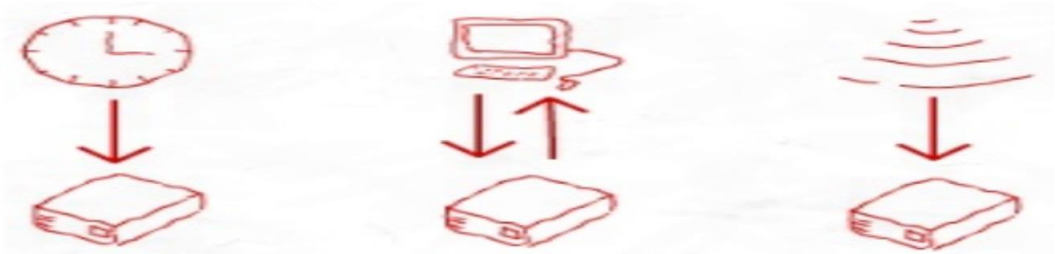
Software Systems Architecture

Event-Driven Architecure

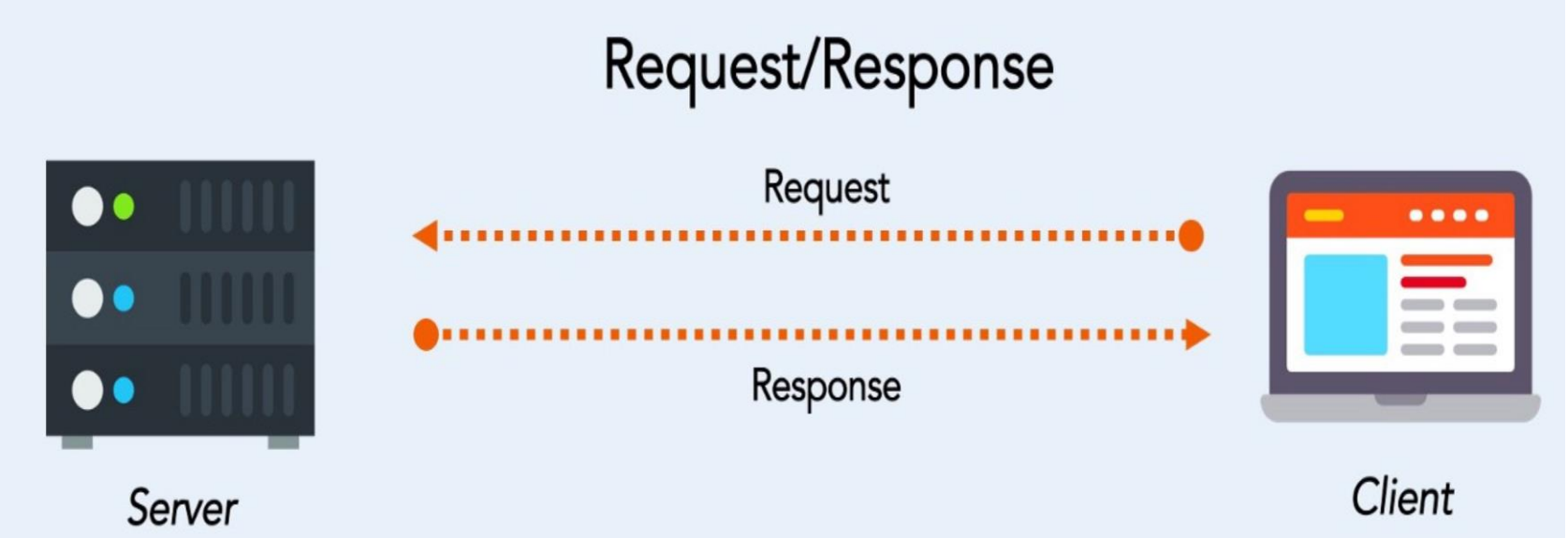
Poștaru Andrei

Communication Styles

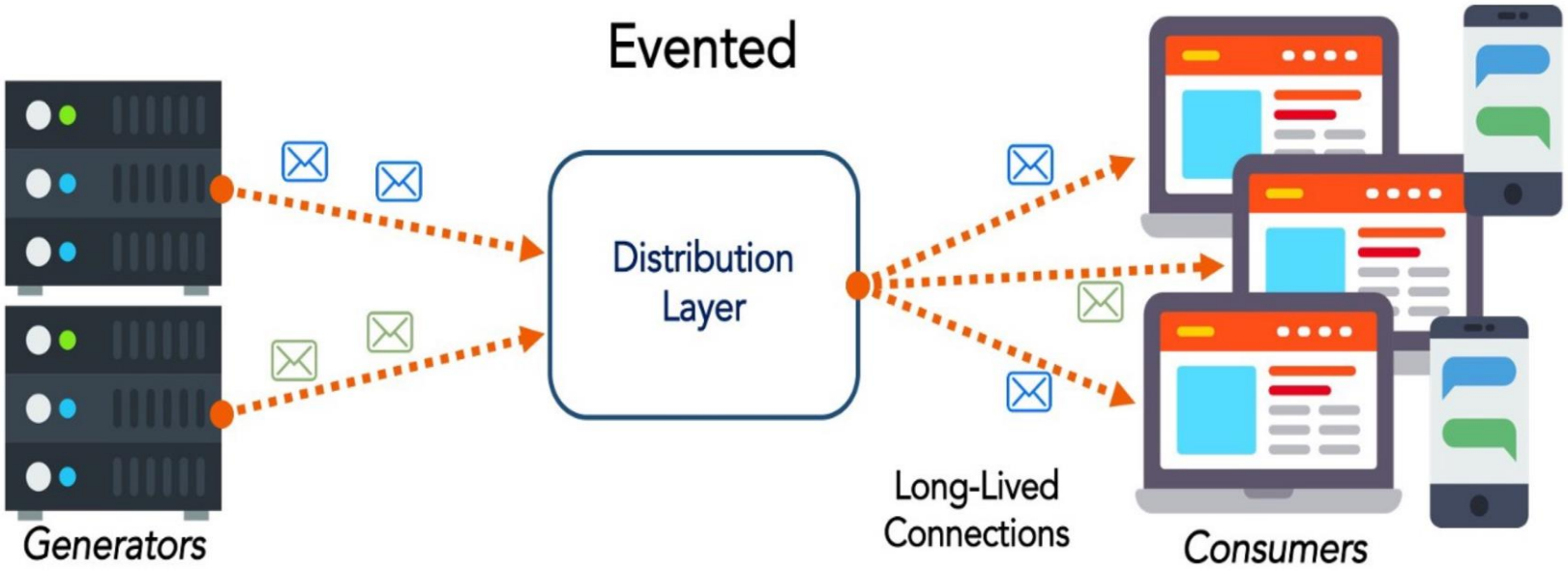
Type of Interaction	Initiator	Participants
Time-driven	Time	The specifice system
Request-driven	Client	Client & Server
Event-driven	event	Open-ended



Request-driven



Event-driven



Event-driven

- Anything happened (or didnt happen).
- A change in the state.
- An event is always named in the past tense and is immutabled
- A condition that triggers a notification.

CustomerAddressChanged

InventoryUpdated

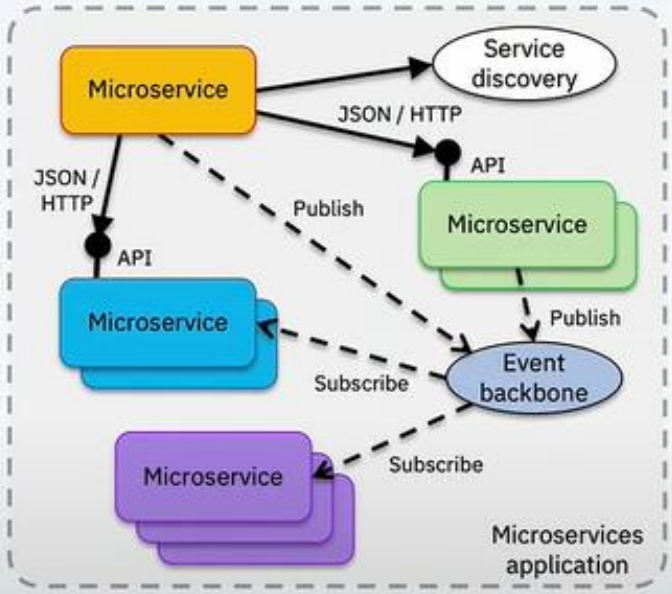
SalesOrderCreated

PurschaseOrderCreated

Characterstics of Events

- “Real-time” events as they happen at the producer
- Push notifications
- One-way “fire-and-forget”
- Immediate action at the consumers
- Informational (“someone logged in”), not commands (“audit this”)

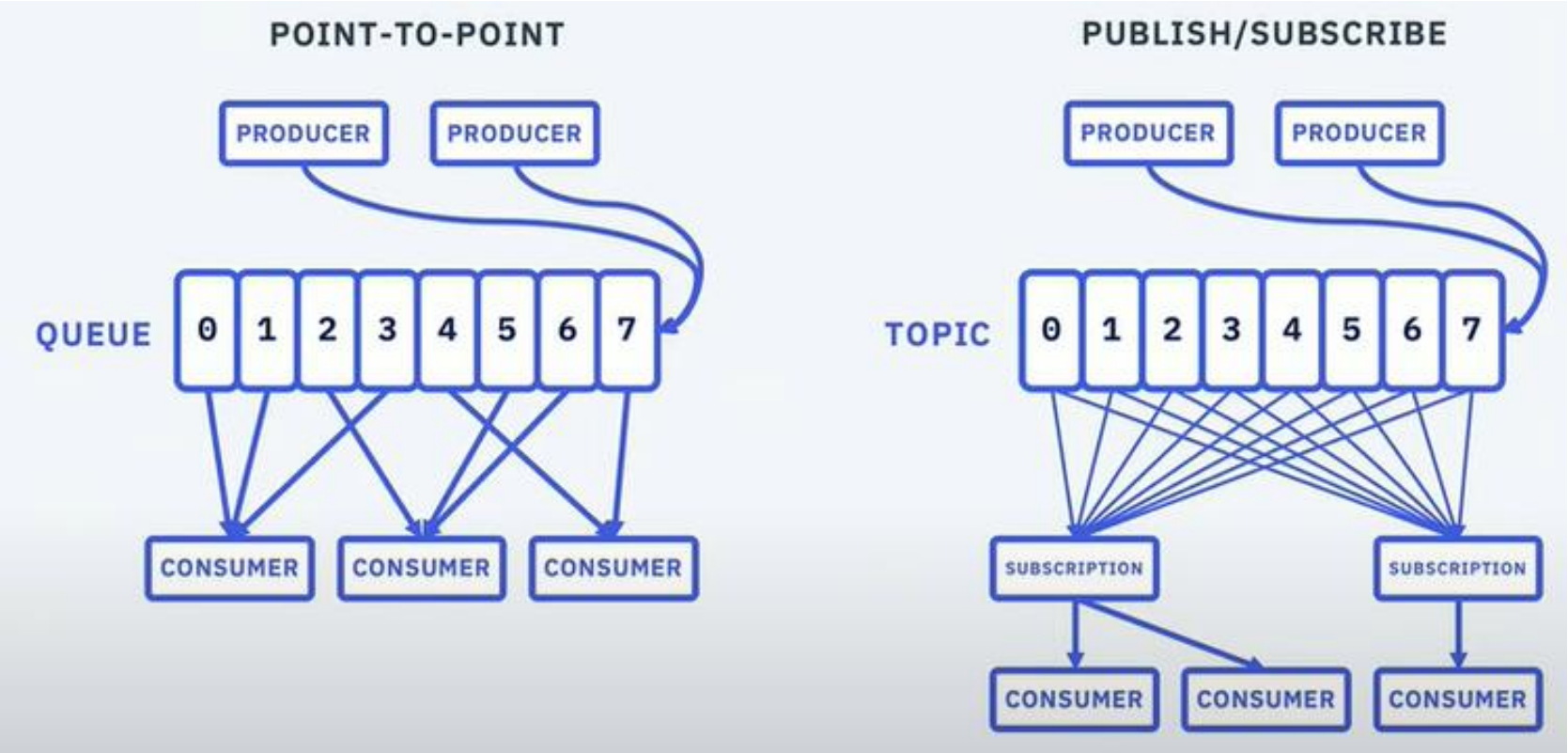
Event-driven microservices



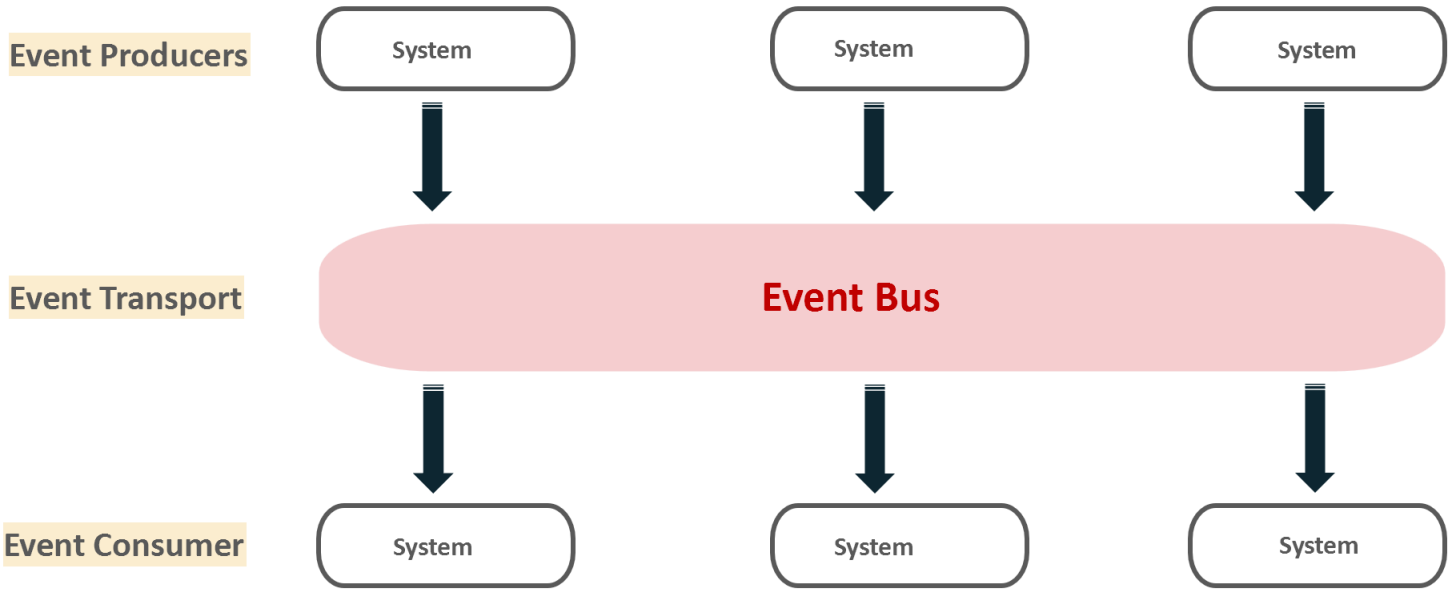
Microservices communicate primarily with events, with APIs where required

- Microservices can produce and consume events using publish/subscribe messaging
- Events are handled by an event backbone
- Data is eventually consistent

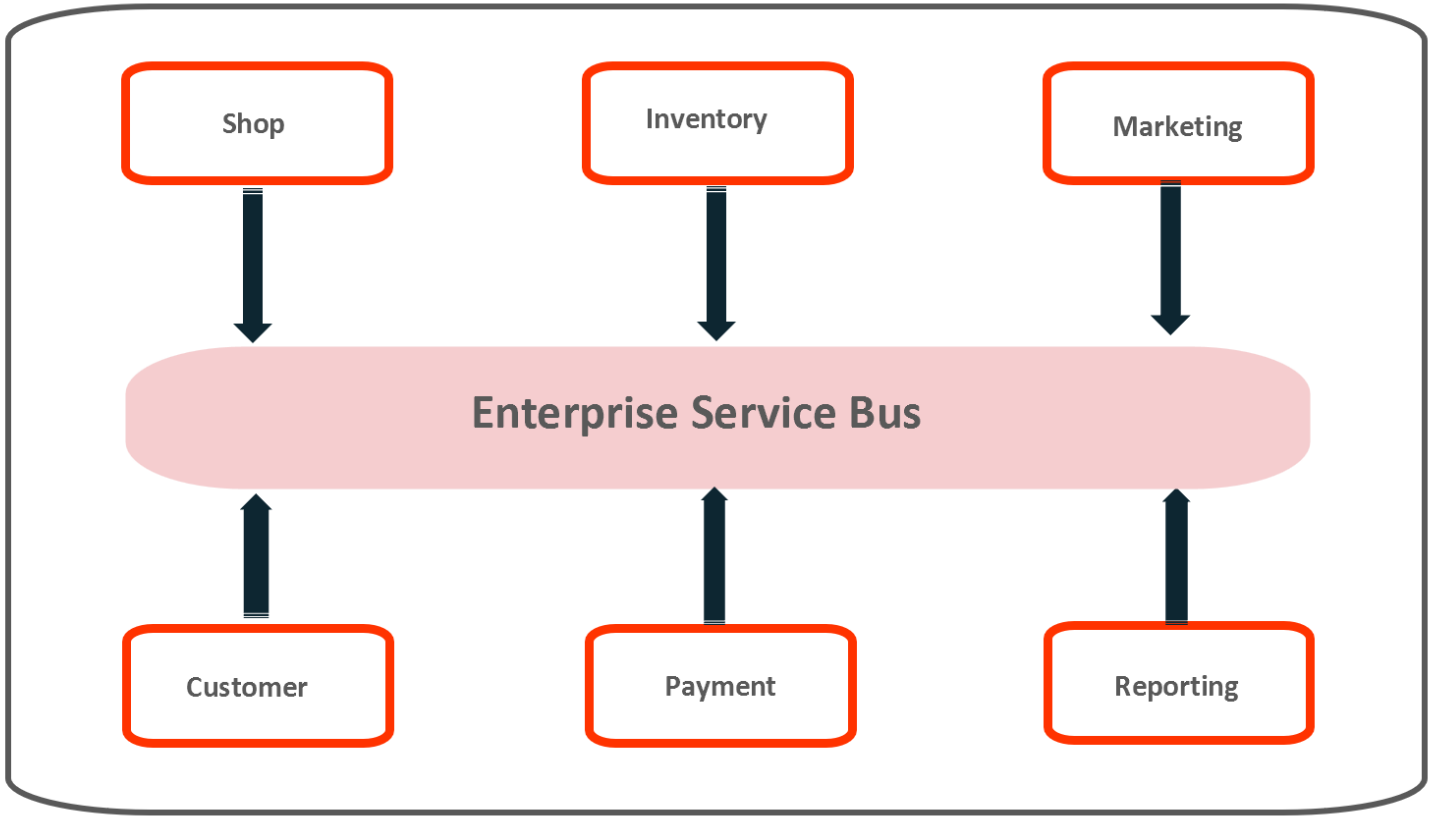
Messaging Patterns



Typical EDA Architecture



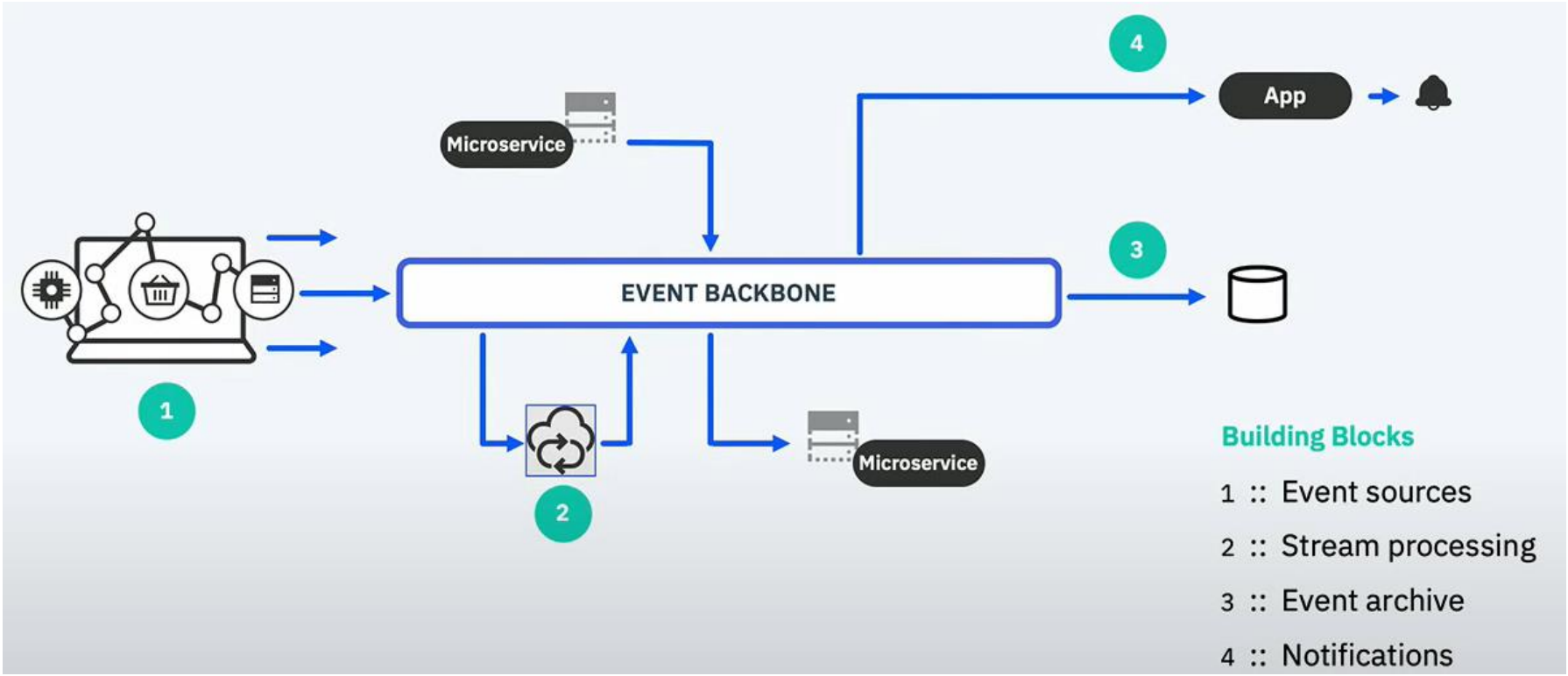
Traditional Archiecture- ESB



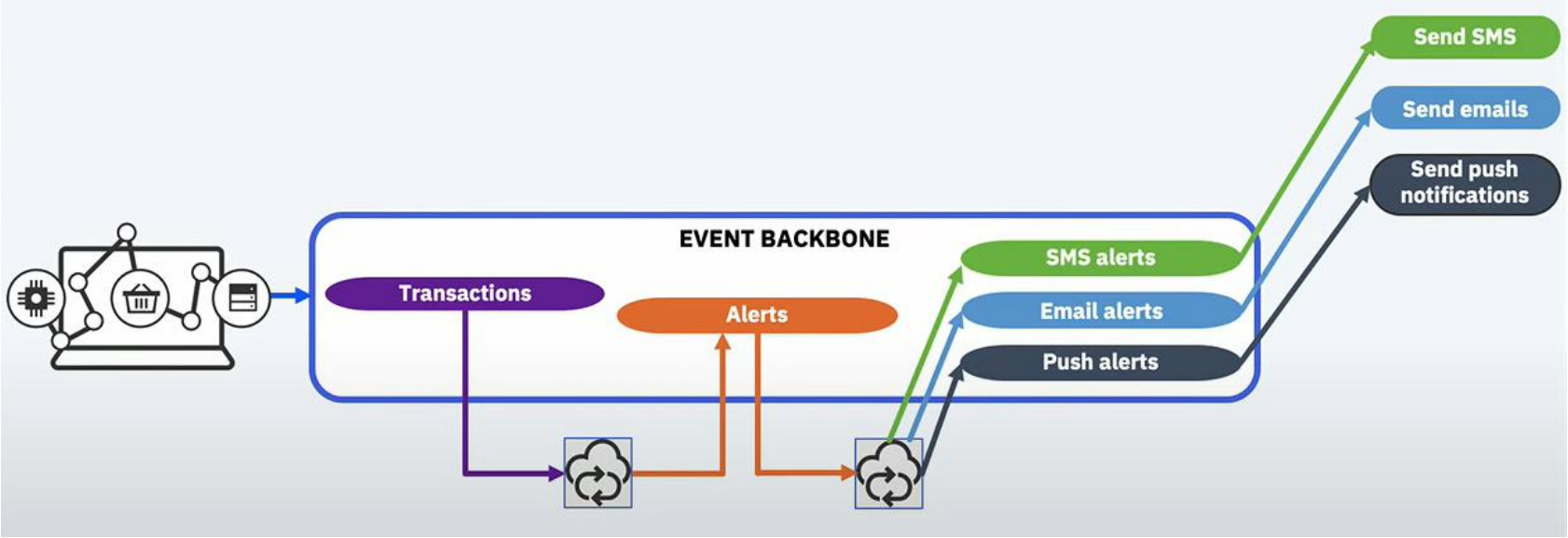
Benefits of EDA

- Supports the business demands for better service (no batch, less waiting)
- No point-to-point integrations (fire & forget)
- Fault tolerance, scalability, versatility, and other benefits of loose coupling.
- Powerful real-time response and analytics.
- Greater operational efficiencies

Event Backbone



Streaming Process

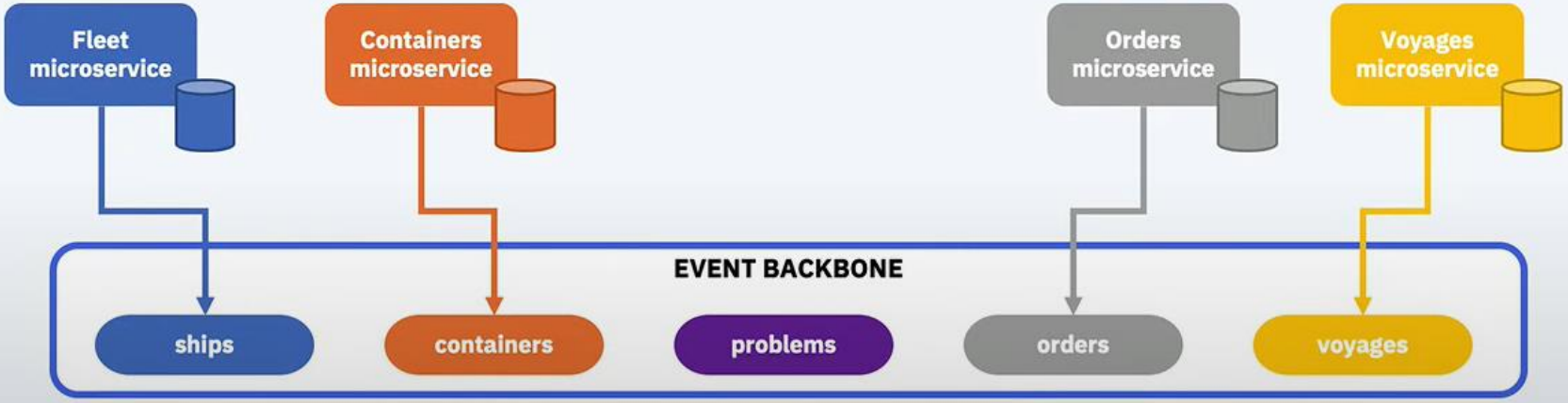


Pattern – Database per service

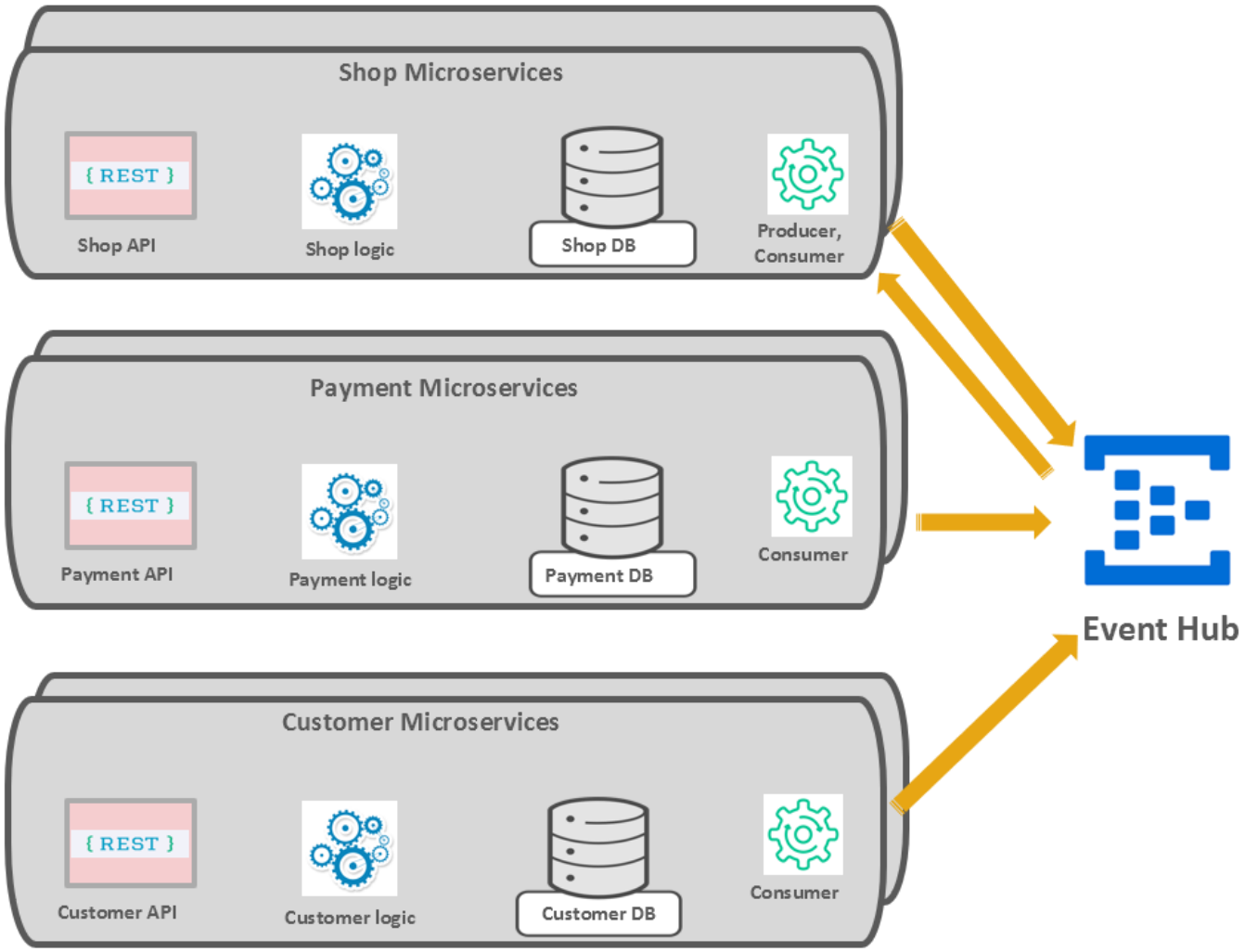
Each microservice persists its own data

Protects independence of the microservice against external change

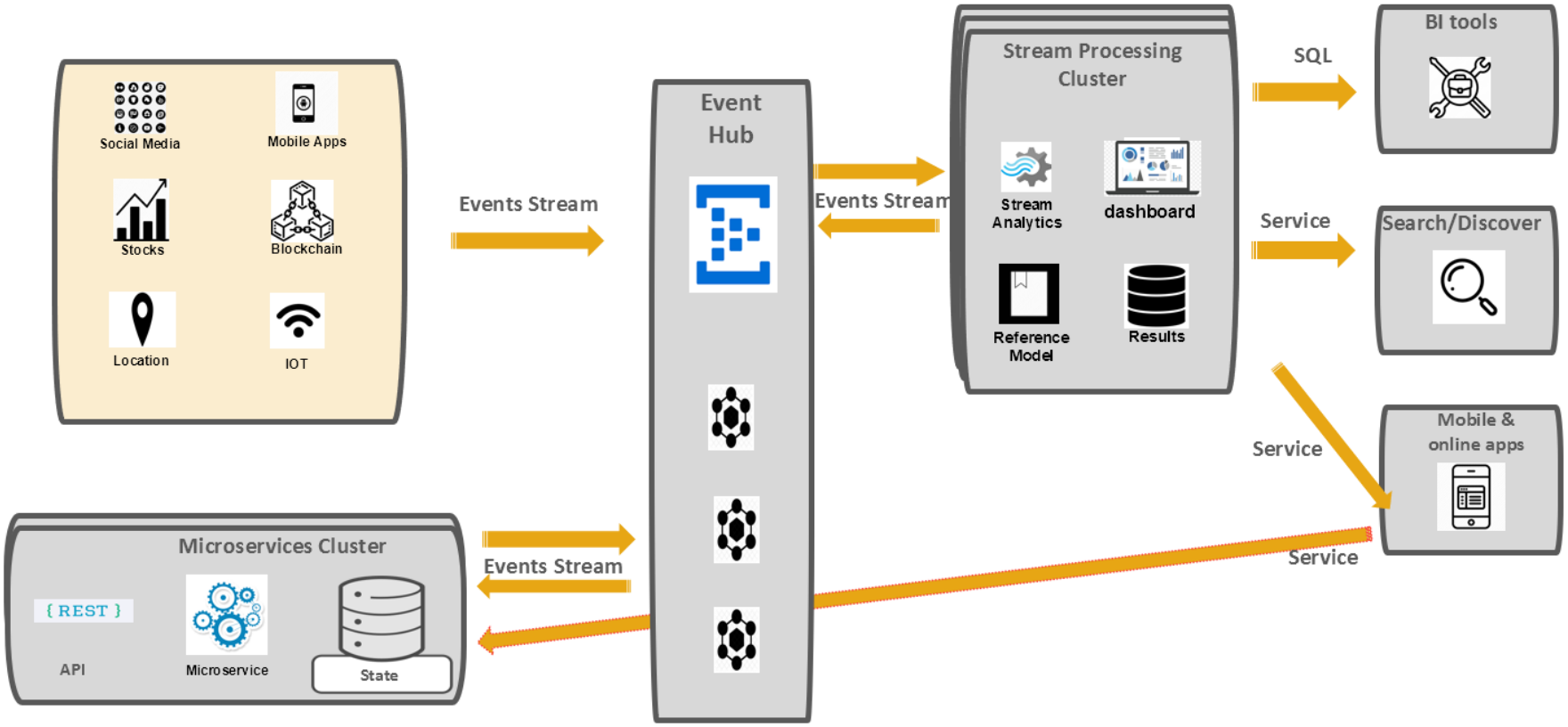
Introduces complexity for data consistency across microservices



Event Driven (Async) in Microservices



Microservice events and Streaming processing



Domain event and event sourcing

- Domain event- In domain-driven design, domain events are described as something that happens in the domain and is important to domain experts.

  - *A user has registered*
  - *An order has been cancelled.*
  - *The payment has been received*

Domain events are relevant both within a bounded context and across bounded contexts for implementing processes within the domain.

Best for communication between bounded context.

Domain event and event sourcing

- Event Sourcing - Event Sourcing ensures that all changes to application state are stored as a sequence of events. It store the events that lead to specific state and state too.
  - *MobileNumberProvided (MobileNumber)*
  - *VerificationCodeGenerated (VerificationCode)*
  - *MobileNumberValidated (no additional state)*
  - *UserDetailsProvided (FullName, Address, …)*

These events are sufficient to reconstruct the current state of the UserRegistration aggregate at any time.

Event Sourcing is for persistent strategy. Event Sourcing makes it easier to fix inconsistencies. Event Sourcing is local for a domain.

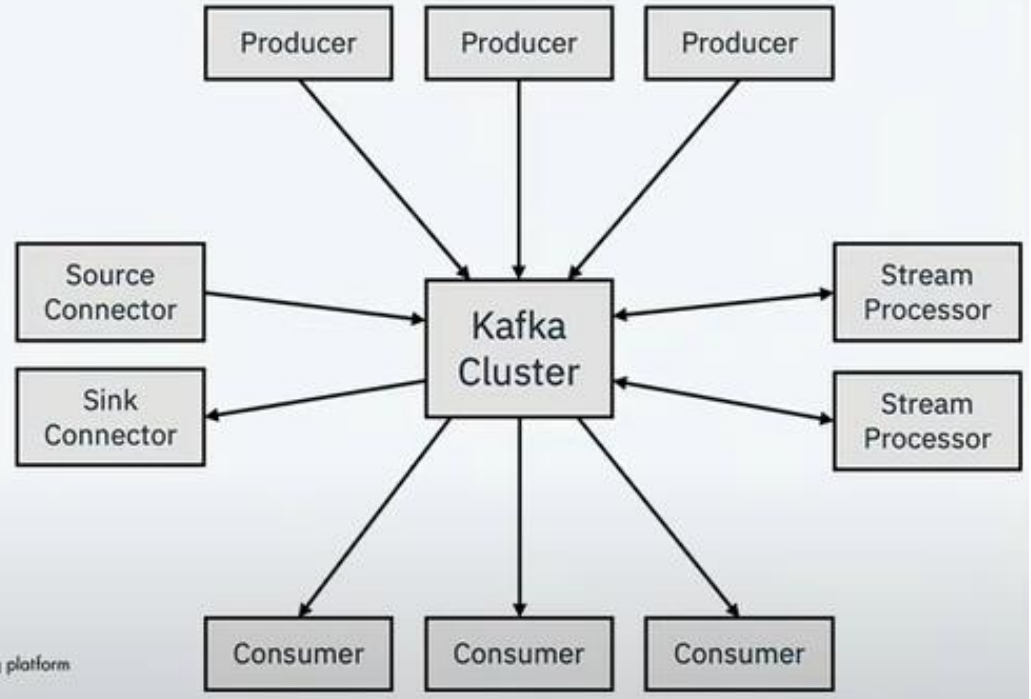
Kafka Streaming Platform

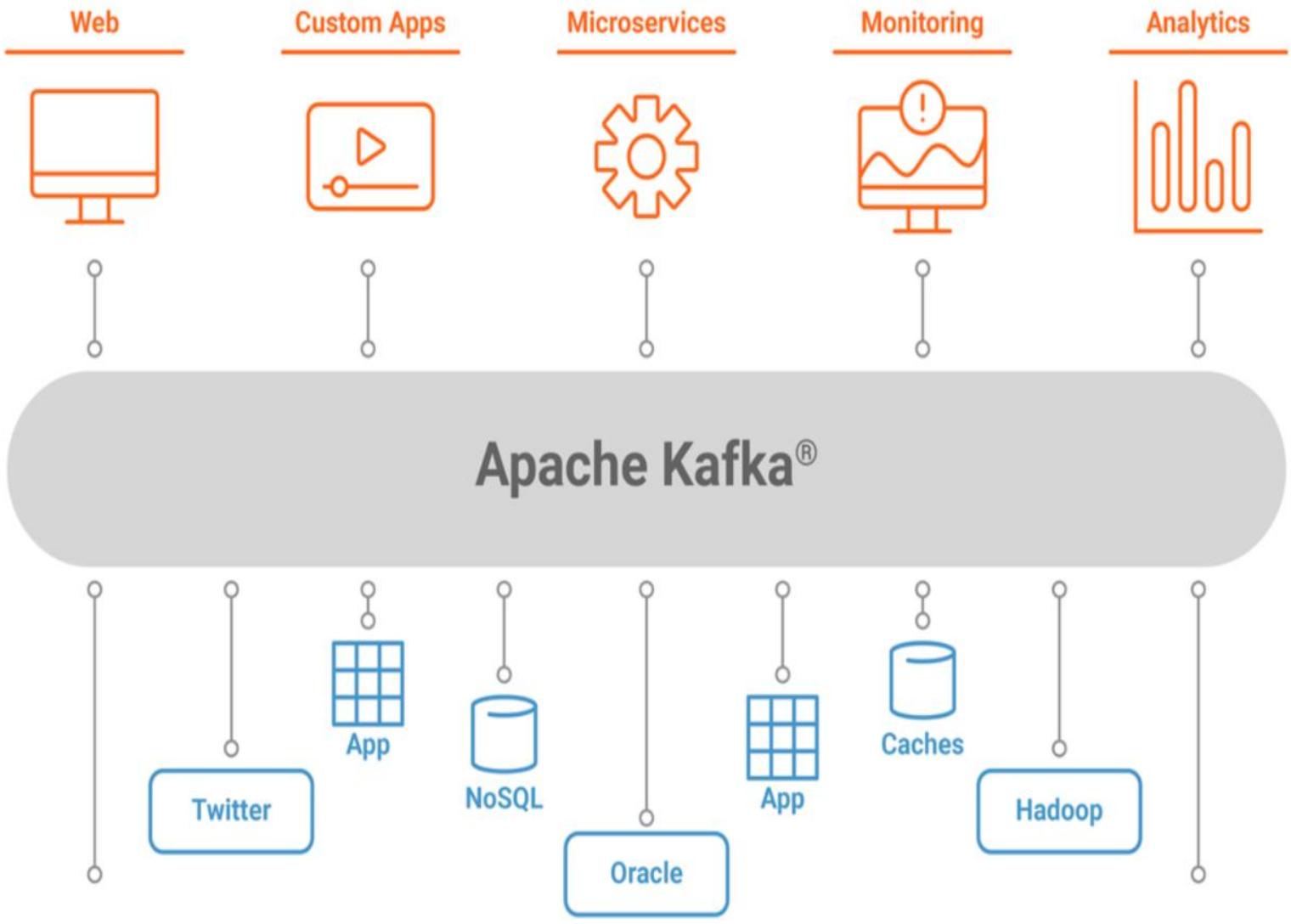
PUBLISH/SUBSCRIBE
Read and write streams of events, like a traditional messaging system

PROCESS
Support scalable stream processing applications that react to events in real time

STORE
Store streams of data safely in a distributed, replicated, fault-tolerant cluster

https://kafka.apache.org/




Web
Custom Apps
Microservices
Monitoring
Analytics
Apache Kafka®
Twitter
App
NoSQL
Oracle
App
Caches
Hadoop

Kafka Overview

- Distributed publish-subscribe messaging system.
- Designed for processing of real time activity stream data (log, metrics, collections, social media streams,…..)
- Does not use JMS API and standards
- Kafka maintains feeds of message in topics
- Initially developed at Linkedin, now part of Apache.

Benefits of Kafka

- **Reliability**. Kafka is distributed, partitioned, replicated, and fault tolerant. Kafka replicates data and is able to support multiple subscribers. Additionally, it automatically balances consumers in the event of failure.

- **Scalability**. Kafka is a distributed system that scales quickly and easily without incurring any downtime.

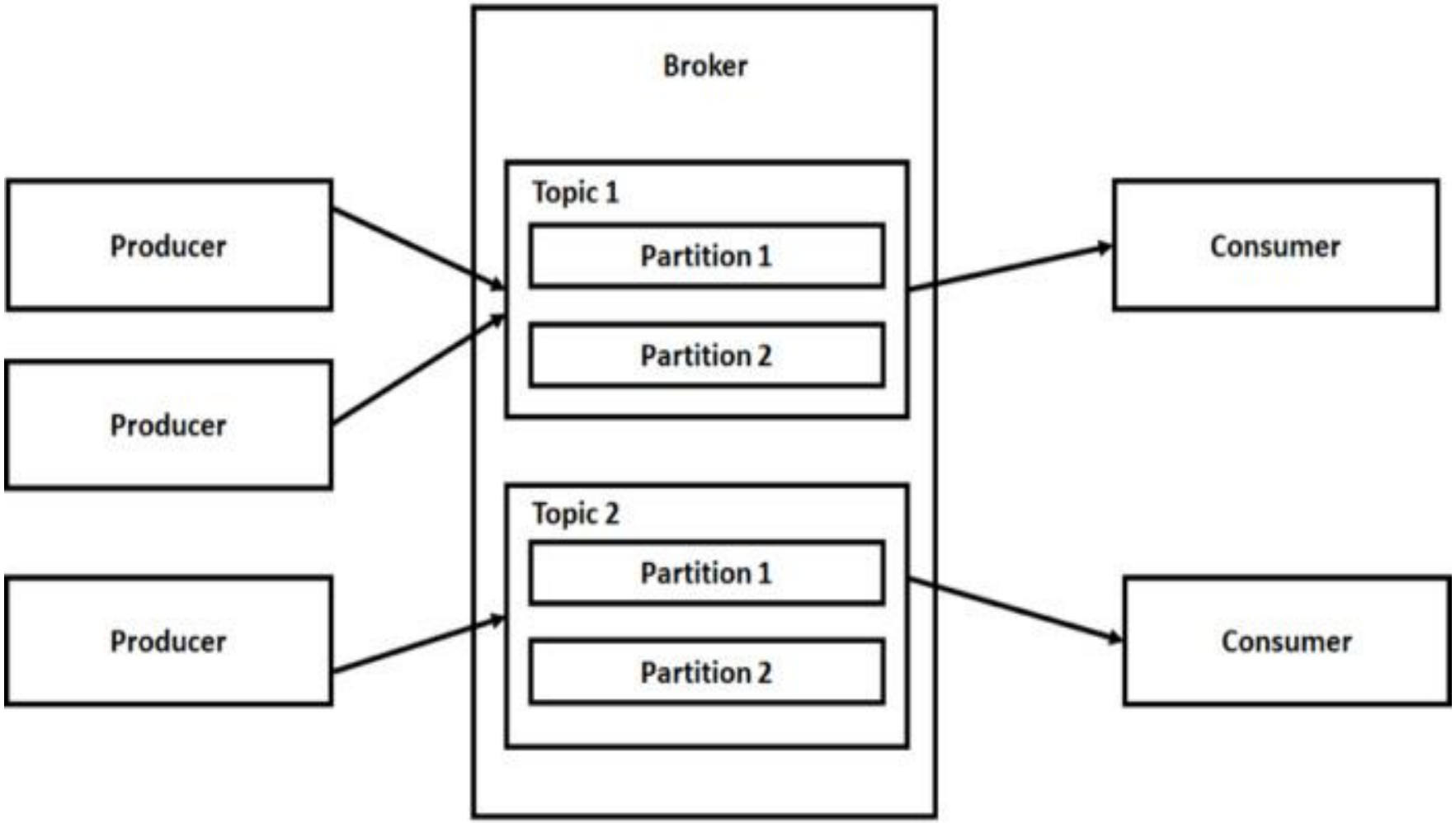
- **Durability**. Kafka uses a distributed commit log, which means messages persists on disk as fast as possible providing intra-cluster replication, hence it is durable.

- **Performance**. Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even when dealing with many terabytes of stored messages.

What is kafka

- Kafka is a messaging system that is designed to be fast, scalable, and durable.
- A **producer** is an entity/application that publishes data to a Kafka cluster, which is made up of **brokers.**
- A **Broker** is responsible for receiving and storing the data when a producer publishes.
- A **consumer** then consumes data from a broker at a specified offset, i.e. position.
- A **Topic** is a category/feed name to which records are stored and published. Topics have partitions and order guaranteed per partitions
- All Kafka records are organized into topics. Producer applications write data to topics and consumer applications read from topics.

Kafka Architecture



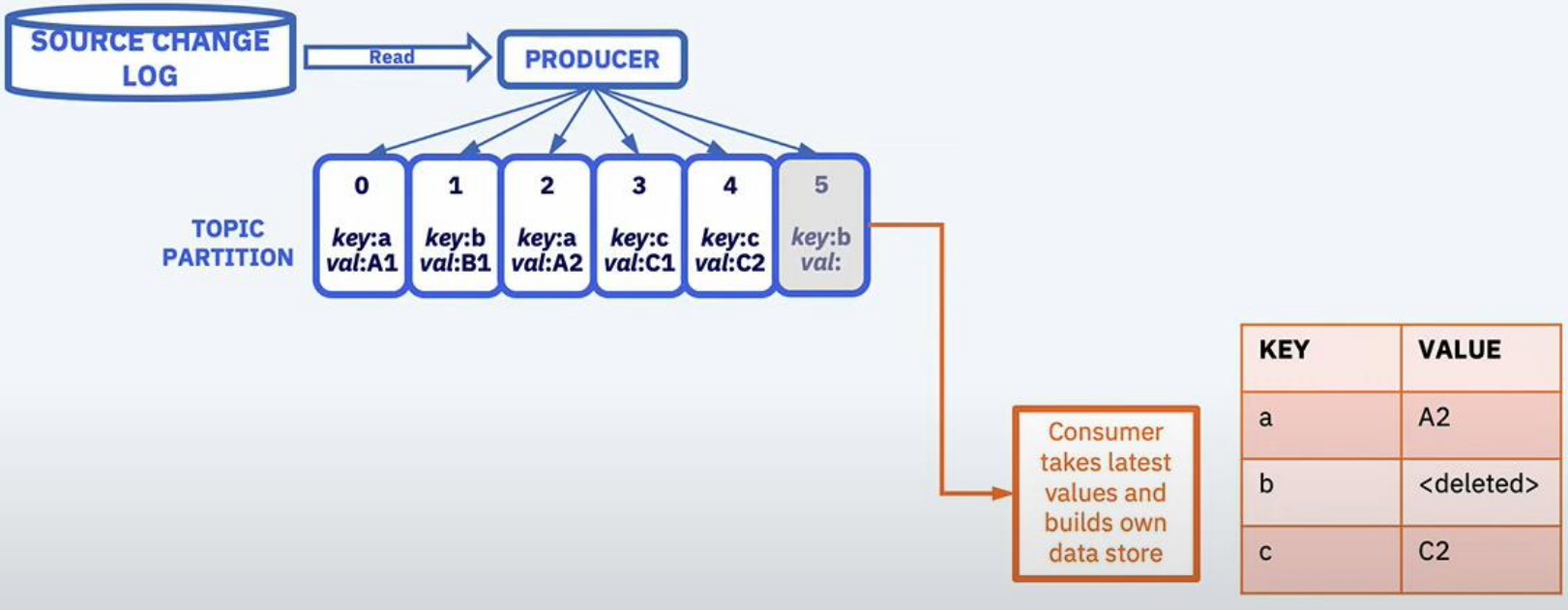
Key Concepts of Kafka

- Topic is divided in partitions.
- The message order is only guarantee inside a partition
- Consumer offsets are persisted by Kafka with a commit/auto-commit mechanism.
- Consumers subscribes to topics
- Consumers with different group-id receives all messages of the topics they subscribe. They consume the messages at their own speed.
- Consumers sharing the same group-id will be assigned to one (or several) partition of the topics they subscribe. They only receive messages from their partitions. So a constraint appears here: the number of partitions in a topic gives the maximum number of parallel consumers.
- The assignment of partitions to consumer can be automatic and performed by Kafka. If a consumer stops polling or is too slow, a process call “re-balancing” is performed and the partitions are re-assigned to other consumers.

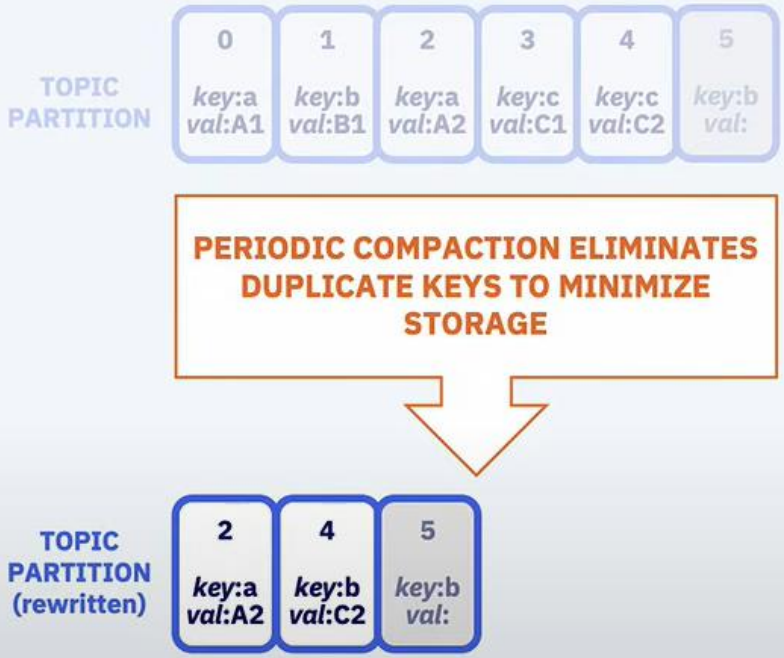
Key Concepts of Kafka

- Kafka normally divides topic in multiply partitions.
- Each partition is an ordered, immutable sequence of messages that is continually appended to.
- A message in a partition is identified by a sequence number called offset.
- The FIFO is only guarantee inside a partition.
- When a topic is created, the number of partitions should be given
- The producer can choose which partition will get the message or let Kafka decides for him based on a hash of the message key (recommended). So the message key is important and will be the used to ensure the message order.
- Moreover, as the consumer will be assigned to one or several partition, the key will also “group” messages to a same consumer.

Kafka log compaction for data replication



Kafka log compaction for data replication



KEY	VALUE
a	A2
b	<deleted>
c	C2

Saga pattern

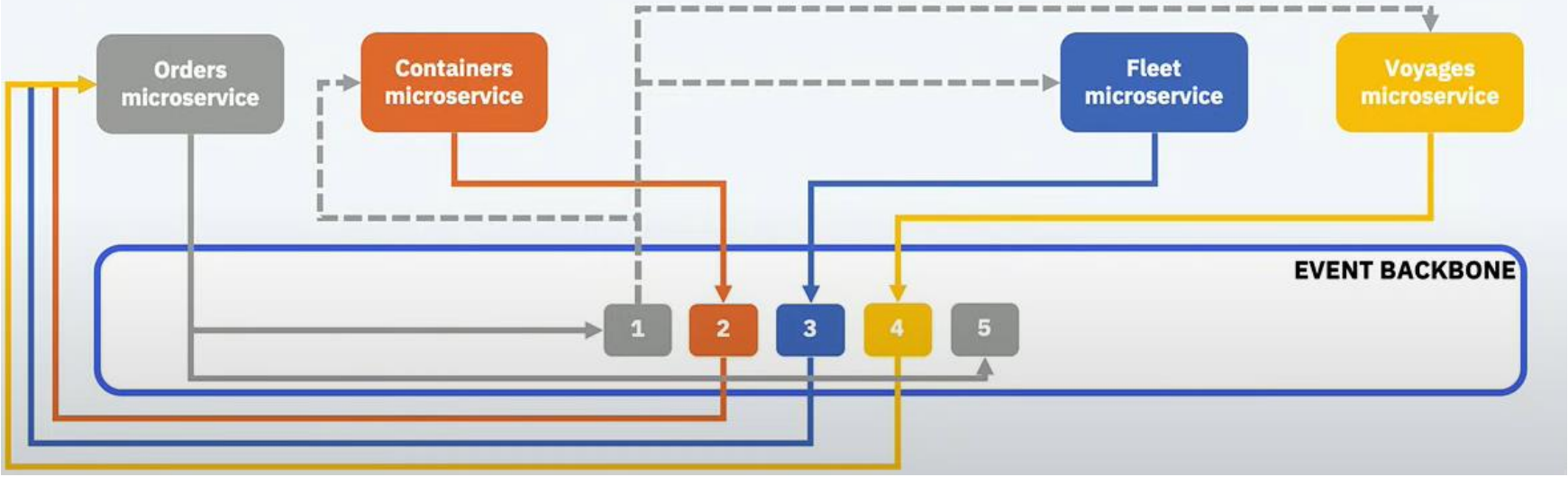
- The Saga pattern provides transaction management using a sequence of local transactions.
- A local transaction is the atomic work effort performed by a saga participant.
- Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.
- If a local transaction fails, the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

Saga pattern

Orchestration of multi-step operations across microservices

Data consistency across microservices without distributed transactions

Programming can be complex, particularly for failure compensation



Saga pattern

Benefits

•Good for simple workflows that require few participants and don't need a coordination logic.

•Doesn't require additional service implementation and maintenance.

•Doesn't introduce a single point of failure, since the responsibilities are distributed across the saga participants.

Drawbacks

•Workflow can become confusing when adding new steps, as it's difficult to track which saga participants listen to which commands.

•There's a risk of cyclic dependency between saga participants because they have to consume each other's commands.

•Integration testing is difficult because all services must be running to simulate a transaction.

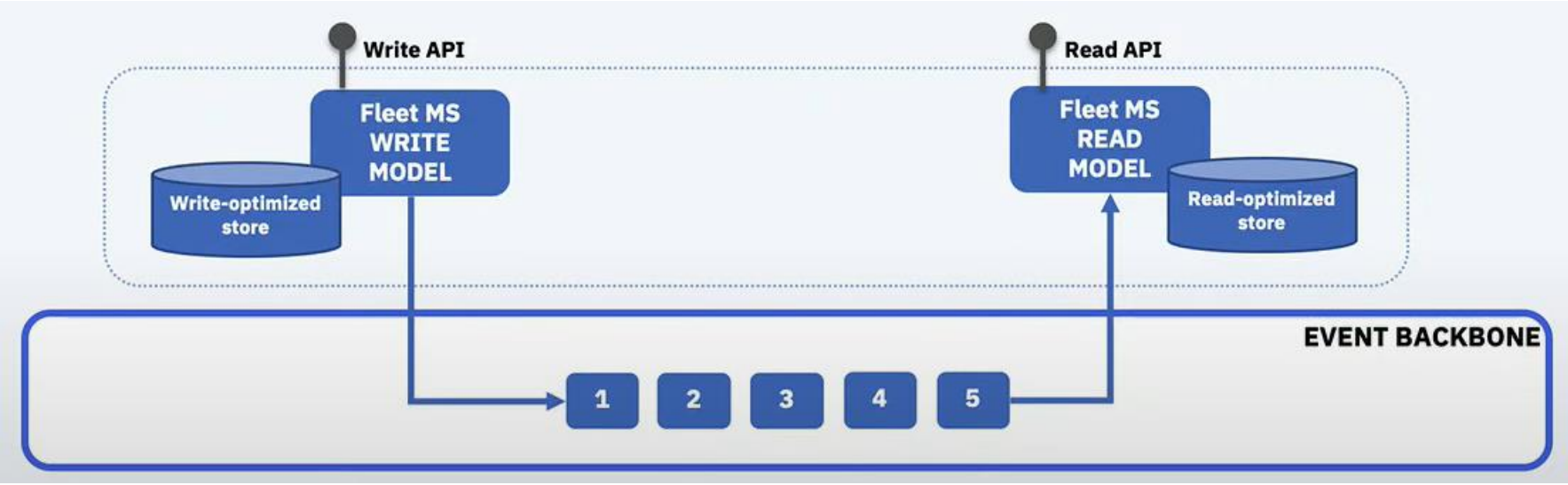
CQRS pattern

- CQRS stands for Command and Query Responsibility Segregation, a pattern that separates read and update operations for a data store.

- Implementing CQRS in your application can maximize its performance, scalability, and security.

- The flexibility created by migrating to CQRS allows a system to better evolve over time and prevents update commands from causing merge conflicts at the domain level.

CQRS pattern



CQRS pattern

CQRS separates reads and writes into different models, using commands to update data, and queries to read data.

- Commands should be task-based, rather than data centric. ("Book hotel room", not "set ReservationStatus to Reserved").
- Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
- Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.