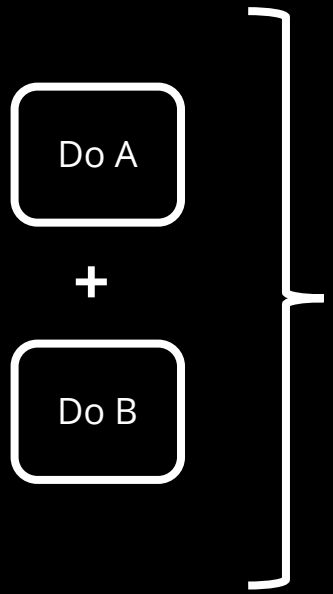
Lost in transaction?

Strategies to deal with
(in)consistency in distributed systems

@berndruecker



All or nothing

Once upon a time:

```
try {
 tx.begin();
 doA();
 doB();
 tx.commit();
} catch (Exception e) {
 tx.rollback();
}
```

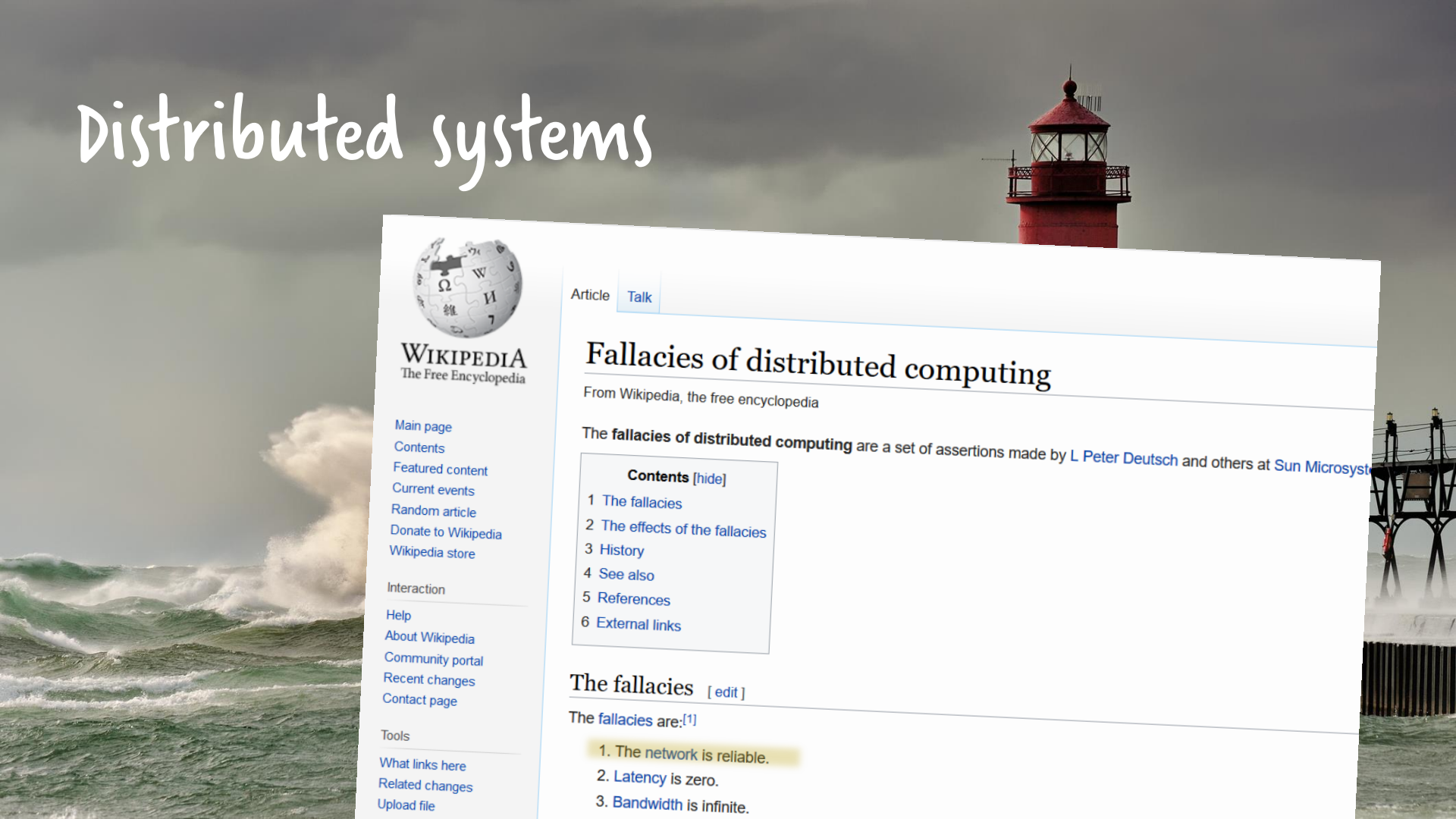
or simply:

```
@Transactional
public void createCustomer(Customer cust) {
  // ...
}
```

Atomicity
Consistency
Isolation
Durability

Distributed systems

Distributed systems



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

What links here
Related changes
Upload file

Article Talk

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]

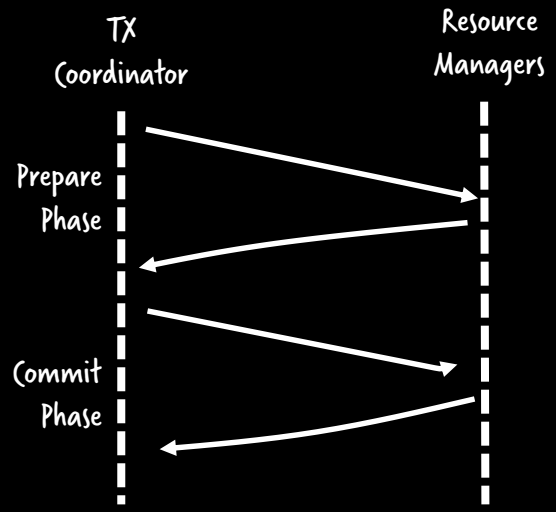
The fallacies [edit]

The fallacies are:[1]

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

Distributed systems

But 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

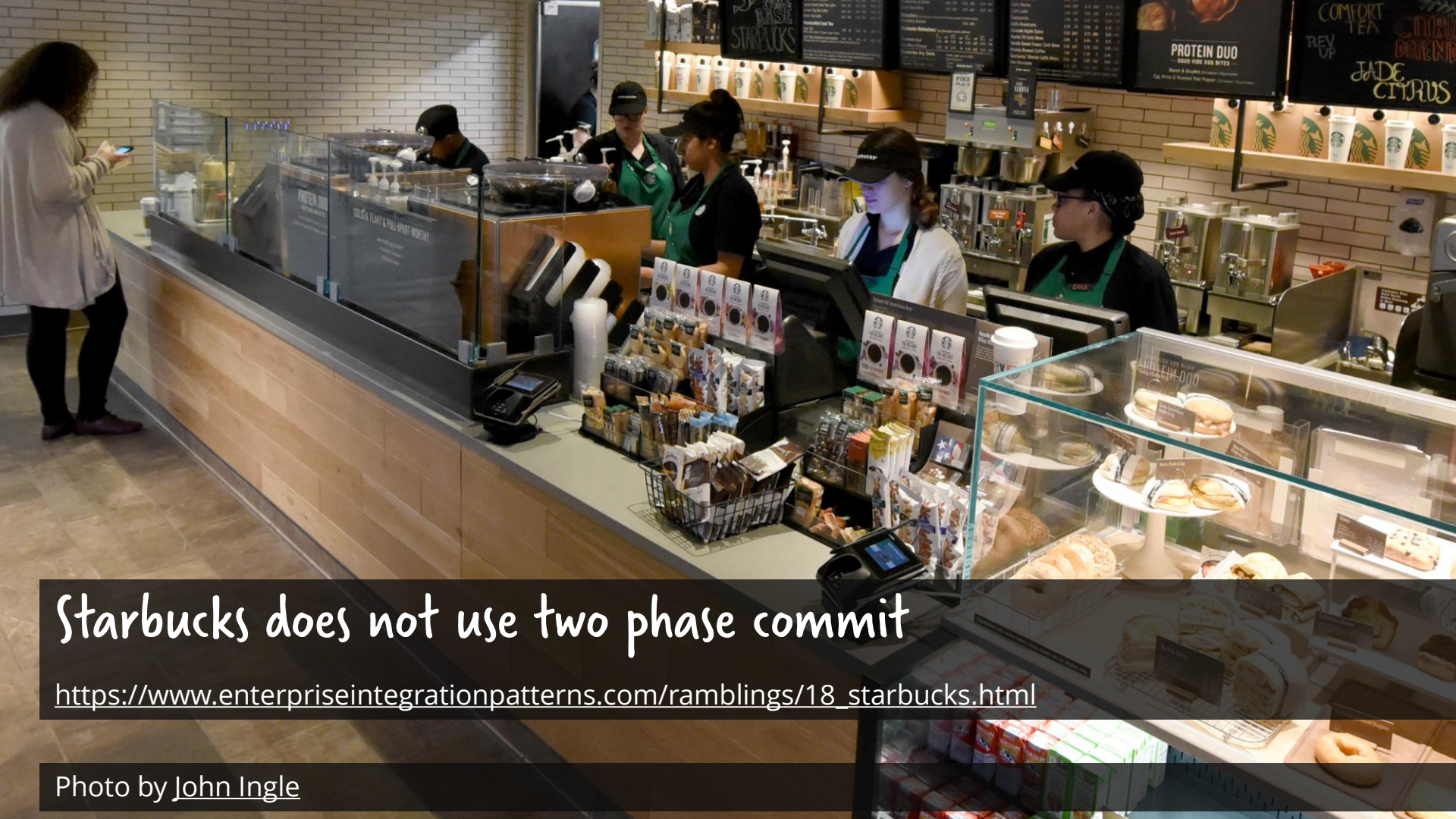
Instead, applications are built using different techniques which do not provide the same transactional guarantees but still meet the needs of their businesses.

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.

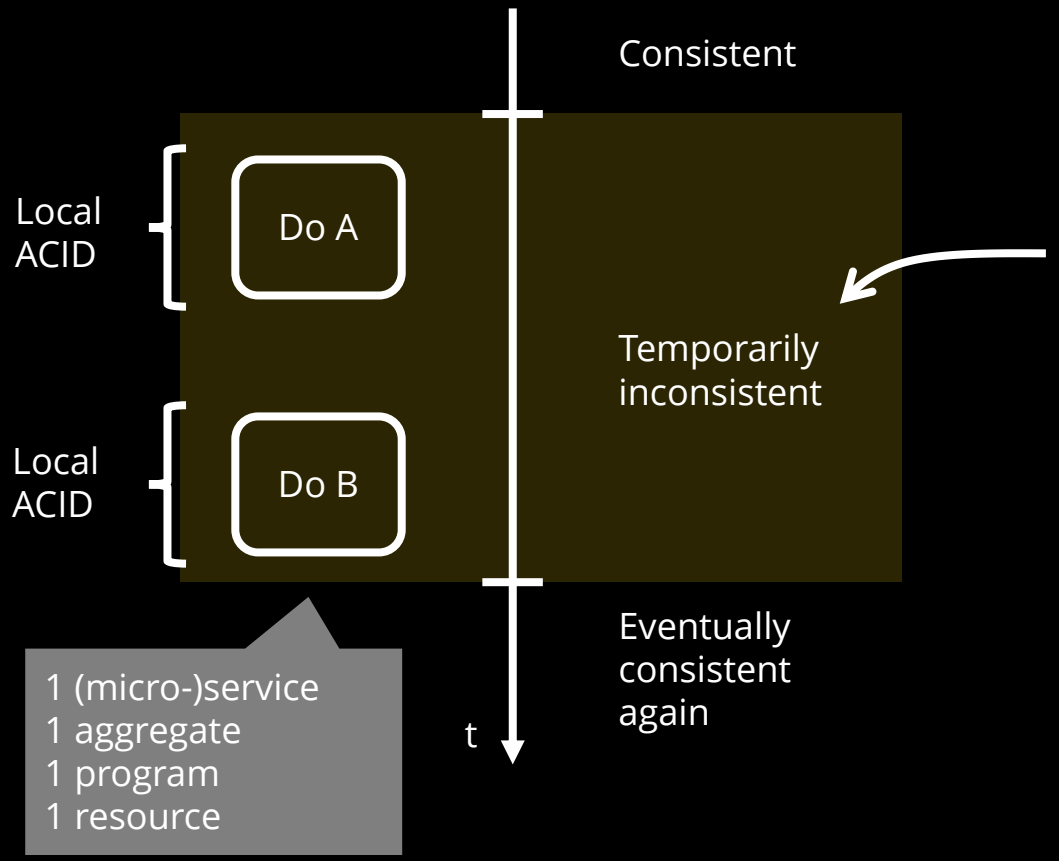
BASE:

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

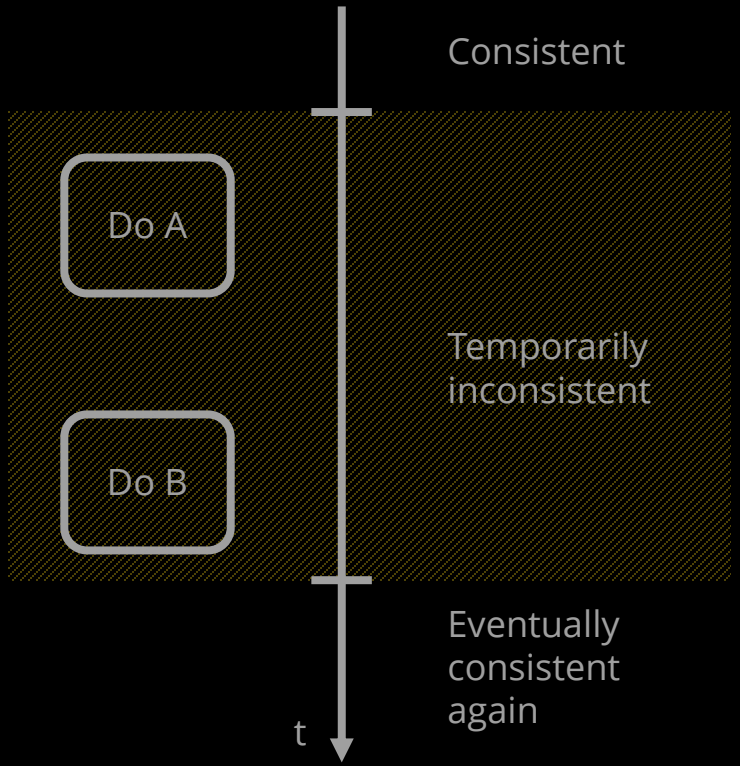
PODC Keynote, July 19, 2000

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

That means



Violates „I" of ACID



You might know this from:

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

Pat Helland

„Building on Quicksand“ Paper

ACID 2.0

Pat Helland

„Building on Quicksand“ Paper

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

Commutative $a + b = b + a$

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

Distributed

2.0

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!

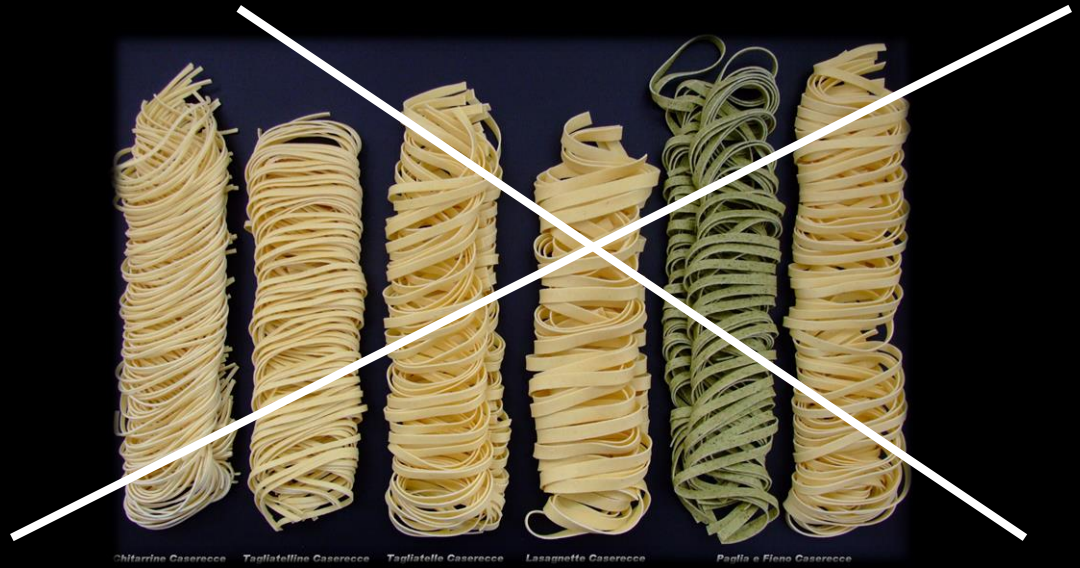
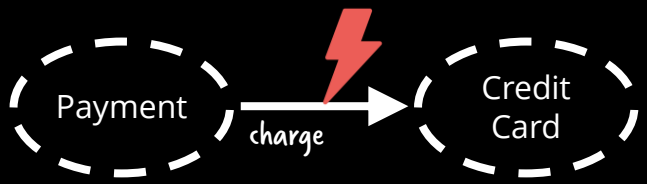
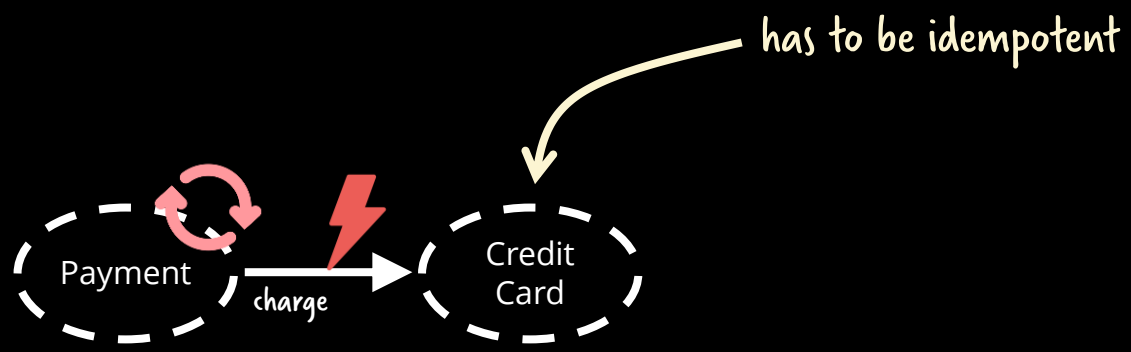


Photo by Chr.Späth, available under Public Domain.

Example



Strategy: retry



Charge Credit Card
cardNumber
amount

Not idempotent

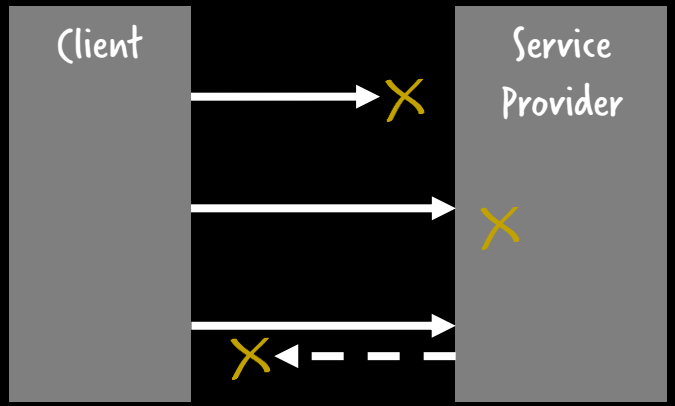
Charge Credit Card
cardNumber
amount
transactionId

Idempotent

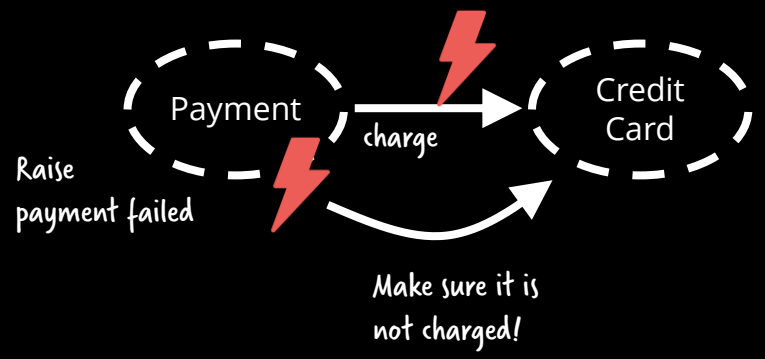
Distributed

It is impossible to differentiate certain failure scenarios:

Independant of communication style!



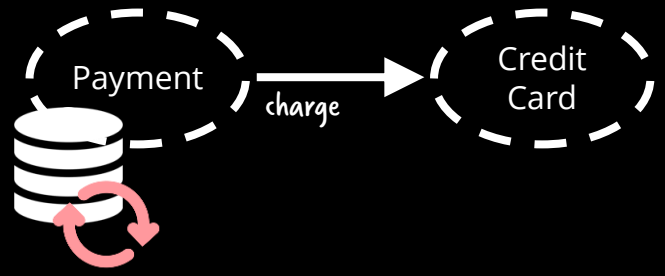
Strategy: Cleanup



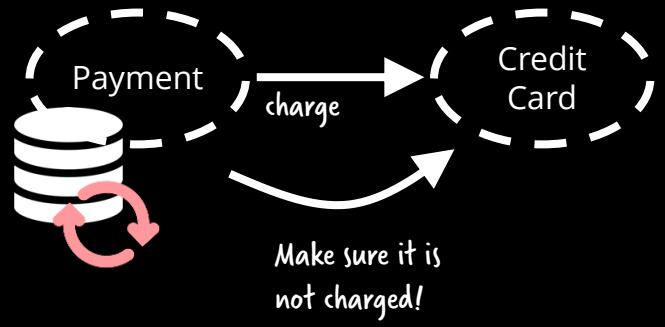
<u>Cancel charge</u>
cardNumber
amount
transactionId

Some communication challenges require state.

Strategy: Stateful retry



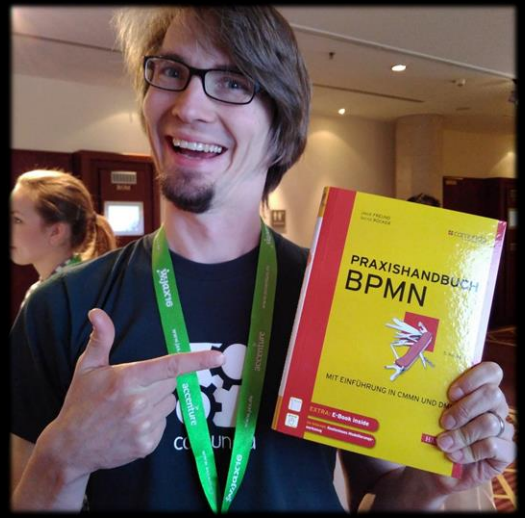
Strategy: Stateful retry



Warning:
Contains Opinion

Bernd Ruecker

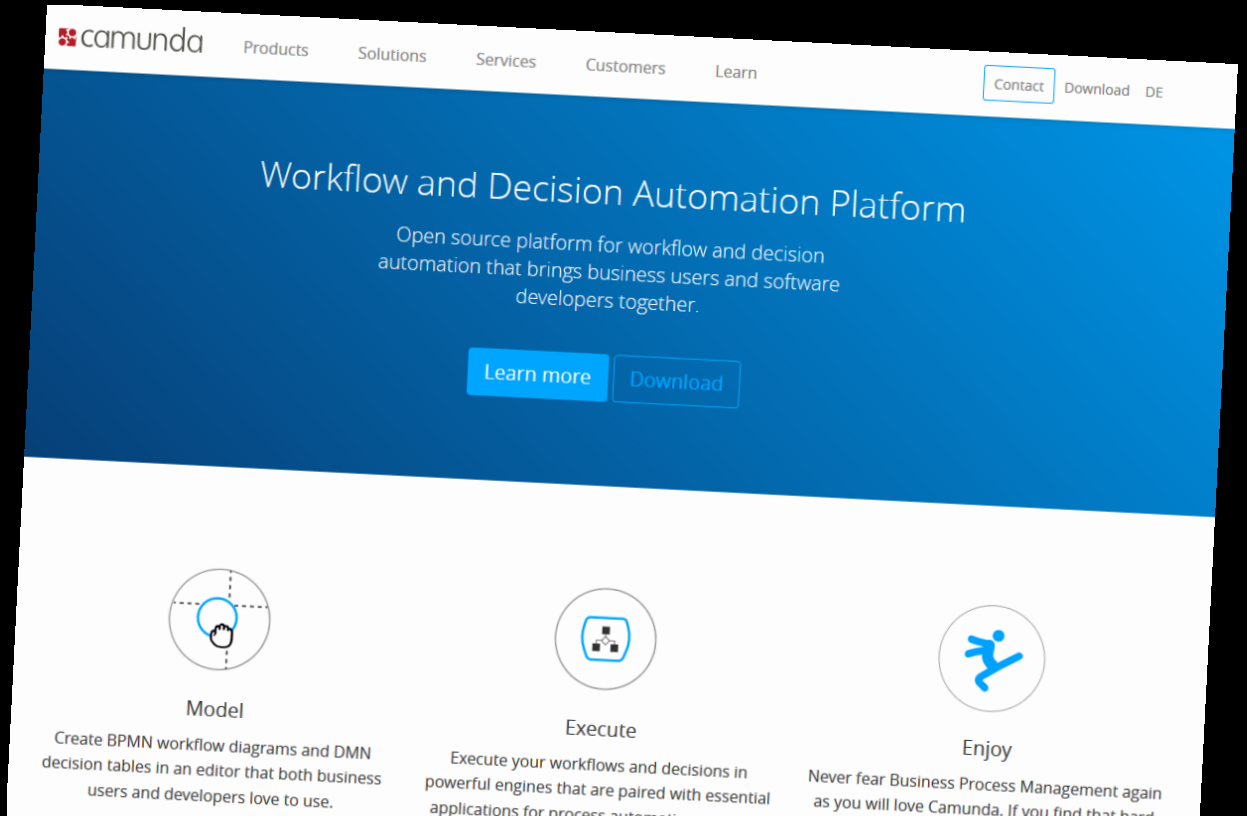
Co-founder and
Chief Technologist of
Camunda



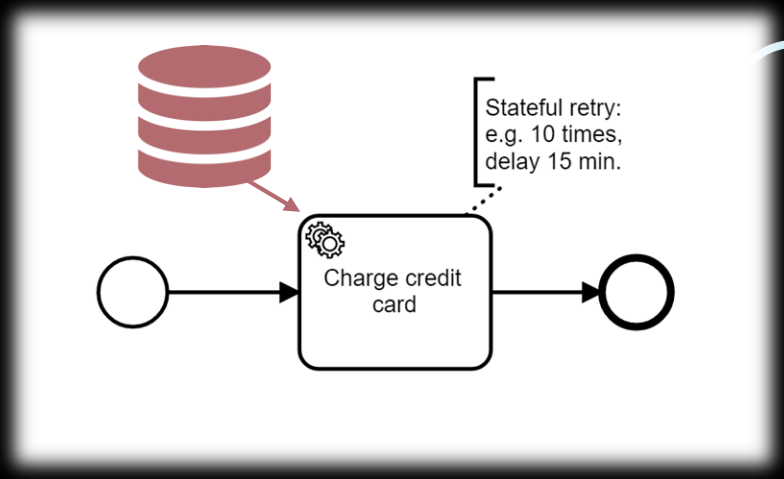
Berlin, Germany

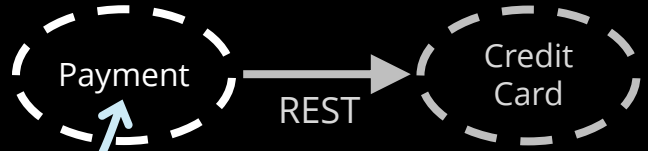
bernd.ruecker@camunda.com
@berndruecker

Let's use a lightweight OSS workflow engine for this:

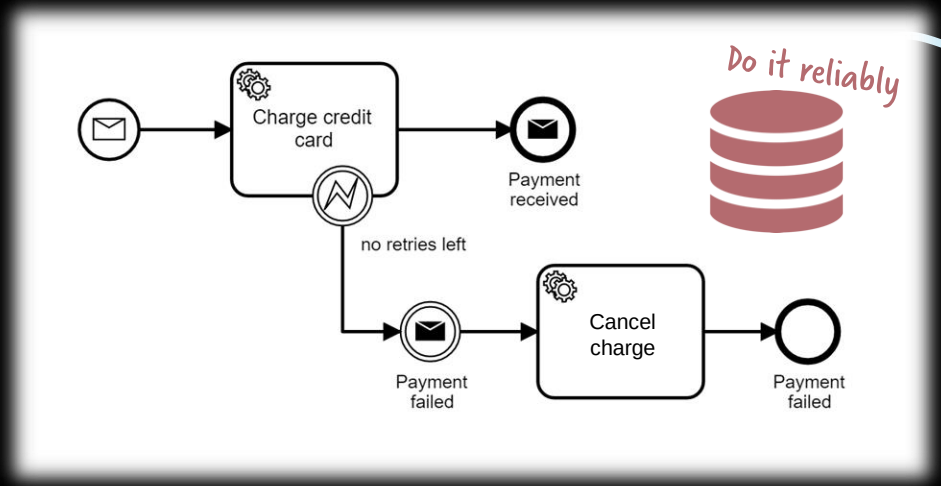


Stateful retry

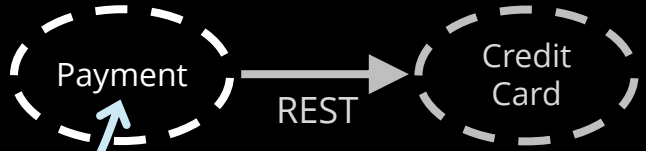




Stateful retry & cleanup



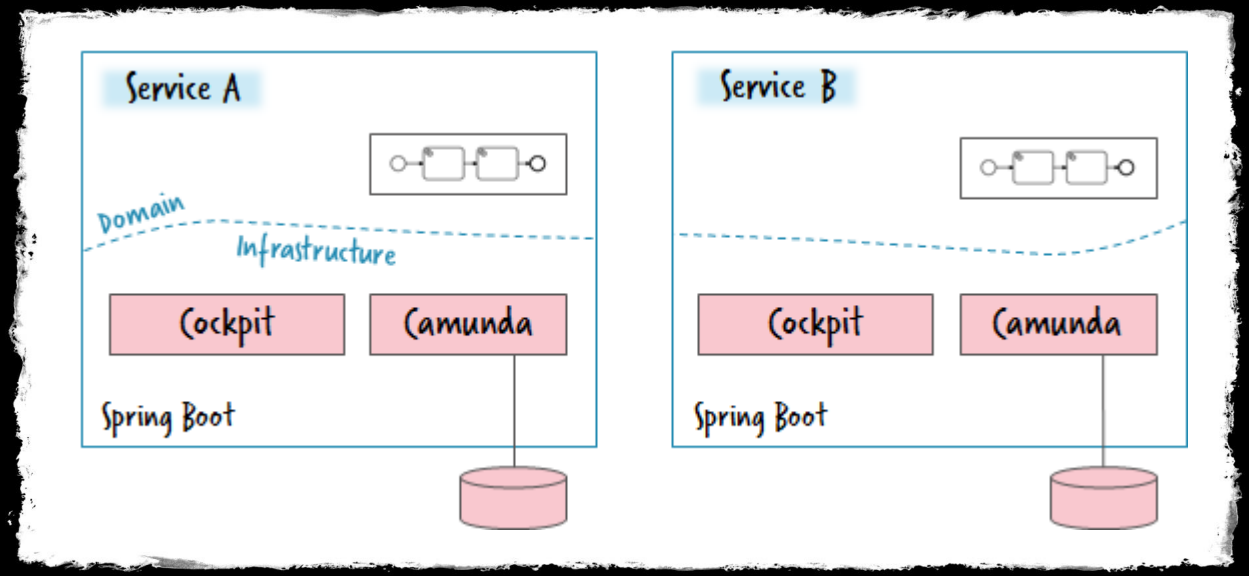
Do it reliably



Live hacking

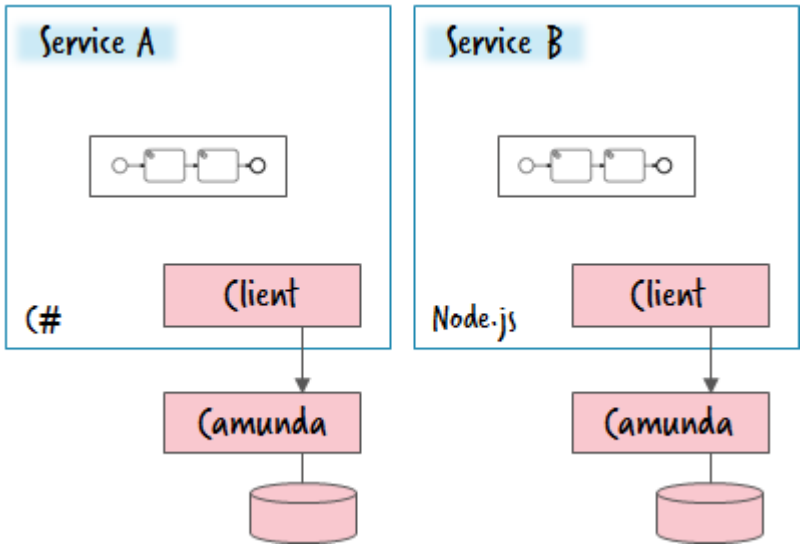
https://github.com/flowing/flowing-retail/tree/master/rest

Embedded Engine Example (Java)



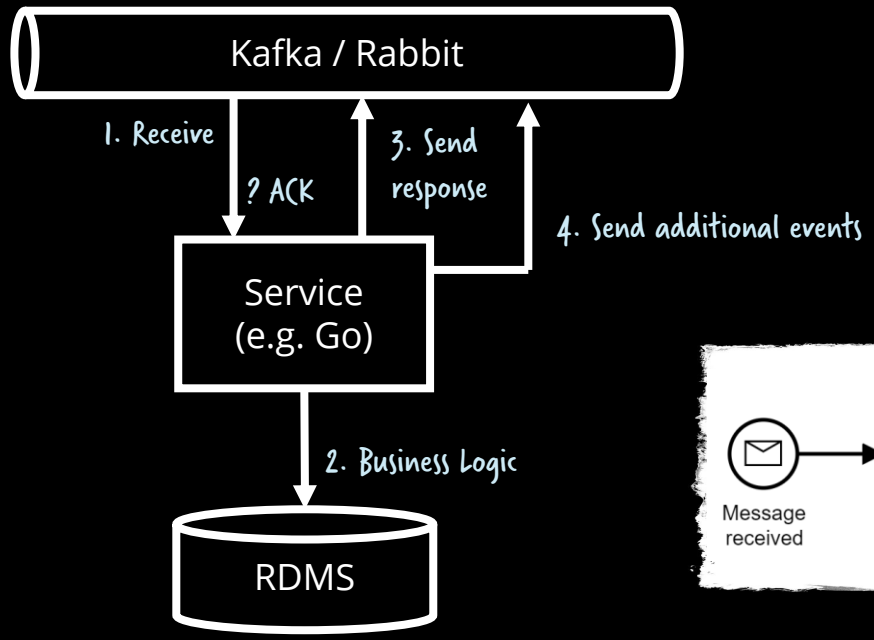
https://blog.bernd-ruecker.com/architecture-options-to-run-a-workflow-engine-6c2419902d91

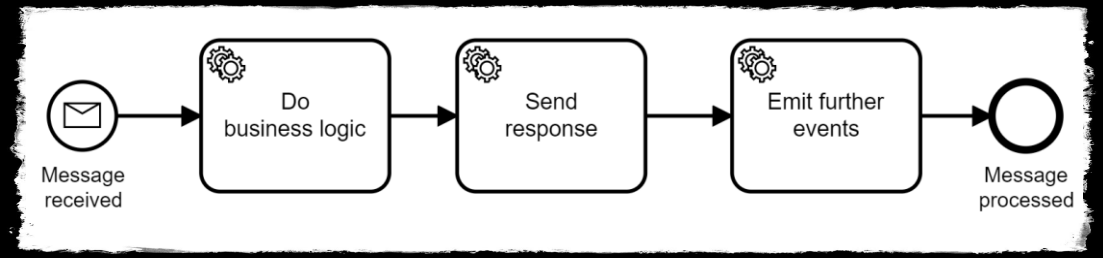
Remote Engine Example (Polyglot)



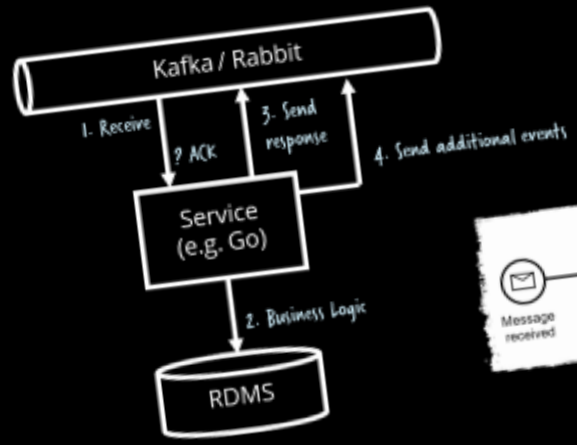
https://blog.bernd-ruecker.com/architecture-options-to-run-a-workflow-engine-6c2419902d91

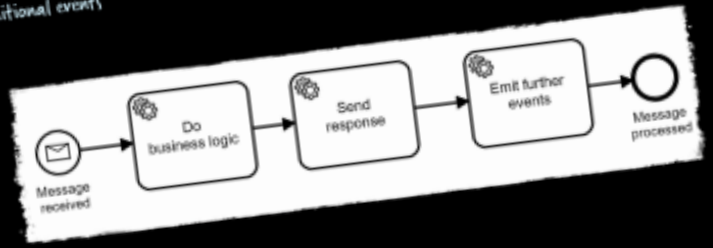
A relatively common pattern





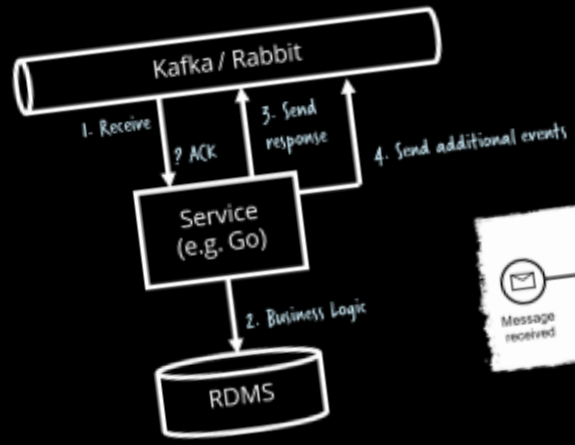
State can solve important basic problems

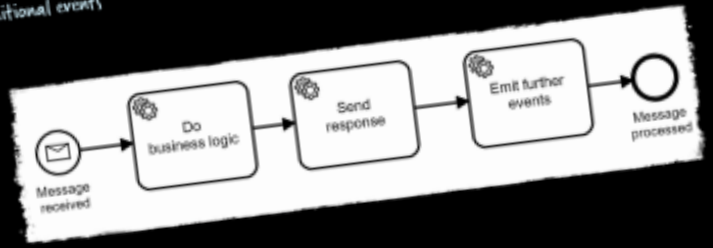




„Can this handle 15k requests per second?"

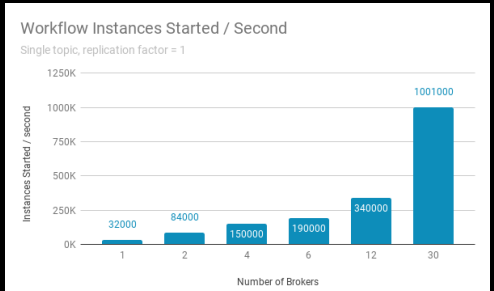
State can solve important basic problems

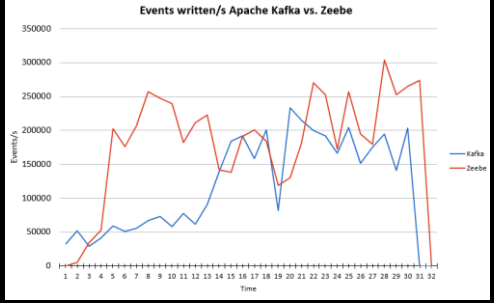




„Yes."

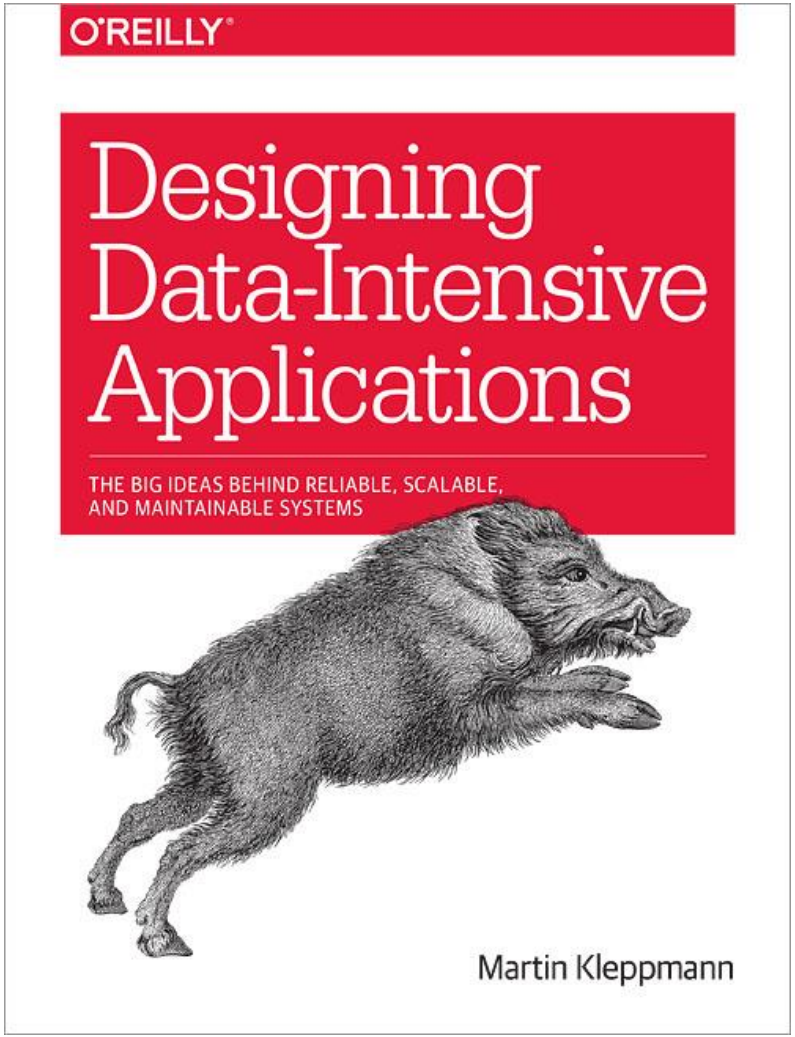
zeebe by Camunda

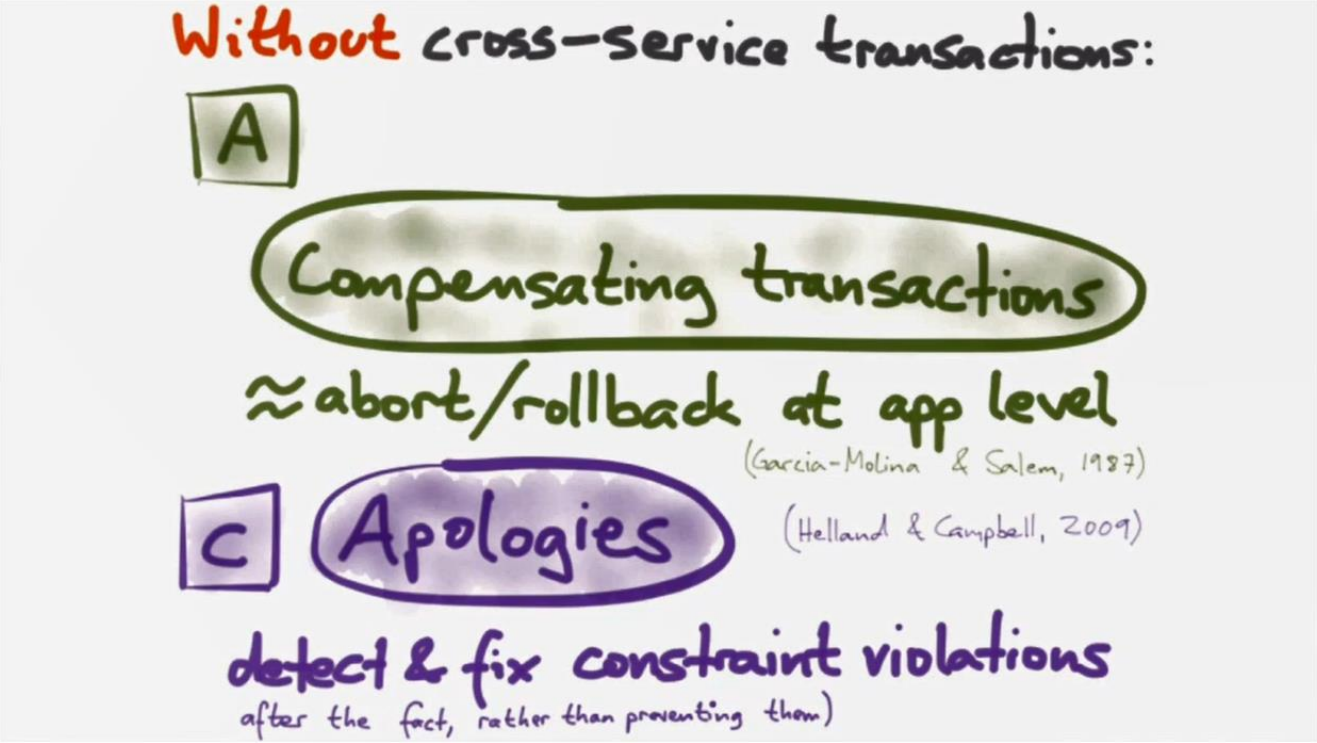




Sept 25-26, 2015

thestrangeloop.com





strangeloop

Sept 25-26, 2015

thestrangeloop.com

Microsoft

PatHelland's WebLog

Memories, Guesses, and Apologies

Rate this article ★★★★★

Pat Helland May 15, 2007

Share 3 12 0 5

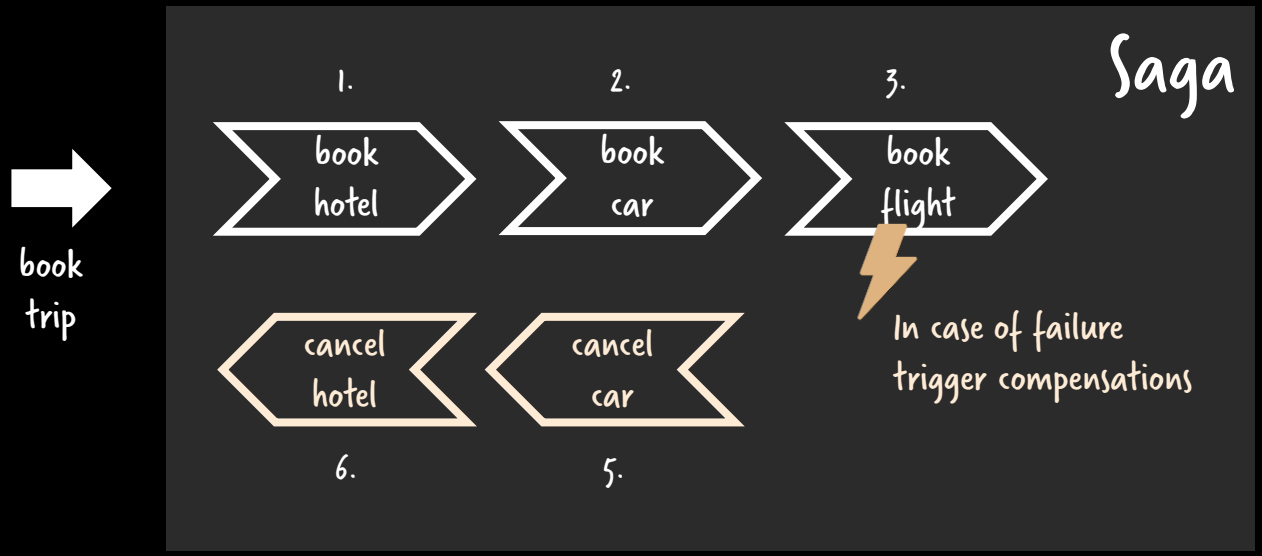
Well, here I am blogging on the bus with my newly installed Windows Live Writer!!!

This blog is a text version of a five minute "Gong Show" presentation I did at CIDR (Conference on Innovative Database Research) on Jan 8,2007.

All computing can be considered as "M

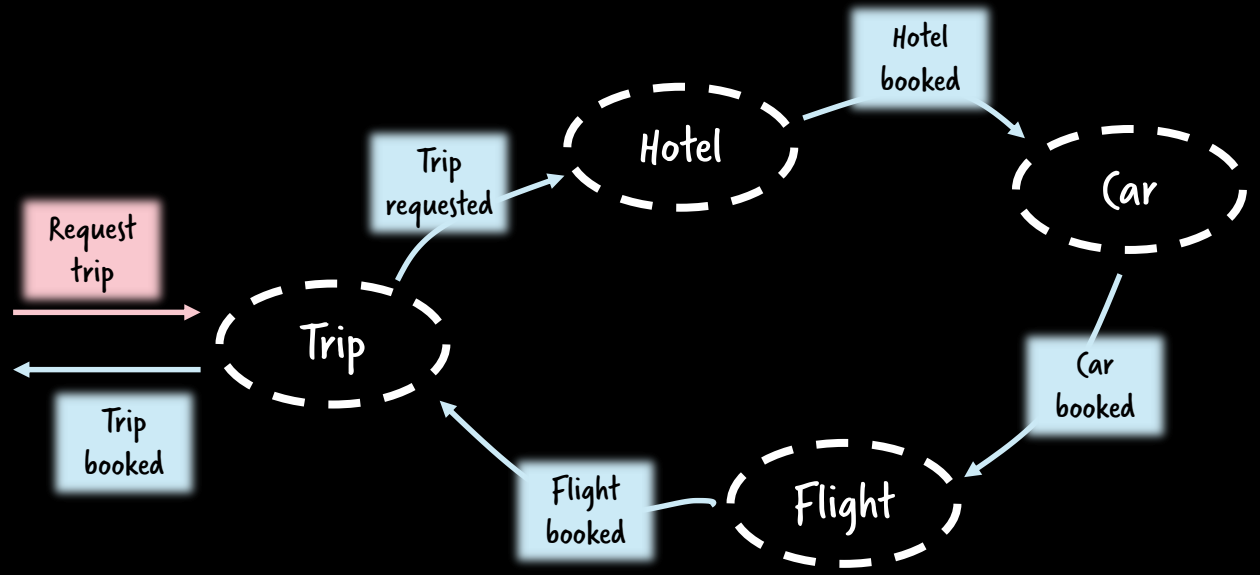
https://blogs.msdn.microsoft.com/pathelland/2007/05/15/memories-guesses-and-apologies/

Compensation – the classical example

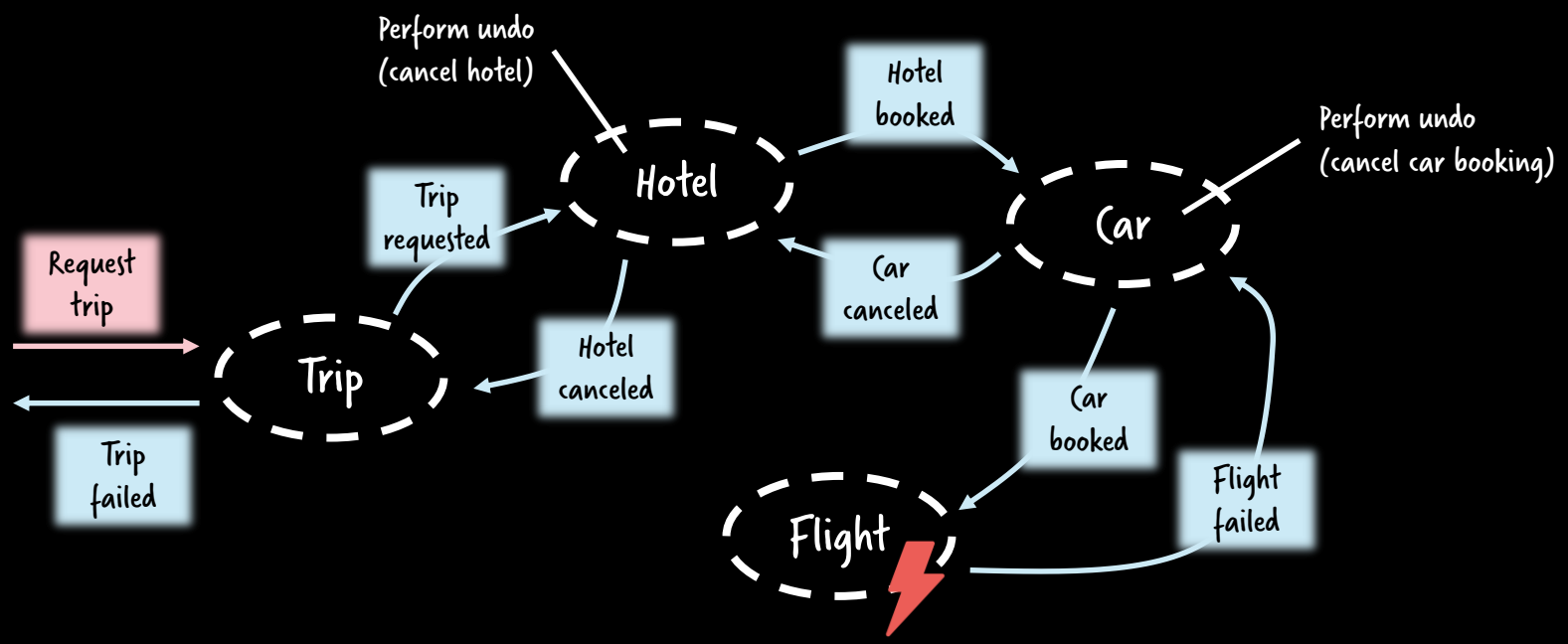


2 alterntive approaches: choreography & orchestration

Event-driven choreography



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

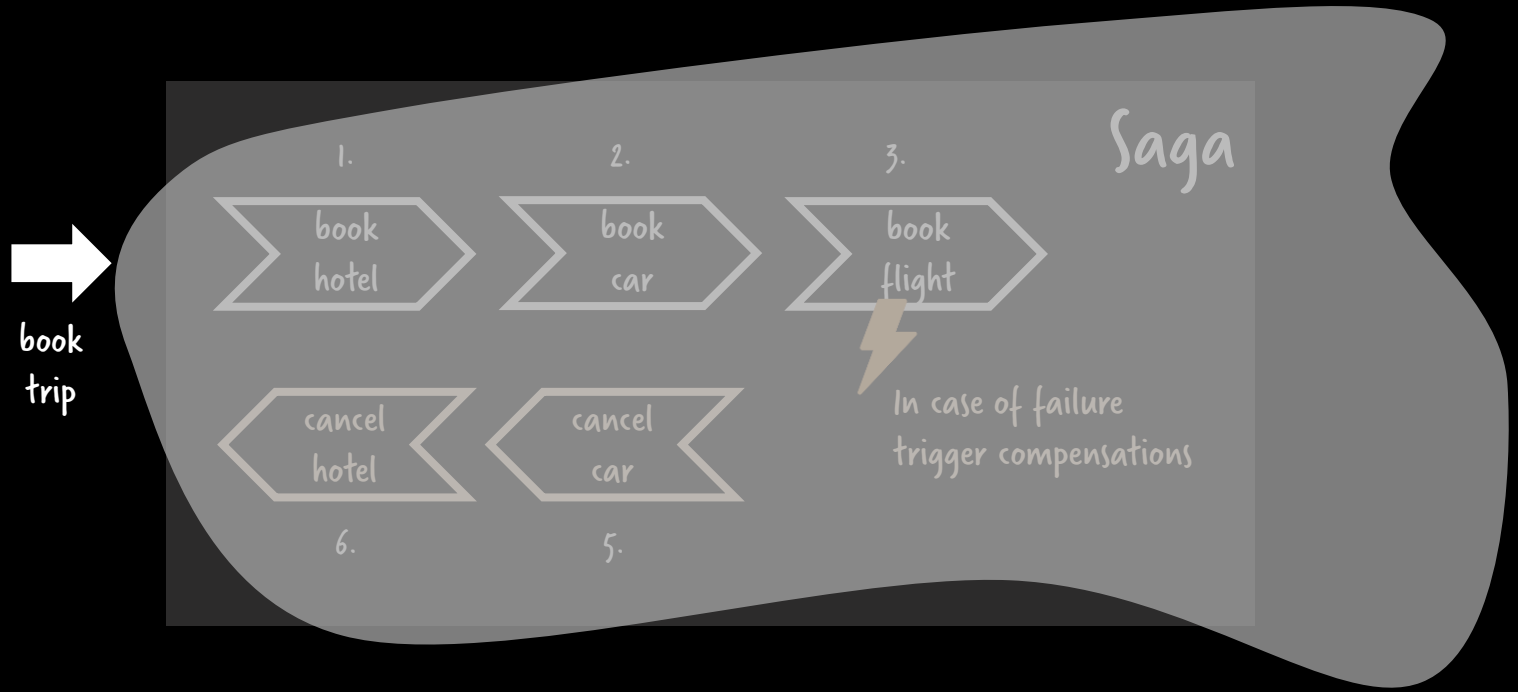
The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow, and thus set yourself up for trouble in future years.

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

Classical example



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/

Microservice pioneers have become aware

Traditionally, some of these processes had been orchestrated in an ad-hoc manner using a combination of pub/sub, making direct REST calls, and using a database to manage the state. However, as the number of microservices grow and the complexity of the processes increases, getting visibility into these distributed workflows becomes difficult without a central orchestrator.

Netflix Technology Blog Follow
Learn more about how Netflix designs, builds, and operates our systems and engineering organizations
Dec 12, 2016 · 7 min read

Netflix Conductor: A microservices orchestrator

The Netflix Content Platform Engineering team runs a number of business processes which are driven by asynchronous orchestration of tasks executing on microservices. Some of these are long running processes spanning several days. These processes play a critical role in getting titles ready for streaming to our viewers across the globe.

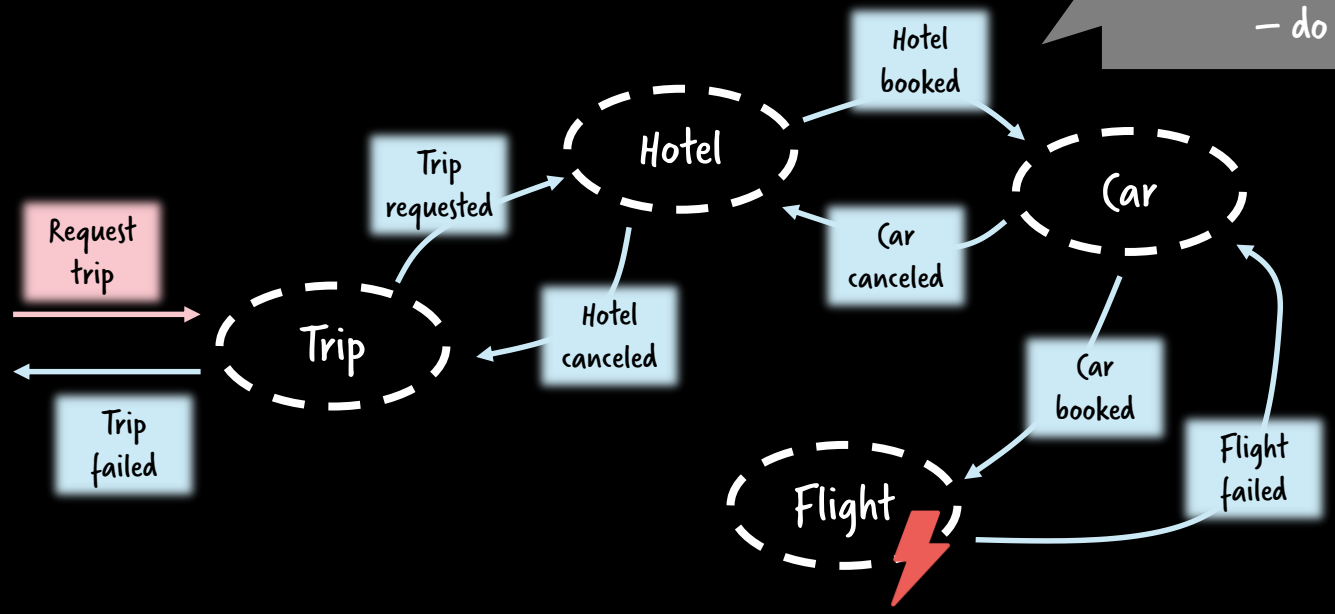
A few examples of these processes are:

- Studio partner integration for content ingestion
- IMF based content ingestion from our partners
- Process of setting up new titles within Netflix
- Content ingestion, encoding, and deployment to CDN

Traditionally, some of these processes had been orchestrated in an ad-hoc manner using a combination of pub/sub, making direct REST calls, and using a database to manage the state. However, as the number of microservices grow and the complexity of the processes increases, getting visibility into these distributed workflows becomes difficult without a central orchestrator.

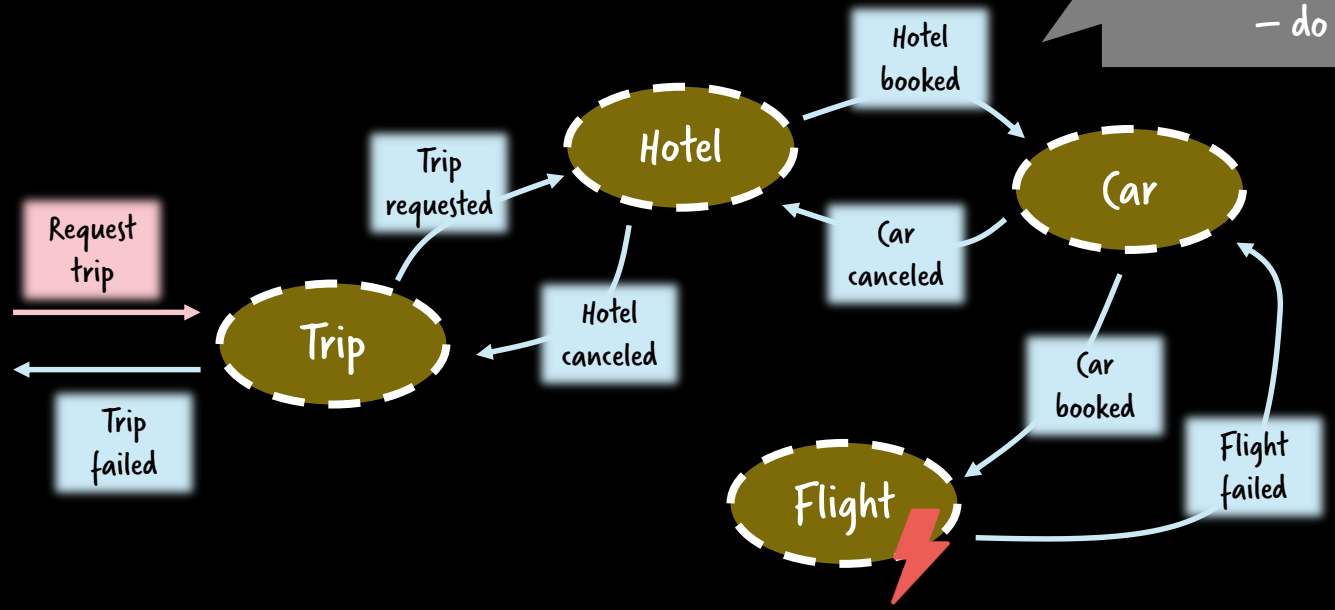
Implementing changes in the process

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!

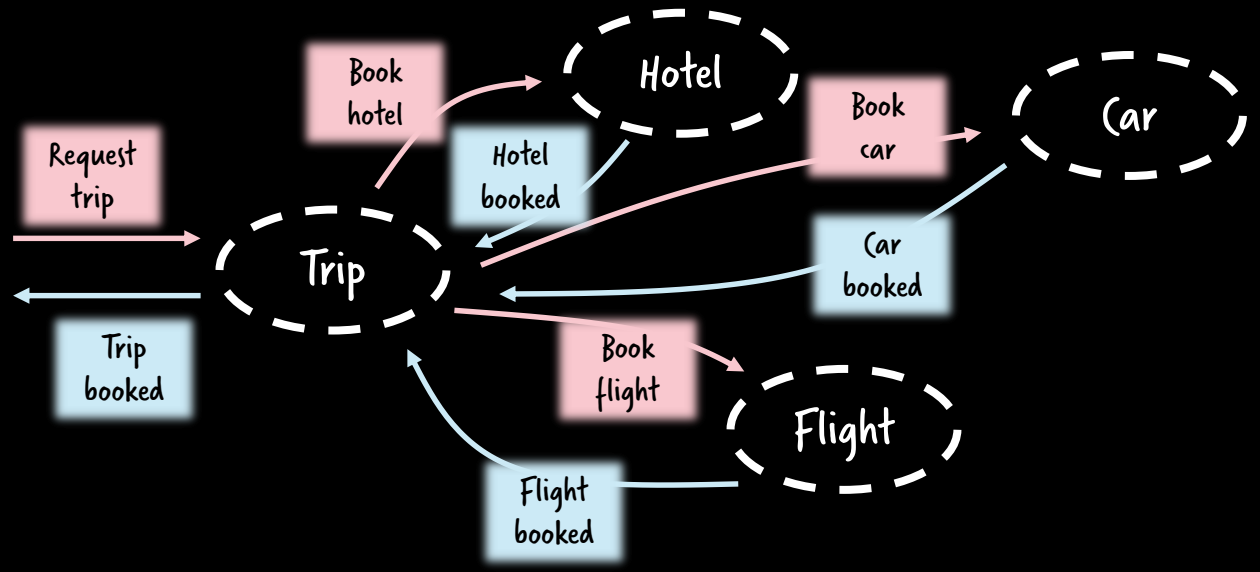
Photo by born1945, available under Creative Commons BY 2.0 license.

What we wanted

vs. what we got

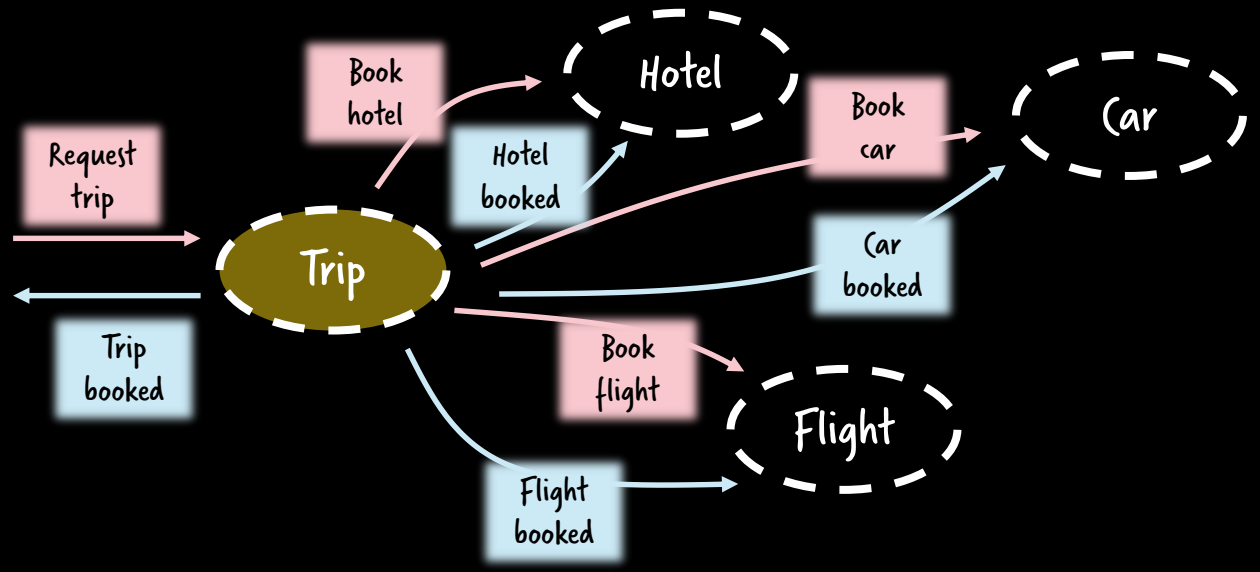
Photo by Lijian Zhang, available under Creative Commons SA 2.0 License and Pedobear19 / CC BY-SA 4.0

orchestration



orchestration

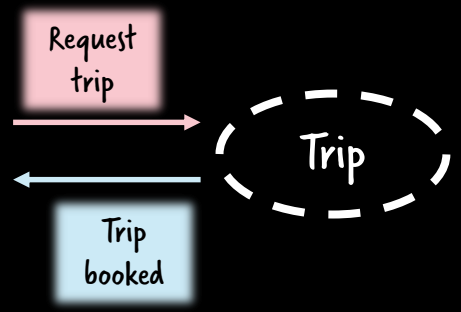
We have a new basic agreement with the car rental agency and can cancel for free within 1 hour – do that first!

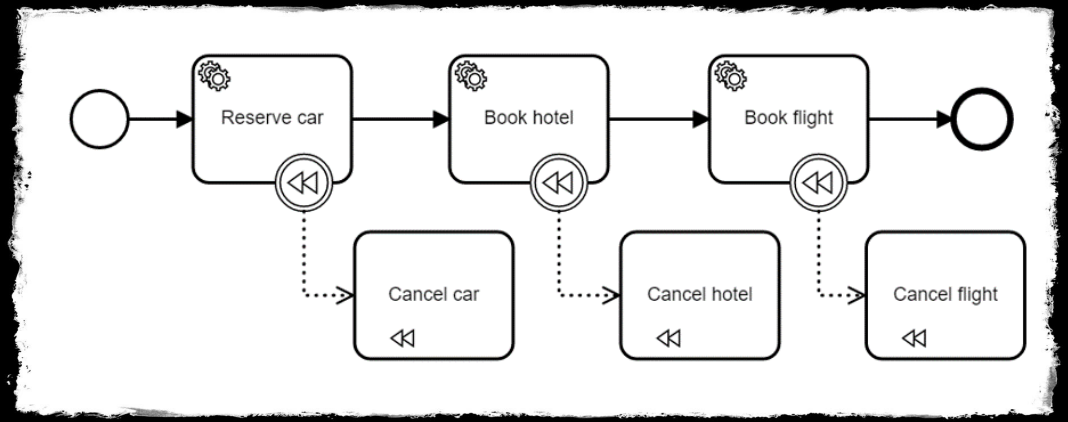


You have to adjust one service and redeploy only this one!

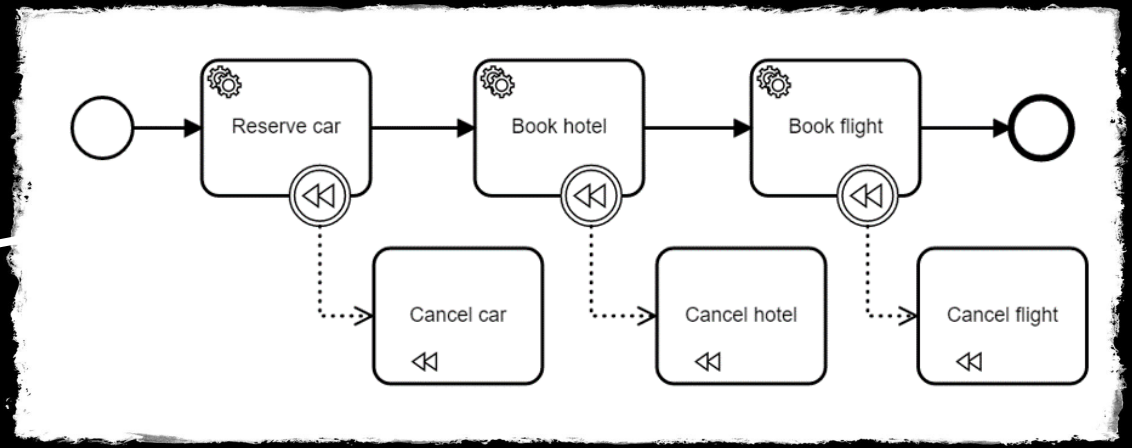
Describe orchestration with BPMN

Saga Pattern
(implemented by BPMN compensation)

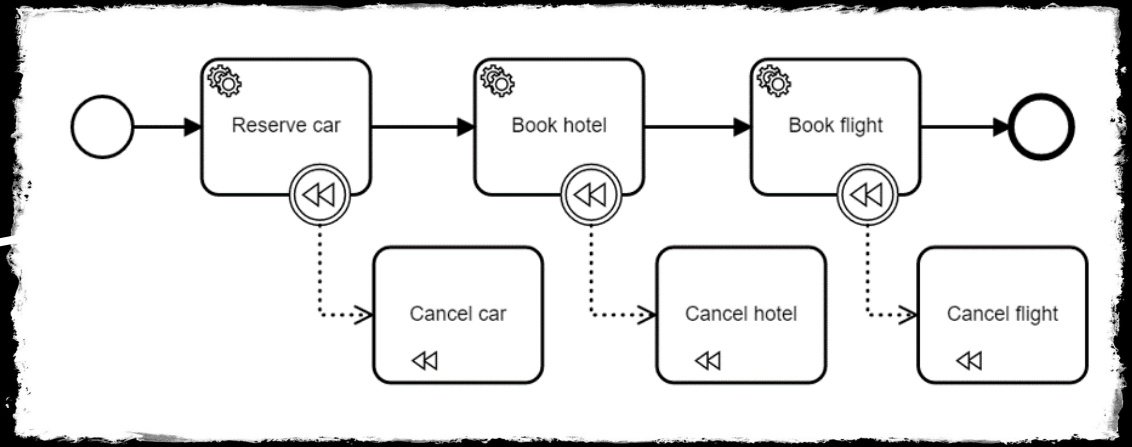




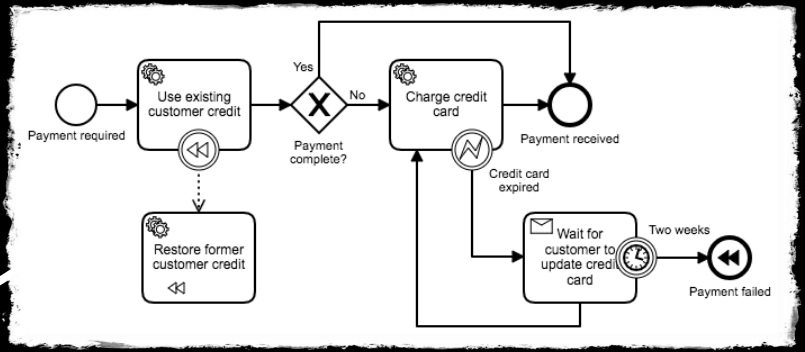
The workflow is part of the service



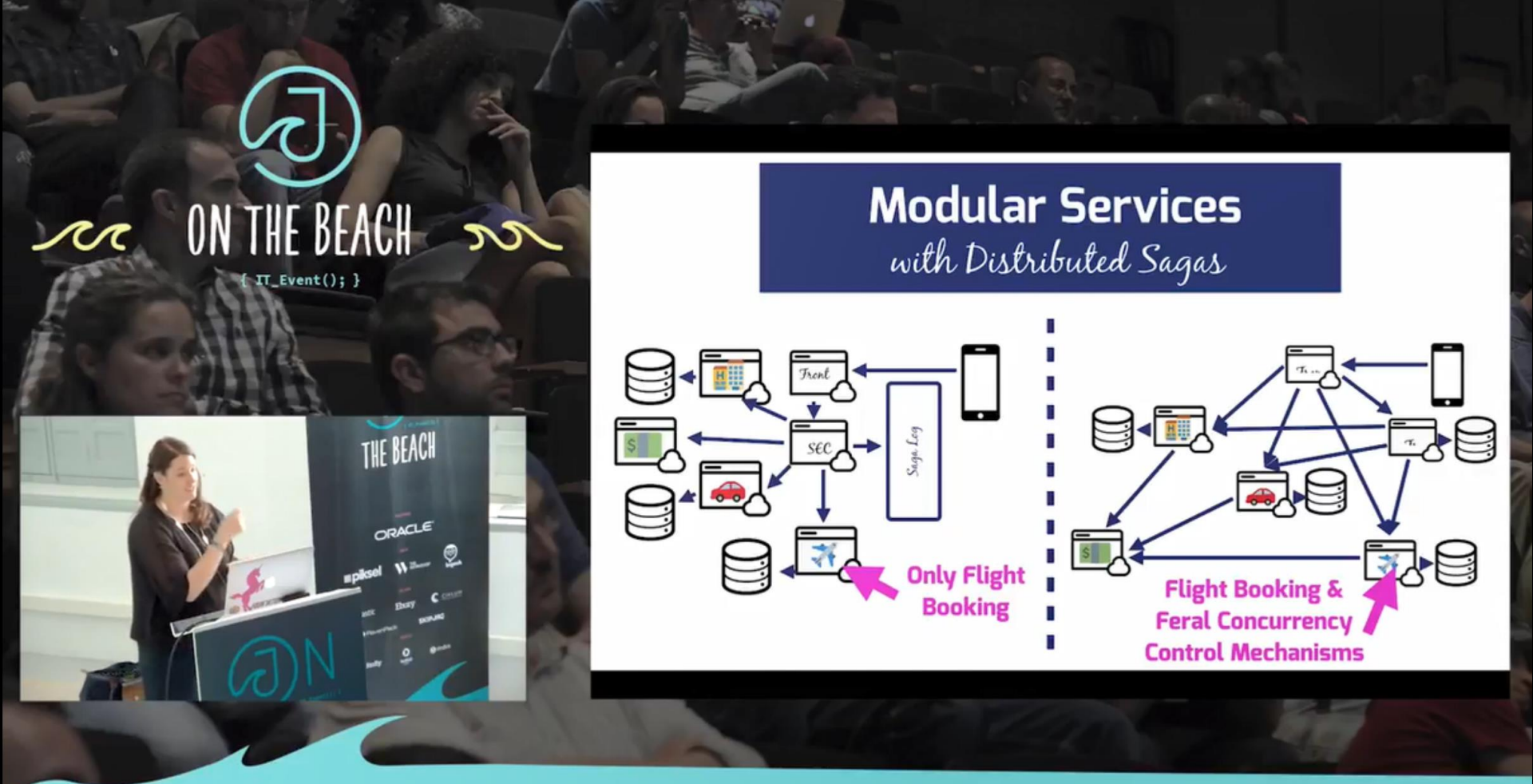
The workflow is part of the service



Trip

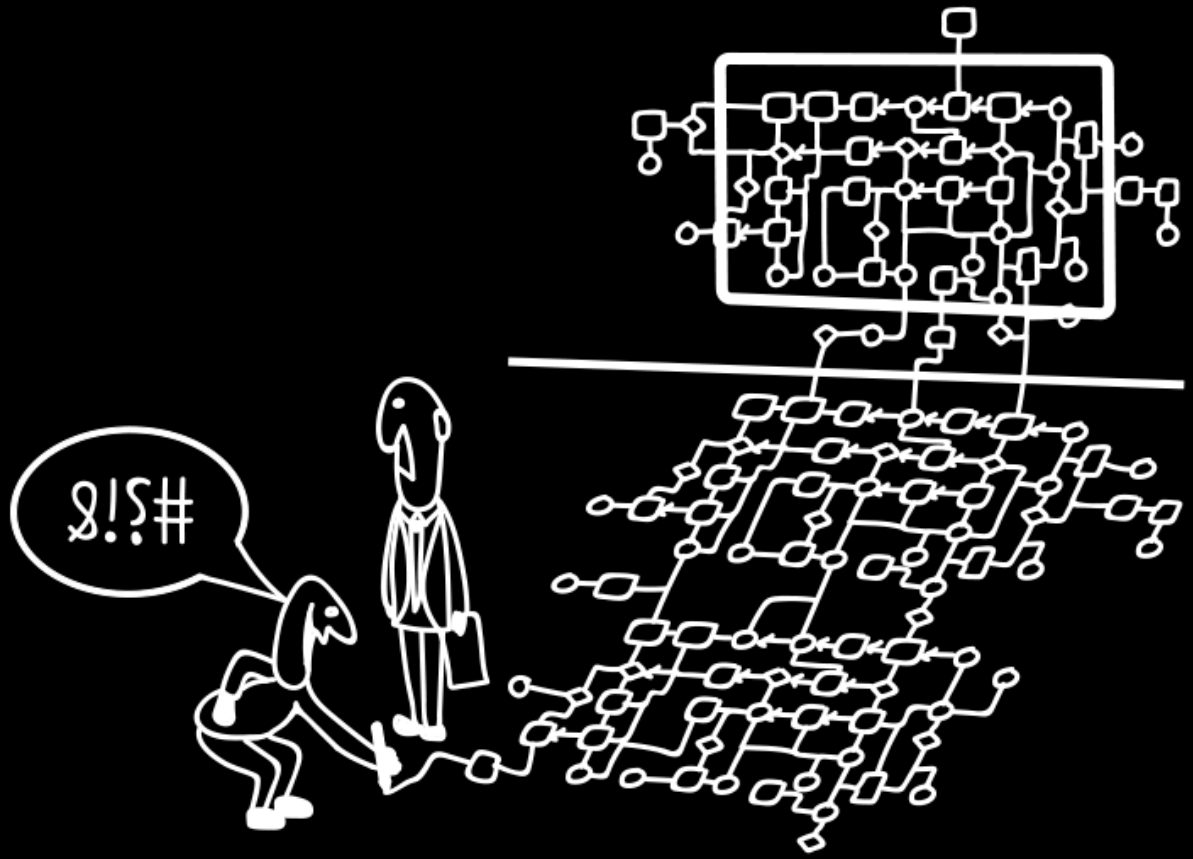


Payment



Caitie McCaffrey | @caitie

Graphical models?



Clemens Vasters
Architect at Microsoft

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

Sagas

Today's has been a lively day in some parts of the Twitternerse 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 for distributed transactions across location and/or trust boundaries) can't really be handled using the classic ACID model with 2-Phase commit and holding locks for the duration of the work. Instead, a Saga splits working individual transactions whose effects can be, somehow, reversed after work has been performed and committed.

The picture shows a simple Saga. If you books 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 those 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 frame 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 he 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 backout 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 model, and not a framework. I wrote this in less than 90 minutes, so don't expect to reuse this.

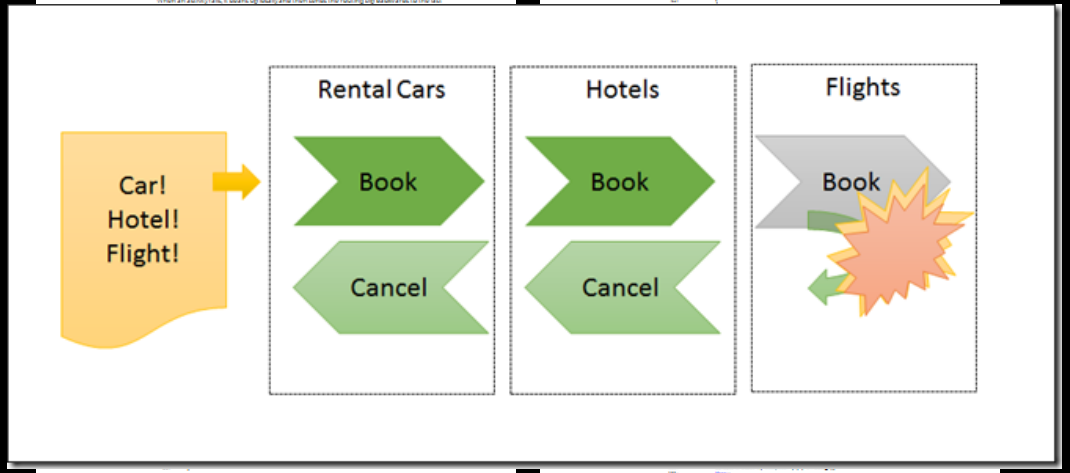
The main program sets up an exemplary routing slip (all the classes are in the one file) and creates three completely independent "processes" (activity hosts) that are each responsible for handling a particular kind of work. The "processes" are linked by a "network" and each kind of activity has an address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

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

The routing slip is "serializable" (it's non, protocol that it is) and it's the only piece of information that flows between the collaborating activities. There is no central coordination. All works 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 there it maintains a stack. The routing slip also handles recording and invoking whatever the "next" thing to call is on the way forward and backward.

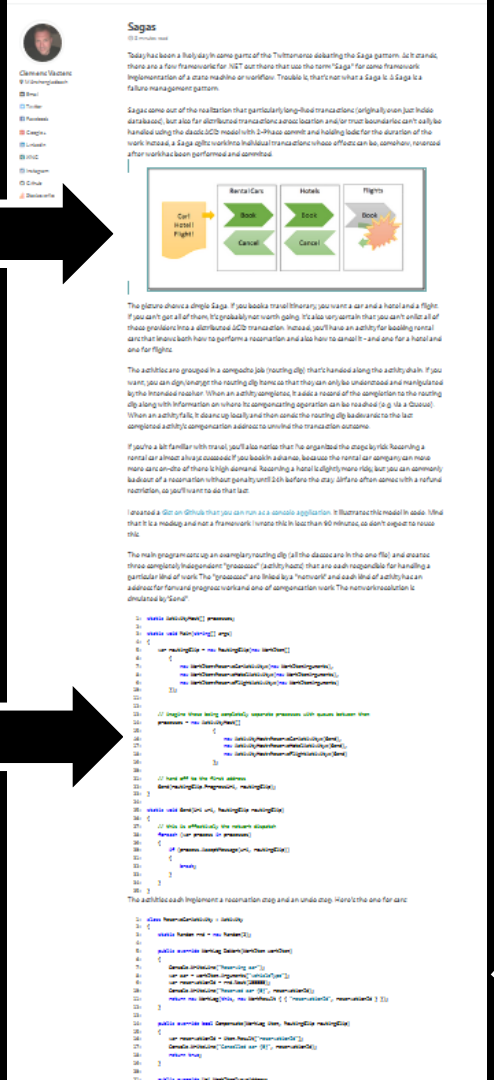
The local work and making the decisions is encapsulated in the IActivityHost, which calls ProcessNext() on the routing slip to resolve the next activity and call its DoWork() function on the way forward or will resolve the last executed activity on the way backward (made to Compensate() function. Again,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

Clemens Vasters
Architect at Microsoft



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

Clemens Vasters
Architect at Microsoft



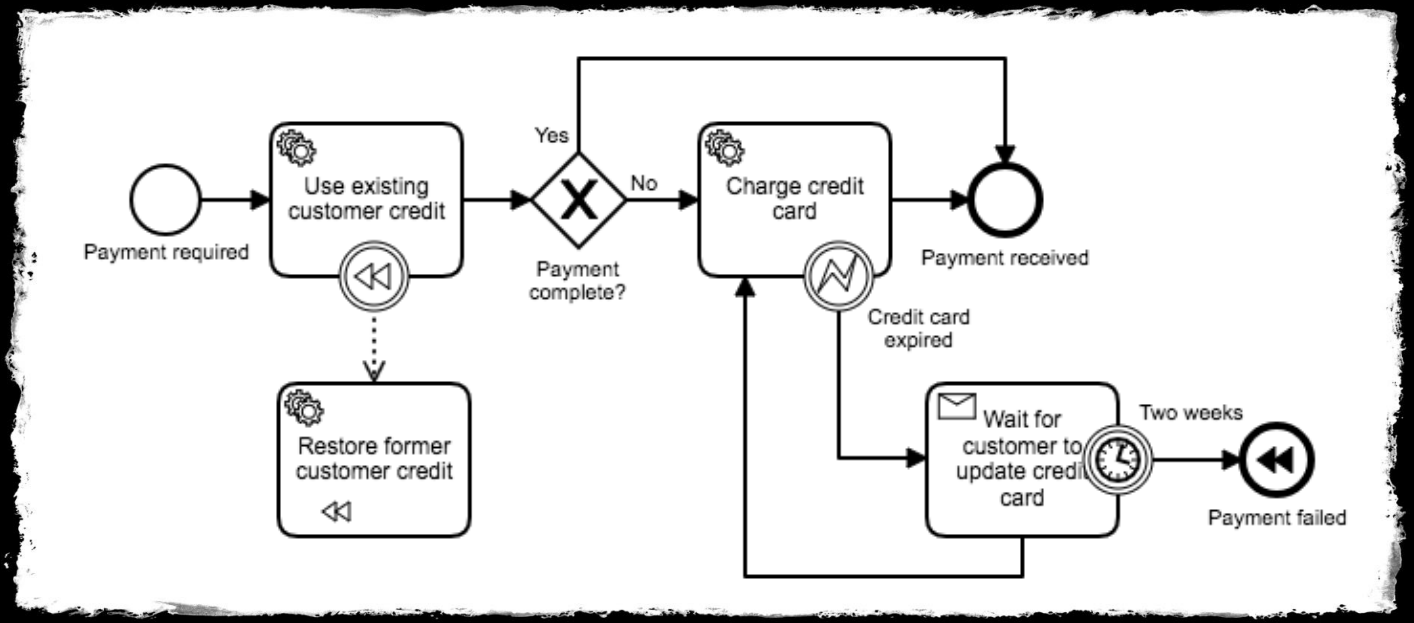
http://vasters.com/archive/Sagas.html

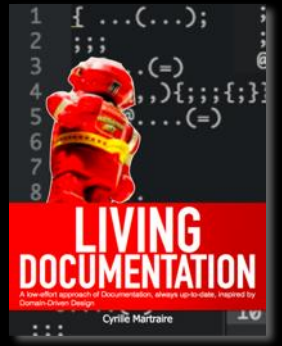
BPMN

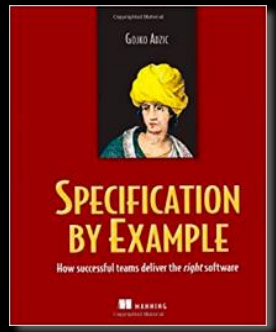
Business Process Model and Notation

ISO Standard

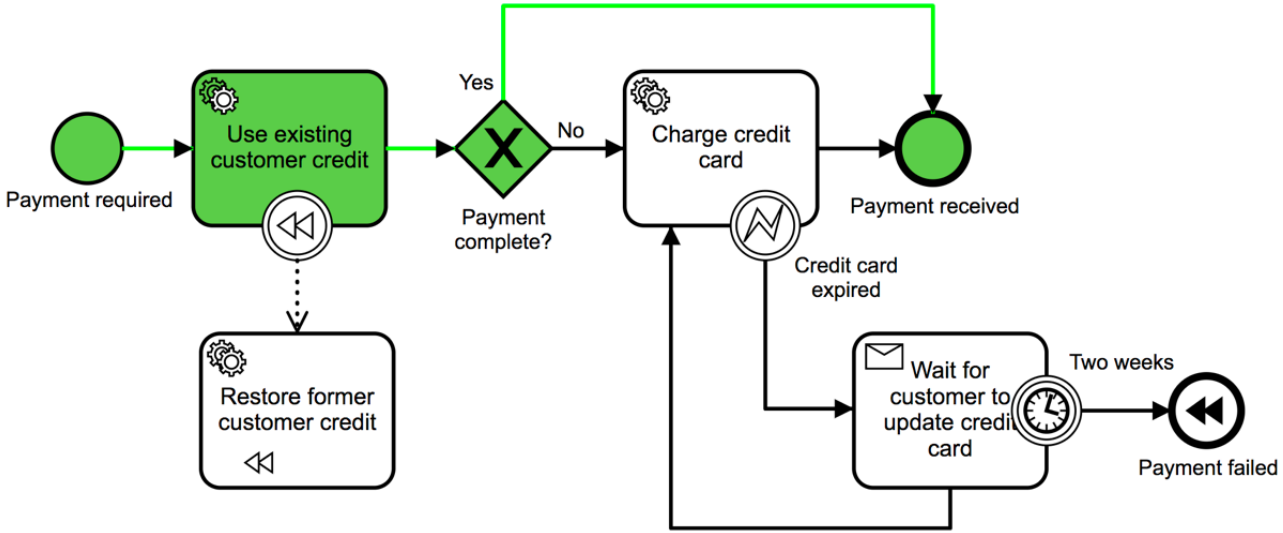
Living documentation for long-running behaviour



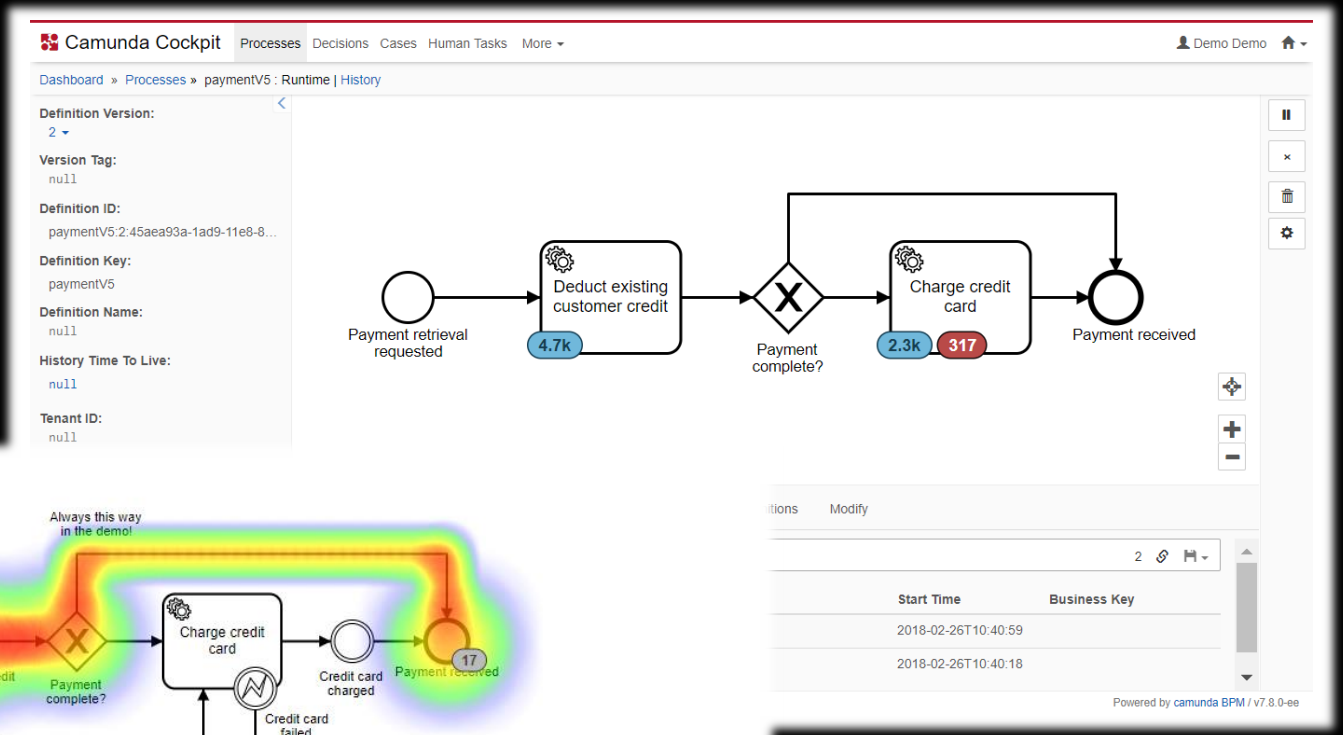


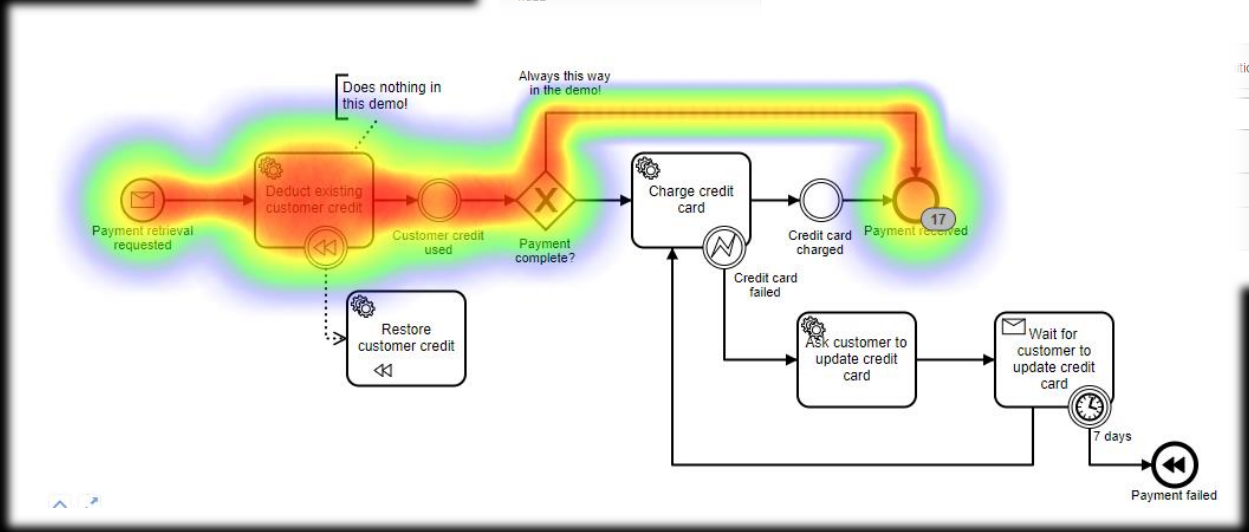


Visual HTML reports for test cases

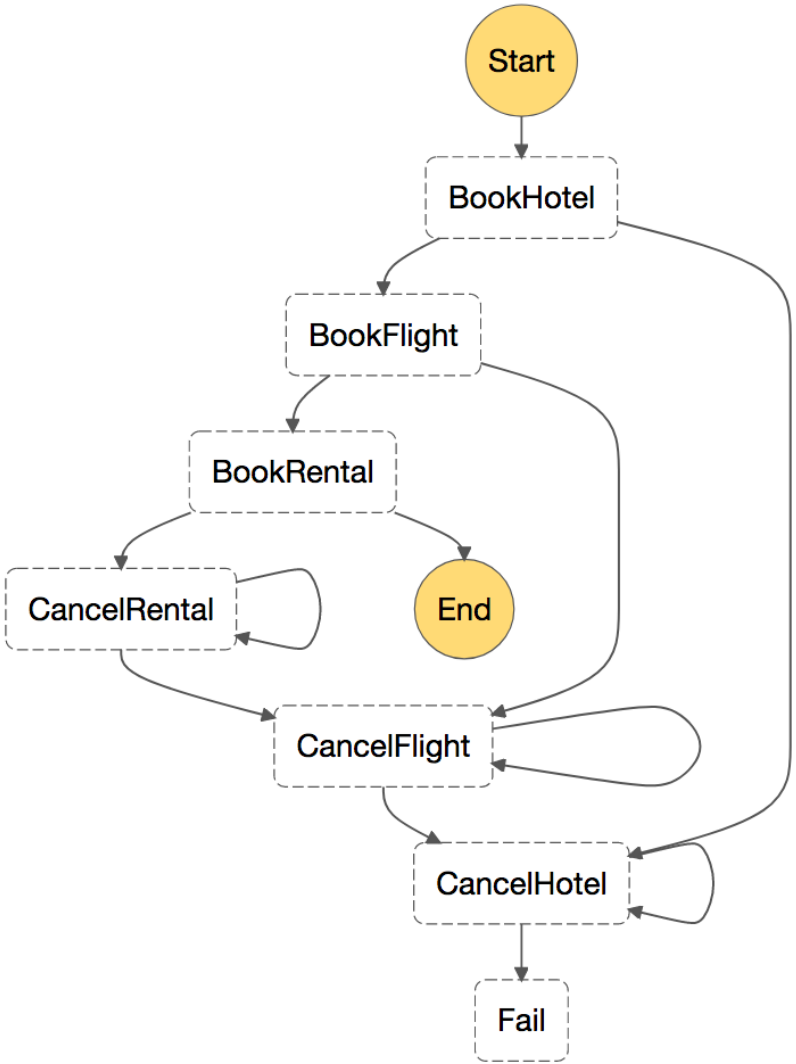


BizDevOps


Camunda Cockpit
Processes
Decisions
Cases
Human Tasks
More
Demo Demo
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
Payment retrieval requested
Deduct existing customer credit
4.7k
Payment complete?
Charge credit card
2.3k
317
Payment received
Modify
Start Time
Business Key
2018-02-26T10:40:59
2018-02-26T10:40:18
Powered by camunda BPM / v7.8.0-ee


Does nothing in this demo!
Always this way in the demo!
Payment retrieval requested
Deduct existing customer credit
Customer credit used
Payment complete?
Charge credit card
Credit card charged
17
Payment received
Credit card failed
Restore customer credit
Ask customer to update credit card
Wait for customer to update credit card
7 days
Payment failed

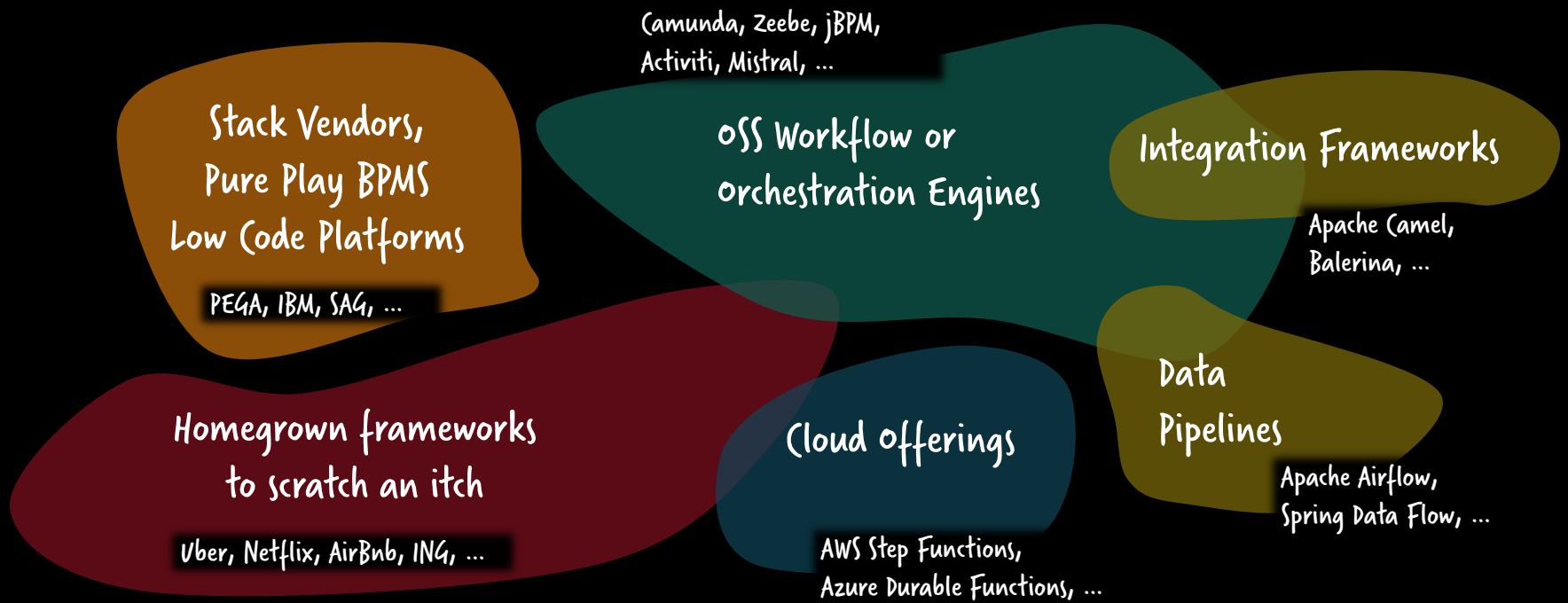
Saga with AWS Step Functions



https://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

Thoughts on the state machine | workflow engine market

Thoughts on the state machine | workflow engine market



BPMN

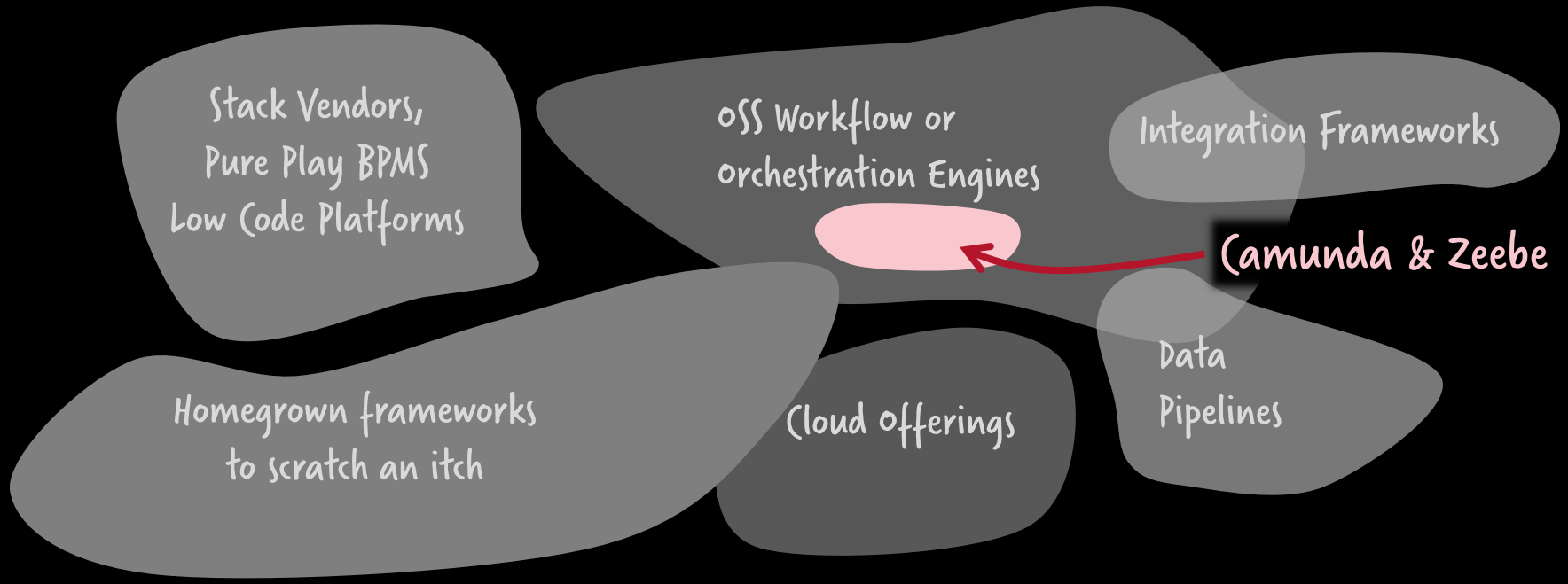
Does it support stateful operations?

Does it support the necessary flow logic?

Does it support BizDevOps?

Does it scale?

My personal pro-tip for a shortlist ;-)



Recap

- Grown ups don't use distributed transactions but eventual consistency
- Idempotency is super important in distributed systems
- Some consistency challenges require state
- Know some strategies
  - Stateful retry & cleanup
  - Saga / Compensation
  - Apologies


Thank you!

Contact: mail@berndruecker.io
@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

InfoQ ueue

https://www.infoq.com/articles/events-workflow-automation

THENEWSTACK

https://thenewstack.io/5-workflow-automation-use-cases-you-might-not-have-considered/