# Order Processing System (Event-Driven Architecture)

A modern order-processing system comprises multiple decoupled microservices (e.g. User, Checkout, Order, Payment, Warehouse/Inventory, Fulfillment/Shipping, Notification). Each service handles a distinct responsibility and communicates via an event bus (Kafka) in real time. Below we enumerate the **functional and non-functional requirements**, **scalability and load challenges**, **data consistency issues**, **operational concerns**, and outline an **event-driven real-time design**.

## Requirements

* **Technical Requirements:**
* **Microservices & Messaging:** Each service (Checkout, Order, Payment, Warehouse, Fulfillment, Notification, User) is independently deployable, communicating via well-defined APIs and an event streaming platform (e.g. Apache Kafka)[[1]](https://www.confluent.io/learn/event-driven-architecture/#:~:text=Event,processing%20technologies%20such%20as%C2%A0Apache%20Kafka)[[2]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Kafka%20). Services use stateless containers (e.g. Docker/Kubernetes) for scalability and isolation.
* **Persistent Storage:** Use appropriate databases per service (e.g. relational DB for orders/payments to support transactions; NoSQL or in-memory caches for inventory and catalog). Implement the *transactional outbox* pattern: within a service’s DB transaction write both the business record (e.g. order row) and an "outbox" event record, then asynchronously publish it to Kafka[[3]](https://aws.amazon.com/blogs/mt/achieve-domain-consistency-in-event-driven-architectures/#:~:text=In%20order%20to%20facilitate%20reliable,as%20there%20is%20a%20new). This ensures atomicity (ACID) within a service before emitting events.
* **Messaging & Streaming:** Kafka is the backbone (event bus). Define domain events/topics such as OrderCreated, PaymentCompleted, StockReserved, ShippingScheduled, etc. Partition topics by key (e.g. order\_id) to preserve per-order ordering while enabling high throughput[[4]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Each%20topic%20is%20configured%20with,order_id). Use Kafka Streams or similar for real-time processing and aggregations if needed.
* **APIs and Integration:** Provide REST or gRPC APIs for front-end services (checkout, user login). Integrate with external systems (payment gateway APIs like Stripe/PayPal, shipping carriers, email/SMS providers) via either direct calls or events. Ensure idempotent interactions (e.g. include idempotency keys on payment requests).
* **Security & Compliance:** Secure APIs with authentication (OAuth/JWT) and enforce TLS encryption. The payment component must comply with **PCI DSS** standards for handling cardholder data[[5]](https://www.bigcommerce.com/articles/ecommerce/pci-compliance/#:~:text=PCI%20DSS%20are%20standards%20all,credit%20card%20must%20abide%20by). Protect user data per privacy laws (GDPR, CCPA) if applicable.
* **Real-Time & Fault Tolerance:** Design for real-time responsiveness – events are processed as they arrive with low latency. Employ patterns like circuit breakers and retries for fault tolerance (e.g. if a downstream service is unavailable). Use consumer acknowledgments and dead-letter queues for unprocessable events.
* **Observability & DevOps:** Instrument all services with logging, metrics, and distributed tracing. Use a monitoring stack (Prometheus/Grafana, ELK, or APM) to track throughput, latency, and errors. Automate deployment with CI/CD pipelines and configuration management. Embed monitoring in the pipeline to catch regressions before production[[6]](https://lumigo.io/microservices-monitoring/#:~:text=Another%20significant%20challenge%20is%20identifying,pipeline%20to%20guarantee%20the).
* **Non-Technical (Business/Operational) Requirements:**
* **High Availability & Scalability:** Target continuous 24×7 operation with minimal downtime. The system must handle variable traffic (promotions, holidays) without failing (e.g. Black Friday spikes[[7]](https://www.rigbyjs.com/blog/ecommerce-scalability-and-flexibility#:~:text=Companies%20like%20Amazon%20and%20Alibaba,manage%20traffic%20during%20major%20events)). Services must scale horizontally (auto-scaling). Data stores (DBs, Kafka) should be replicated and sharded.
* **Performance:** Provide fast checkout (e.g. end-to-end order placement within hundreds of ms). Pages and services should meet SLAs (e.g. 99th percentile latencies under thresholds, similar to e-commerce norms). As 47% of customers expect pages <2s[[8]](https://www.rigbyjs.com/blog/ecommerce-scalability-and-flexibility#:~:text=Failing%20to%20manage%20traffic%20surges,Source), the API responsiveness and backend processing should be optimized (caching catalog, denormalized read models).
* **Reliability & Resilience:** Ensure *exactly-once* processing or idempotency to avoid duplicate orders/payments. Maintain data integrity (no inventory oversell, no double charges). Have graceful degradation: e.g. if the notification service is down, continue processing orders and send notifications later (eventual delivery).
* **Compliance & Security:** Besides PCI, follow industry regulations (e.g. audit logging for financial transactions). Conduct regular security reviews and penetration tests.
* **Business Continuity:** Define recovery time objectives (RTO) and recovery point objectives (RPO). Implement backups, multi-region failover (if global). Maintain runbooks and on-call procedures for incidents.

## Scalability and Load

* **High Throughput:** Even without a fixed target, plan for thousands of orders per second. Large e‑commerce sites experience enormous peaks (Alibaba handles ~544,000 orders/s during peak events[[7]](https://www.rigbyjs.com/blog/ecommerce-scalability-and-flexibility#:~:text=Companies%20like%20Amazon%20and%20Alibaba,manage%20traffic%20during%20major%20events)). Realistically, mid-size systems might see 100–1,000 orders/s on major sales days, and bursts beyond that.
* **Partitioning & Parallelism:** Kafka topics should have many partitions (e.g. 12–24) to allow parallel consumers[[4]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Each%20topic%20is%20configured%20with,order_id). Partitioning by order\_id (or customer ID) preserves per-order event ordering while distributing load across partitions[[4]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Each%20topic%20is%20configured%20with,order_id). Similarly, shard databases by customer or region to spread load.
* **Stateless Services:** Design services stateless (except database state) so they can scale out easily. Use load balancers and container orchestrators to add instances during peaks. For example, several instances of the OrderService, PaymentService, etc., each in its own Kafka consumer group, can process distinct partitions in parallel.
* **Database Scaling:** Use read replicas or NoSQL for high-volume read workloads (e.g. product catalog). For writes (orders/payments), employ scaled SQL databases (sharding or multi-master) if load is high. Use caching (Redis/Memcached) for session/cart data.
* **Peak Traffic Handling:** Handle sudden spikes (e.g. flash sales) by elastic scaling and pre-warming caches. Serve static content via CDNs. Implement back-pressure: if downstream (e.g. payment gateway) is slow, Kafka will queue events, but monitor consumer lag to avoid backlog.
* **Latency Constraints:** Real-time order processing demands low end-to-end latency. Profile each stage (payment gateway response time, DB writes, Kafka publish/consume). Use asynchronous flows to avoid synchronous bottlenecks.
* **Multi-Region & Global Scale:** If serving global users, replicate services/data across regions. Use regional Kafka clusters or cross-region topics to reduce latency. Handle cross-region consistency via careful event design (e.g. user selects region).

## Data Integrity and Consistency

* **Transactional Boundaries:** Within each service’s database, use ACID transactions where needed (e.g. debiting inventory stock, recording payment). However, distributed ACID across services is impractical. Instead, use *sagas* or compensating transactions[[9]](https://dev.to/dinesh_dunukedeniya_539a3/designing-e-commerce-order-processing-orchestration-vs-choreography-2k6d#:~:text=)[[3]](https://aws.amazon.com/blogs/mt/achieve-domain-consistency-in-event-driven-architectures/#:~:text=In%20order%20to%20facilitate%20reliable,as%20there%20is%20a%20new). For example, if the warehouse reports *out of stock*, trigger a refund via a compensation event to PaymentService, or cancel the order. If any step fails irrecoverably, roll back by emitting compensating events.
* **Eventual Consistency:** Embrace eventual consistency across services. For instance, the OrderService can publish OrderCreated, then other services (Payment, Warehouse) update their local state asynchronously. The Order entity’s final status (confirmed/failed) is determined after all events resolve.
* **Outbox Pattern:** Ensure that updates and event emissions are atomic using the transactional-outbox pattern[[3]](https://aws.amazon.com/blogs/mt/achieve-domain-consistency-in-event-driven-architectures/#:~:text=In%20order%20to%20facilitate%20reliable,as%20there%20is%20a%20new). This prevents a service from updating its database without emitting the corresponding event or vice versa. A helper process reliably reads the outbox table and publishes to Kafka.
* **Idempotency and Ordering:** Consumers must be idempotent: processing the same event twice has the same effect as once[[10]](https://microservices.io/post/microservices/patterns/2020/10/16/idempotent-consumer.html#:~:text=Handling%20duplicate%20messages%20using%20the,as%20processing%20the%20message%20once). Implement idempotency keys (e.g. unique order/payment IDs) so retries/duplicates do not double-charge or double-deduct stock. Kafka’s per-partition ordering (with order\_id as key) maintains event order for each order[[4]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Each%20topic%20is%20configured%20with,order_id), which is crucial for correctness (e.g. never process PaymentCompleted before OrderCreated).
* **Data Validation:** Consistency checks (e.g. negative stock, order total mismatches) should trigger alerts or rollbacks. Use database constraints or application checks. Perform reconciliations (e.g. nightly batch comparing inventory records) to catch missed events.
* **Consistency vs Availability:** In line with CAP theorem, prioritize consistency for critical data (payments, inventory) and availability for non-critical parts (notifications). Where possible, use strong consistency (SQL transactions) for inventory reservation and defer less critical updates (e.g. sending confirmation emails) to asynchronous flows.
* **Schema Evolution:** Version events carefully. Use a schema registry (e.g. Avro/Protobuf) to evolve event formats without breaking consumers[[11]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=,using%20Prometheus%20or%20Kafka%20Manager). Maintain backward compatibility or handle migrations gracefully.

## Operational Challenges

* **Monitoring & Observability:** Tracking a distributed system is complex. Implement centralized logging and tracing (e.g. use correlation IDs on events to trace a request flow across services[[12]](https://lumigo.io/microservices-monitoring/#:~:text=A%20major%20challenge%20to%20monitoring,user%20experience%20is%20also%20challenging)). Monitor key metrics: requests per second, consumer lag, error rates, success/failure counts[[13]](https://lumigo.io/microservices-monitoring/#:~:text=Platform%20Metrics). Dashboards and alerts should cover Kafka health (broker status, ISR), DB health, and each service’s API performance.
* **Service Coordination:** With many services, dependency management is challenging[[14]](https://www.opslevel.com/resources/challenges-of-implementing-microservice-architecture#:~:text=Managing%20a%20large%20number%20of,rolling%20updates%20and%20system%20resilience). Use a service registry or API gateway to manage endpoints. Deploy orchestrated infrastructure (Kubernetes) for lifecycle management (rollouts, rolling restarts). Container orchestration (K8s) simplifies upgrades and provides resilience (autorestarts, pod health checks).
* **Configuration & Deployment:** Maintain consistent configuration (secrets, API keys for gateways, Kafka brokers). Automate deployments (CI/CD with canary or blue/green). Ensure backward compatibility: e.g. when updating event schemas or API versions, deploy consumers before producers.
* **Fault Recovery:** Design for fail-stop and recovery. For example, if a service crashes, Kubernetes should restart it automatically. If a Kafka broker fails, replicas should take over without data loss. Regularly test failover and restore procedures.
* **Scaling Kafka:** Manage the Kafka cluster: monitor disk usage (log segments), reassign partitions on new brokers, tune retention. Backup critical data (e.g. store long-term events in cold storage if needed). Handle Kafka upgrades and configuration changes carefully (rolling restarts of brokers).
* **Data Backups and Migrations:** Plan for database schema changes. Use migrations tools and version control. Back up databases and have tested restore processes. For event stores (if using event sourcing), consider log compaction to retain only current state where appropriate[[15]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=%E2%9C%85%203).
* **Security Operations:** Keep Kafka/DB secure (TLS, ACLs). Rotate credentials and certificates regularly. Monitor for suspicious activity (e.g. unusual data volumes). Comply with audit logging requirements.
* **Team and Process:** Assign service ownership and on-call rotations. Maintain runbooks for common incidents (e.g. consumer lags, DB deadlocks). Use chaos engineering to uncover hidden faults. Foster a DevOps/SRE culture to proactively improve reliability[[16]](https://www.opslevel.com/resources/challenges-of-implementing-microservice-architecture#:~:text=Challenge%205%3A%20Operational%20Complexity%20and,DevOps%20Culture).

## System Design (Event-Driven, Real-Time)

*Figure: Event-driven order processing flow. Services communicate via Kafka events in an asynchronous choreography.* The architecture centers on **Kafka** as the event bus[[2]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Kafka%20). For example: when a customer places an order, the Checkout or OrderService publishes an OrderCreated event (topic key = order\_id) to Kafka. PaymentService subscribes to this event and attempts to charge the customer, then emits PaymentCompleted or PaymentFailed. A Warehouse/Inventory service listens and reserves stock, emitting StockReserved or StockOut. FulfillmentService (or ShippingService) reacts to reservation success and schedules delivery (ShipmentScheduled). Finally, NotiService sends confirmations to the user. UserService provides account data when needed (e.g. shipping address), and PaymentService interacts with external gateways. Each service also updates its own database (and local query views) based on events, implementing CQRS if needed[[17]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=2,Segregation).

In this **real-time event-driven choreography**, services are loosely coupled: they depend only on the defined event schema, not on synchronous calls. Kafka partitions ensure high throughput and per-order ordering[[4]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Each%20topic%20is%20configured%20with,order_id). Each topic is *domain-aligned* (e.g. order.\*, payment.\*, inventory.\*), facilitating clear responsibility[[18]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Let%E2%80%99s%20dive%20in%21)[[19]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Order%20Service%20emits%3A). We employ **event sourcing**: every state change (order created, paid, shipped) is an event, so the full history is available for auditing or replay[[20]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=1). This yields real-time responsiveness and fault tolerance: on failure, consumers can replay missed events from Kafka to rebuild state[[21]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=%2A%20Real,Clean%20separation%20of%20concerns).

To guarantee consistency where needed, we use **sagas**: a Saga Orchestrator (or choreography logic) ensures that if any step fails, compensating actions are triggered (e.g. refunding payment if stock cannot be reserved)[[9]](https://dev.to/dinesh_dunukedeniya_539a3/designing-e-commerce-order-processing-orchestration-vs-choreography-2k6d#:~:text=). Internally, each service uses database transactions for its own data, and emits events via an outbox table[[3]](https://aws.amazon.com/blogs/mt/achieve-domain-consistency-in-event-driven-architectures/#:~:text=In%20order%20to%20facilitate%20reliable,as%20there%20is%20a%20new). For example, OrderService writes the new order row and an order.created outbox record in one transaction, then a daemon publishes the event to Kafka.

Real-time requirements are met by using **Kafka Streams or consumer clients** in each service to process events immediately upon arrival. This decoupling allows horizontal scaling: many consumers can process different partitions in parallel[[22]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=4). Overall, this design achieves the stated requirements: **resilience** (services isolated by Kafka), **scalability** (partitioned topics, stateless services), **consistency** (transactions + sagas + idempotency) and **observability** (event logs and metrics)[[21]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=%2A%20Real,Clean%20separation%20of%20concerns)[[23]](https://www.confluent.io/learn/event-driven-architecture/#:~:text=EDA%20has%20advantages%20such%20as,effective%20event%20modeling%20and%20management).

**Sources:** We have drawn on best practices for event-driven e-commerce architectures[[24]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Imagine%20a%20customer%20places%20an,order%20goes%20through%20multiple%20stages)[[25]](https://www.confluent.io/learn/event-driven-architecture/#:~:text=Event,events%2C%20facilitating%20system%20reconstruction%20at)[[3]](https://aws.amazon.com/blogs/mt/achieve-domain-consistency-in-event-driven-architectures/#:~:text=In%20order%20to%20facilitate%20reliable,as%20there%20is%20a%20new)[[4]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Each%20topic%20is%20configured%20with,order_id) and on guidelines for distributed system operations[[14]](https://www.opslevel.com/resources/challenges-of-implementing-microservice-architecture#:~:text=Managing%20a%20large%20number%20of,rolling%20updates%20and%20system%20resilience)[[26]](https://lumigo.io/microservices-monitoring/#:~:text=Another%20significant%20challenge%20is%20identifying,performance%20of%20new%20code%20releases). Notably, dividing the order process into events (created, paid, fulfilled, shipped) decouples services and supports real-time workflows[[24]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Imagine%20a%20customer%20places%20an,order%20goes%20through%20multiple%20stages). Partitioning by order ID ensures scalable parallelism without losing order accuracy[[4]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Each%20topic%20is%20configured%20with,order_id). The transactional outbox and saga patterns keep data consistent across this async environment[[3]](https://aws.amazon.com/blogs/mt/achieve-domain-consistency-in-event-driven-architectures/#:~:text=In%20order%20to%20facilitate%20reliable,as%20there%20is%20a%20new)[[9]](https://dev.to/dinesh_dunukedeniya_539a3/designing-e-commerce-order-processing-orchestration-vs-choreography-2k6d#:~:text=).

[[1]](https://www.confluent.io/learn/event-driven-architecture/#:~:text=Event,processing%20technologies%20such%20as%C2%A0Apache%20Kafka) [[23]](https://www.confluent.io/learn/event-driven-architecture/#:~:text=EDA%20has%20advantages%20such%20as,effective%20event%20modeling%20and%20management) [[25]](https://www.confluent.io/learn/event-driven-architecture/#:~:text=Event,events%2C%20facilitating%20system%20reconstruction%20at) Event-Driven Architecture (EDA): A Complete Introduction

<https://www.confluent.io/learn/event-driven-architecture/>

[[2]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Kafka%20) [[4]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Each%20topic%20is%20configured%20with,order_id) [[11]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=,using%20Prometheus%20or%20Kafka%20Manager) [[15]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=%E2%9C%85%203) [[17]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=2,Segregation) [[18]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Let%E2%80%99s%20dive%20in%21) [[19]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Order%20Service%20emits%3A) [[20]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=1) [[21]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=%2A%20Real,Clean%20separation%20of%20concerns) [[22]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=4) [[24]](https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26#:~:text=Imagine%20a%20customer%20places%20an,order%20goes%20through%20multiple%20stages) 414. Scalable Order Management with Kafka: Topic Strategy, Partitioning, and Design Patterns | by Ilakkuvaselvi (Ilak) Manoharan | Jul, 2025 | Medium

<https://medium.com/@ilakk2023/414-scalable-order-management-with-kafka-topic-strategy-partitioning-and-design-patterns-168d5600de26>

[[3]](https://aws.amazon.com/blogs/mt/achieve-domain-consistency-in-event-driven-architectures/#:~:text=In%20order%20to%20facilitate%20reliable,as%20there%20is%20a%20new) Achieve domain consistency in event-driven architectures | AWS Cloud Operations Blog

<https://aws.amazon.com/blogs/mt/achieve-domain-consistency-in-event-driven-architectures/>

[[5]](https://www.bigcommerce.com/articles/ecommerce/pci-compliance/#:~:text=PCI%20DSS%20are%20standards%20all,credit%20card%20must%20abide%20by) PCI Compliance: Requirements Explained + PCI DSS Checklist

<https://www.bigcommerce.com/articles/ecommerce/pci-compliance/>

[[6]](https://lumigo.io/microservices-monitoring/#:~:text=Another%20significant%20challenge%20is%20identifying,pipeline%20to%20guarantee%20the) [[12]](https://lumigo.io/microservices-monitoring/#:~:text=A%20major%20challenge%20to%20monitoring,user%20experience%20is%20also%20challenging) [[13]](https://lumigo.io/microservices-monitoring/#:~:text=Platform%20Metrics) [[26]](https://lumigo.io/microservices-monitoring/#:~:text=Another%20significant%20challenge%20is%20identifying,performance%20of%20new%20code%20releases) Microservices Monitoring: Challenges, Metrics & Tips for Success

<https://lumigo.io/microservices-monitoring/>

[[7]](https://www.rigbyjs.com/blog/ecommerce-scalability-and-flexibility#:~:text=Companies%20like%20Amazon%20and%20Alibaba,manage%20traffic%20during%20major%20events) [[8]](https://www.rigbyjs.com/blog/ecommerce-scalability-and-flexibility#:~:text=Failing%20to%20manage%20traffic%20surges,Source) Scalability and Flexibility in E-commerce: Handling Traffic Spikes and Maintaining User Experience | Rigby Blog

<https://www.rigbyjs.com/blog/ecommerce-scalability-and-flexibility>

[[9]](https://dev.to/dinesh_dunukedeniya_539a3/designing-e-commerce-order-processing-orchestration-vs-choreography-2k6d#:~:text=) Designing E-Commerce Order Processing: Orchestration vs Choreography - DEV Community

<https://dev.to/dinesh_dunukedeniya_539a3/designing-e-commerce-order-processing-orchestration-vs-choreography-2k6d>

[[10]](https://microservices.io/post/microservices/patterns/2020/10/16/idempotent-consumer.html#:~:text=Handling%20duplicate%20messages%20using%20the,as%20processing%20the%20message%20once) Handling duplicate messages using the Idempotent consumer pattern

<https://microservices.io/post/microservices/patterns/2020/10/16/idempotent-consumer.html>

[[14]](https://www.opslevel.com/resources/challenges-of-implementing-microservice-architecture#:~:text=Managing%20a%20large%20number%20of,rolling%20updates%20and%20system%20resilience) [[16]](https://www.opslevel.com/resources/challenges-of-implementing-microservice-architecture#:~:text=Challenge%205%3A%20Operational%20Complexity%20and,DevOps%20Culture) Challenges of Implementing Microservice Architecture

<https://www.opslevel.com/resources/challenges-of-implementing-microservice-architecture>