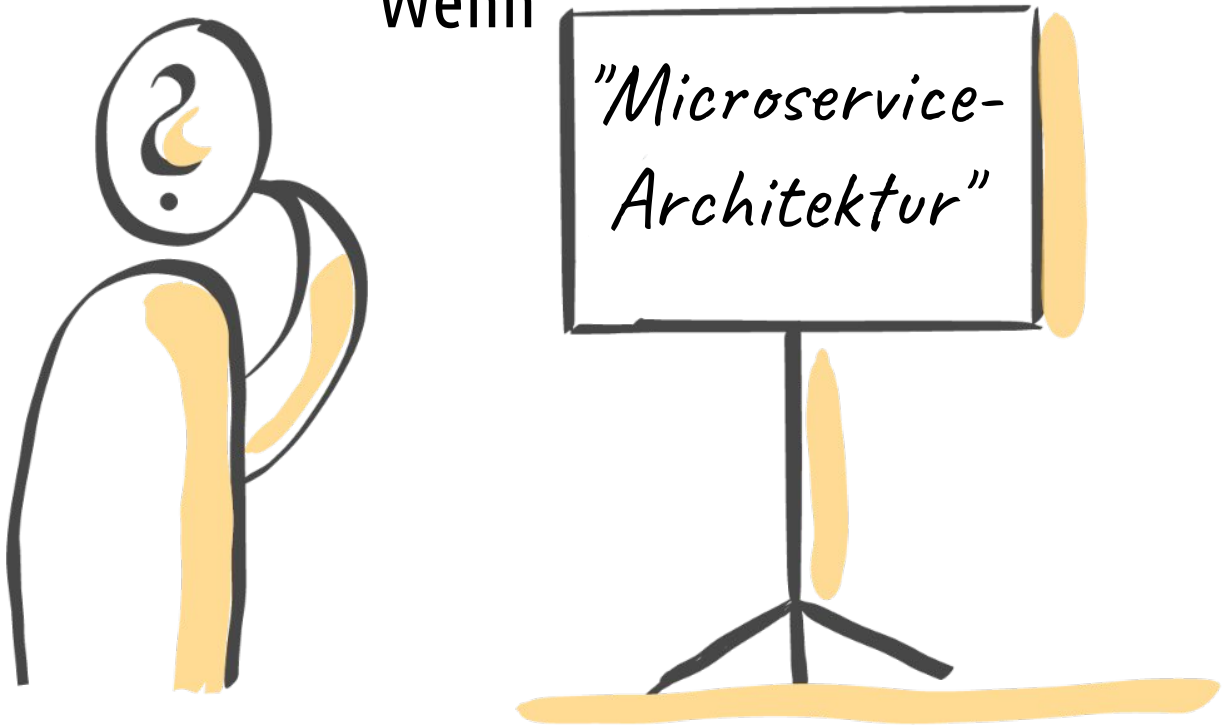
Wenn



die Antwort ist,
was war dann eigentlich die Frage?

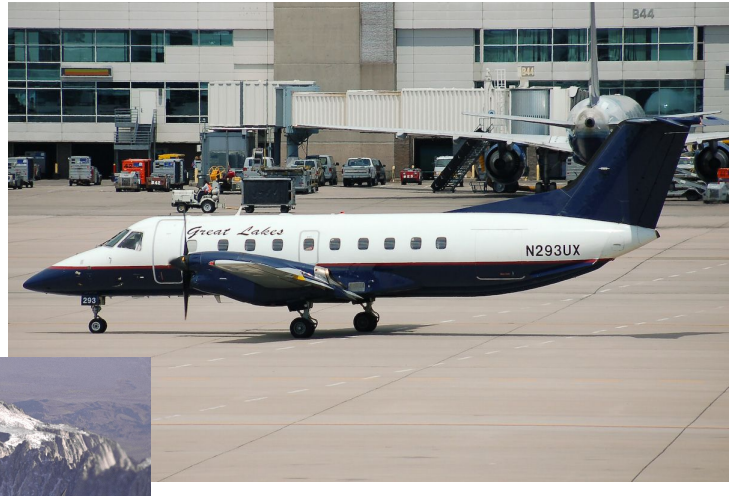
peter.fichtner@fiduciagad.de

@petfic

tilmann@tgnowledgy.me

@TGnowledgy

Stell dir vor, wir sollen ein Flugzeug bauen...



Was wir damit sagen wollen...

Es gibt außergewöhnliche Projekte, für deren Erfolg außergewöhnliche, teils verrückte Maßnahmen unabdingbar sind! Wahrscheinlich macht deren Anzahl 1% aller Projekte aus.

Aber dieser Vortrag geht nicht um diese außergewöhnlichen Projekte und ihre Verrücktheiten, sondern um fehlende Vernunft in den 99% der “gewöhnlichen” Projekte

Außergewöhnliches Projekt?

- Pro Minute sollen ca. 4 Millionen Suchanfragen beantwortet werden
- Zu durchsuchen sind alle Dokumente des Internets
  (ca. 33.000 EB, also 33 Millionen TB bzw. 33.000 Milliarden GB)
- Jede Suchanfrage muss in unter einer Sekunde
  beantwortet werden
- 15% aller Suchanfragen sind neu
  (das sind pro Tag 525 Millionen)
- Ca. alle zwei Jahre verdoppelt sich
  der zu durchsuchende Datenbestand

AWASON - ein einfaches Shopsystem

Aller**W**elts**A**nwendungs**S**oftware für **O**nline **N**utzer

Elevator Pitch

> *“AWASON ist eine historisch gewachsene, monolithische Anwendung, geschrieben von Entwicklern, die keine Lust auf automatisierte Tests hatten und mittlerweile betreut v.a. von Entwicklern, die zu träge sind, sich ein anderes Projekt zu suchen oder solchen, die sich erstmal ihre Sporen verdienen müssen.”*

Kennzahlen

- ~ 2Mio Umsatz pro Monat
- ~ 3.000 Bestellungen pro Monat
- ~ 20.000 registrierte Kunden
- ~12 Entwickler

Unser Problem: AWASON ist nicht zukunftssicher

Um am Markt erfolgreich zu sein, brauchen wir ein System, das schnell auf geänderte Anforderungen angepasst werden kann

Das kann AWASON nicht leisten!!!11!elf!

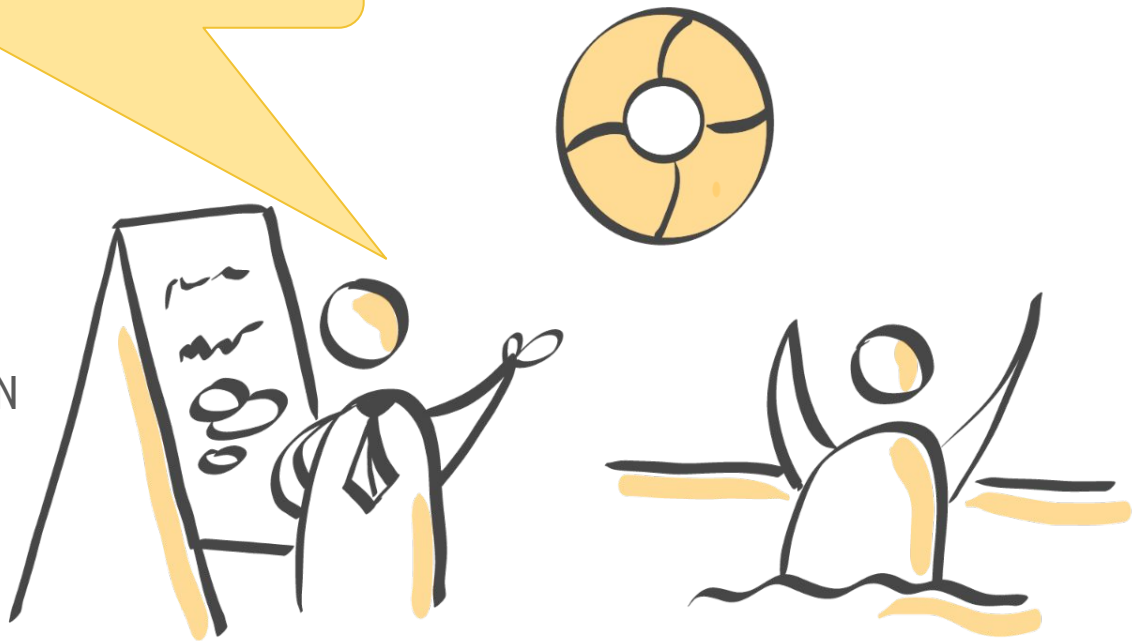
Denkt doch an die Quereffekte, manuellen Testphasen, Wissensfluktuation, hässliche UI, ...

Managemententscheidung for the rescue!

AWASON wird durch eine Neuentwicklung abgelöst

Dazu fordert das Management

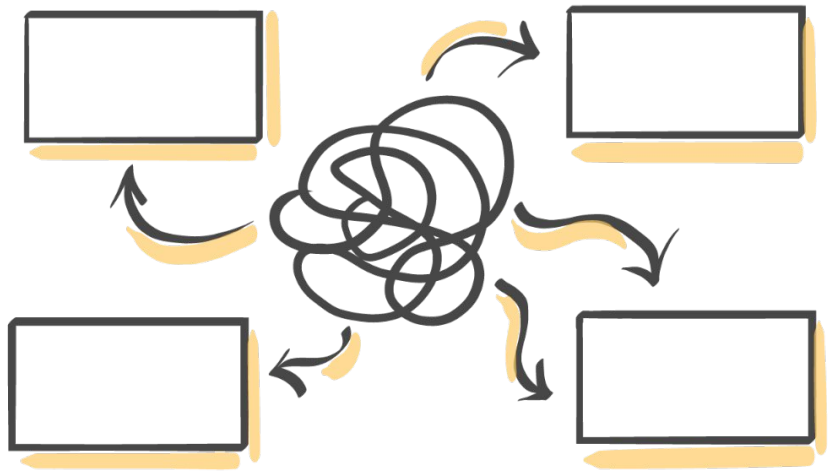
- Root-Cause Analyse zu AWASON
- Blick über den Tellerrand



Root-Cause Analyse

Recherche zu “Typische Probleme Monolithischer Architekturen”

- Verständlichkeit für Entwickler: schwer
- Kopplung der Komponenten: eng
- Wiederverwendbarkeit von Code: gering
- Deploymenthäufigkeit: gering
- Änderungen: aufwändig und kostenintensiv
- Stabilität: gering
- Skalierbarkeit: schlecht



Root-Cause Analyse

Ergebnis

“Alle Nachteile, die wir über Monolithen recherchiert haben, finden wir in AWASON.”

Schlussfolgerung

Es muss am Monolithen liegen

Blick über den Tellerrand - Wie machen es andere?

Recherche zu “Vorteile von Microservices”

- Verständlichkeit für Entwickler: hoch
- Kopplung: gering
- Wiederverwendbarkeit: hoch
- Kosten: gering
- Änderungen: schnell und kostengünstig
- Stabilität: hoch
- Betrieb: einfach

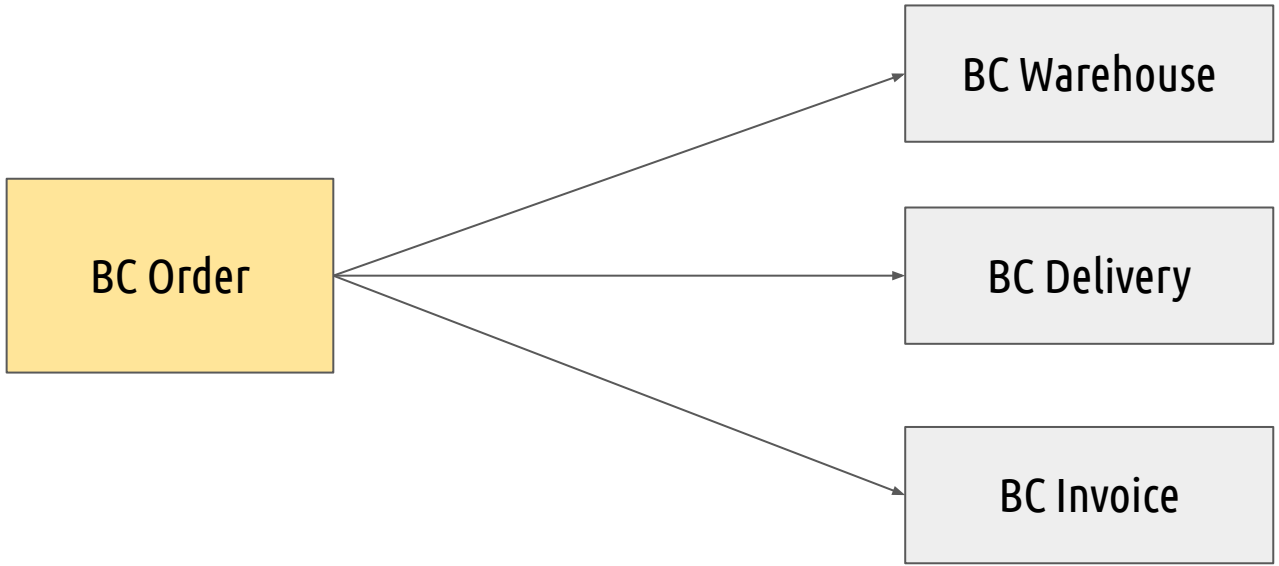
Die Lösung liegt nahe

Wenn wir unsere Anwendung als Microservice-Architektur bauen, erschlagen wir all unsere Probleme

Wir eröffnen uns sogar neue Möglichkeiten

Wir haben dann ein flexibles System
und werden so cool wie Spotify, Netflix und Twitter

Oh man, jetzt reden die von Bounded Contexts. Was ist das denn?

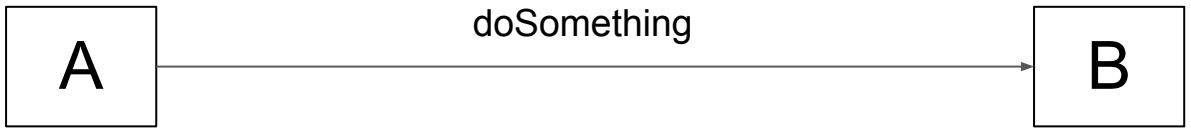


Codebeispiel aus bisherigem AWASON Order Handling

```
@Inject Warehouse warehouse;

@Inject Delivery delivery;

@Inject Invoice invoice;

void handleNewOrder(Order order) {

      try {

            myOwnLogic();

            warehouse.decreaseStock(toWarehouseDO(order.items()));

            delivery.shipOrder(true, toDeliveryDO(order.items()));

            invoice.createInvoice("CREDIT", toInvoiceDO(order.items()), dateFormatX(now()));

      } catch (Exception e) { /** uuuh! How to and when to inform/rollback
warehouse/delivery/invoice? Do what if one of them fails during rollback? **/ }

}
```

Wir wollen Änderungen jederzeit ausbringen können!

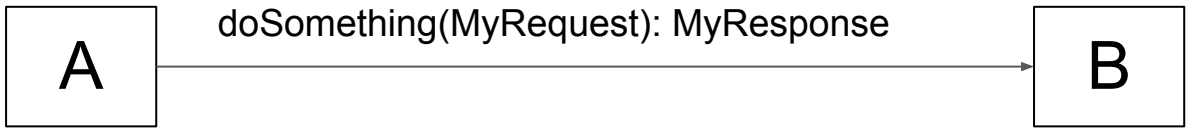
Was ist eines der grundlegenden Probleme warum uns das nicht gelang?



Problem Methodenname?

Wir wollen Änderungen jederzeit ausbringen können!

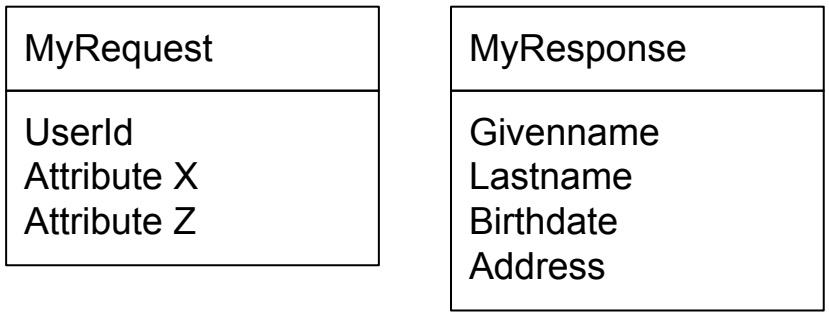
Was ist eines der grundlegenden Probleme warum uns das nicht gelang?



Problem Klassennamen Request/Response?

Wir wollen Änderungen jederzeit ausbringen können!

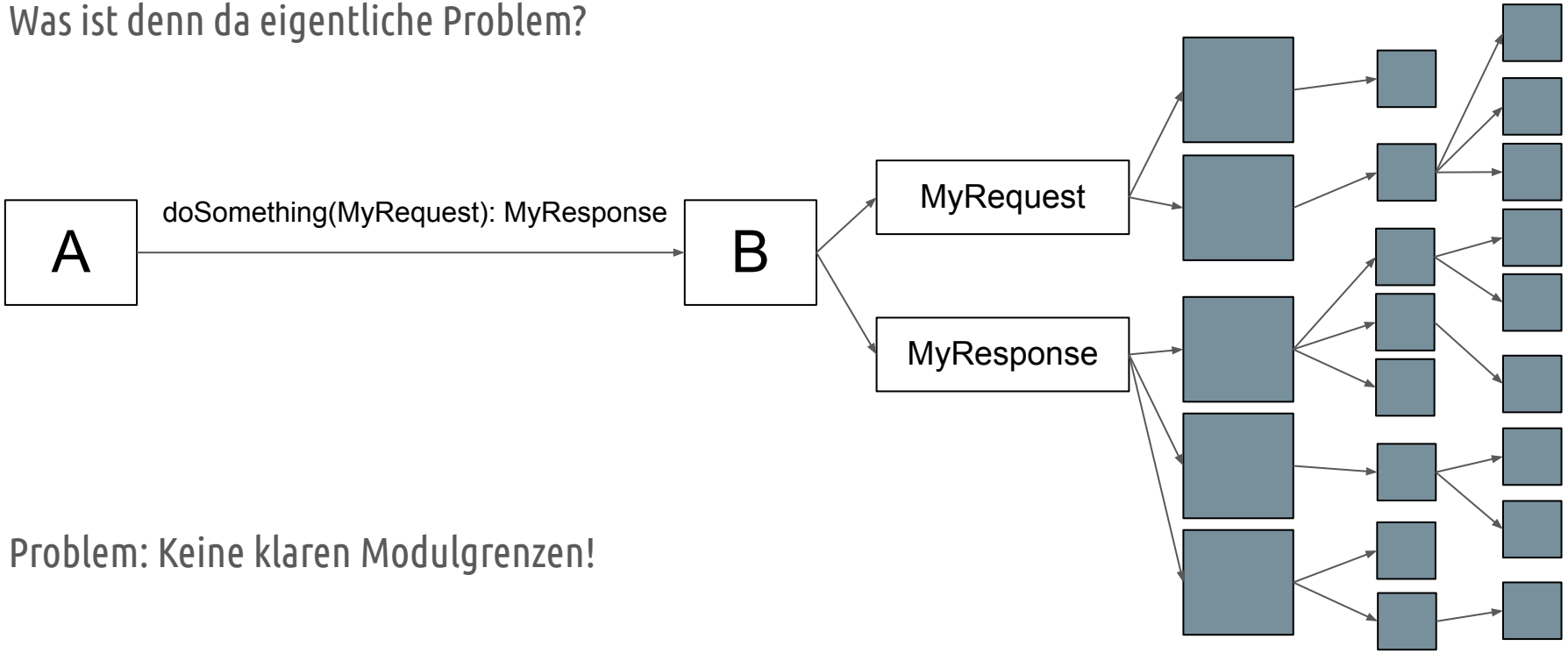
Was ist eines der grundlegenden Probleme warum uns das nicht gelang?



Problem Attribute Request/Response?

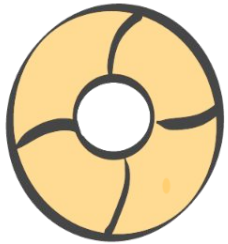
Wir wollen Änderungen jederzeit ausbringen können!

Was ist denn da eigentliche Problem?



Problem: Keine klaren Modulgrenzen!

REST (JSON) for the rescue?!



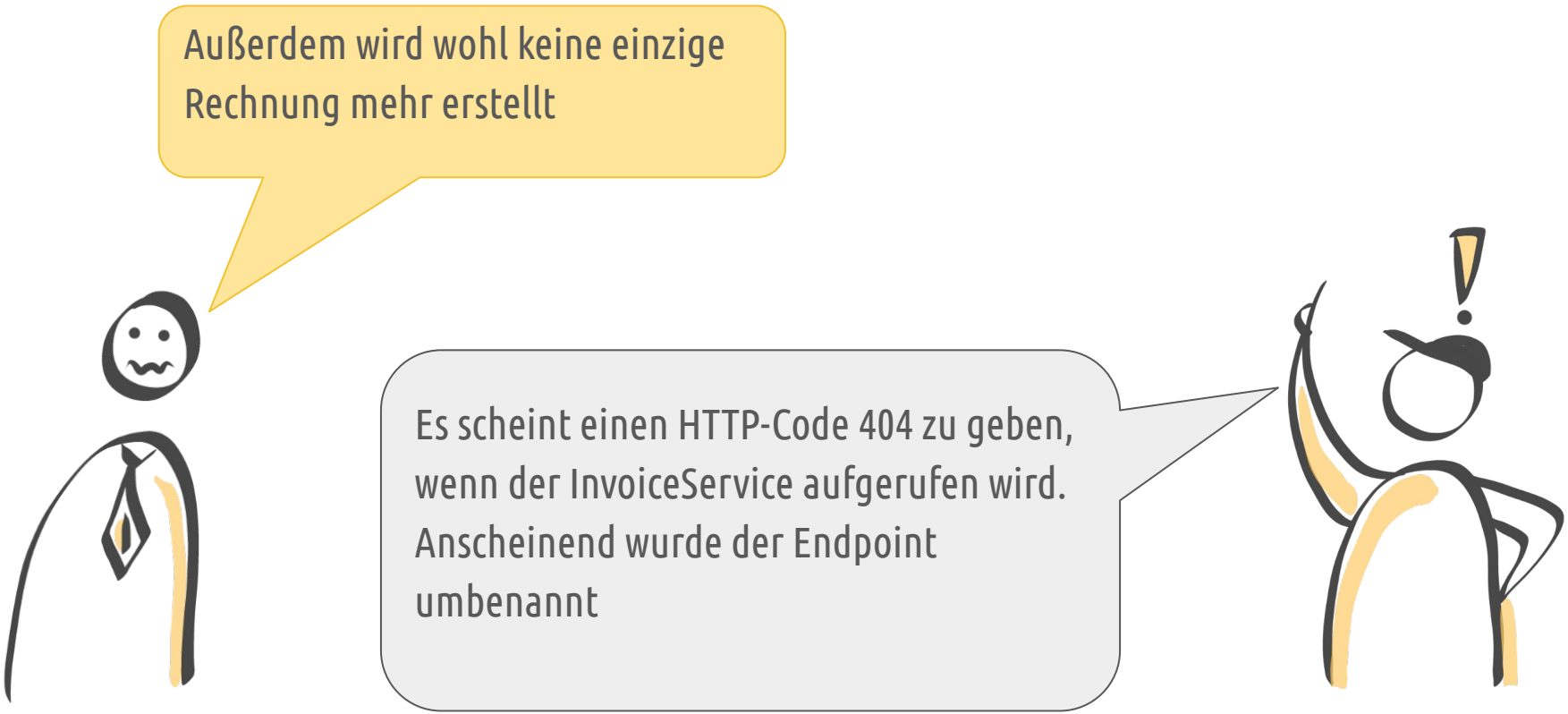
```
@Inject RestClient warehouse;

@Inject RestClient delivery;

@Inject RestClient invoice;

void handleNewOrder(Order order) {

      try {

            myOwnLogic();

            warehouse.doPut("/decreaseStock", toWarehouseDO(order.items()));

            delivery.doPut("/shipOrder", true, toDeliveryDO(order.items()));

            invoice.doPost("/createInvoice", "CREDIT", toInvoiceDO(order.items()), dateFormatX(now()));

      } catch (Exception e) { /** uuuh! How to and when to inform/ rollback
warehouse/delivery/invoice? Do what if one of them fails during  rollback? **/ }

}
```

Ein paar Updates nach dem Go-Live...

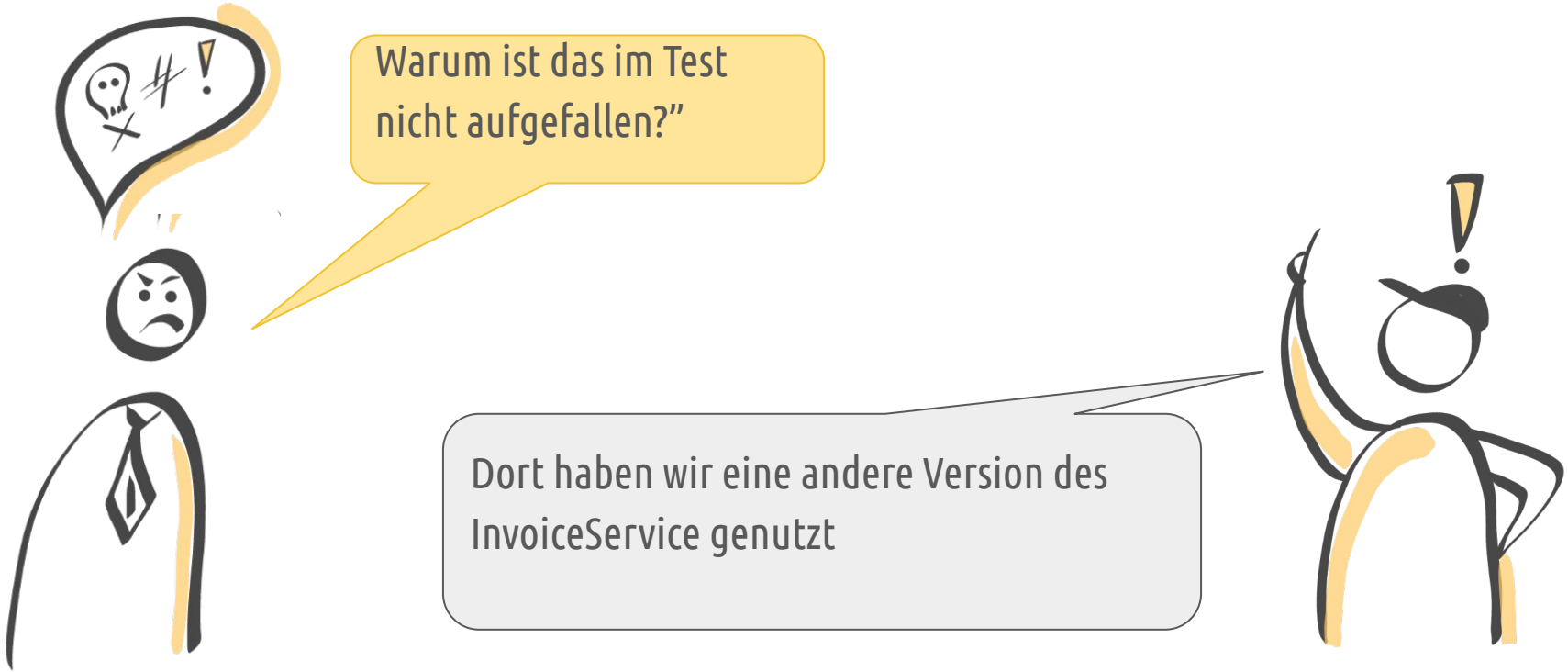
Wir haben ein massives Problem in Produktion. Im DeliveryService kommen einige Bestellungen nicht mehr an

In den Logfiles sieht es aus, als könnten Attribute vom Typ ***multiItem*** des JSON nicht gemapped werden. Das wurde wohl in ***bundle*** umbenannt

Ein paar Updates nach dem Go-Live...



Ein paar Updates nach dem Go-Live...



Oh Shit...

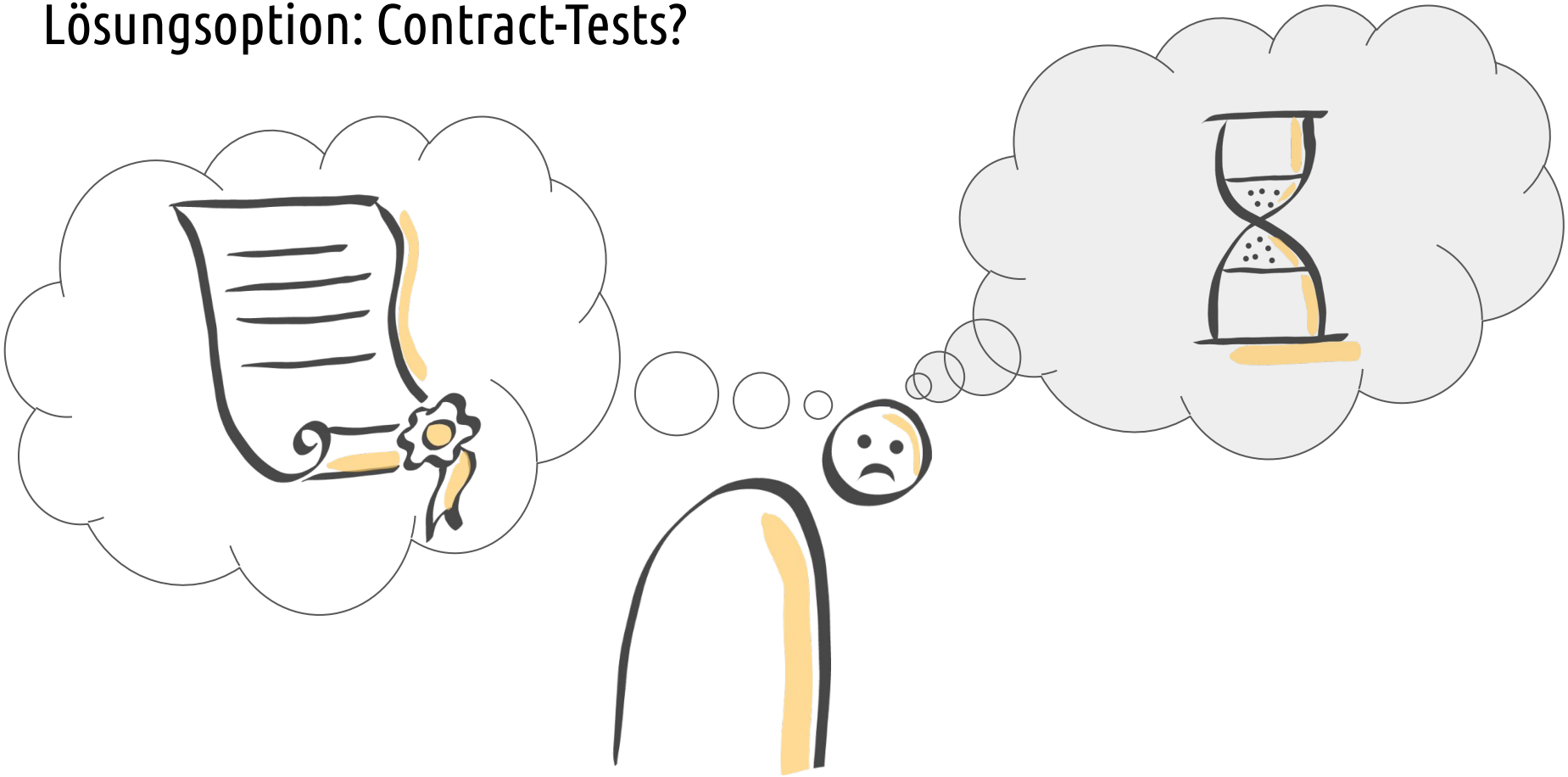
Microservices wiegen uns zur Entwicklungszeit in der Illusion, technisch entkoppelt zu sein.

Erst viel später, zur Laufzeit merken wir dann, dass das nicht stimmt.

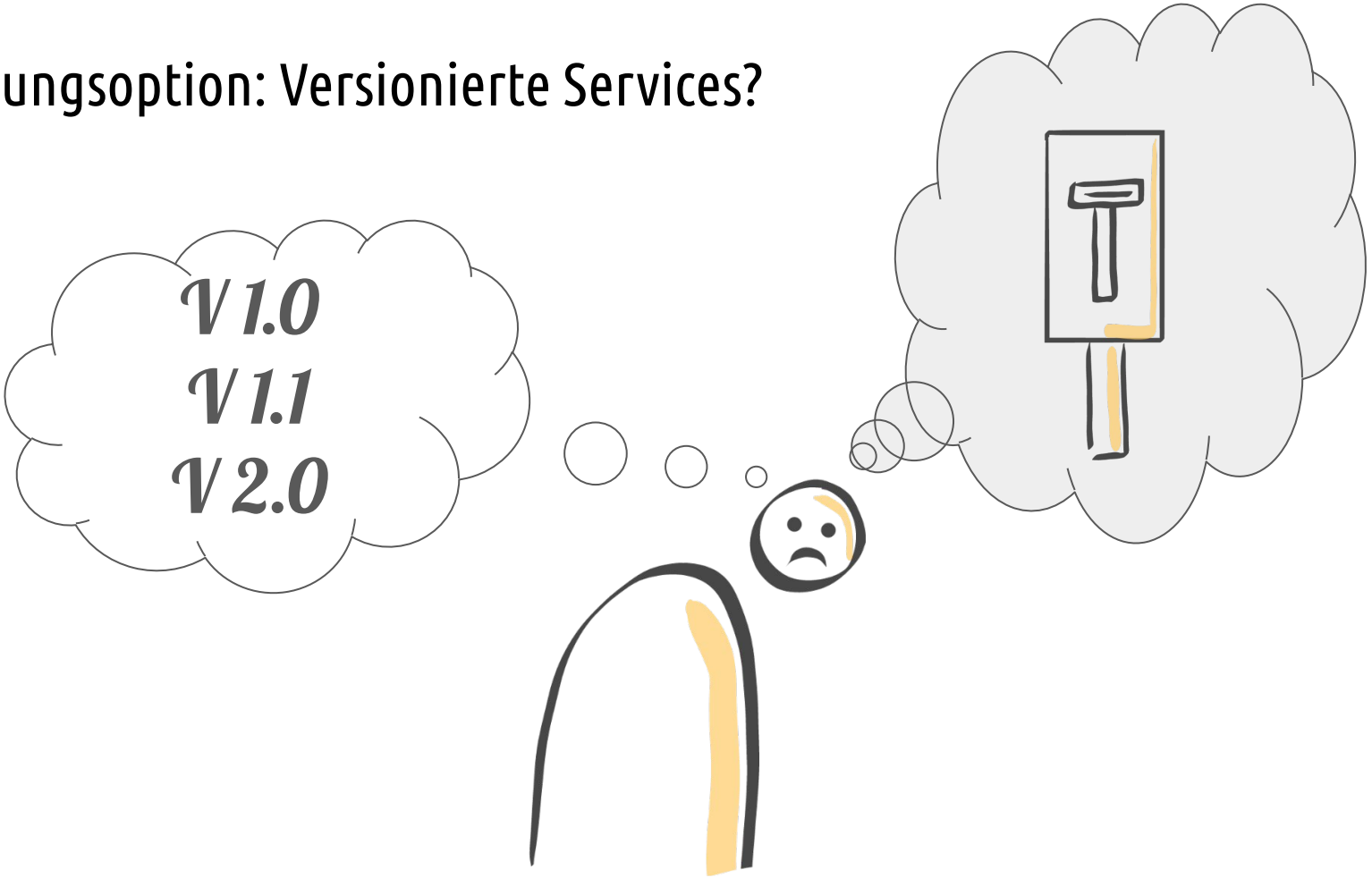
Schlimmer noch: Wir werden nun nicht mehr rechtzeitig vom Compiler gewarnt

Lösungsoption: Release-Koordinator?

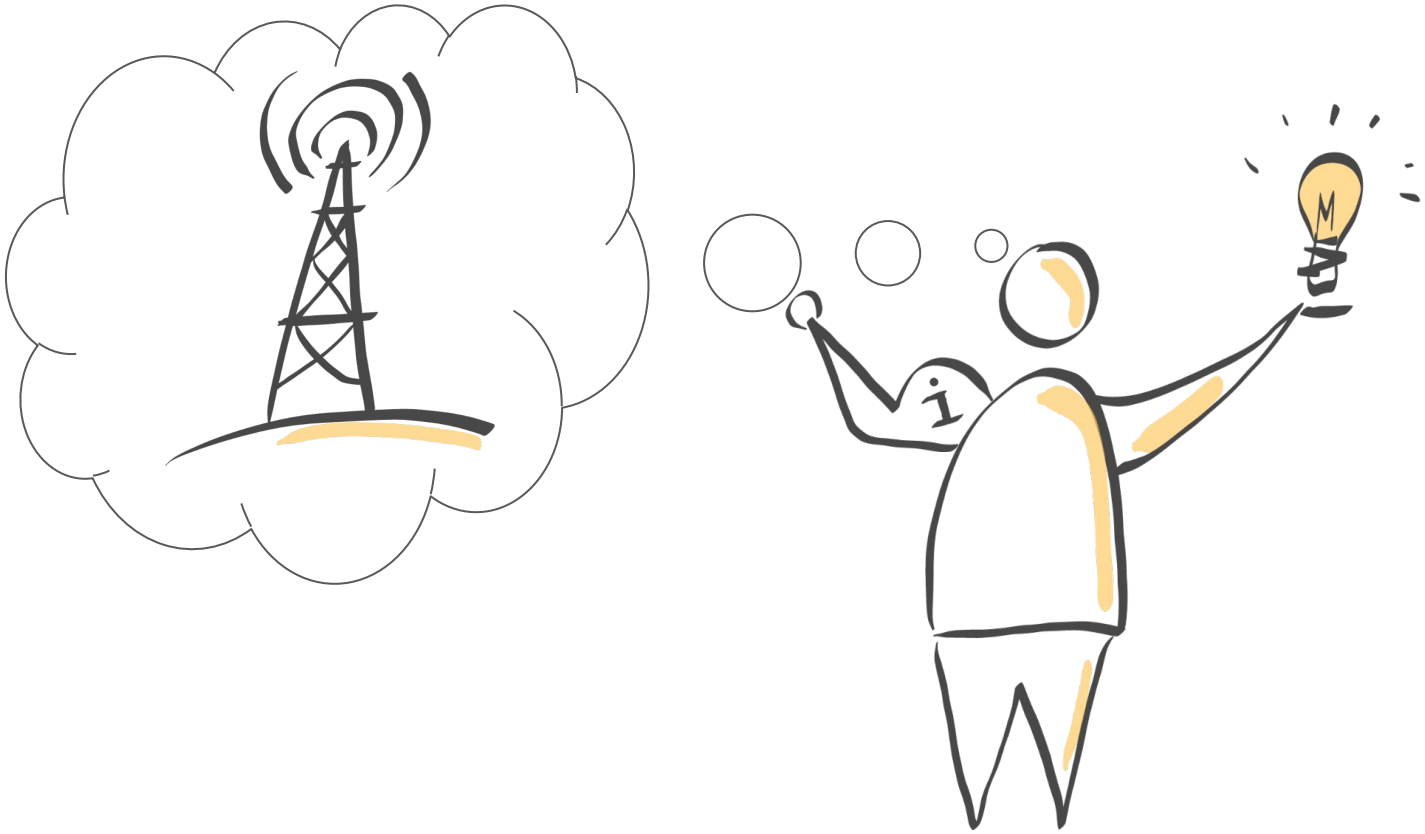
Lösungsoption: Contract-Tests?



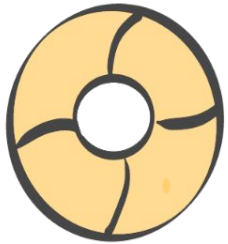
Lösungsoption: Versionierte Services?



Lösungsoption: Message-Orientation!

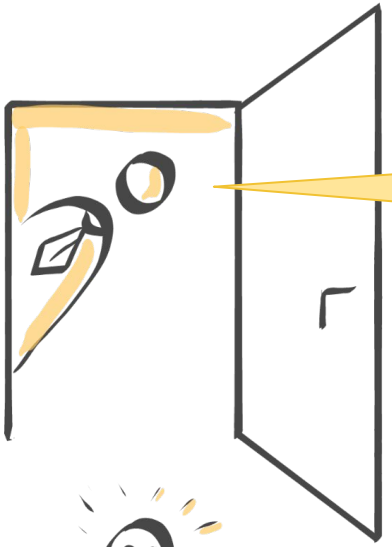


Okay, dann eben Kafka for the rescue!



```
@Inject KafkaTemplate<String, String> kafkaTpl;

void handleNewOrder(Order order) {

     try {

          myOwnLogic();

          kafkaTpl.send("decreaseStock", toWarehouseDO(order.items()));

          kafkaTpl.send("shipOrder", true, toDeliveryDO(order.items()));

          kafkaTpl.send("createInvoice", "CREDIT", toInvoiceDO(order.items()), dateFormatX(now()));

     } catch (Exception e) { /** Rollback problem solved? **/ }

}
```

Kurz nach der Umstellung...



Kein Problem. Wir erweitern das createInvoice Event so, dass es einen Rabatt und eine Begründung dafür enthalten kann, und passen die Rechnung entsprechend an.

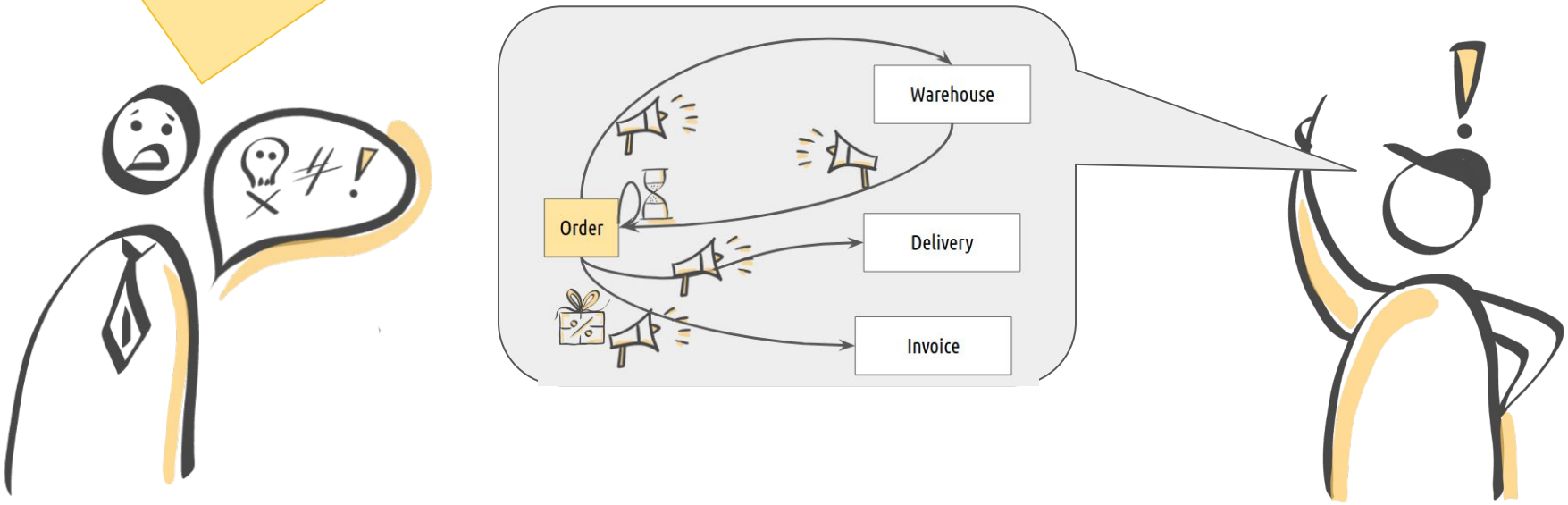
Das klingt voll einfach. Microservices sind toll!

Zwei Wochen später, nach Ausbringung des neuen INVOICE Service

Hey Team INVOICE, das neue Feature funktioniert nicht

Zwei Wochen später, nach Ausbringung des neuen INVOICE Service

Heißt das, es hätten noch andere etwas tun müssen???



Oh Shit...

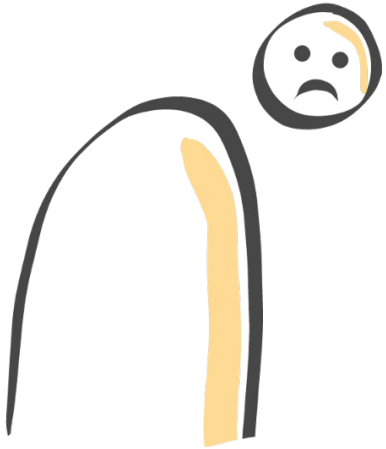
Über einen Message-Bus kommunizierende Microservices sind technisch zwar entkoppelt

Aber die Fachlichkeit ist trotzdem quer über die Services verteilt

Es müssen also viele Services angepasst werden, um *ein Feature* bereitzustellen

Schlimmer noch: Durch die technische Entkopplung sieht sich niemand mehr in der Verantwortung, dieses Feature in der Gesamtheit zu konzipieren. Entscheidungen liegen “zwischen den Teams”

Die bittere Erkenntnis



Wir haben fast genau die gleichen Probleme wie zuvor.

- Technische Leakages
- Falsch verteiltes Domänenwissen

Sie werden teilweise nur durch noch komplexere Probleme überdeckt

Nicht der Monolith des Altsystems war das Problem, sondern wie wir ihn gebaut haben

Microservices schützen nicht vor schlecht konzipierter Software

Microservices heilen nicht auf magische Weise unsere Unfähigkeit, sauber zu modularisieren

Die unrühmliche Rolle der Entwickler

Lieben den Lösungsraum

- Versuchen Probleme mit Technologie (Tools, Frameworks) zu erschlagen
- Verletzen dabei i.d.R. das KISS Prinzip und vergrößern die Probleme
- Nutzen Tools ohne die Konzepte zu verstehen

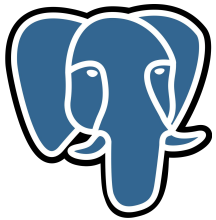
Meiden den fachlichen Problemraum

- Kundenwerte werden nur peripher beachtet
- Fachlich tiefes Verständnis der Probleme fehlt. Dadurch keine gute Lösungsfindung möglich

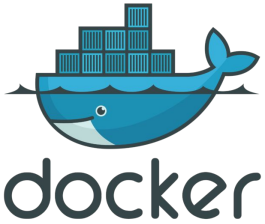
Ergebnis

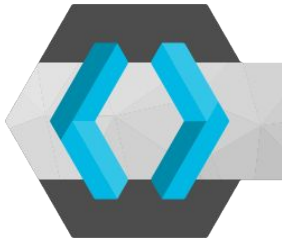
- Mit Microservices werden wir uns noch um viele weitere Probleme kümmern müssen, die wir im Monolithen gar nicht haben.

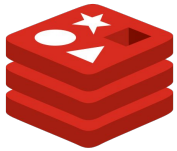
Techstack AWASON

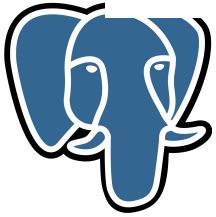


Techstack neu geschriebene Anwendung AWASON 2.0









Woher kommt der Irrglaube dass Microservices helfen?

Vermutung

- Success-Stories zu Microservices werden stärker promoted als Fehlschläge und sind daher überrepräsentiert.
- Korrelation von Problemen und Einsatz von Monolithen wird fälschlicherweise als Kausalität gesehen
- Es wird oft Monolith mit “Big Ball of Mud” gleichgesetzt
- Es wird “Microservice” mit “Idealem Domänenschnitt in Containern” gleichgesetzt

Sind Microservices also immer schlecht?

- Nein, ganz und gar nicht!
- Zur Erinnerung
  - Wir sind weder gegen Microservices noch pro Monolith
  - Wir sind gegen den Irrglaube, es gäbe Silver-Bullets und gegen unsinnige Grundsatzentscheidungen bei Softwareprojekten
  - Wir sind für Vernunft, für Inspect and Adapt und für technisch / fachlich sauberen Modulschnitt
- SR-71, Netflix, Twitter
  - Standen bedarfsgetrieben vor der Herausforderung, außergewöhnliche Komplexität und Risiken einzugehen
- Instagram, GMail
  - Sind sehr erfolgreich mit monolithischen Architekturen
- Architektur entscheidet nicht über den Hippness-Faktor für den Anwender, aber durchaus über die Wahrscheinlichkeit des Scheiterns

Die eigentlichen Fragen

- Welches fachliche Problem soll gelöst werden?
- Welche Architekturziele müssen dafür erreicht werden?
- Welchen Kundenwert adressiert das jeweilige Ziel?
- Was ist der minimalste Weg, um diese Ziele zufriedenstellend zu erreichen? (KISS)

→ Ist die Antwort dann wirklich “Microservice Architektur”?

Ein Lösungsansatz - Jetzt nochmal wirklich richtig

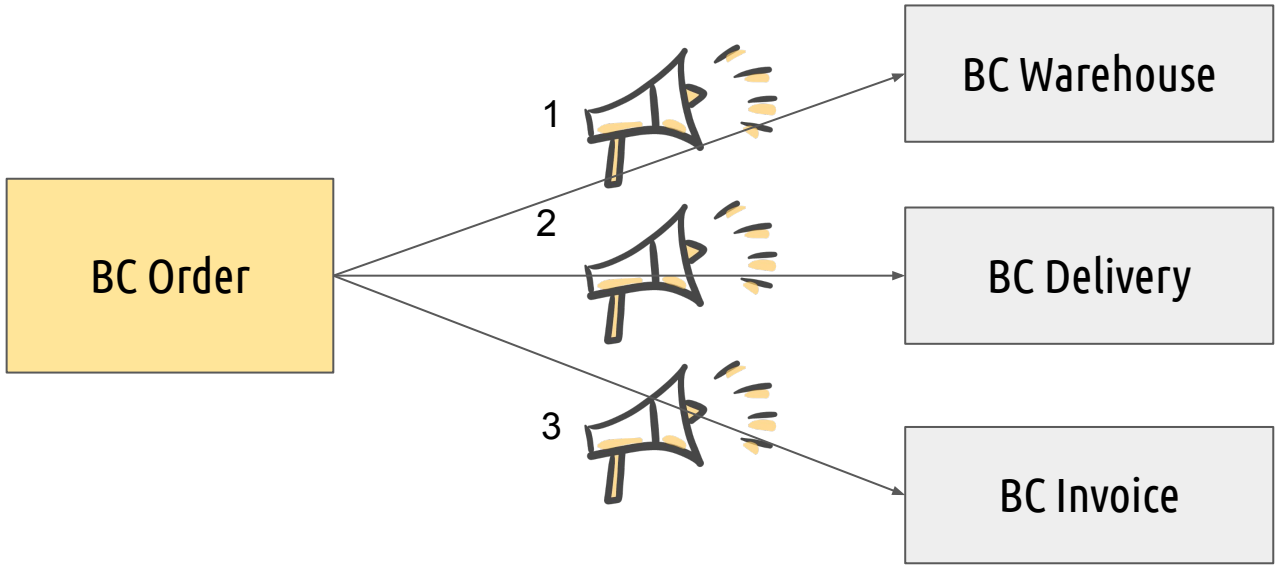
Ursprungsprobleme

- Schlecht wartbarer Legacy-Code
- enge Kopplung
- Änderungen/Erweiterungen ziehen sich quer durch das System
- Context Mapping nicht beachtet (war fehlerhaft)

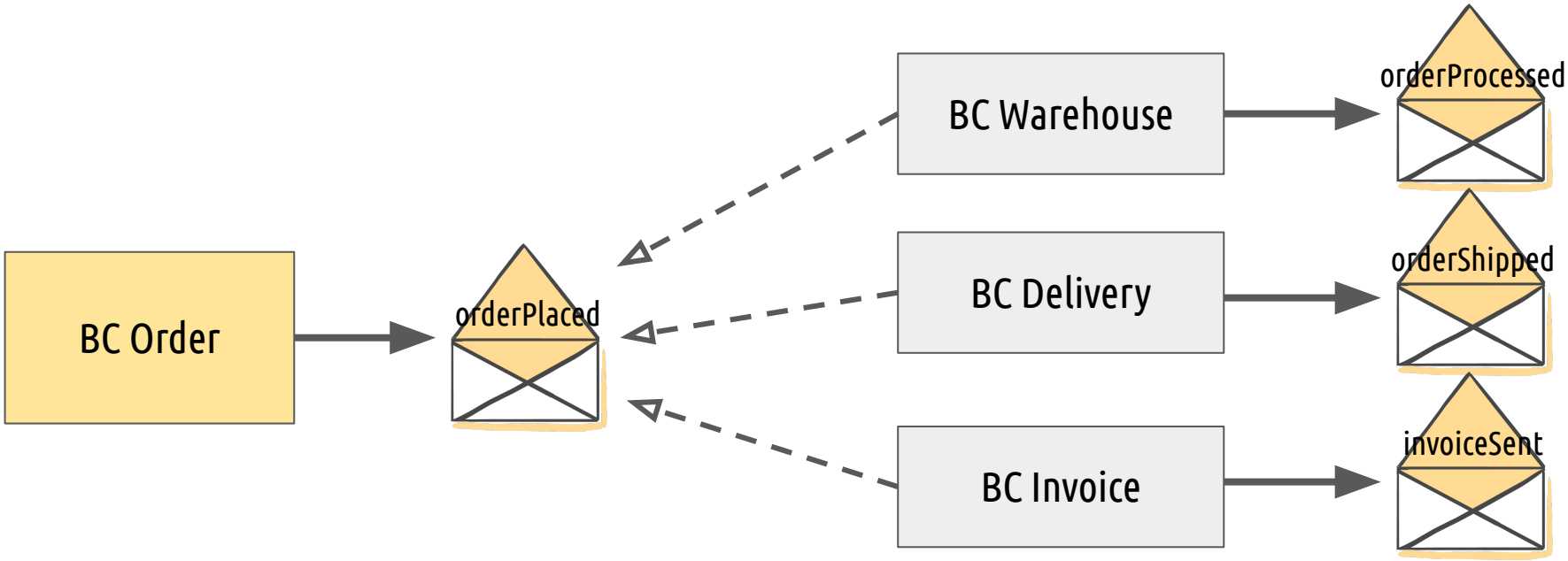
Wie können wir diese Lösen?

- Keine Aufträge an andere BCs schicken (Method-Call/RPC), sondern Events publizieren
- Gemeinsam entscheiden, wer was kommuniziert
- Darauf achten, dass jedes Modul einen eigenständigen fachlichen Wert hat

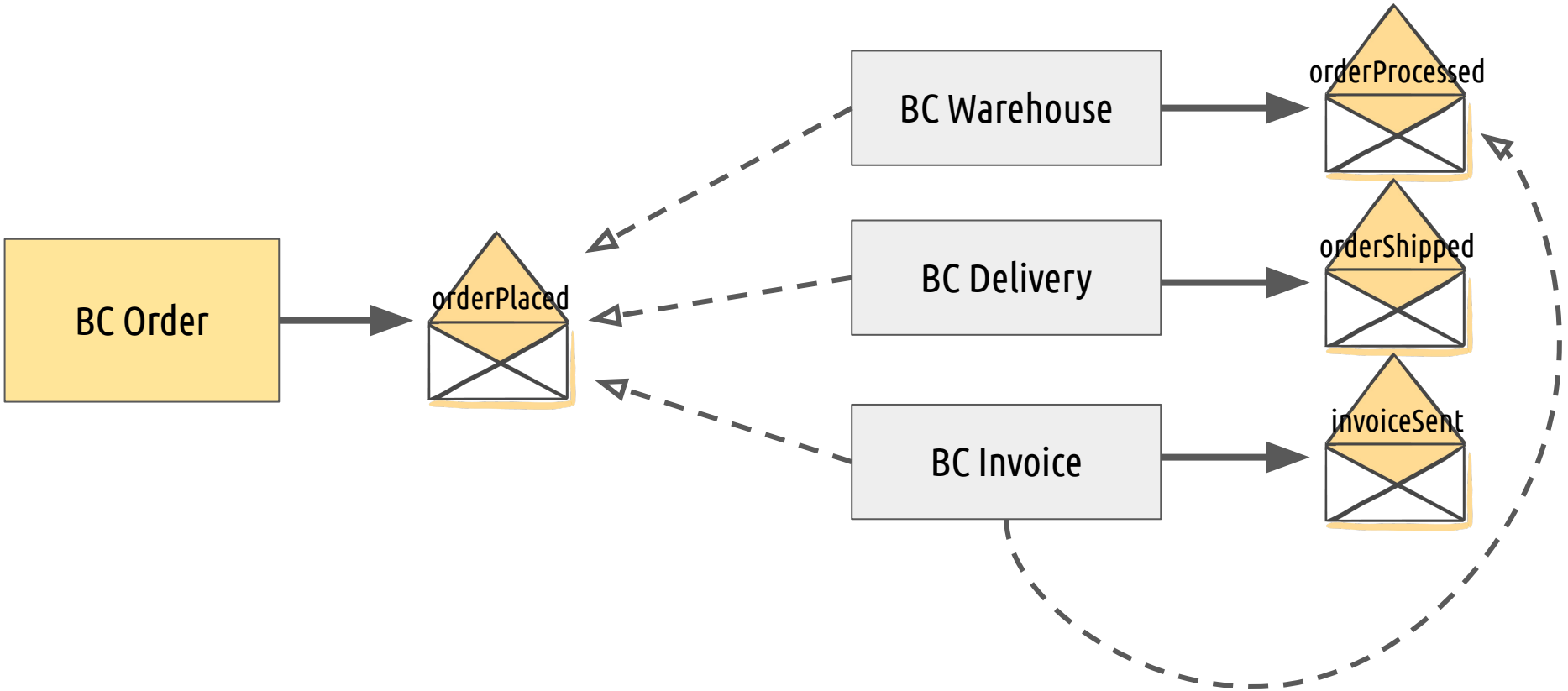
Schlechtes Konzept: “Kommandos”



Gutes Konzept: “Mitteilungen”



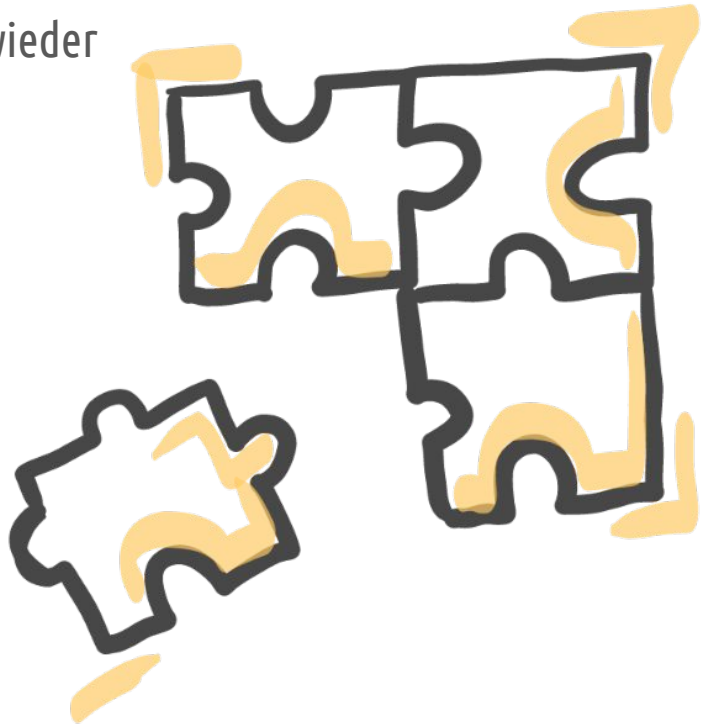
Gutes Konzept: Erweiterung wirklich atomar



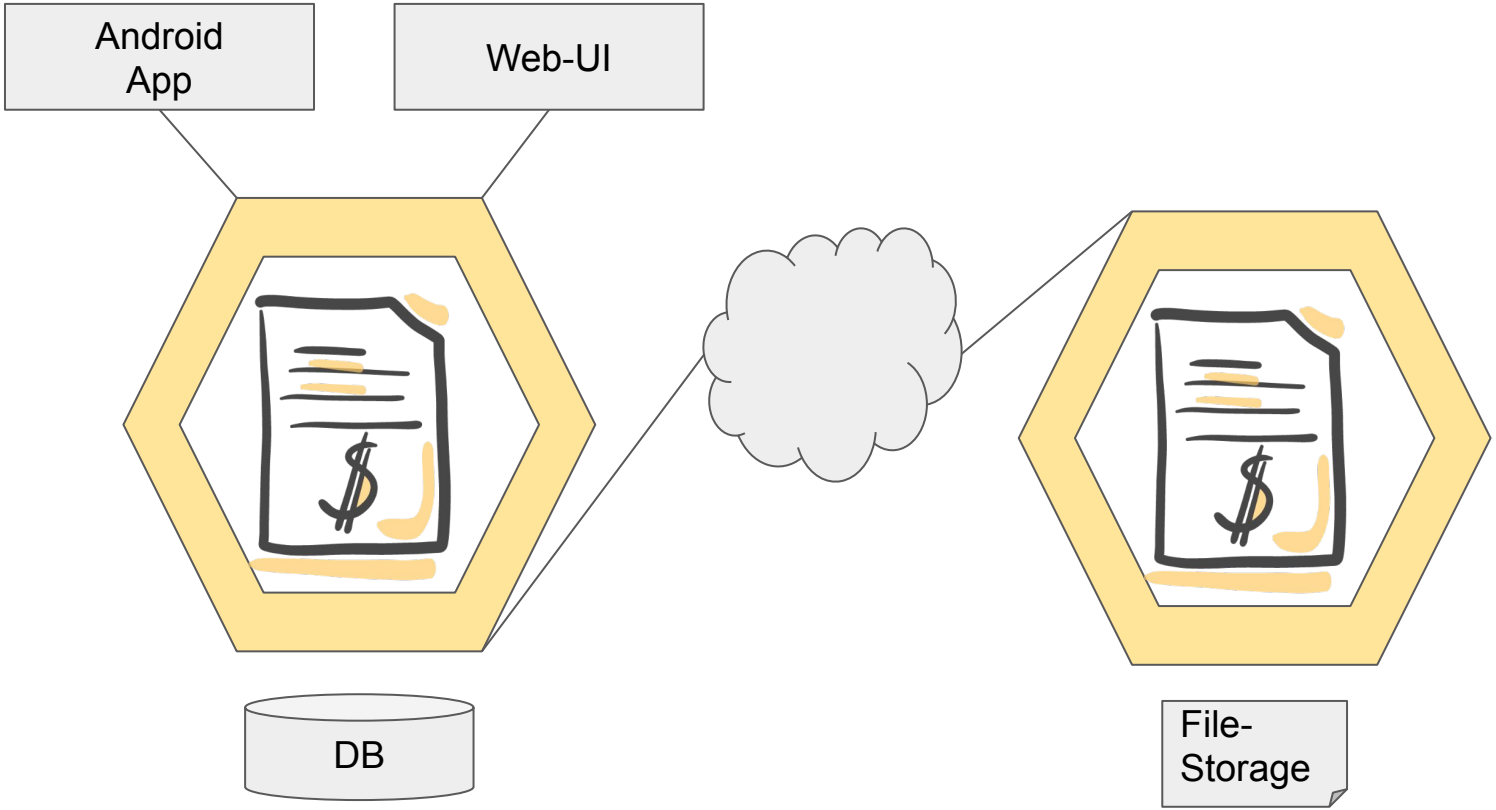
Ergebnis bei sauberem Modulschnitt

Frage der System-Architektur (MSA, Monolith, Modulith,...) wird nachrangig

Bedarfsgetrieben kann System-Architektur immer wieder verändert werden

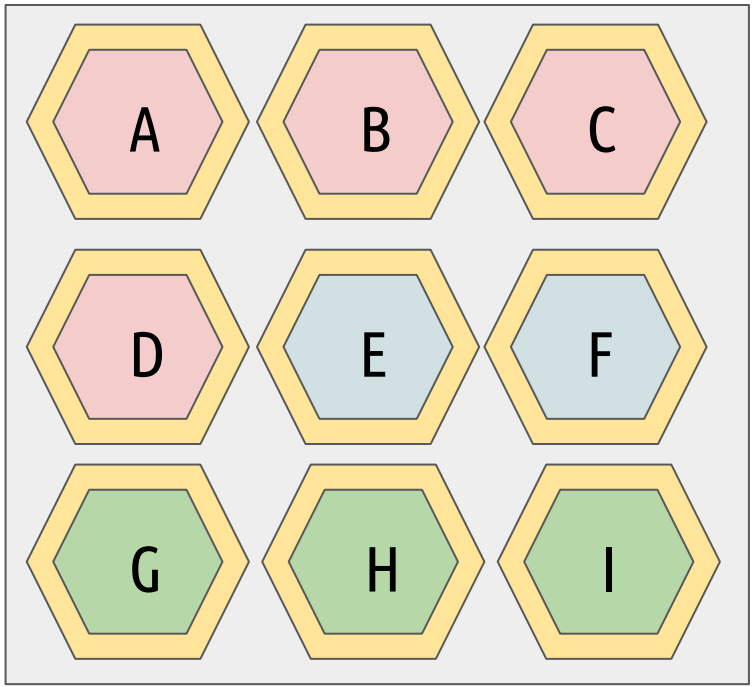


Gutes Konzept: Ports and Adapters innerhalb jedes Moduls



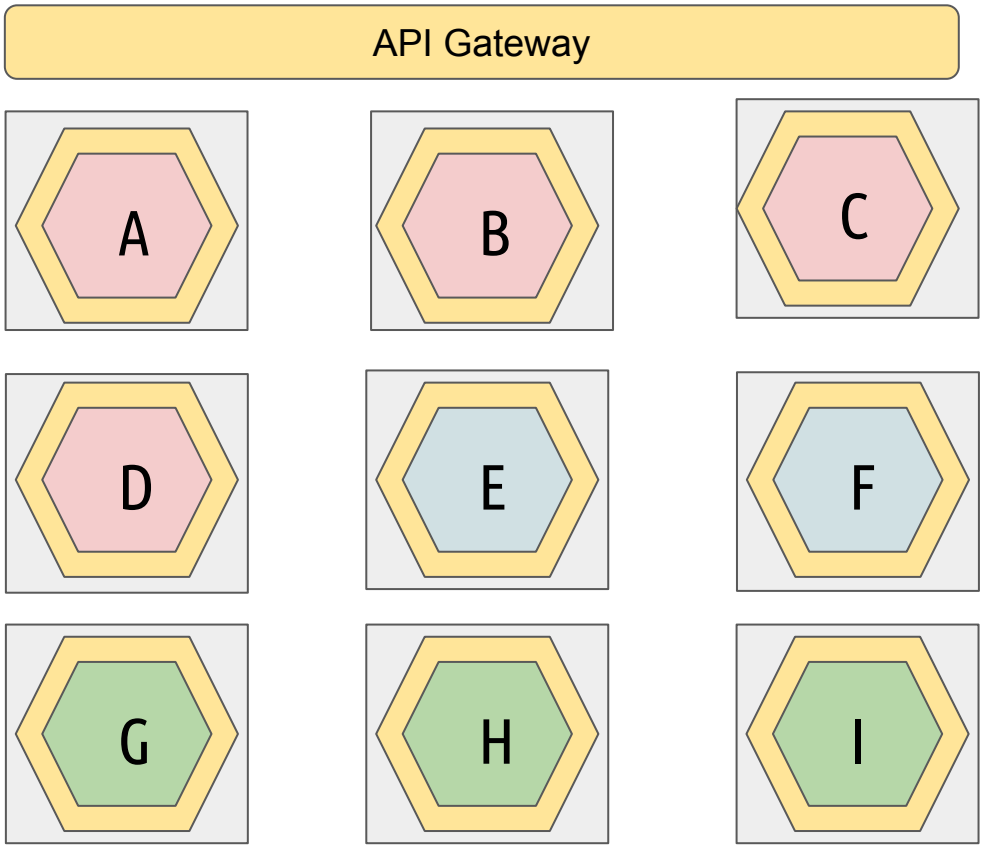
So könnte die Zielarchitektur aussehen

Modulith/Monolith



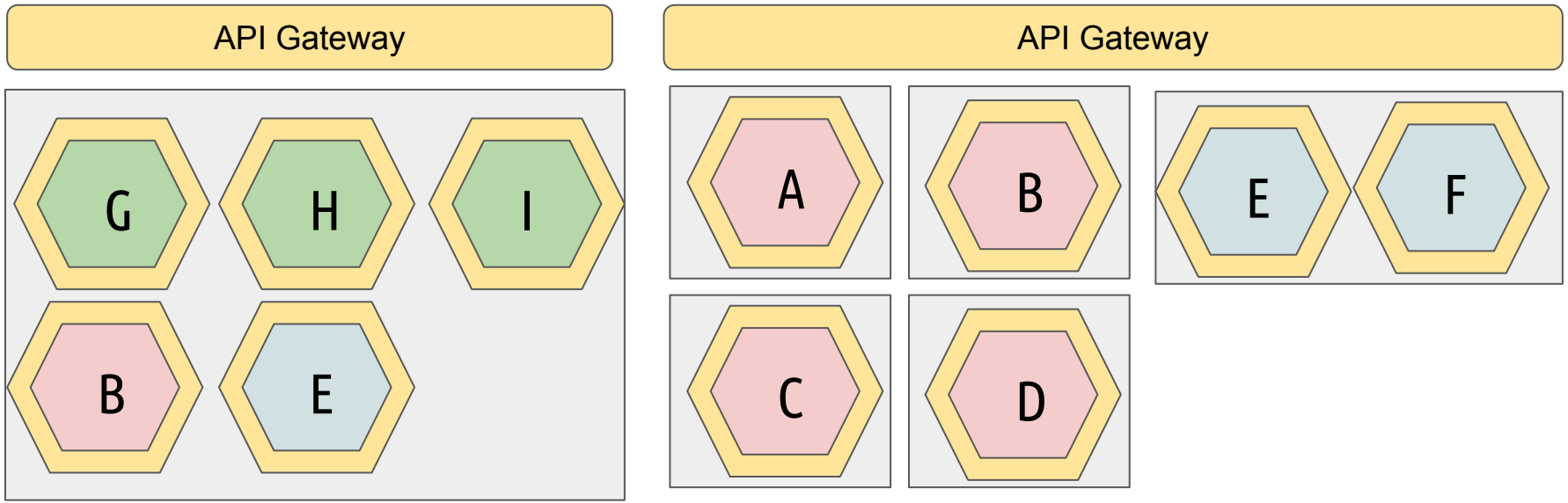
Oder so

Microservices



Oder sogar so

Mischformen



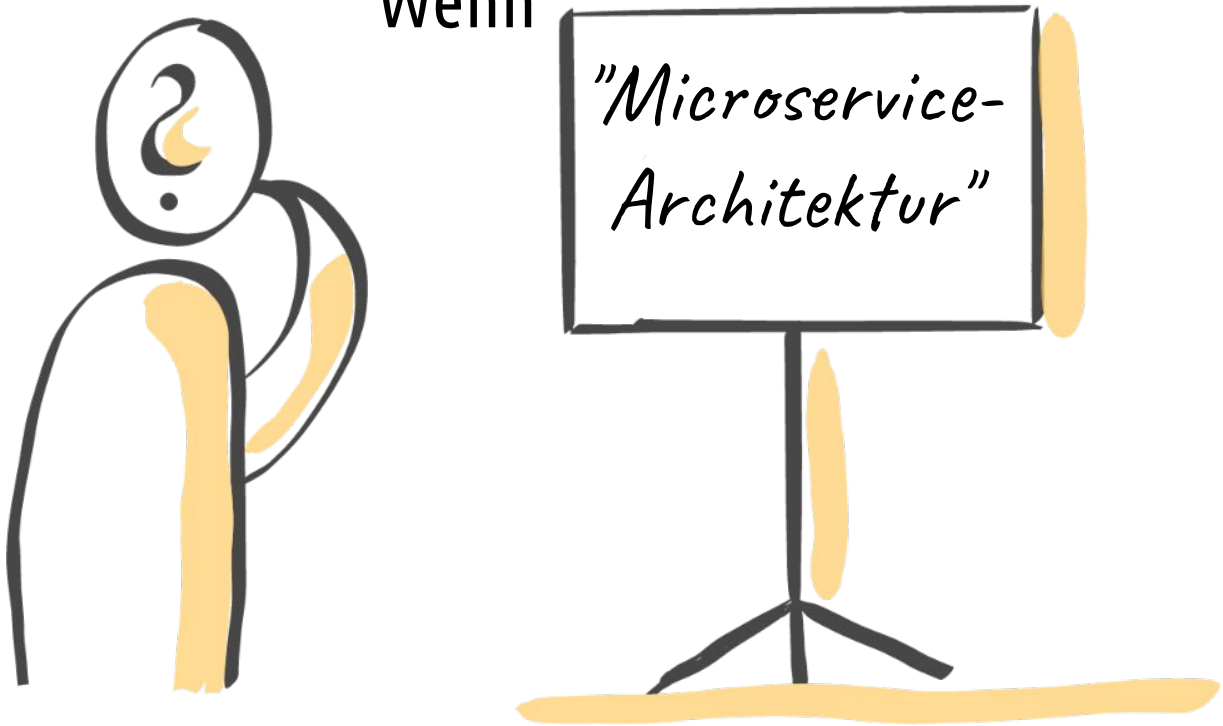
Tipps

Architekturziele / Qualitätsmerkmale “ehrlich” definieren, ohne Umsetzung im Kopf! Business-Ziele!

Welche (Qualitäts-)Anforderungen bestehen? Verschiedene Architekturstile daran bewerten

Denkt in Bounded-Contexts, nicht in Microservices!

Wenn



die Antwort ist,
was war dann eigentlich die Frage?

peter.fichtner@fiduciagad.de

@petfic

tilmann@tgnowledgy.me

@TGnowledgy