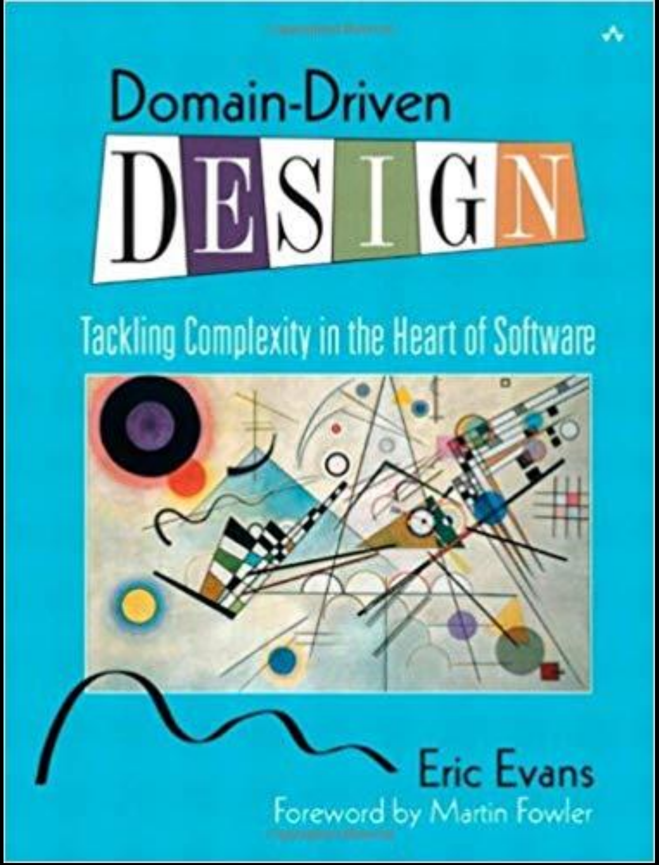
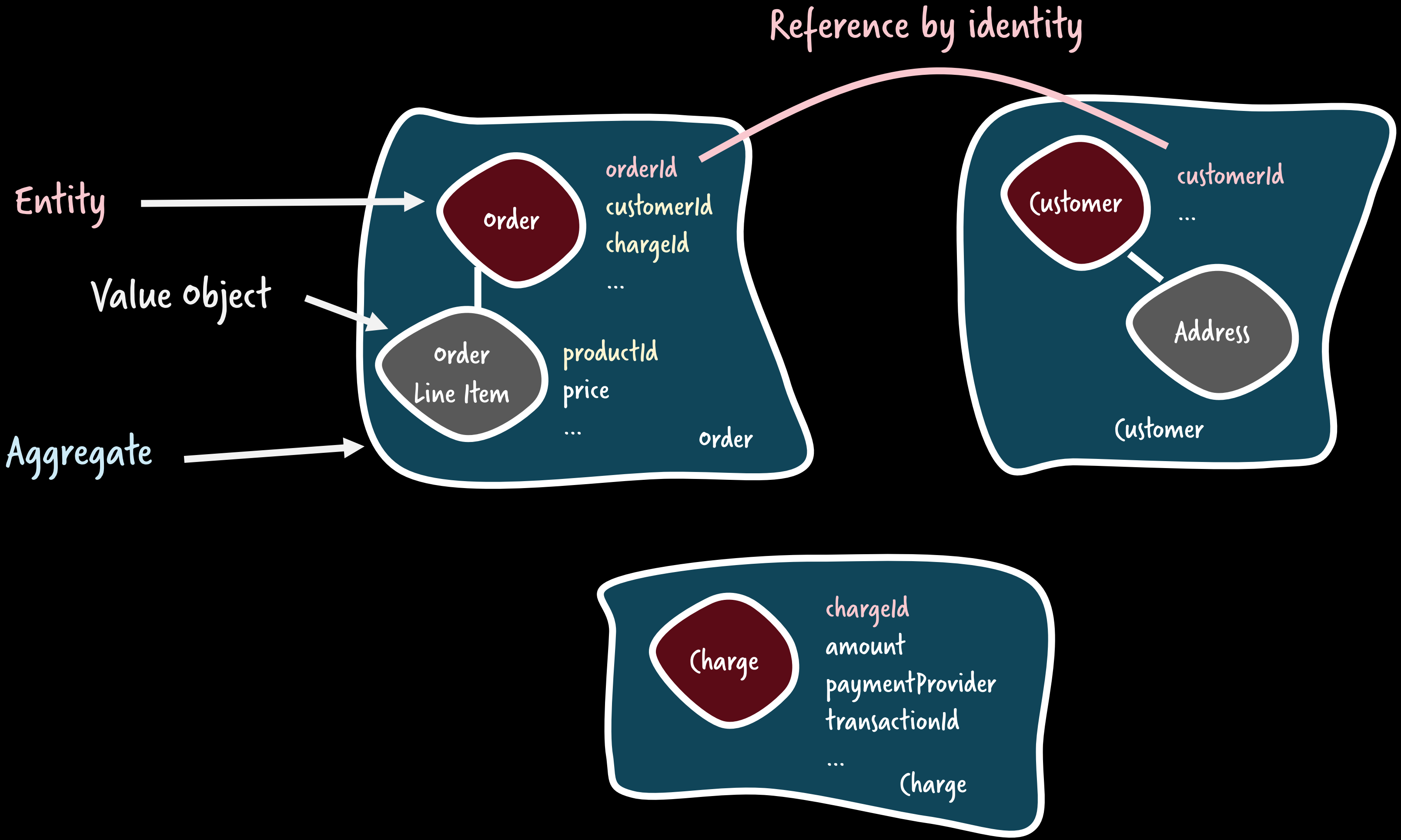
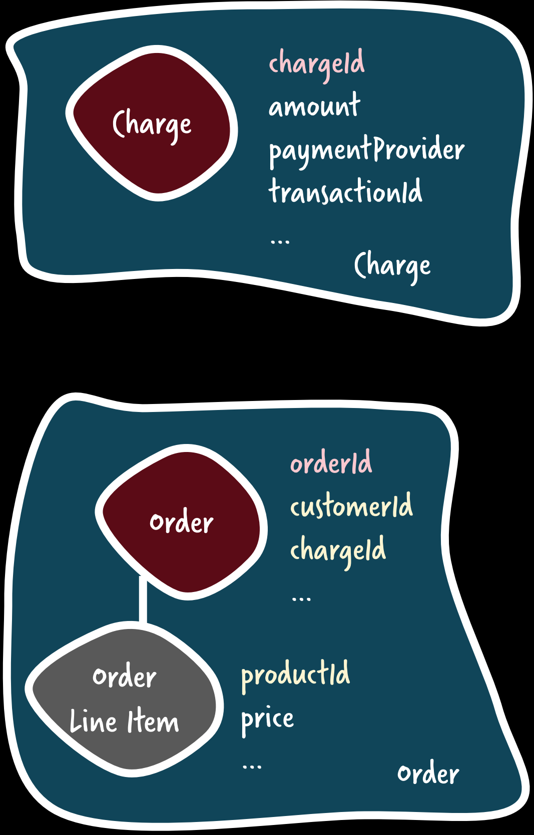
# Lost in transaction?

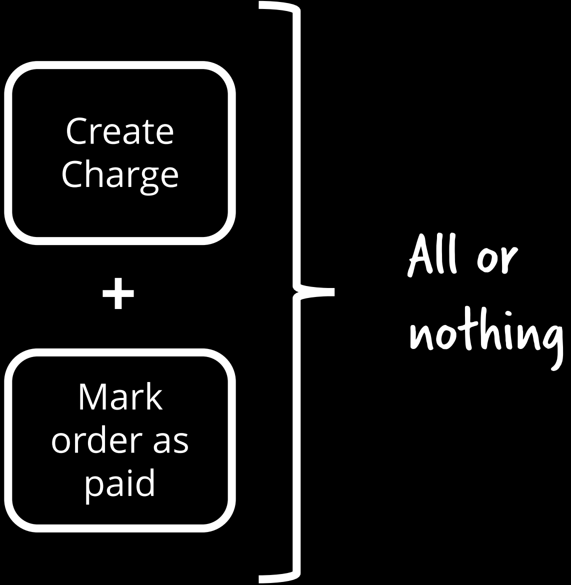
## Strategies to manage consistency in distributed systems

@berndruecker


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


Reference by identity
Entity
Value Object
Aggregate
Order
orderId
customerId
chargeId
...
Order
Line Item
productId
price
...
Order
Customer
customerId
...
Address
Customer
Charge
chargeId
amount
paymentProvider
transactionId
...
Charge




Create Charge

+

Mark order as paid

All or nothing

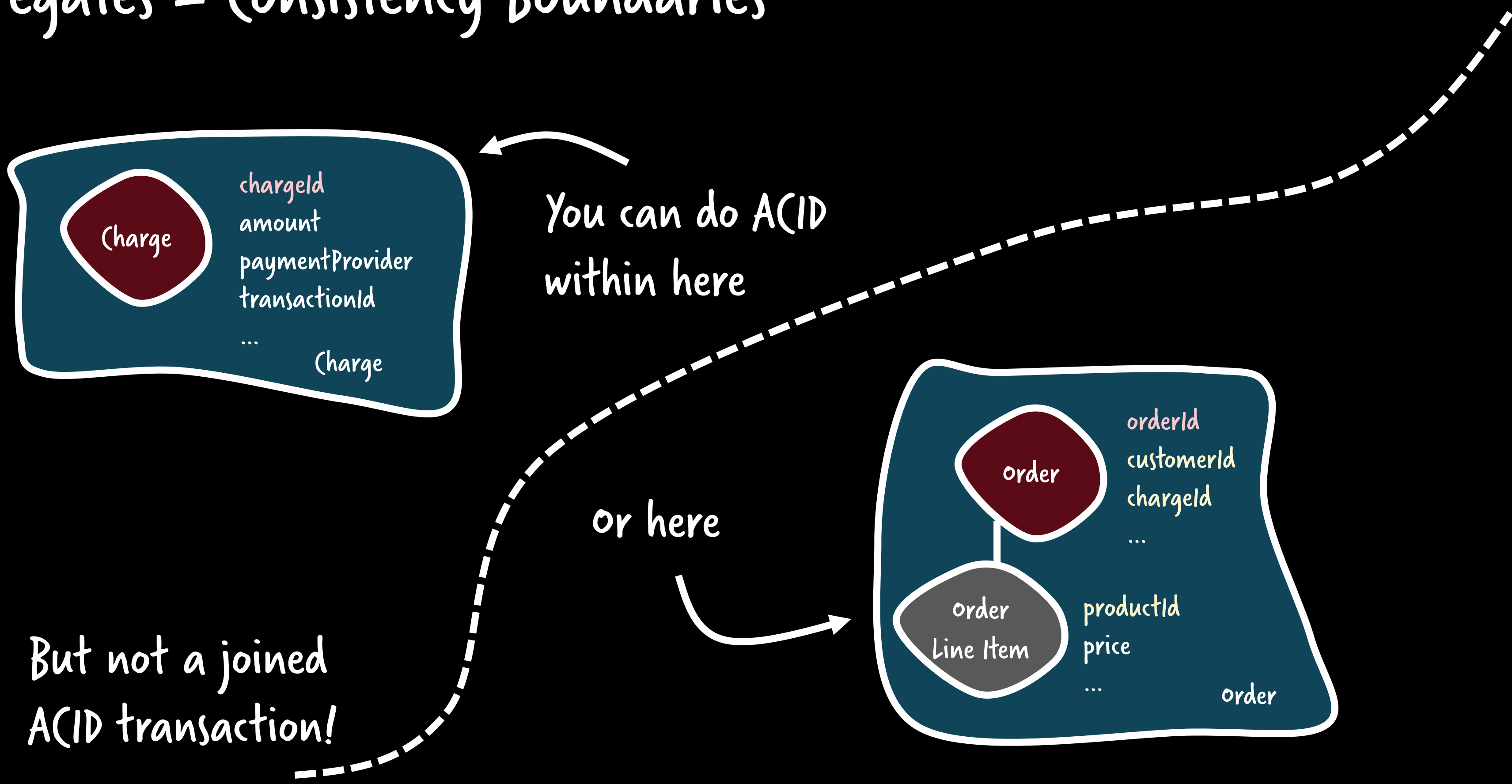
Atomicity

Consistency

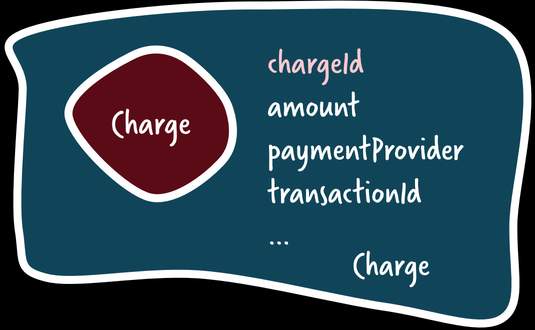
Isolation

Durability

# Aggregates = Consistency Boundaries



# Aggregates = Consistency Boundaries



Charge

chargeId
amount
paymentProvider
transactionId
...

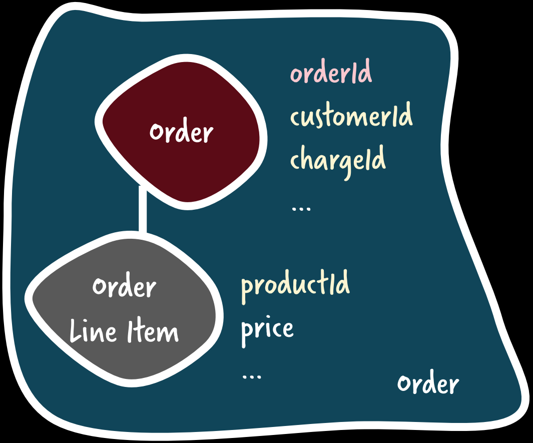
Charge

Own DB / autonomous technology decisions

API-driven

Scalability

...



Order

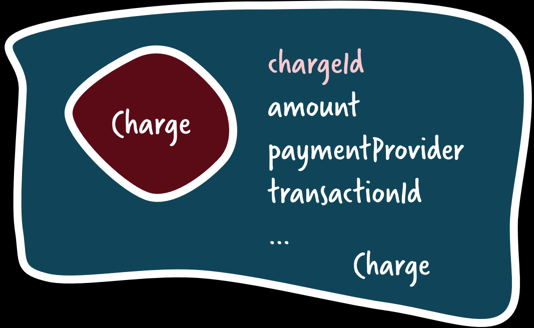
orderId
customerId
chargeId
...

Order Line Item

productId
price
...

Order

# Aggregates = Consistency Boundaries

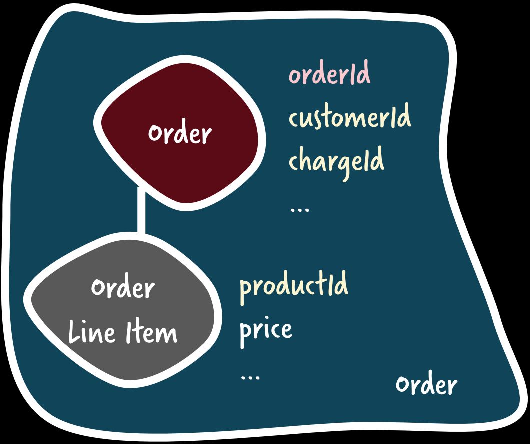

Charge

chargeId
amount
paymentProvider
transactionId
...

Charge

Own DB / autonomous technology decisions
API-driven
Scalability
...

You might know this from microservices


Order

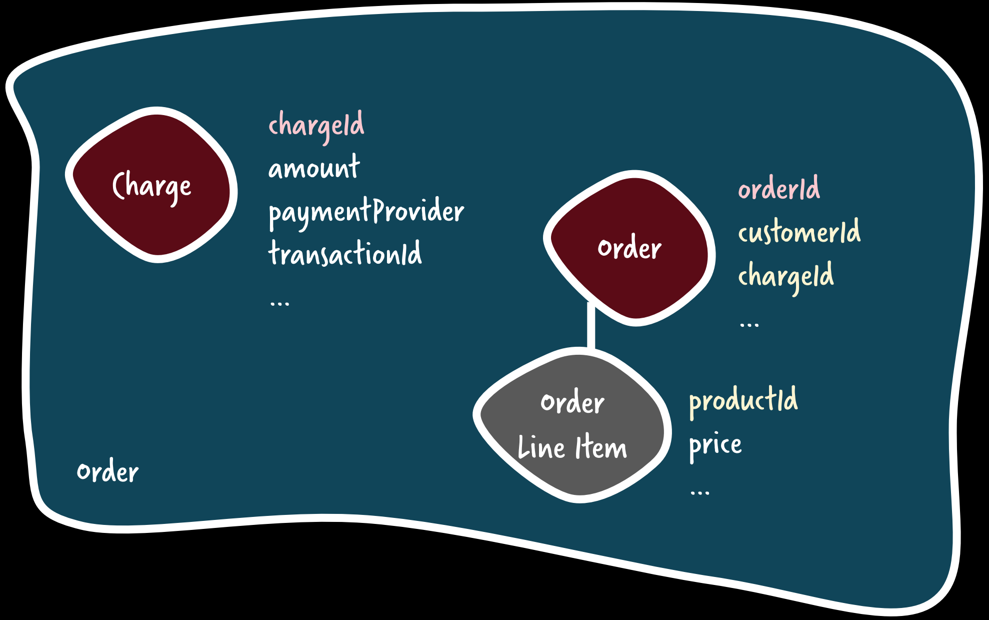
orderId
customerId
chargeId
...

Order Line Item

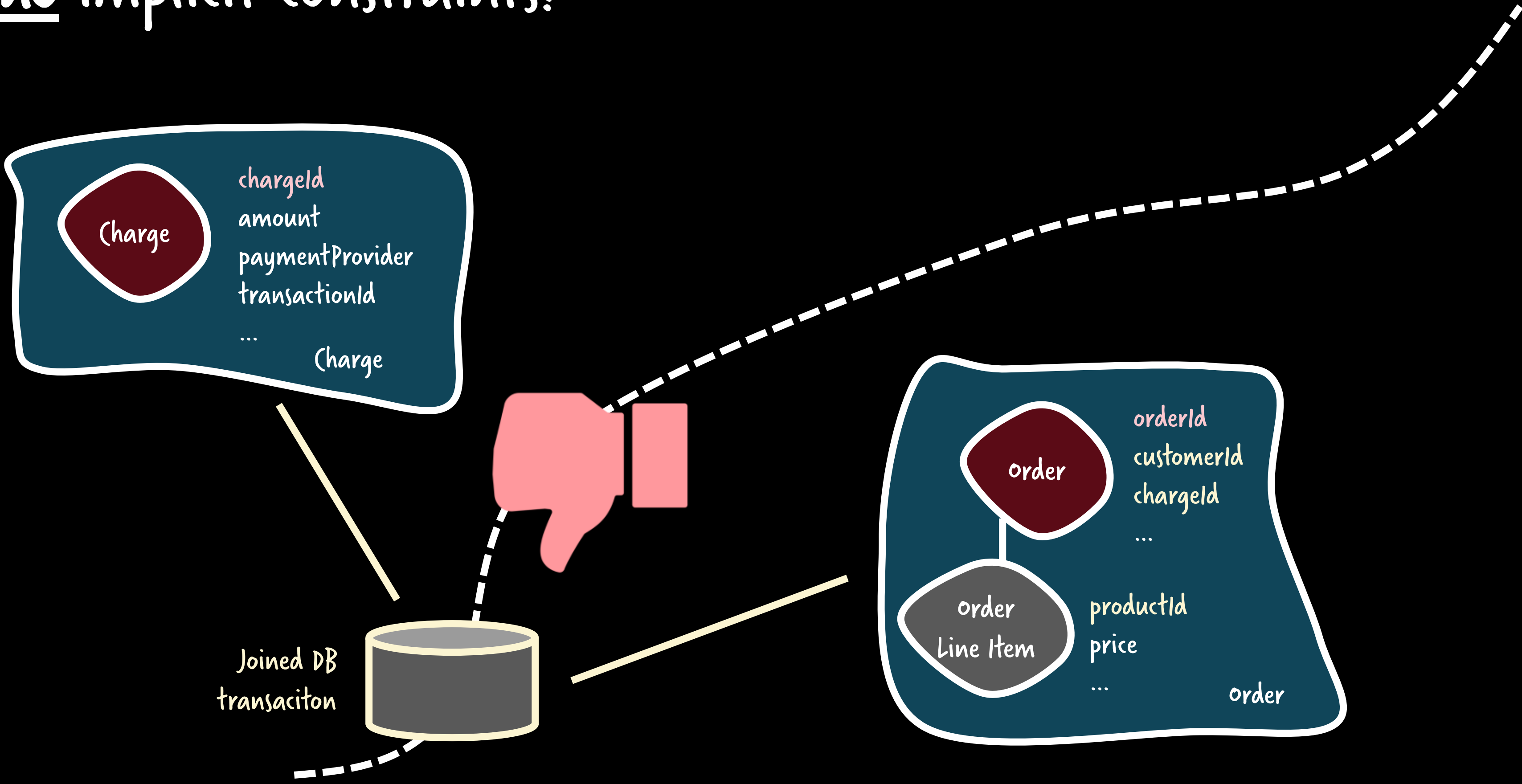
productId
price
...

Order

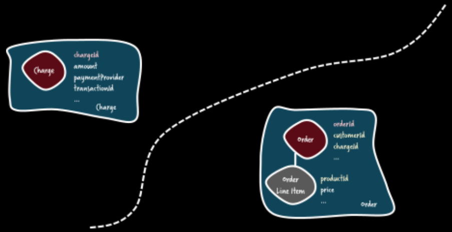
# Boundaries need to be designed carefully

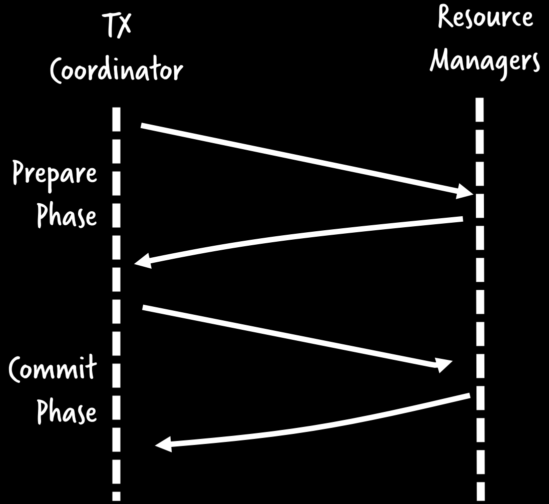


# But <u>no</u> implicit constraints!



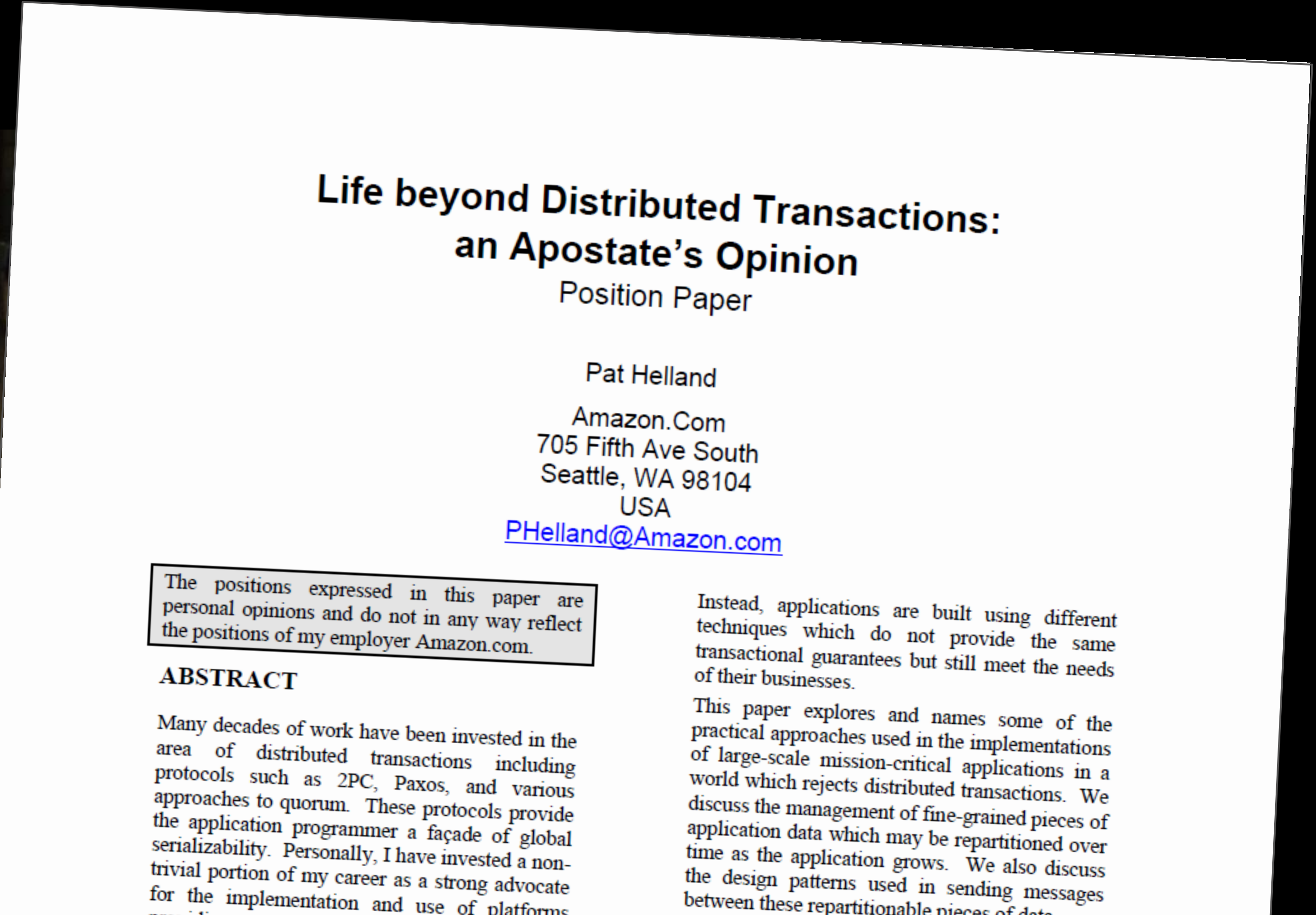
# There is two-phase commit (XA)!!





## Pat Helland

Distributed Systems Guru
Worked at Amazon,
Microsoft & Salesforce



# Life beyond Distributed Transactions: an Apostate's Opinion

Position Paper

Pat Helland

Amazon.Com
705 Fifth Ave South
Seattle, WA 98104
USA
PHelland@Amazon.com

The positions expressed in this paper are personal opinions and do not in any way reflect the positions of my employer Amazon.com.

## ABSTRACT

Many decades of work have been invested in the area of distributed transactions including protocols such as 2PC, Paxos, and various approaches to quorum. These protocols provide the application programmer a façade of global serializability. Personally, I have invested a non-trivial portion of my career as a strong advocate for the implementation and use of platforms

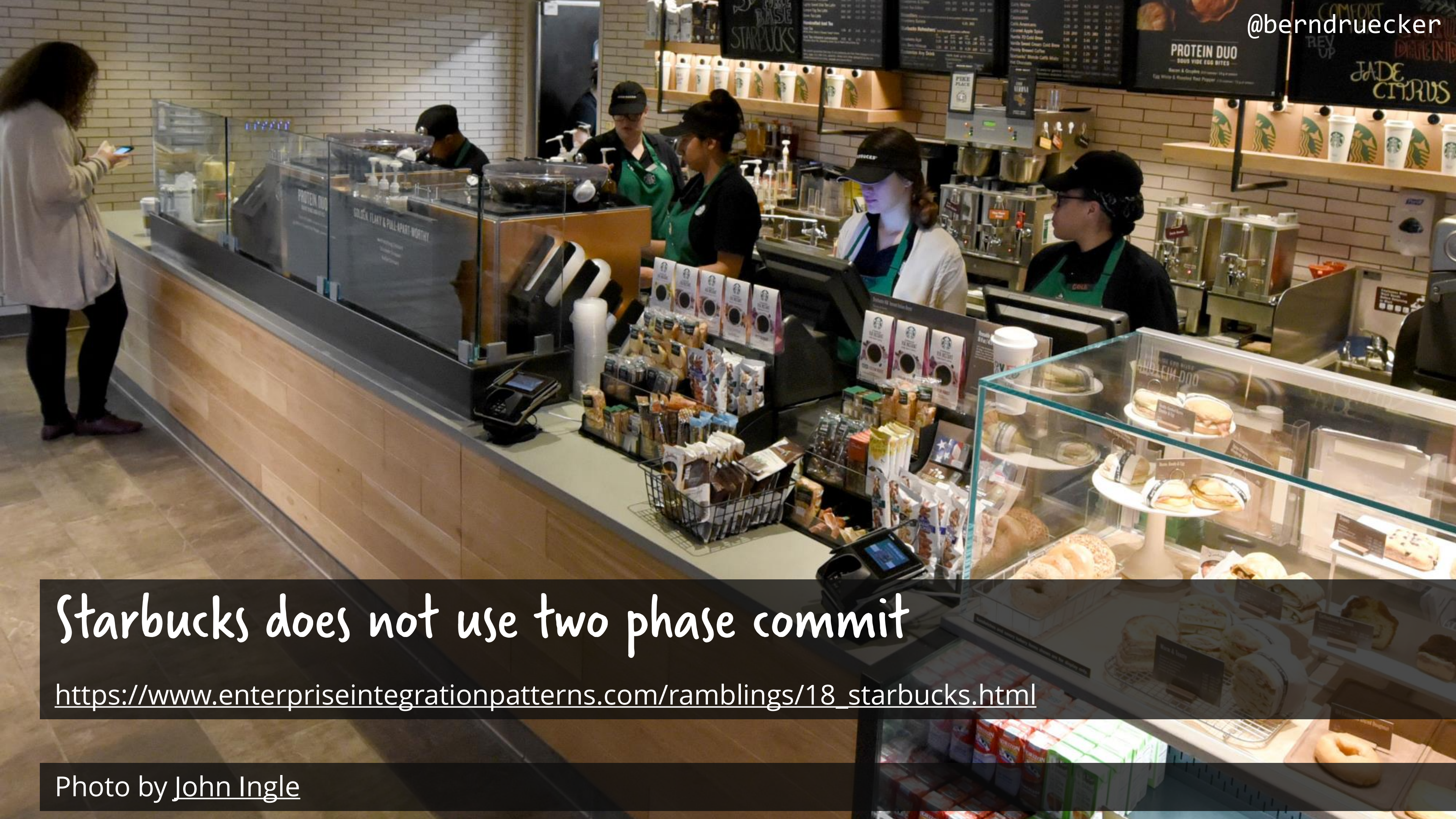
Instead, applications are built using different techniques which do not provide the same transactional guarantees but still meet the needs of their businesses.

This paper explores and names some of the practical approaches used in the implementations of large-scale mission-critical applications in a world which rejects distributed transactions. We discuss the management of fine-grained pieces of application data which may be repartitioned over time as the application grows. We also discuss the design patterns used in sending messages between these repartitionable pieces of data

Pat Helland

Distributed Systems Guru
Worked at Amazon,
Microsoft & Salesforce

> "Grown-Ups Don't Use Distributed Transactions



# Starbucks does not use two phase commit

https://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html

Photo by John Ingle

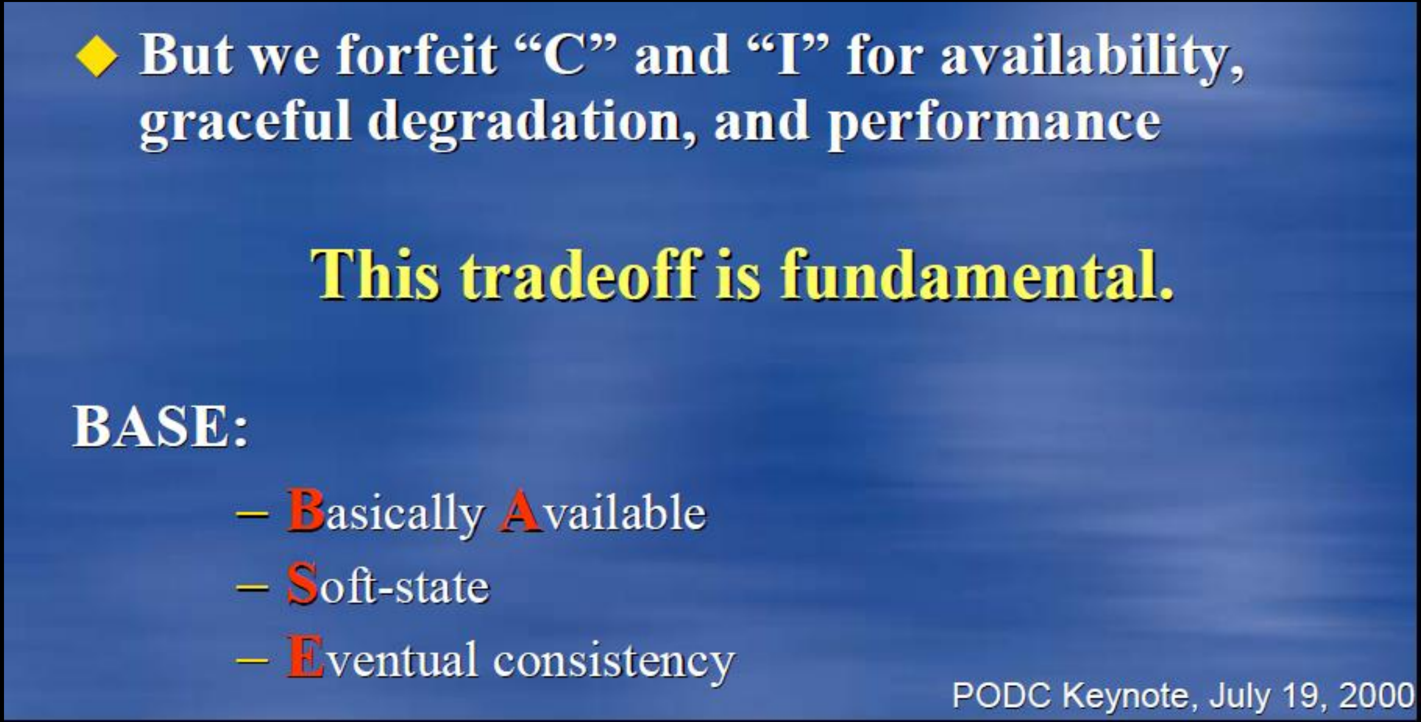
# Eric Brewer

Atomicity

~~Consistency~~

~~Isolation~~

Durability



- But we forfeit "C" and "I" for availability, graceful degradation, and performance

**This tradeoff is fundamental.**

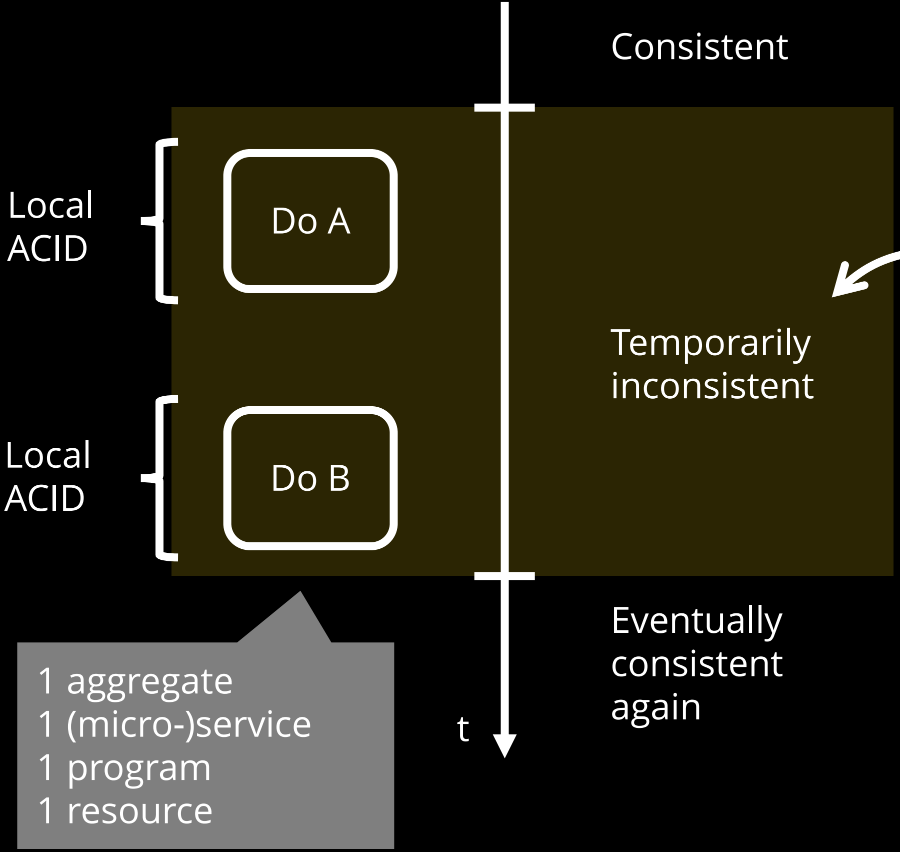
BASE:

- **B**asically **A**vailable
- **S**oft-state
- **E**ventual consistency

PODC Keynote, July 19, 2000

http://pld.cs.luc.edu/courses/353/spr11/notes/brewer_keynote.pdf

# That means



Consistent

Local ACID

Do A

Local ACID

Do B

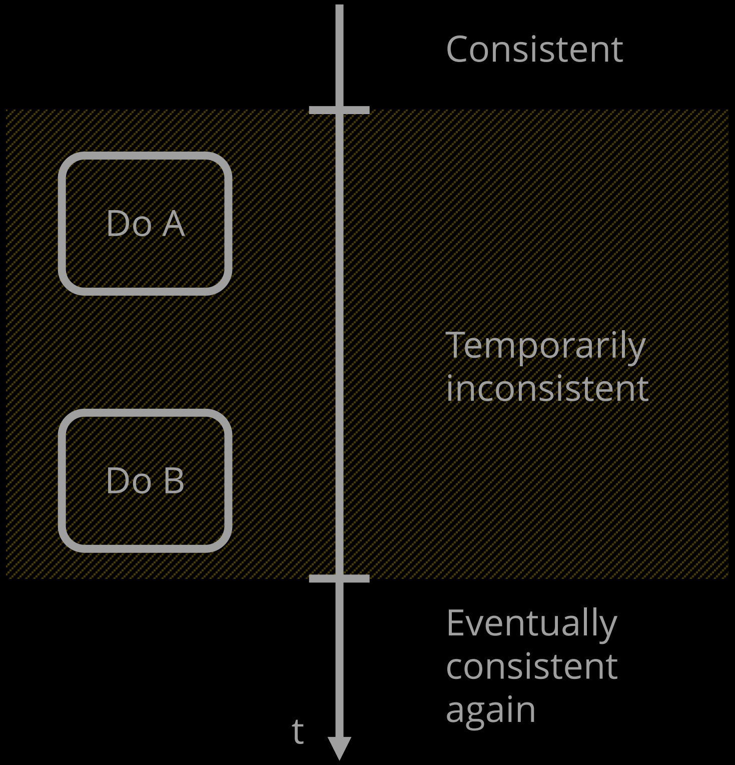
Temporarily inconsistent

Eventually consistent again

t

1 aggregate
1 (micro-)service
1 program
1 resource

Violates „I" of ACID



You might know this from:

BCE ECB EZB EKT EKP 2002
Angaben zum Zahlungsempfänger: Name, Vorname/Firma
WALTER MÜLLER
IBAN
DE70 4991 0000 0000 7
BIC des Kreditinstituts/Zahlungsdienstleisters
BYALDME1PA

Photo by Gerhard51, available under Creative Commons CC0 1.0 license.

Pat Helland

„Building on Quicksand" Paper

A

C

I

D

2.0

Associative $(a + b) + c = a + (b + c)$

Commutative $a + b = b + a$

Idempotent $f(x) = f( f(x) )$

Distributed

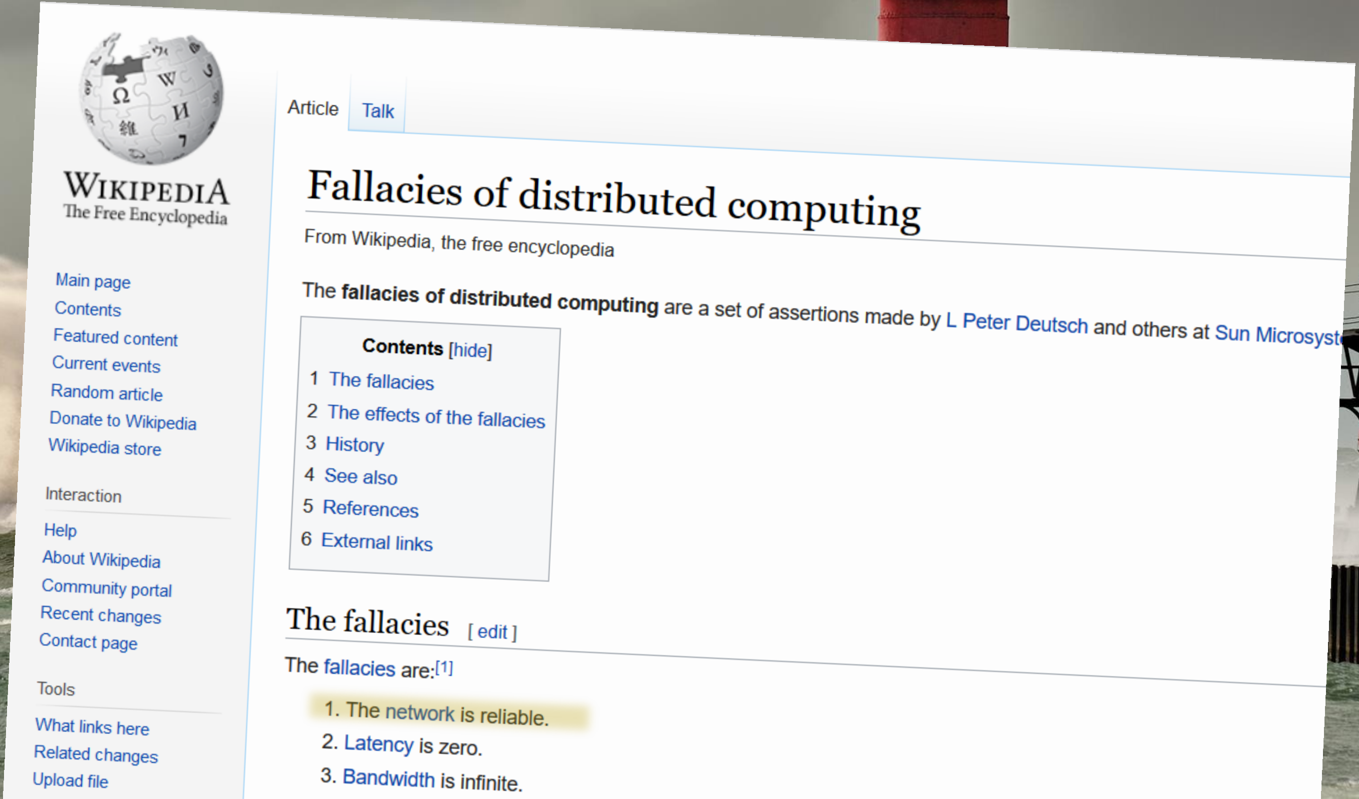
2.0

Pat Helland

„Building on Quicksand" Paper

# Distributed

# Distributed systems



## Fallacies of distributed computing

From Wikipedia, the free encyclopedia

The **fallacies of distributed computing** are a set of assertions made by L Peter Deutsch and others at Sun Microsyst

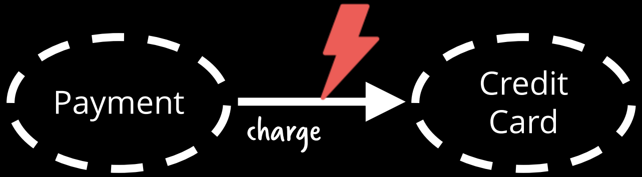
**Contents** [hide]

### The fallacies [ edit ]

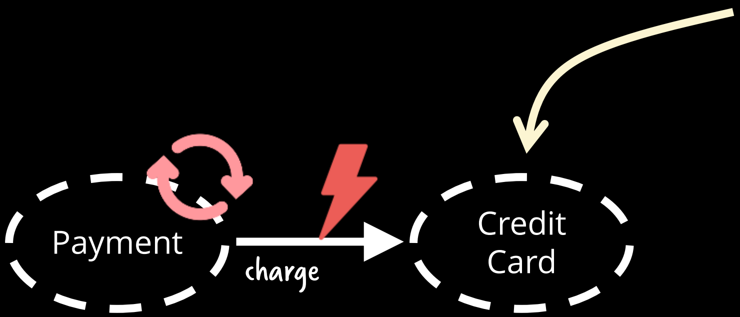
The fallacies are:[1]

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.

# Network problems



# Strategy: retry



Payment

charge

Credit Card

has to be idempotent

**Charge Credit Card**
cardNumber
amount

← Not idempotent

**Charge Credit Card**
cardNumber
amount
transactionId

← Idempotent

Photo by pixabay, available under Creative Commons CC0 1.0 license.

# Requirement: Idempotency of services!

Photo by pixabay, available under Creative Commons CC0 1.0 license.

# Requirement: Idempotency of services!

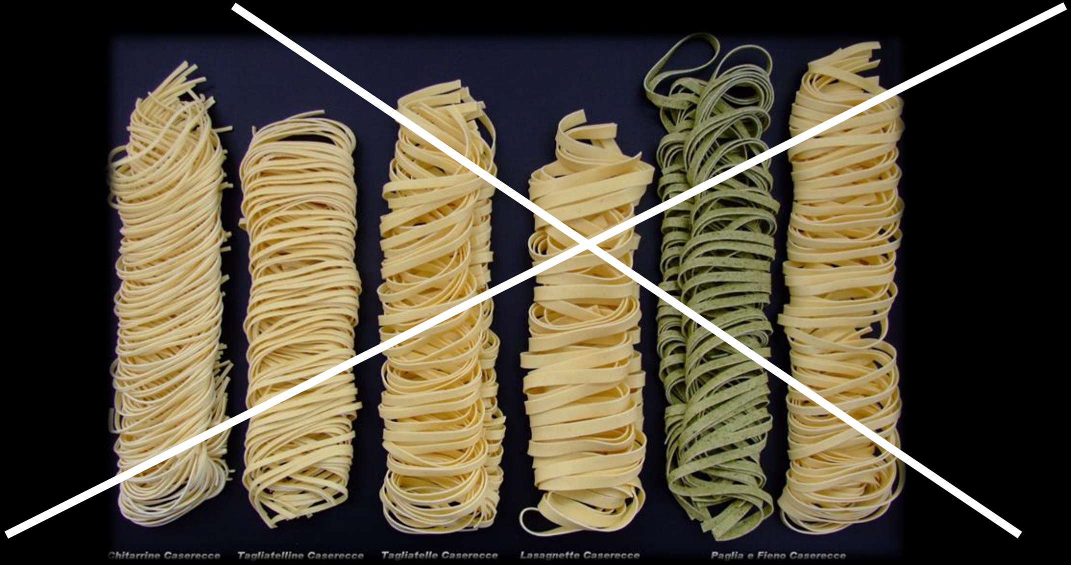
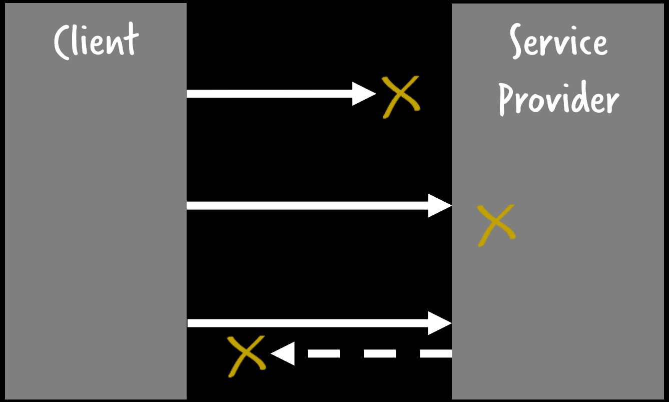


Photo by Chr.Späth, available under Public Domain.

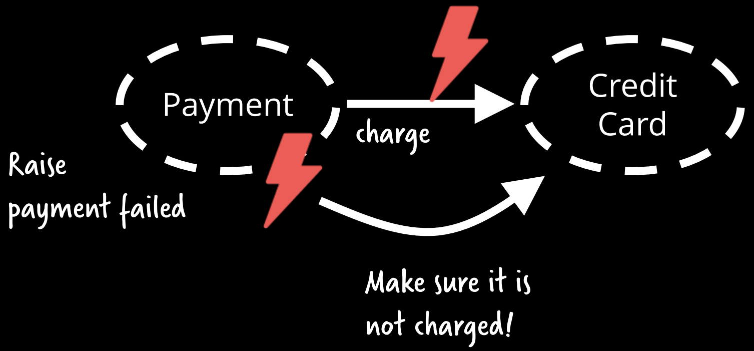
# Distributed systems

# It is impossible to differentiate certain failure scenarios:

Independant of communication style!



# Strategy: Cleanup

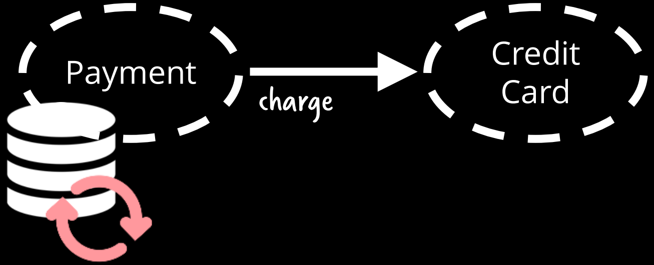


<u>Cancel charge</u>
cardNumber
amount
transactionId

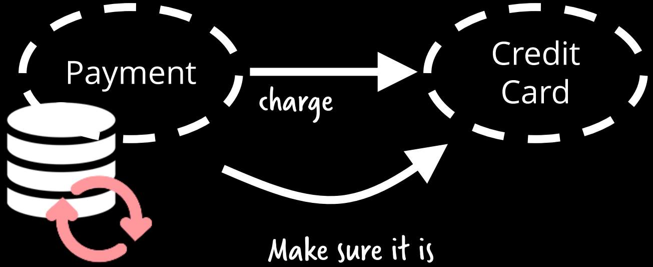
# Distributed systems

Some communication challenges require state.

# Strategy: Stateful retry



# Strategy: Stateful retry

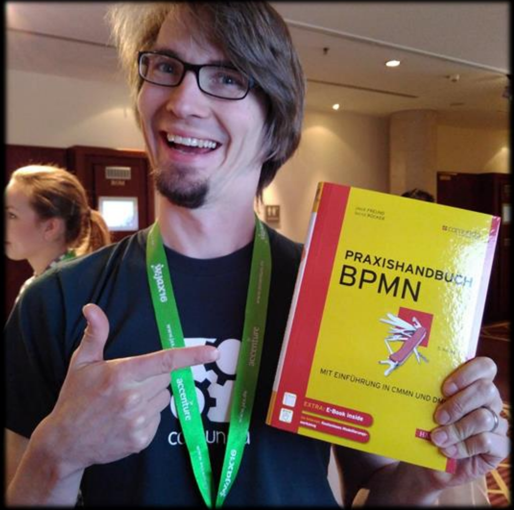


Make sure it is not charged!

# Warning:
# Contains Opinion

# Bernd Ruecker

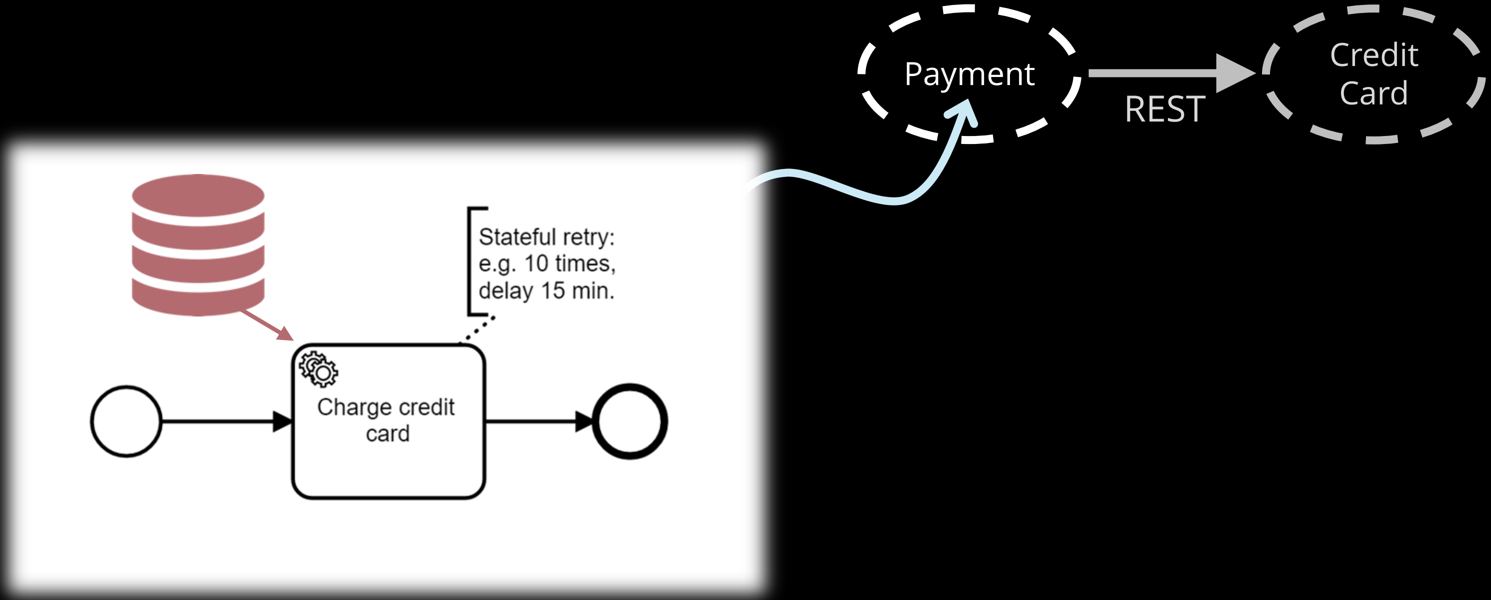
Co-founder and
Technologist of
Camunda



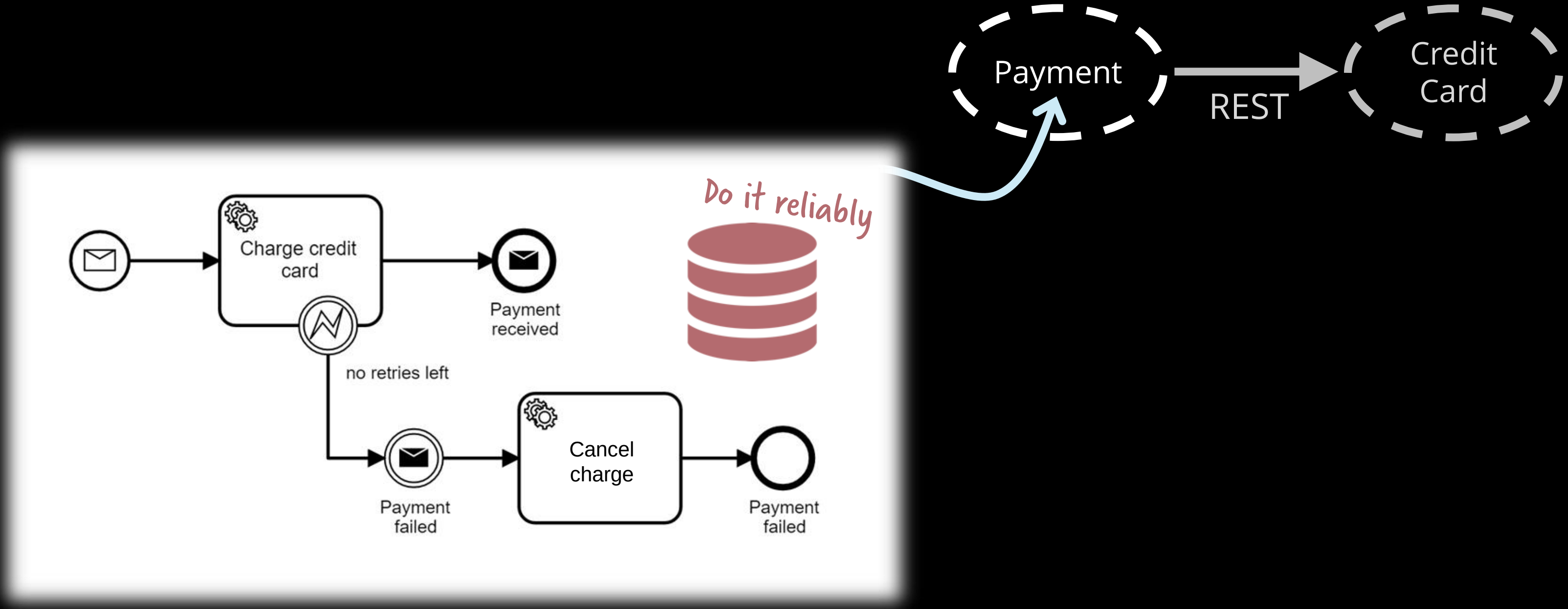
Berlin, Germany

bernd.ruecker@camunda.com
@berndruecker

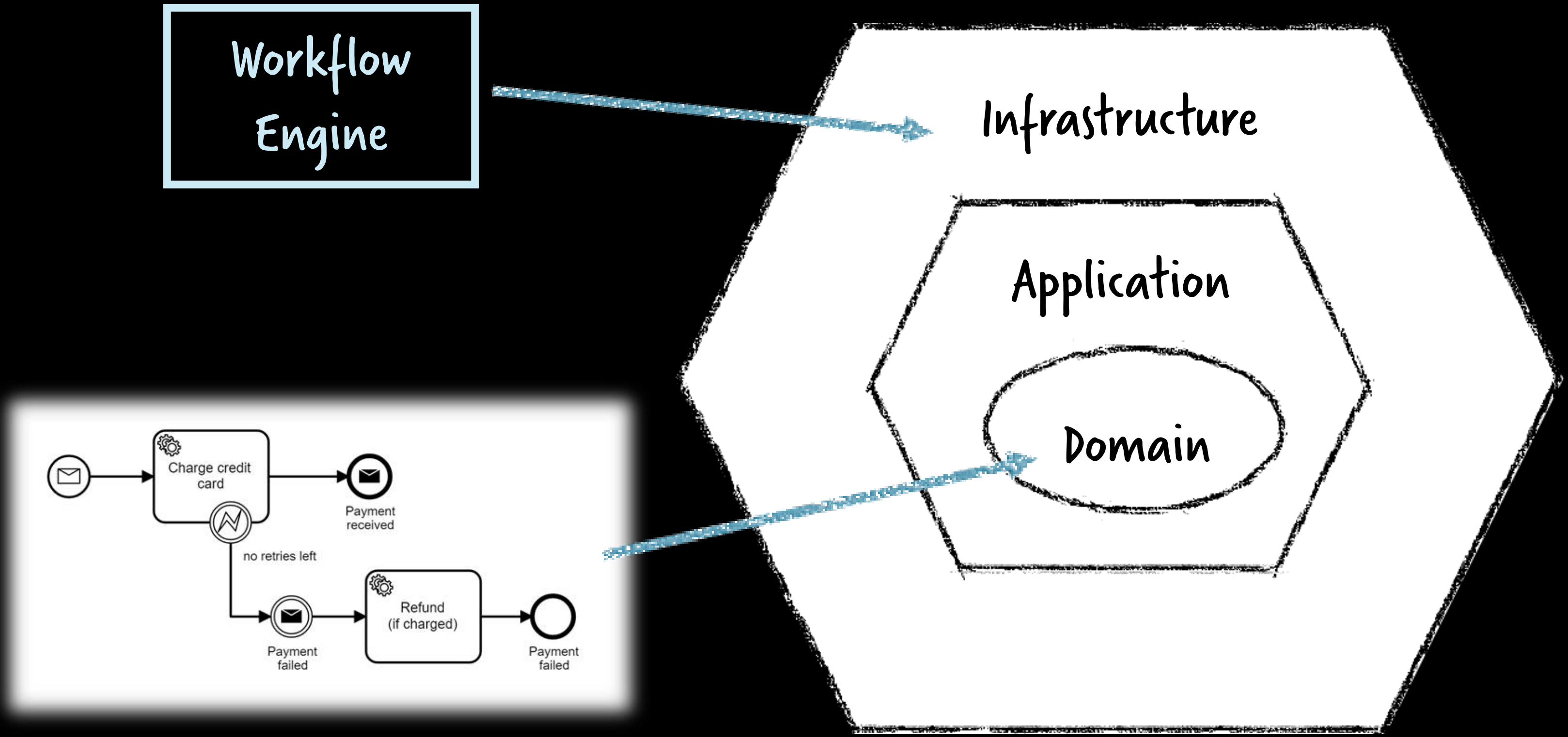
# Stateful retry



# Stateful retry & cleanup



# Architecture



Workflow Engine

Infrastructure

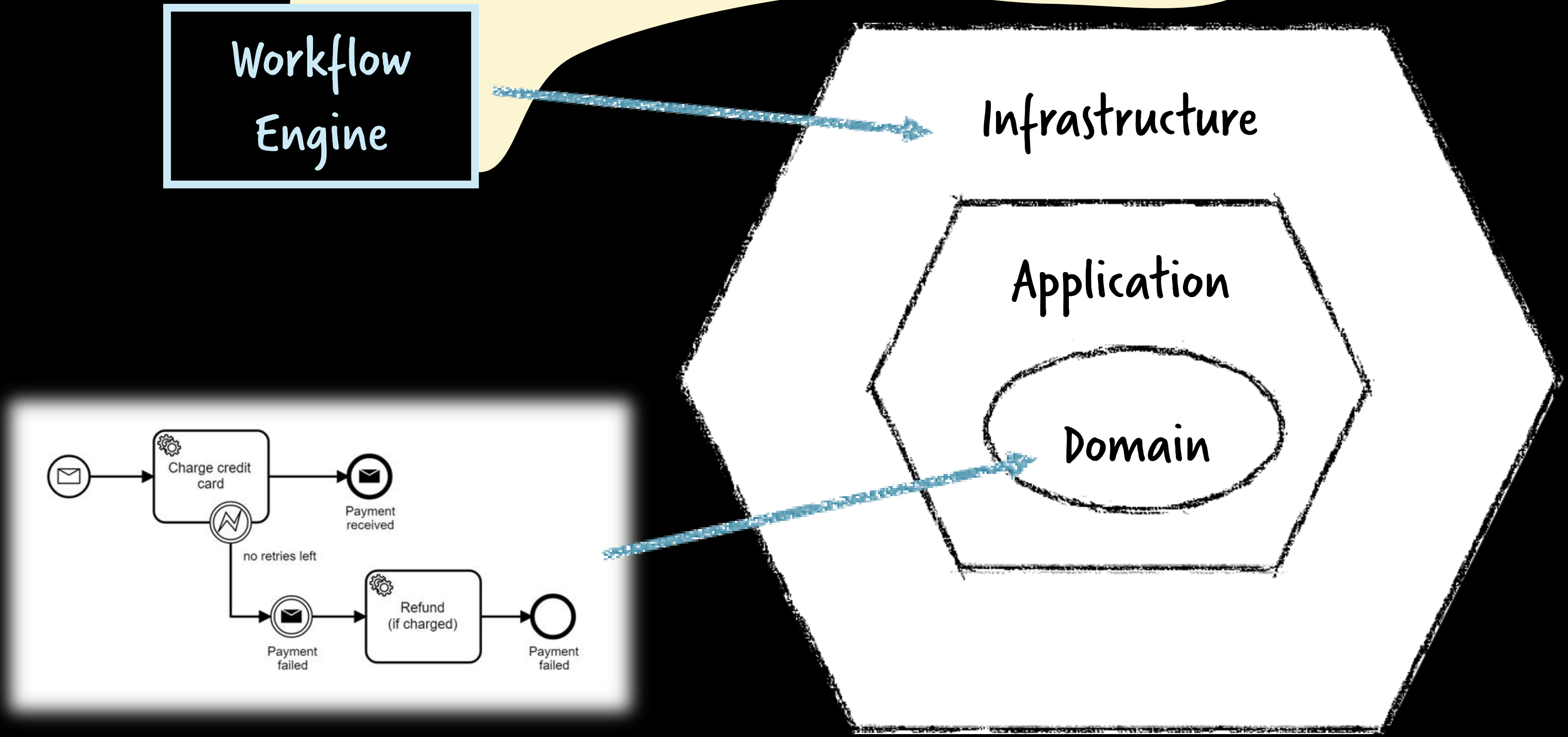
Application

Domain

# Architecture

camunda

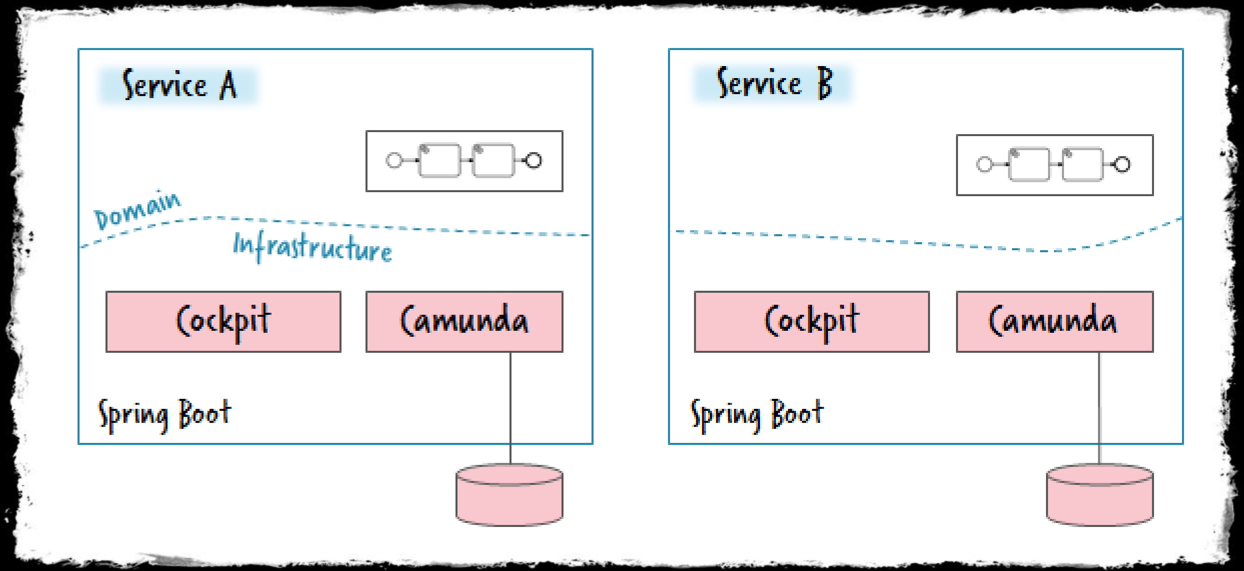
Lightweight OSS workflow engine



# Live hacking

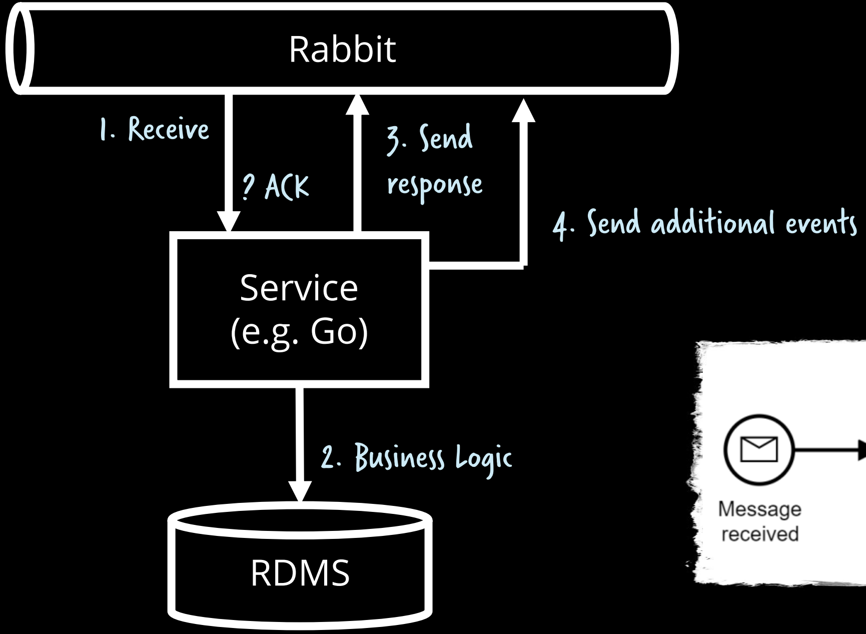
https://github.com/flowing/flowing-retail/tree/master/rest

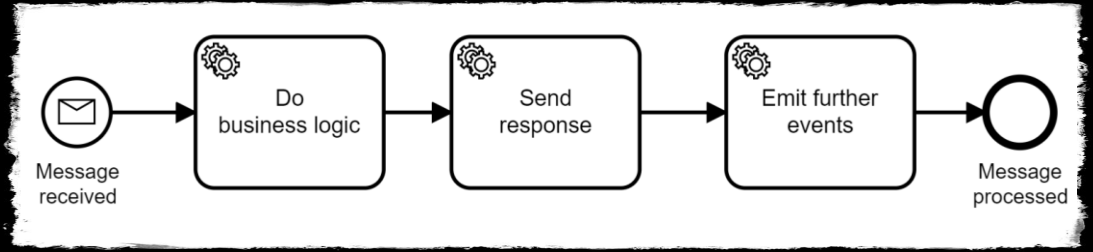
# Embedded Engine Example (Java)

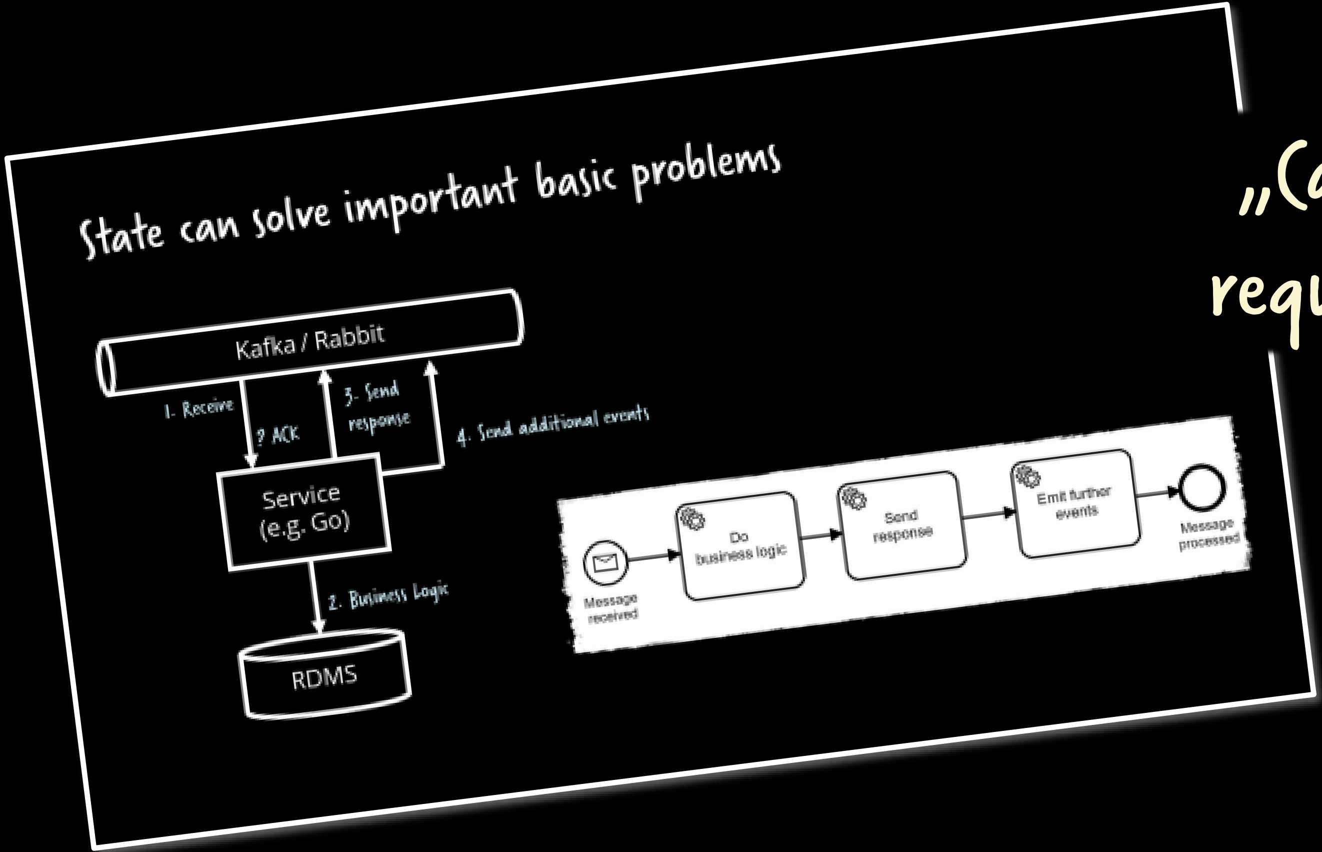


https://blog.bernd-ruecker.com/architecture-options-to-run-a-workflow-engine-6c2419902d91

# A relatively common pattern







State can solve important basic problems

Kafka / Rabbit

1. Receive

? ACK

3. Send response

4. Send additional events

Service (e.g. Go)

2. Business Logic

RDMS

Message received

Do business logic

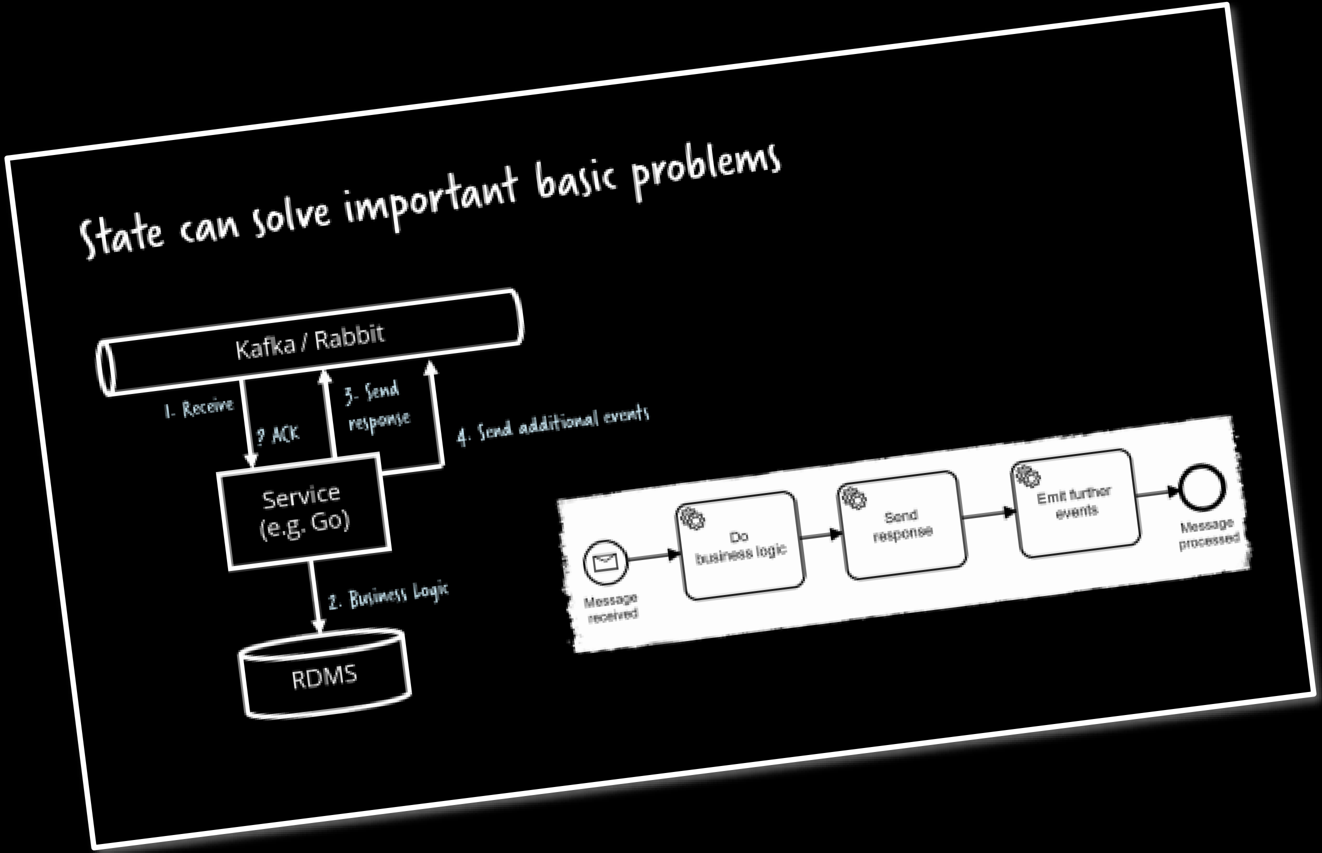
Send response

Emit further events

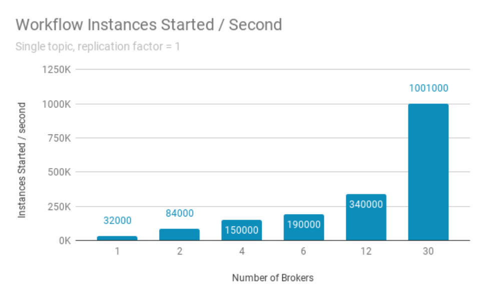
Message processed

„Can this handle 15k requests per second?"

„Yes."

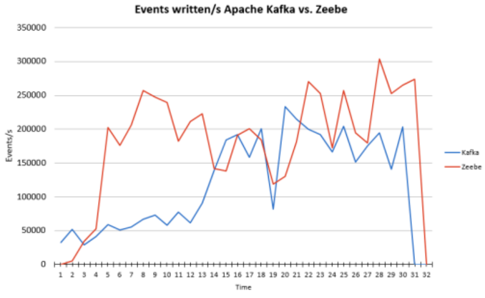


State can solve important basic problems



Workflow Instances Started / Second

Single topic, replication factor = 1

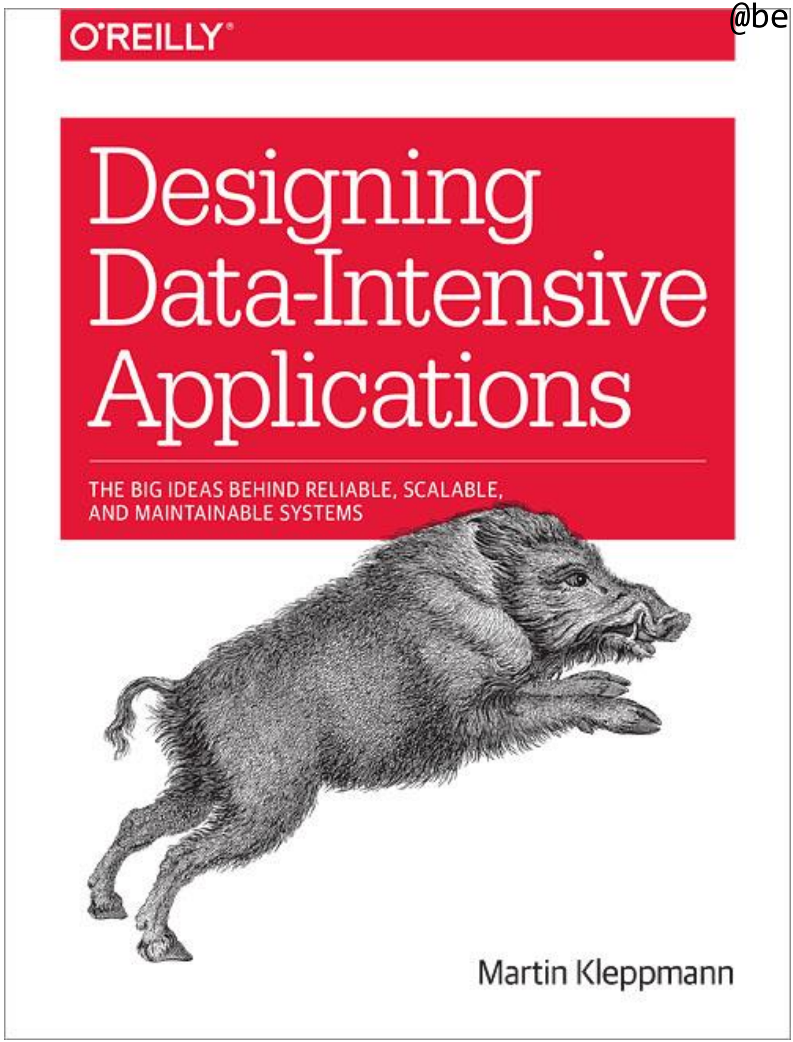


Events written/s Apache Kafka vs. Zeebe

strangeloop

Sept 25-26, 2015

thestrangeloop.com



O'REILLY®

Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS

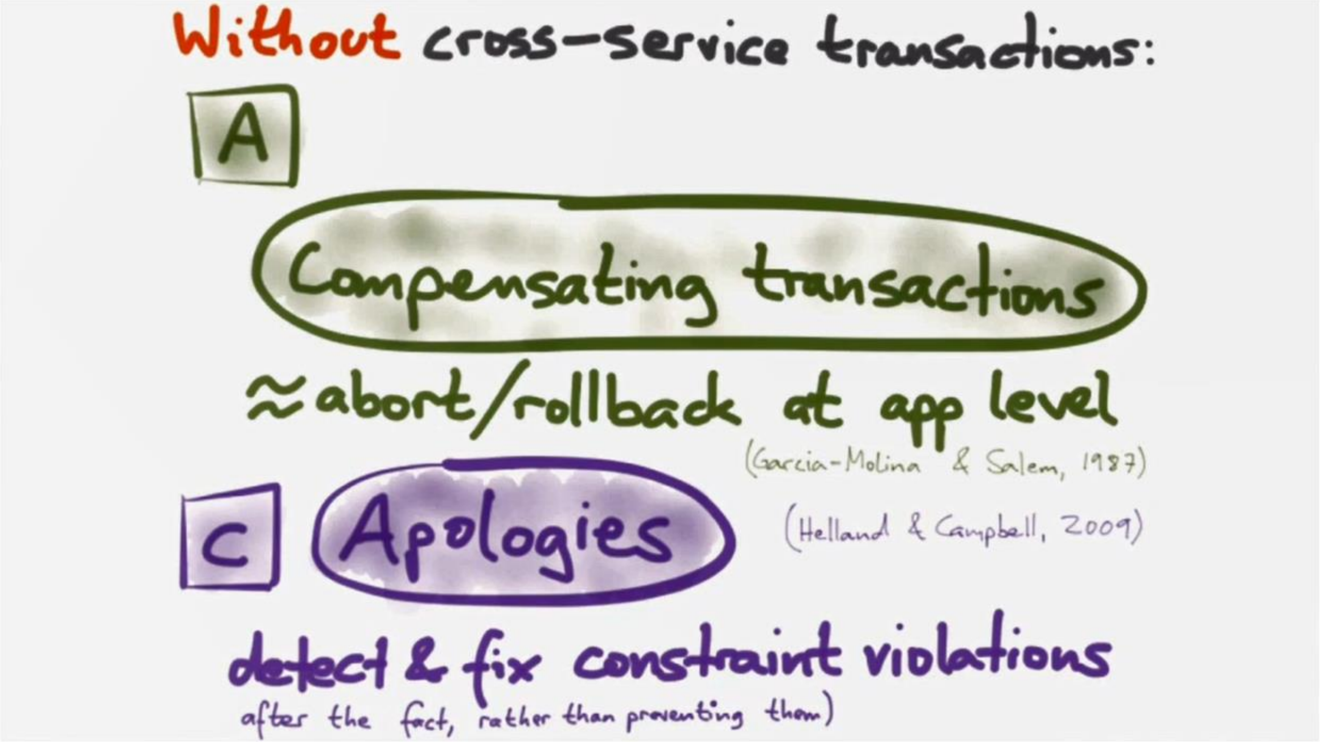
Martin Kleppmann

@berndruecker

strangeloop

Sept 25-26, 2015

thestrangeloop.com



Without cross-service transactions:

A

Compensating transactions

≈ abort/rollback at app level

(Garcia-Molina & Salem, 1987)

C

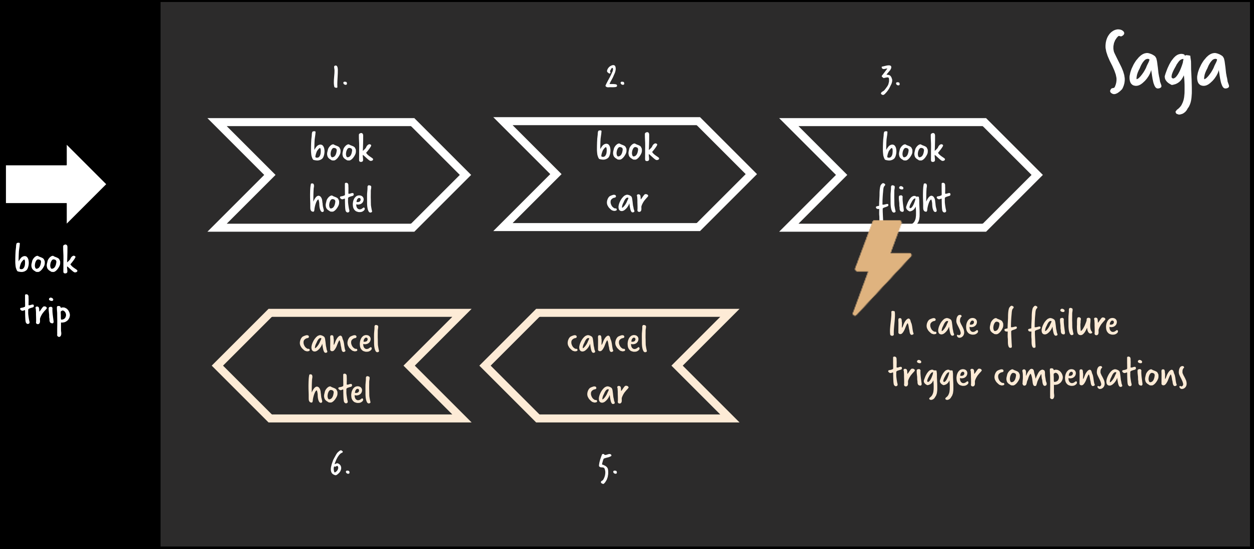
Apologies

(Helland & Campbell, 2009)

detect & fix constraint violations

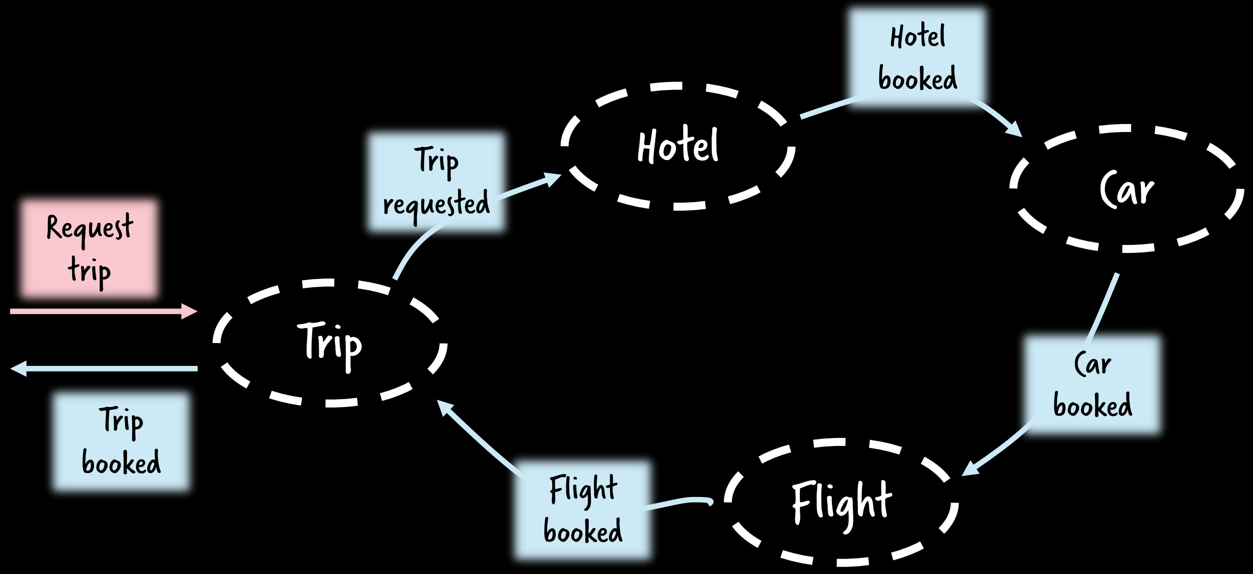
after the fact, rather than preventing them)

# Compensation – the classical example



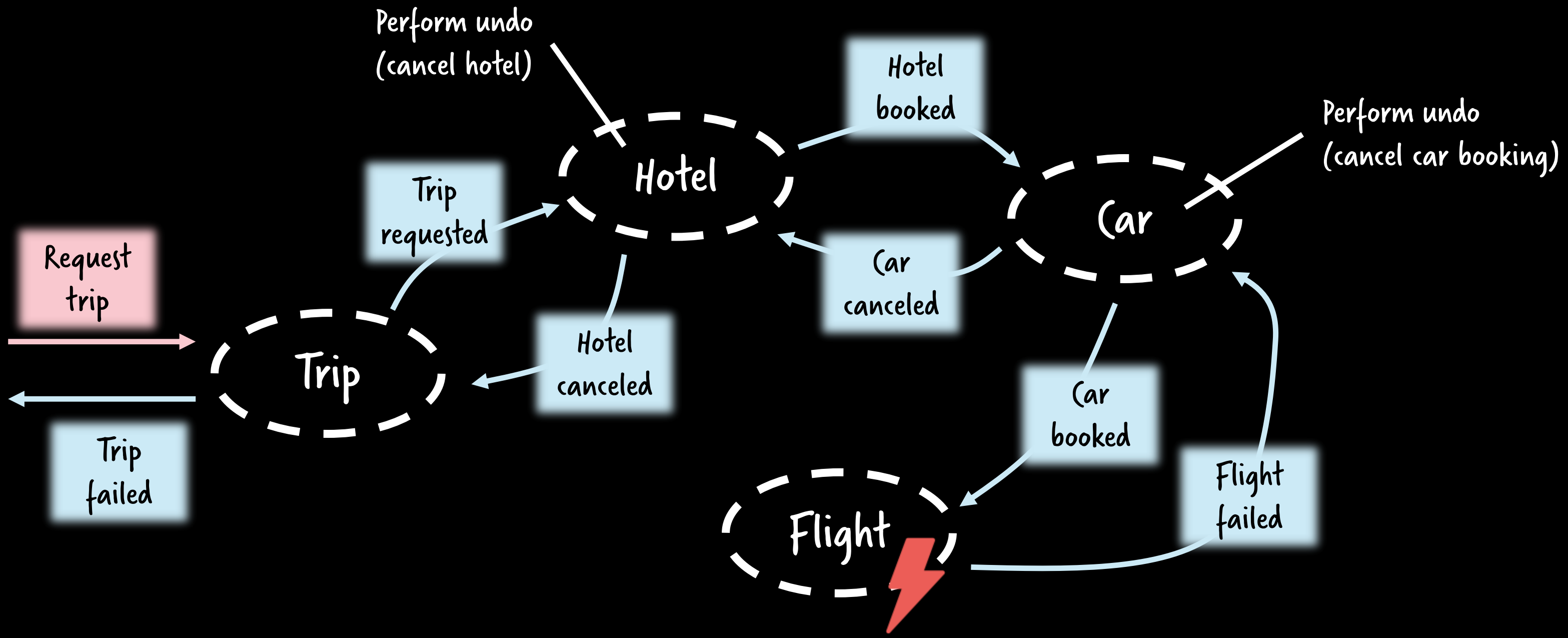
# 2 alterntive approaches: choreography & orchestration

# Event-driven choreography



@berndruecker

# Event-driven choreography



The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow, and thus set yourself up for trouble in future years.

https://martinfowler.com/articles/201701-event-driven.html

The danger is that it's very easy to make nicely decoupled systems with event notification, **without realizing that you're losing sight of that larger-scale flow**, and thus set yourself up for trouble in future years.

https://martinfowler.com/articles/201701-event-driven.html

The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow, and thus **set yourself up for trouble in future years**.

https://martinfowler.com/articles/201701-event-driven.html

Denis Rosa
Couchbase

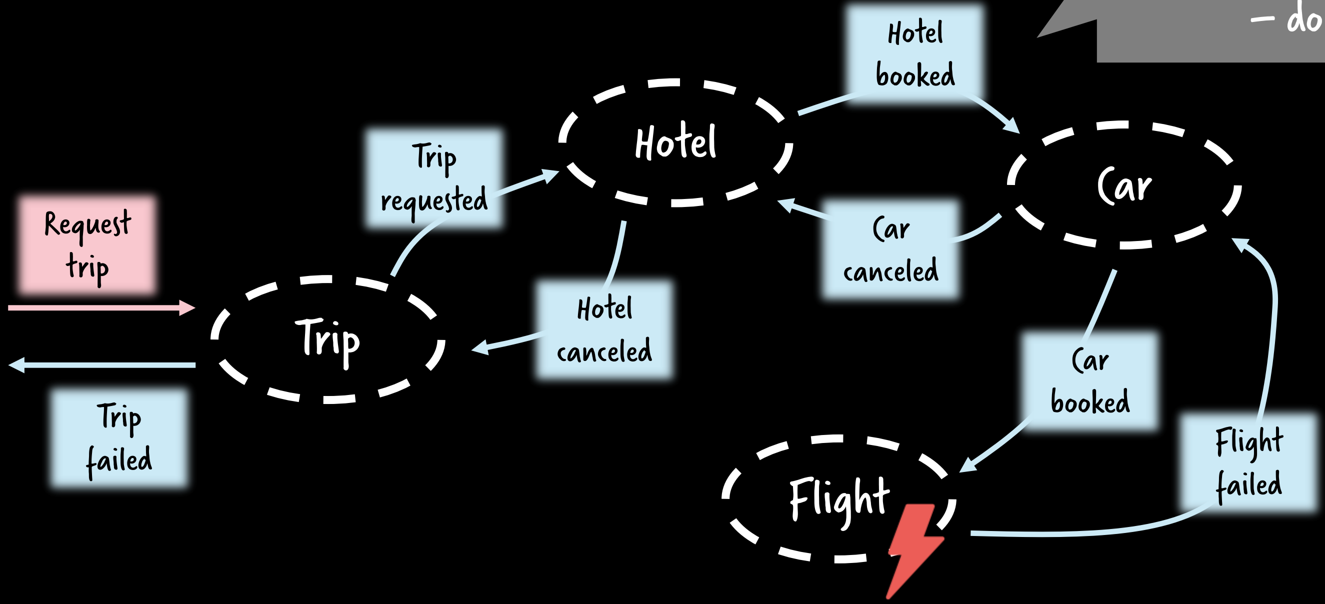
If your transaction involves 2 to 4 steps, choreography might be a very good fit.

However, **this approach can rapidly become confusing** if you keep adding extra steps in your transaction as **it is difficult to track which services listen to which events**. Moreover, it also might **add a cyclic dependency** between services as they have to subscribe to one another’s events.

https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/

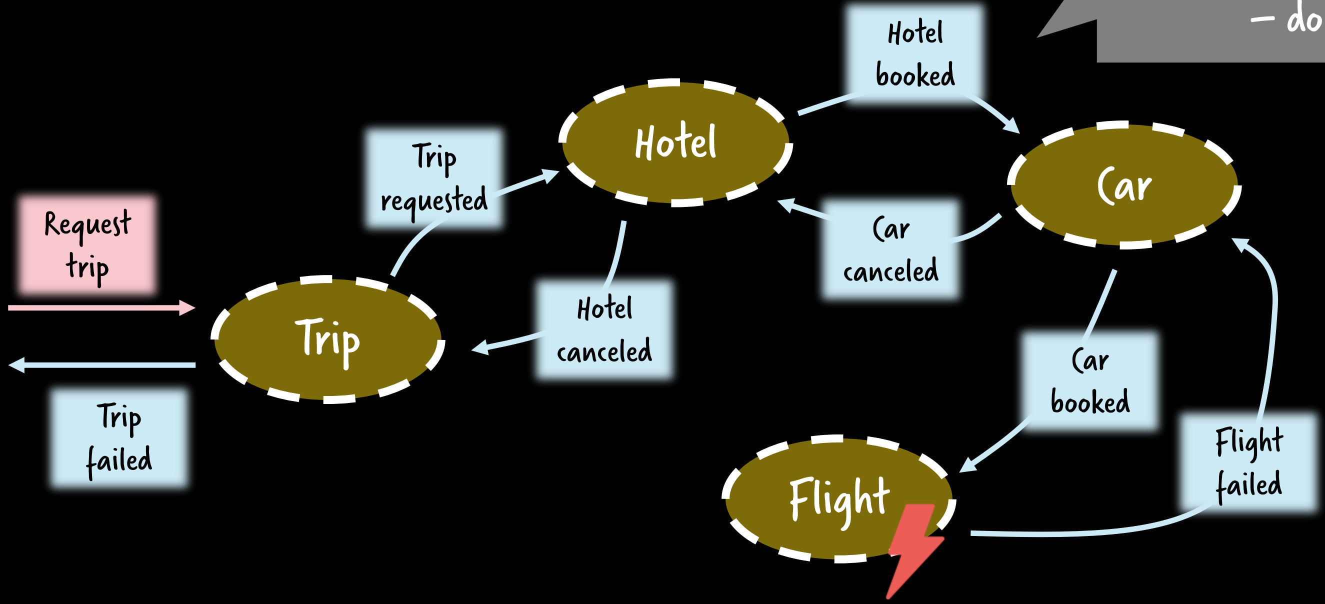
# Implementing changes in the process

We have a new basic agreement with the car rental agency and can cancel for free within 1 hour – do that first!



# Implementing changes in the process

We have a new basic agreement with the car rental agency and can cancel for free within 1 hour – do that first!



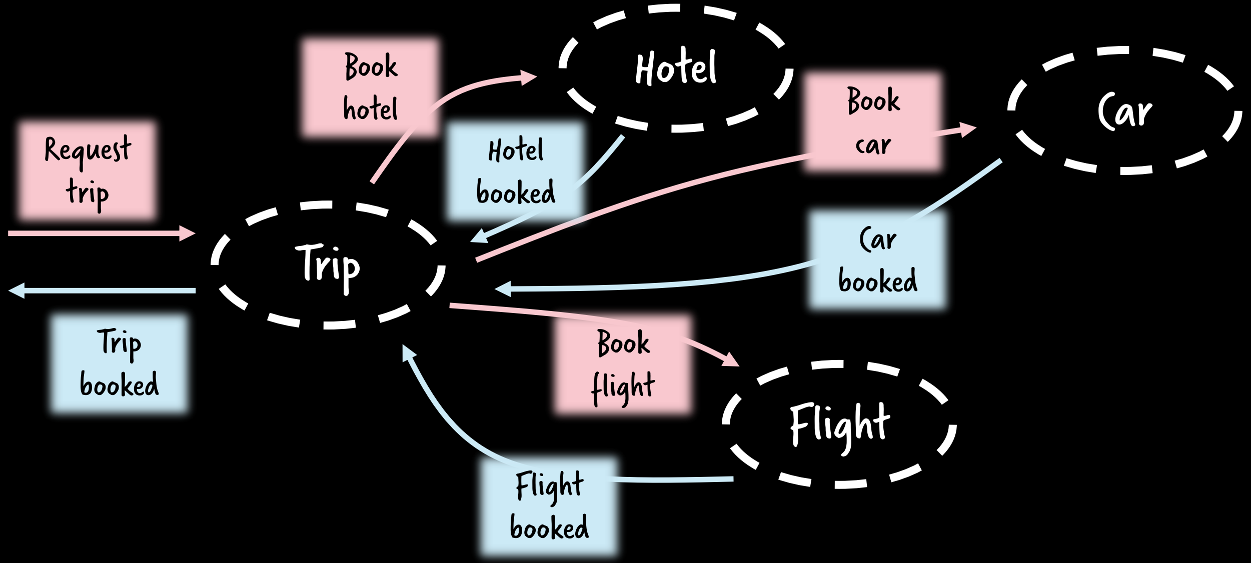
You have to adjust all services and redeploy at the same time!

What we wanted

vs. what we got

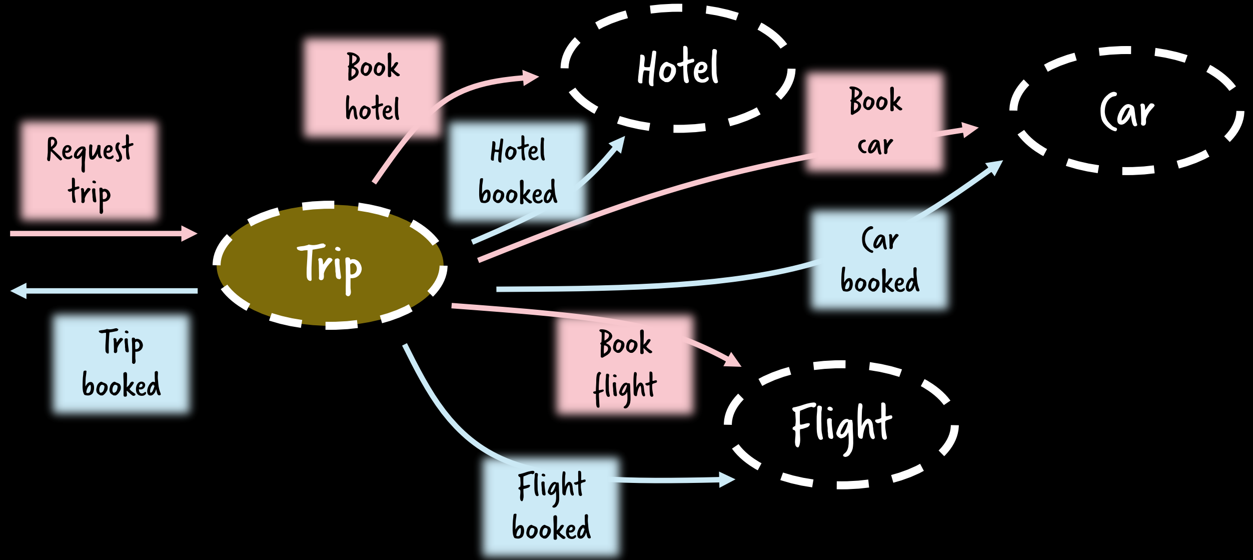
Photo by Lijian Zhang, available under Creative Commons SA 2.0 License and Pedobear19 / CC BY-SA 4.0

# orchestration



# Orchestration

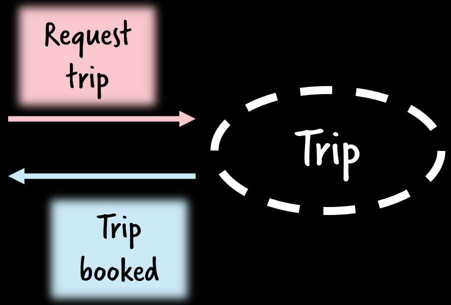
We have a new basic agreement with the car rental agency and can cancel for free within 1 hour – do that first!

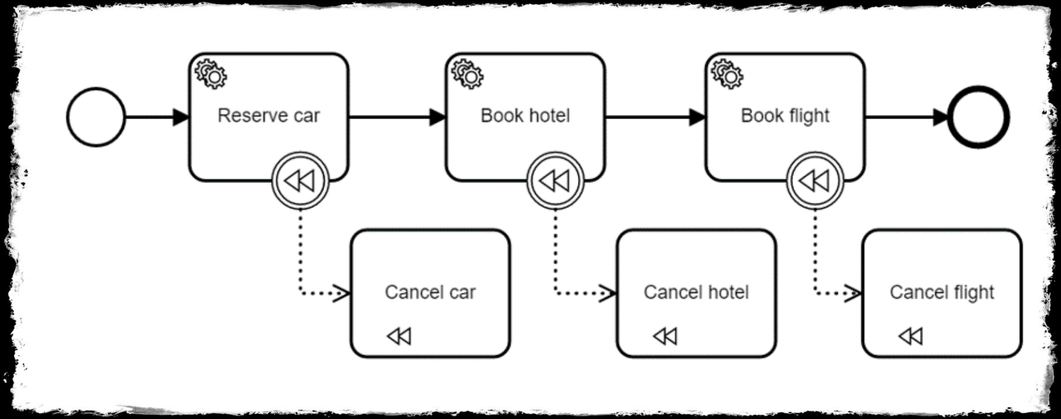


You have to adjust one service and redeploy only this one!

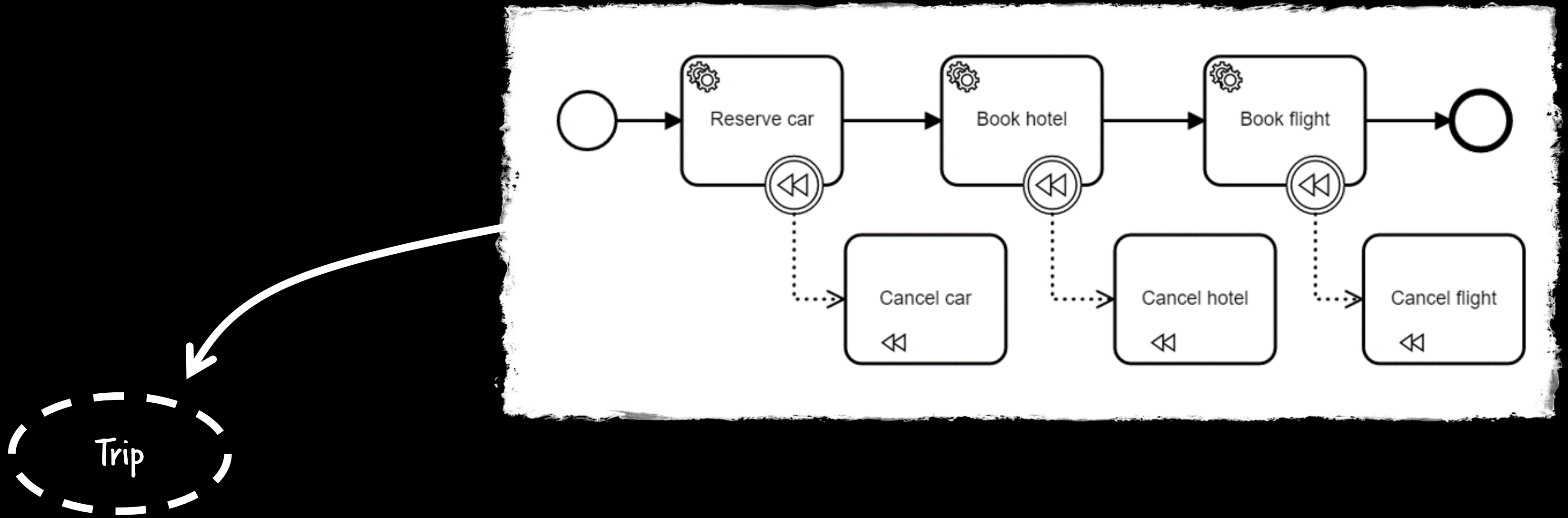
# Describe orchestration with BPMN

Saga Pattern
(implemented by BPMN compensation)

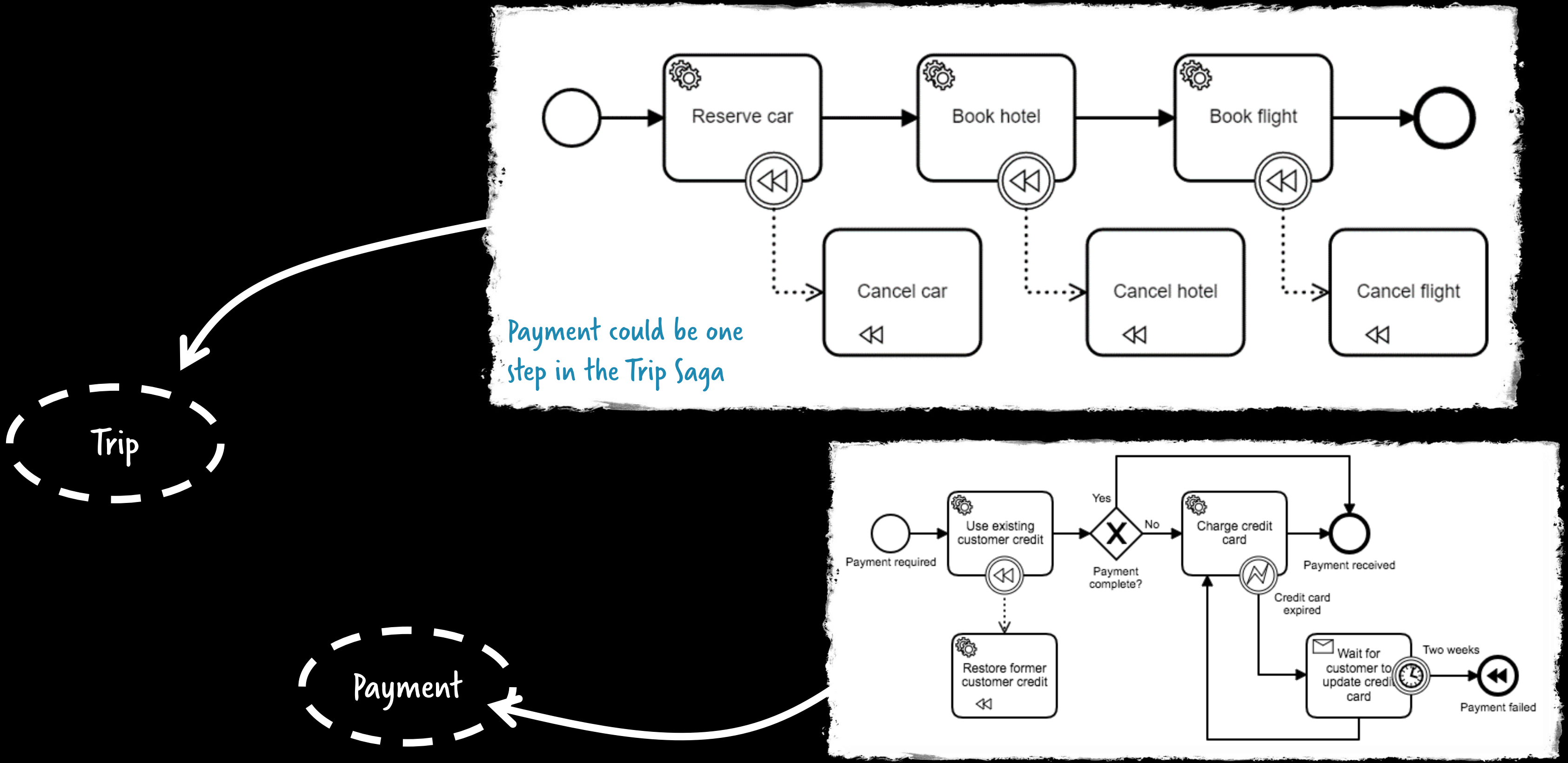




# The workflow is domain logic as part of the service



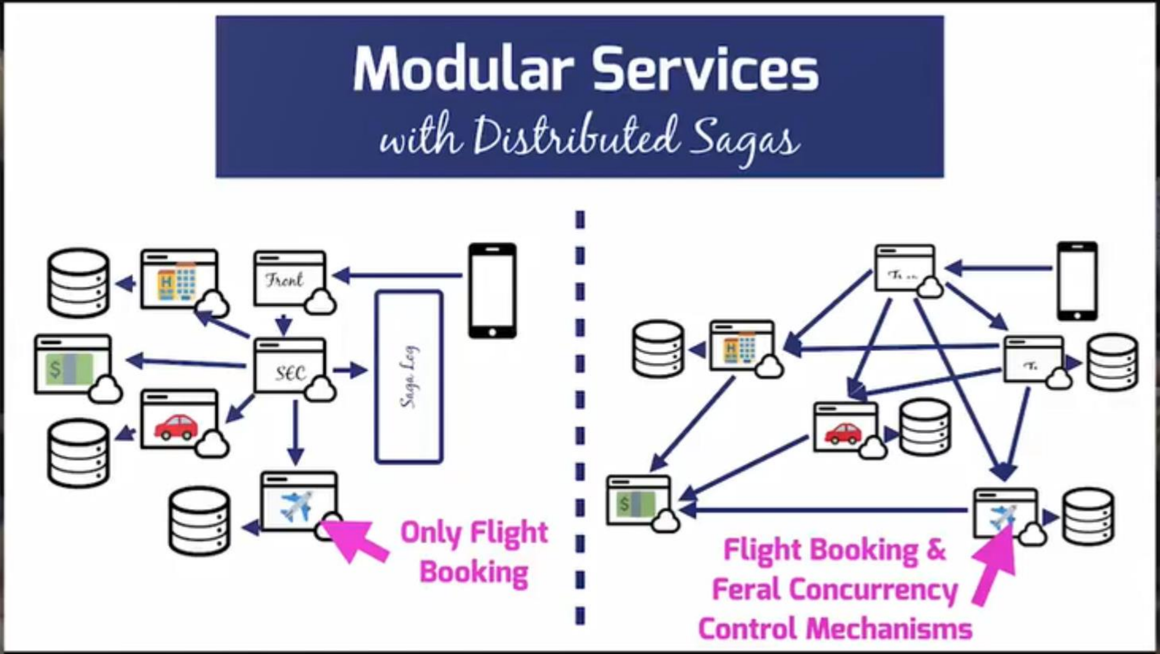
# The workflow is domain logic as part of the service



Payment could be one step in the Trip Saga

ON THE BEACH

{ IT_Event(); }

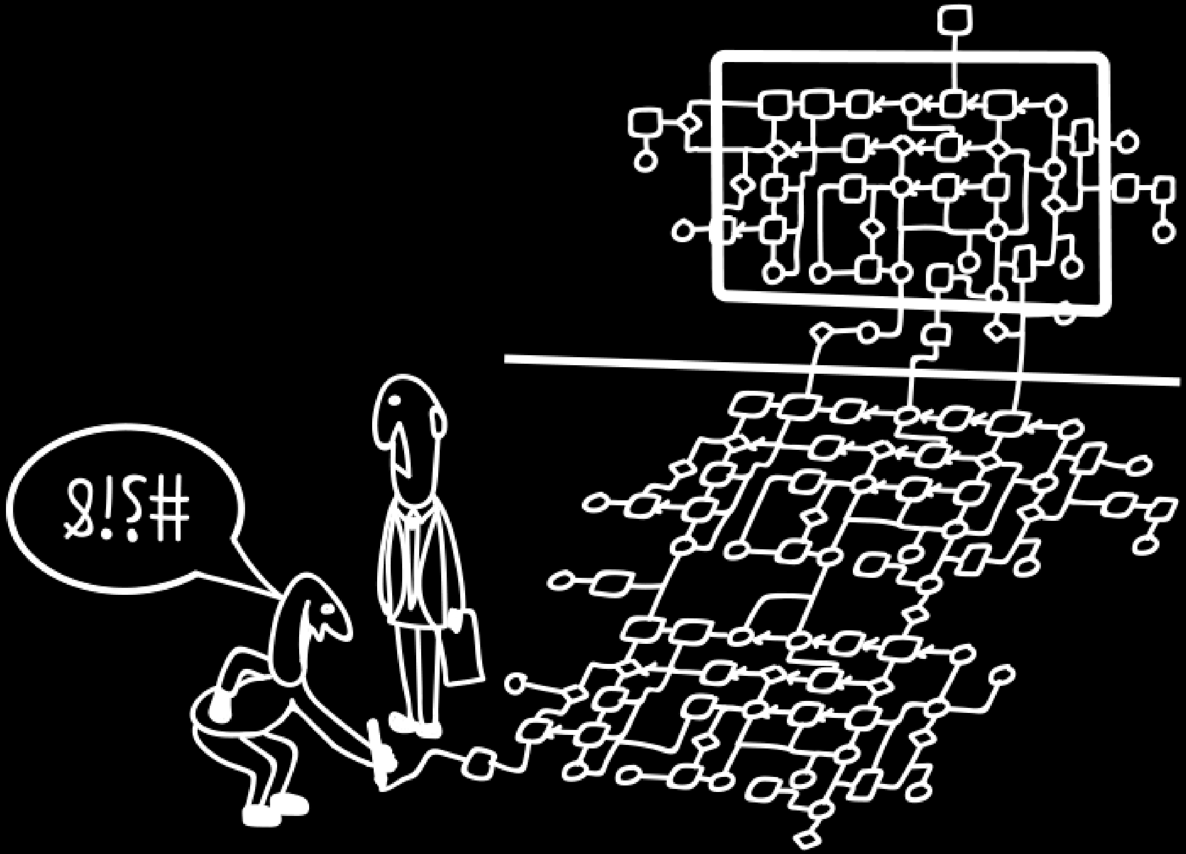


## Modular Services

*with Distributed Sagas*

**Caitie McCaffrey | @caitie**

# Graphical models?



@berndruecker

Clemens Vasters
Architect at Microsoft

http://vasters.com/archive/Sagas.html

Clemens Vasters

## Sagas

Today has been a lively day in some parts of the Twittersphere debating the Saga pattern. As it stands, there are a few frameworks for .NET out there that use the term "Saga" for some framework implementation of a state machine or workflow. Trouble is, that's not what a Saga is. A Saga is a failure management pattern.

Sagas come out of the realization that particularly long-lived transactions (originally even just inside databases), but also far distributed transactions across location and/or trust boundaries can't easily be handled using the classic ACID model with 2-Phase commit and holding locks for the duration of the work. Instead, a Saga splits work into individual transactions whose effects can be, somehow, reversed after work has been performed and committed.

Rental Cars
Hotels
Flights
Car!
Hotel!
Flight!
Book
Cancel

The picture shows a simple Saga. If you book a travel itinerary, you want a car and a hotel and a flight. If you can't get all of them, it's probably not worth going. It's also very certain that you can't enlist all of these providers into a distributed ACID transaction. Instead, you'll have an activity for booking rental cars that knows both how to perform a reservation and also how to cancel it - and one for a hotel and one for flights.

The activities are grouped in a composite job (routing slip) that's handed along the activity chain. If you want, you can sign/encrypt the routing slip items so that they can only be understood and manipulated by the intended receiver. When an activity completes, it adds a record of the completion to the routing slip along with information on where its compensating operation can be reached (e.g. via a Queue). When an activity fails, it cleans up locally and then sends the routing slip backwards to the last completed activity's compensation address to unwind the transaction outcome.

If you're a bit familiar with travel, you'll also notice that I've organized the steps by risk. Reserving a rental car almost always succeeds if you book in advance, because the rental car company can move more cars on-site of there is high demand. Reserving a hotel is slightly more risky, but you can commonly back out of a reservation without penalty until 24h before the stay. Airfare often comes with a refund restriction, so you'll want to do that last.

I created a Gist on Github that you can run as a console application. It illustrates this model in code. Mind that it is a mockup and not a framework. I wrote this in less than 90 minutes, so don't expect to reuse this.

The main program sets up an exemplary routing slip (all the classes are in the one file) and creates three completely independent "processes" (activity hosts) that are each responsible for handling a particular kind of work. The "processes" are linked by a "network" and each kind of activity has an address for forward progress work and one of compensation work. The network resolution is simulated by "Send".

```
[illegible]
```

The activities each implement a reservation step and an undo step. Here's the one for cars:

```
[illegible]
```

Clemens Vasters

```
[illegible]
```

The chaining happens solely through the routing slip. The routing slip is "serializable" (it's not, pretend that it is) and it's the only piece of information that flows between the collaborating activities. There is no central coordination. All work is local on the nodes and once a node is done, it either hands the routing slip forward (on success) or backward (on failure). For forward progress data, the routing slip has a queue and for backwards items it maintains a stack. The routing slip also handles resolving and invoking whatever the "next" thing to call is on the way forward and backward.

```
[illegible]
```

The local work and making the decisions is encapsulated in the ActivityHost, which calls ProcessNext() on the routing slip to resolve the next activity and call its DoWork() function on the way forward or will resolve the last executed activity on the way back and invoke its Compensate() function. Again, there's nothing centralized here; all that work hinges on the routing slip and the three activities and their execution is completely disjoint.

```
[illegible]
```

@berndruecker

Clemens Vasters
Architect at Microsoft

http://vasters.com/archive/Sagas.html

## Sagas

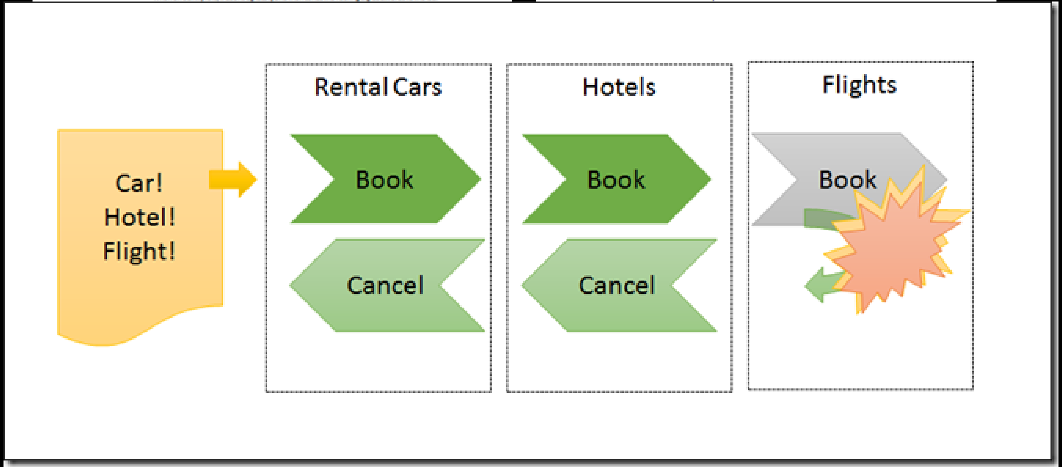
Today has been a lively day in some parts of the Twittersphere debating the Saga pattern. As it stands, there are a few frameworks for .NET out there that use the term "Saga" for some framework implementation of a state machine or workflow. Trouble is, that's not what a Saga is. A Saga is a failure management pattern.

Sagas come out of the realization that particularly long-lived transactions (originally even just inside databases), but also far distributed transactions across location and/or trust boundaries can't eaily be handled using the classic ACID model with 2-Phase commit and holding locks for the duration of the work. Instead, a Saga splits work into individual transactions whose effects can be, somehow, reversed after work has been performed and commited.

The picture shows a simple Saga. If you book a travel itinerary, you want a car and a hotel and a flight. If you can't get all of them, it's probably not worth going. It's also very certain that you can't enlist all of these providers into a distributed ACID transaction. Instead, you'll have an activity for booking rental cars that knows both how to perform a reservation and also how to cancel it - and one for a hotel and one for flights.

The activities are grouped in a composite job (routing slip) that's handed along the activity chain. If you want, you can sign/encrypt the routing slip items so that they can only be understood and manipulated by the intended receiver. When an activity completes, it adds a record of the completion to the routing slip along with information on where its compensating operation can be reached (e.g. via a Queue). When an activity fails, it cleans up locally and then sends the routing slip backwards to the last

The chaining happens solely through the routing slip. The routing slip is "serializable" (it's not, pretend that it is) and it's the only piece of information that flows between the collaborating activities. There is no central coordination. All work is local on the nodes and once a node is done, it either hands the routing slip forward (on success) or backward (on failure). For forward progress data, the routing slip has a queue and for backwards items it maintains a stack. The routing slip also handles resolving and invoking whatever the "next" thing to call is on the way forward and backward.



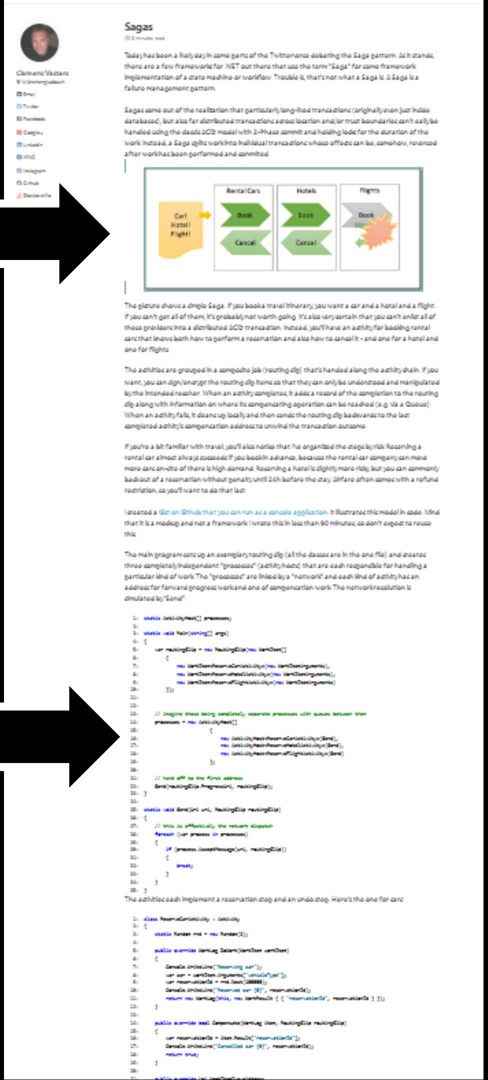
The activities each implement a reservation step and an undo step. Here's the one for cars:

The local work and making the decisions is encapsulated in the ActivityHost, which calls ProcessNext() on the routing slip to resolve the next activity and call its DoWork() function on the way forward or will resolve the last executed activity on the way back and invoke its Compensate() function. Again, there's nothing centralized here; all that work hinges on the routing slip and the three activities and their execution is completely disjoint.

@berndruecker

Clemens Vasters
Architect at Microsoft

http://vasters.com/archive/Sagas.html



## Sagas

3 minutes read

Today has been a lively day in some parts of the Twittersphere debating the Saga pattern. As it stands, there are a few frameworks for .NET out there that use the term "Saga" for some framework implementation of a state machine or workflow. Trouble is, that's not what a Saga is. A Saga is a failure management pattern.

Sagas come out of the realization that particularly long-lived transactions (originally even just inside databases), but also far distributed transactions across location and/or trust boundaries can't easily be handled using the classic ACID model with 2-Phase commit and holding locks for the duration of the work. Instead, a Saga splits work into individual transactions whose effects can be, somehow, reversed after work has been performed and committed.

The picture shows a simple Saga. If you book a travel itinerary, you want a car and a hotel and a flight. If you can't get all of them, it's probably not worth going. It's also very certain that you can't enlist all of these providers into a distributed ACID transaction. Instead, you'll have an activity for booking rental cars that knows both how to perform a reservation and also how to cancel it - and one for a hotel and one for flights.

The activities are grouped in a composite job (routing slip) that's handed along the activity chain. If you want, you can sign/encrypt the routing slip items so that they can only be understood and manipulated by the intended receiver. When an activity completes, it adds a record of the completion to the routing slip along with information on where its compensating operation can be reached (e.g. via a Queue). When an activity fails, it cleans up locally and then sends the routing slip backwards to the last completed activity's compensation address to unwind the transaction outcome.

If you're a bit familiar with travel, you'll also notice that I've organized the steps by risk. Reserving a rental car almost always succeeds if you book in advance, because the rental car company can move more cars on-site of there is high demand. Reserving a hotel is slightly more risky, but you can commonly back out of a reservation without penalty until 24h before the stay. Airfare often comes with a refund restriction, so you'll want to do that last.

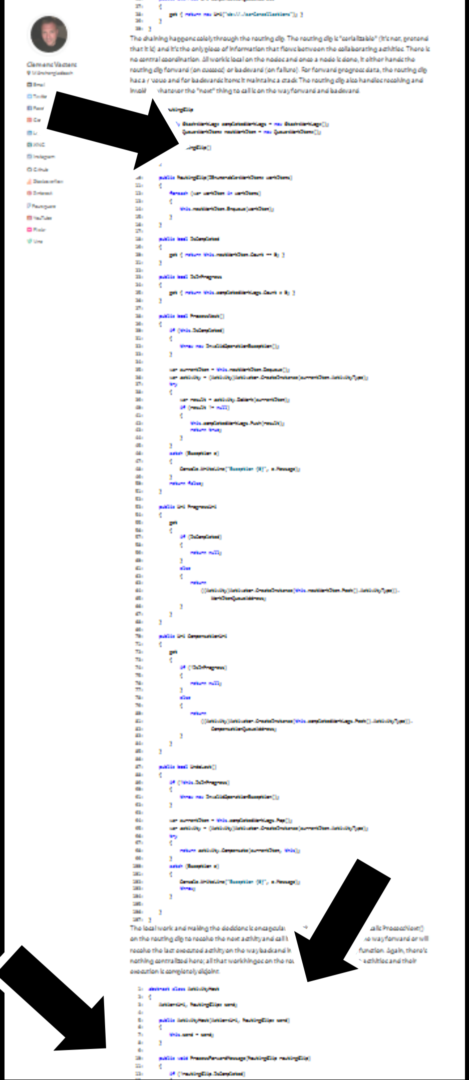
I created a Gist on Github that you can run as a console application. It illustrates this model in code. Mind that it is a mockup and not a framework. I wrote this in less than 90 minutes, so don't expect to reuse this.

The main program sets up an exemplary routing slip (all the classes are in the one file) and creates three completely independent "processes" (activity hosts) that are each responsible for handling a particular kind of work. The "processes" are linked by a "network" and each kind of activity has an address for forward progress work and one of compensation work. The network resolution is simulated by "Send".

```
static ActivityHost[] processes;

static void Main(string[] args)
{
    var routingSlip = new RoutingSlip(new WorkItem[]
[illegible]
```

The activities each implement a reservation step and an undo step. Here's the one for cars:

```
class ReserveCarActivity : Activity
{
[illegible]
```



The chaining happens solely through the routing slip. The routing slip is "serializable" (it's not, pretend that it is) and it's the only piece of information that flows between the collaborating activities. There is no central coordination. All work is local on the nodes and once a node is done, it either hands the routing slip forward (on success) or backward (on failure). For forward progress data, the routing slip has a queue and for backwards items it maintains a stack. The routing slip also handles resolving and invoking whatever the "next" thing to call is on the way forward and backward.

```
[illegible]
```

The local work and making the decisions is encapsulated [illegible] calls ProcessNext() on the routing slip to resolve the next activity and call [illegible] the way forward or will resolve the last executed activity on the way back and in[illegible] function. Again, there's nothing centralized here; all that work hinges on the ro[illegible] activities and their execution is completely disjoint.

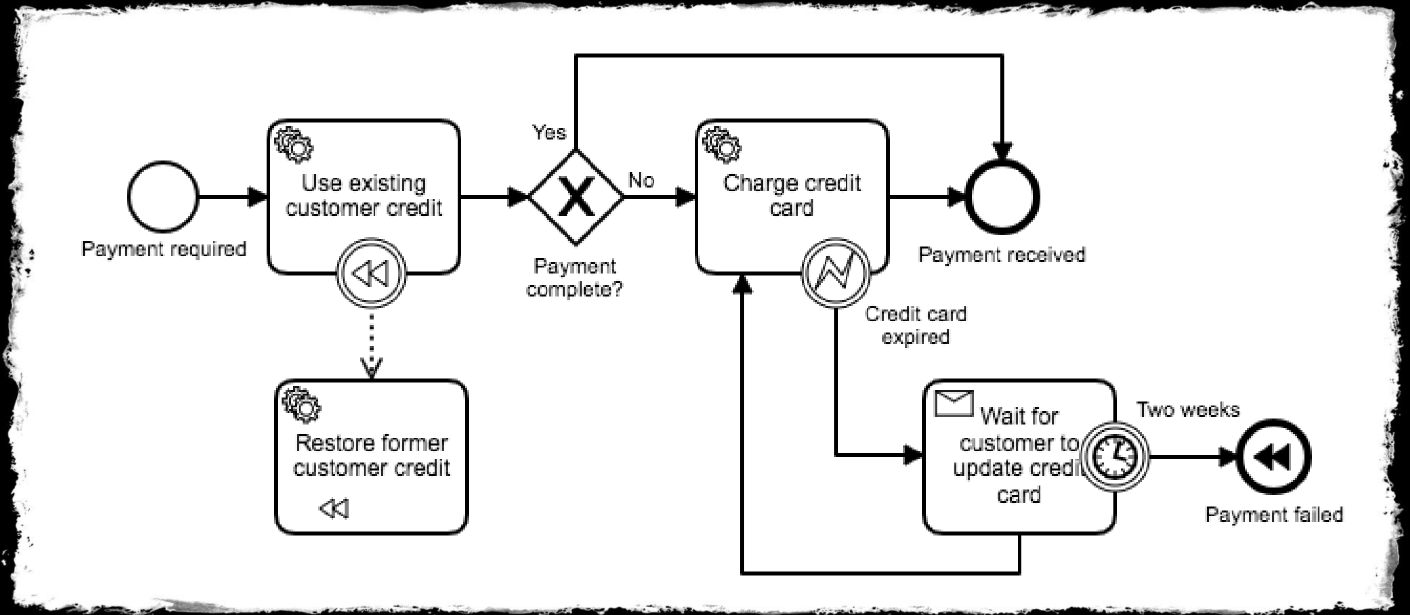
```
abstract class ActivityHost
{
[illegible]
```

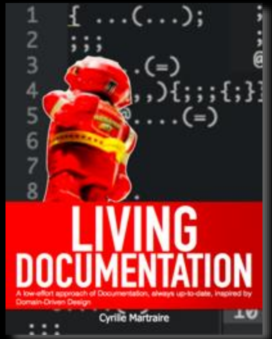
# BPMN

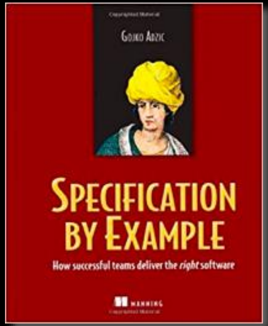
Business Process Model and Notation

ISO Standard

# Living documentation for long-running behaviour

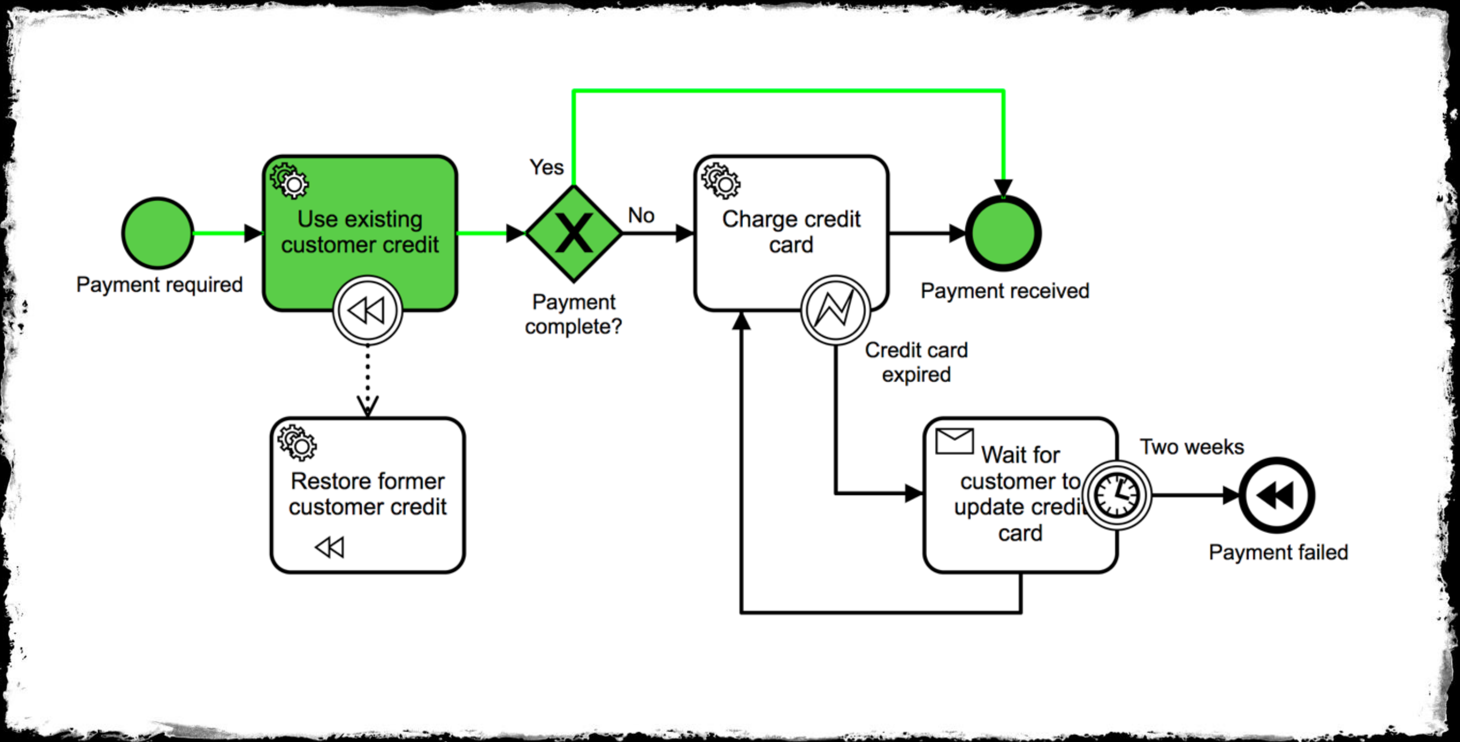




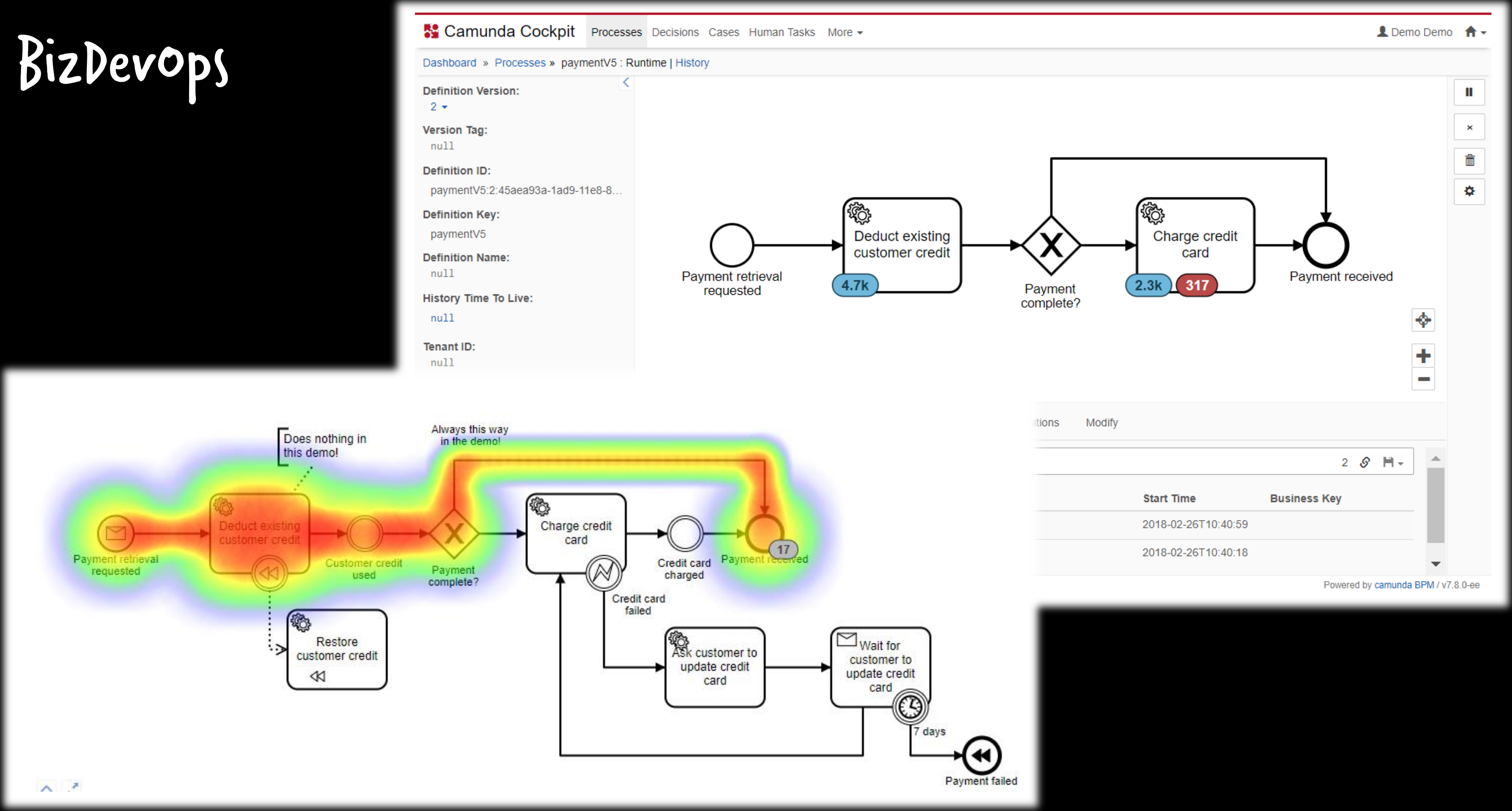


@berndruecker

# Visual HTML reports for test cases



# BizDevops



Camunda Cockpit

Dashboard » Processes » paymentV5 : Runtime | History

**Definition Version:**
2

**Version Tag:**
null

**Definition ID:**
paymentV5:2:45aea93a-1ad9-11e8-8…

**Definition Key:**
paymentV5

**Definition Name:**
null

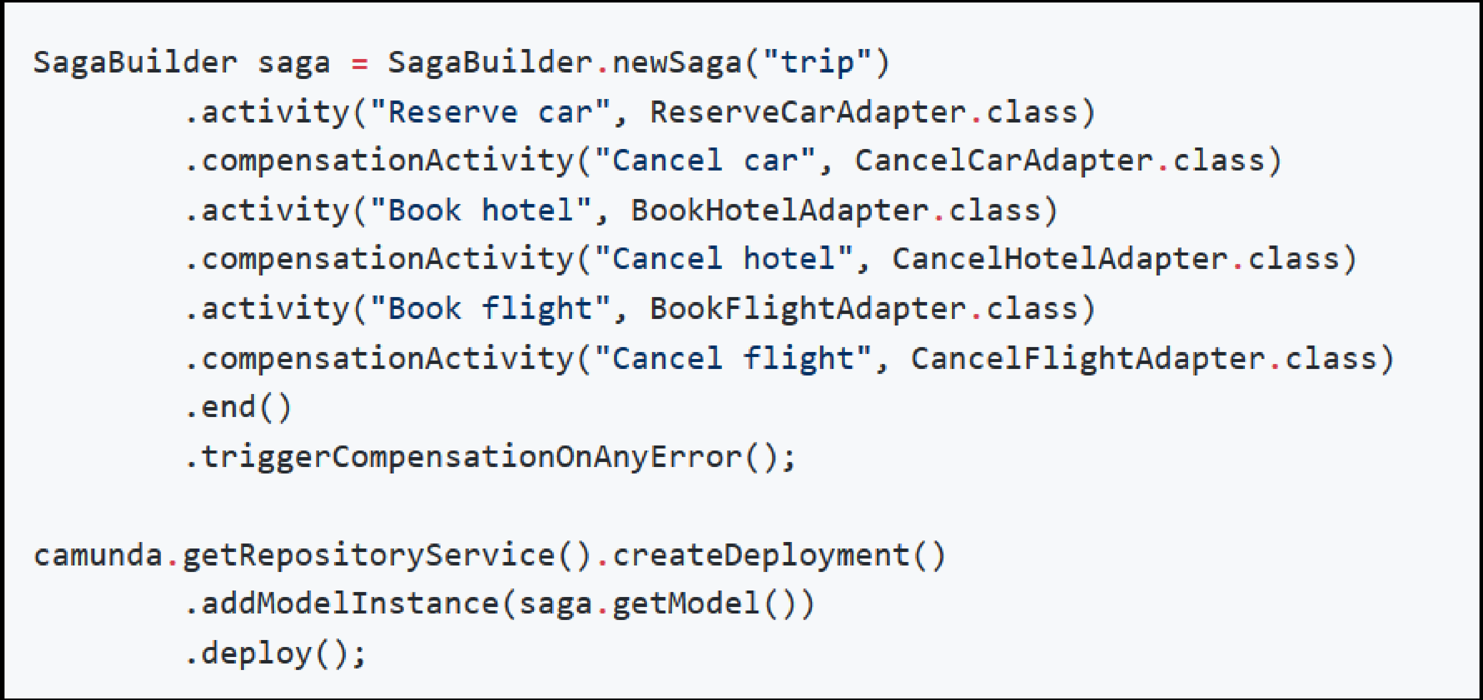
**History Time To Live:**
null

**Tenant ID:**
null

| Start Time | Business Key |
|---|---|
| 2018-02-26T10:40:59 | |
| 2018-02-26T10:40:18 | |

Powered by camunda BPM / v7.8.0-ee

# Fancy a DSL? Just do it!



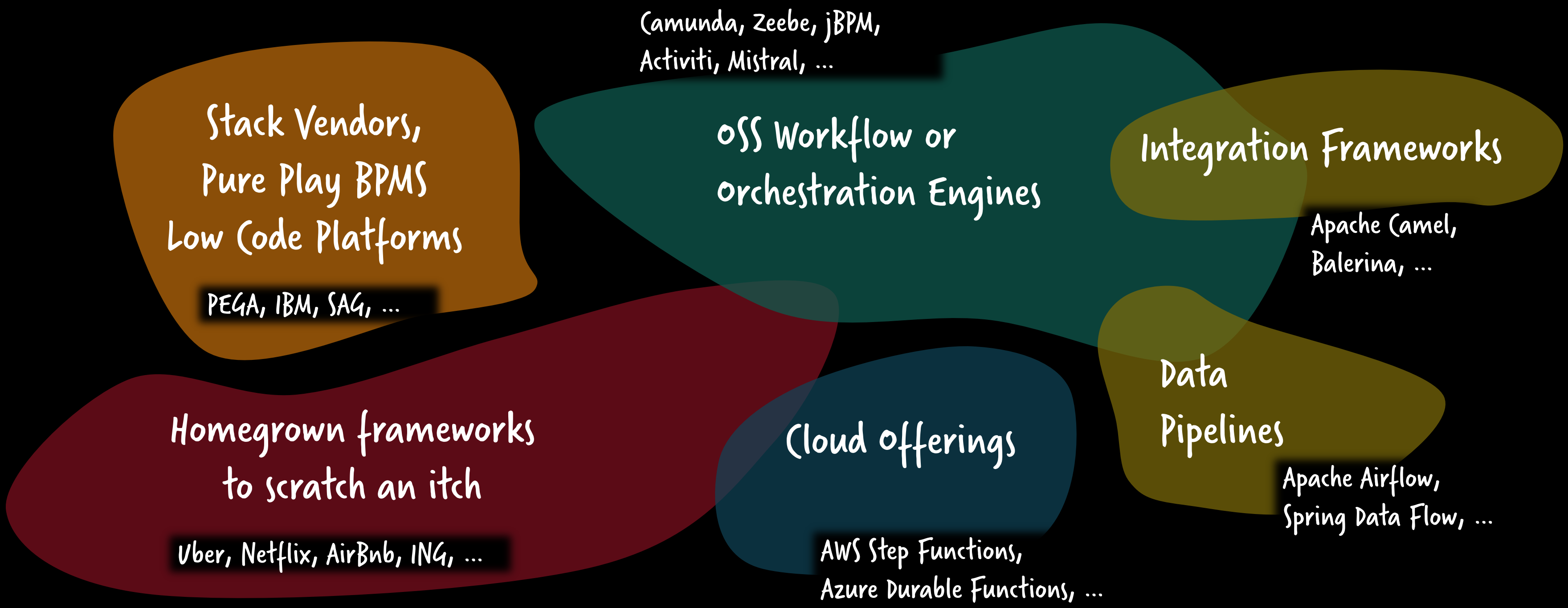
```java
SagaBuilder saga = SagaBuilder.newSaga("trip")
        .activity("Reserve car", ReserveCarAdapter.class)
        .compensationActivity("Cancel car", CancelCarAdapter.class)
        .activity("Book hotel", BookHotelAdapter.class)
        .compensationActivity("Cancel hotel", CancelHotelAdapter.class)
        .activity("Book flight", BookFlightAdapter.class)
        .compensationActivity("Cancel flight", CancelFlightAdapter.class)
        .end()
        .triggerCompensationOnAnyError();

camunda.getRepositoryService().createDeployment()
        .addModelInstance(saga.getModel())
        .deploy();
```

The visual get auto-generated...

https://github.com/berndruecker/flowing-trip-booking-saga

# Thoughts on the state machine | workflow engine market

# Thoughts on the state machine | workflow engine market



Camunda, Zeebe, jBPM,
Activiti, Mistral, ...

Stack Vendors,
Pure Play BPMS
Low Code Platforms

PEGA, IBM, SAG, ...

OSS Workflow or
Orchestration Engines

Integration Frameworks

Apache Camel,
Balerina, ...

Homegrown frameworks
to scratch an itch

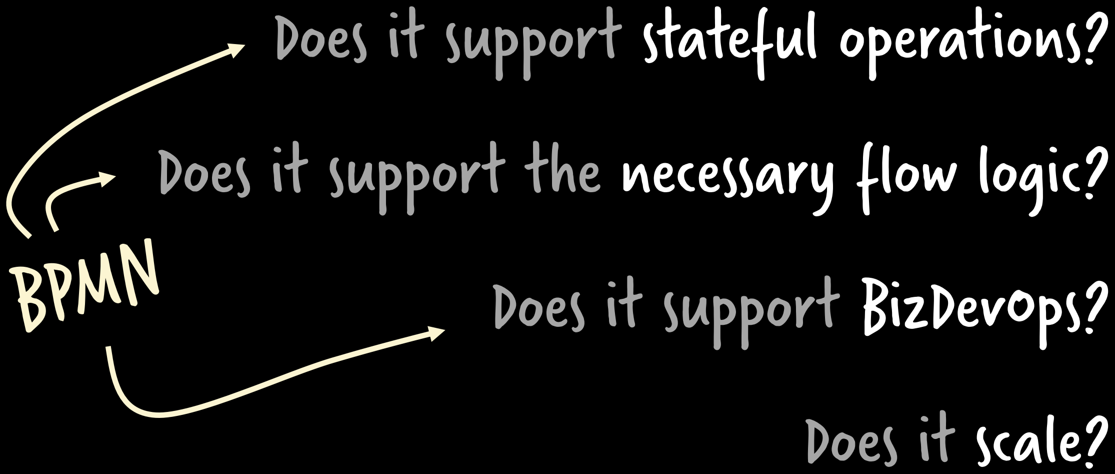
Uber, Netflix, AirBnb, ING, ...

Cloud offerings

AWS Step Functions,
Azure Durable Functions, ...

Data
Pipelines

Apache Airflow,
Spring Data Flow, ...



BPMN

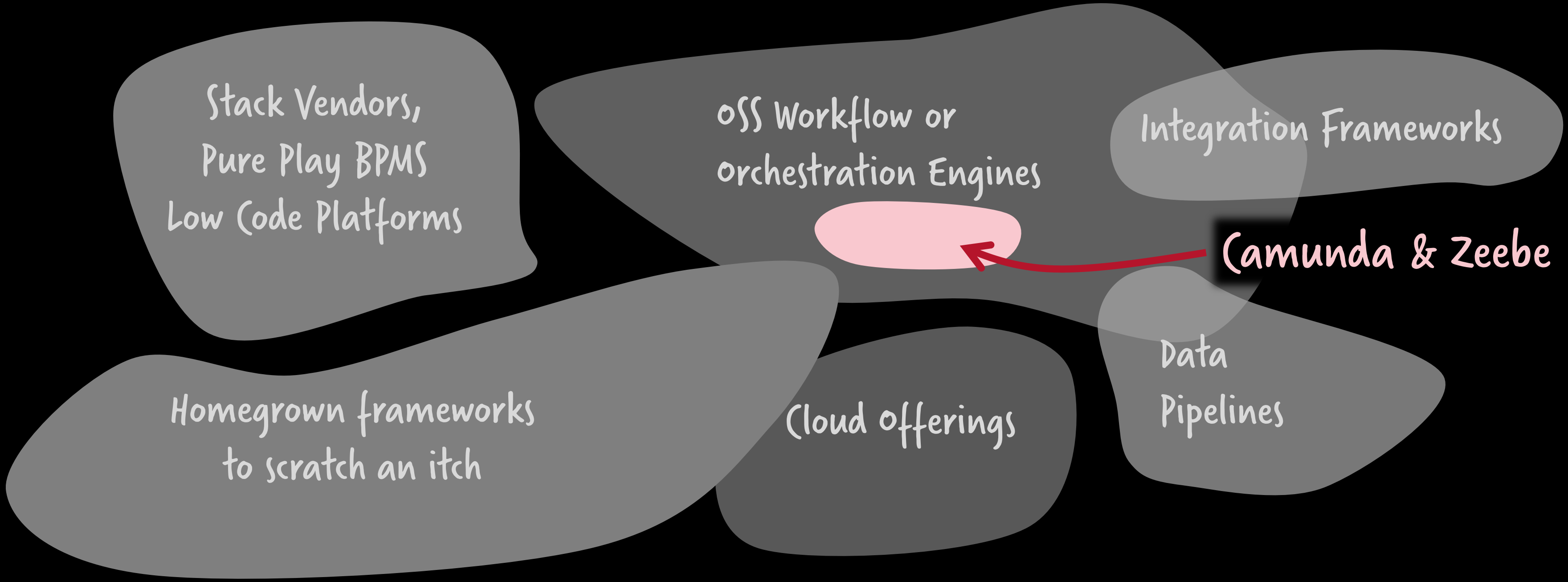
Does it support stateful operations?

Does it support the necessary flow logic?

Does it support BizDevops?

Does it scale?

# My personal pro-tip for a shortlist ;-)



Stack Vendors,
Pure Play BPMS
Low Code Platforms

OSS Workflow or
orchestration Engines

Integration Frameworks

Camunda & Zeebe

Homegrown frameworks
to scratch an itch

Cloud offerings

Data
Pipelines

# Recap

- Aggregates = Consistency boundaries
- Grown ups don't use distributed transactions but eventual consistency
- Idempotency is super important
- Some consistency challenges require state
  - Stateful retry & cleanup
  - Saga / Compensation

Thank you!

| | |
|---|---|
| Contact: | mail@berndruecker.io<br>@berndruecker |
| Slides: | https://berndruecker.io |
| Blog: | https://medium.com/berndruecker |
| Code: | https://github.com/berndruecker |

| | |
|---|---|
| InfoWorld FROM IDG | https://www.infoworld.com/article/3254777/application-development/3-common-pitfalls-of-microservices-integrationand-how-to-avoid-them.html |
| InfoQueue | https://www.infoq.com/articles/events-workflow-automation |
| THENEWSTACK | https://thenewstack.io/5-workflow-automation-use-cases-you-might-not-have-considered/ |