

Microservices patterns

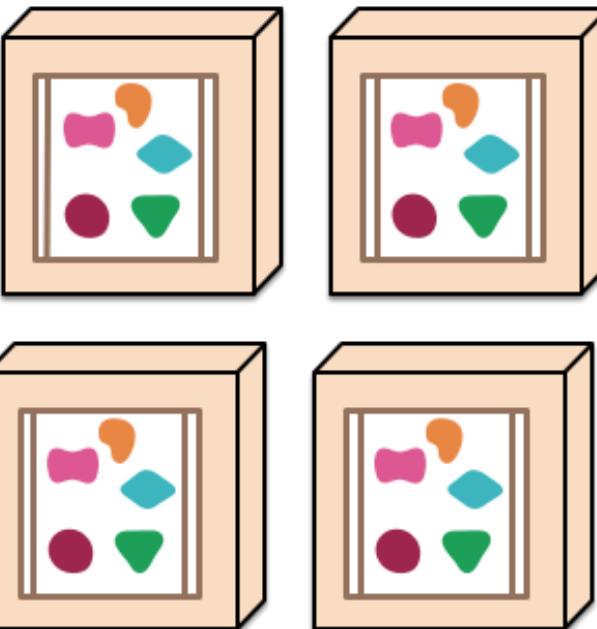
Giovanni Jiayi Hu

Scaling ¹

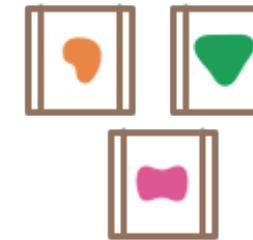
A monolithic application puts all its functionality into a single process...



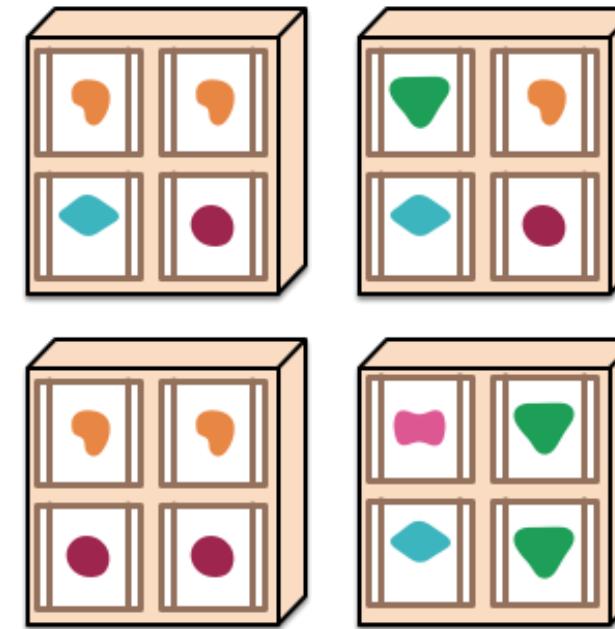
... and scales by replicating the monolith on multiple servers



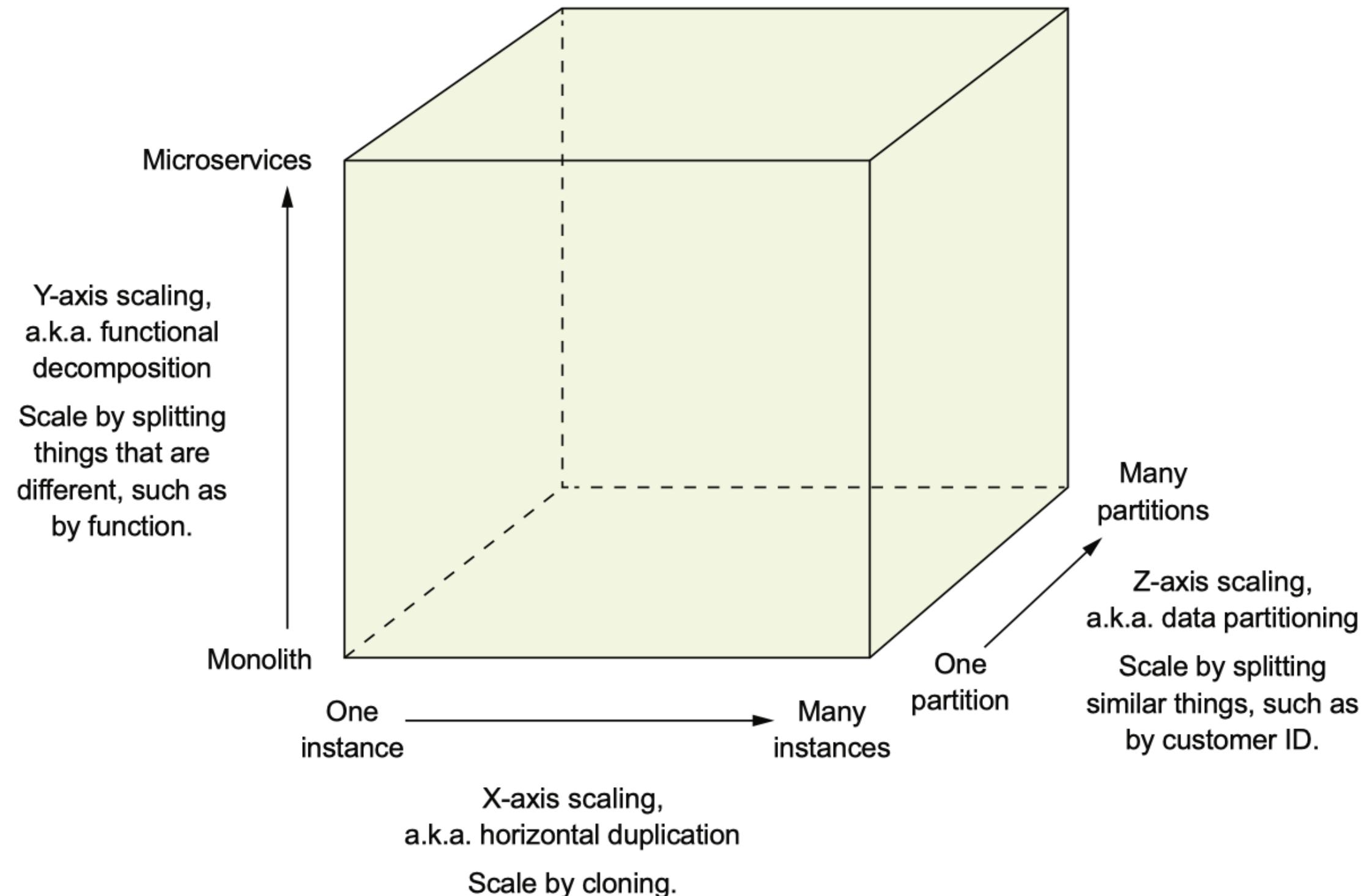
A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



¹ Martin Fowler - Microservices

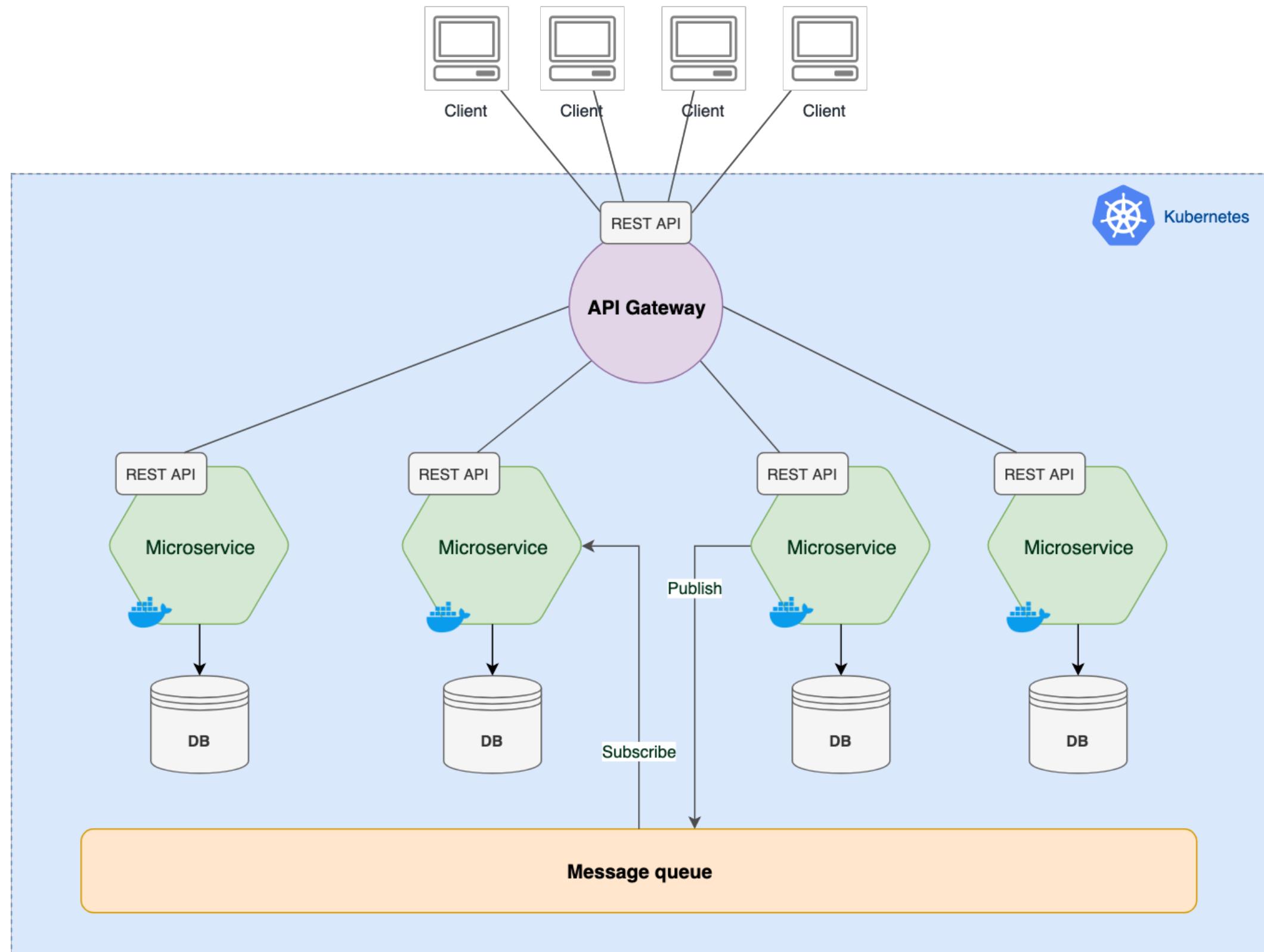


Microservices

A microservice architecture is a **service-oriented architecture** composed of loosely coupled elements that have **bounded contexts**.

Microservice

A microservice is understood as a small **self-contained** application that has a **single responsibility**, a lightweight stack, and can be deployed, scaled and tested **independently**.



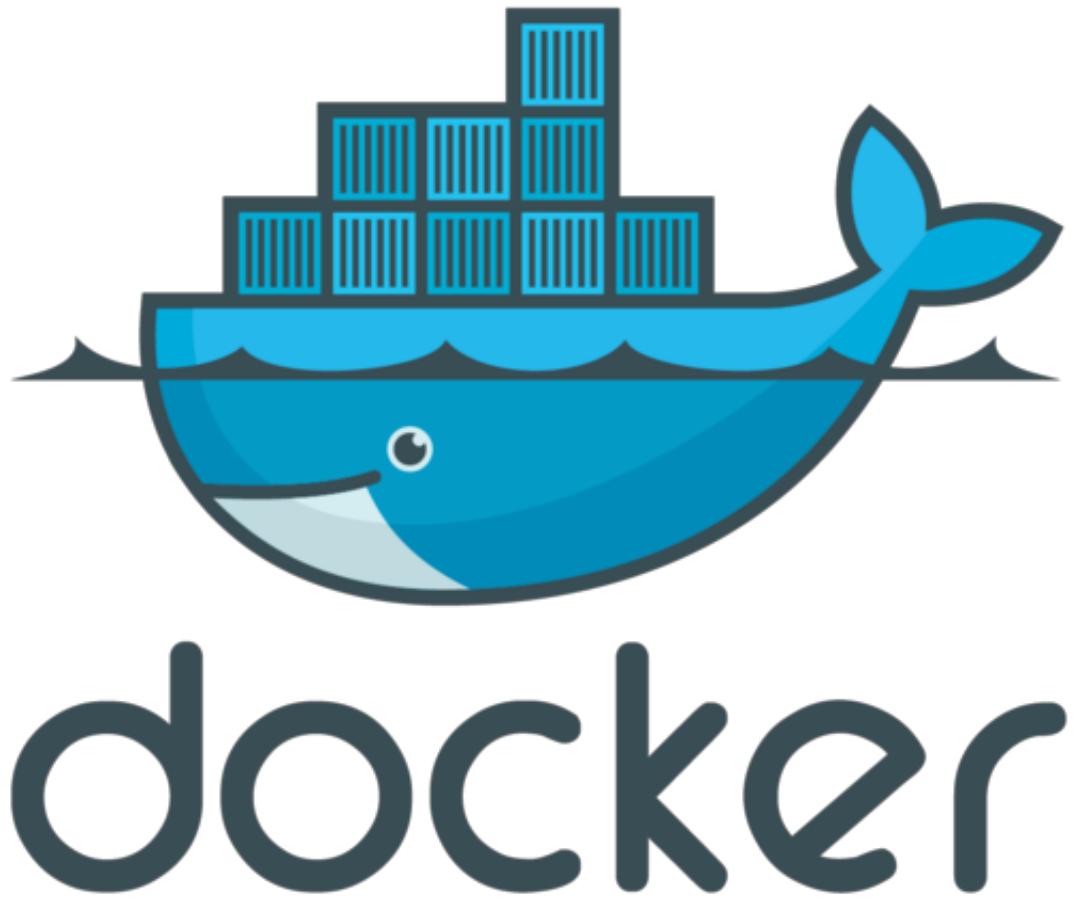
Self-containing: Docker

Forces

- Encapsulation of technology stack
- Service isolation
- Resource constraining
- Location independence

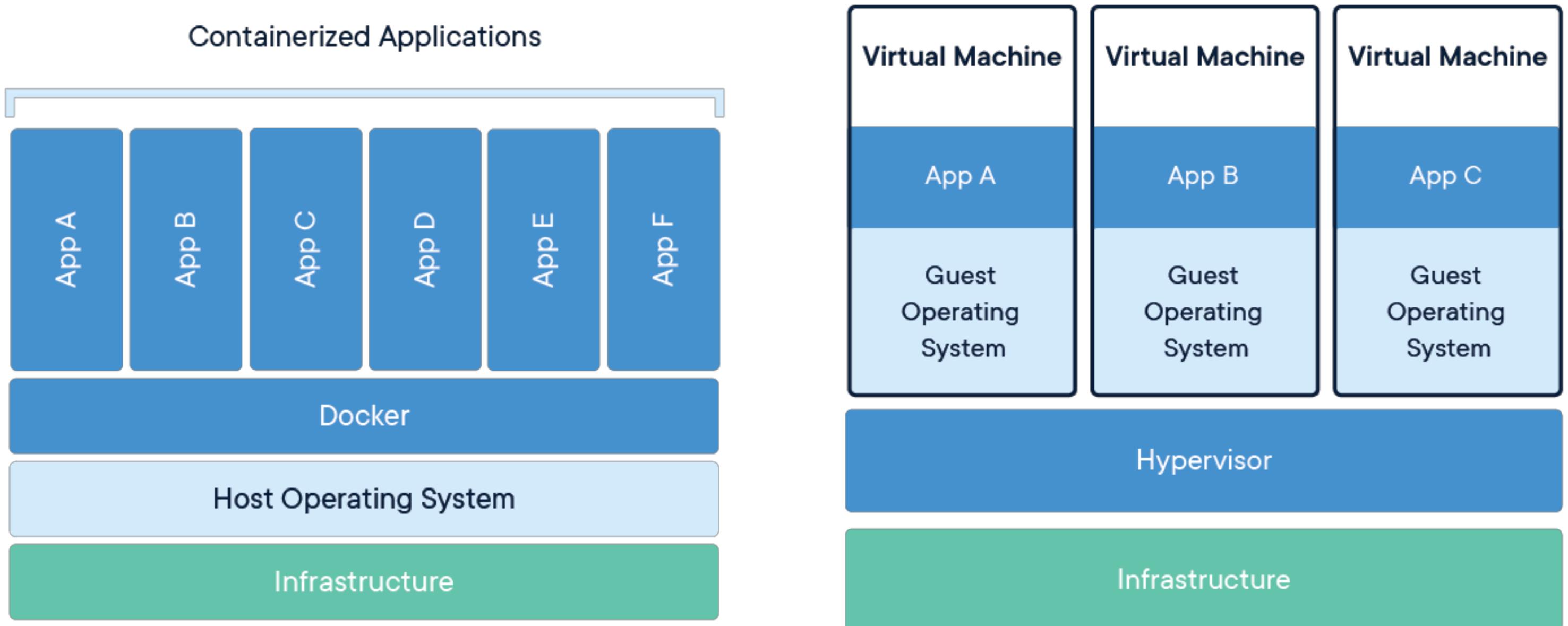
Solution: Docker

- **Image:** an executable package that includes everything needed to run an application
- **Isolated containers:** runtime instance of image



Containerization

The mechanism by which a **host operating system** runs programs in **isolated user-space environments**.



Container vs Virtual Machines

- Shared host kernel and OS resources
- Less isolation but more lightweight
- Snapshot images
- Weights MB

Orchestrating - Kubernetes

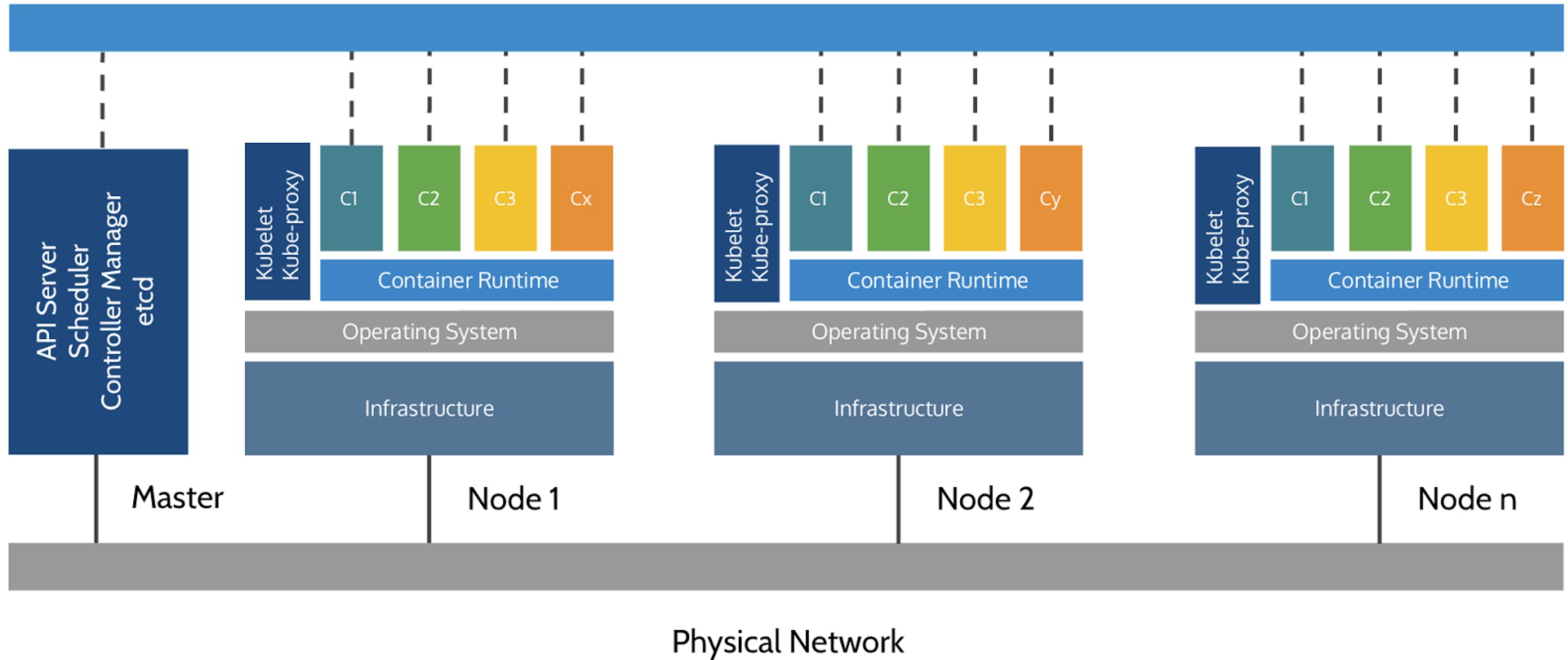
Kubernetes (K8s)

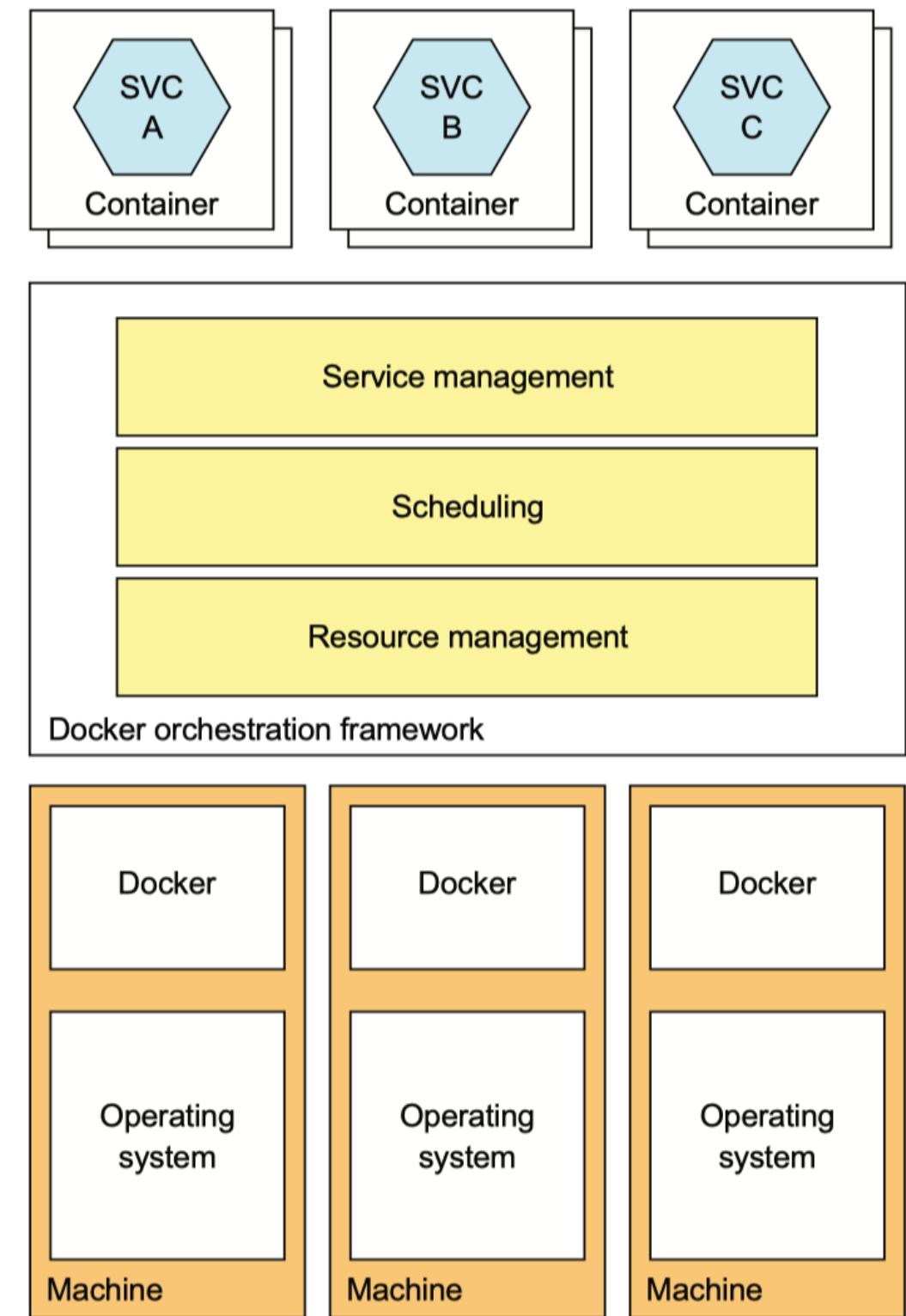
Declarative configuration and automation for managing containerized services



kubernetes

Overlay Network (Flannel/OpenVSwitch/Weave)





Auto-scaling

- $\text{desiredReplicas} = \text{currentReplicas} * (\text{currentMetricValue} / \text{desiredMetricValue})$
- Avoids over/under-provisioning
- Avoids *thrashing*

Health checks

- **Liveness:** dead or alive (no zombies)
- **Readiness:** ready to serve traffic
- Probes: HTTP request / TCP connection / exec

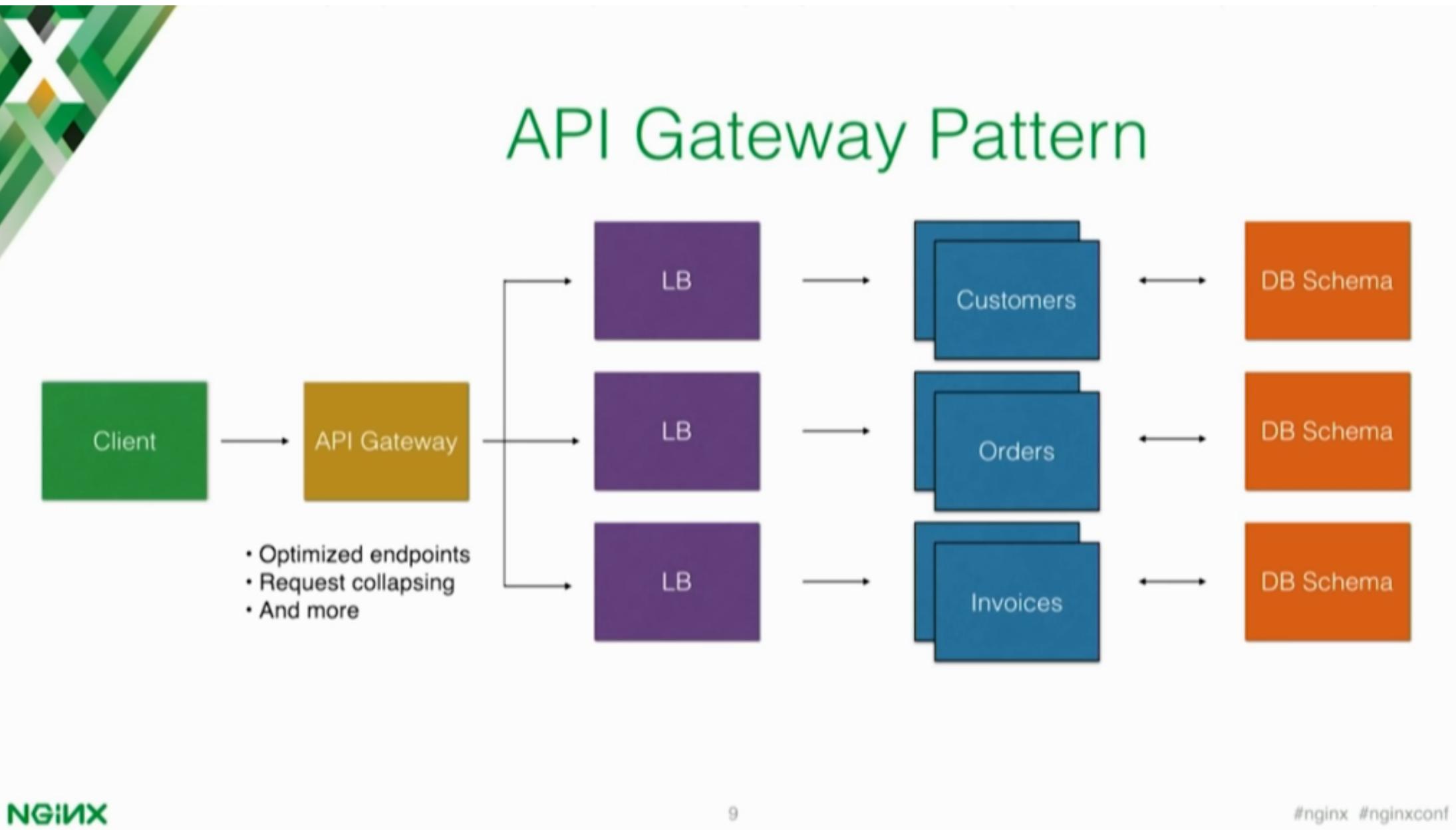
Rolling Updates

- Zero down-time
- Guarantee minimum availability
- Use health checks
- Ability to roll back to a past version (**rollout**)

Forces

- Encapsulation of microservices
- Single entry point
- Stable HTTP endpoint to reach services
- Route Level 7 traffic

API Gateway



API Gateway - Ambassador

- Level 7 load balancing
- HTTP routing
- Authentication
- Rate limiting



Database architecture

Relational databases scalability

1. Inefficient joins across nodes
2. Distributed transactional processing: X/Open XA
3. Reduced availability

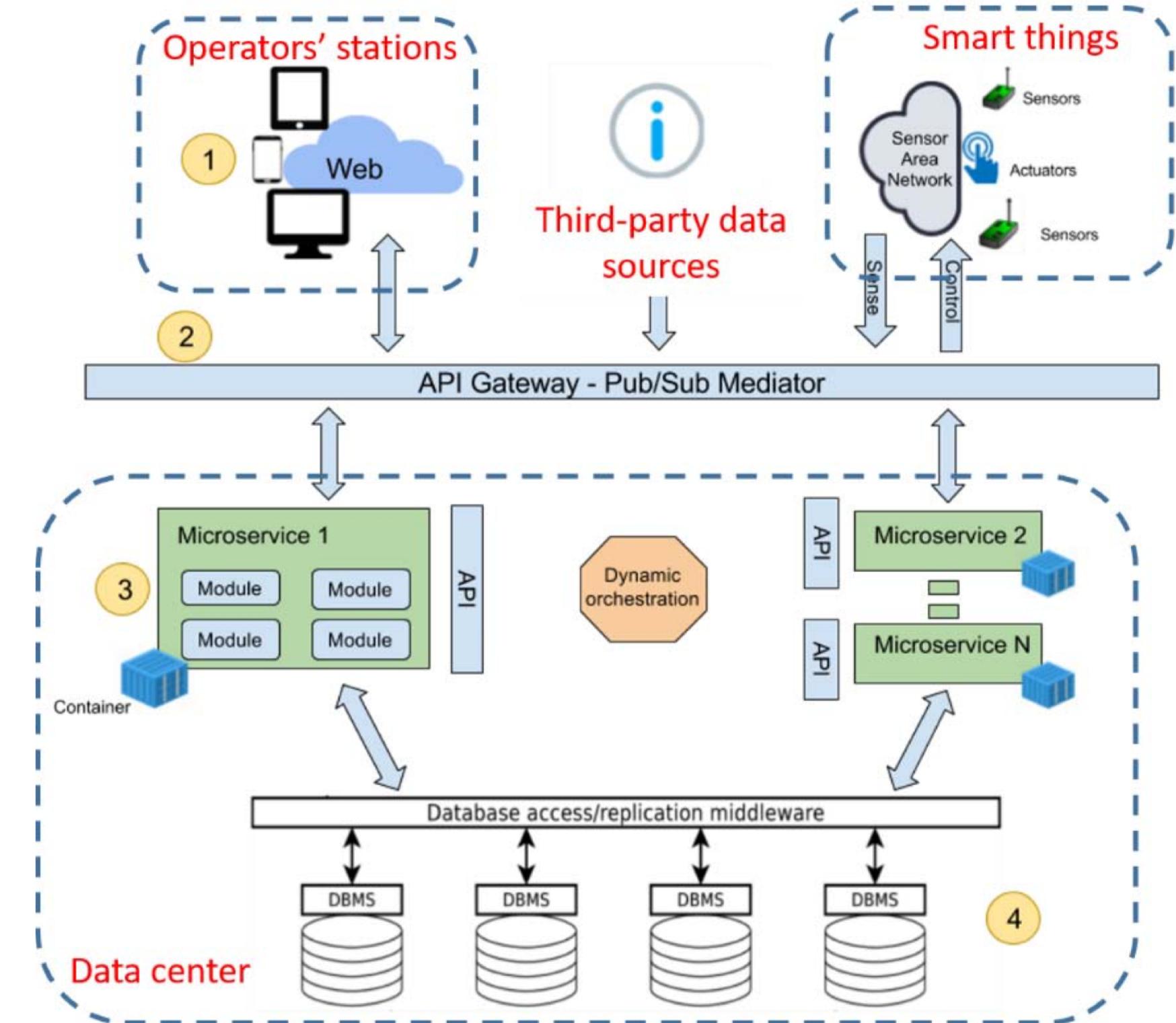
CAP Theorem

It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees: consistency, availability, partition tolerance.

- Eric Brewer

Forces

- Services must be **loosely coupled**
- **Enforce invariants** that span multiple services
- Need to **query/join** data that is owned by multiple services
- Must be **replicated and sharded**
- Data storage have **different requirements**



Database per service - Advantages

1. Loose coupling
2. Different data storages
3. Deployed, scaled and tested independently
4. Resilience

Database per service - Drawbacks 😭

1. Transactions across services
 - **Saga pattern** at application level
2. Joining data is challenging
 - **API composition** at application level
 - **CQRS pattern** at application level
3. Complexity

The hardest part about microservices: your data *

*The Hardest Part About Microservices: Your Data

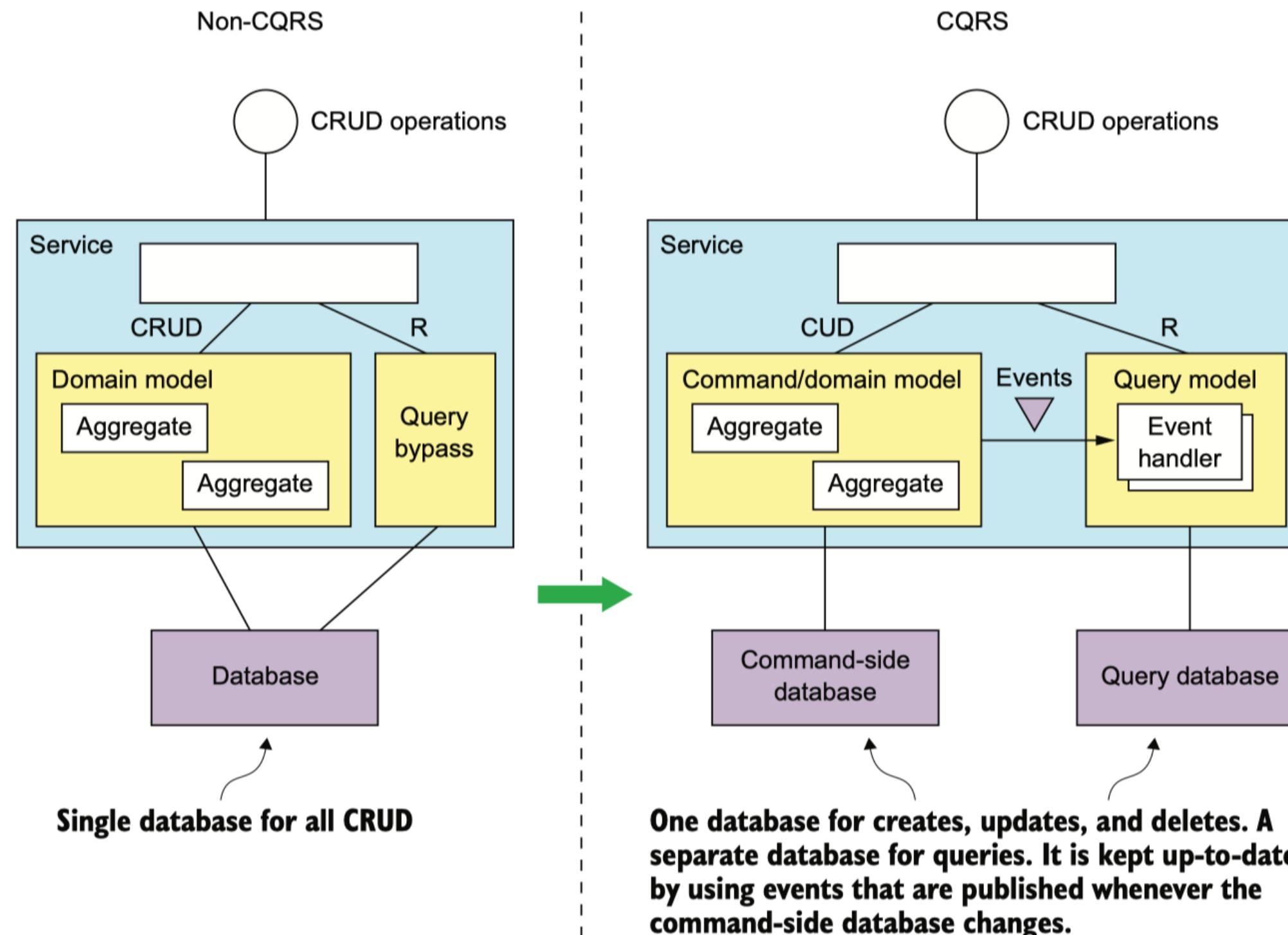
Application level patterns

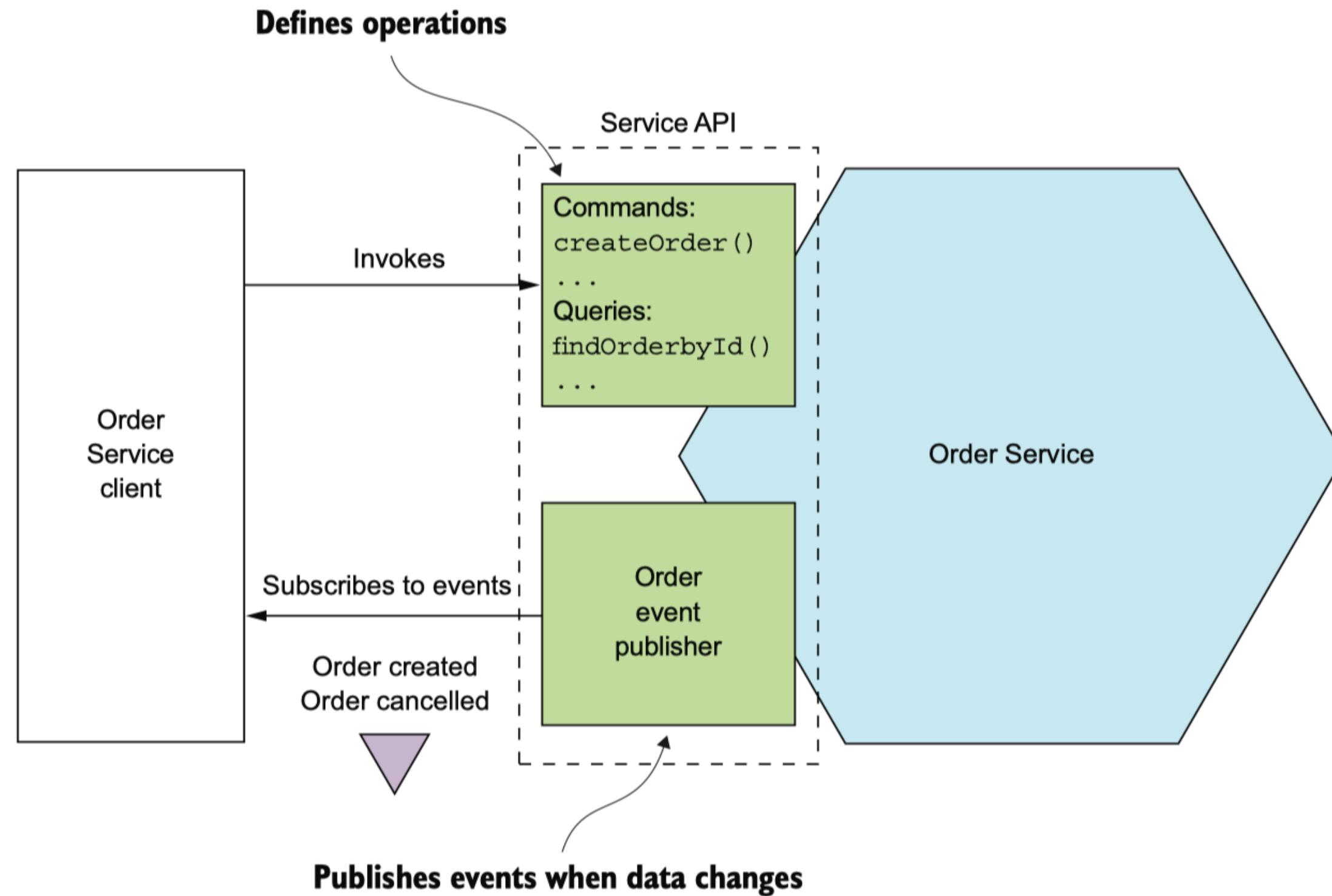
Forces

1. Retrieve data scattered across multiple services
2. Enforcing business invariants
3. Handle inefficient/unsupported query in service storage database

Solutions

- API Composition / Gateway
- Command Query Responsibility Segregation (CQRS)





Interprocess communication

Sync vs async

- Sync is blocking: a service waits for the response
- Sync reduces availability
- But sync REST API are required

Forces

- Async and time decoupled communication
- Availability
- Loose coupling
- Message buffering

RPC vs Message Queuing

- RPC is *usually* sync
- RPC increases coupling and reduces availability
- MQ can be persisted and guarantees delivery
- MQ is **loosely coupled in time**



The biggest issue in changing a monolith into microservices lies in changing the communication pattern.

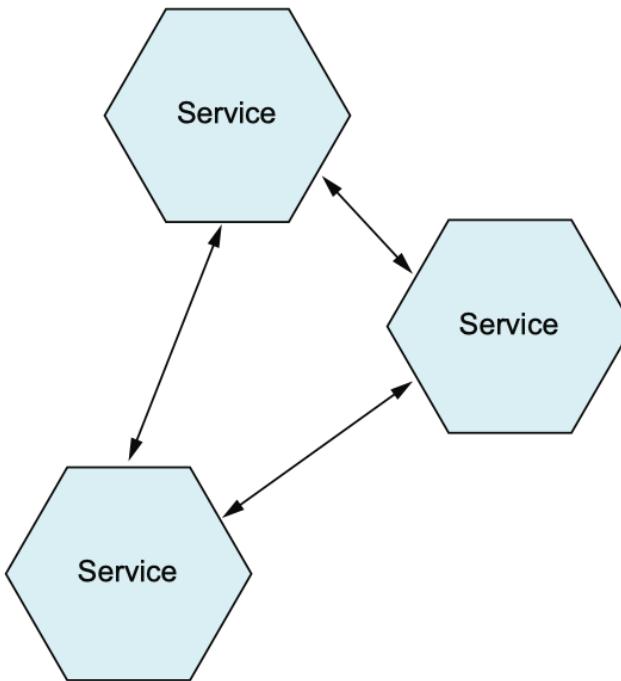
A naive conversion from in-memory method calls to RPC leads to chatty communications which don't perform well.

— Martin Fowler

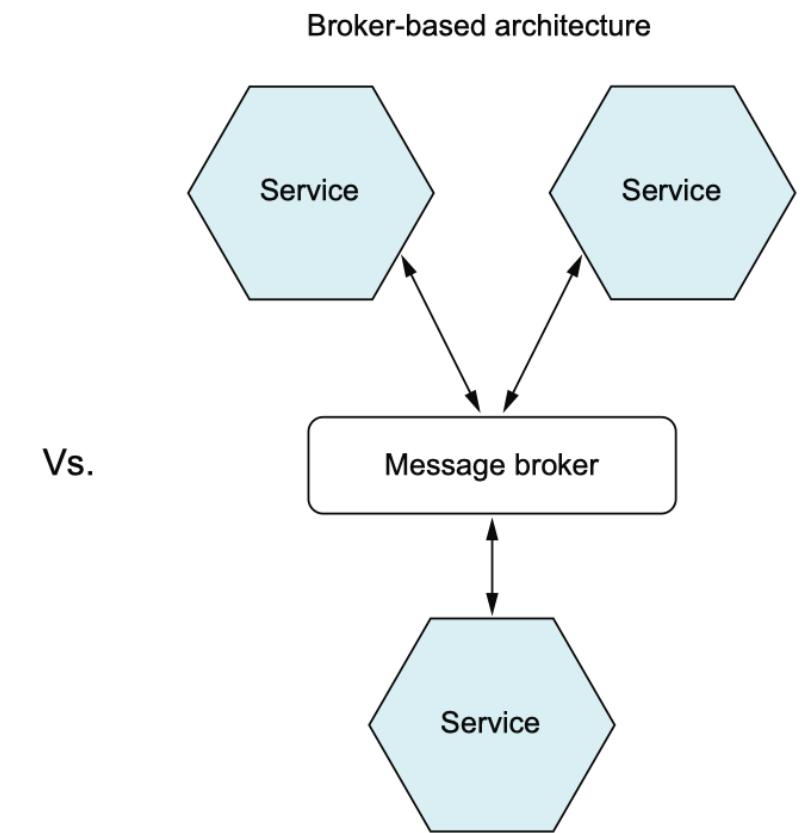
Broker-based MQ - Advantages

- Loose coupling
- No discovery mechanism
- Message buffering
- Explicit interprocess communication
- Can convert messages to be understood by the destination

Brokerless architecture



Broker-based architecture

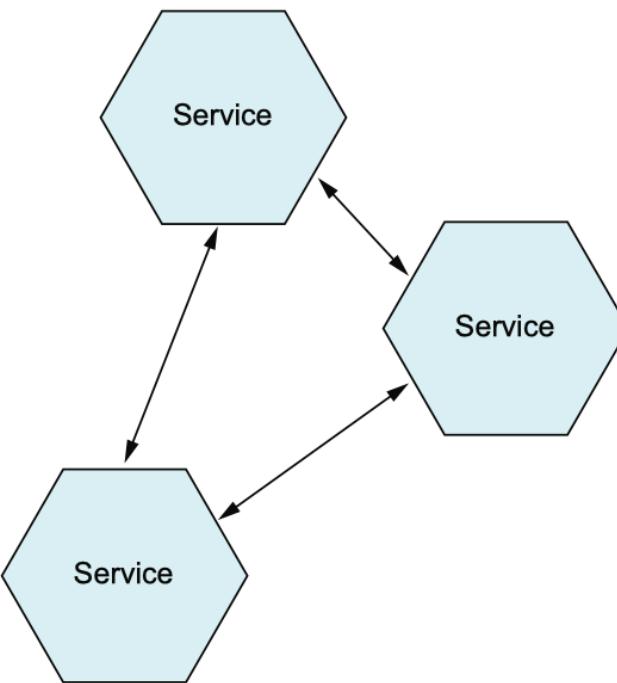


Vs.

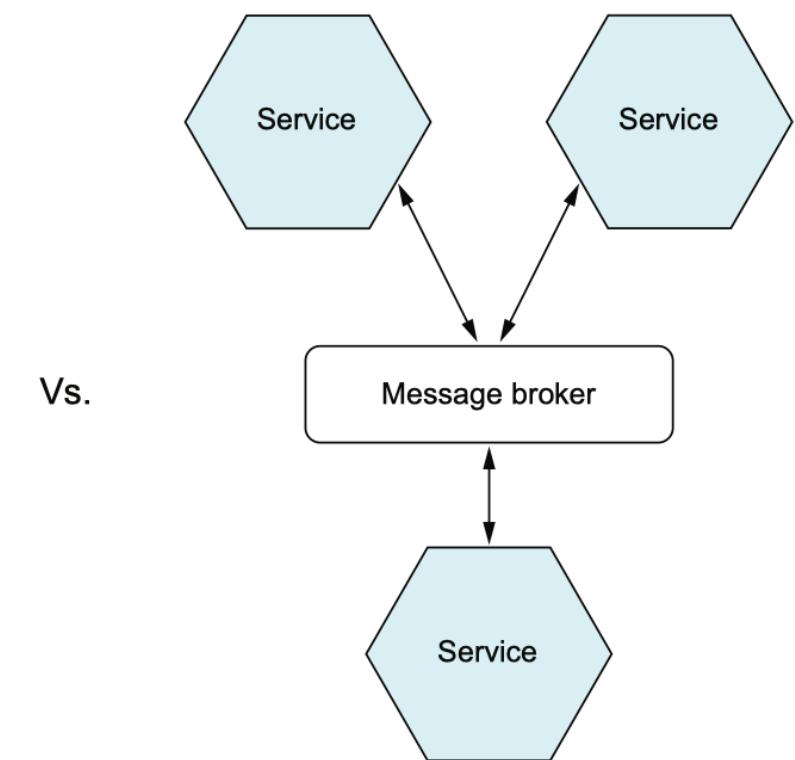
Broker-based MQ - Disadvantages

- Potential performance bottleneck
- Potential single point of failure

Brokerless architecture



Broker-based architecture



Vs.

Message broker characteristics

- Message ordering
- Delivery guarantees
- Persistence
- Durability
- Latency

Not a panacea

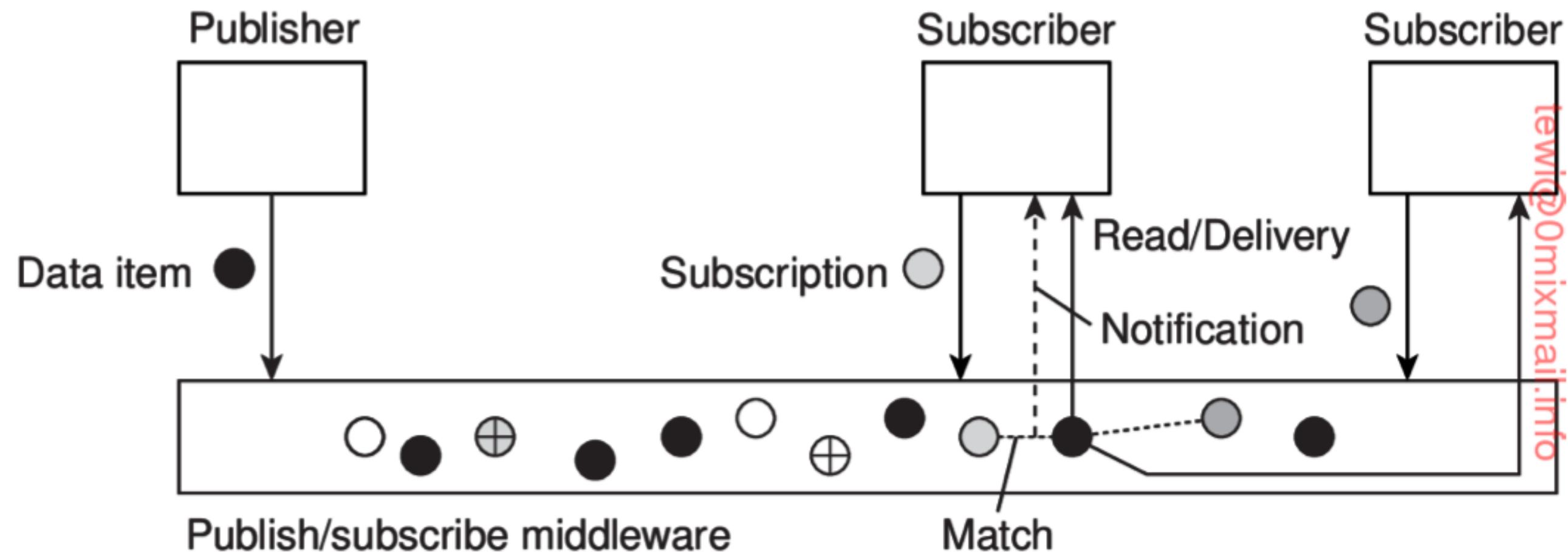
- Message duplication
 - **Idempotent message handlers**
 - Track messages
- Transactional messaging

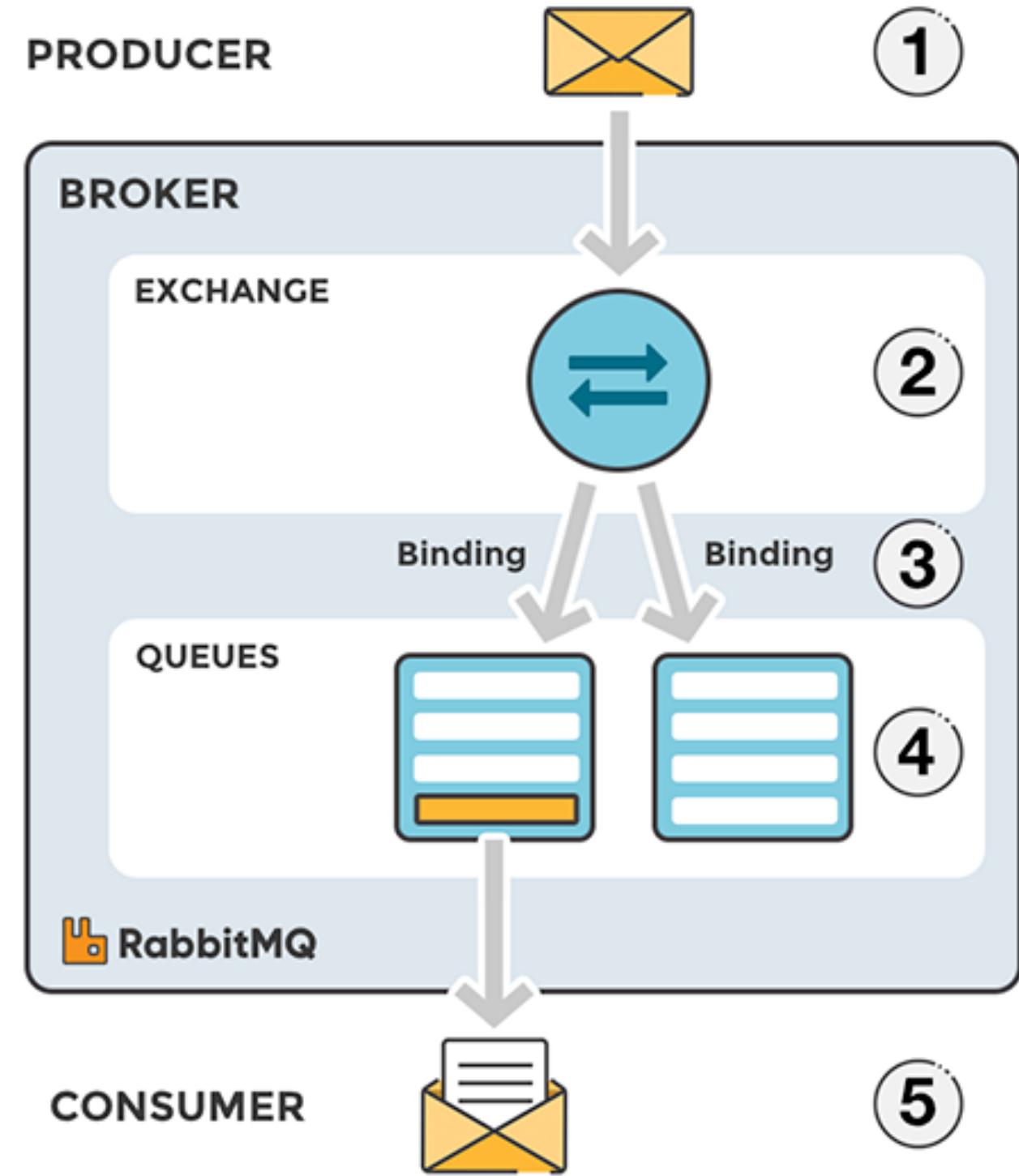
RabbitMQ

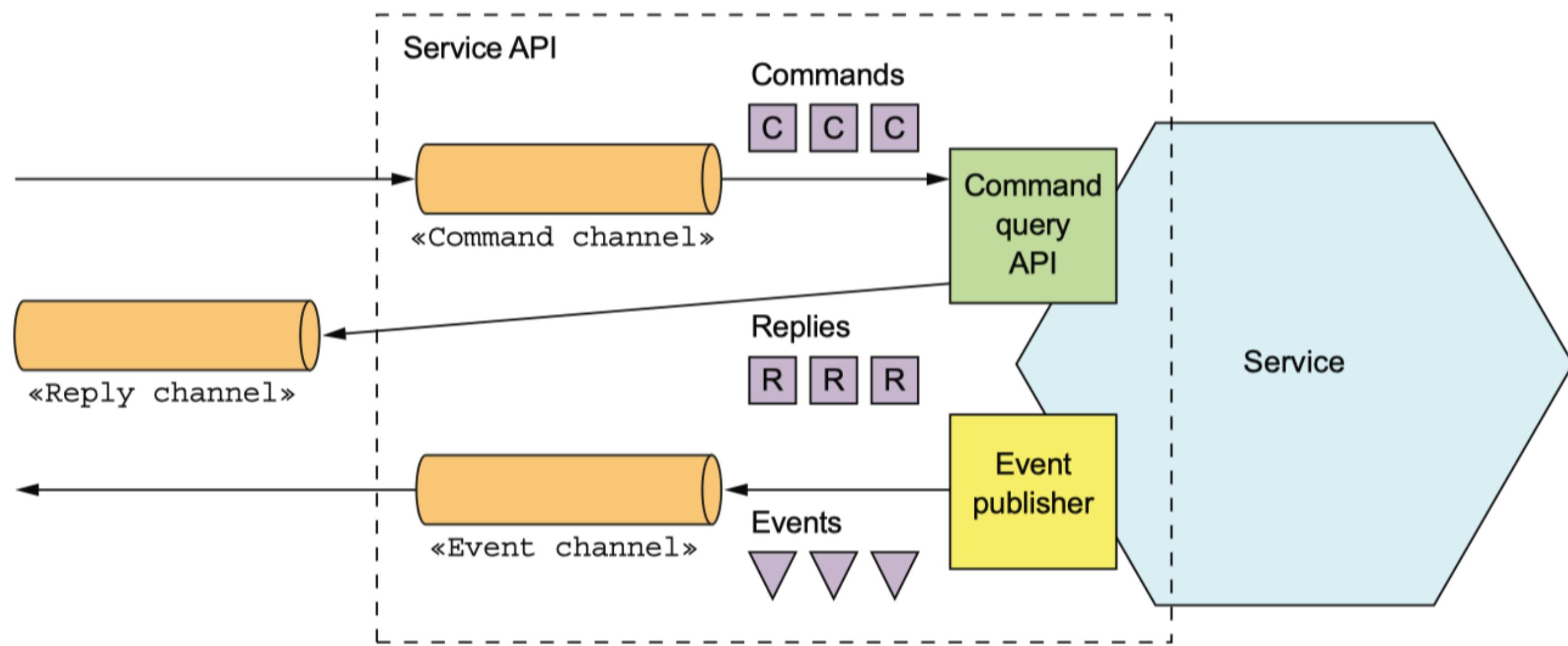
- AMQP implementation
- Connection, channel
- Producer, consumer, queue
- Durability



RabbitMQ™

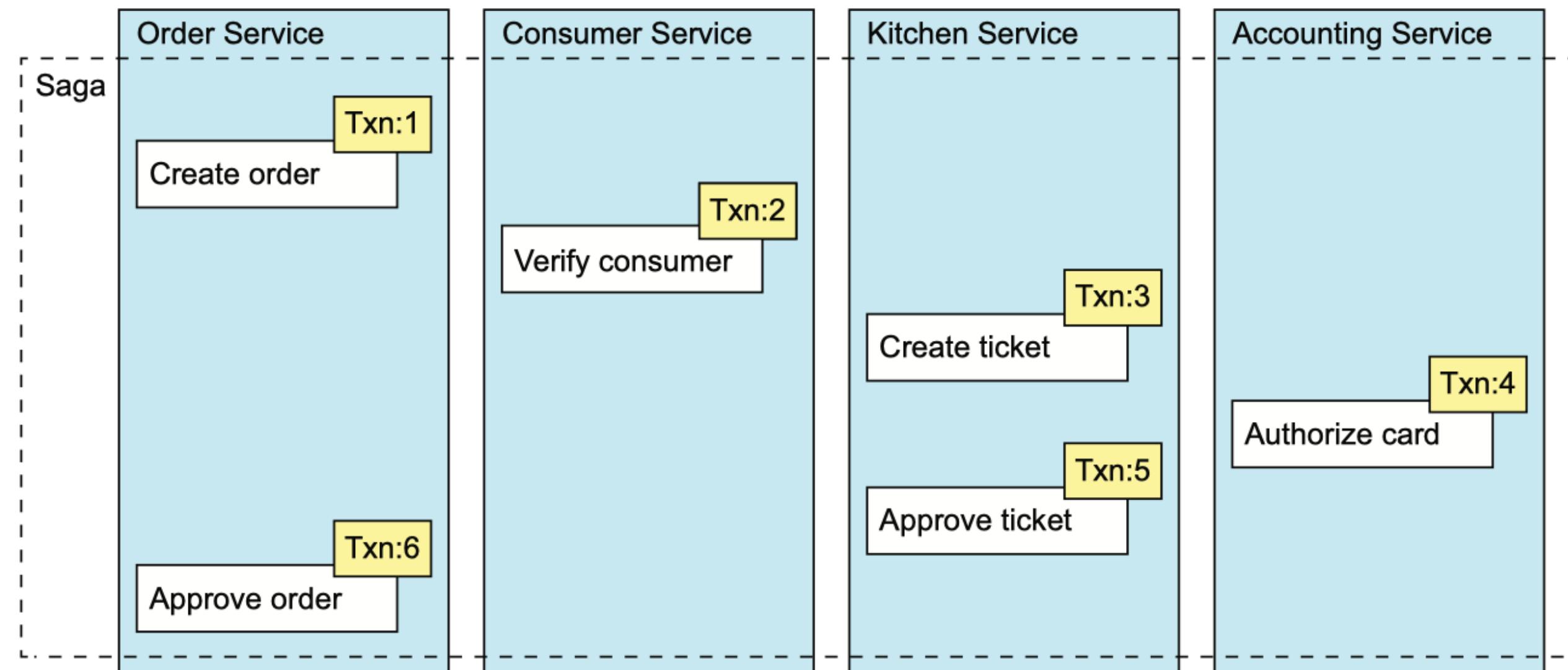






Distributed transaction management

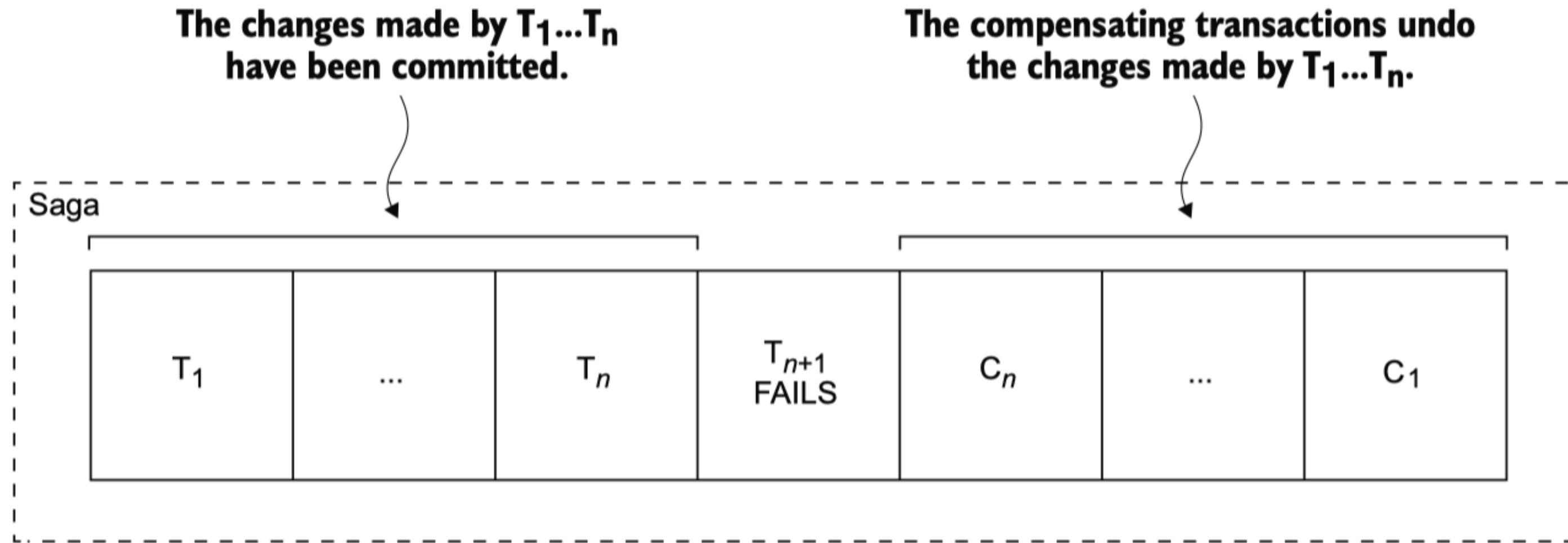
Orchestrator Saga



Saga

- A message-driven sequence of local transactions
- Reacts to command events and generates new commands
- **ACD** (no Isolation)
- Must use countermeasures

Compensating transactions



Anomalies

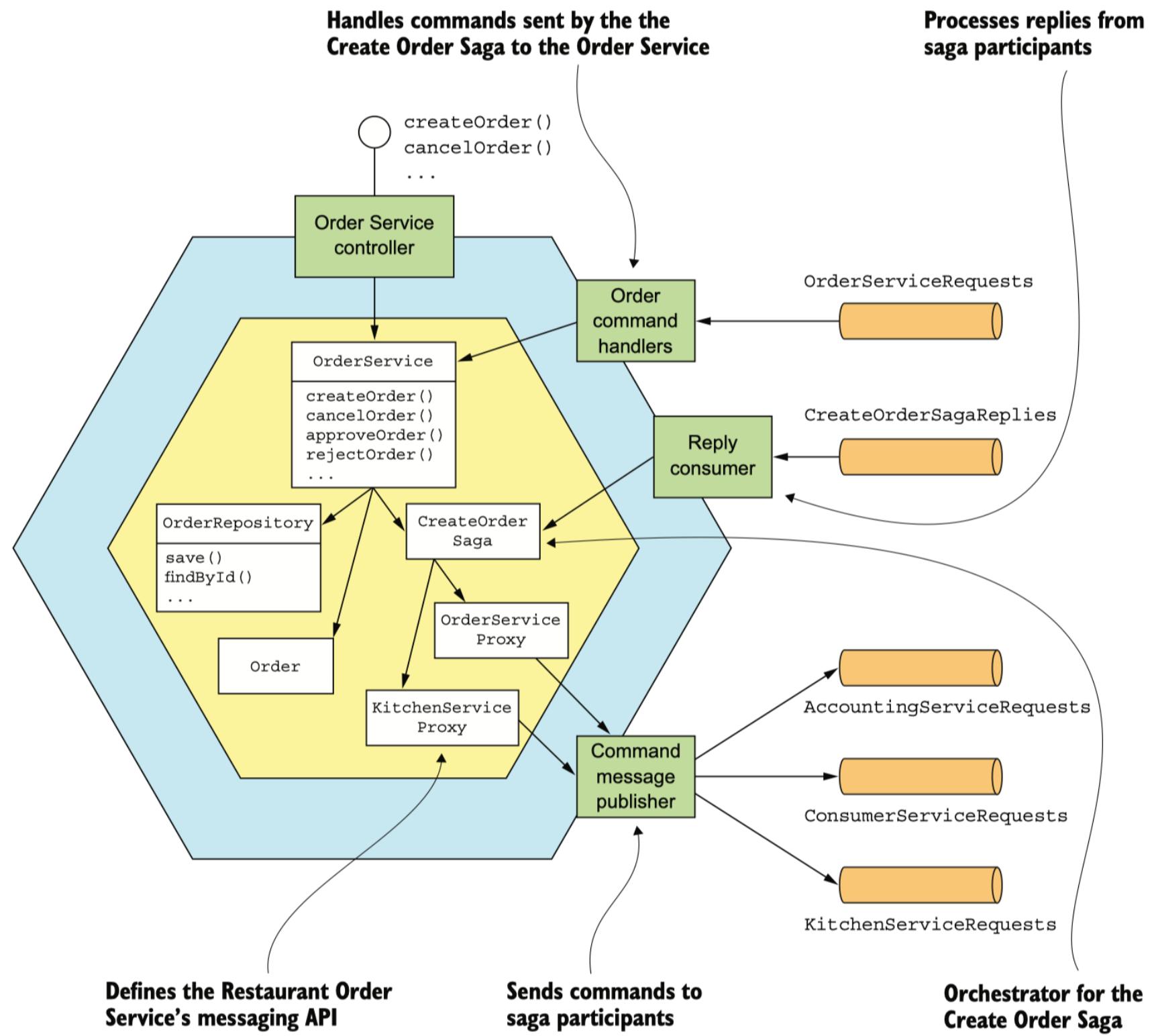
An anomaly is when a transaction reads or writes data in a way that it wouldn't if transactions were executed one at time.

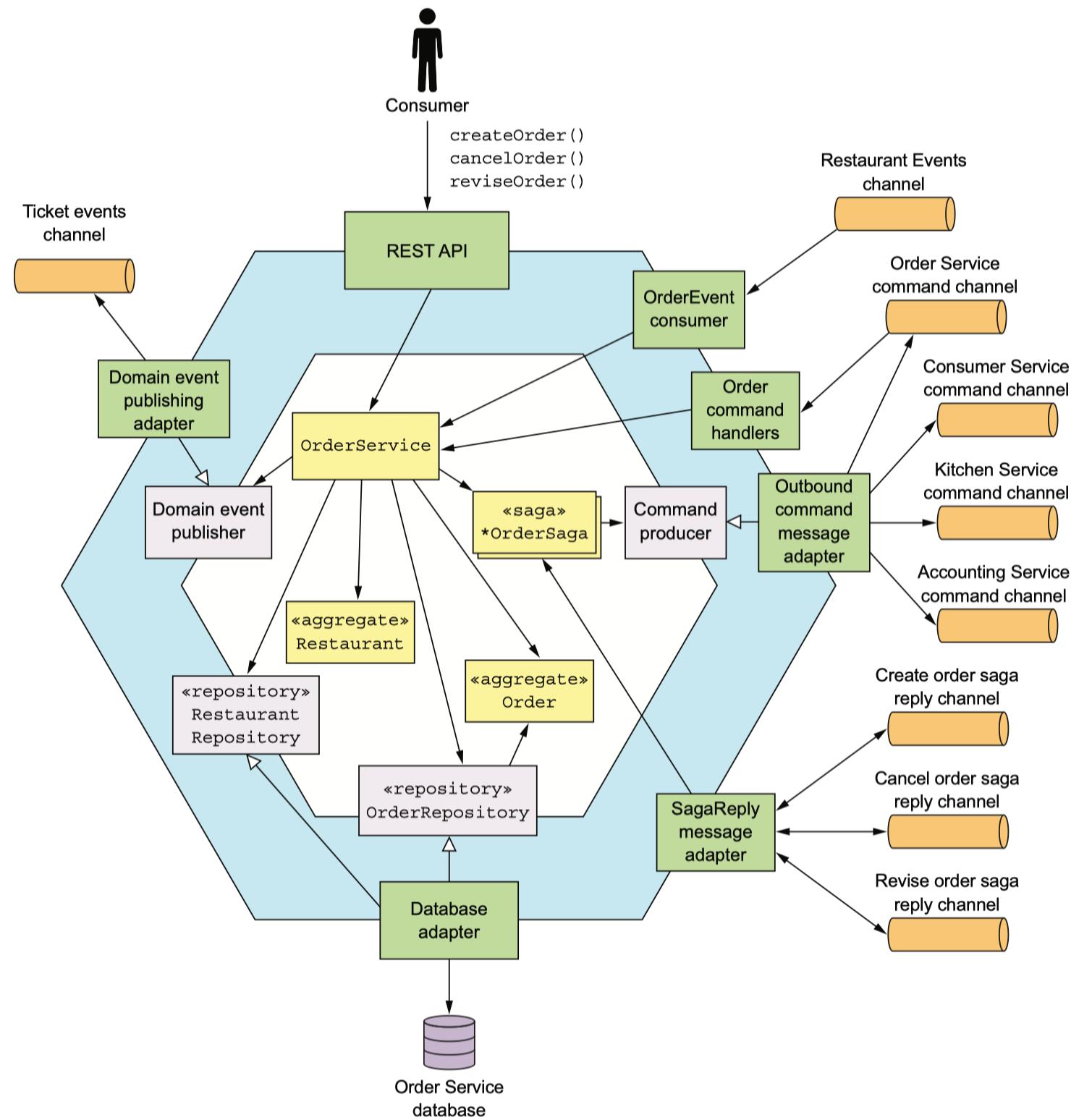
1. Lost updates
2. Dirty reads

Counter-measures²

- Semantic lock
- Commutative updates
- Pessimistic view
- Reread value

² Semantic ACID properties in multidatabases using remote procedure calls and update propagations





Forces

- A paradigm to manage **sequences of async events**
- Push based programming
- Responsive
- Non-blocking

Reactive systems ³

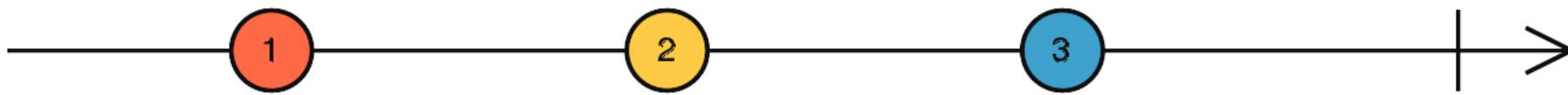
- Systems which continuously respond to inputs
- Reactive systems interact with the environment at a pace dictated by the environment

³ Reactive programming and its effect on performance and the development process

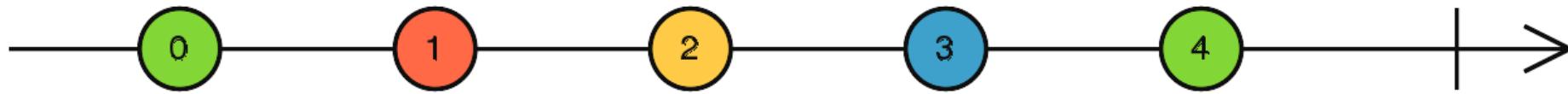
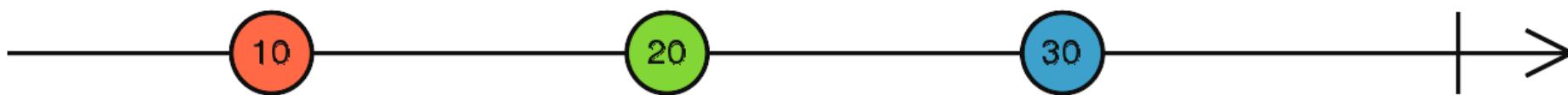
Reactive programming⁴

⁴ Functional Reactive Animation 1997

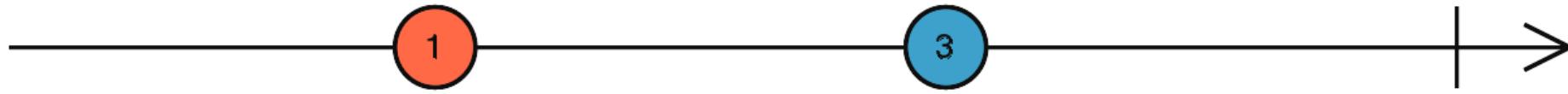
Reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change.

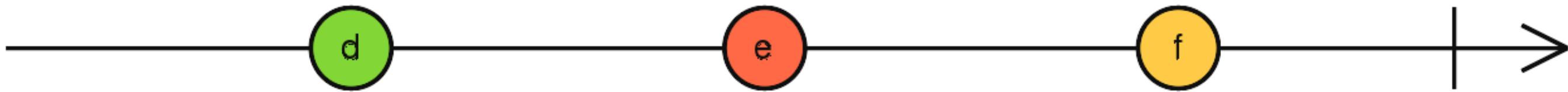
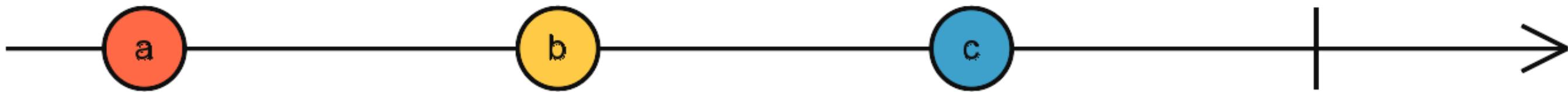


```
map(x => 10 * x)
```

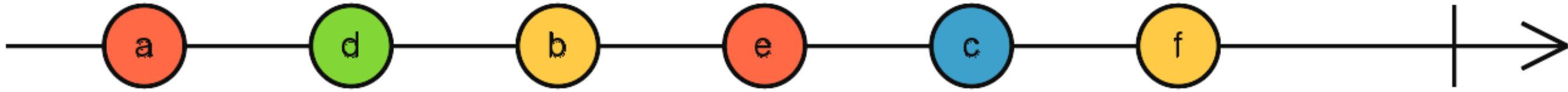


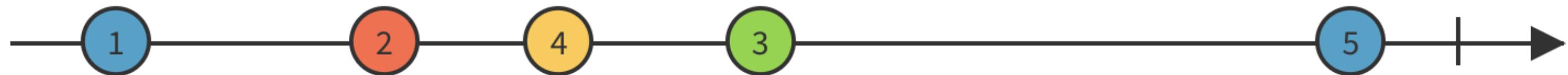
```
filter(x => x % 2 === 1)
```





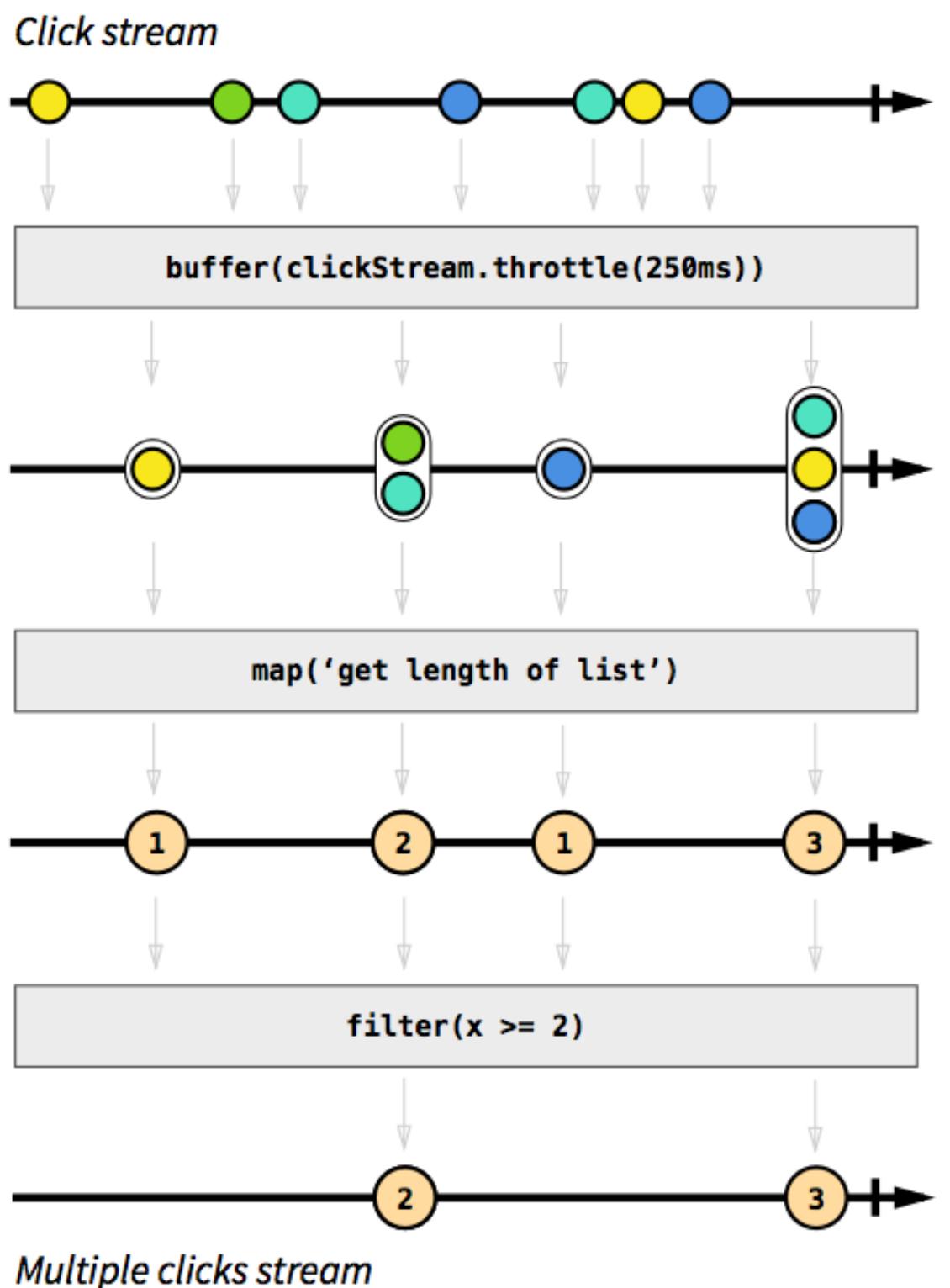
merge





zip





Monolith

Simplicity

Consistency

Inter-module refactoring

Microservice

Partial Deployment

Availability

Preserve Modularity

Multiple Platforms

Independence

The future is already here — it's just not very evenly distributed.

— William Gibson
