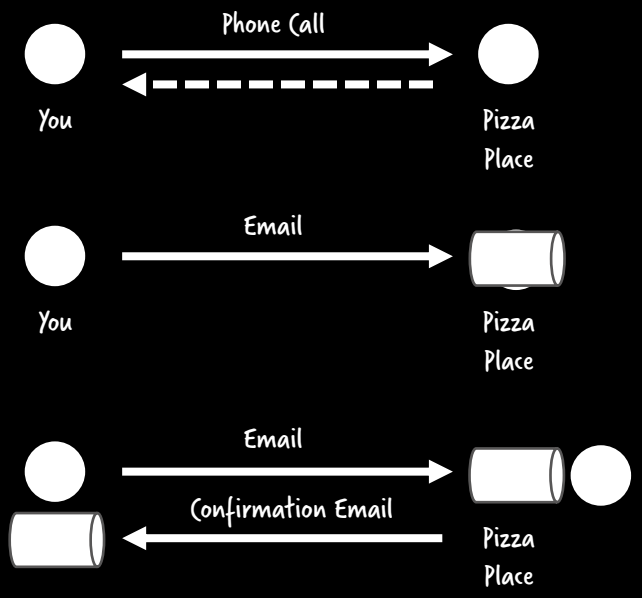
# Loosely or Lousily Coupled?

## Understanding Communication Patterns in Microservices Architectures

@berndruecker

# Let's talk about food

# How does ordering Pizza work?




Synchronous blocking communication
Feedback loop (ack, confirmation or rejection)
Temporal coupling (e.g. busy, not answering)

Asynchronous non-blocking communication
No temporal coupling

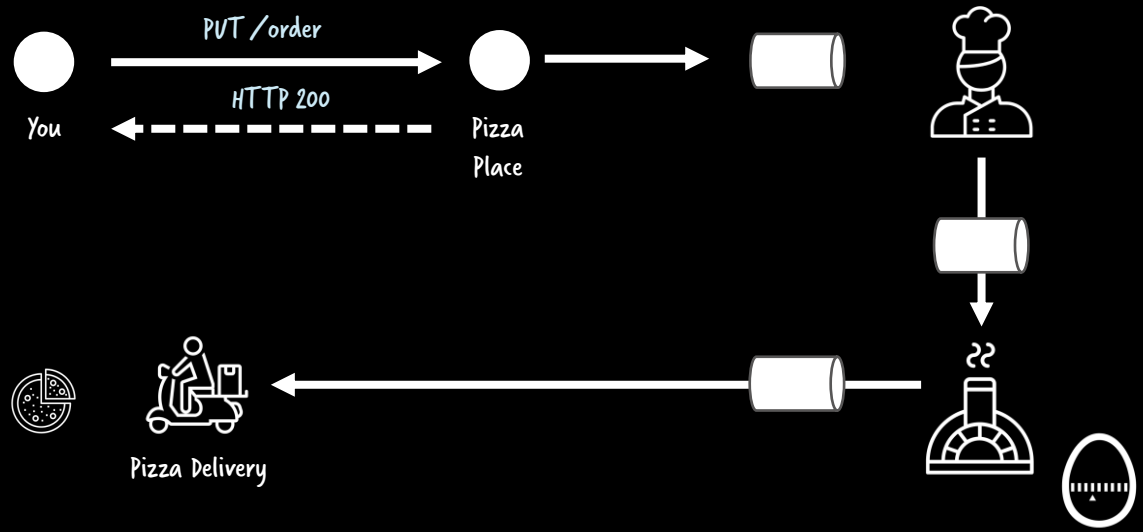
A feedback loop might make sense
(ack, confirmation or rejection)

# Feedback loop != result


Feedback (ACK, confirmation, rejection)


Result

# only the first communication step is synchronous / blocking




The task of Pizza making is long running

# Synchronous blocking behavior for the result?

Bad user experience
Does not scale well

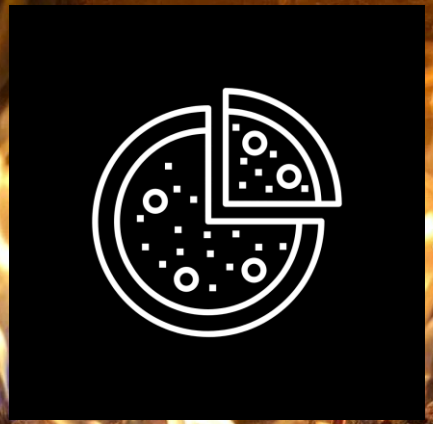

# Scalable Coffee Making

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

Photo by John Ingle

# Long running

# Long running




Long running basically means waiting

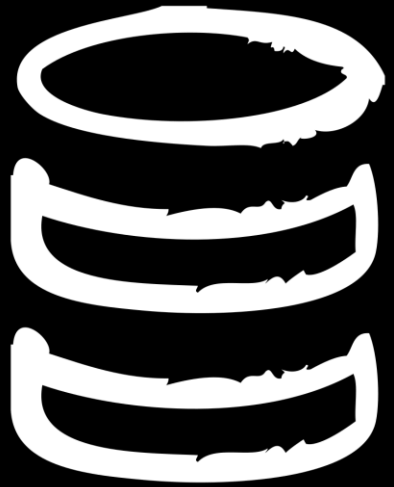
# When do services want to wait? Some business reasons...

Human work

Waiting for response

Let some time pass

# Why is waiting a pain?



Persistent state

Monitoring & Operations

Versioning

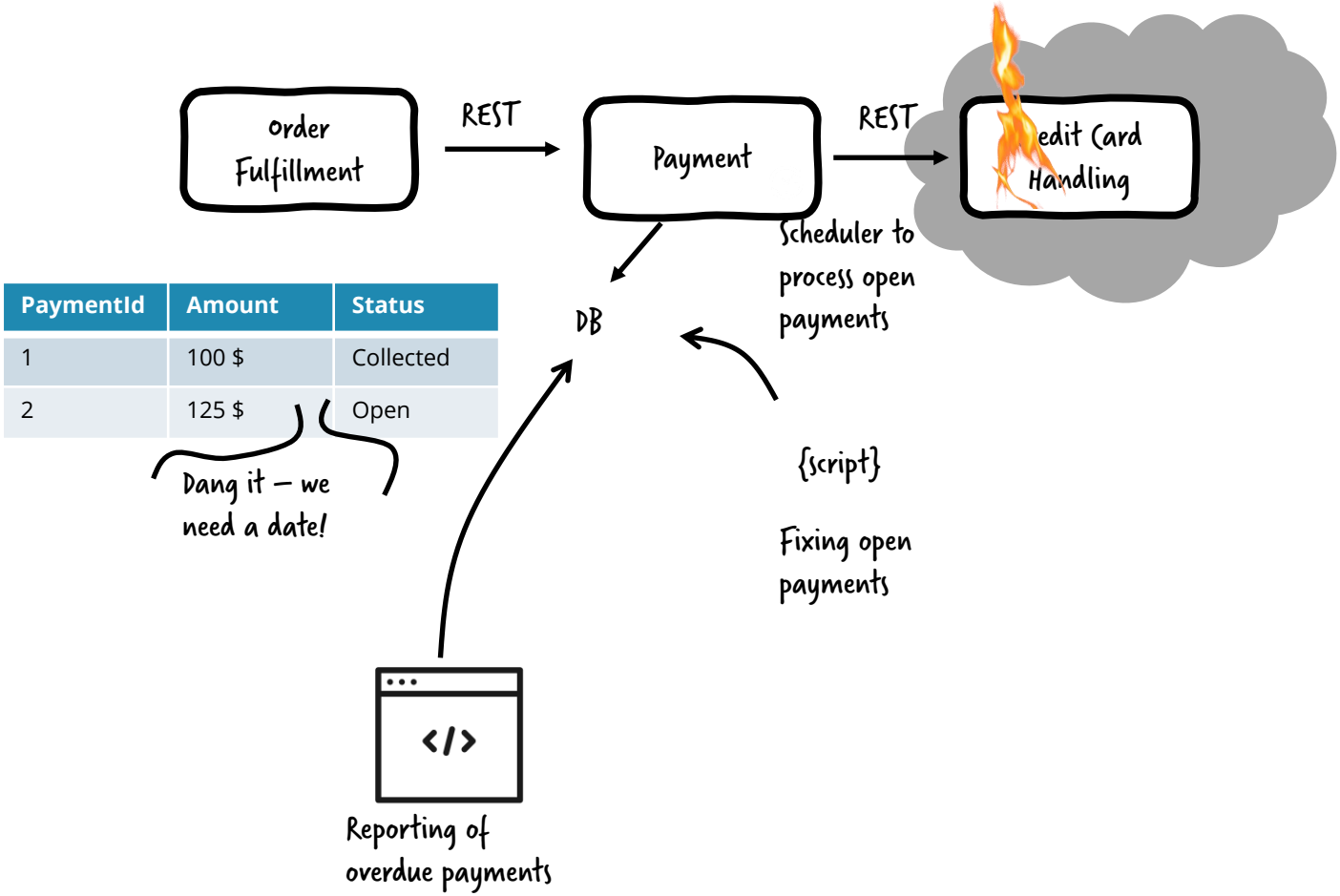
Scheduling & Timeouts

Visibility

Domain Logic

Scalability & Resilience

How to solve the technical challenges without adding accidential complexity?

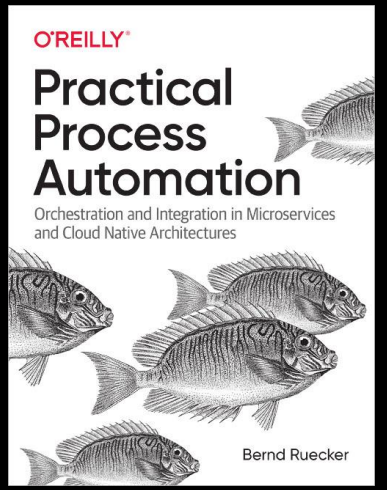

Order Fulfillment
REST
Payment
REST
Credit Card Handling
Scheduler to process open payments
DB
PaymentId
Amount
Status
1
100 $
Collected
2
125 $
Open
Dang it – we need a date!
{script}
Fixing open payments
Reporting of overdue payments

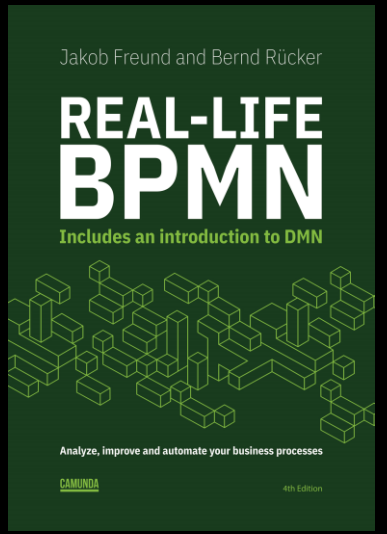
# Warning:
# Contains Opinion

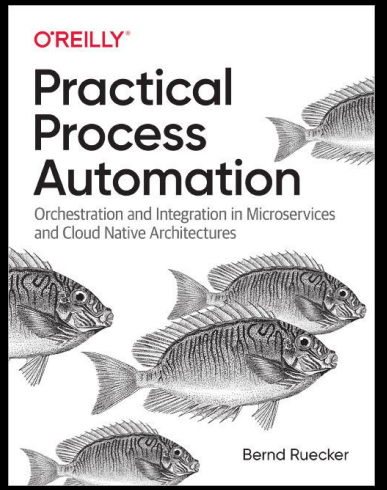
# Bernd Ruecker

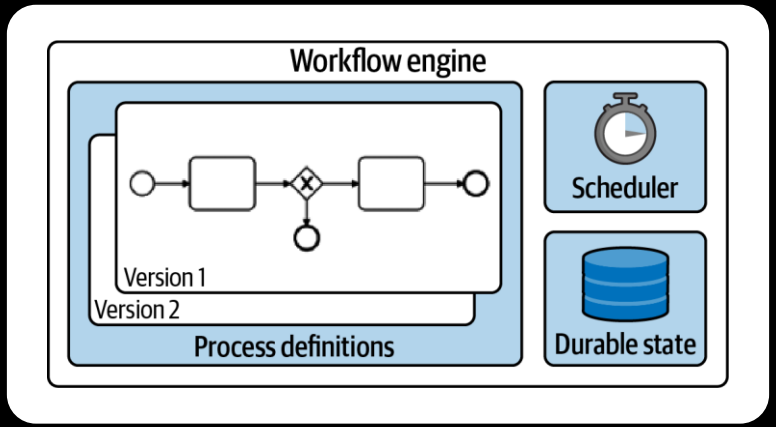
Co-founder and
Chief Technologist of
Camunda


bernd.ruecker@camunda.com
@berndruecker
http://berndruecker.io/

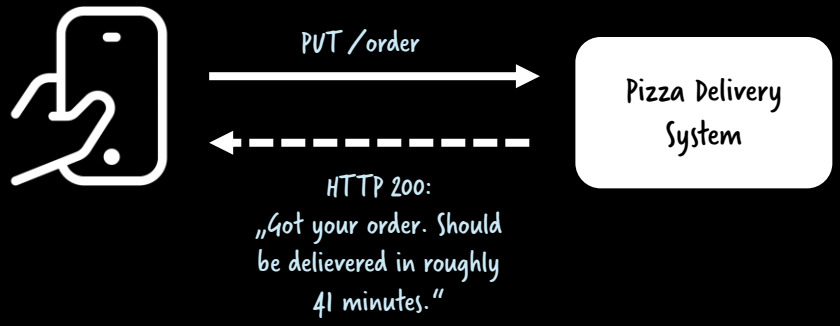





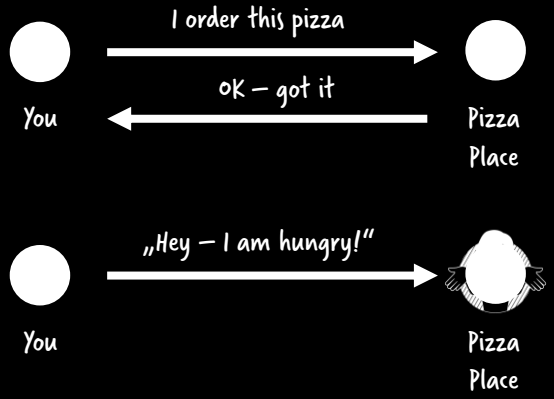




Workflow Engine aka Process Engine aka Orchestration Engine

# Building a pizza ordering app



# Command vs. event-based communication




Command = Intent
Cannot be ignored
Independant of communication channel

Event = Fact
Sender can't control what happens

# Definitions

| | | |
|---|---|---|
| Event | = | Something happened in the past. It is a fact.<br>Sender does not know who picks up the event. |
| Command | = | Sender wants s.th. to happen. It has an intent.<br>Recipient does not know who issued the command. |

Some prefer request over command

# Communication options – Quick Summary

| **Communication Style** | Synchronous Blocking | Asynchronous Non-Blocking | |
|---|---|---|---|
| **Collaboration Style** | Command-Driven | | Event-Driven |
| **Example** | REST | Messaging (Queues) | Messaging (Topics) |
| **Feedback Loop** | HTTP Response | Response Message | - |
| **Pizza Ordering via** | Phone Call | E-Mail | Twitter |

This is not the same!

# Events vs. Commands


@berndruecker
Orchestrator
Command

# Example: Build a pizza ordering app using events

PUT /order

HTTP 200:
„Got your order. Should be delievered in roughly 41 minutes."

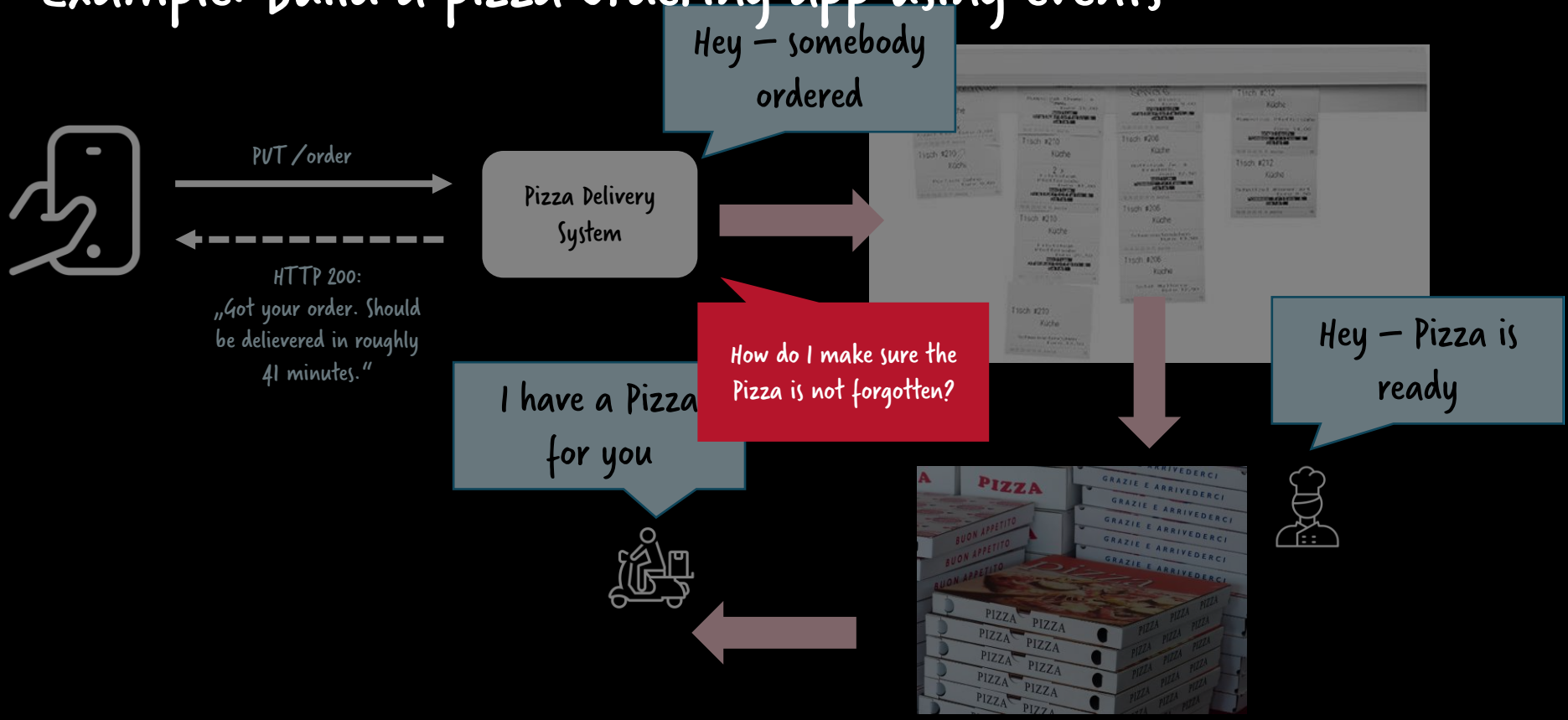
Pizza Delivery System

Hey – somebody ordered

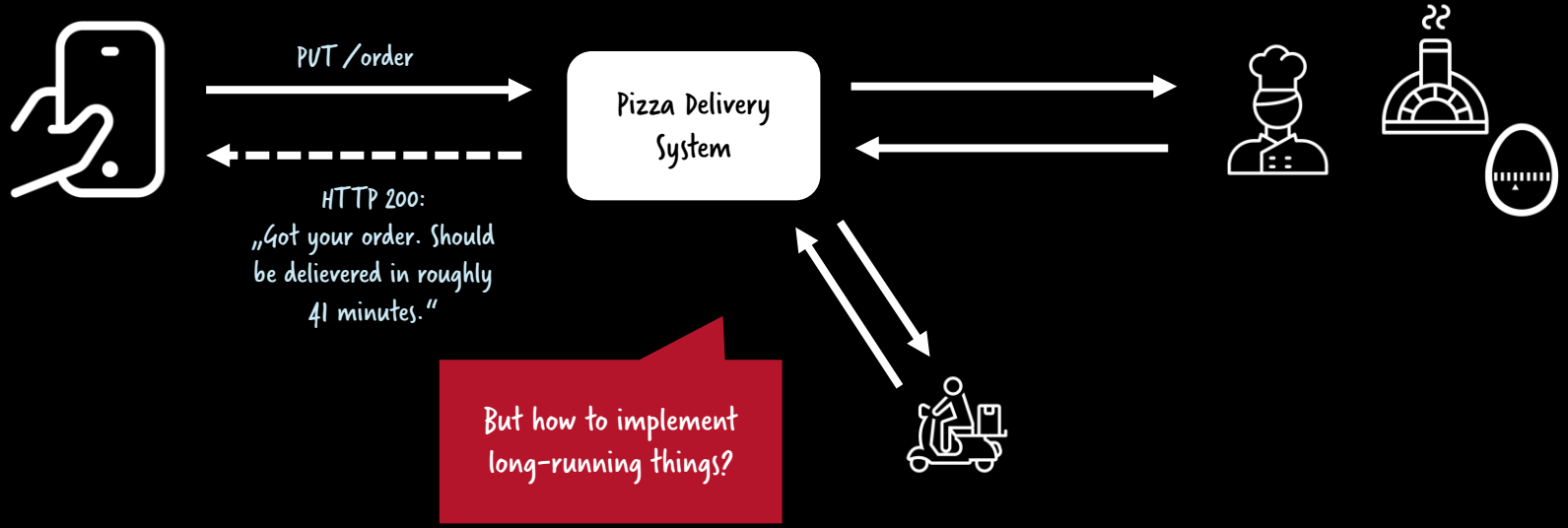
Hey – Pizza is ready


I have a Pizza for you

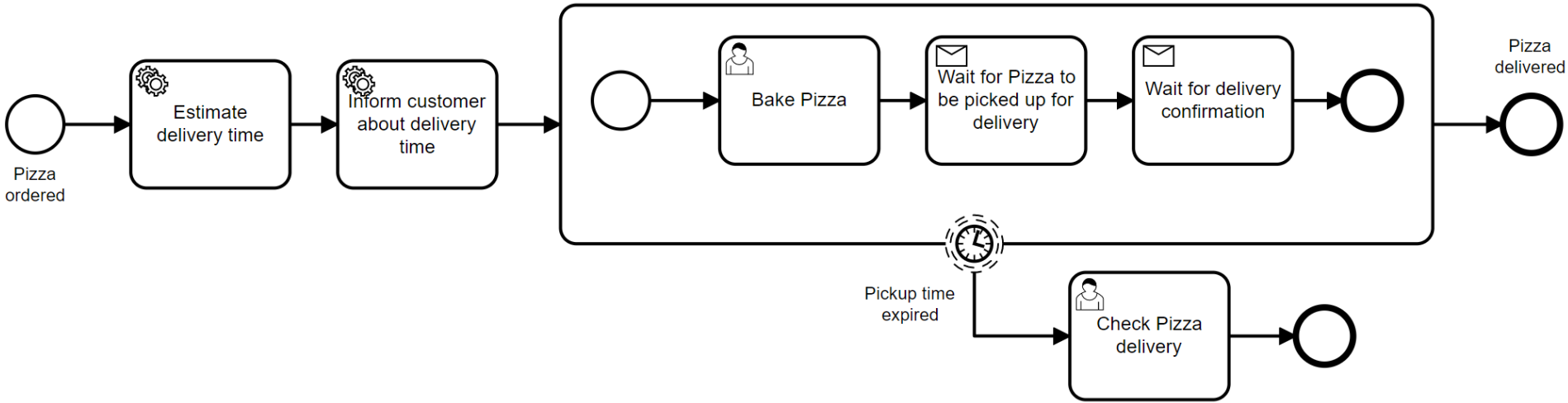
# Example: Build a pizza ordering app using events



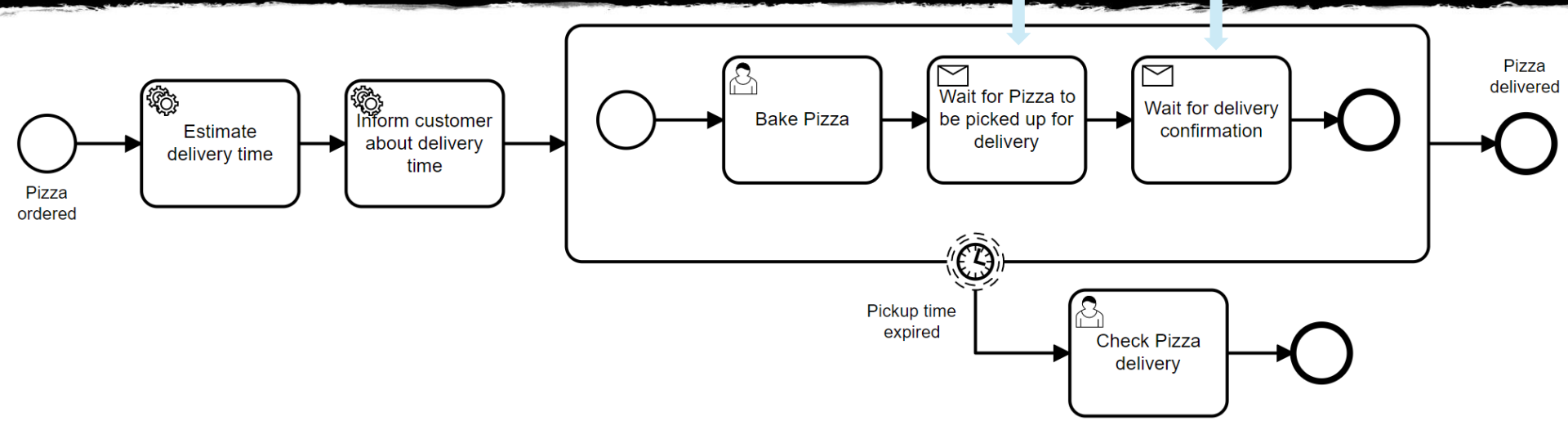
# Example: Build a pizza ordering app via orchestration



# A process for the Pizza ordering system



# You can still work with events

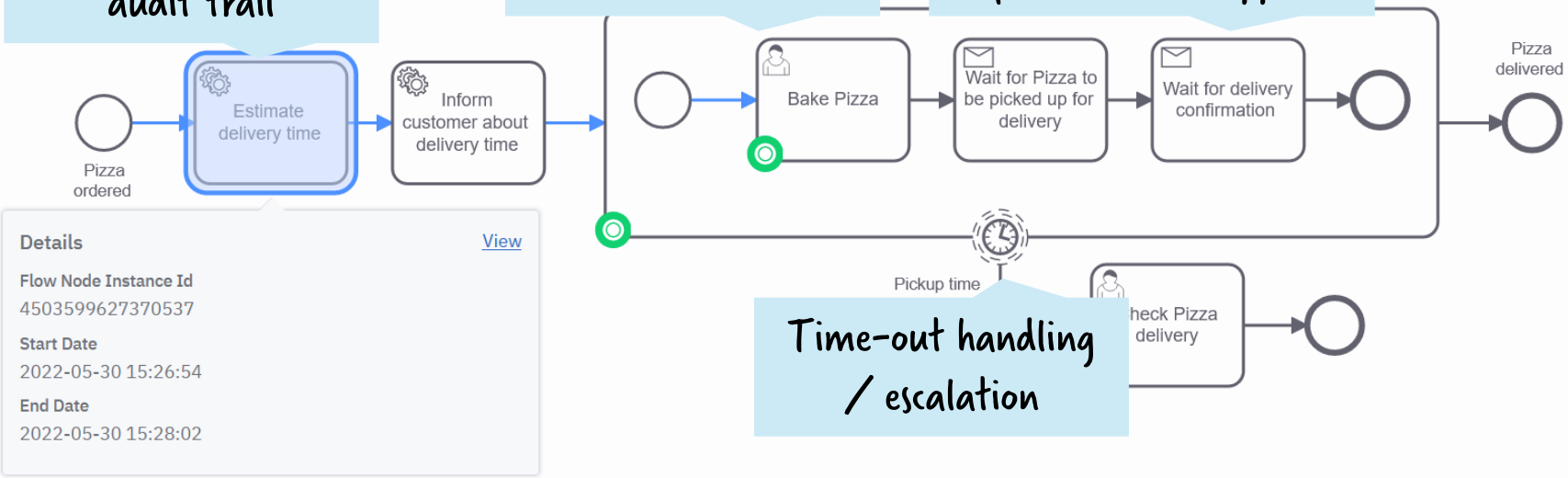


# Advantages

Visibility: History and audit trail

Visibility: What's the current status?

Long running: Waiting for events to happen




Time-out handling / escalation

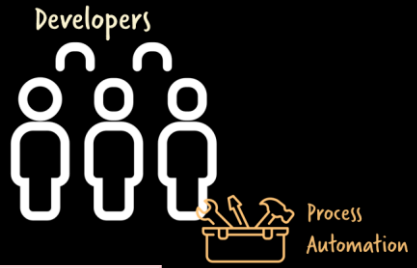
# Developer-friendly workflow engines

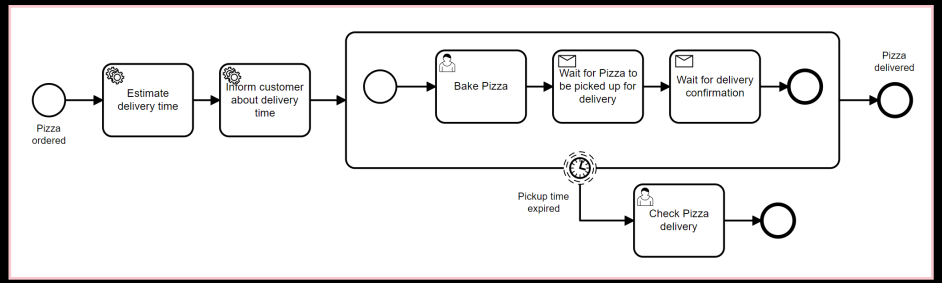
## Your code to provide a REST endpoint

```
@PutMapping("/pizza-order")
public ResponseEntity<PizzaOrderResponse pizzaOrderReceived(...) {
  HashMap<String, Object> variables = new HashMap<String, Object>();
  variables.put("orderId", orderId);

  ProcessInstanceEvent processInstance = camunda.newCreateInstanceCommand()
            .bpmnProcessId("pizza-order")
            .latestVersion()
            .variables(variables)
            .send().join();

  return ResponseEntity.status(HttpStatus.ACCEPTED).build();
}
```



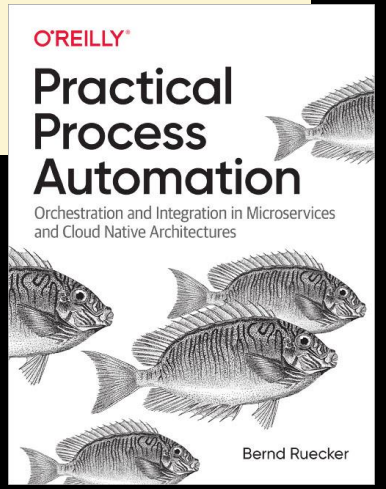



# orchestration vs. Choreography

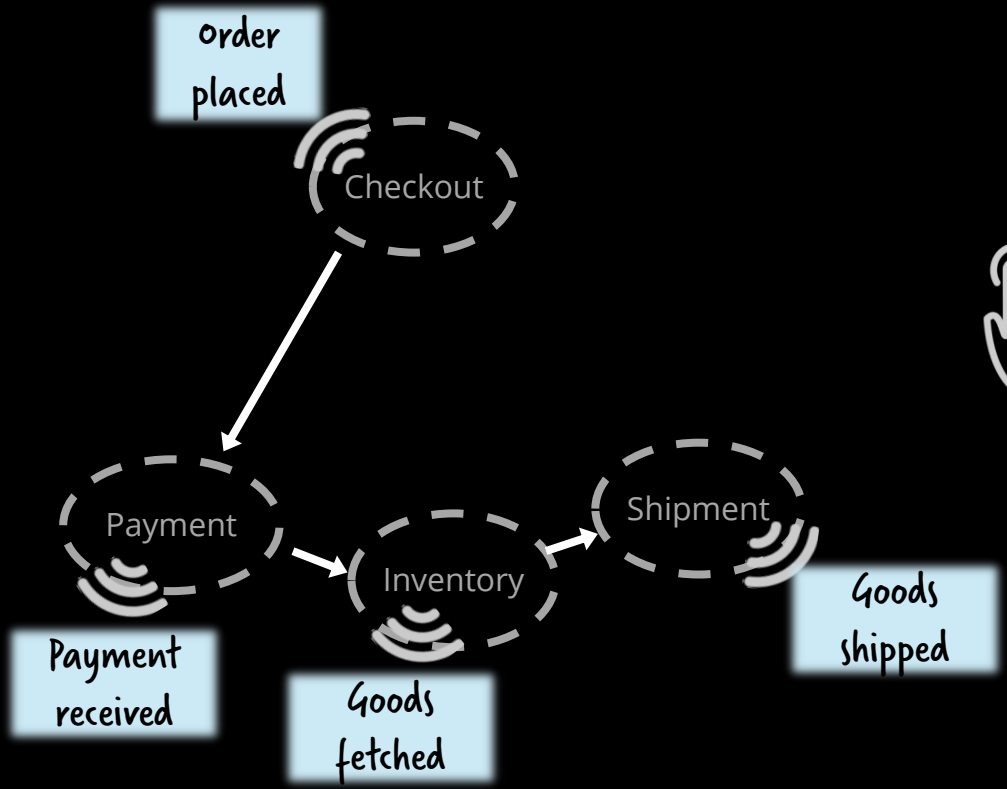
# Definition

orchestration = command-driven communication

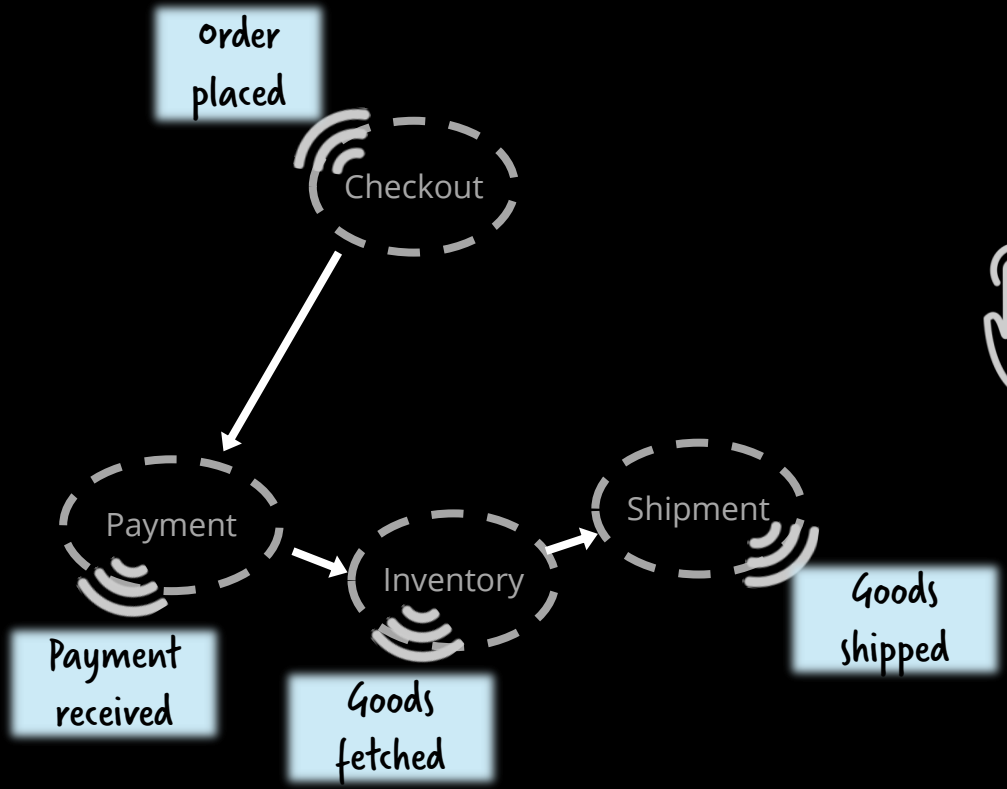
Choreography = event-driven communication

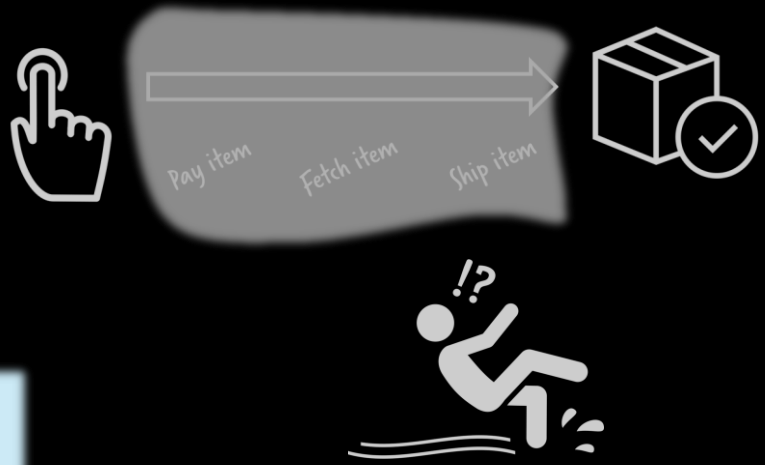


# Let's switch examples: order fulfillment




# Event chains



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

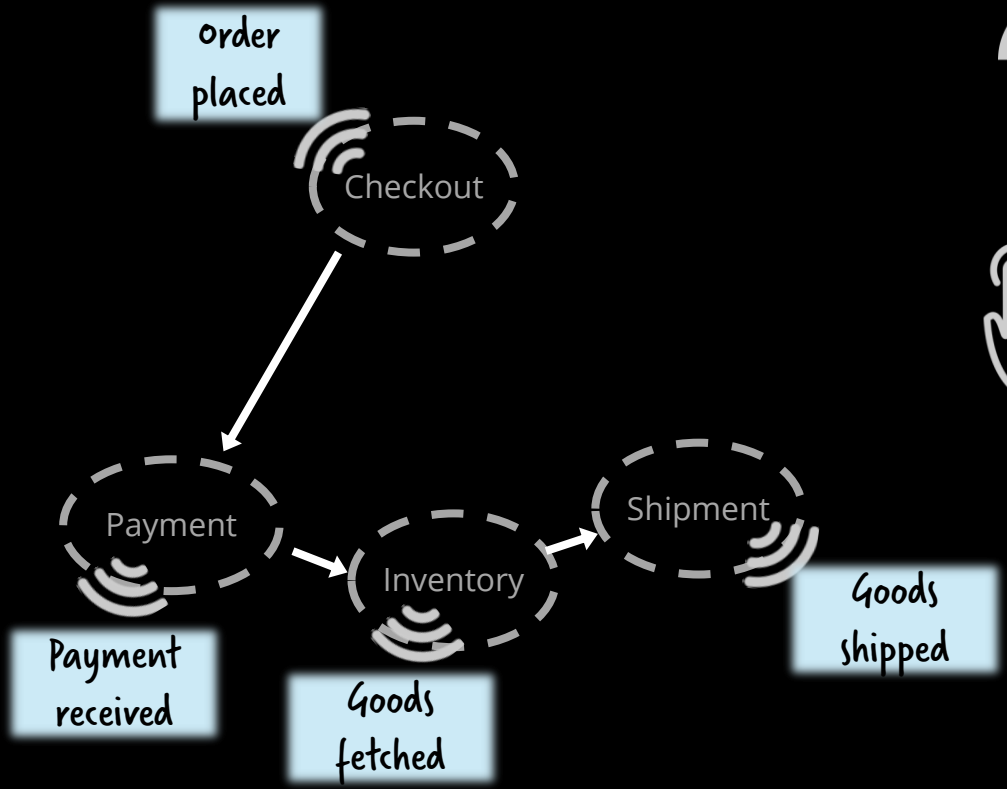
The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow, and thus set yourself up for trouble in future years.

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

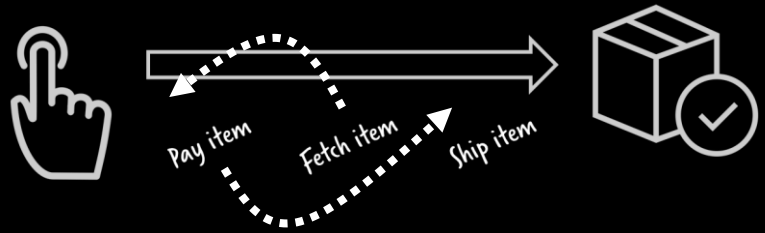
The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow, and thus set yourself up for trouble in future years.

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

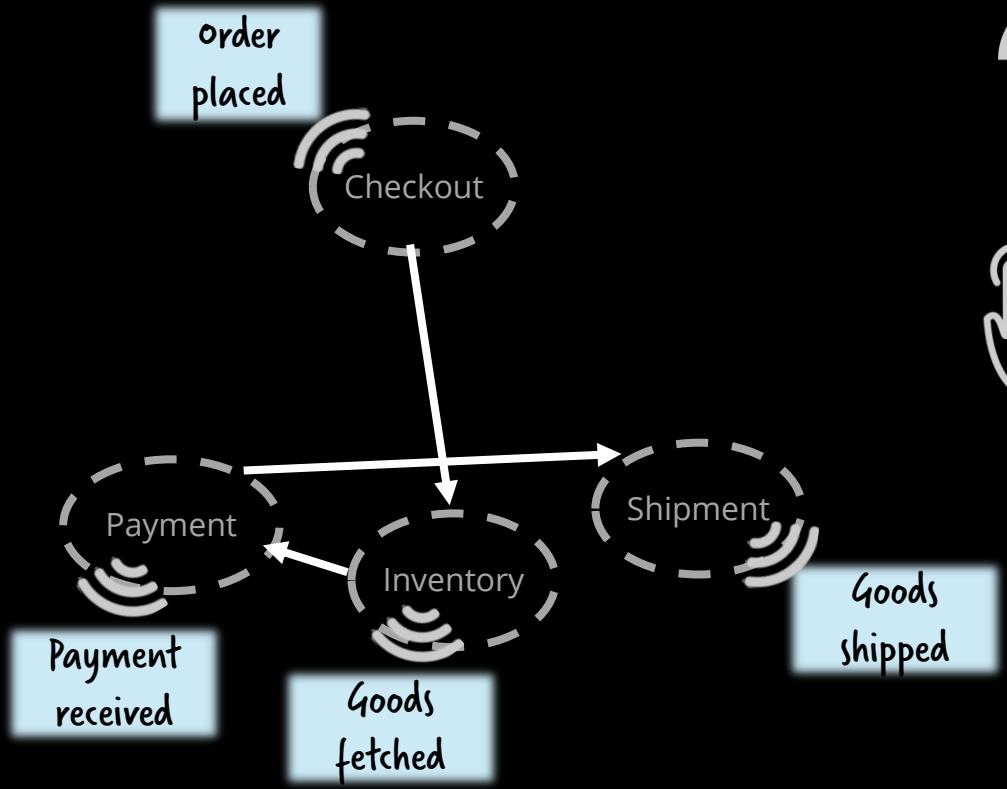
# Peer-to-peer event chains



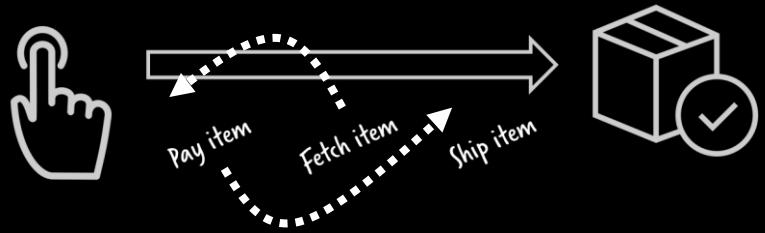


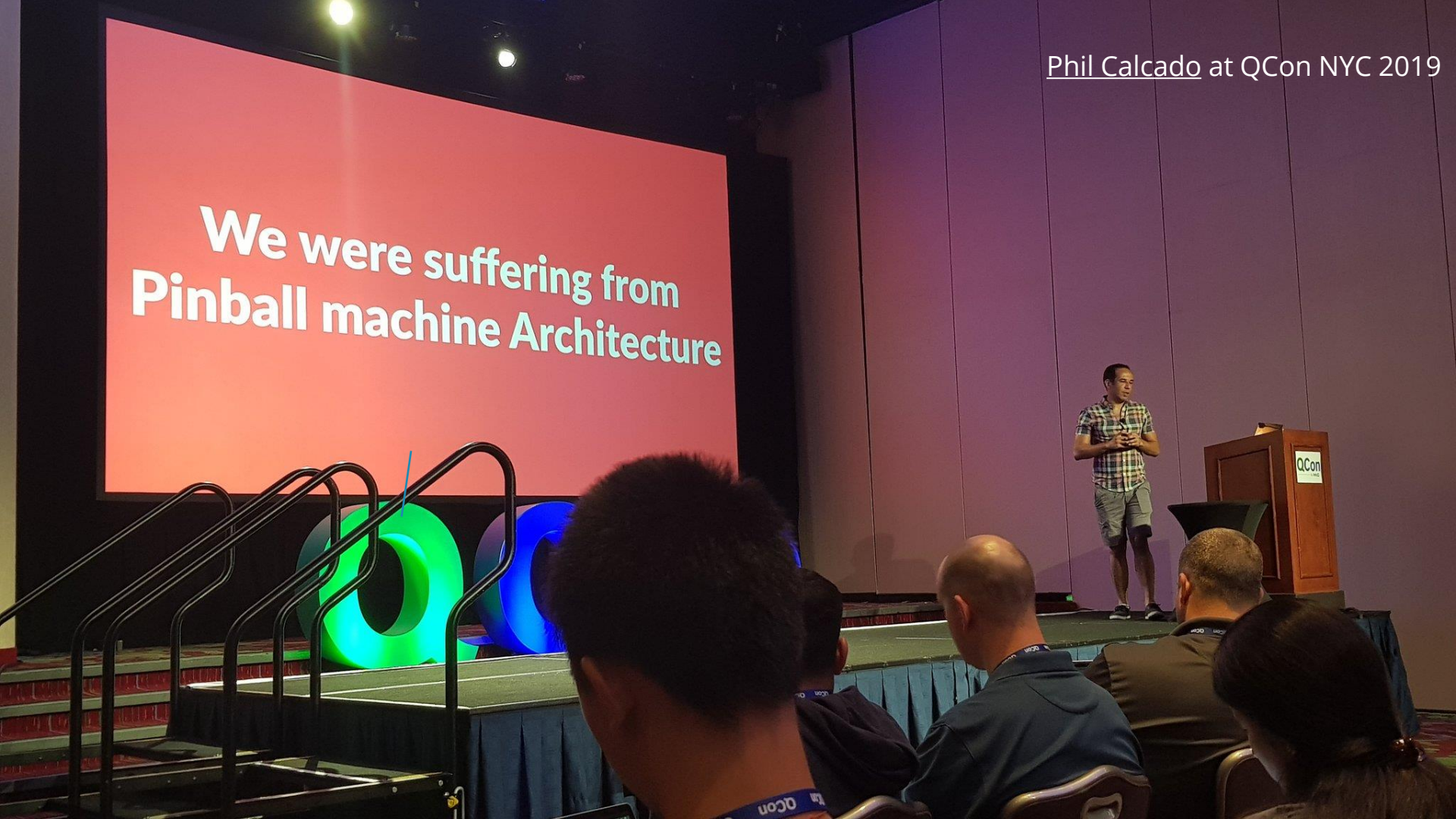


# Peer-to-peer event chains










Phil Calcado at QCon NYC 2019

# Pinball Machine Architecture

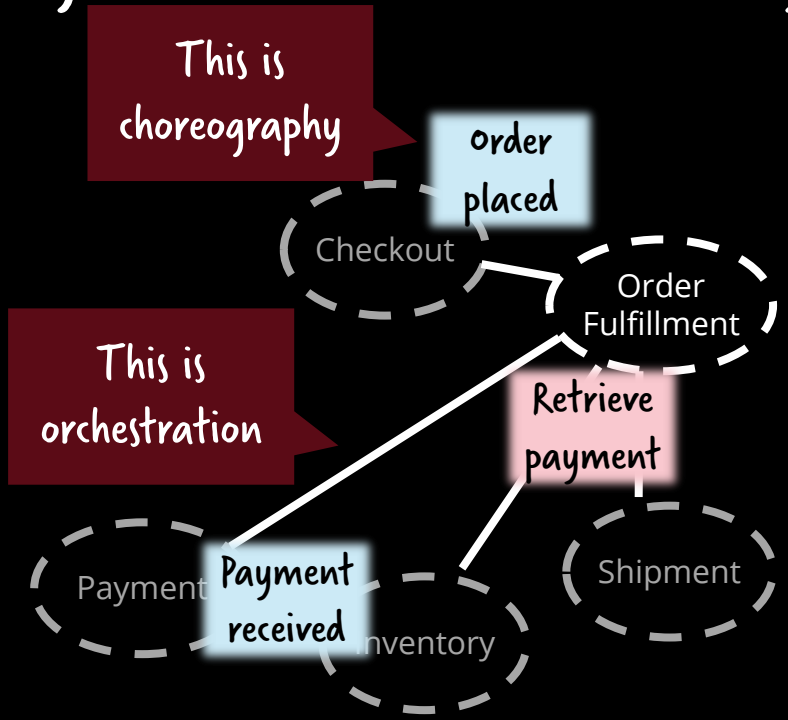




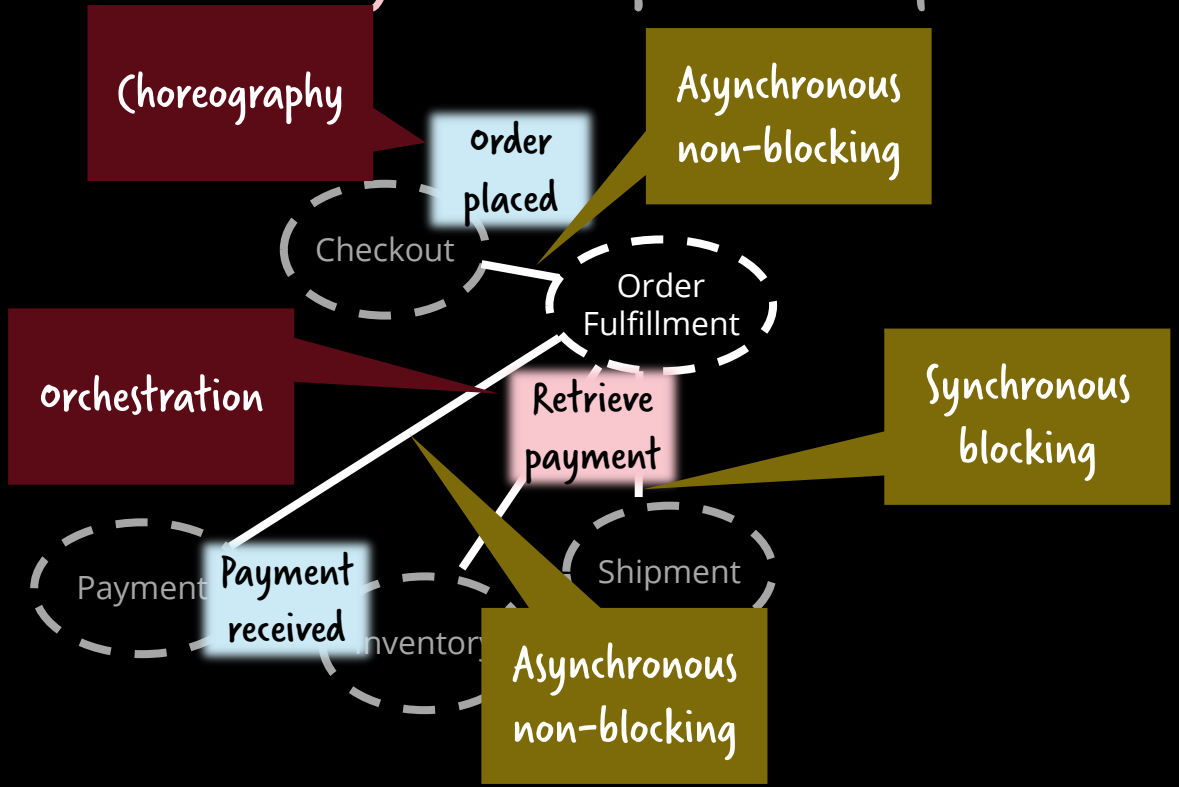
What we wanted

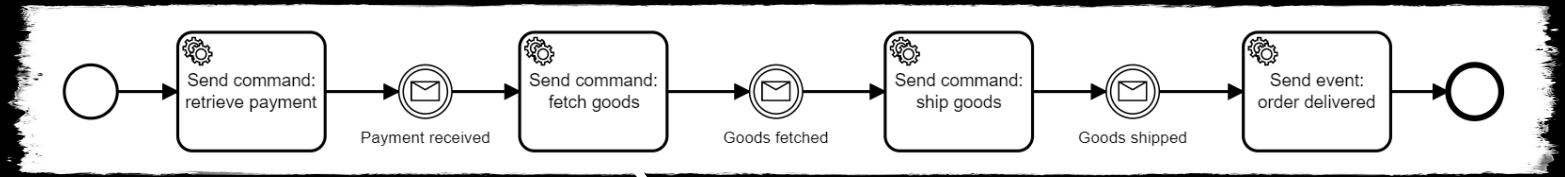
vs. what we got

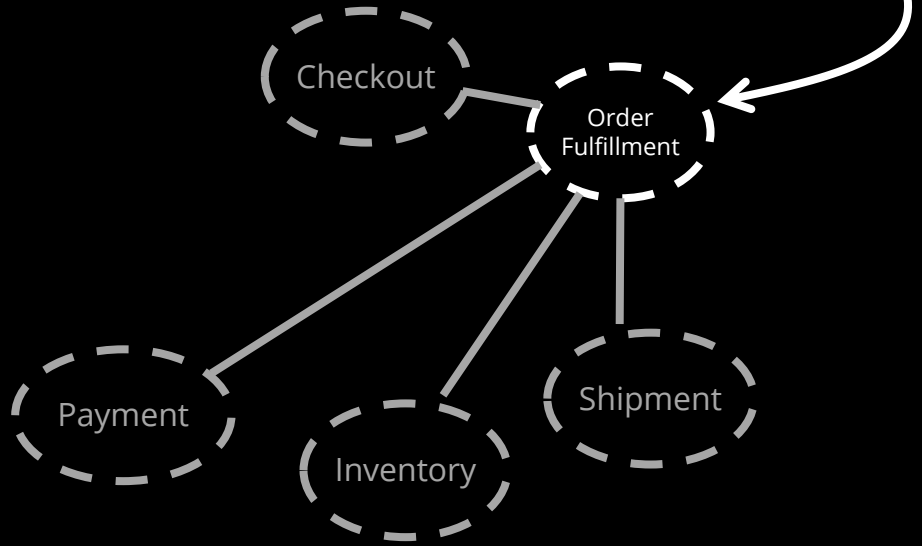
# Using orchestration and choreography



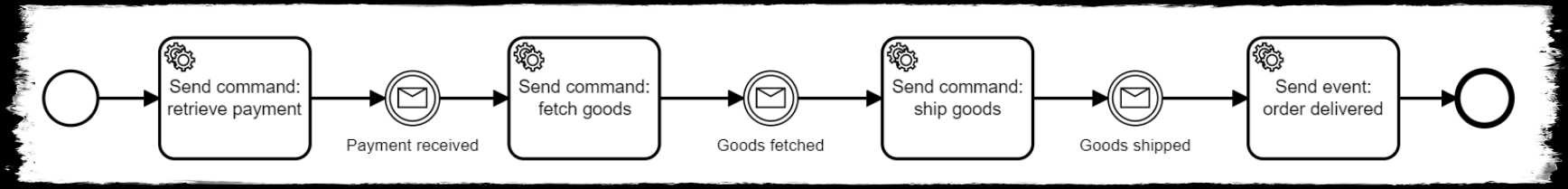
# Collaboration style is independant of communication style

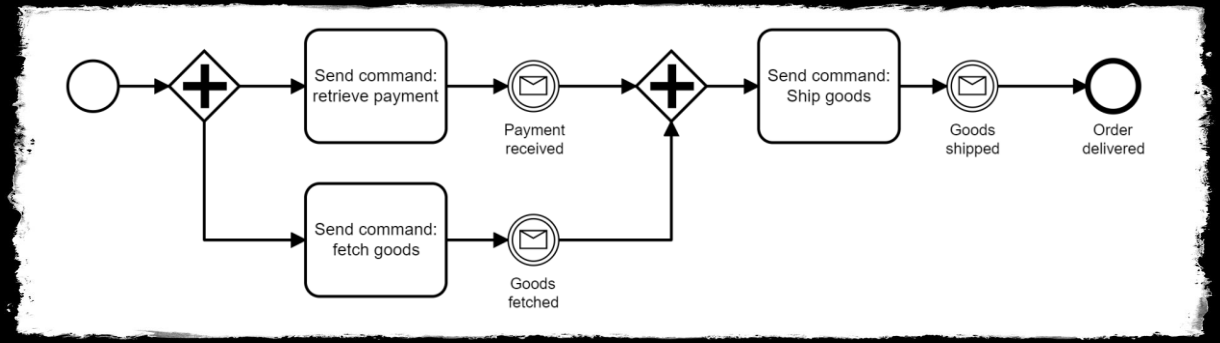



Send command: retrieve payment
Payment received
Send command: fetch goods
Goods fetched
Send command: ship goods
Goods shipped
Send event: order delivered

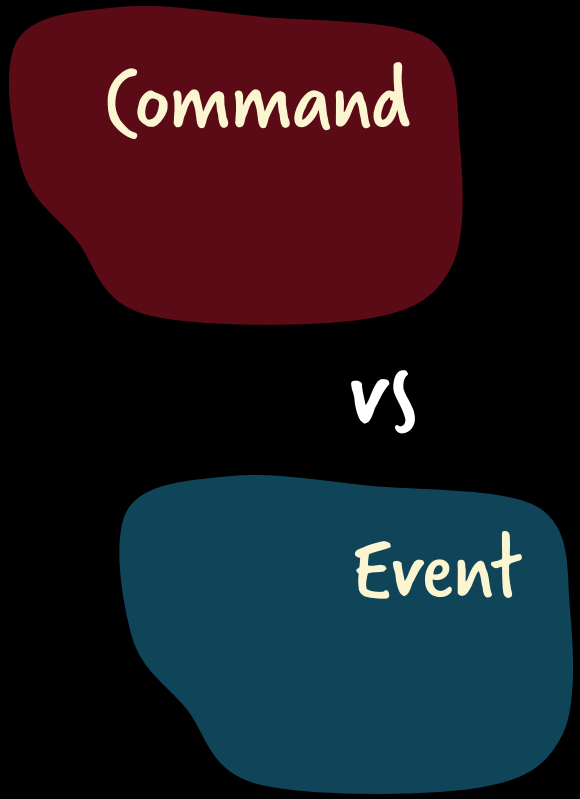

Checkout
Order Fulfillment
Payment
Inventory
Shipment

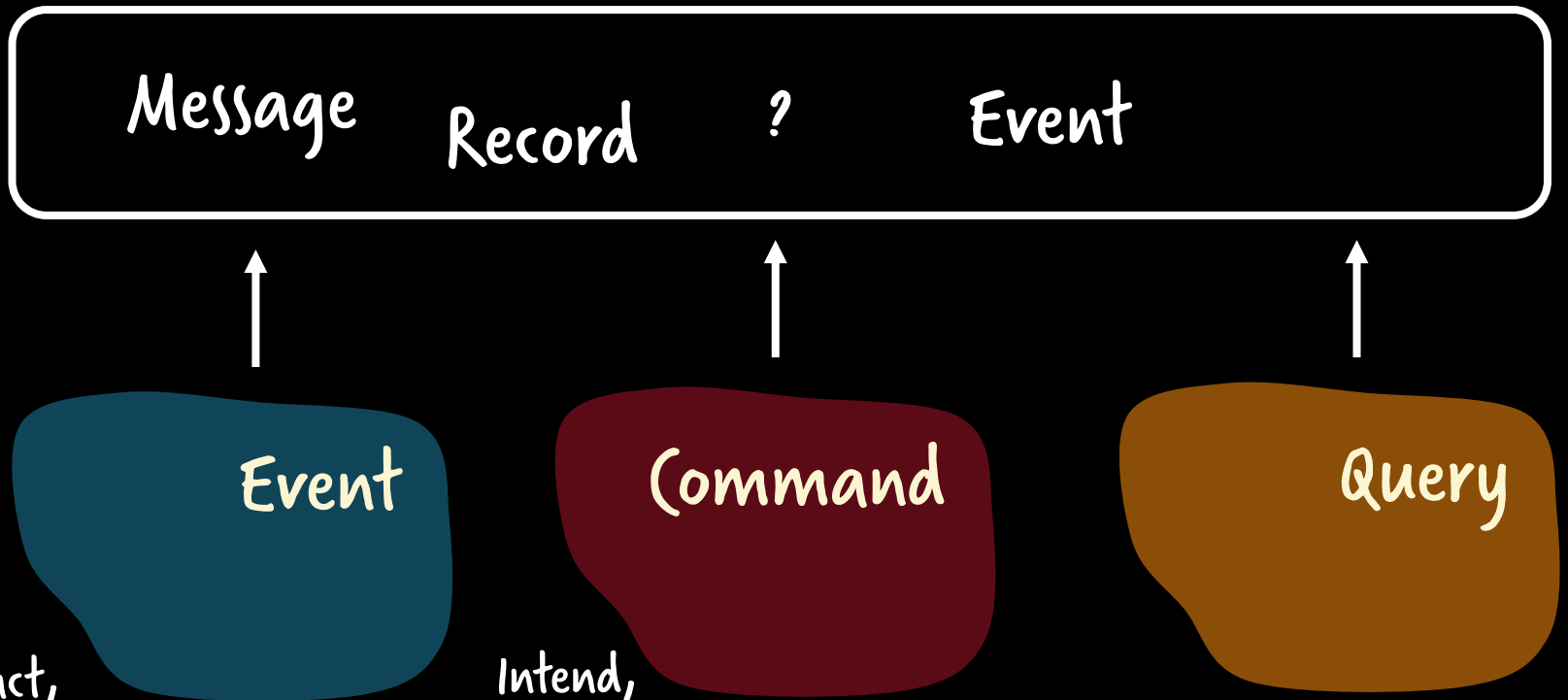
# Now it is easy to change the process flow

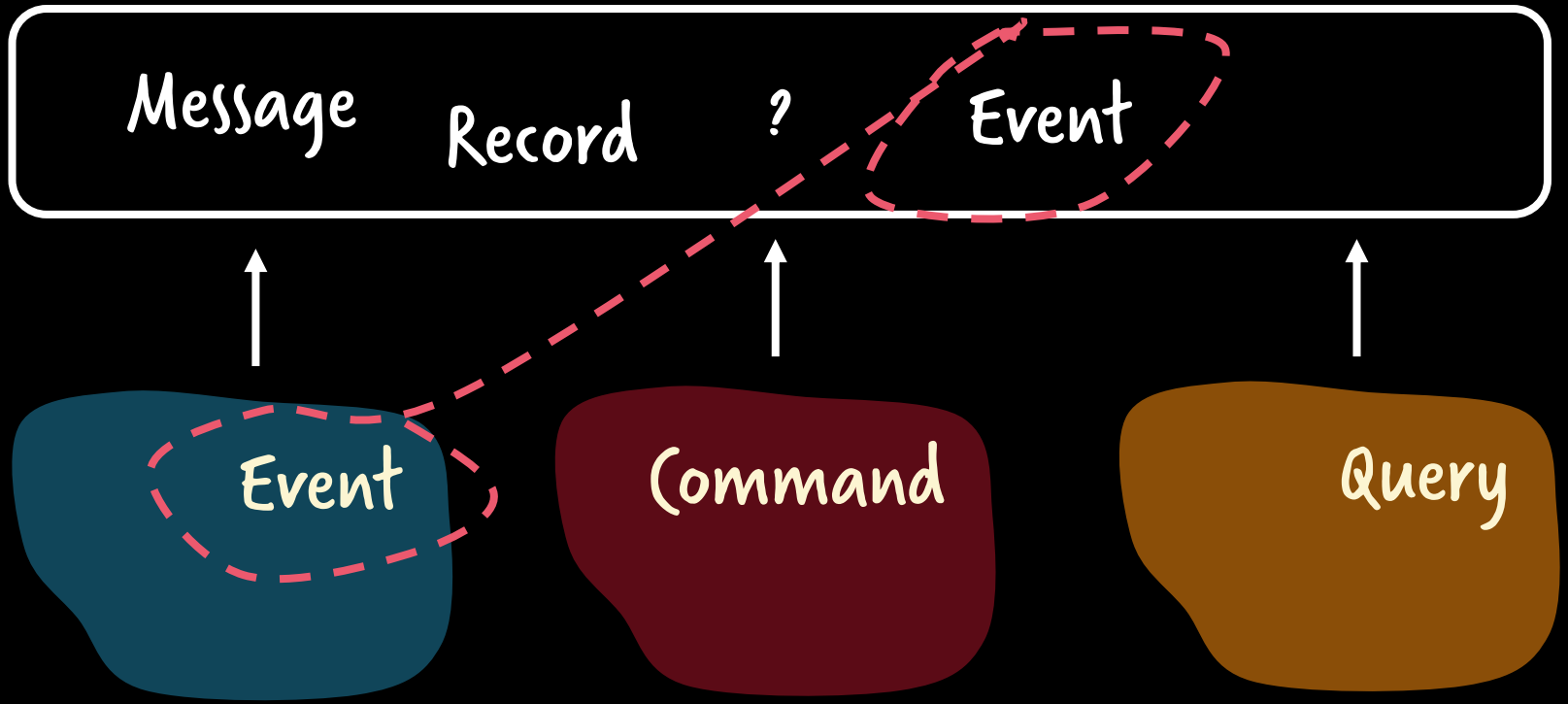


# Challenge: Command vs. Event




Message
Record
?
Event
Event
Command
Query
Fact,
happened in the past,
immutable
Intend,
Want s.th. to happen,
The intention itself is a fact


Message
Record
?
Event
Event
Command
Query

Commands in disguise

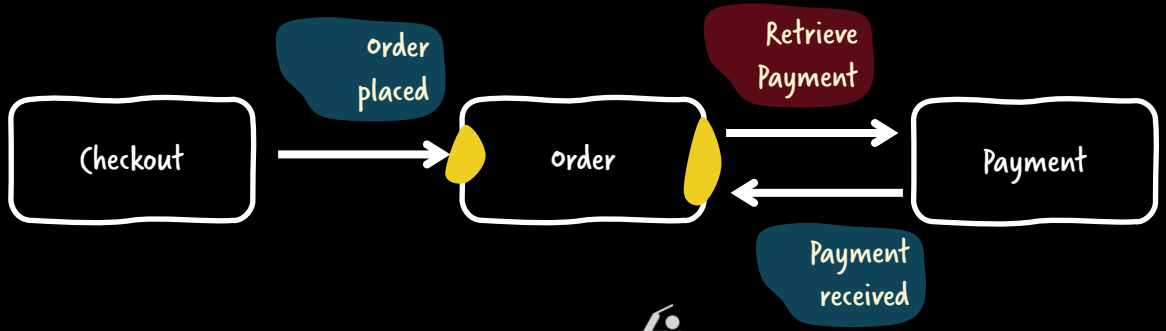
Wording of recipient

Send Message

The Customer Needs To Be Sent A Message To Confirm Address Change Event

Wording of Sender

# Direction of dependency

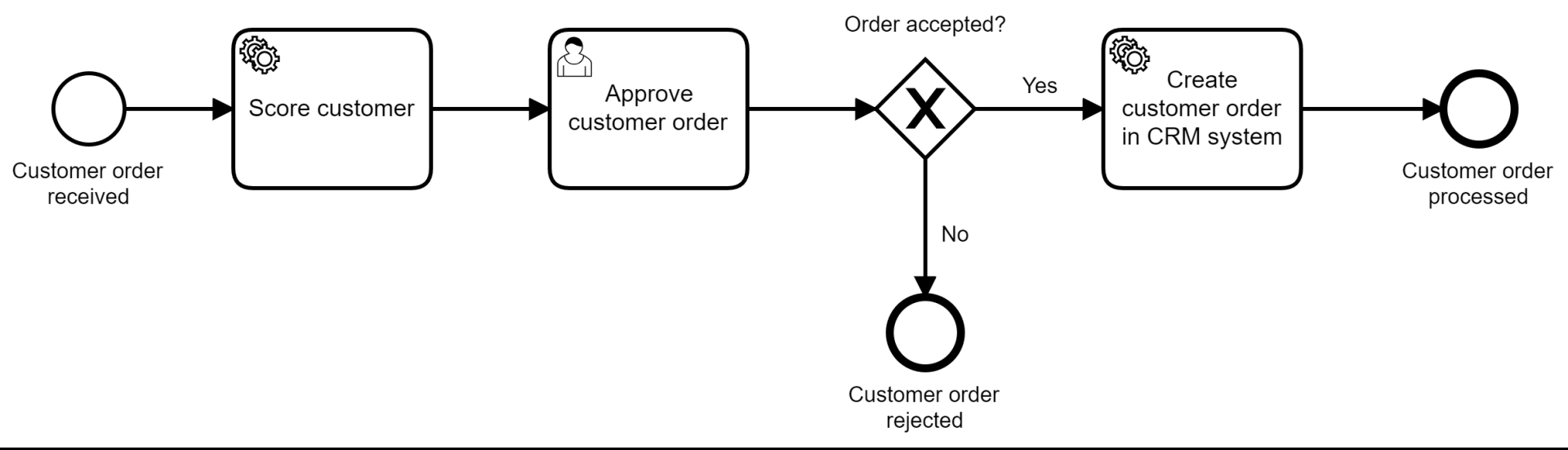



Event-driven:
Decision to couple is on the **receiving** side

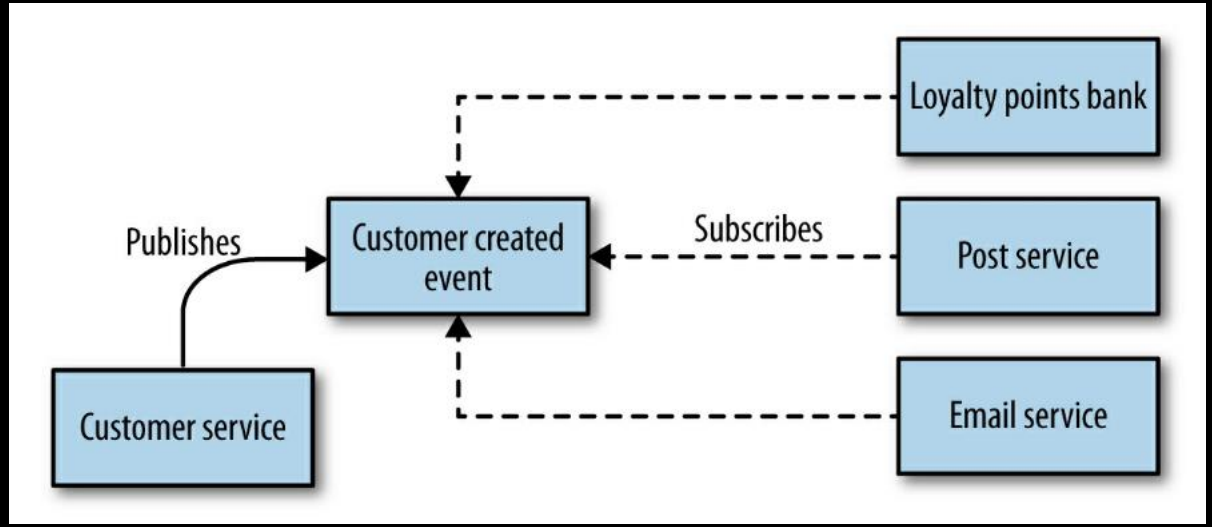
Command-driven
Decision to couple is on the **sending** side

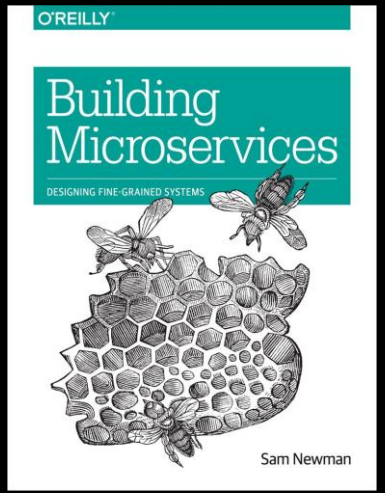
Direction of dependency

# Customer onboarding



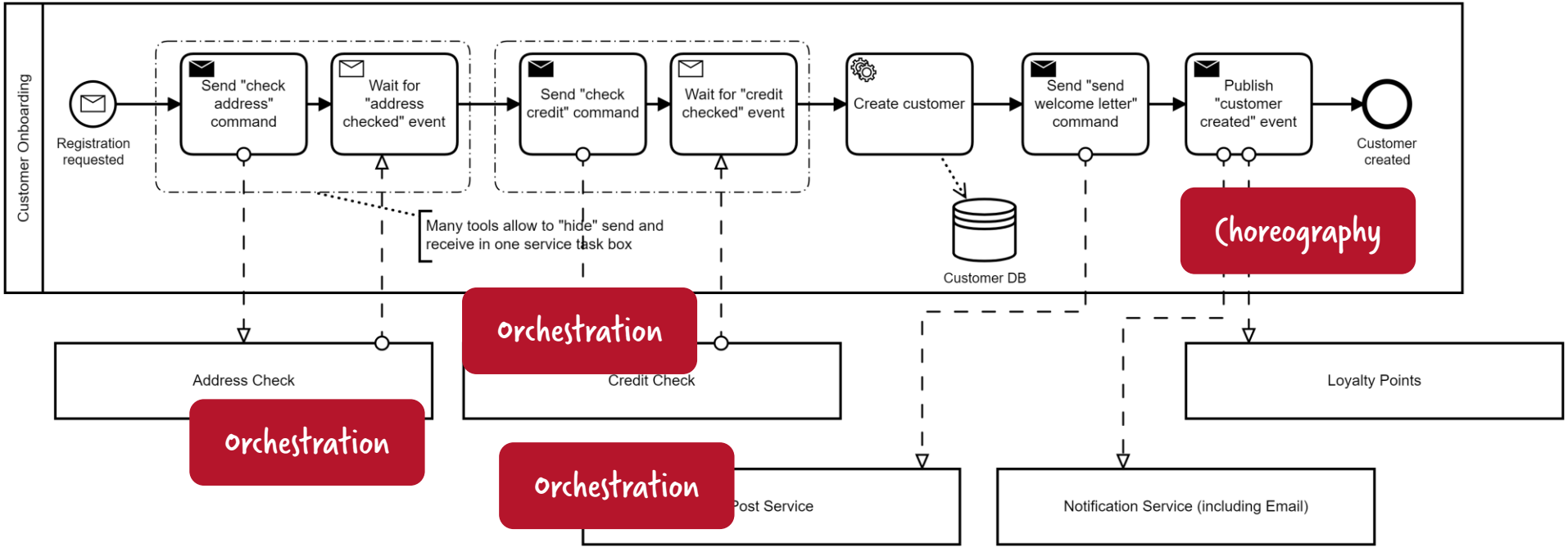

https://github.com/berndruecker/customer-onboarding-camunda-8-springboot



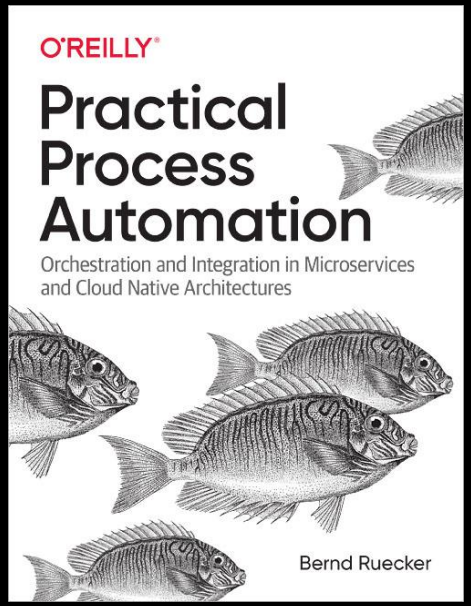

Sam Newman: Building Microservices

# Mix orchestration and choreography

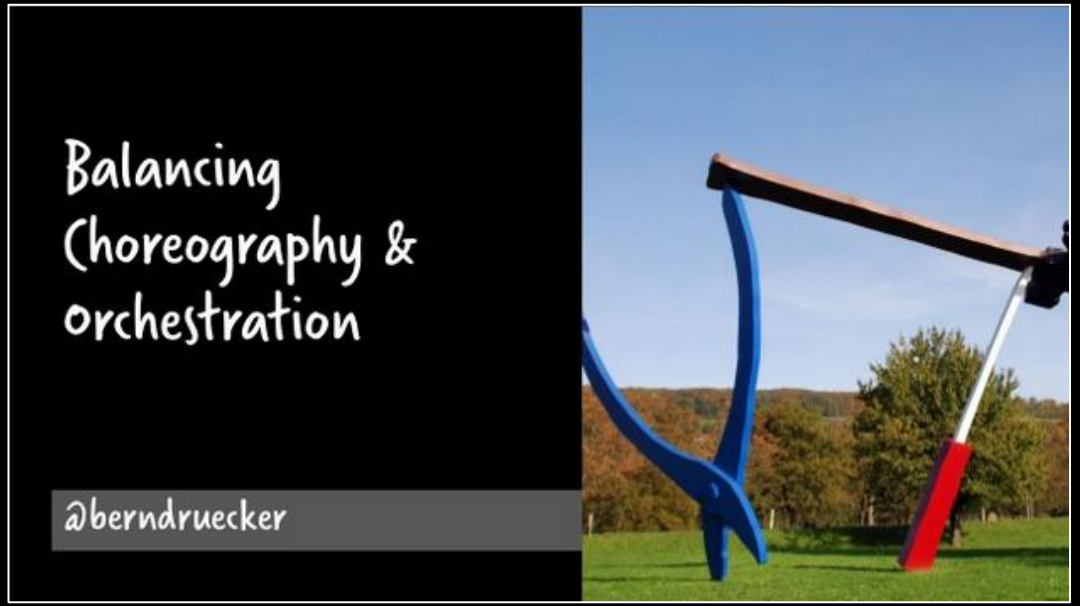


# Want to learn more about choreography vs. orchestration?

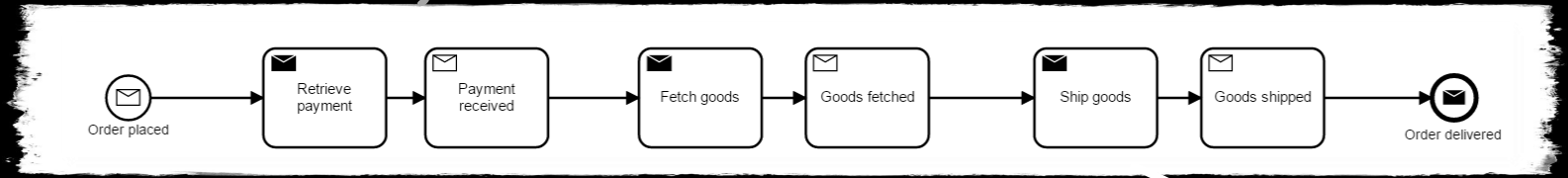
http://berndruecker.io




https://processautomationbook.com/



# Processes are domain logic and live inside service boundaries



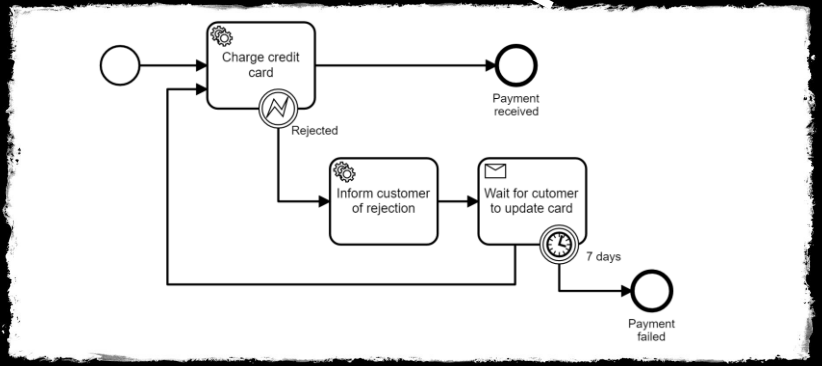

Checkout

Order

Payment

Inventory

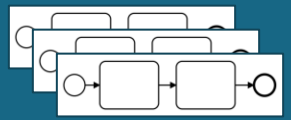
Shipment



# orchestration is not centralized – PaaS operations might be

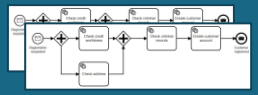
Every microservice (process solution) owns its process model, glue code, and any additional artifacts

Microservice A



Microservice B

Microservice C



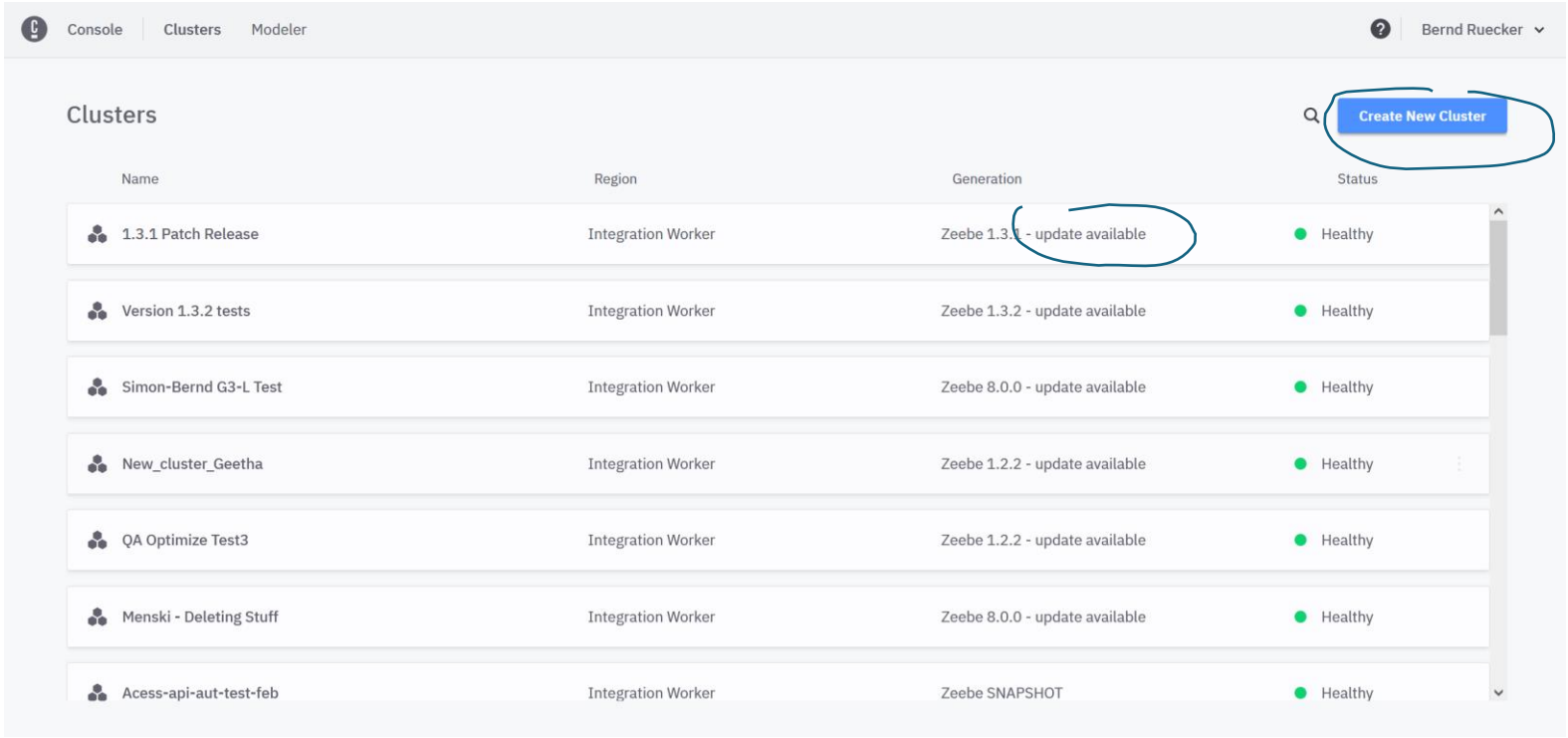
Orchestration Engine

Orchestration Engine

Orchestration Engine

Ideally SaaS
(probably self-managed)

# Example: Self-service control plane



# Some code?

berndruecker / flowing-retail Public

Unpin | Unwatch 116 | Fork 420 | Starred 1.2k

Code | Issues 6 | Pull requests 14 | Discussions | Actions | Projects | Wiki | Security 56 | Insights

master | flowing-retail / kafka / java / | Go to file | Add file

berndruecker Added payment microservice alternative using Zeebe (related to #73) 27bc0ee 7 days ago History

| | | |
|---|---|---|
| README.md | adjusted readme to latest version/ports | 7 days ago |
| pom.xml | added build for event ingestion to CI | 2 years ago |

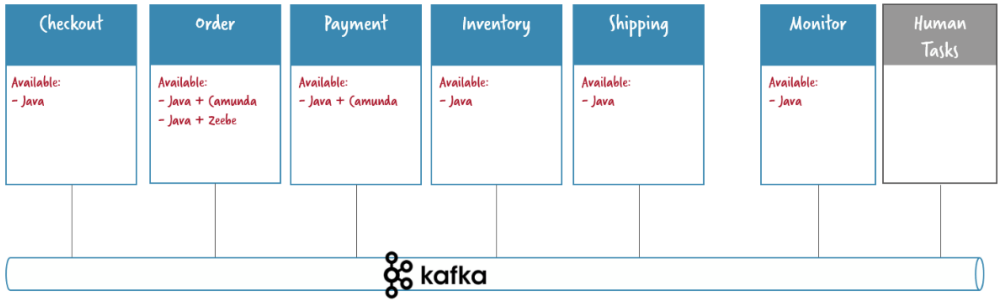
README.md

## Flowing Retail / Apache Kafka / Java

This folder contains services written in Java that connect to Apache Kafka as means of communication between the services.

Tech stack:

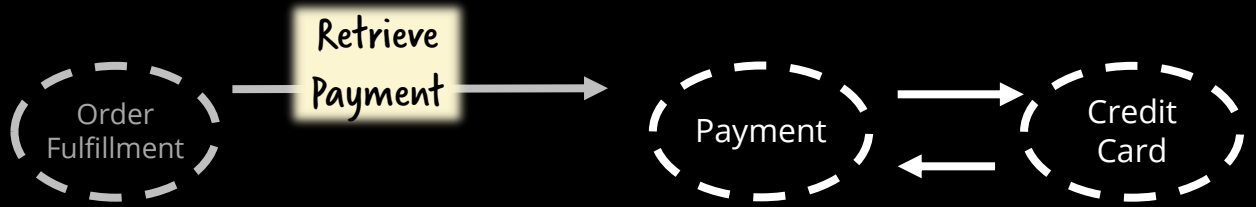
- Java 8
- Spring Boot 2.6.x
- Apache Kafka (and Spring Kafka)
- Camunda Zeebe 8.x (and Spring Zeebe)



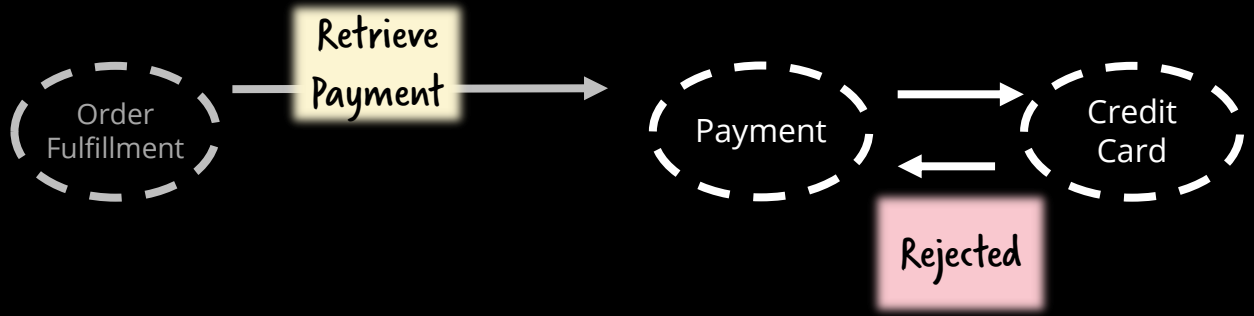

https://github.com/berndruecker/flowing-retail/tree/master/kafka

# Let's talk about long running...

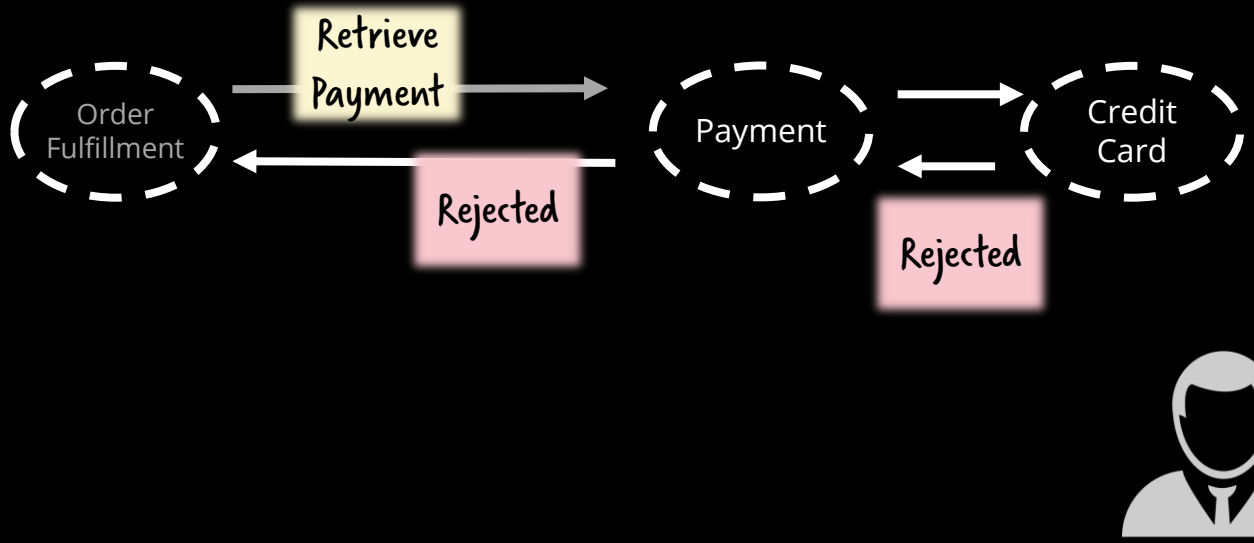
# Example



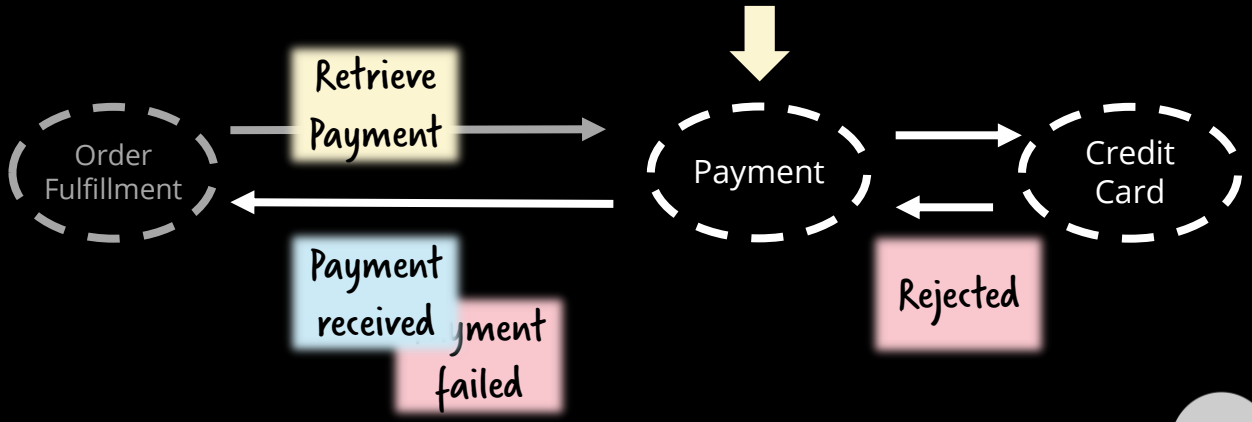
# Example


Order Fulfillment
Retrieve Payment
Payment
Credit Card
Rejected

# Example

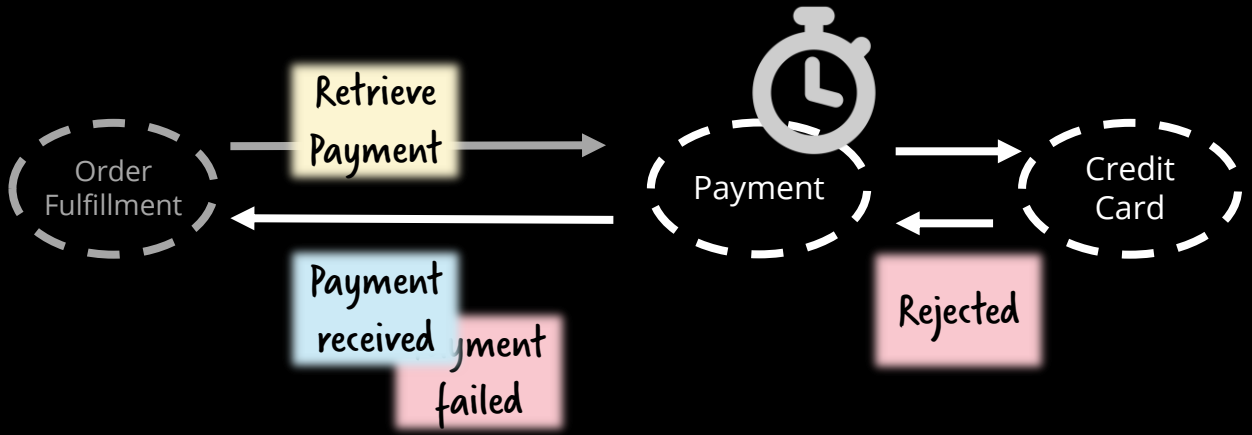

Retrieve Payment
Order Fulfillment
Payment
Credit Card
Rejected
Rejected
If the credit card was rejected, the customer can provide new details

# Who is responsible?

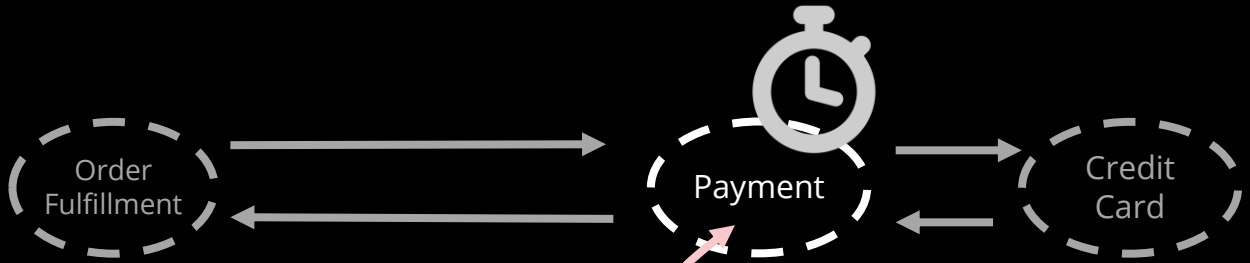


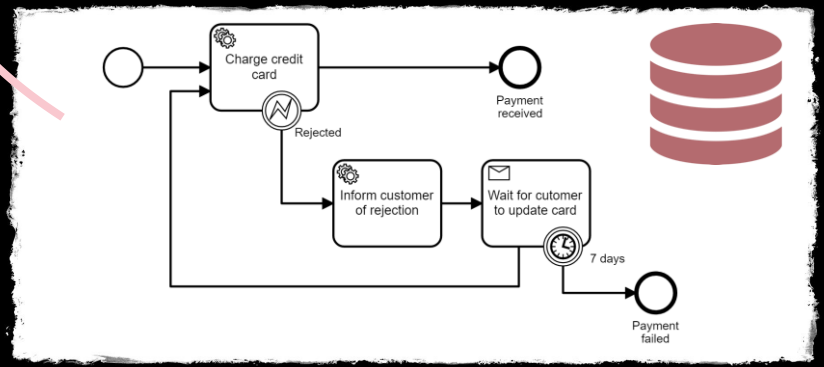

If the credit card was rejected, the customer can provide new details

# Long running services

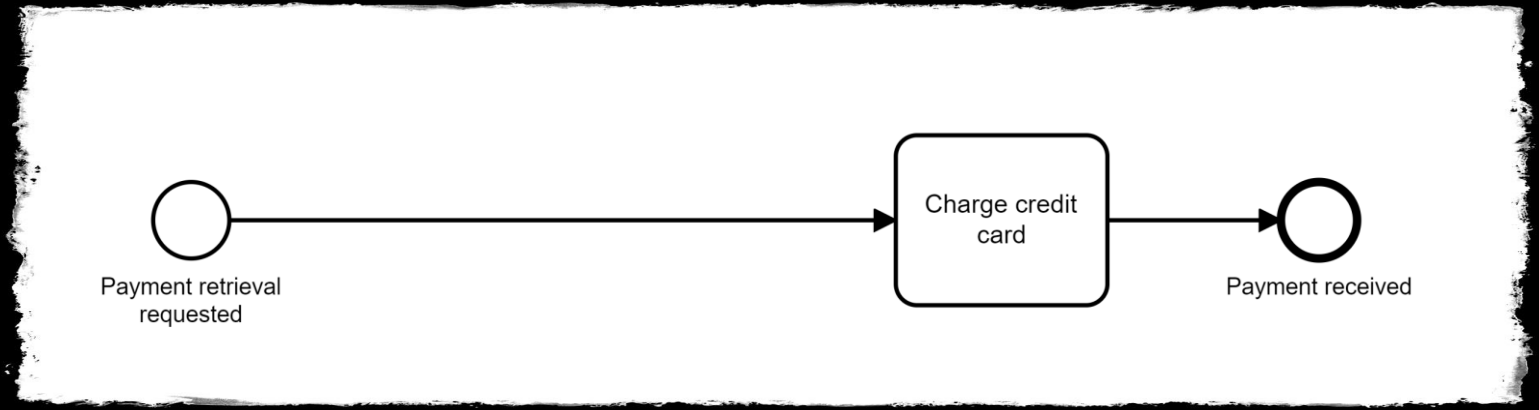


# Long running services

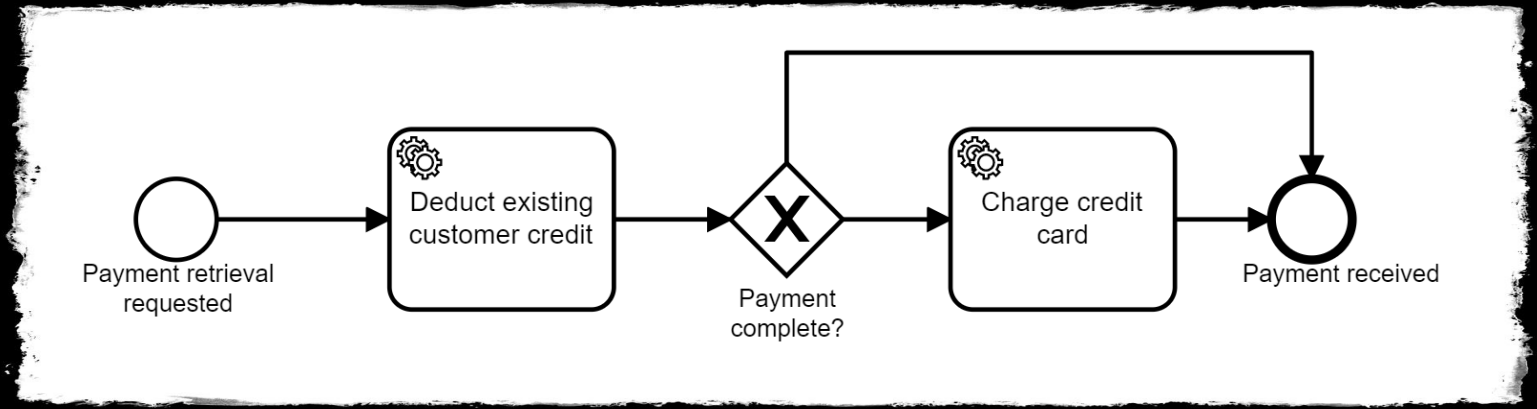


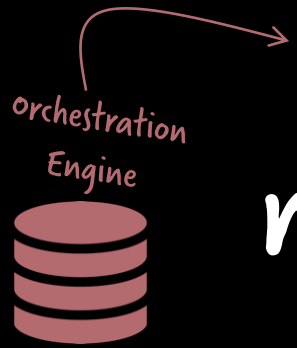
# Long Running Services Allow More Flexible Changes



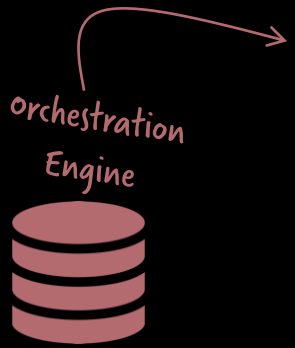
# Long Running Services Allow More Flexible Changes



Being able to implement
long running services
makes it easy to distribute
responsibilities correctly

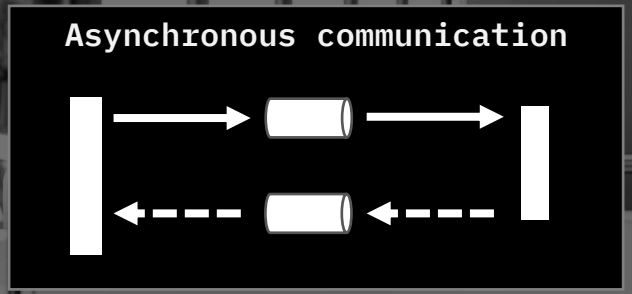


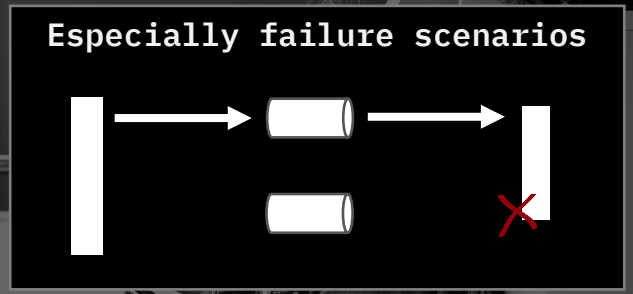
Being able to implement long running services makes it easy to embrace async/non-blocking



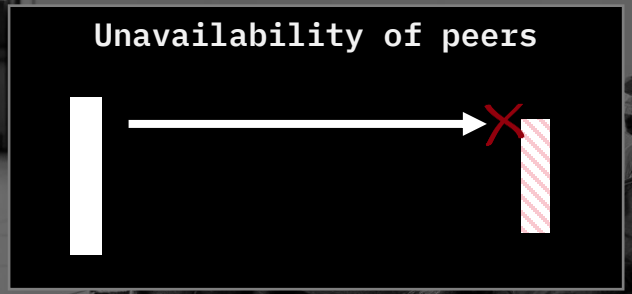
# When do services need to wait? Some technical reasons...

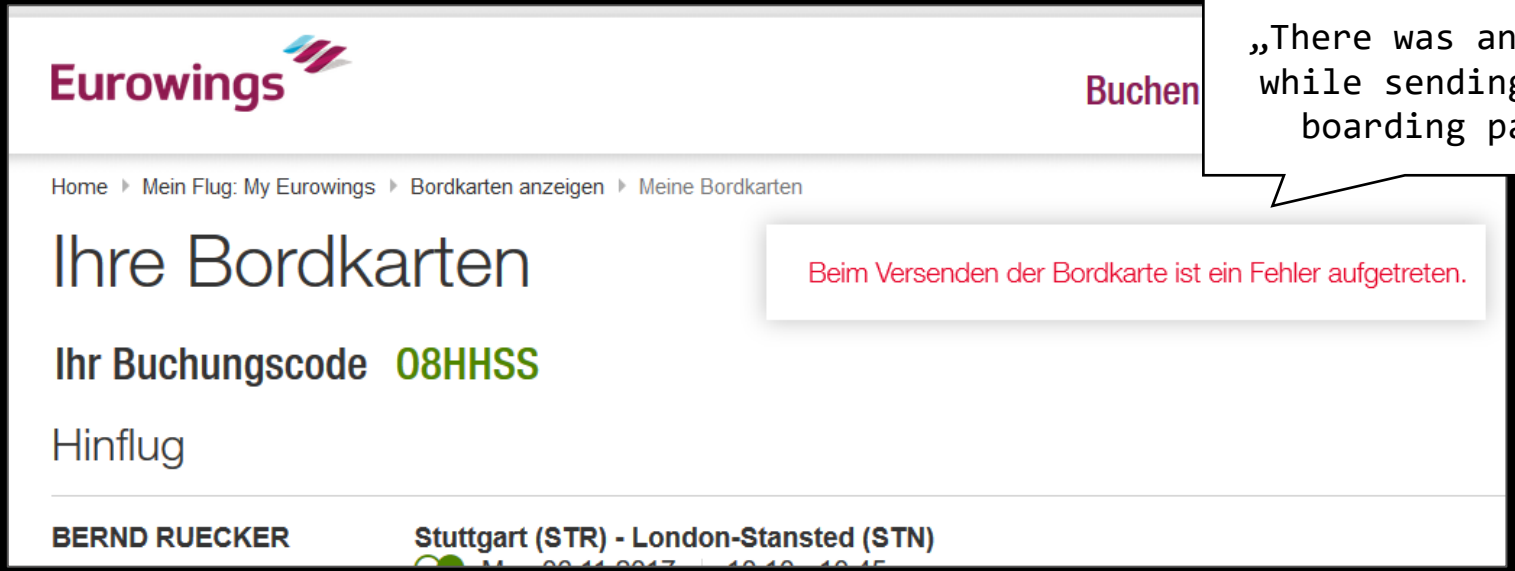
Wait for responses



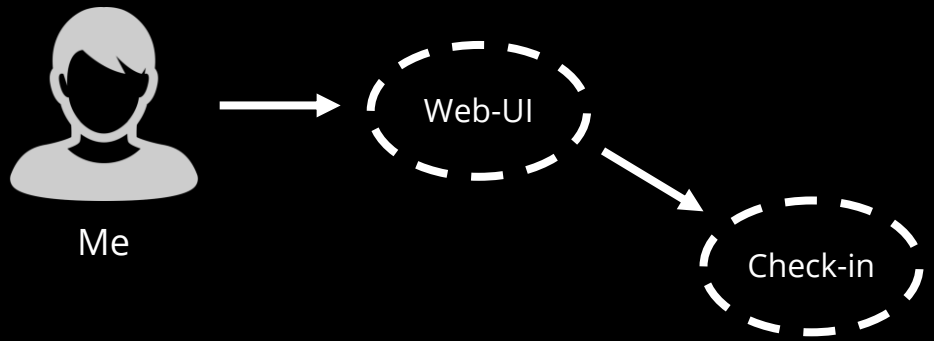

Wait for availability



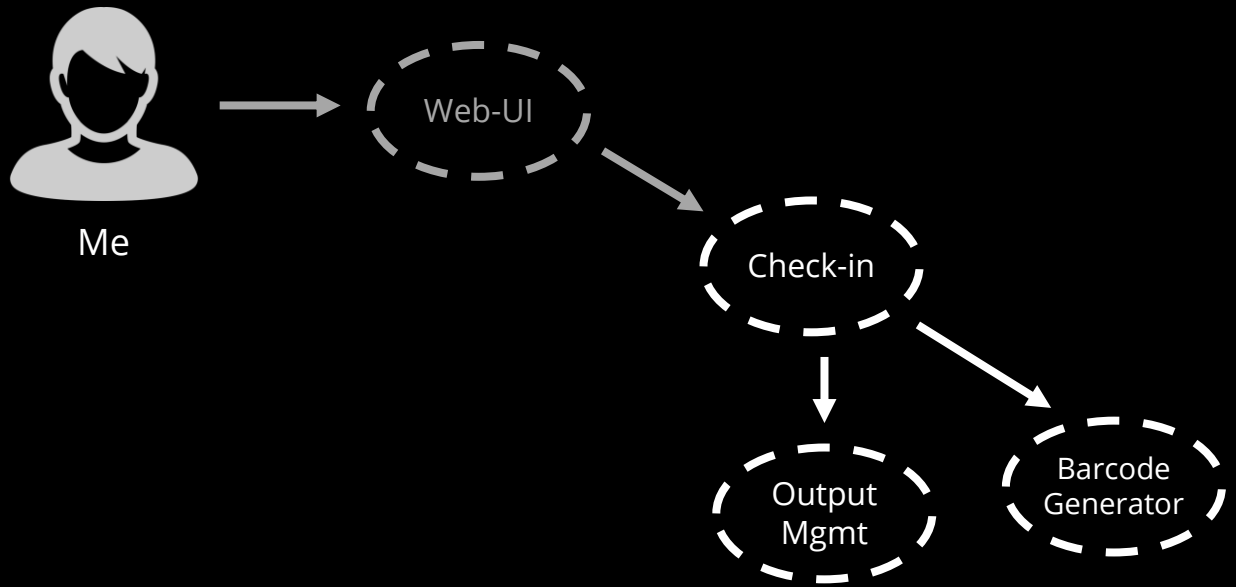



„There was an error while sending your boarding pass“

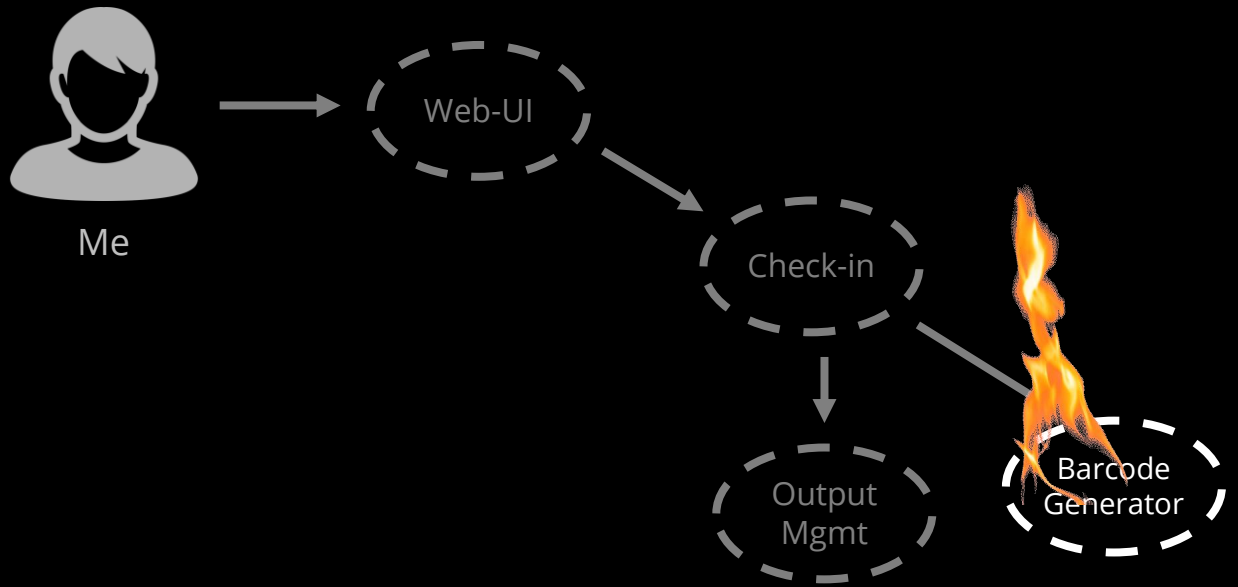
# Current situation

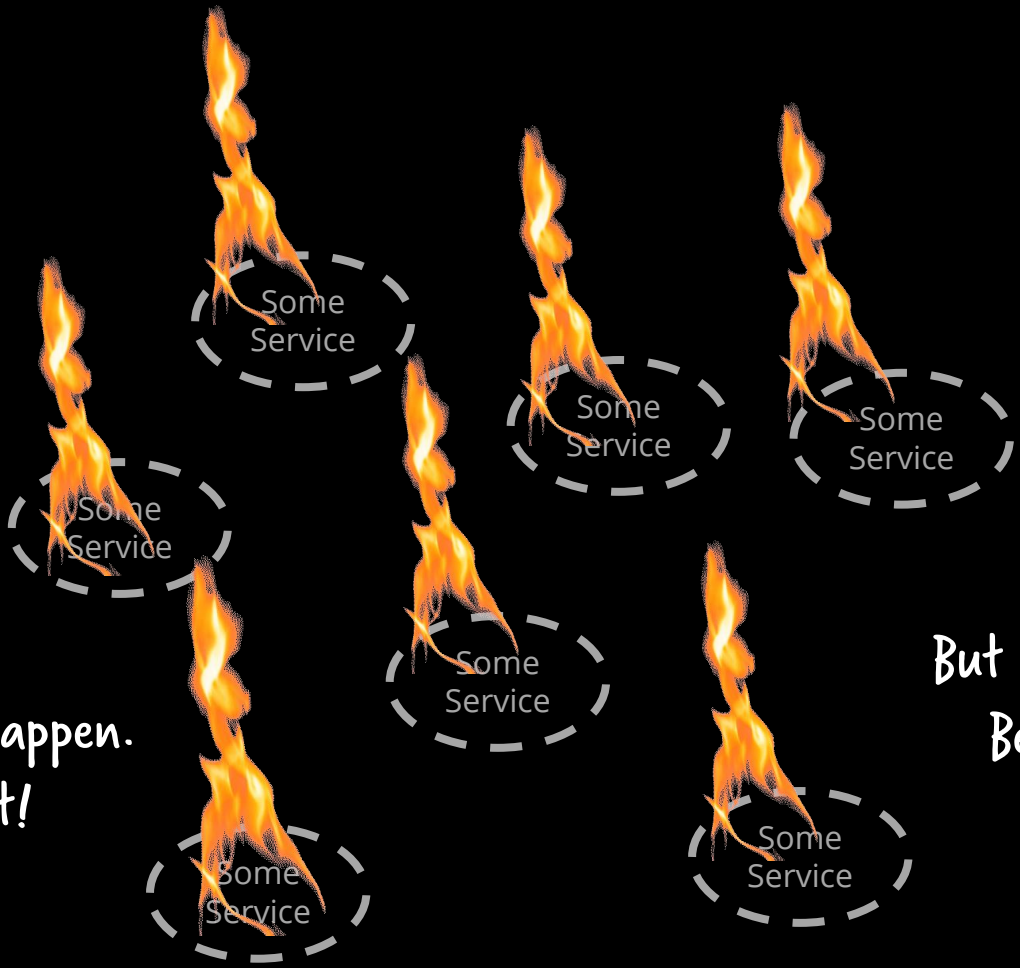


# Current situation



# Current situation

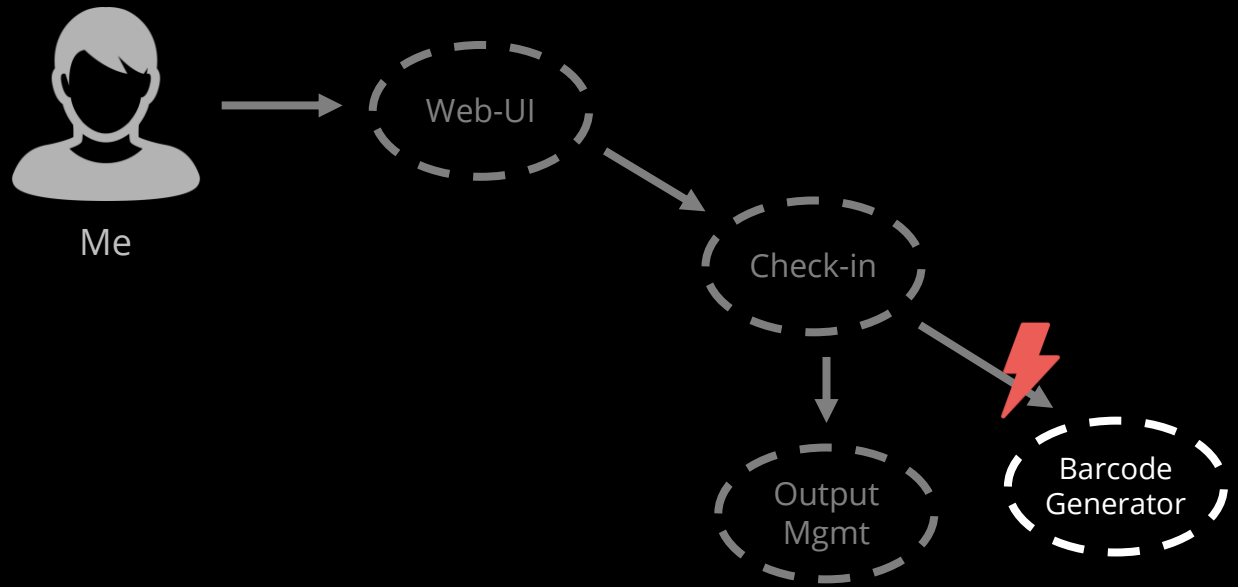




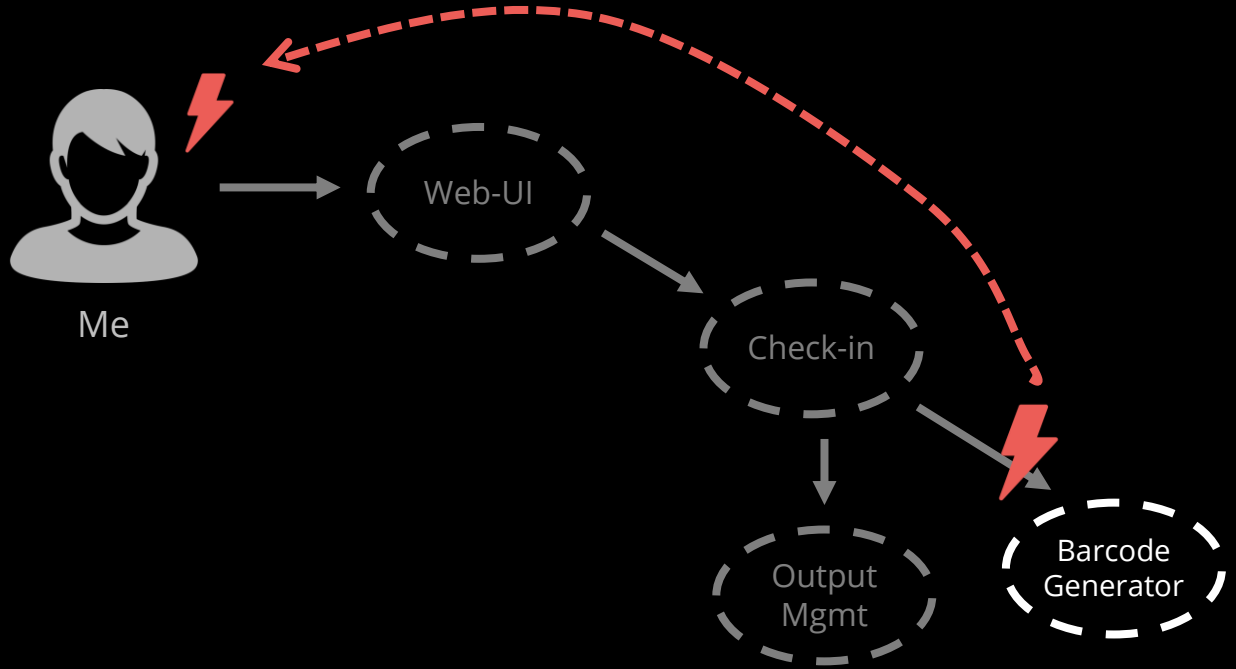

Failure will happen.
Accept it!

But keep it local!
Be resilient.

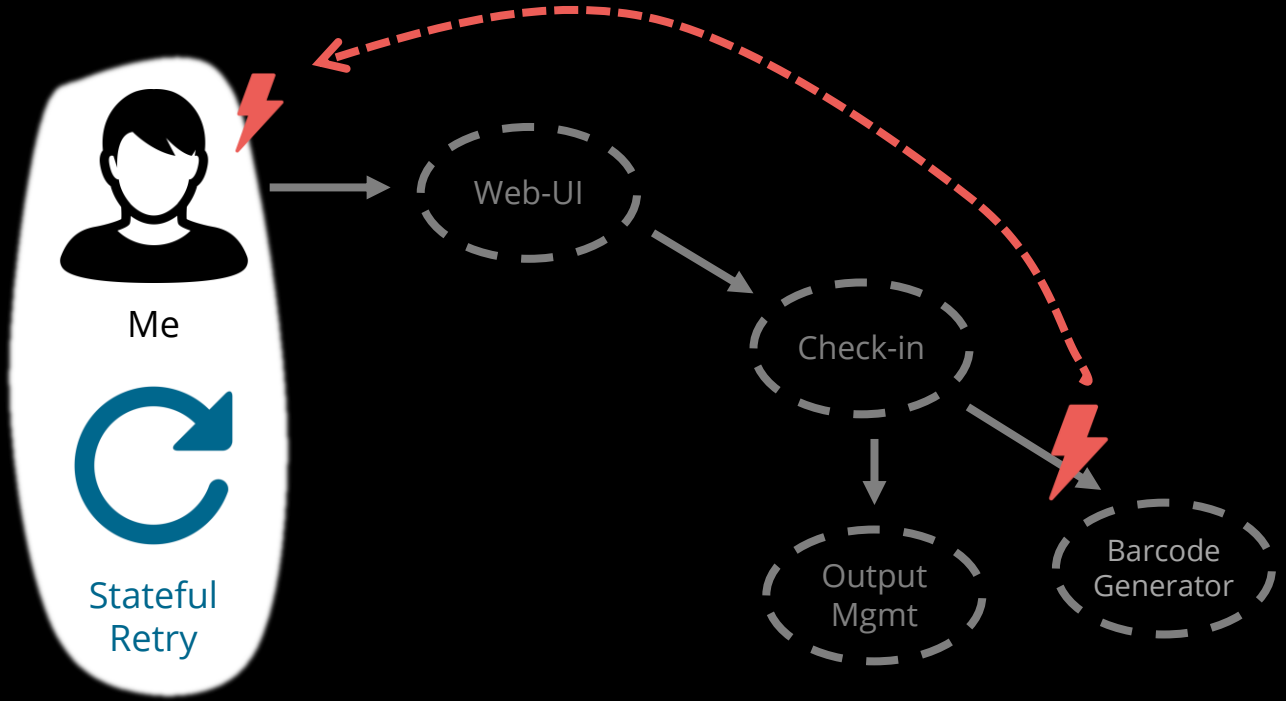
# Current situation – the bad part

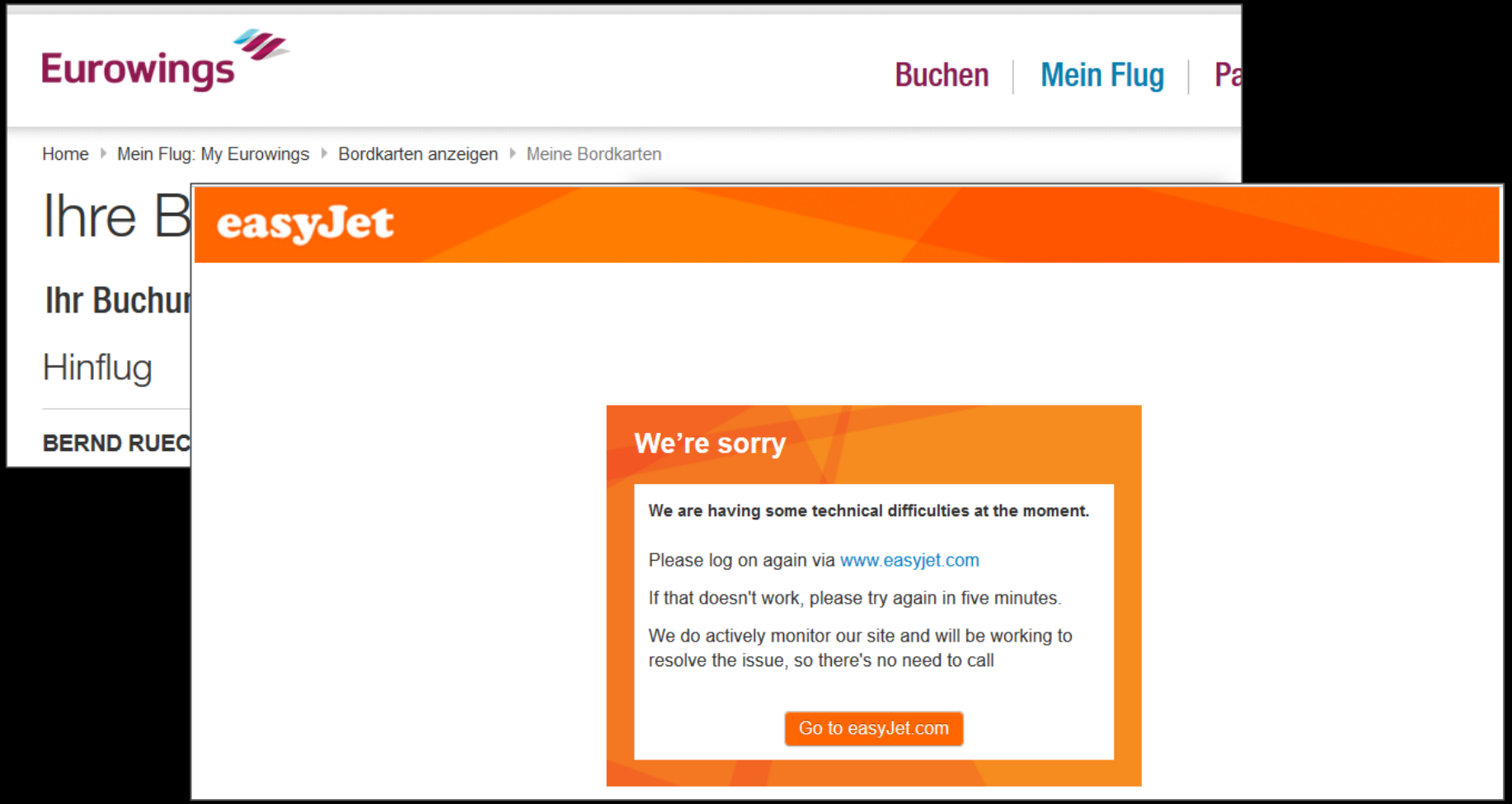


# Current situation – the bad part



# Current situation – the bad part




Eurowings
Buchen
Mein Flug
Home
Mein Flug: My Eurowings
Bordkarten anzeigen
Meine Bordkarten
Ihre B
Ihr Buchu
Hinflug
BERND RUEC
easyJet
We're sorry
We are having some technical difficulties at the moment.
Please log on again via www.easyjet.com
If that doesn't work, please try again in five minutes.
We do actively monitor our site and will be working to resolve the issue, so there's no need to call
Go to easyJet.com

easyJet

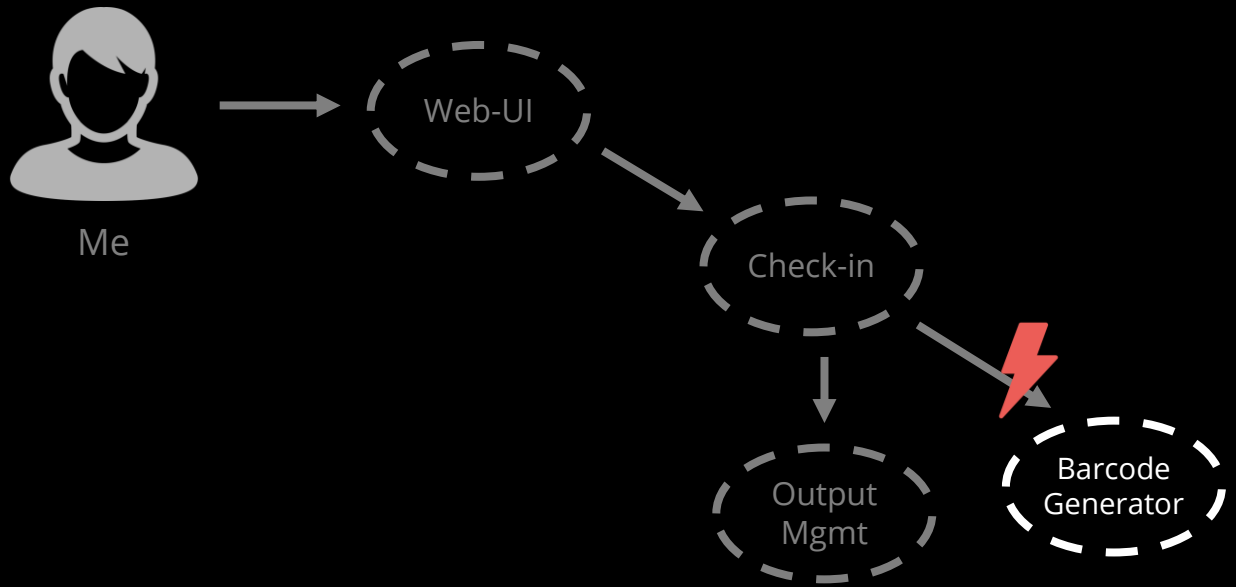
...I just made this up...

**We're sorry**

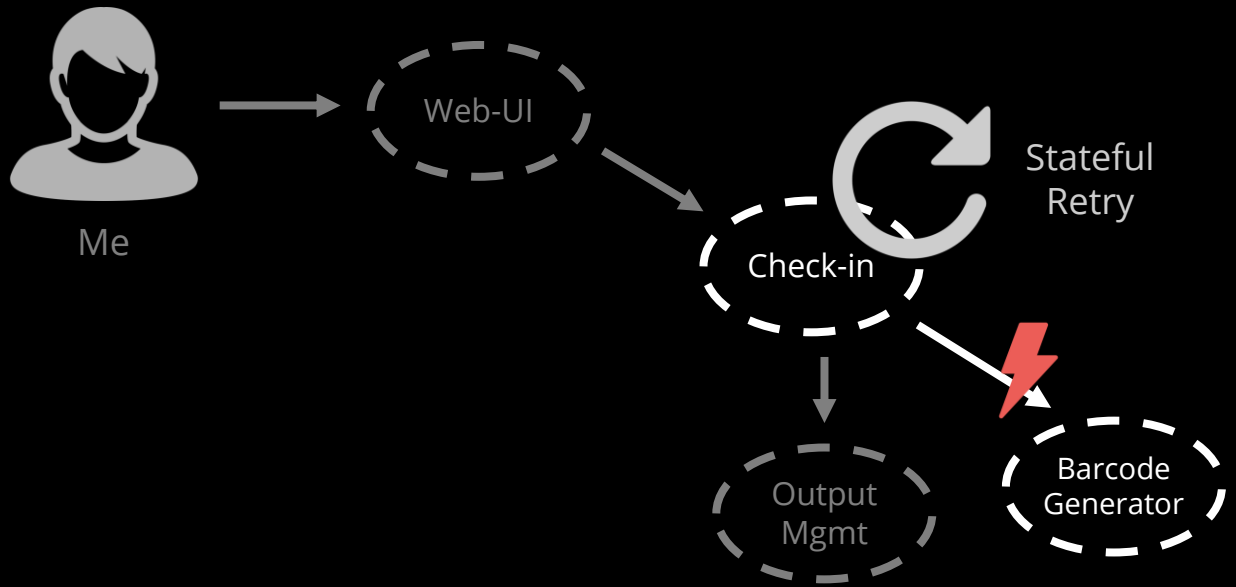
We are having some technical difficulties and cannot present you your boarding pass right away.

But we do actively retry ourselves, so lean back, relax and we will send it on time.

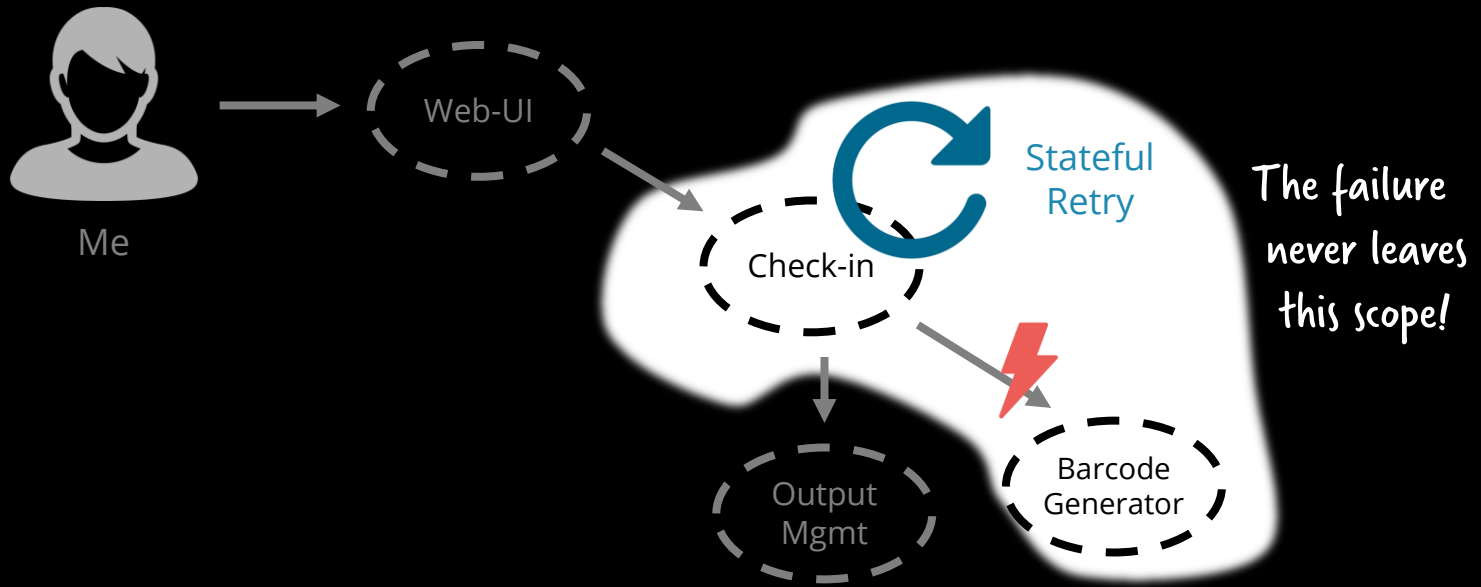
# Possible situation – much better!



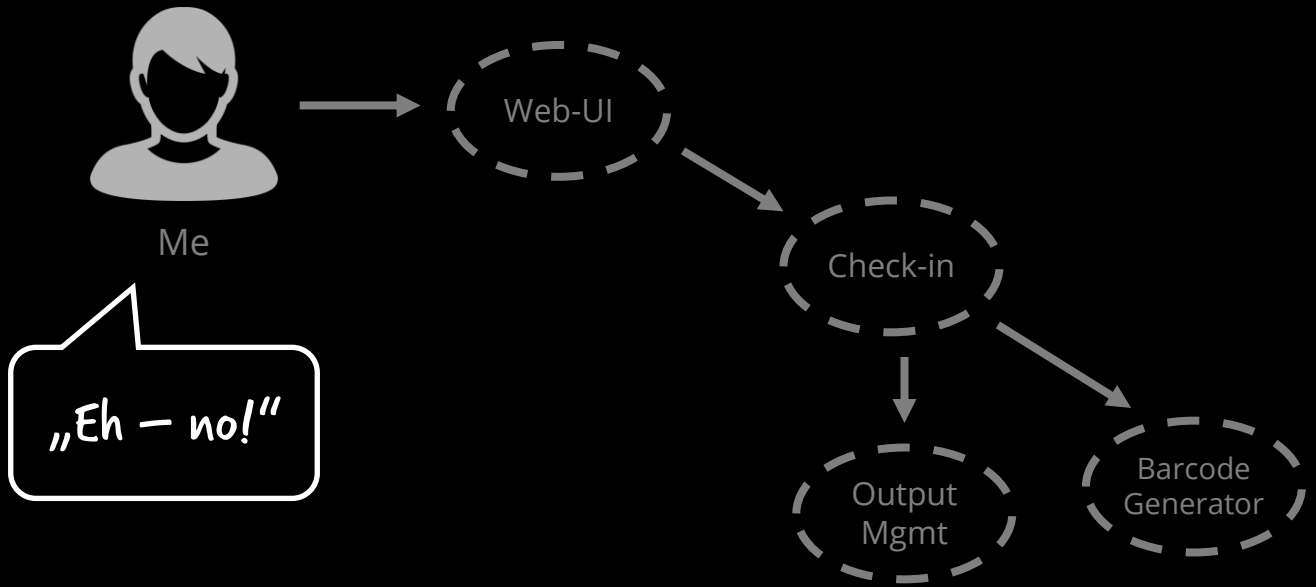
# Possible situation – much better!



# Possible situation – much better!



# „But the customer wants a synchronous response!“



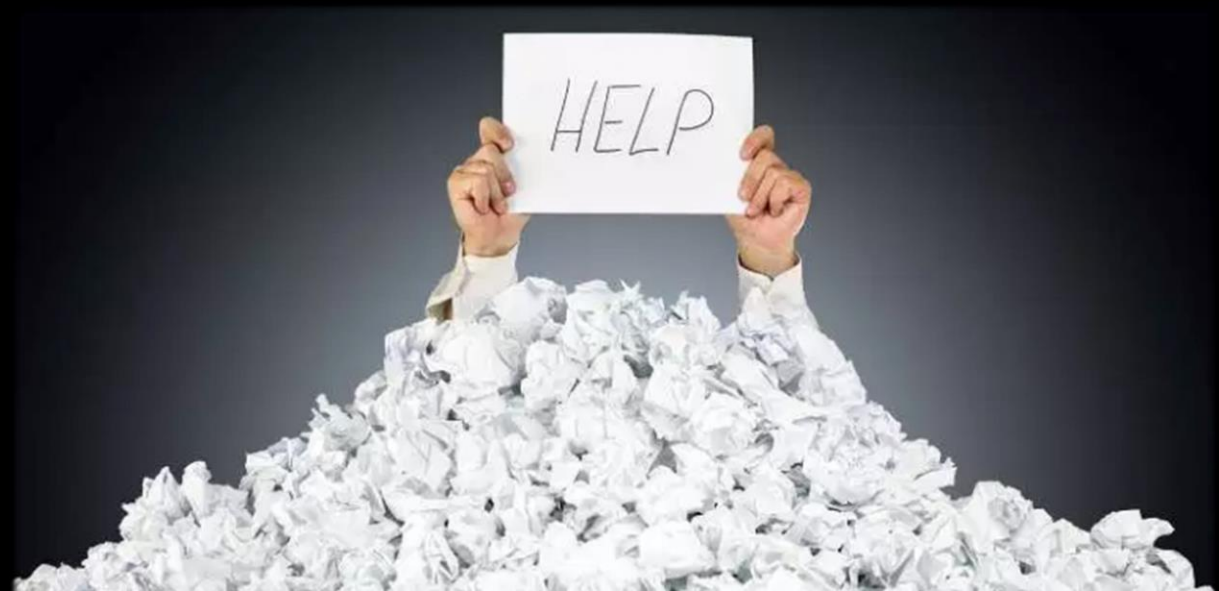
# Synchronous communication

Todd Montgomery and Martin Thompson
in "How did we end up here" at GOTO Chicago 2015

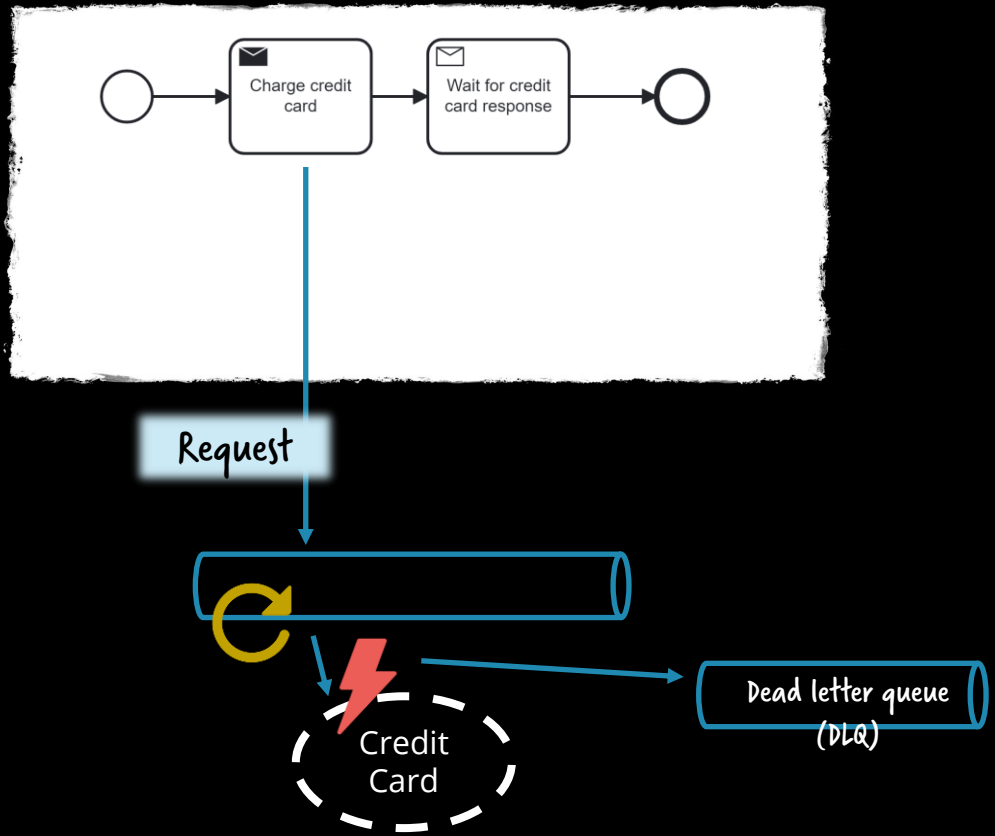
Synchronous communication is the crystal meth of distributed programming

Todd Montgomery and Martin Thompson
in "How did we end up here" at GOTO Chicago 2015

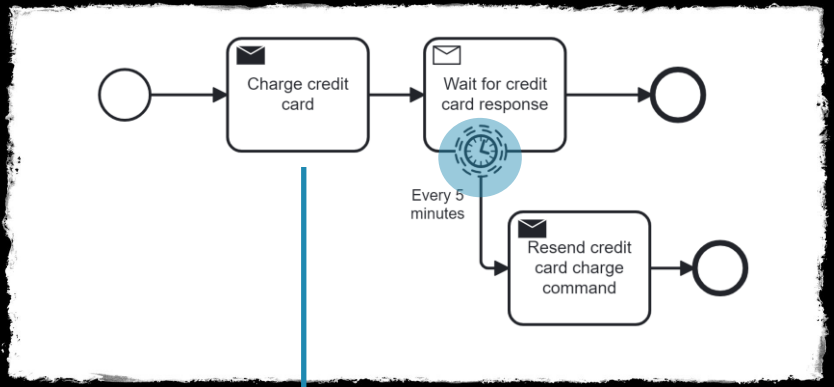
# Messaging?

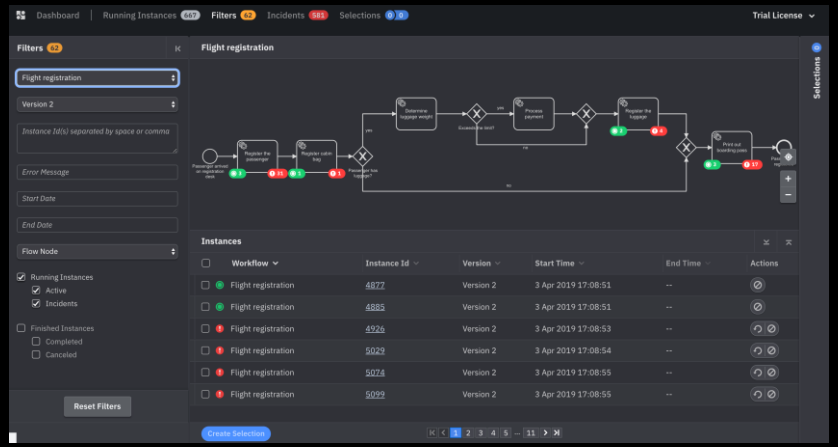


# Using messaging




# Using messaging




Request

Credit Card

Dead letter queue (DLQ)

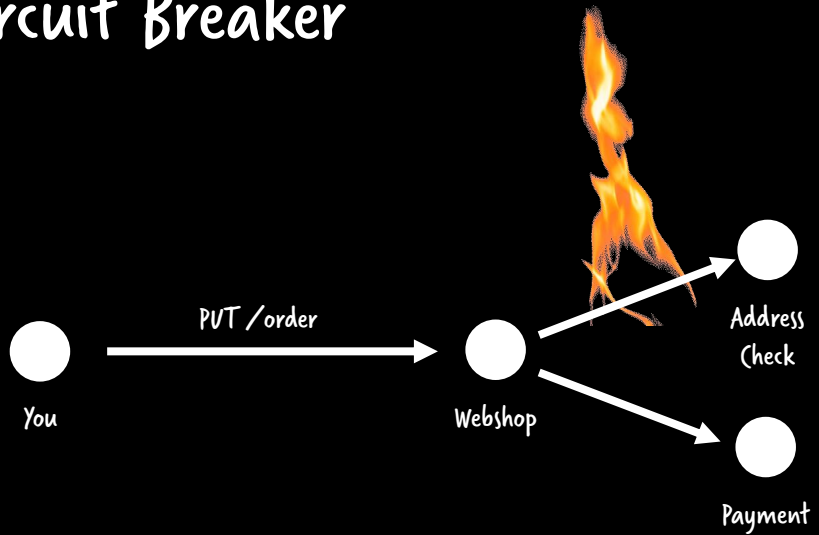
# Patterns To Survive Remote Communication

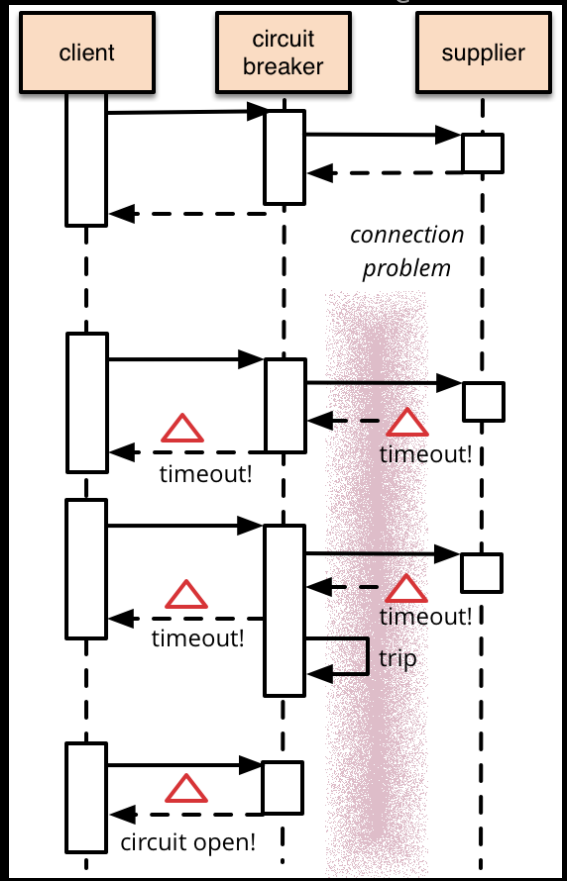
| Service Consumer | Pattern/Concept | Use With | Service Provider |
|---|---|---|---|
| X | Service Discovery | Sync | (X) |
| X | Circuit Breaker | Sync | |
| X | Bulkhead | Sync | |
| (X) | Load Balancing | Sync | X |
| X | Retry | Sync / Async | |
| X | Idempotency | Sync / Async | X |
| | De-duplication | Async | X |
| (X) | Back Pressure & Rate Limiting | Sync / (Async) | X |
| X | Await feedback | Async | |
| X | Sagas | Sync / Async | (X) |

...

# Circuit Breaker

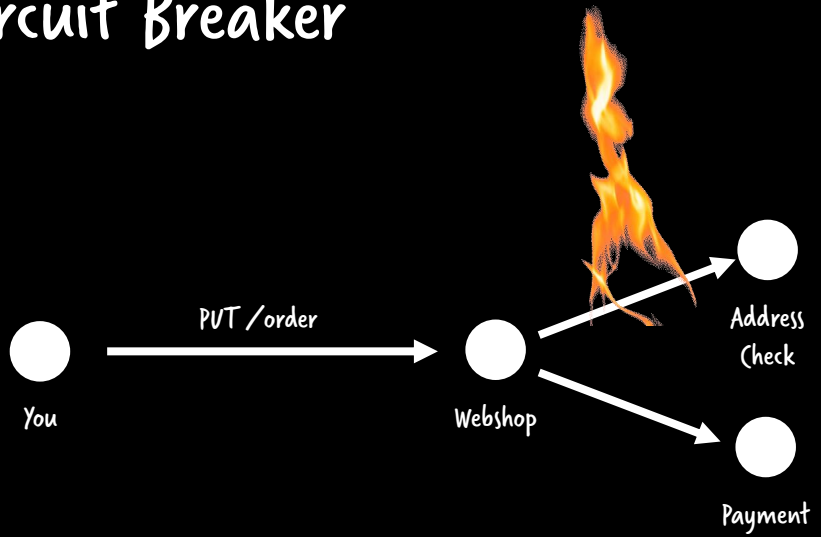
# Circuit Breaker




from https://martinfowler.com/bliki/CircuitBreaker.html

# Circuit Breaker




e.g. Resilience4J:

```
@CircuitBreaker(name = BACKEND, fallbackMethod =
"fallback")
public boolean addressValid(Address a) {
    return httpEndpoint.GET(...);
}

private boolean fallback(Address a) {
    return true;
}
```

```
resilience4j.circuitbreaker:
    instances:
        BACKEND:
            registerHealthIndicator: true
            slidingWindowSize: 100
            permittedNumberOfCallsInHalfOpenState: 3
            minimumNumberOfCalls: 20
            waitDurationInOpenState: 50s
            failureRateThreshold: 50
```

# Patterns To Survive Remote Communication

| Service Consumer | Pattern/Concept | Use With | Service Provider |
|---|---|---|---|
| X | Service Discovery | Sync | (X) |
| X | Circuit Breaker | Sync | |
| X | Bulkhead | Sync | |
| (X) | Load Balancing | Sync | X |
| X | Retry | Sync / Async | |
| X | Idempotency | Sync / Async | X |
| | De-duplication | Async | X |
| (X) | Back Pressure & Rate Limiting | Sync / (Async) | X |
| X | Await feedback | Async | |
| X | Sagas | Sync / Async | (X) |

...


Coupling

# Types of Coupling

| Type of coupling | Description | Example | Recommendation |
| --- | --- | --- | --- |
| **Implementation** Coupling | Service knows internals of other services | Joined database | |
| | | | |
| | | | |
| | | | |

# Types of Coupling

| Type of coupling | Description | Example | Recommendation |
| --- | --- | --- | --- |
| **Implementation** Coupling | Service knows internals of other services | Joined database | **Avoid** |
| | | | |
| | | | |
| | | | |

# Types of Coupling

| Type of coupling | Description | Example | Recommendation |
|---|---|---|---|
| **Implementation** Coupling | Service knows internals of other services | Joined database | **Avoid** |
| **Temporal** Coupling | Service depends on availability of other services | Synchronous blocking communication | |
| | | | |
| | | | |

# Types of Coupling

| Type of coupling | Description | Example | Recommendation |
|---|---|---|---|
| **Implementation** Coupling | Service knows internals of other services | Joined database | **Avoid** |
| **Temporal** Coupling | Service depends on availability of other services | Synchronous blocking communication | **Reduce or manage** |
| | | | |
| | | | |

# Types of Coupling

| Type of coupling | Description | Example | Recommendation |
|---|---|---|---|
| **Implementation** Coupling | Service knows internals of other services | Joined database | **Avoid** |
| **Temporal** Coupling | Service depends on availability of other services | Synchronous blocking communication | **Reduce or manage** |
| **Deployment** Coupling | Multiple services can only be deployed together | Release train | |
| | | | |

# Types of Coupling

| Type of coupling | Description | Example | Recommendation |
|---|---|---|---|
| **Implementation** Coupling | Service knows internals of other services | Joined database | **Avoid** |
| **Temporal** Coupling | Service depends on availability of other services | Synchronous blocking communication | **Reduce or manage** |
| **Deployment** Coupling | Multiple services can only be deployed together | Release train | Typically **avoid**, but depends |
| | | | |

# Types of Coupling

| Type of coupling | Description | Example | Recommendation |
|---|---|---|---|
| **Implementation** Coupling | Service knows internals of other services | Joined database | **Avoid** |
| **Temporal** Coupling | Service depends on availability of other services | Synchronous blocking communication | **Reduce or manage** |
| **Deployment** Coupling | Multiple services can only be deployed together | Release train | Typically **avoid**, but depends |
| **Domain** Coupling | Business capabilities require multiple services | Order fulfillment requires payment, inventory and shipping | |

# Types of Coupling

| Type of coupling | Description | Example | Recommendation |
|---|---|---|---|
| **Implementation** Coupling | Service knows internals of other services | Joined database | **Avoid** |
| **Temporal** Coupling | Service depends on availability of other services | Synchronous blocking communication | **Reduce or manage** |
| **Deployment** Coupling | Multiple services can only be deployed together | Release train | Typically **avoid**, but depends |
| **Domain** Coupling | Business capabilities require multiple services | Order fulfillment requires payment, inventory and shipping | **Unavoidable** unless you change business requirements or service boundaries |

| Type of coupling | Recommendation |
| --- | --- |
| **Implementation** Coupling | **Avoid** |
| **Temporal** Coupling | **Reduce or manage** |
| **Deployment** Coupling | Typically **avoid**, but depends |
| **Domain** Coupling | **Unavoidable** unless you change business requirements or service boundaries |

The **communication** style can reduce temporal coupling

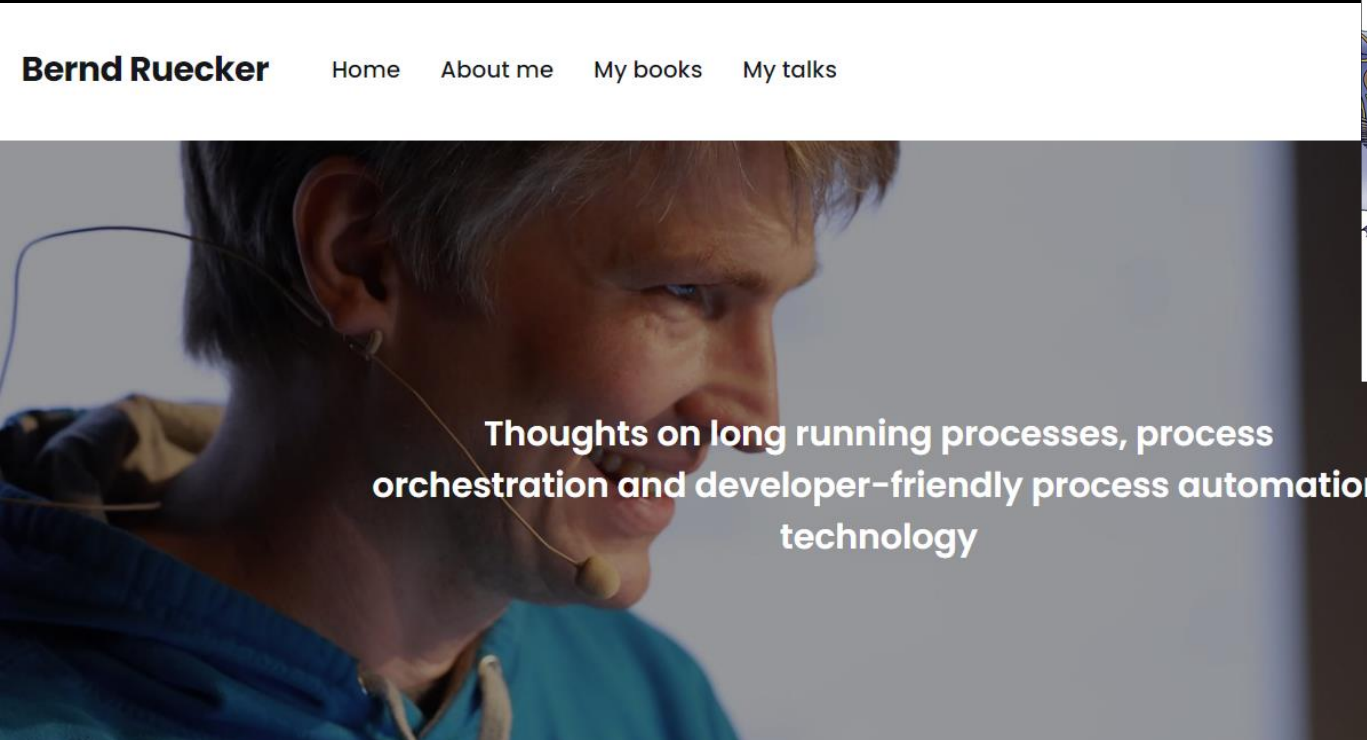
Some people say that event-driven systems decouple better. But in reality, it just turns the direction of the dependency around.
The **collaboration** style does **not** decouple!

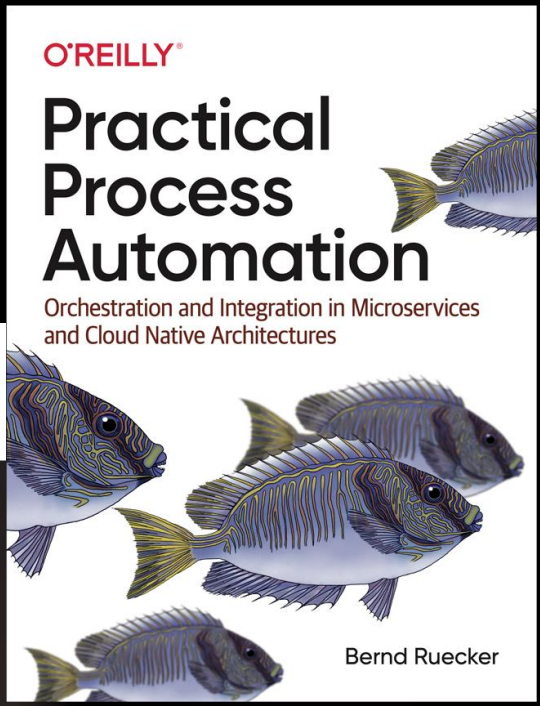
# Summary

- Know
  - communication styles (sync/async)
  - collaboration styles (command/event)
- You can get rid of temporal coupling with asynchronous communication
  - Make sure you or your team can handle it
  - You will need long running capabilities (you might need it anyway)
  - Synchronous communication + correct patterns might also be OK
- Domain coupling does not go away!
  - Design boundaries to limit coupling (high cohesion within boundaries)
- Long running capabilities help to distribute responsibilities correctly

# Want To Know More?

https://berndruecker.io/ | https://camunda.com/




Thank you!
@uwendruecker

Contact: bernd.ruecker@camunda.com
@berndruecker

Slides: https://berndruecker.io

Blog: https://blog.bernd-ruecker.com/

Code: https://github.com/berndruecker