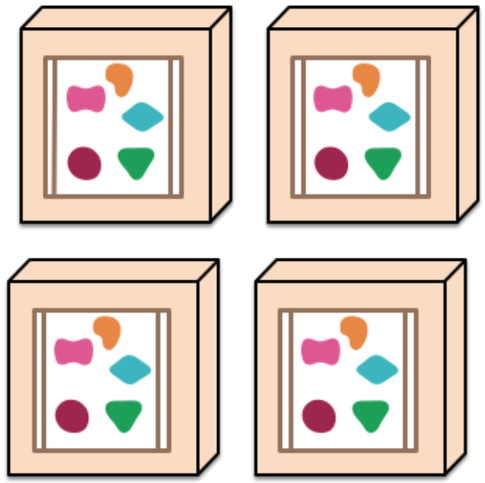
Microservices patterns

Giovanni Jiayi Hu

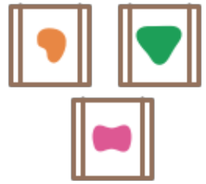
Scaling [1]

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

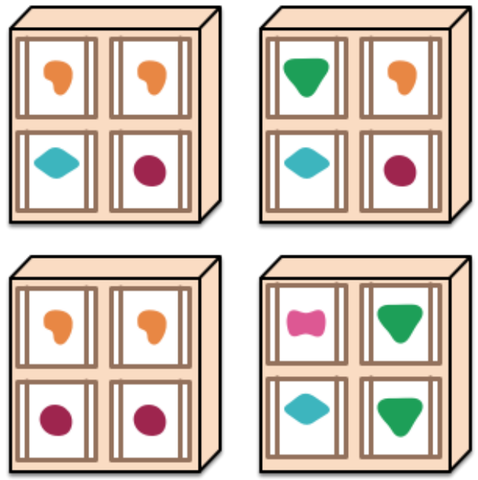
... and scales by replicating the monolith on multiple servers



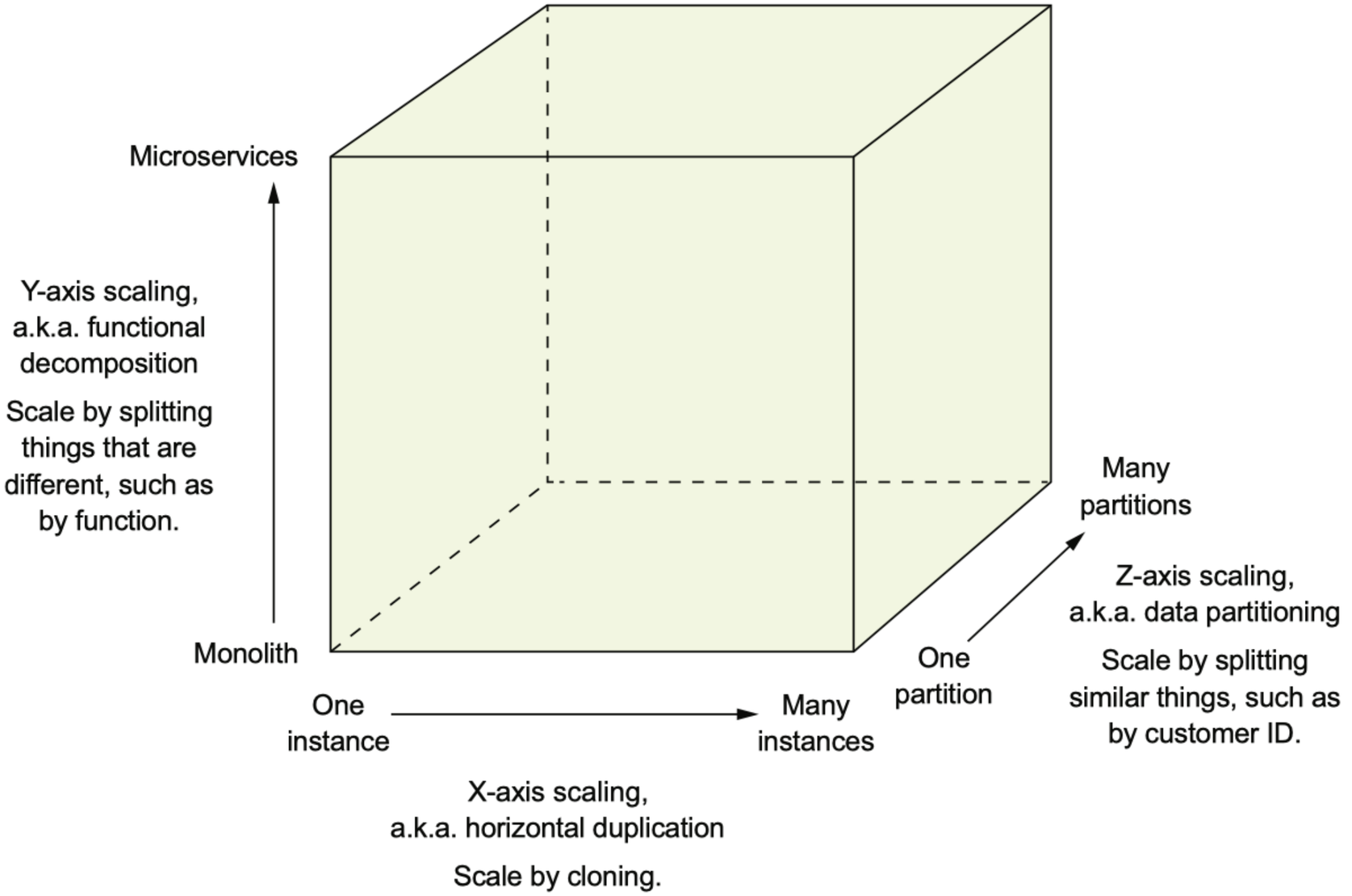
A microservices architecture puts each element of functionality into a separate service...



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



[1] Martin Fowler - Microservices

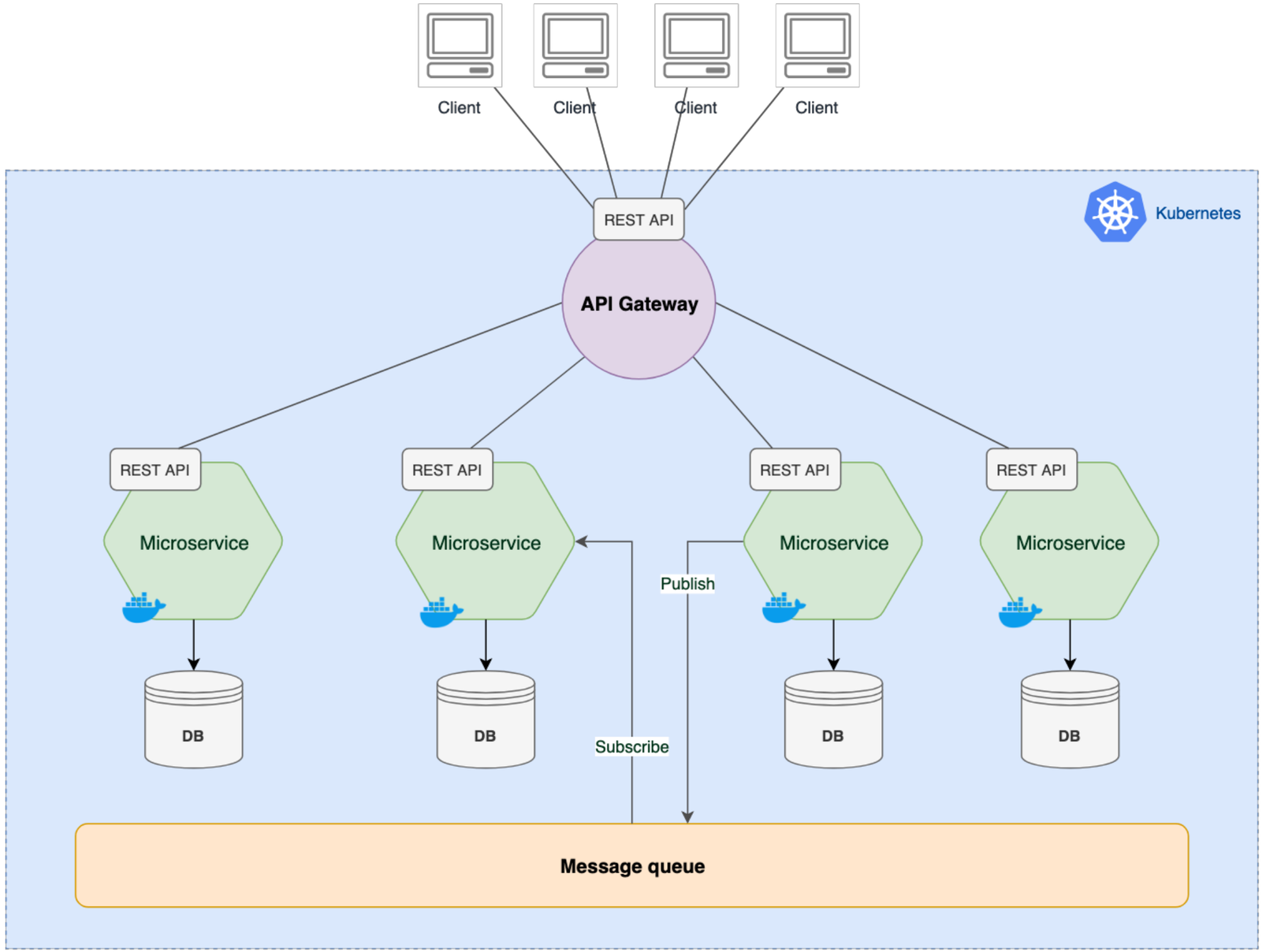

Microservices
Y-axis scaling,
a.k.a. functional
decomposition
Scale by splitting
things that are
different, such as
by function.
Monolith
One
instance
Many
instances
X-axis scaling,
a.k.a. horizontal duplication
Scale by cloning.
One
partition
Many
partitions
Z-axis scaling,
a.k.a. data partitioning
Scale by splitting
similar things, such as
by customer ID.

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**.


Client
Client
Client
Client
Kubernetes
REST API
API Gateway
REST API
REST API
REST API
REST API
Microservice
Microservice
Microservice
Microservice
Publish
DB
DB
DB
DB
Subscribe
Message queue

Self-containing: Docker

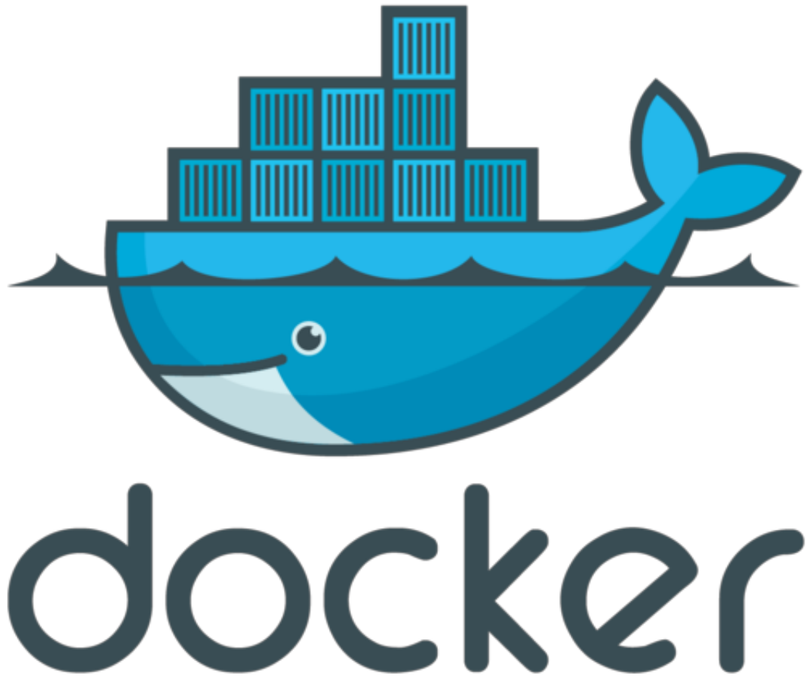
Forces

— Encapsulation of technology stack
— Service isolation
— Resource constraining
— Location indipendence

Solution: Docker

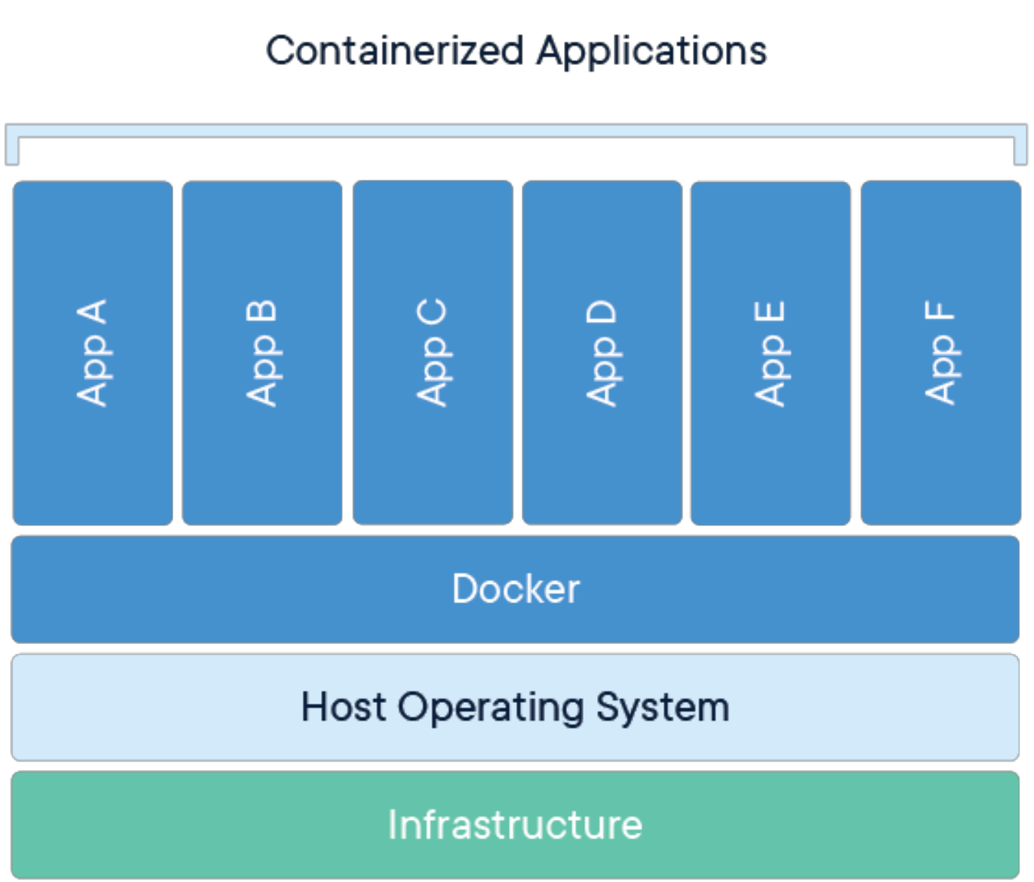
— **Image**: an executable package that includes everything needed to run an application

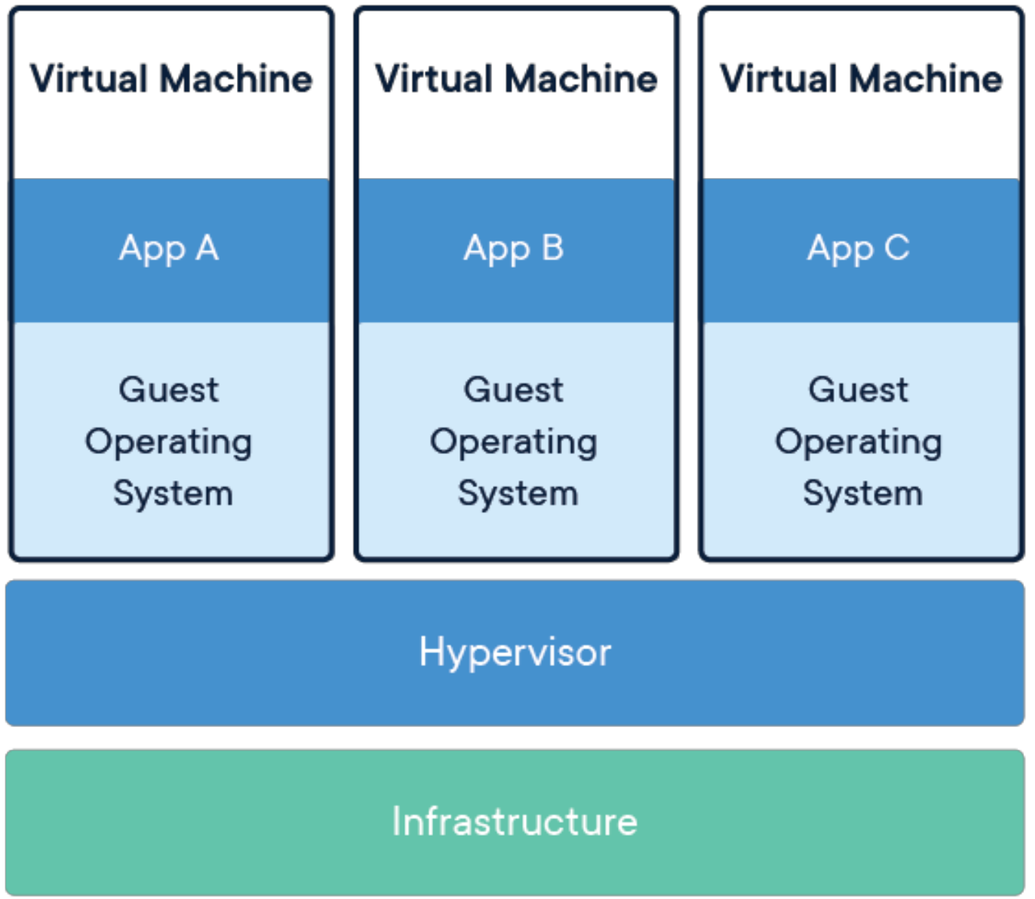
— **Isolated containers**: runtime instance of image



Containerization

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


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure


Virtual Machine
Virtual Machine
Virtual Machine
App A
App B
App C
Guest Operating System
Guest Operating System
Guest Operating System
Hypervisor
Infrastructure

Container vs Virtual Machines

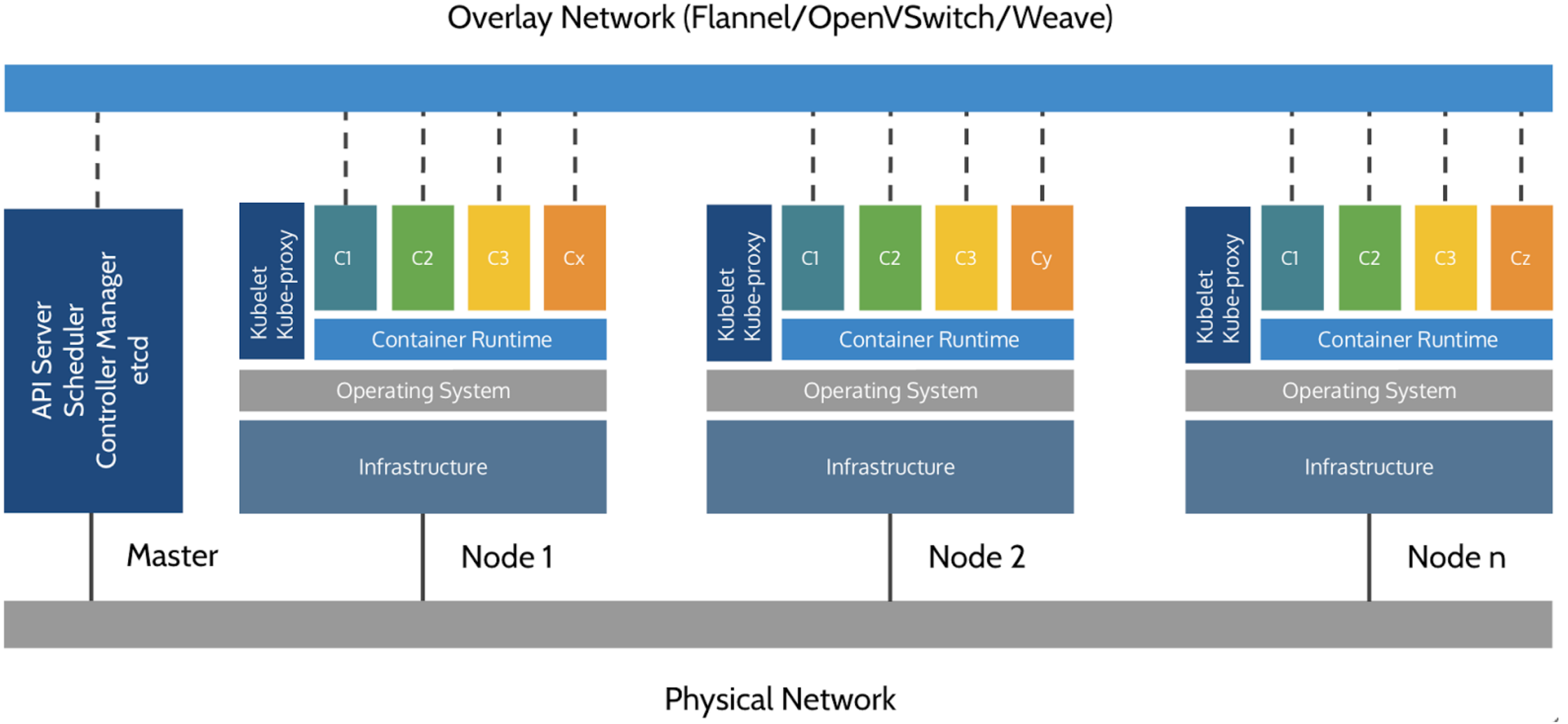
- Shared host kernel and OS resources
- Less isolation but more lightweight
- Snapshot images
- Weights MB

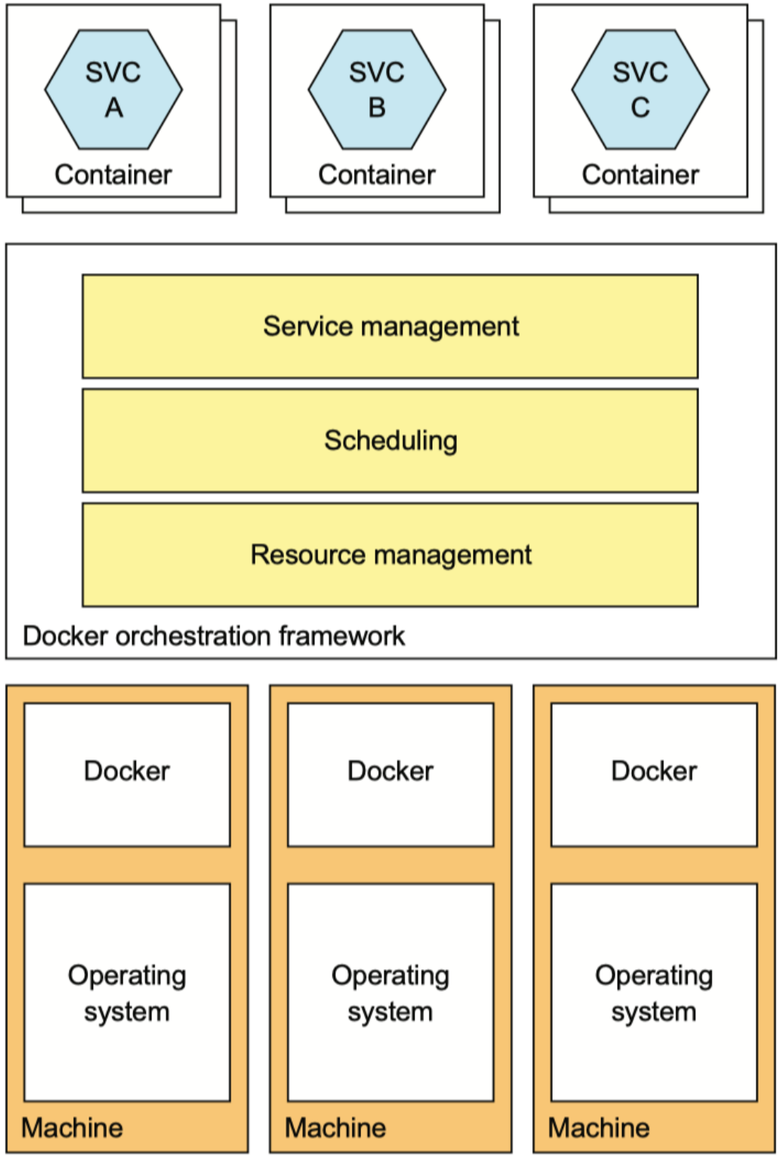
Orchestrating - Kubernetes

Kubernetes (K8s)

Declarative configuration and **automation** for for managing containerized services

kubernetes


Overlay Network (Flannel/OpenVSwitch/Weave)
API Server
Scheduler
Controller Manager
etcd
Kubelet
Kube-proxy
C1
C2
C3
Cx
Container Runtime
Operating System
Infrastructure
Master
Kubelet
Kube-proxy
C1
C2
C3
Cy
Container Runtime
Operating System
Infrastructure
Node 1
Kubelet
Kube-proxy
C1
C2
C3
Cz
Container Runtime
Operating System
Infrastructure
Node 2
Node n
Physical Network


SVC A
Container
SVC B
Container
SVC C
Container
Service management
Scheduling
Resource management
Docker orchestration framework
Docker
Operating system
Machine
Docker
Operating system
Machine
Docker
Operating system
Machine

Auto-scaling

— *desiredReplicas = currentReplicas * (currentMetricValue / 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

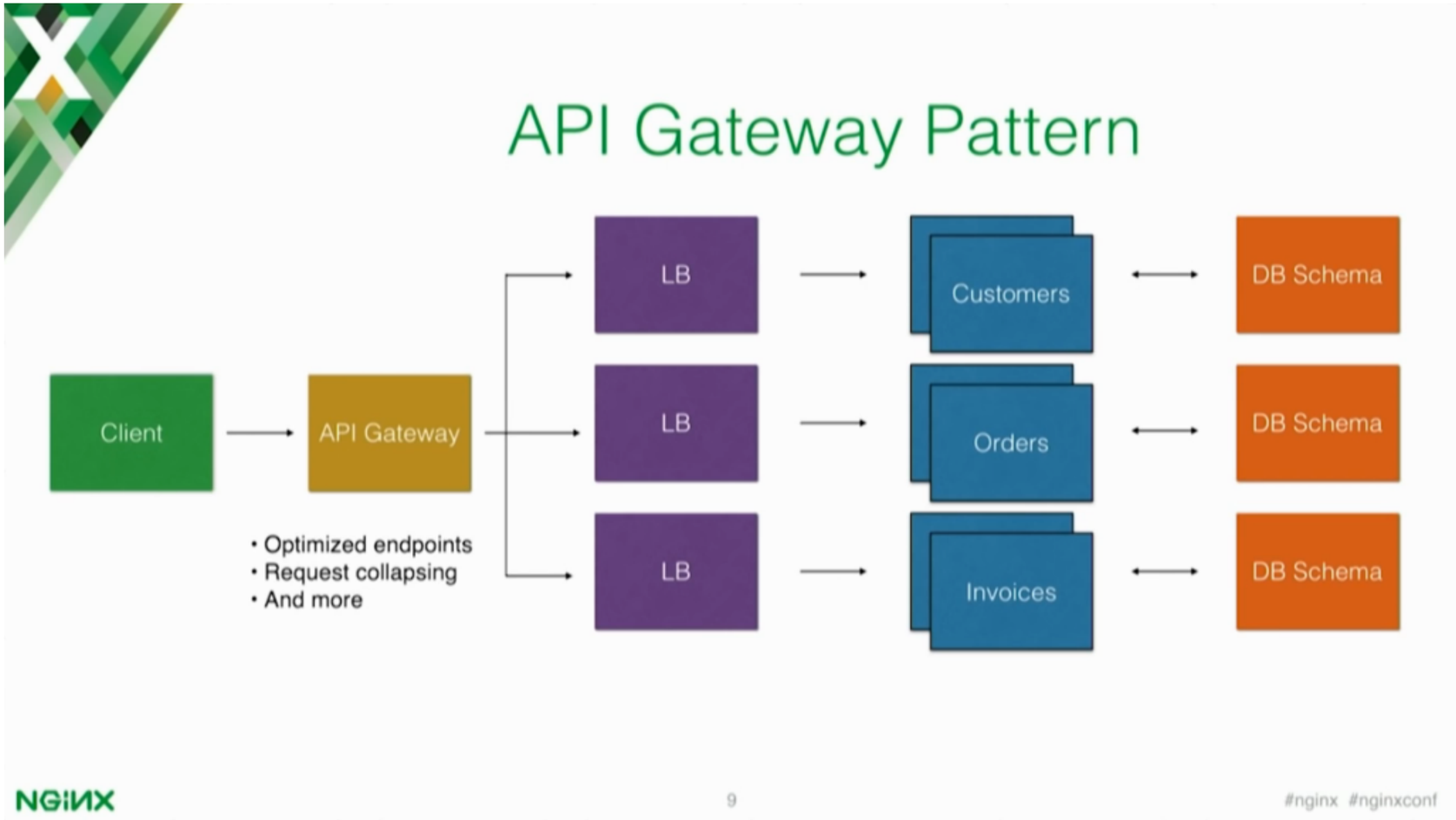
— Use health checks

— 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 Pattern
Client
API Gateway
• Optimized endpoints
• Request collapsing
• And more
LB
LB
LB
Customers
Orders
Invoices
DB Schema
DB Schema
DB Schema
NGINX
9
#nginx #nginxconf

API Gateway - Ambassador

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

Database architecture

Relational databases scalability

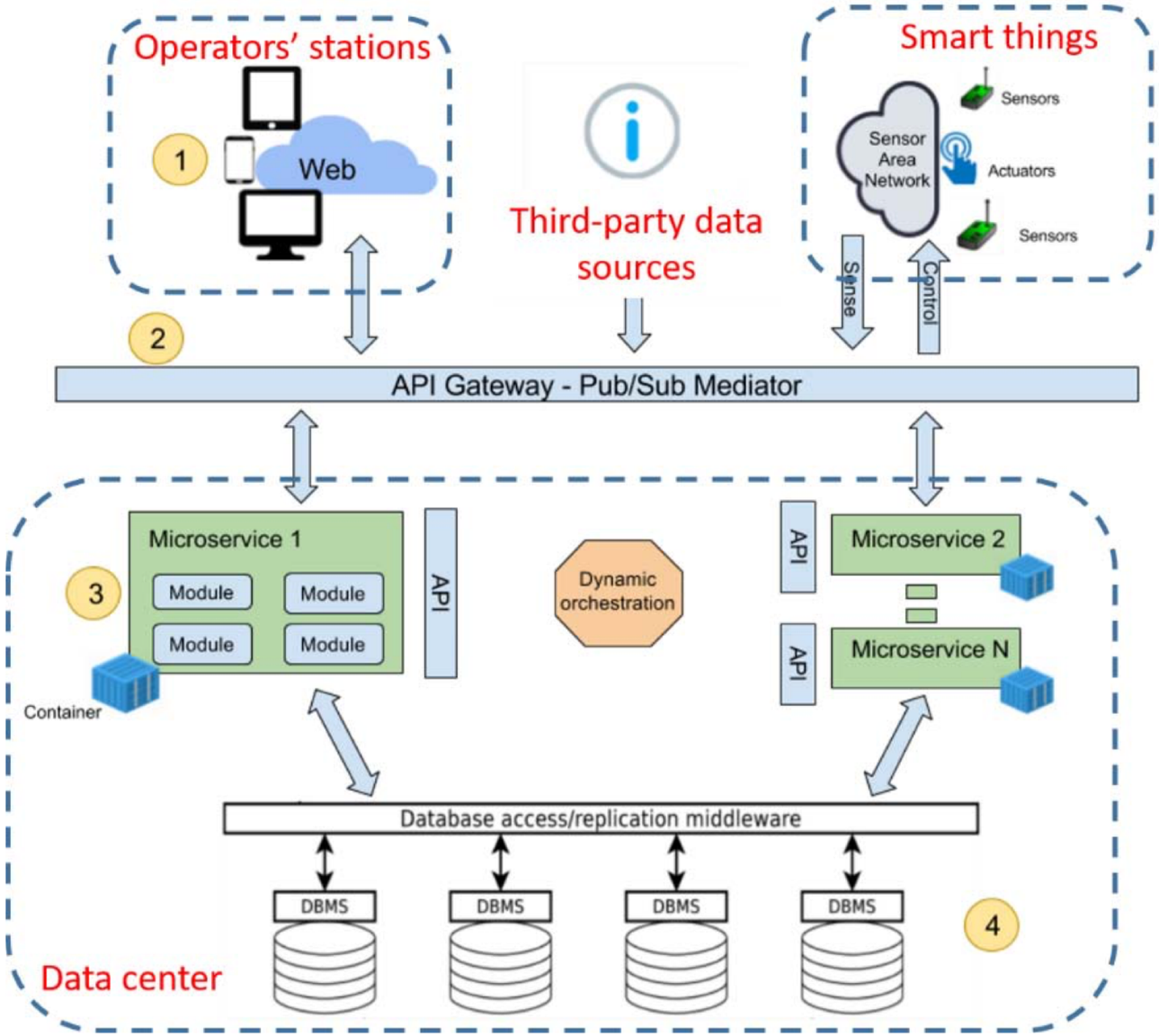
1. Inefficient joins across nodes
2. Distributed transactional processing: X/Open XA
3. Reduced availability

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**


Operators' stations
1
Web
Third-party data sources
Smart things
Sensor Area Network
Sensors
Actuators
Sensors
Sense
Control
2
API Gateway - Pub/Sub Mediator
Microservice 1
3
Module
Module
Module
Module
API
Container
Dynamic orchestration
API
Microservice 2
API
Microservice N
Database access/replication middleware
DBMS
DBMS
DBMS
DBMS
4
Data center

Database per service - Advantages

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

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

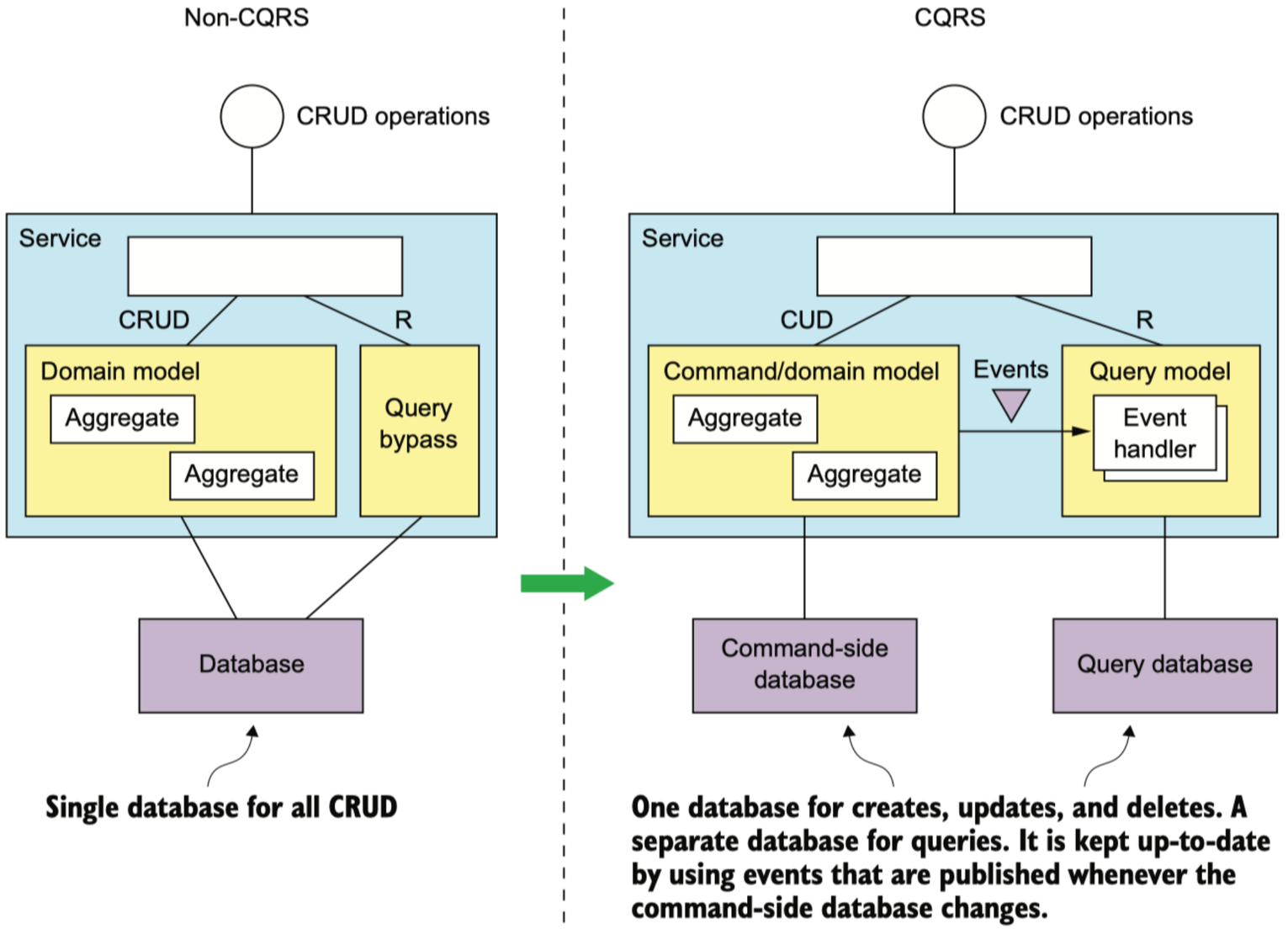
Application level patterns

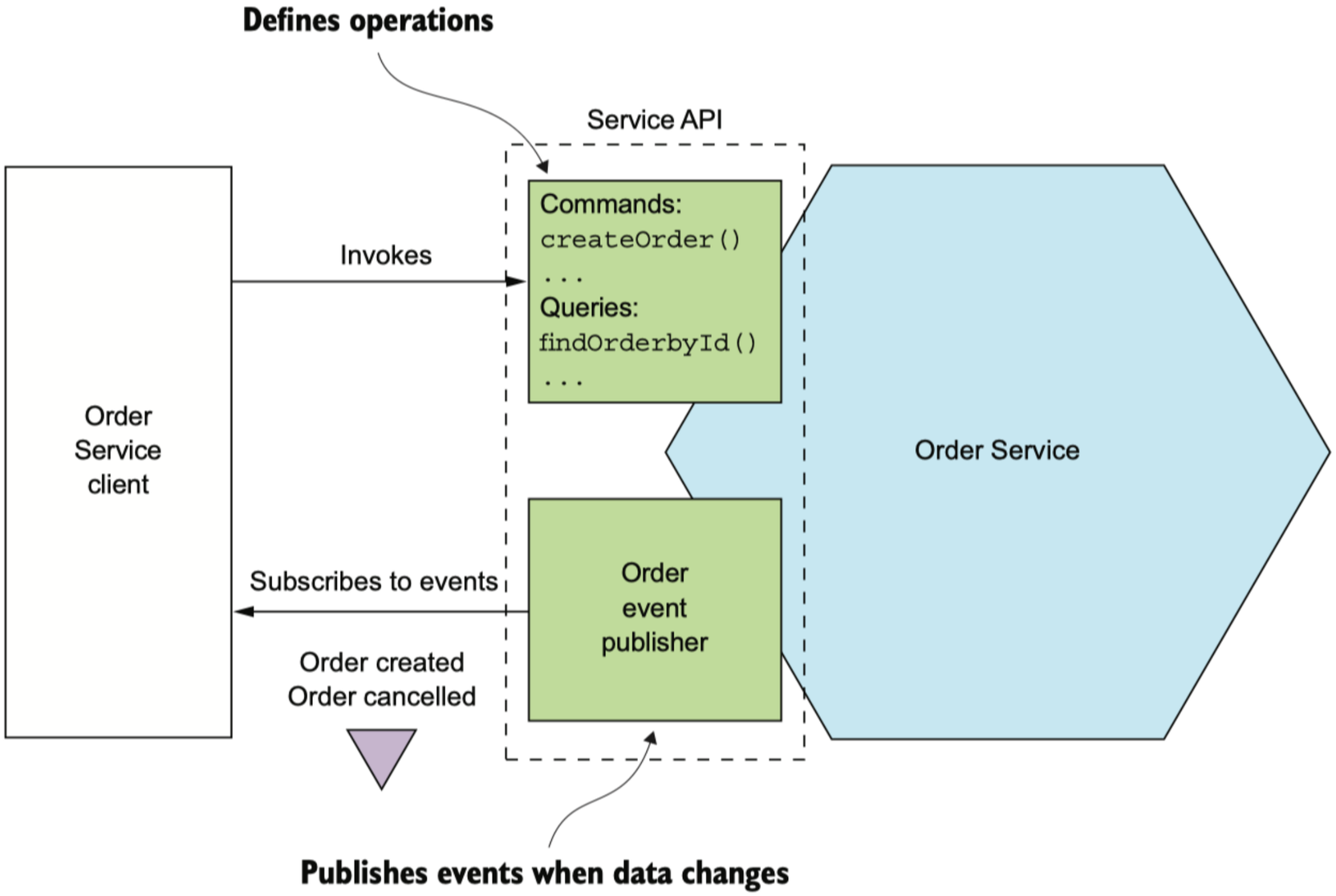
Forces

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)**


Non-CQRS
CRUD operations
Service
CRUD
R
Domain model
Aggregate
Aggregate
Query bypass
Database
Single database for all CRUD
CQRS
CRUD operations
Service
CUD
R
Command/domain model
Aggregate
Aggregate
Events
Query model
Event handler
Command-side database
Query database
One database for creates, updates, and deletes. A separate database for queries. It is kept up-to-date by using events that are published whenever the command-side database changes.


Defines operations
Service API
Commands:
createOrder()
...
Queries:
findOrderbyId()
...
Invokes
Order
Service
client
Order Service
Subscribes to events
Order
event
publisher
Order created
Order cancelled
Publishes events when data changes

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**

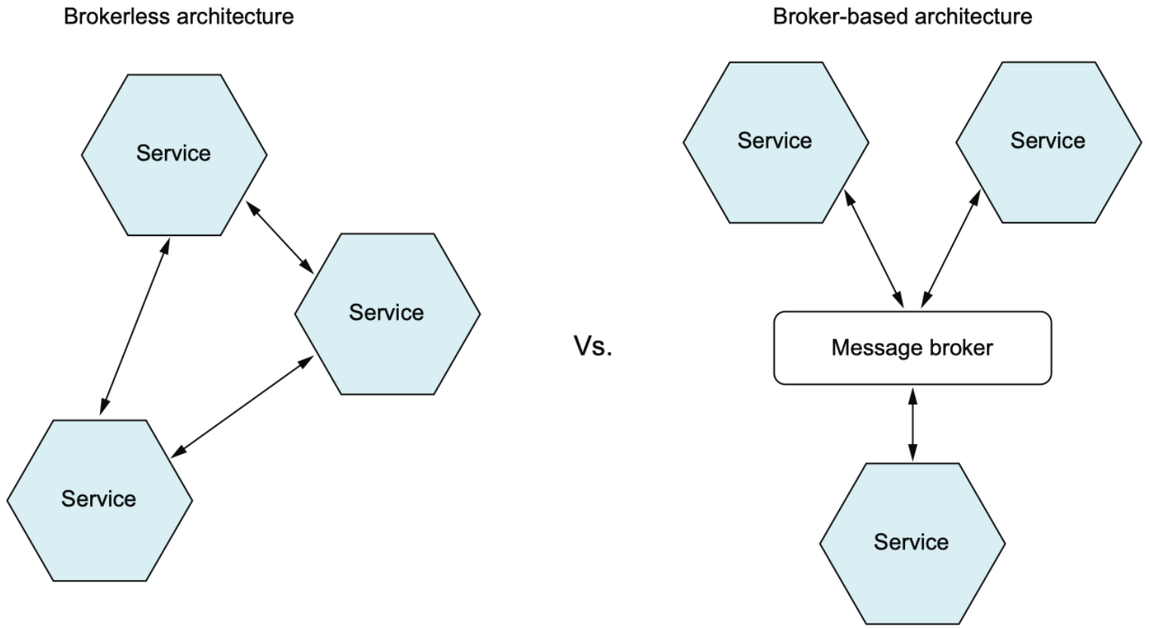
The biggest issue in changing a monolith into microservices lies in changing the communication pattern.

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

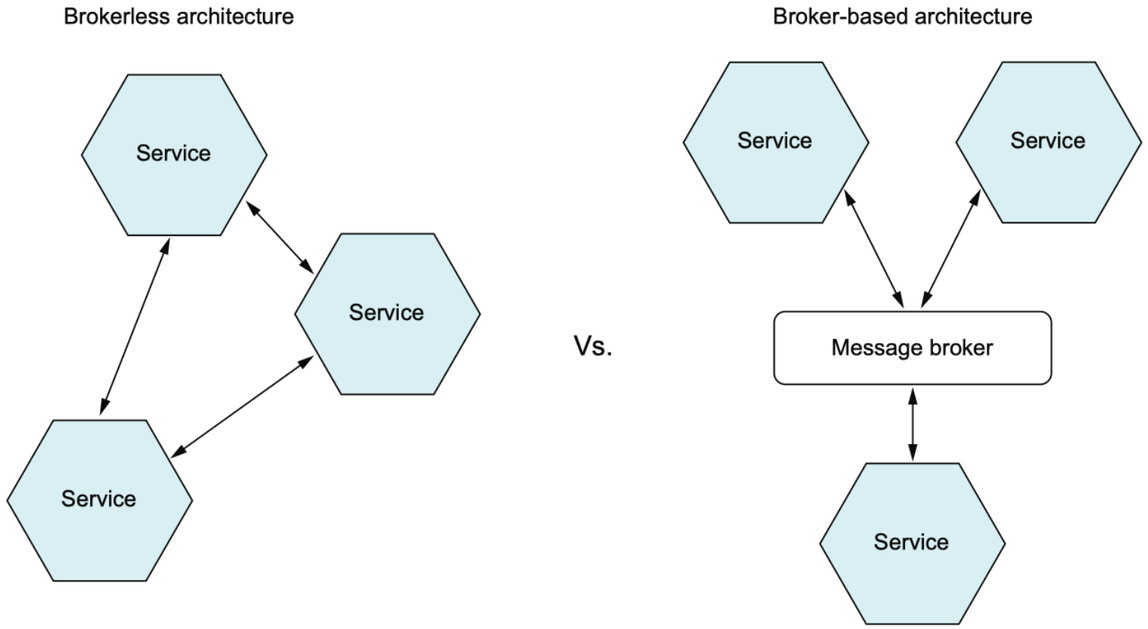
- Loose coupling
- No discovery mechanism
- Message buffering
- Explicit interprocess communication
- Can convert messages to be understood by the destination



Broker-based MQ - Disadvantages

— Potential performance bottleneck

— Potential single point of failure



Message broker charateristics

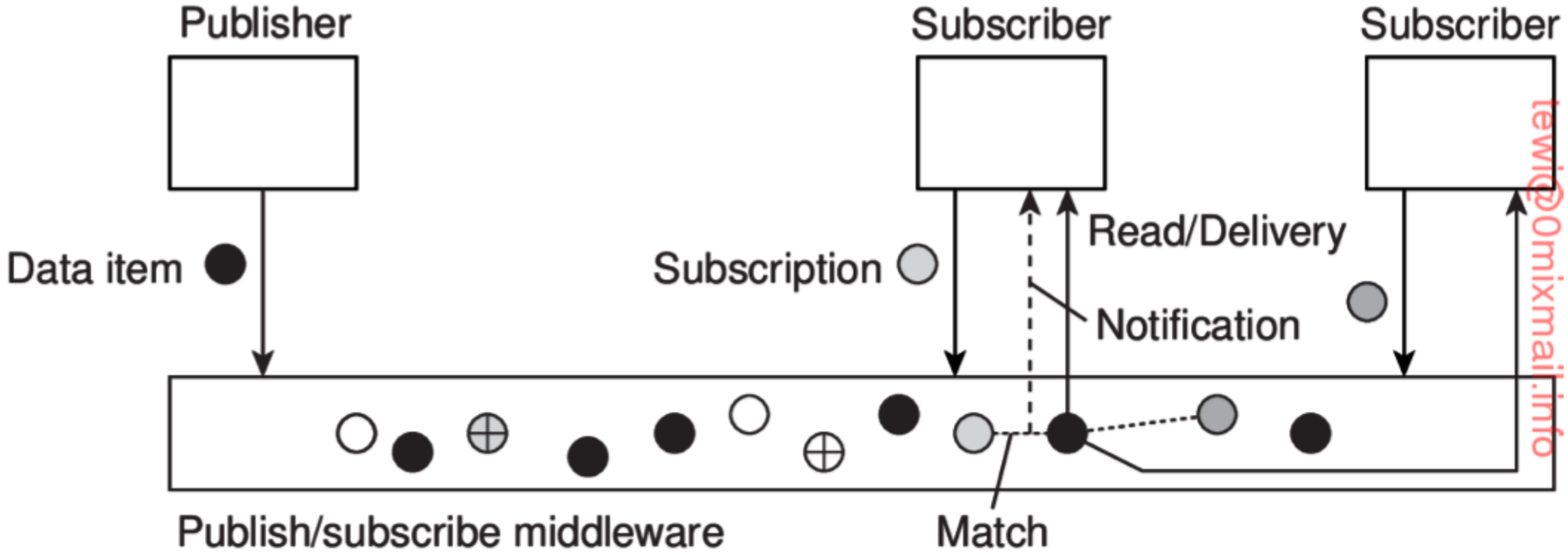
— Message ordering
— Delivery guarantees
— Persistence
— Durability
— Latency

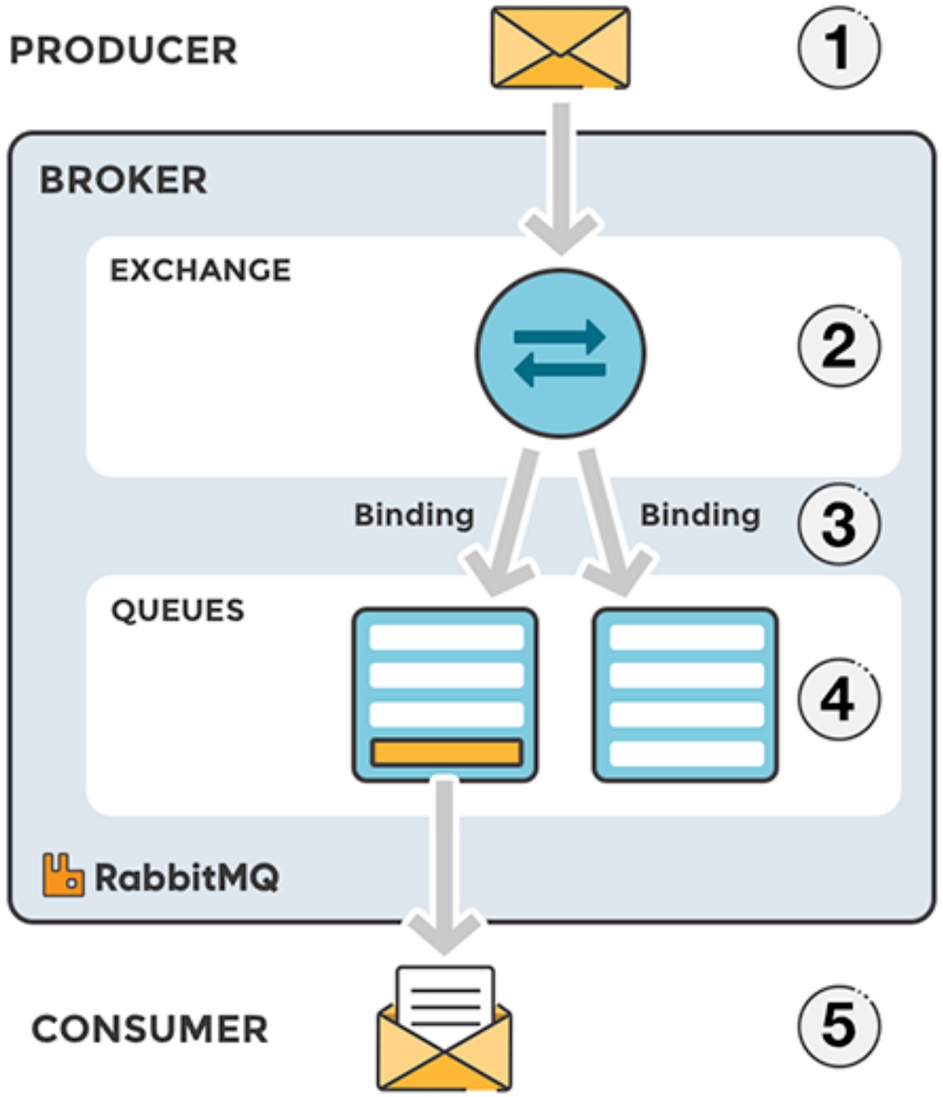
Not a panacea

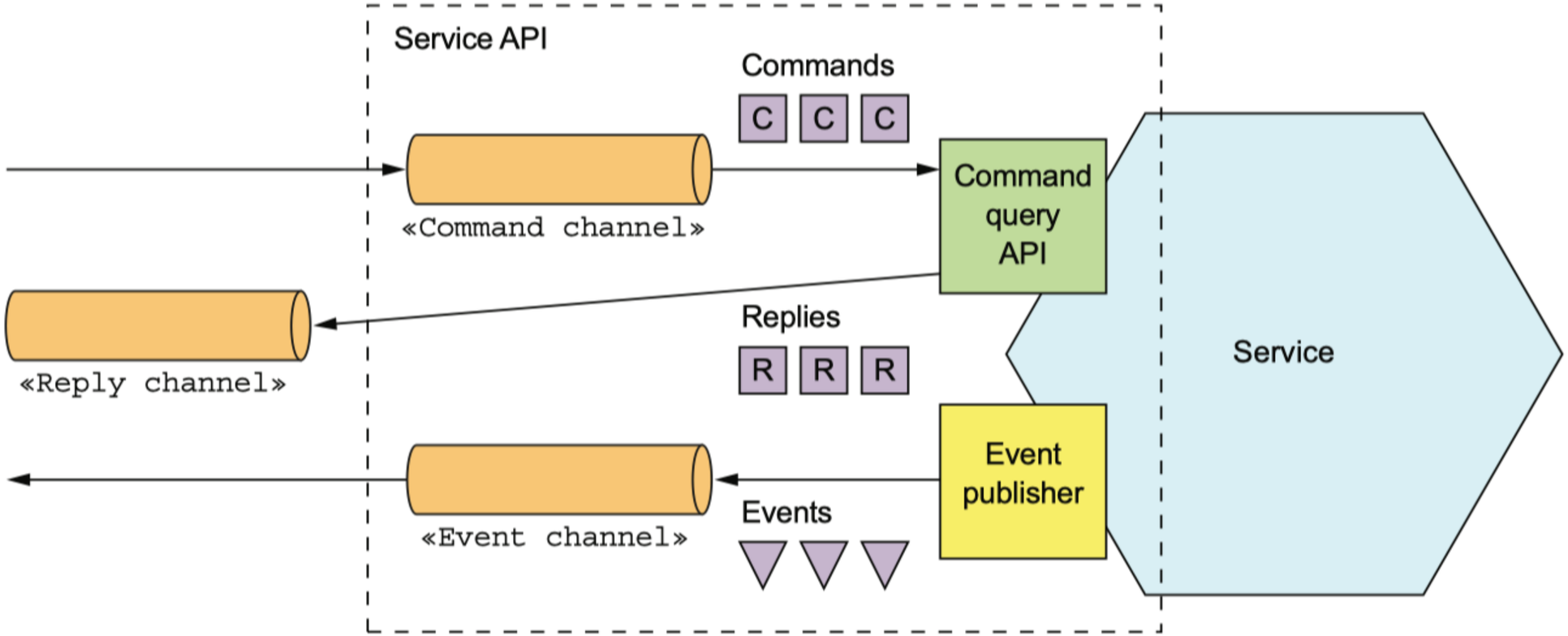
- Message duplication
  - **Idempotent** message handlers
  - Track messages
- Transactional messaging

RabbitMQ

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

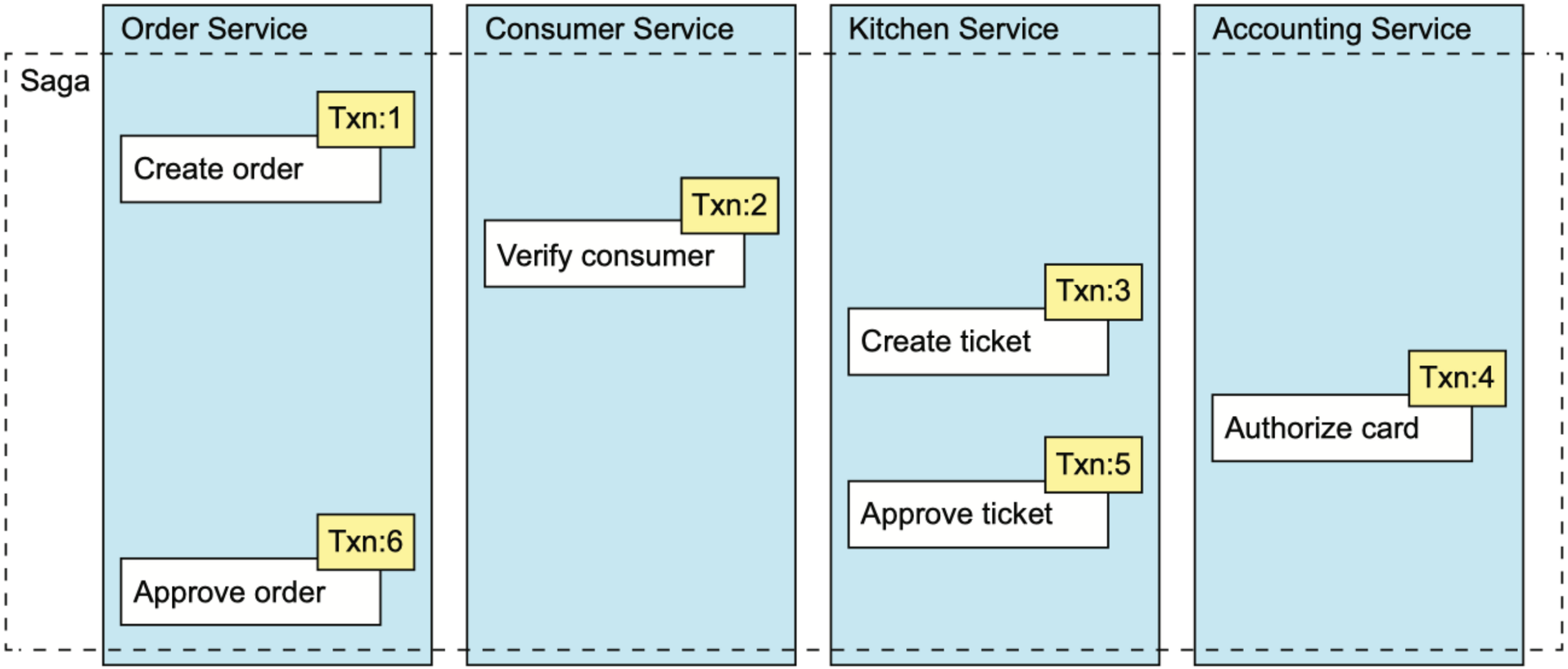



PRODUCER
1
BROKER
EXCHANGE
2
Binding
Binding
3
QUEUES
4
RabbitMQ
CONSUMER
5


Service API
Commands
C
C
C
«Command channel»
Command query API
Service
Replies
R
R
R
«Reply channel»
Event publisher
Events
«Event channel»

Distributed transaction management

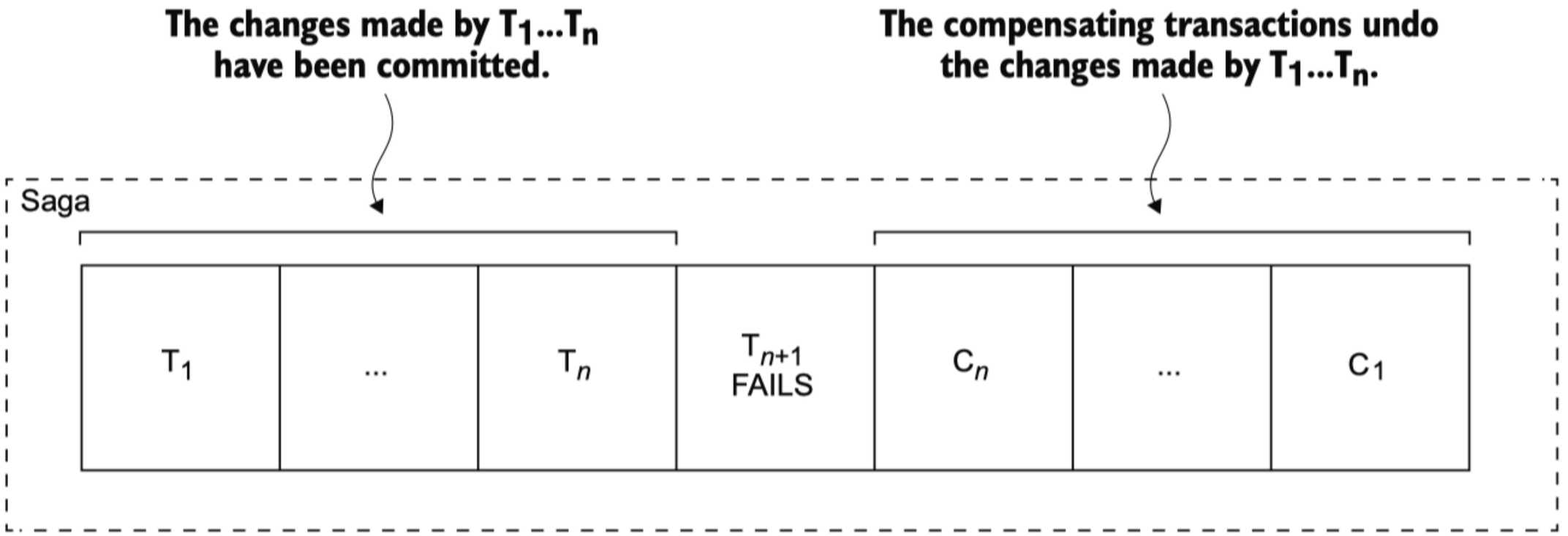
Orchestrator Saga



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

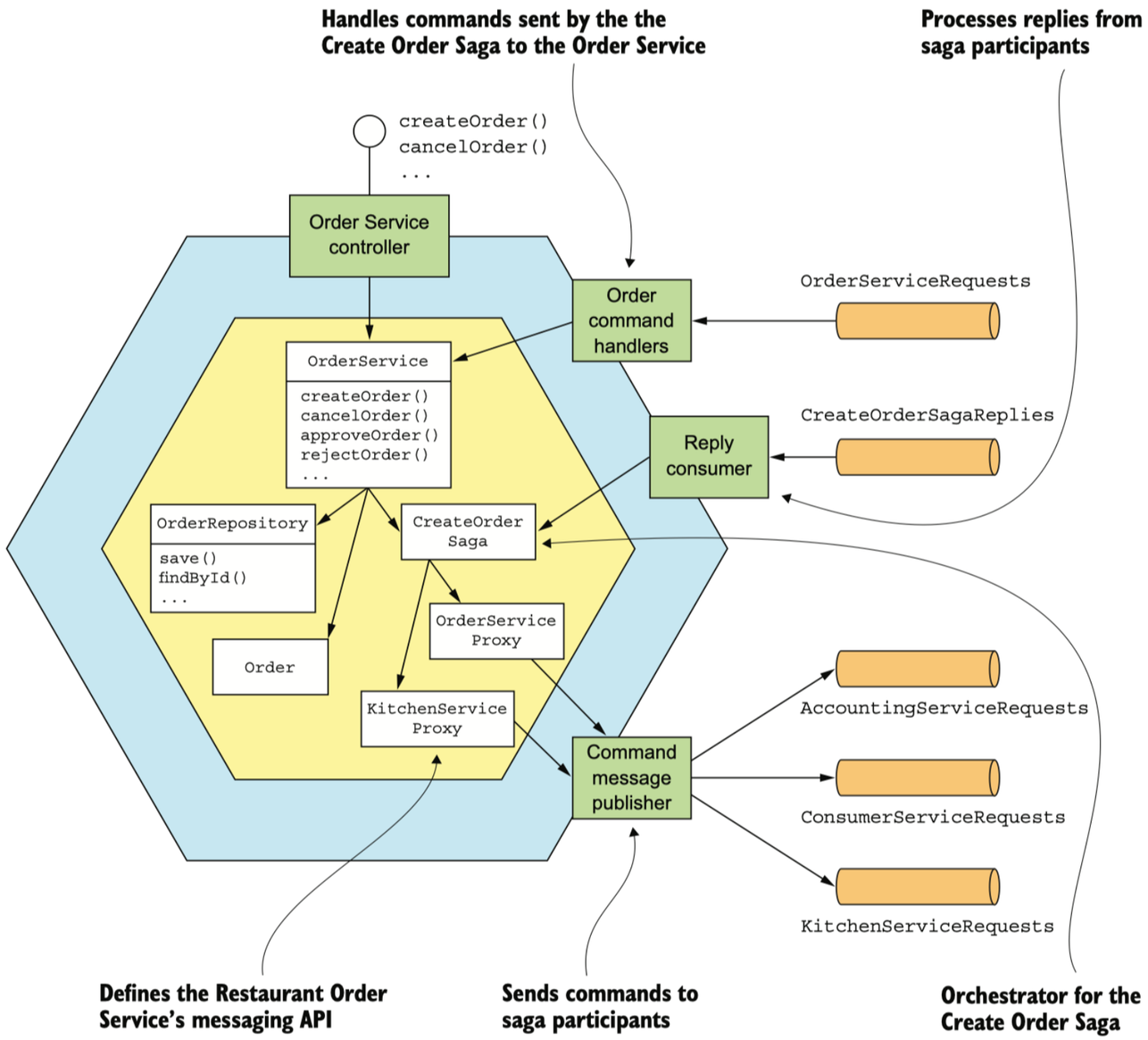
An anomaly is when a transaction reads or writes data in a way that it wouldn’t if transactions were executed one at time.

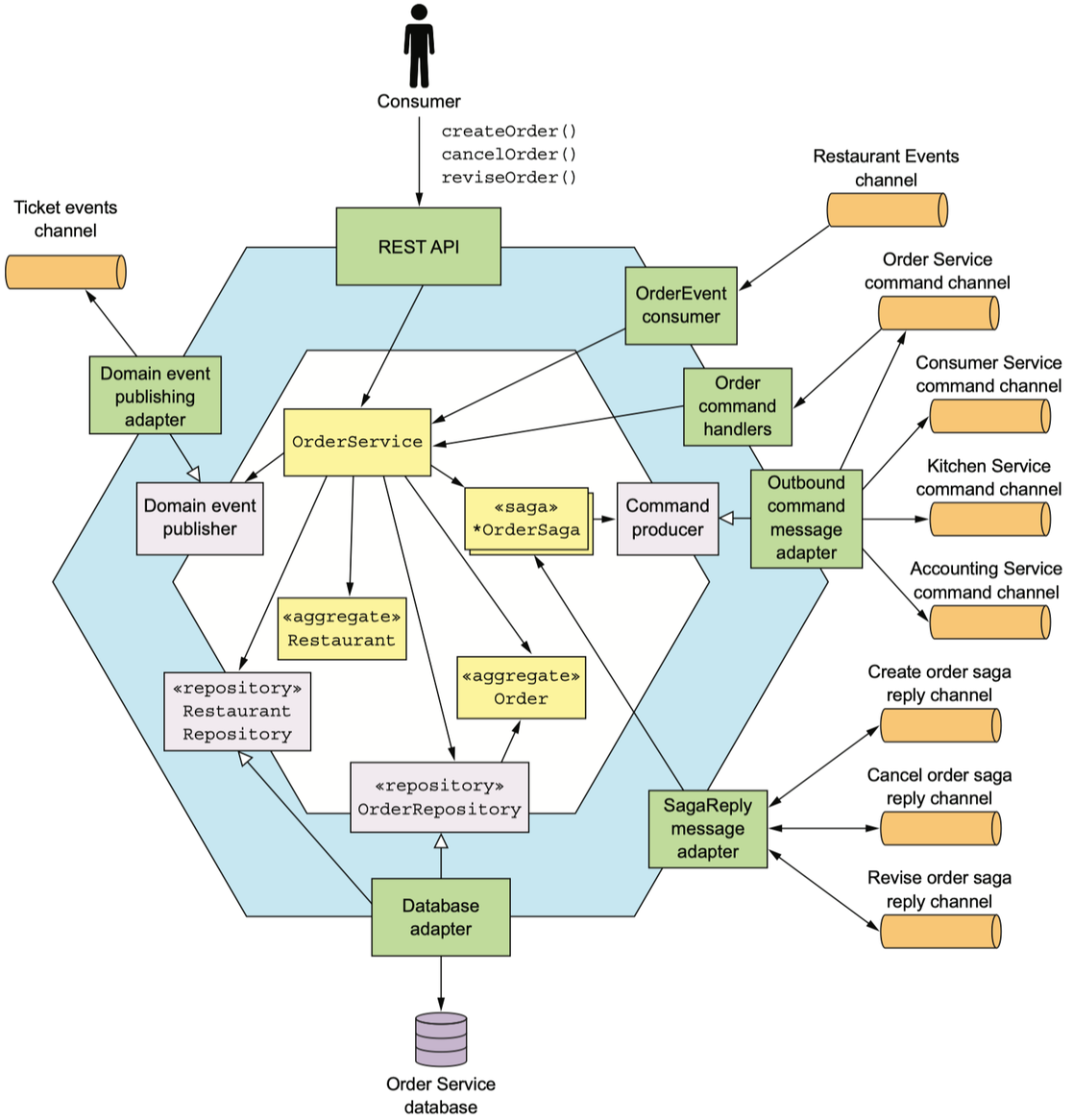
1. Lost updates
2. Dirty reads

Counter-measures [2]

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

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


Handles commands sent by the the Create Order Saga to the Order Service
Processes replies from saga participants
createOrder()
cancelOrder()
...
Order Service controller
Order command handlers
OrderServiceRequests
OrderService
createOrder()
cancelOrder()
approveOrder()
rejectOrder()
...
Reply consumer
CreateOrderSagaReplies
OrderRepository
save()
findById()
...
CreateOrder Saga
OrderService Proxy
Order
KitchenService Proxy
Command message publisher
AccountingServiceRequests
ConsumerServiceRequests
KitchenServiceRequests
Defines the Restaurant Order Service's messaging API
Sends commands to saga participants
Orchestrator for the Create Order Saga


Consumer
createOrder()
cancelOrder()
reviseOrder()
Ticket events channel
REST API
Restaurant Events channel
OrderEvent consumer
Order Service command channel
Domain event publishing adapter
Order command handlers
Consumer Service command channel
OrderService
Kitchen Service command channel
Domain event publisher
«saga» *OrderSaga
Command producer
Outbound command message adapter
Accounting Service command channel
«aggregate» Restaurant
«aggregate» Order
Create order saga reply channel
«repository» Restaurant Repository
Cancel order saga reply channel
«repository» OrderRepository
SagaReply message adapter
Revise order saga reply channel
Database adapter
Order Service database

Forces

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

Reactive systems [3]

— Systems which continuously respond to inputs

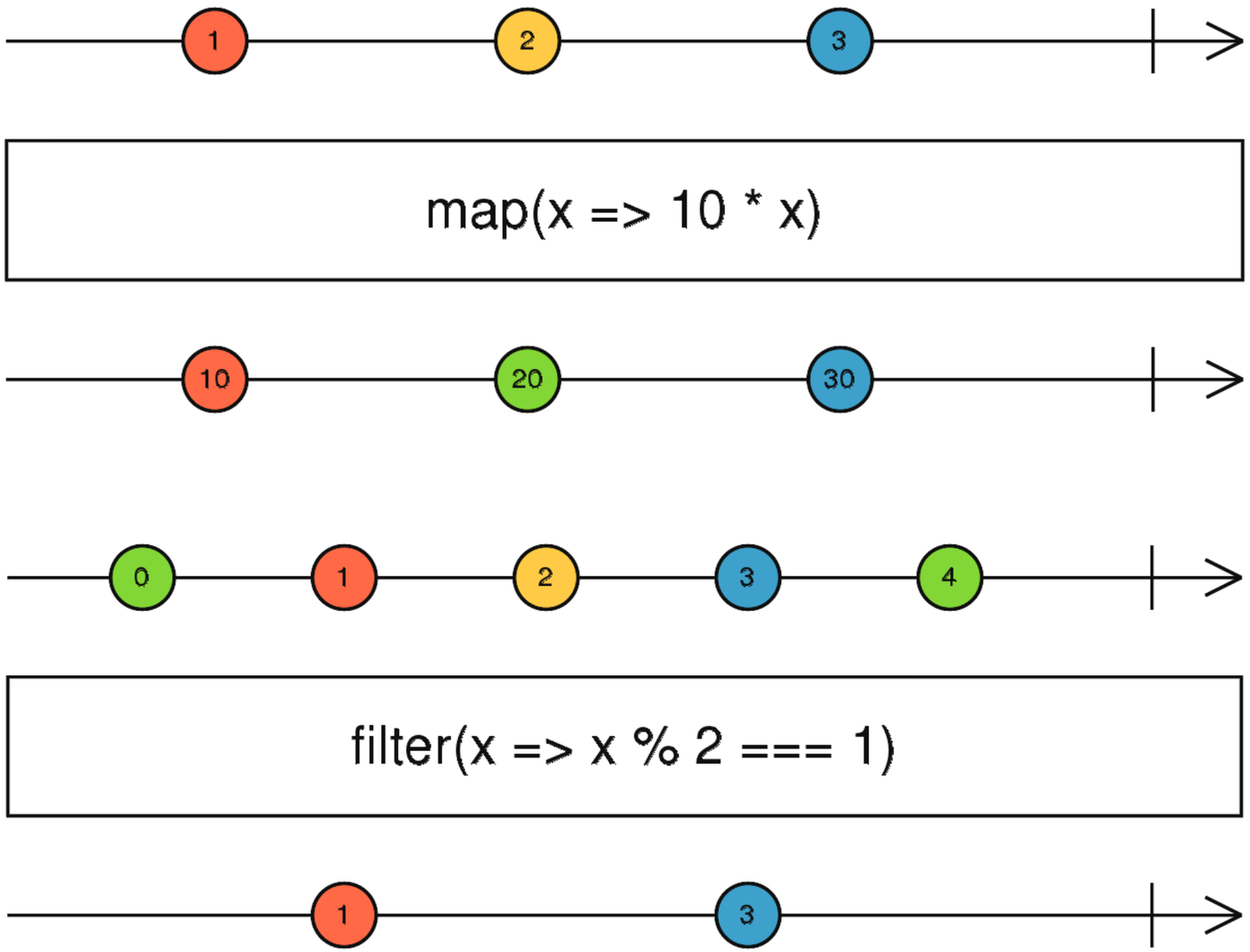
— Reactive systems interact with the environment at a pace dictated by the environment

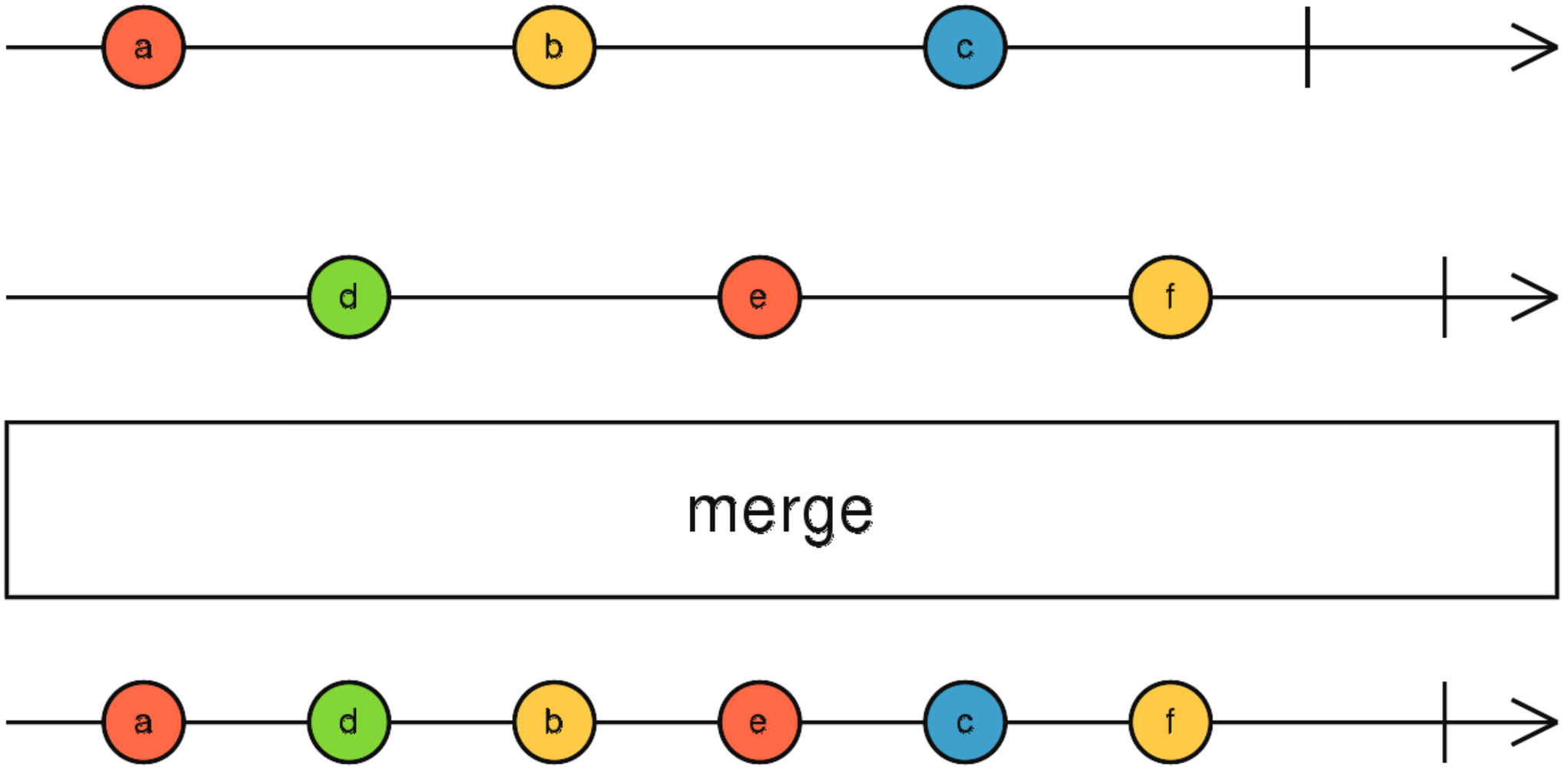
[3] Reactive programming and its effect on performance and the development process

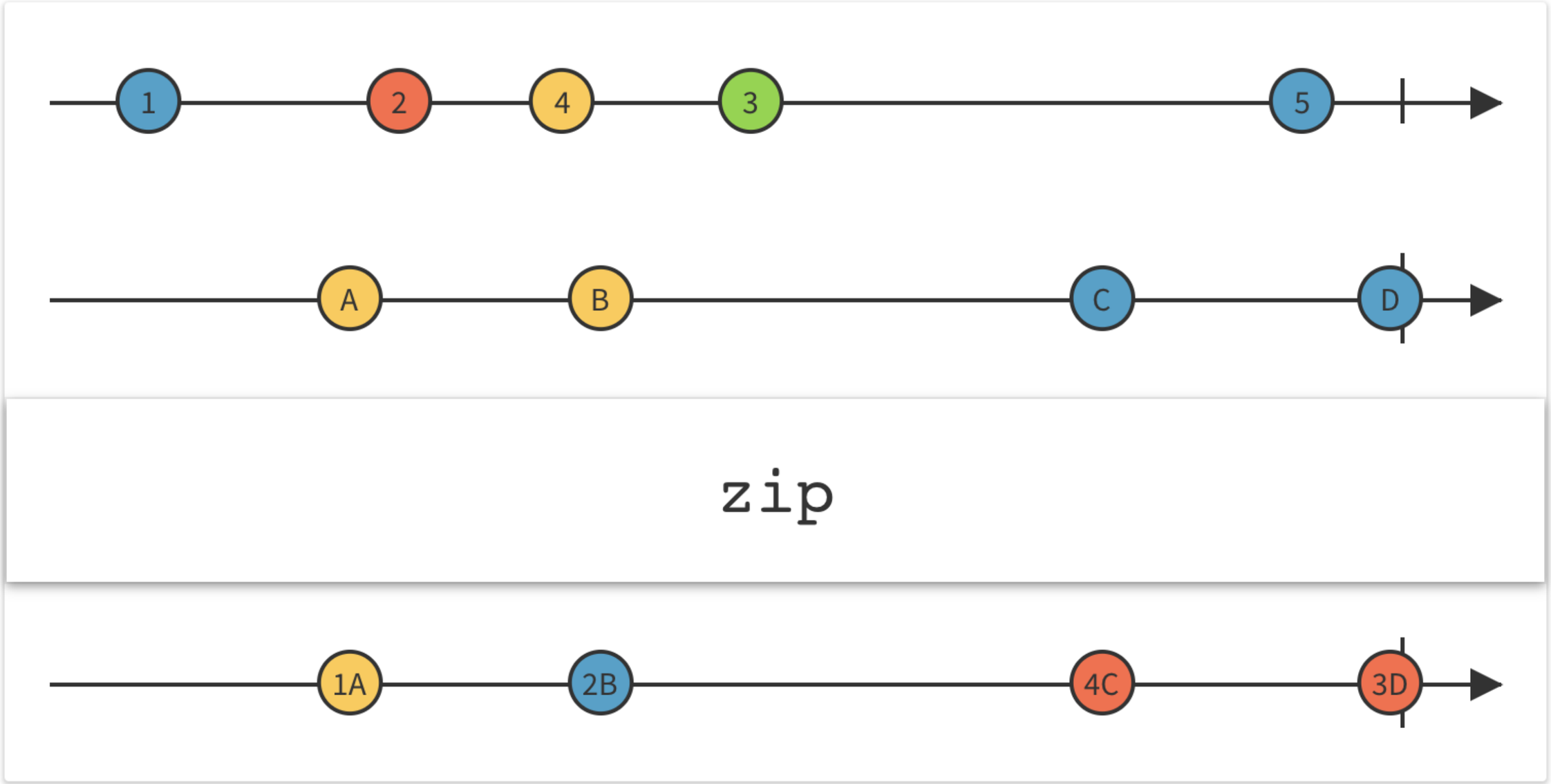
Reactive programming [4]

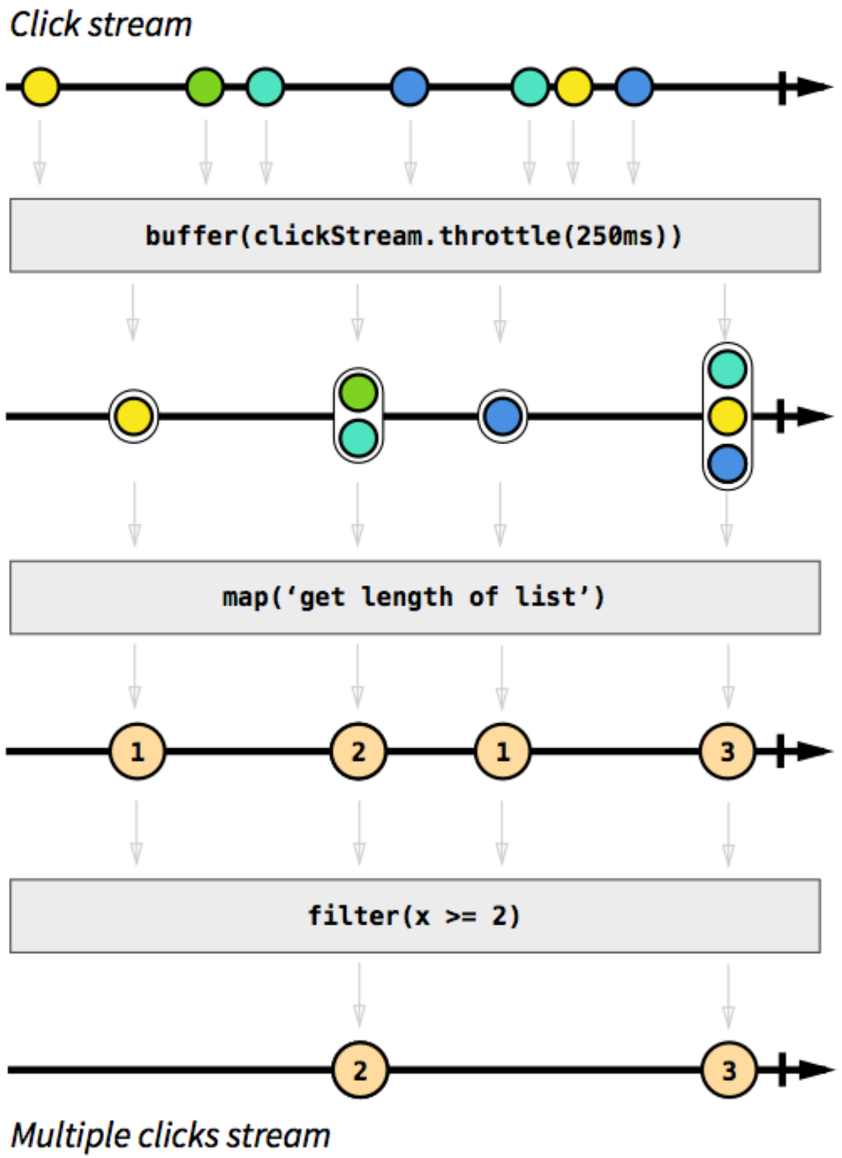
[4] Functional Reactive Animation 1997

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


1
2
3
map(x => 10 * x)
10
20
30
0
1
2
3
4
filter(x => x % 2 === 1)
1
3


a
b
c
d
e
f
merge
a
d
b
e
c
f


1
2
4
3
5
A
B
C
D
zip
1A
2B
4C
3D


Click stream
buffer(clickStream.throttle(250ms))
map('get length of list')
1
2
1
3
filter(x >= 2)
2
3
Multiple clicks stream

Monolith	Microservice
Simplicity	Partial Deployment
Consistency	Availability
Inter-module refactoring	Preserve Modularity
	Multiple Platforms

Autonomy

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

— William Gibson