Architetture Software a Microservizi
Corso di Laurea Magistrale in Informatica

Microservices strategies and patterns

Davide Rossi
Dipartimento di Informatica – Scienze e Ingegneria
Università di Bologna

Interest in microservices


Uber
NETFLIX
GILT
ebay
zalando
Walmart
SOUNDCLOUD
amazon
GROUPON
monzo
Spotify
comcast
HAIL
Microservices success stories

From hero

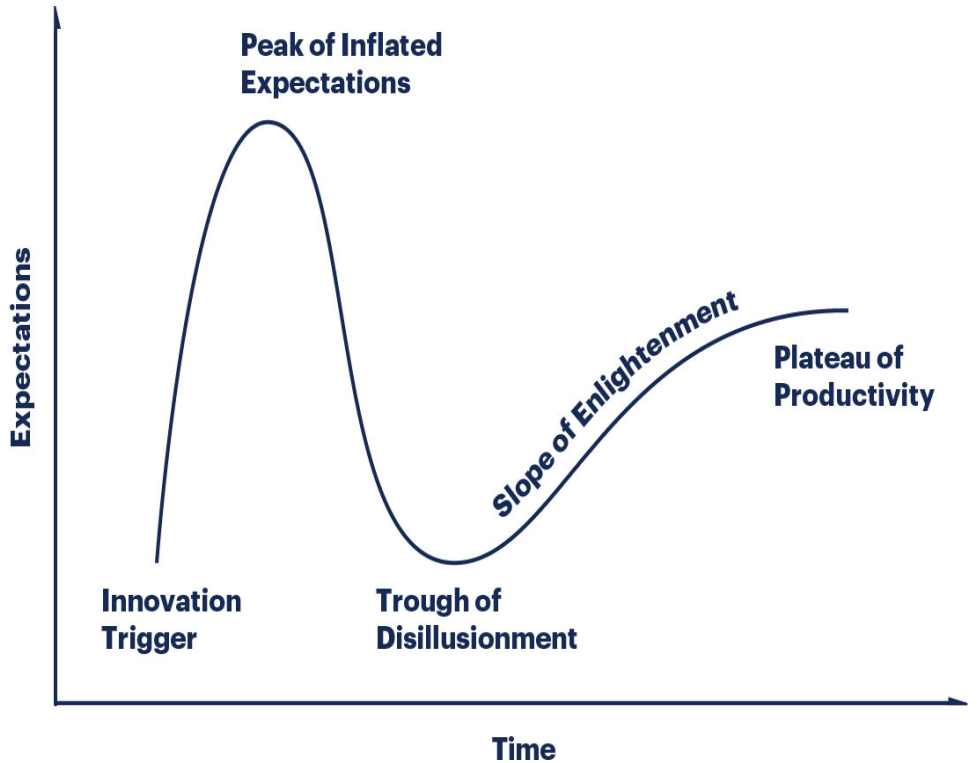

Segment
ENGINEERING
Why Microservices Work For Us
Calvin French-Owen on December 15th 2015

To zero

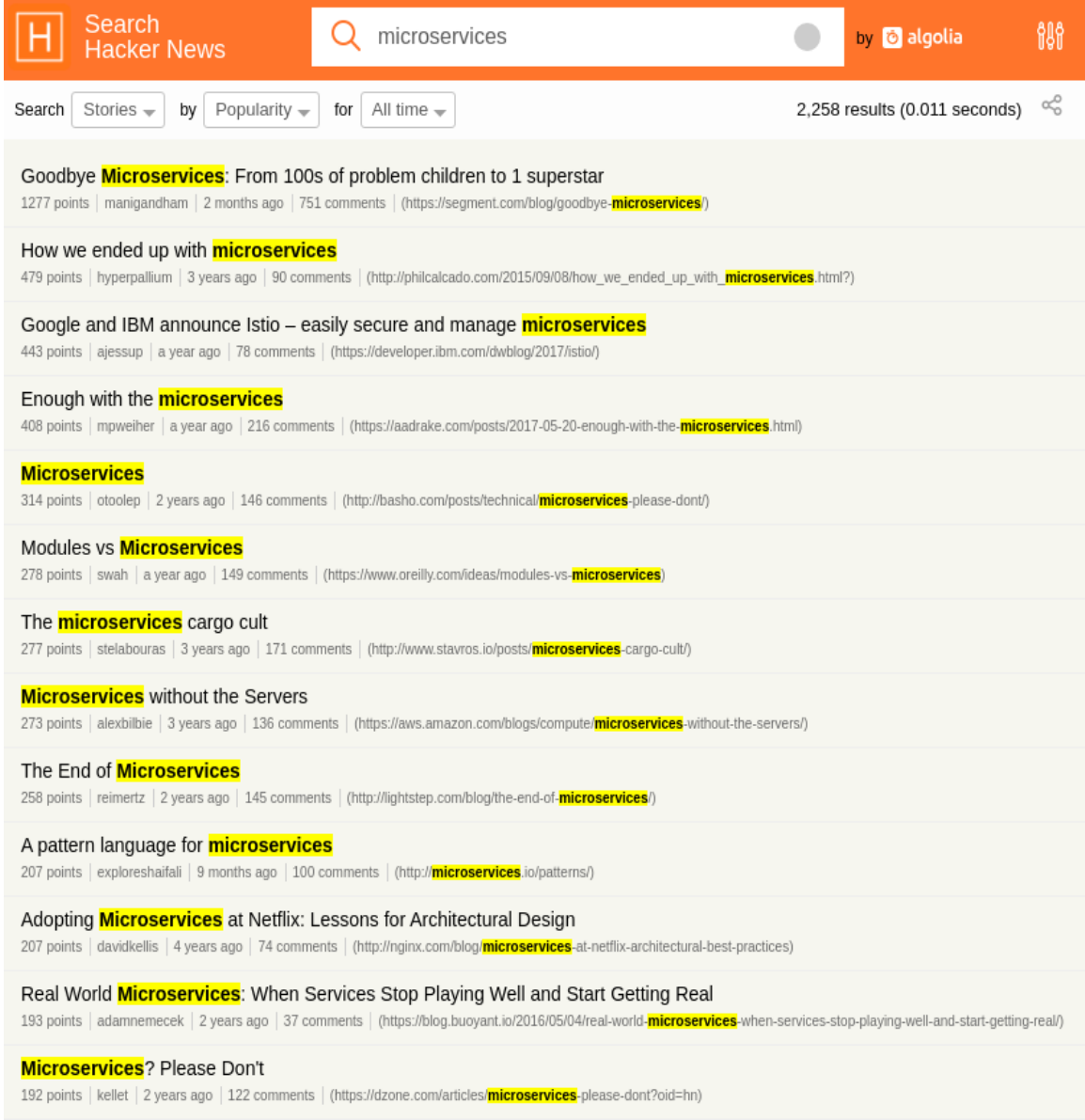

Segment
ENGINEERING
Why Microservices Work For Us
Calvin French-Owen on December 15th 2015


Segment
ENGINEERING
Goodbye Microservices: From 100s of problem children to 1 superstar
Alexandra Noonan on July 10th 2018

Living the hype cycle



Disillusionment?



Microservices are different

In all declinations of a SOA, services are loosely coupled, re-usable, autonomous, stateless, discoverable, composable, based on standards.

The difference with Microservices is that they are designed to be **developed**, **deployed** and **scaled independently**.

Enablers: agile software development; virtualization/containers; evolution in data management.

Microservices are distributed systems

Systems designed on top of a microservices architecture are distributed systems - even through the mist of containers and Vms.

We know something about distributed systems, although it seems like this knowledge does not percolate so promptly towards people adopting microservices.

One of the things we know if that **distributed systems are hard.**

A distributed system is one in which your computer can be rendered useless by the failure of a computer that you didn’t even know existed. [Leslie Lamport]

Quality dimensions for microservices

- **Consistency** and **availability** are the most exemplary contrasting non-functional requirements that large, multi-user, distributed applications struggle with.
- The **CAP theorem** (Gilbert and Lynch, 2002) states that in a partitionable system it is not possible to achieve full consistency and maximum availability.

Availability enablers in distributed systems

Replication.

Availability is hindered by **overloading** and **crashing.**

Replication is the basic mechanism for horizontal scalability and failover techniques.

The raise of eventual consistency

- The data management needs of Web 2.0 companies shifted the focus from SQL and ACID to NewSQL/NoSQL and BASE (Basically Available, Soft state, Eventual consistency, Pritchett, 2008).
- With microservices it is usual to look for trade-offs in which a price is paid in terms of consistency in order to achieve better availability → the raise of **eventual consistency**.

Safety and liveness

- A ***safety*** property is one which states that something will not happen (never!).
- A ***liveness*** property is one which states that something must happen (eventually...).

[L. Lamport 1977]

Consistency classes

- **(Strong) consistency**: every read receives the most recent write
- **Eventual consistency** (EC): if no new updates are made to the object, eventually every read receives the most recent write (Vogels, 2009).
- **Strong eventual consistency** (SEC): EC + correct replicas that have delivered the same updates have equivalent state (Shapiro at al. 2011).

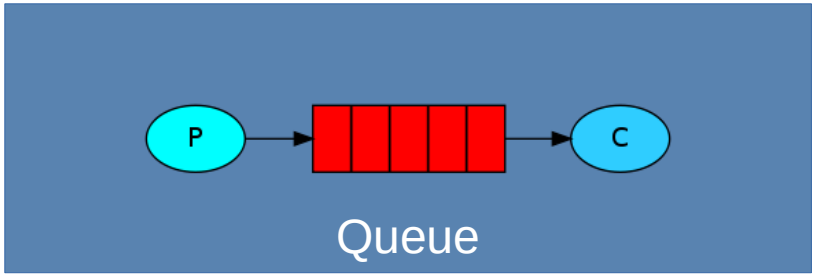
Communication strategies

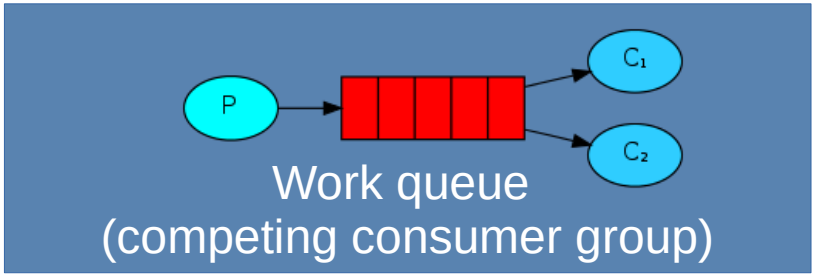
- Explicit addressing
  - Request/response (a.k.a. RPC)
    - Sync: the response uses the request’s channel
    - Async: the response uses a different channel (the caller must expose a response endpoint)
  - Streaming
- Implicit addressing (via message brokers)
  - Queues
  - Publish/subscribe channels (topics)
  - Logs

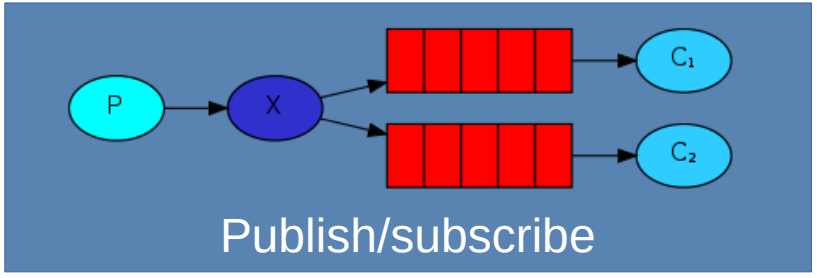
A note of warning

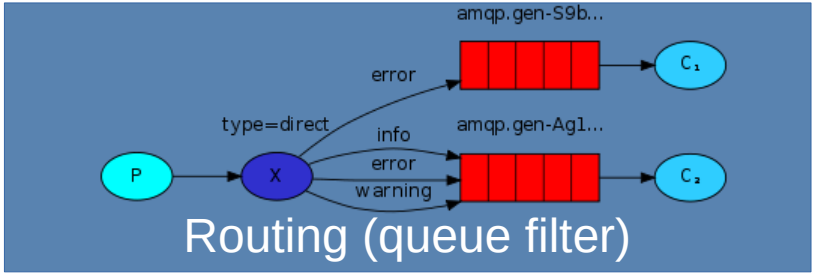
[...] *when we look in detail at the various styles of asynchronous communication, there are a lot of different, interesting ways in which you can get yourself into a lot of trouble* [Sam Newman]

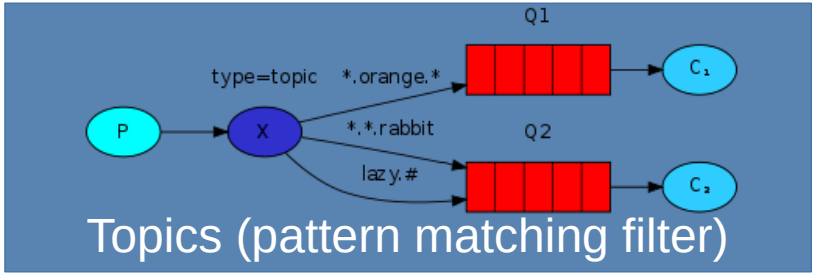
Message brokers

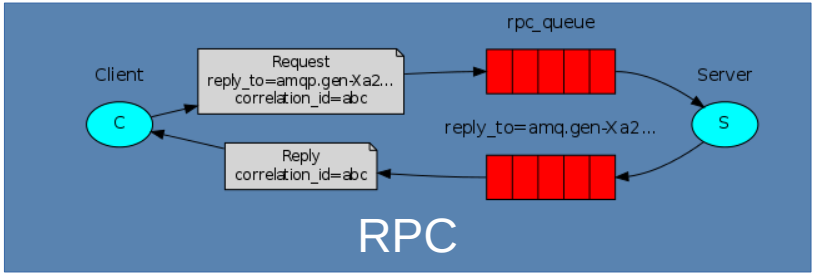












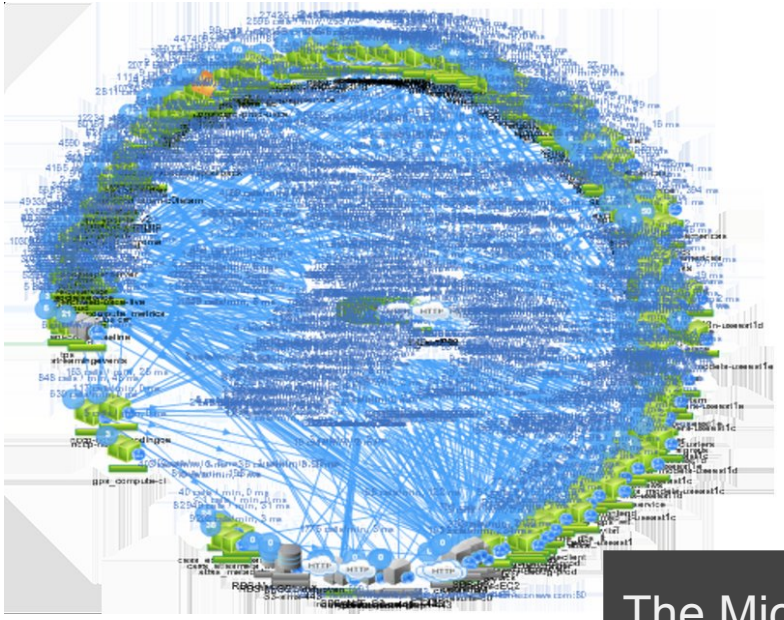
Message brokers: reliability

- Message brokers are usually transactional
  - Yet they usually explicitly discourage the use of transactions (RabbitMQ says that transactional operations cause a 250x drop in throughput).
- Other reliability measures
  - Messages acknowledgments
  - Persistent queues (including logs)
  - Replicated queues

Message brokers: message reliability

- Basic guarantee: **at most once**.
- **Consumer acknowledgments**: the broker holds a copy of a message after dispatching it until an ack is received. Unack'd messages are re-queued when the consumer connection closes. Because of possible broker crashes (and unprocessed ack messages) messages can be delivered more than once → **at least once**.
- **Publisher confirms**: a confirm message is sent to the publisher when the message is *accepted* in the queue (made durable for persistent queues, stored by a quorum replicas for replicated queues, …).

Explicit addressing: reliability



The Microservices Death Star

Explicit addressing: reliability

- If the average chain size is **N** and the average availability of each service is **A**, the average availability of the system will be $\mathbf{A^N}$.
- For example, if the average availability for the services is 99.999% (also known as *five-nines*, a measure usually perceived as very good for a real-world system) and the average chain length is 5, the resulting availability will be 99.995%. That means an increase in downtime from 5 minutes 15 seconds per year to 26 minutes 17 seconds per year.

Explicit addressing: reliability

- In practical terms, it’s much worse than that.
- Each communication path can be seen as a potential carrier for the spreading of malfunctions.
- With no explicit **mitigation strategies** these kind of systems are highly unstable: a local failure can rapidly cause global failures.

Moving away from direct addressing

- Exploit local state replicas synchronized via events – fits naturally with Domain-Driven Design (DDD).
- When a microservice’s local state changes, it emits a *domain event.*
- Interested microservices update their local state replica whenever an event is notified.

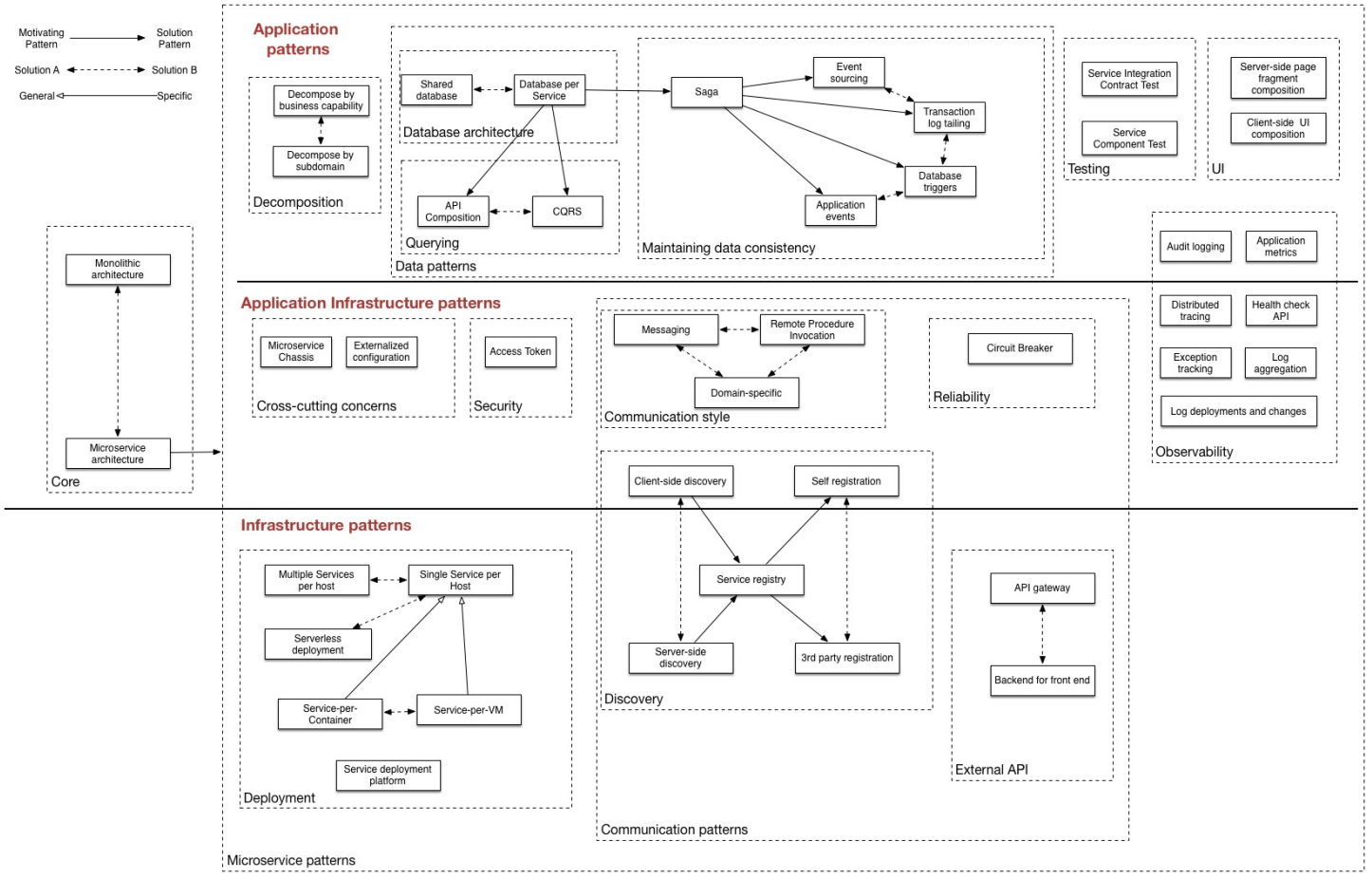
Moving away from direct addressing

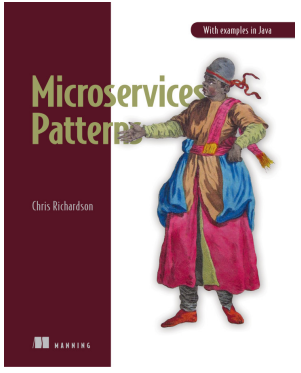
- Example: we want to the product page of an e-shop to also show whether the product is in stock.
- (At least) two microservices involved: `products` and `wharehouse`.
- Whenever the stock availability of a product changes, `wharehouse` emits an event.
- `products` listen to these events to update a local (potentially in-memory) availability table.

Event-driven architectural style

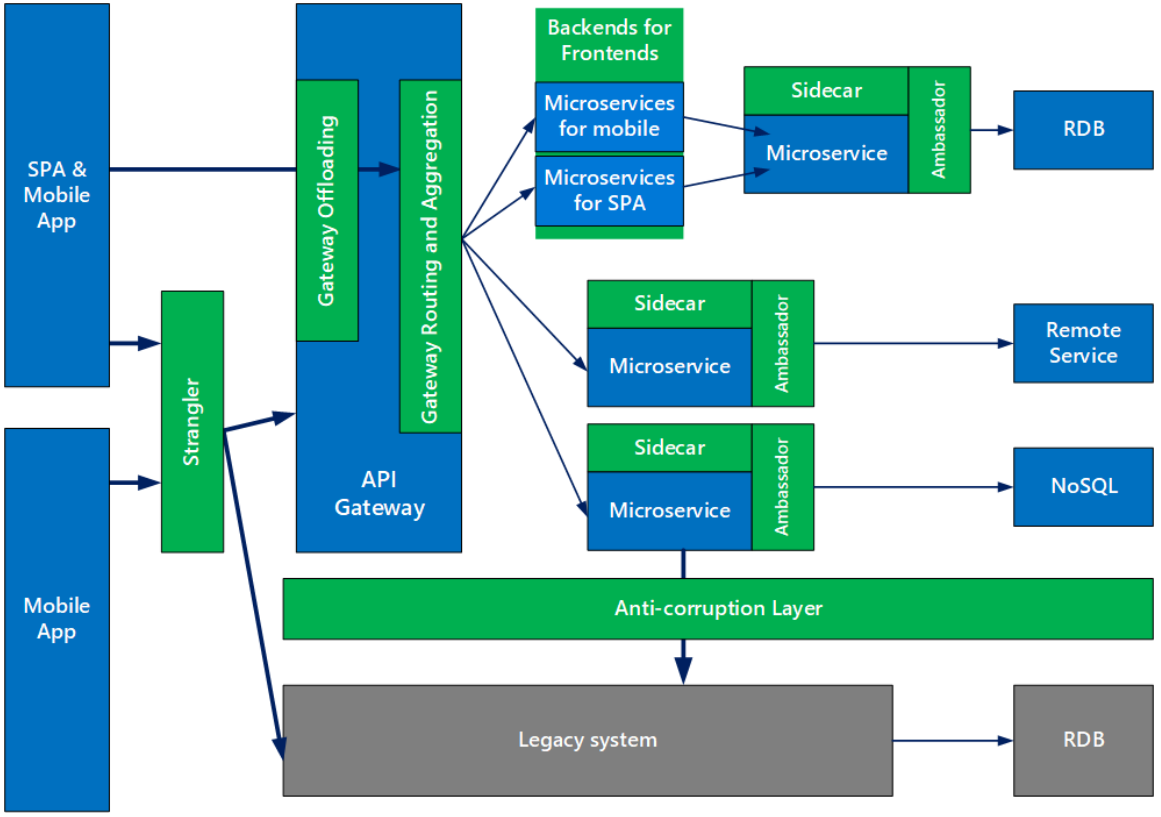
- The main components are event **producers** and event **consumers**, which are decoupled (producers are unaware of consumers and vice versa)
- **Pub/sub model**: consumers register (subscribe) with a message infrastructure. Generated (published) events are dispatched to registered consumers. Subscribers only receive messages published after their subscription
- **Event streaming model**: generated events are persisted in a log where they are stored in order (typically withing a partition). Clients can process the events by retrieving them from the stream and are responsible for positioning (which means that they can replay events generated at any time)

Microservices patterns





Microservices patterns



Microservices patterns and practices

- API Gateway
- Fail fast (and recover/and retry)
- Circuit breaker
- Bulkhead
- Sidecar (service mesh)
- CQRS (and event sourcing)
- Long-running compensating transactions (D-SAGAs)

API Gateway

An API Gateway is a front component for a microservices architecture. It is responsible for request routing, composition, and protocol translation.

Duties:

- Facade + authentication
- Monitoring
- Load balancing
- Caching
- Request shaping and management
- Static response handling

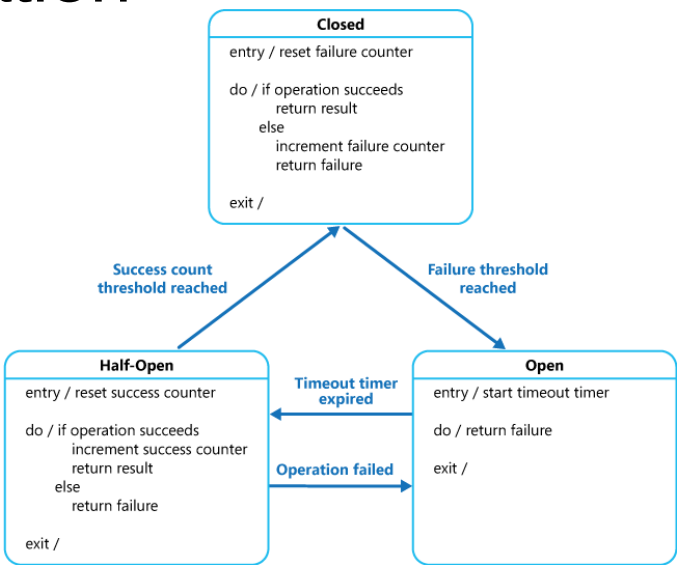
Fail fast

And retry: set short timeouts for outstanding connections, on failure just retry (a few times). If you are trying to contact a replicated service, chances are that a retry will be redirected to an available replica. Also: in several cases the reason for unavailability are transient, e.g. a (virtual) network reconfiguration

And recover: when a service starts behaving erratically (unreleased resources, logical errors, …) just shut it down and restart it. If it’s a critical service it will have replicas, it’s better to deal with a short period of overloaded replicas rather than having to deal with cascading failures

Circuit breaker

Circuit breaker (Nygard, 2018) is a pattern vastly employed to improve stability and resiliency in microservices architectures in the presence of direct service-to-service invocation



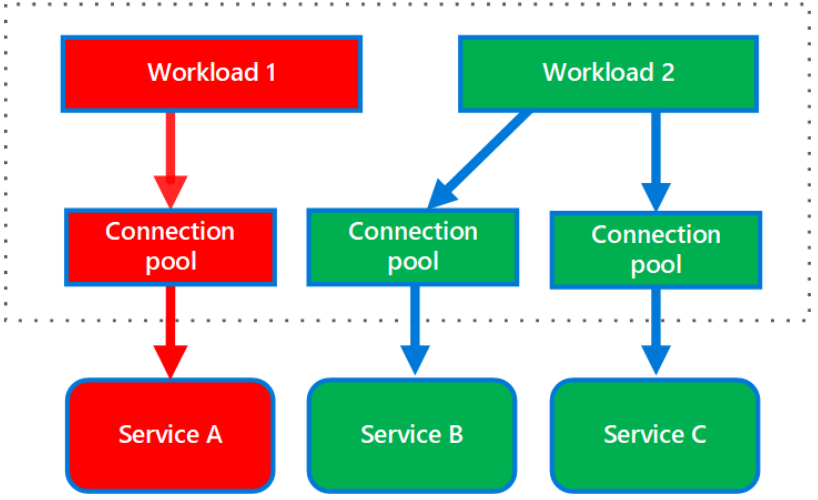
Circuit breaker: states

- **Closed** – request is forwarded to the service
- **Open** – request fails immediately
- **Half-open** – a subset of the requests are forwarded, the others fail immediately

  The Half-Open state is useful to prevent a recovering service from suddenly being flooded with requests. As a service recovers, it might be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress a flood of work can cause the service to time out or fail again.

Bulkhead

Bulkhead (Nygard, 2018) is a pattern that suggests to partition service instances into different groups, based on consumer load and availability requirements in order to avoid the risk for a troubling connection to starve other concurrent workloads.

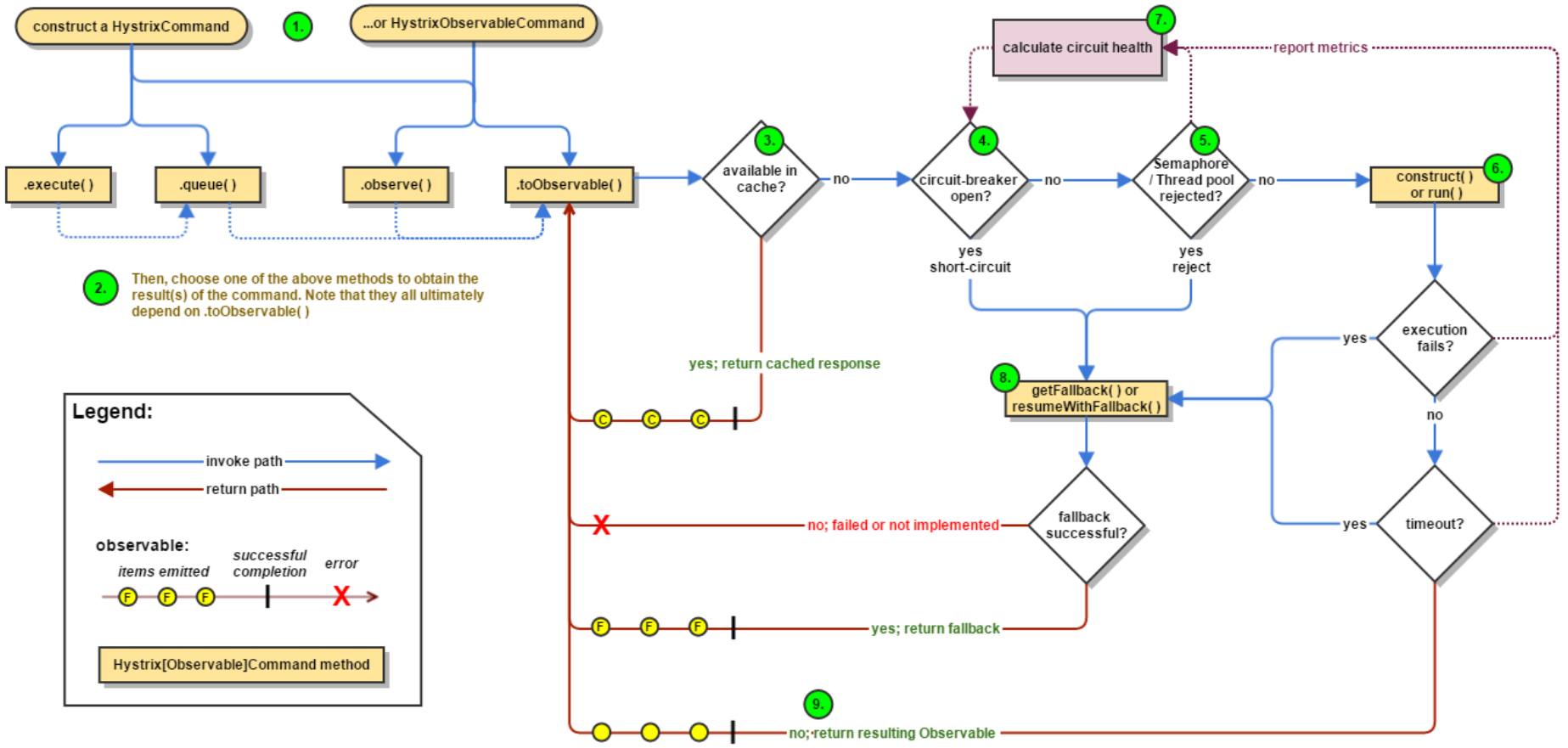


Implementation strategies

- **In-process**: e.g. libraries used consistently when implementing a microservice or a client
- **Out-of-process**: interact through an external man-in-the-middle (see the sidecar pattern)

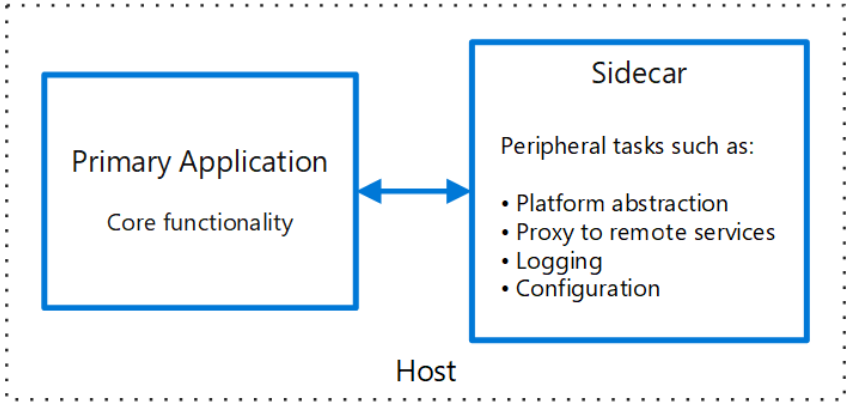
Netflix's Hystrix

HYSTRIX
DEFEND YOUR APP



Sidecar

Applications and services often require related functionality, such as monitoring, logging, configuration, and networking services. Sidecar suggests to co-locate a cohesive set of tasks with the primary application, but place them inside their own process or container to improve isolation and independence (for example w.r.t. the language used)



Service mesh

- A service mesh is a configurable infrastructure layer for a microservices application. The mesh provides service discovery, load balancing, encryption, authentication and authorization, support for the circuit breaker pattern, and other capabilities.
- A service mesh is built by associating each (micro)service to a mesh-provided sidecar.

CQRS

CQRS (Command Query Responsibility Segregation) proposes to use two different APIs, adopting distinct data models: a **command** one and a **query** one.

Commands are update operations (can be queued for asynchronous processing).

Query returns data transfer objects (DTOs), i.e. passive data containers with no domain knowledge.

CQRS can be implemented with different persistence management systems for commands and queries. This allows for command and query API to scale independently: in this case the (replicated) query data stores have to be (eventually) reconciled with the command data store.

CQRS and event sourcing

In event sourcing the data store does not contains the latest version of the data but rather the sequence of generated update events.

CQRS naturally fits in this scenario: the same events that are stored in the command data store are used to synchronize the query data stores.

At bootstrap, or in case of failures, query data stores can re-generate local consistent copies by *replaying* all events from the beginning (or from snapshots generated at scheduled intervals).

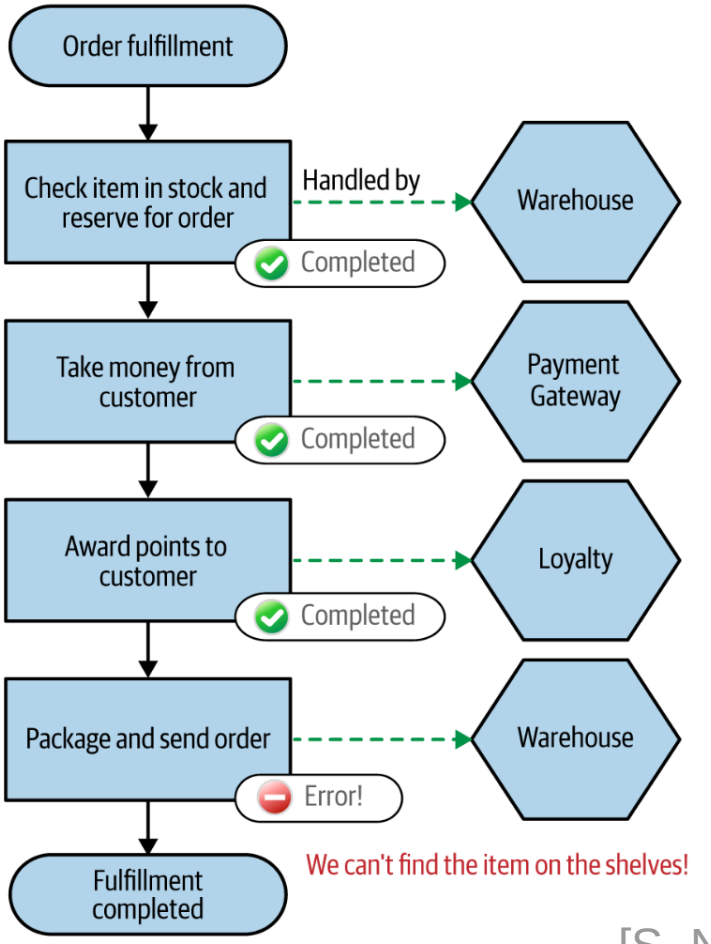
LRCP

- Microservices-based solutions, for the most part, adopted ad hoc solutions, also known as feral concurrency control (Bailis at al., 2015).
- Long-running compensating transactions (a.k.a. distributed SAGAs) are a viable alternative.

D-SAGAs

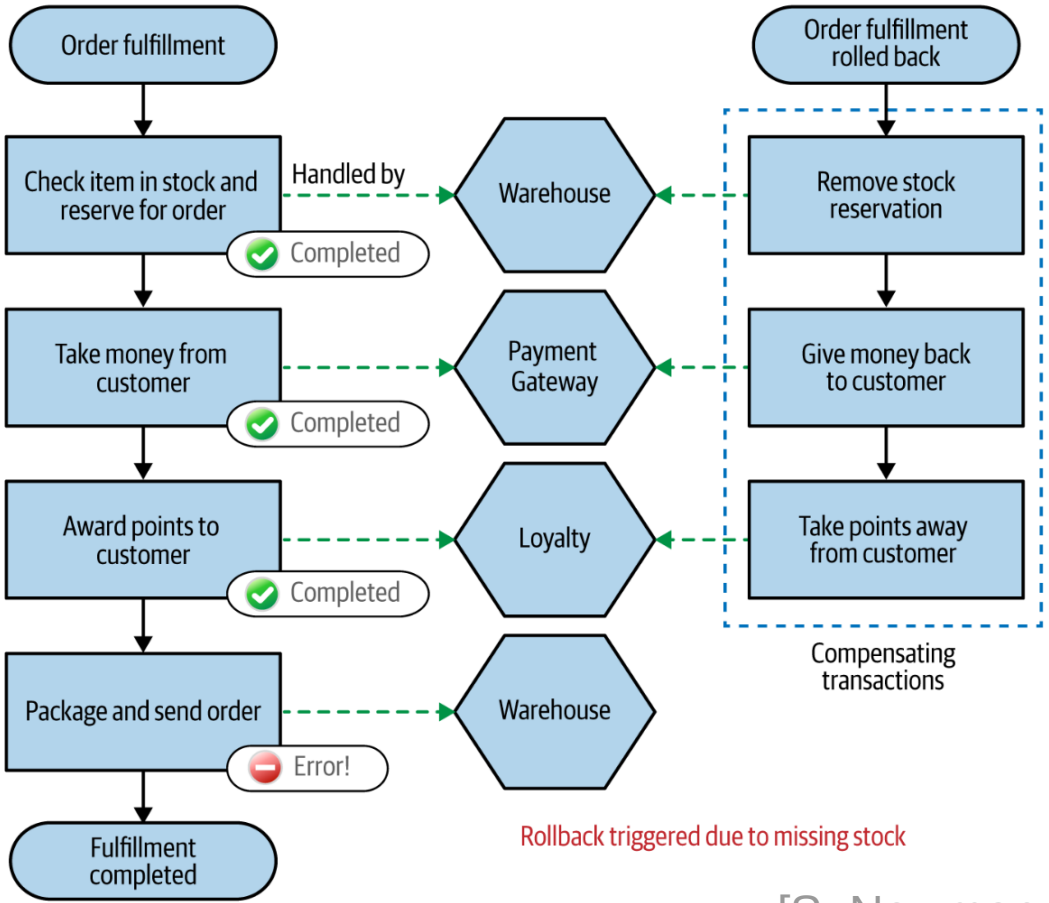
- A complete transaction is split in smaller sub-transactions (implemented as services' operations).
- When a transaction fails, a *backward recovery* or a *forward recovery* can take place.
- Backward recoveries need compensating transactions to be defined and invoked for all sub-transactions that completed successfully before the failure.

D-SAGAs



[S. Newman - Building Microservices]

D-SAGAs - compensations

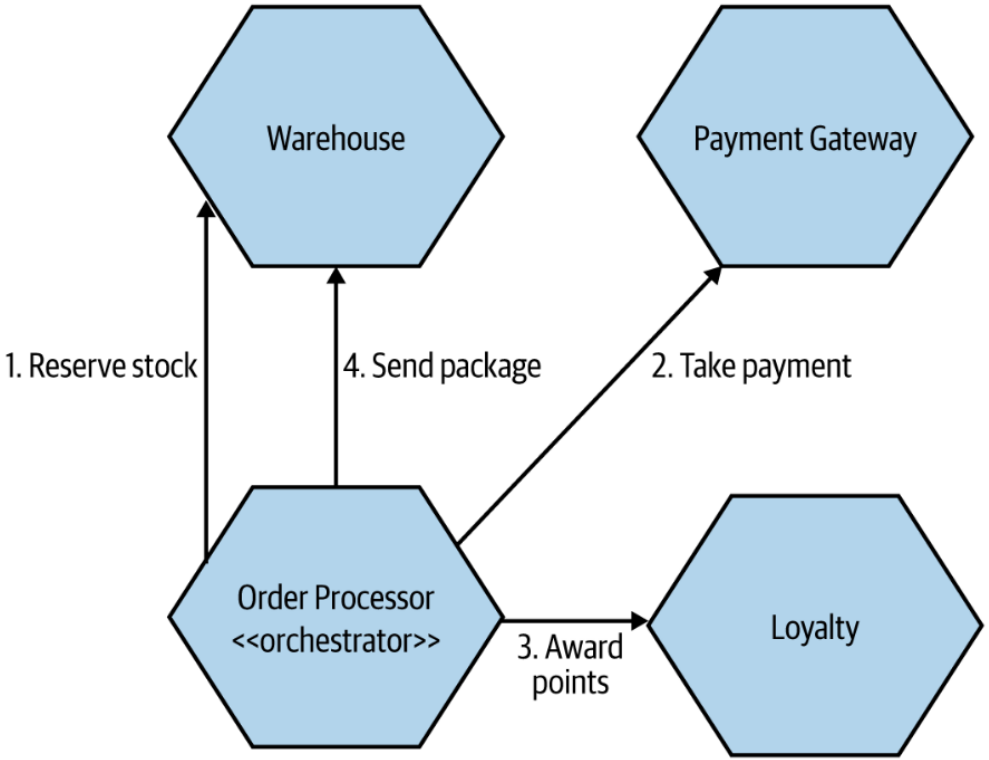


[S. Newman - Building Microservices]

Implementing D-SAGAs

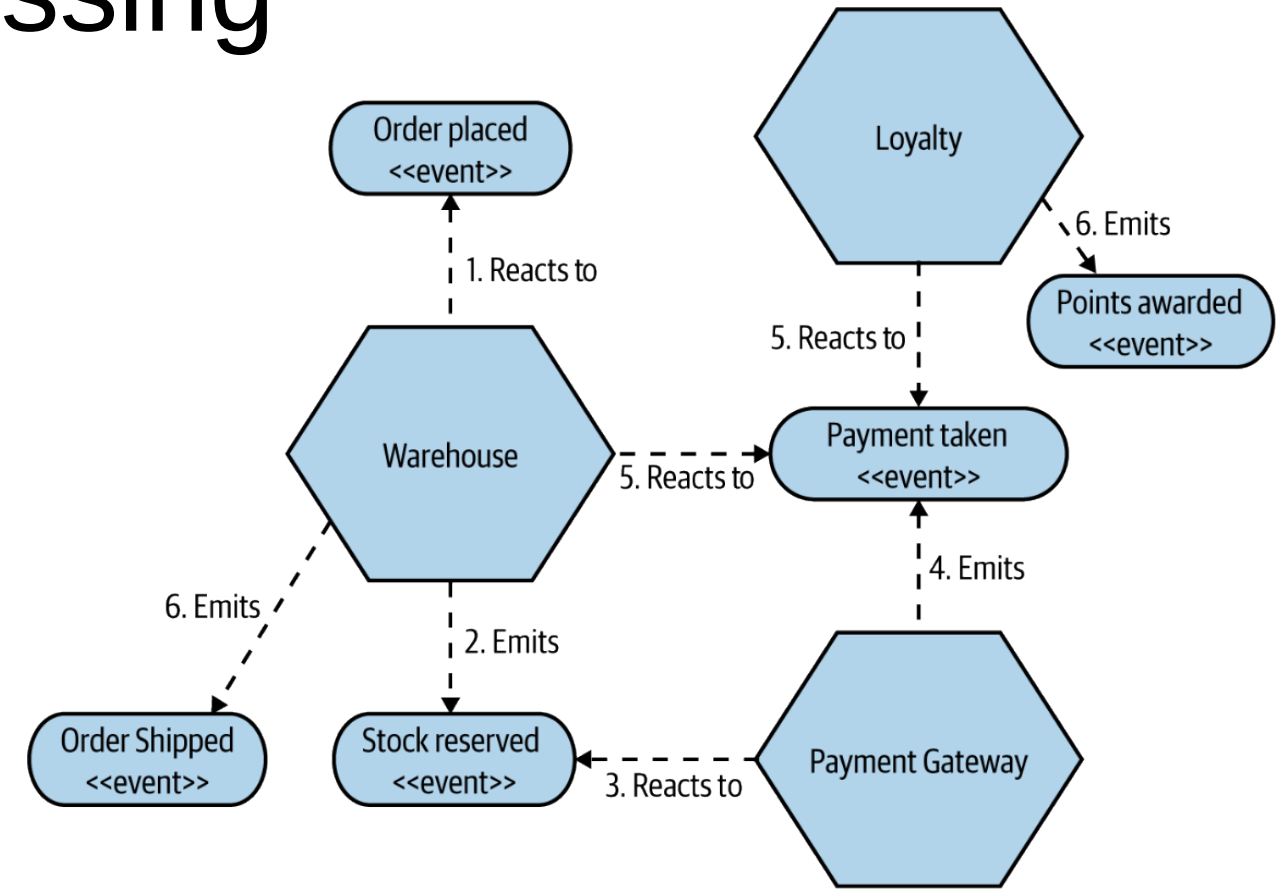
- Orchestration approach
  A centralized coordinator invokes the transactional operations, decides the recovery strategy in case of a failure and implements it.
- Choreography approach
  Each transactional operations is aware of the transaction and the coordination is distributed. Both explicit and implicit addressing can be used to make the services cooperate.

D-SAGA orchestration



[S. Newman - Building Microservices]

D-SAGA choreography with implicit addressing



[S. Newman - Building Microservices]