#RedHatOSD

RED HAT FORUMS

GESTIONE DELLE TRANSAZIONI IN ARCHITETTURE DISTRIBUITE

Mauro Vocale — Solution Architect
Giovanni Marigi — Principal Consultant

Red Hat

```
apiVersion: redhat/v2
kind: PrincipalConsultant
metadata:
  name: Giovanni Marigi
  namespace: ITI
  websites:
    linkedin: giovannimarigi
    github: hifly81
    twitter: hiflyisflying
  annotations:
     specialist: openshift, cloud, integration, kafka, bpm
  labels:
     sports: mountain bike, trekking
spec:
  replicas: 1
  containers:
        - image: redhat.io/giovanni:latest
```

Microservices

The microservice architecture is the main trend in information technology and we learned a lot about it during these years ...

What are Microservices?

"...an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."

Martin Fowler

MICROSERVICES ADVANTAGES

- Fast to develop, easier to maintain and understand
- Starts faster, speeds up deployments
- Fault isolation
- Services can be scaled independently
- Deltas and patches can be applied to each microservice individually
- Local changes can be deployed easily
- Flexible choice of technology
- Security can be applied to each microservice as opposed to the whole system in a blanket approach

MICROSERVICES DISADVANTAGES

- Additional **architectural complexity** of distributed systems
  - Maintaining strong consistency is extremely difficult
  - Testing a distributed system is difficult
  - Requires a shift in coding paradigm:

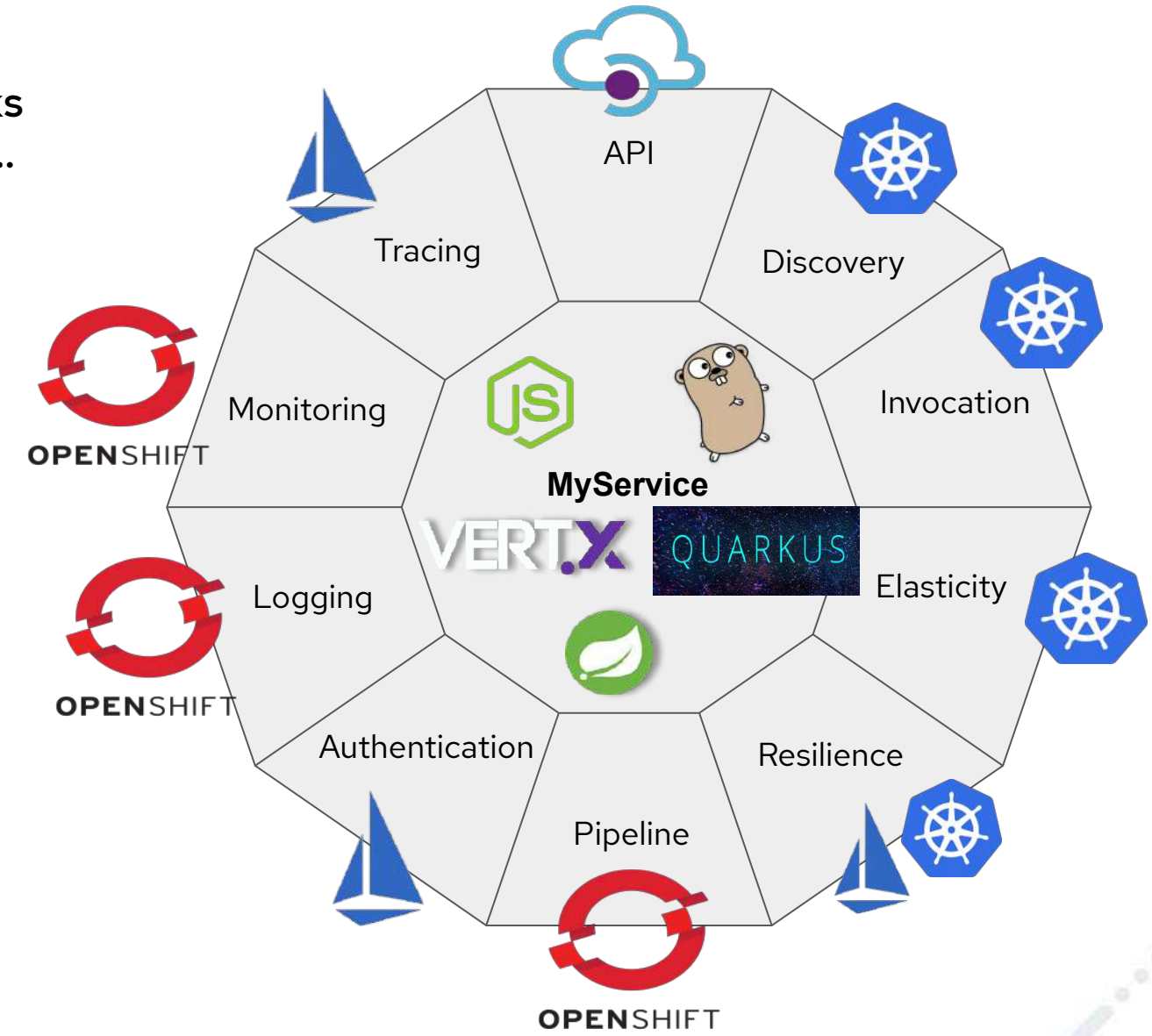
    Change in approach to application architecture design and testing
- Significant **operational complexity**. Requires a high degree of automation
  - Deployments require coordination and rollout plan

MICROSERVICES FRAMEWORKS

There are many ways to build a microservice

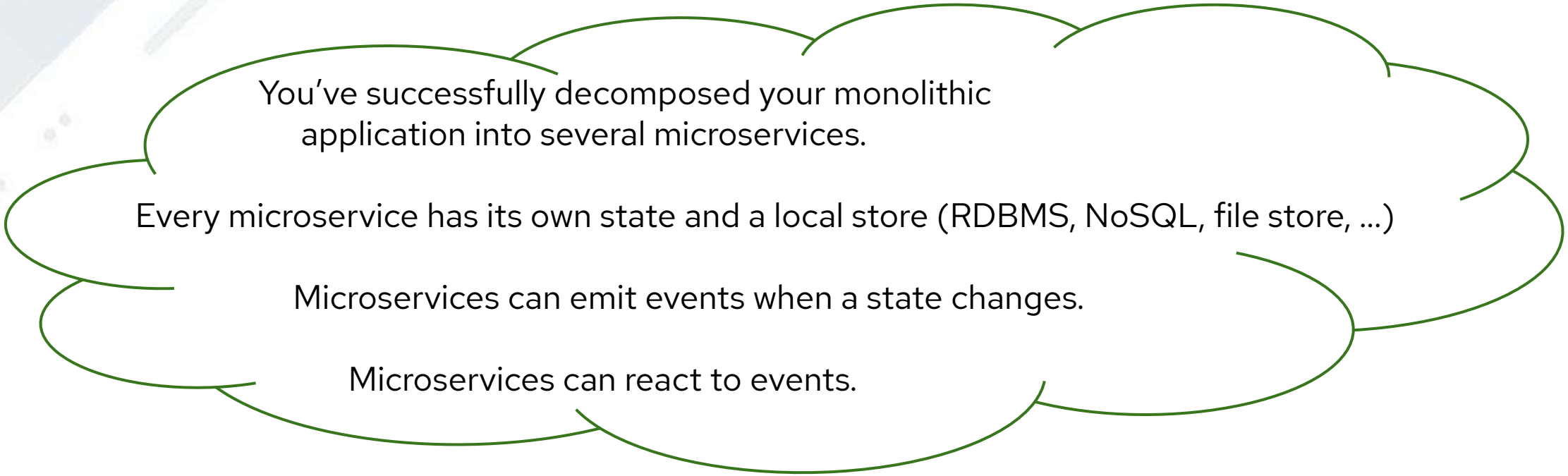
Microservices'ilities

Platform and frameworks cover a lot ... but not all ...



MICROSERVICES DILEMMA: How to handle transactions?

Microservices and transactions



But you still want that a business process spanning several services to be consistent and correct regardless of the level of decomposition of your application.

Why distributed transactions don’t work

Traditional distributed transactions are implemented using a two-phase commit, briefly 2PC.

Why can’t we use 2PC in microservices architecture?

- You don’t have a single shared store anymore, every service has its own data store (micro-db)
- A microservices architecture involves many parties, realized using different technologies that adhere to different specifications: 2PC implicitly assumes closely coupled environment
- Synchronization and isolation reduces performance and scalability.
- A business function potentially lasting for hours or days: lock strategy doesn't work well in long duration activities

Microservices: eventual consistency

Consistency states that the entire software system should be in a valid state.

2PC guarantees the consistency with a pessimistic approach; all the changes must be done at the same time or rollback.

Microservices architectures at the opposite guarantee the consistency with a more relaxed approach. The state of the entire system can't be valid at any time but at the end of the business transaction. The system is eventually consistent.

Eventual consistency is not easy achieve and there is no a magic box out there.

Saga Pattern

Le Radeau de la Méduse -
Théodore Géricault, 1818-19

Saga Pattern: overview

Saga is the *de facto* solution to guarantee consistency in a microservices architecture.

Saga is not a new pattern [1] and it can be applied also in traditional monolithic architectures.
"*For specific applications, it may be possible to alleviate the problems by relaxing the requirement that an LLT be executed as an atomic action. In other words, without sacrificing the consistency of the database, it may be possible for certain LLTs to release their resources before they complete, thus permitting other waiting transactions to proceed*"

Saga usually performs better than distributed transactions and doesn't require all the services to be available at the same time.

Developers need to take care of the implementation of the Saga.

[1]
https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

Transaction ACID Properties

- **Atomicity**: this property guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely
- **Consistency**: this property is related to the logical consistency of the data. When a new transaction starts, it must ensure that the data maintains a state of logical consistency, regardless of the final outcome
- **Isolation**: this property ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially
- **Durability**: this property guarantees that all of the changes made during a transaction, once it is committed, it must be persistent and definitive, even in the case of system crashes

Saga Pattern: ACD

Saga is a series of local transactions; every local transaction happens within the boundary of the (micro)-service.

Saga has ACD characteristics, distributed transactions ACID characteristics [1]; the real challenge is to deal with the lack of isolation (I) in an elegant and effective way.

A transaction in a microservice architecture should be eventually consistent.

Compensations are the actions to apply when a failure happens to leave the system in a consistent state.

Compensations actions must be idempotent; they might be called more than once.

[1] https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.4.0/product-overview/acid.html

Saga Pattern: the lack of I

Lack of isolation causes:

- Dirty reads: *a transaction read data from a row that is currently modified by another running transaction*
- Lost updates: *two different transactions trying to update the same "data". One of them doesn't see the new value when trying to update*
- Non-repeatable reads: *re-reads of the same record (during an inflight transaction) don't produce the same results*

A Saga should take actions to minimize the impact of lack of isolation.

How you implement this set of countermeasures (against isolation anomalies) determine how good is a microservice.

Several techniques available: semantic lock, design commutative operations, ...

Saga Pattern: semantic lock

*_PENDING states, saved into the local microservice store, indicate that a Saga instance is in progress and it is manipulating some data needing an isolation level (for example a customer's account)

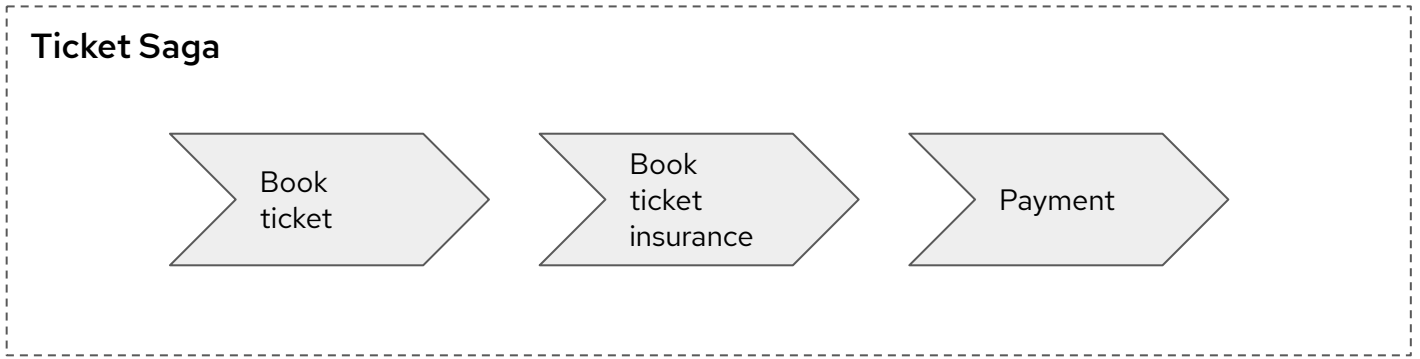
If another Saga instance starts, it must evaluate the existing *_PENDING states and pay attention on them.

Some strategies when detecting PENDING states:

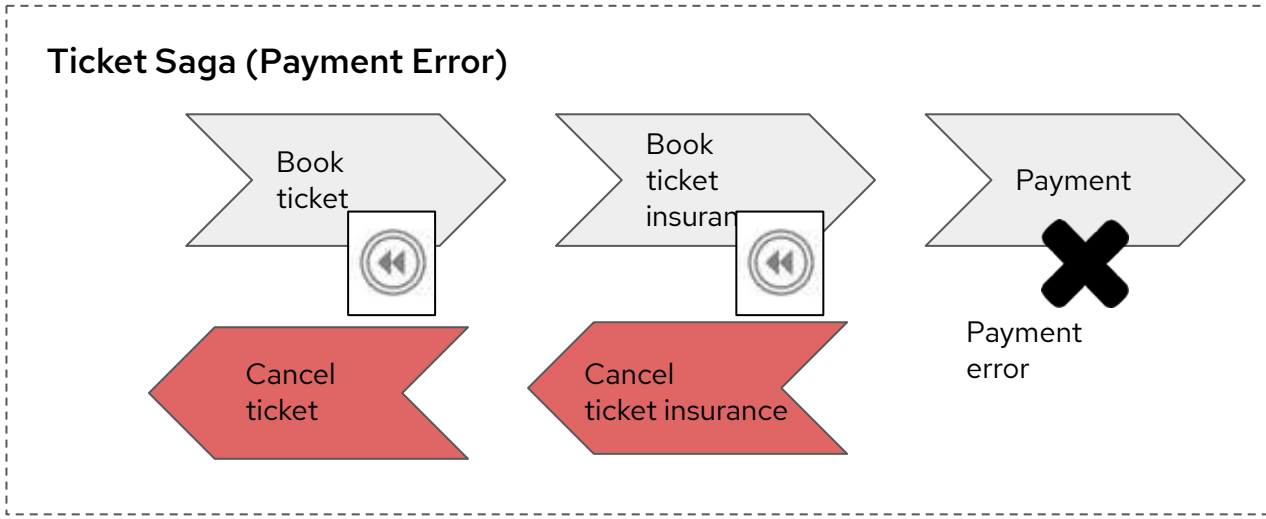
- The Saga instance will fail.
- The Saga instance will block until the lock is released.

Saga Pattern

book a ticket



compensation



book a ticket

Saga Choreography vs Orchestration

How to implement a Saga and how to coordinate the execution of local transactions within a microservice?

Two approaches:

- **Choreography**: the (micro)-service is responsible for emitting events at the end of its local transaction. The event triggers the start of new local transactions in (micro)-services subscribed to this event.
  The (micro)-service must provide the logic to compensate.

- **Orchestration**: there is a central coordinator (a stateful entity) that triggers the local transactions in (micro)-services. The coordinator has the logic to compensate and maintain the status of the global transaction.

Saga Choreography

Every (micro)-service is responsible for sending events and subscribing to **events**.
It must define a strategy to handle the events.

Decentralized approach, all the systems work independently and they cooperate for a common goal (without having knowledge on what is it).

Participants cannot be available during the execution of a Saga instance, no **SPOF**.

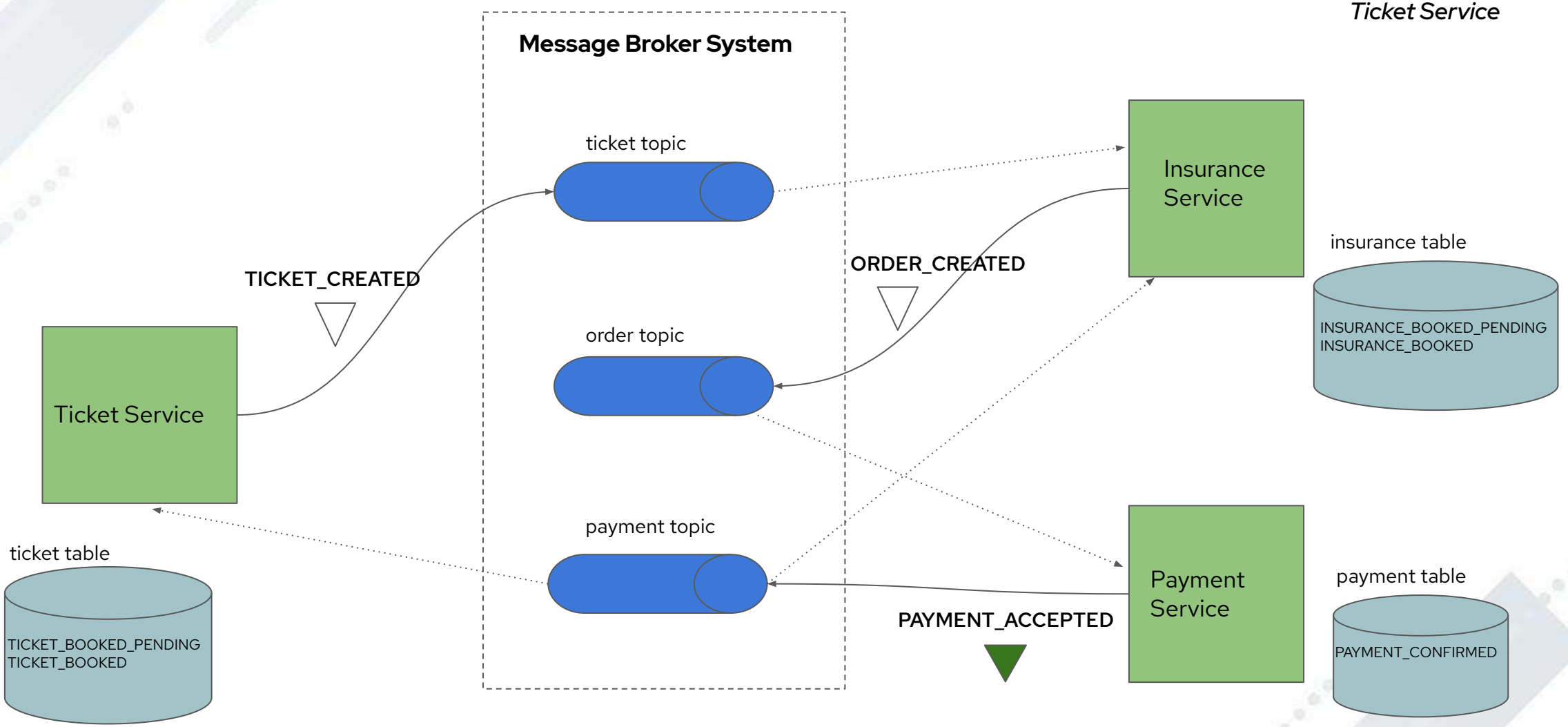
Events are sent to a **message broker system** (Apache Kafka, ActiveMQ, ...) and they contain a **correlation-id**

It works if you have a limited number of services participating in the transaction (basic sagas)

Easy to code but difficult to govern it. It's difficult to monitor and reconstruct the overall status of a Saga.

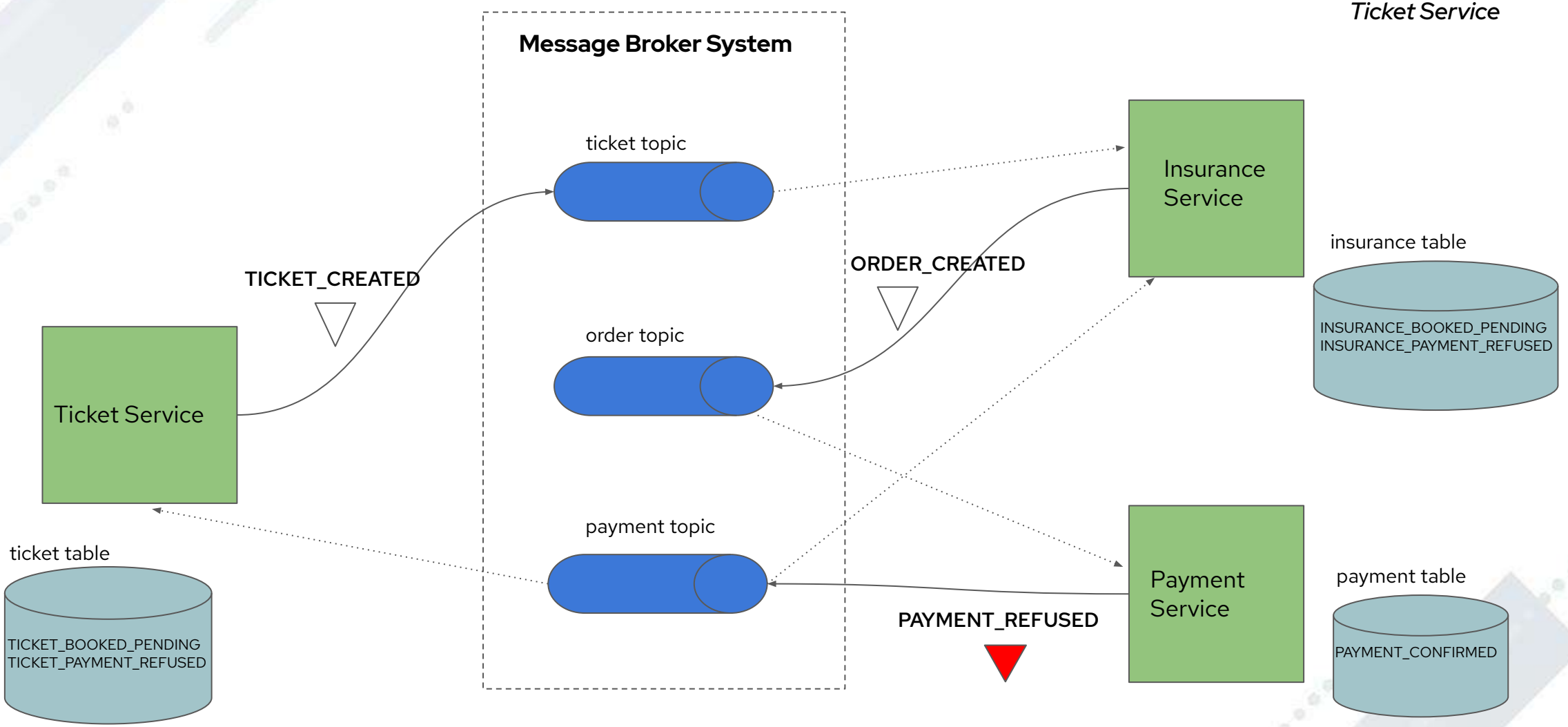
Saga Choreography

PAYMENT_ACCEPTED event must be handled by
Insurance Service
Ticket Service



Saga Choreography

PAYMENT_REFUSED event must be handled by
Insurance Service
Ticket Service



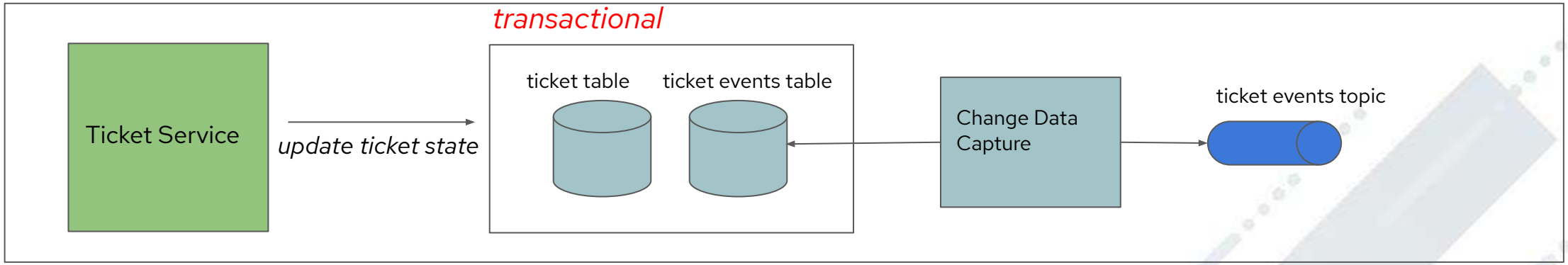
Saga Choreography

We don't want to lose any events and leave the system inconsistent.
How to atomically update the store and send the event?

One approach could be the usage of the **outbox** pattern:

- Create a database table for the events.
- Atomically update the internal microservice database and insert a record into the table for the events.

The **Change Data Capture Component** (Connector) reads the table for the events and publish the events to the message broker.
https://github.com/debezium/debezium-examples/tree/master/outbox



Saga Choreography

We don't want to lose any events and leave the system inconsistent.
How to atomically update the store and send the event?

Other approaches:

- **Event Sourcing**: services only store changing-state events using an **Event Store:** Event Store is the events database and also behaves as a message broker.
  The state of an entity is reconstructed by a service, replaying the events from the Event Store.

- **Database transaction log mining:** a process extracts the database updates using the database transaction log and publish them to the message broker.

Saga Choreography

Services must discard duplicate events.

A **message log** table can be used to track all events already processed.

The correlation-id (event-id) can be used to find the event into message log table.

Saga Choreography

Let's imagine that we don't receive a ***PAYMENT_ACCEPTED*** *or* ***PAYMENT_REFUSED*** *event.*

Are we sure that the Payment Service authorized the operation?
How long should we wait before compensating (timeout)?

The real limit of choreography is the complexity in implementing the logic to coordinate the overall business transaction.

The lack of a **Saga Coordinator** is the real limit of the Saga Choreography approach.

Saga Choreography

a custom solution with Quarkus, Debezium and Kafka

- Ticketing Service, *a java native ms with quarkus*
- Insurance Service, *a java native ms with quarkus*
- Payment Service, *a java native ms with quarkus*
- Debezium is the change data capture:
  streams events from event database to Kafka
- Debezium also sends data to Elasticsearch (Kibana)
- Apache Kafka as message broker
- Run all on OpenShift
- Apache Kafka on OpenShift using AMQ Streams
- Some Prometheus and Grafana stuff
- Some images from quay.io

DEMO

SAGA Application - Source Code