. Data Engineering Challenges in Event-Driven Microservices

4.1. Data Consistency and Eventual Consistency

4.1.1. Handling Distributed Transactions (SAGA Pattern)

In event-driven microservices architectures, maintaining data consistency across distributed services is a significant challenge, particularly when multiple services are involved in a single business transaction. The SAGA pattern is a widely adopted approach to manage distributed transactions by breaking them down into a series of smaller, manageable transactions. Each service involved in the SAGA performs its local transaction and publishes an event to notify other services of the outcome.

There are two main types of SAGA implementations: choreography and orchestration. In the choreographed approach, each service knows the next step to take based on the events it receives, allowing for a decentralized flow of control. Conversely, orchestration relies on a central coordinator that directs the process, managing the sequence of service calls and compensating actions if any transaction fails.

For example, consider an e-commerce application where placing an order involves multiple services: inventory management, payment processing, and shipping. When a user places an order, the inventory service checks stock levels and reserves items. If successful, it publishes an event to proceed with payment processing. If payment fails, the SAGA pattern allows for a compensating transaction to release the reserved inventory, thus ensuring that all services maintain consistent states.

4.1.2. Eventual Consistency

In many microservices architectures, achieving immediate consistency across all services is impractical due to latency and network partitioning issues. Instead, these systems often adopt eventual consistency, which accepts that temporary inconsistencies may exist but guarantees that all services will converge to a consistent state over time. This model allows services to operate independently while asynchronously propagating updates through events.

To implement eventual consistency effectively, developers must design their services to handle scenarios where data may not be immediately synchronized. This can involve utilizing patterns such as Command Query Responsibility Segregation (CQRS) and event sourcing, where state changes are logged as events that can be replayed or processed later to achieve consistency.

4.2. Scalability and Performance Considerations

4.2.1. Optimizing Message Brokers and Event-Driven Pipelines

Scalability is a crucial consideration in event-driven microservices architectures due to the dynamic nature of workloads and user interactions. As the number of events generated by various services increases, optimizing message brokers becomes essential for maintaining performance and reliability. Message brokers like Apache Kafka, RabbitMQ, or Amazon SNS act as intermediaries that facilitate communication between producers (event publishers) and consumers (event subscribers).

To optimize message brokers for scalability, organizations can employ several strategies:

- **Partitioning**: By partitioning topics in message brokers like Kafka, organizations can distribute event processing across multiple consumers. Each partition can be processed independently, allowing for parallelism that enhances throughput.
- **Load Balancing**: Implementing load balancing among consumers ensures that no single consumer becomes a bottleneck. This can be achieved through consumer groups in Kafka or round-robin distribution in RabbitMQ.
- **Retention Policies**: Setting appropriate retention policies helps manage storage costs while ensuring that events are available for processing when needed. Organizations should balance between retaining enough historical data for replaying events and managing storage efficiently.
- **Monitoring and Scaling**: Continuous monitoring of message broker performance metrics (e.g., throughput, latency) allows organizations to identify potential bottlenecks early. Auto-scaling mechanisms can be implemented to dynamically adjust resources based on workload demands.

4.2.2. Use Cases and Benefits in Microservices

The benefits of optimizing scalability and performance in event-driven architectures are significant. For instance, real-time analytics applications can process high volumes of streaming data from various sources without sacrificing performance or responsiveness. E-commerce platforms can handle spikes in traffic during sales events by scaling their microservices dynamically based on incoming orders. Moreover, optimized event-driven pipelines enhance fault tolerance by enabling services to process events independently. If one service experiences downtime or fails to process an event, other services can continue functioning without disruption.

4.3. Schema Evolution and Data Versioning

4.3.1. Handling Changes in Event Schema Over Time

In event-driven microservices architectures, schema evolution refers to the process of managing changes to the structure of events over time without disrupting existing services. As business requirements evolve, it is common for event schemas to change, necessitating careful handling to ensure compatibility between producers and consumers of events.

One effective approach to manage schema evolution is to adopt backward compatibility principles. This means that new versions of an event schema should accommodate older versions, allowing consumers that rely on previous schemas to

continue functioning without modification. For instance, if a new field is added to an event, it should be marked as optional so that consumers not expecting this field can still process the event without errors.

Various strategies can be employed for schema evolution, including:

- **Versioning Events**: Each event can include a version number indicating its schema version. This allows consumers to process events according to their expected version, enabling them to handle multiple versions of the same event type concurrently.
- **Schema Registries**: Implementing a schema registry (e.g., Confluent Schema Registry) can help manage schema versions and enforce compatibility rules. The registry can validate changes against defined compatibility types (e.g., backward, forward, or full compatibility), ensuring that producers and consumers remain aligned.
- **Event Transformation**: When an event schema changes, it may be necessary to transform old events into the new format during processing. This can be done using transformation logic in the consumer that adapts old events into a format compatible with the current business logic.
- **Decoupling Services**: By designing services to be less dependent on strict schemas, organizations can reduce the impact of schema changes. This might involve using generic data structures or employing techniques such as event sourcing, where the state is derived from a sequence of events rather than a fixed schema.

4.4. Observability and Debugging

4.4.1. Logging, Tracing, and Monitoring Event Flows

Observability is crucial in event-driven microservices architectures due to the complexity introduced by asynchronous communication between services. Effective logging, tracing, and monitoring are essential for diagnosing issues and ensuring system reliability. Logging involves capturing detailed information about events as they are produced and consumed across services. Each service should log relevant information about incoming and outgoing events, including timestamps, event types, and any associated metadata. Structured logging formats (e.g., JSON) facilitate easier parsing and analysis of logs by centralized logging systems like ELK Stack (Elasticsearch, Logstash, Kibana) or Splunk. Tracing provides insights into the flow of events through various services in the architecture. Distributed tracing tools like OpenTelemetry or Jaeger allow developers to track individual requests as they propagate through different microservices. By instrumenting services with tracing libraries, organizations can visualize how events traverse the system and identify bottlenecks or failures in real-time.

Monitoring involves continuously observing system performance metrics related to event processing. Key performance indicators (KPIs) such as event throughput, latency, error rates, and consumer lag are critical for maintaining system health. Tools like Prometheus and Grafana can be employed to collect metrics from services and visualize them in dashboards for real-time monitoring. Implementing observability practices enables teams to quickly identify issues arising from event processing delays or failures in downstream services. For example, if a consumer service fails to process an incoming event due to a schema mismatch or a transient error, observability tools can help pinpoint the root cause and facilitate rapid recovery.

5. Implementation Case Study: Event-Driven Data Engineering in Microservices Architectures

5.1. Case Study: EDEKA Group

EDEKA Group, one of Germany's leading supermarket chains, faced significant challenges with its traditional infrastructure, which relied heavily on batch updates. This approach resulted in siloed systems among its supply chain, in-store operations, merchandise management applications, and customer-facing platforms. To address these issues and enhance operational efficiency, EDEKA decided to implement an event-driven architecture (EDA).

5.2. Solution Implementation

EDEKA collaborated with a cloud consulting firm to develop a comprehensive event-driven architecture that dismantled legacy data silos. The new architecture utilized an event mesh to enable real-time master data streaming across the company's enterprise systems and touchpoints. Key components of the solution included:

- **Dynamic Event-Driven Data Exchange**: EDEKA replaced traditional batch updates with a dynamic event-driven model, allowing for immediate data sharing across all systems.
- **Flexible Platform Adoption**: The architecture supported various protocols and environments, ensuring compatibility with existing technologies while allowing for future scalability.
- **Enhanced Data Visibility**: By implementing real-time data streaming, EDEKA gained better insights into system data flows, improving decision-making processes.

The event-driven approach allowed EDEKA to create a more responsive and integrated system where changes in inventory or customer orders were immediately reflected across all platforms. This capability not only improved customer experience but also streamlined internal operations.

5.3. Benefits Realized

The implementation of EDA at EDEKA yielded several significant benefits:

- **Real-Time Omnichannel Experience**: Customers experienced instant updates regarding product availability and promotions, enhancing their shopping experience.
- **Simplified Integration**: The new architecture reduced the complexity of integrating various systems, making it easier to add or remove services without disrupting existing operations.
- **Scalability**: The event-driven model facilitated the addition of new services and functionalities as needed without impacting overall system performance.

6. Future Trends and Research Directions

As we look toward the future of event-driven data engineering in microservices architectures, several key trends and research directions are emerging that promise to shape the landscape significantly. One of the most notable trends is the increasing adoption of event-driven architectures (EDA), which enable microservices to communicate asynchronously through events. This shift allows organizations to enhance scalability, responsiveness, and decoupling among services. As industries such as e-commerce, banking, and IoT demand real-time processing capabilities, the reliance on event-driven models is expected to grow, facilitating immediate reactions to user actions and system changes.

Another critical trend is the integration of artificial intelligence (AI) and machine learning within microservices. These technologies are becoming integral to enhancing the functionality and efficiency of microservices-based applications. AI can automate various processes, enabling predictive analytics and personalized user experiences. For instance, AI-driven microservices can analyze customer behavior in real-time to optimize recommendations and improve service delivery. As organizations increasingly leverage AI capabilities, research will focus on developing more sophisticated algorithms that can operate effectively within distributed microservices environments.

The rise of serverless computing is also transforming how microservices are deployed and managed. By abstracting infrastructure management, serverless frameworks allow developers to focus on writing code without worrying about underlying resources. This trend is particularly beneficial for startups and organizations with fluctuating workloads, as it enables rapid scaling and cost optimization. Future research may explore the implications of serverless architectures on event-driven systems, particularly in terms of performance metrics and operational efficiencies.

Lastly, as microservices architectures evolve, observability and monitoring will become increasingly crucial. With the complexity introduced by asynchronous communication and distributed systems, organizations will need robust strategies for logging, tracing, and monitoring event flows. Research directions may include developing advanced tools that provide deeper insights into system performance and help identify bottlenecks or failures in real-time. Enhanced observability practices will be essential for maintaining system reliability and ensuring seamless operation across diverse microservices.

7. Conclusion

In conclusion, event-driven data engineering in microservices architectures represents a transformative approach that empowers organizations to build responsive, scalable, and resilient systems. By leveraging events as the primary means of communication between decoupled services, businesses can enhance their ability to react to real-time changes and user interactions. This architectural style not only facilitates better resource utilization and operational efficiency but also enables organizations to innovate rapidly in response to evolving market demands.

However, the journey toward implementing an effective event-driven architecture is not without its challenges. Issues such as data consistency, schema evolution, and observability require careful consideration and strategic planning. As organizations continue to embrace event-driven models, they must invest in robust solutions that address these challenges while also exploring emerging trends such as AI integration and serverless computing. By doing so, they can harness the full potential of event-driven data engineering, positioning themselves for success in an increasingly data-driven world.

Ultimately, the future of event-driven microservices is bright, with ongoing advancements promising to further enhance system capabilities and user experiences. As organizations adopt these innovative approaches, they will not only improve their operational agility but also create new opportunities for growth and differentiation in the marketplace.

References

1. DS Stream. (n.d.). Microservices in data engineering: How to break a monolith into smaller parts. Retrieved from https://dsstream.com/microservices-in-data-engineering-how-to-break-a-monolith-into-smaller-parts/
2. Red Hat Developers. (n.d.). Event-driven architecture overview. Retrieved from https://developers.redhat.com/topics/event-driven
3. Microservices.io. (n.d.). Event-driven architecture patterns. Retrieved from https://microservices.io/patterns/data/event-driven-architecture.html
4. XenonStack. (n.d.). Event-driven architecture in microservices. Retrieved from https://www.xenonstack.com/blog/event-driven-architecture
5. Volt Active Data. (2022). What is event-driven microservices architecture? Retrieved from https://www.voltactivedata.com/blog/2022/10/what-is-event-driven-microservices-architecture/
6. Kashyap, V. (n.d.). Microservices architecture: Event-driven data handling. LinkedIn. Retrieved from https://www.linkedin.com/pulse/microservices-architecture-event-driven-data-vaibhav-kashyap
7. Akamai. (n.d.). What is an event-driven microservices architecture? Retrieved from https://www.akamai.com/blog/edge/what-is-an-event-driven-microservices-architecture
8. AWS. (n.d.). Event-driven architecture on AWS. Retrieved from https://aws.amazon.com/event-driven-architecture/
9. ScyllaDB. (n.d.). Event-driven architecture fundamentals. Retrieved from https://www.scylladb.com/glossary/event-driven-architecture/
10. Solace. (n.d.). What is event-driven architecture? Retrieved from https://solace.com/what-is-event-driven-architecture/
11. Birlasoft. (n.d.). Embracing event-driven architecture: Core principles, patterns, and best practices. Retrieved from https://www.birlasoft.com/articles/embracing-event-driven-architecture-core-principles-patterns-and-best-practices
12. Confluent. (n.d.). Understanding event-driven architecture. Retrieved from https://www.confluent.io/learn/event-driven-architecture/
13. Orkes. (n.d.). Change data capture (CDC) in event-driven microservices. Retrieved from https://orkes.io/blog/change-data-capture-cdc-in-event-driven-microservices/
14. Debezium. (2020). Event sourcing vs. change data capture (CDC). Retrieved from https://debezium.io/blog/2020/02/10/event-sourcing-vs-cdc/
15. Hevo Data. (n.d.). Kafka, Debezium, and event sourcing setup. Retrieved from https://hevodata.com/learn/kafka-debezium-event-sourcing-setup/
16. InfoQ. (n.d.). CDC and microservices: Challenges and solutions. Retrieved from https://www.infoq.com/presentations/cdc-microservices/
17. Daily.dev. (n.d.). 10 methods to ensure data consistency in microservices. Retrieved from https://daily.dev/blog/10-methods-to-ensure-data-consistency-in-microservices
18. SayOne Tech. (n.d.). Managing data consistency in microservice architecture. Retrieved from https://www.sayonetech.com/blog/managing-data-consistency-microservice-architecture/
19. GeeksforGeeks. (n.d.). Event-driven APIs in microservice architectures. Retrieved from https://www.geeksforgeeks.org/event-driven-apis-in-microservice-architectures/
20. Confluent. (n.d.). Schema evolution in microservices. Retrieved from https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html
21. Akamai. (n.d.). Event-driven data management in microservices. Retrieved from https://www.akamai.com/blog/edge/what-is-an-event-driven-microservices-architecture
22. DZone. (n.d.). Event-driven microservices explained. Retrieved from https://dzone.com/articles/event-driven-microservices-1
23. Nexocode. (n.d.). Event-driven architecture in logistics: A case study. Retrieved from https://nexocode.com/blog/posts/event-driven-architecture-in-logistics-case-study/
24. xCubelabs. (n.d.). The future of microservices architecture and emerging trends. Retrieved from https://www.xcubelabs.com/blog/the-future-of-microservices-architecture-and-emerging-trends/
25. Charter Global. (n.d.). Microservices trends and future directions. Retrieved from https://www.charterglobal.com/microservices-trends/
26. Blazeclan. (n.d.). The trend of event-driven microservices architecture. Retrieved from https://blazeclan.com/india/blog/the-trend-of-event-driven-microservices-architecture/