

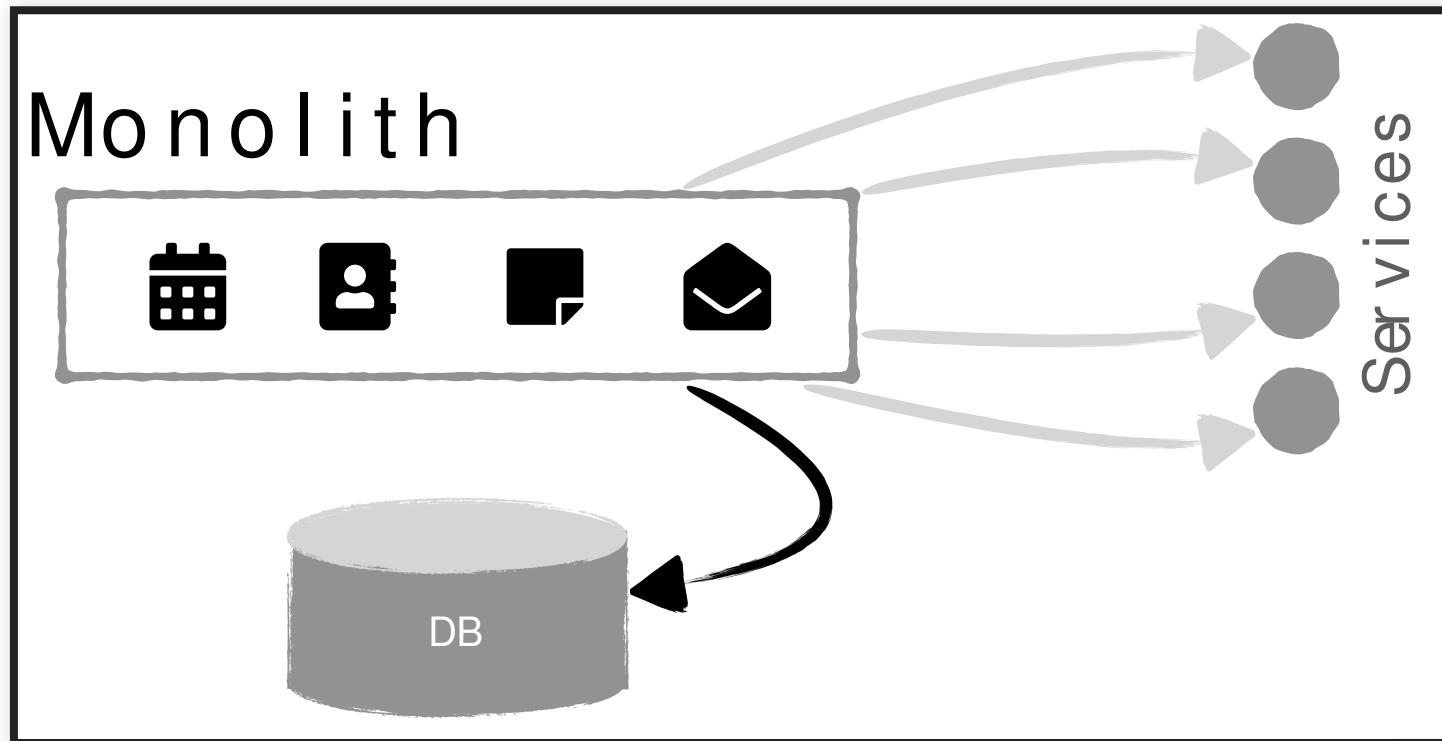
# KAPITEL 3: ARCHITEKTURSTILE UND REFERENZARCHITEKTUREN



# LERNZIELE

- Erklären was Vor- und Nachteile eines Monolithen sind
- Eigenschaften von Micro-Service Architekturen erklären
- Micro-Service Architekturen von anderen Ansätzen (wie SOA oder Middleware-Ansätzen) differenzieren
- REST-APIs und gRPCs unterscheiden
- Einflussfaktoren auf die Größe eines Micro-Services erklären können
- Hexagonale Architekturen auf ein konkreten Anwendungsfall anwenden
- Ereignisbasierte Konzepte mit Request-Response-Ansätzen differenzieren
- Komponenten einer typischen Cloud-Infrastruktur nennen und erklären können

## 3.1 EINLEITUNG



*Vom Software-Monolithen zu verteilten Architekturen*



# SOFTWARE-ARCHITEKTUR

*Softwarearchitektur: Die grundsätzliche Organisation eines Systems, verkörpert durch dessen Komponenten, deren Beziehung zueinander und zur Umgebung sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten.*

Gernold Starke: Effektive Softwarearchitekturen. Hanser, 9. Auflage. 2020.

## *Abgrenzung Entwurfsmuster*

- **Entwurfsmuster** beschreiben einfache und elegante Lösungen für häufige Entwurfsprobleme
- Architektur vs. Entwurfsmuster: Im Gegensatz zu Entwurfsmustern oder Idiomen bestimmen Architekturen nicht ein konkretes (meist kleines oder lokales) Teilproblem, sondern die grundlegende Organisation und Interaktion zwischen den Komponenten einer Anwendung.



## *Architekturmuster*

- Komponenten-Strukturen: Blackboard, Pipes und Filter, Domain-driven Design, Command Query Responsibility Segregation
- Verteilte Systeme: Client-Server, Peer-to-Peer, Service-oriented Architecture, Edge-Computing, Microservice-Architecture
- Interaktive Systeme: Model-View-Controller, Model-View-ViewModel, Model-View-Presenter, usw.

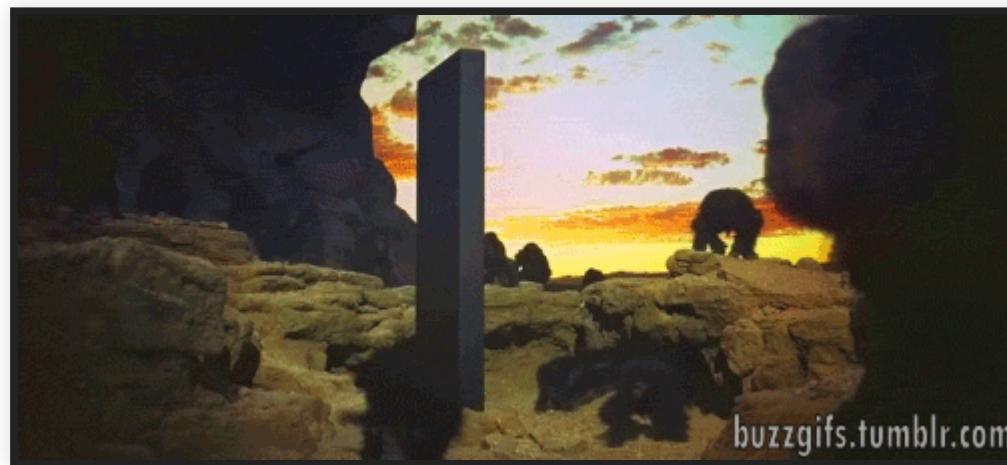


## *Referenzarchitektur*

*Eine Referenzarchitektur ist eine IT-Architektur, die standardisierend für die IT-Architekturen einer Gruppe von Informationssystemen wirkt.*

Gernold Starke: Effektive Softwarearchitekturen. Hanser, 9. Auflage. 2020.

# SOFTWARE-MONOLITH



- Ein Build und Deployment Element
- Eine Code Basis
- Ein Technologie Stack



## **Einfaches mentales Modell für Entwickler**

Einheitlicher Zugang für Development, Build und Deployment

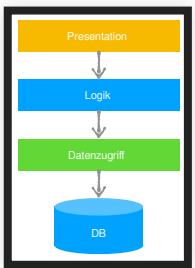
**„Einfaches“ Skalierungsmodell für den Betrieb**

Mehrere Kopien hinter einem Loadbalancer laufen lassen

## Nachteile

- Riesige und einschüchternde Codebasis für Entwickler
- Entwicklungswerkzeuge werden überlastet
  - Refactoring dauert Minuten
  - Builds dauern Stunden
  - Testen in kontinuierlicher Integration dauert Tage
- Skalierung ist begrenzt
  - Das Ausführen einer Kopie des gesamten Systems ist ressourcenintensiv.
  - Es skaliert nicht mit dem Datenvolumen out-of-the-box
- Deployment ist begrenzt
  - Re-Deployment bedeutet, das ganze System zu stoppen.
  - Re-Deployment scheitern und erhöhen das wahrgenommene Risiko des Einsatzes.

- Schichtsystem zur Zerlegung von Monolithen
- (Normalerweise) Präsentation, Logik, Datenzugriff mit einem Technologiestapel pro Lage
- Beispiele für Technologien
  - Präsentation: Servlets / JSPs, EJB-Client, HTML/JavaScript
  - Logik: EJB-Container, Beans
  - Datenzugriff: Linux, JVM, EJB JPA, EJB Container, Libs



**„Einfach“ verständliches Modell, einfache Abhängigkeiten**

**„Einfaches“ Deployment und Skalierungsmodell**

## Herausforderungen

- Riesige Codebasen (eine pro Schicht) mit den gleichen Auswirkungen auf Development, Build und Deployment
- Skalierung funktioniert besser, aber immer noch begrenzt
- Begrenztes Personalwachstum: Im Allgemeinen funktioniert Team pro Schicht, vgl. Convey's Law (später)
- Entwickler werden zu Spezialisten auf ihrer Ebene
- Die Kommunikation zwischen den Teams wird durch die Erfahrung der Schichten (oder deren Fehlen) beeinflusst.



*Weitere Architekturkonzepte innerhalb der Software anwendbar, um die interne Struktur zu verbessern.*

**Ziel in der Veranstaltung: Weg von einem Software-Monolithen hin zu vielen kleineren (unabhängigen?) Software-Komponenten**



## ÜBUNG: SCHLECHTES BEISPIEL

Nutzen Sie das bereitgestellte Projekt-Archiv `java-bad.zip`, entpacken Sie dieses und importieren Sie den resultierenden Ordner.

# ÜBUNG: ARCHITEKTUREN

*Welche Aussagen stimmen? Software-Architekturen ...*

- ... beziehen sich ausschließlich auf Entwurfsmuster wie Observer, Adapter, Factory oder Proxy.
- ... können zum Beispiel Client-Server-Modelle als auch Muster wie *Pipes und Filter*
- ... beschreiben die grundsätzliche Organisation eines Systems, dessen Komponenten, deren Beziehung zueinander und zur Umgebung.
- ... beziehen sich auf ein Referenzmodell für eine Klasse von Architekturen.

# ÜBUNG: MONOLITHEN

*Welche Aussagen stimmen?*

- Skalierung ist mit Monolithen nicht möglich
- Skalieren von Monolithen kann ressourcen Intensiv sein, da Bestandteile der Anwendung skaliert werden, die ggf. nicht unter starker Last sind
- Monolithen haben häufig eine einfach verständliche und überschaubare Code-Basis.
- Monolithen können ein einfaches mentales Modell für Entwickler haben

## 3.2 SERVICE-ORIENTIERTE ARCHITEKTUR



## *Was ist ein Service?*

*A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities. To be used, a service must be realized by a concrete provider agent.*

<https://www.w3.org/TR/ws-gloss/>

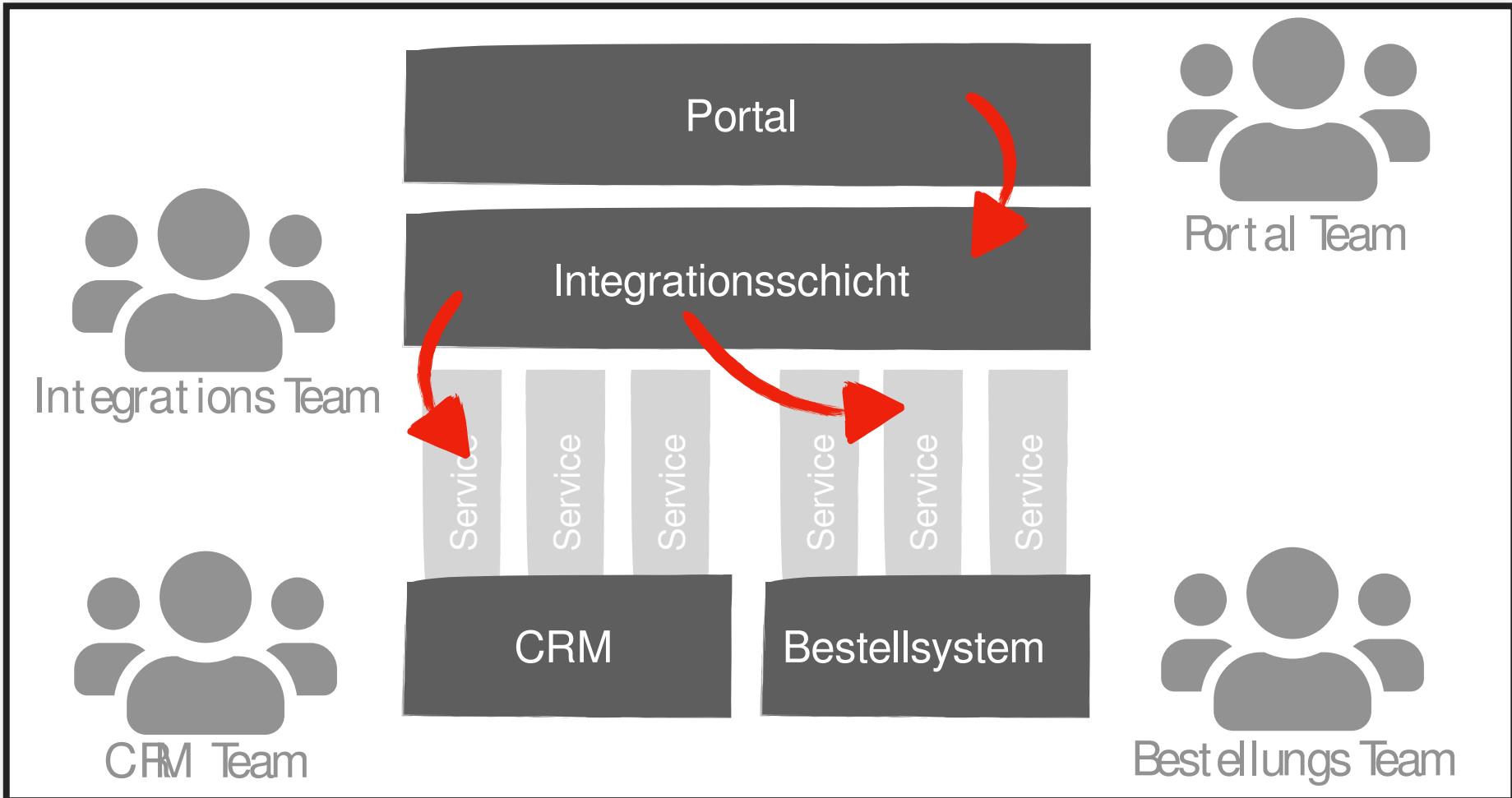


## *Was ist eine service-orientierte Architektur?*

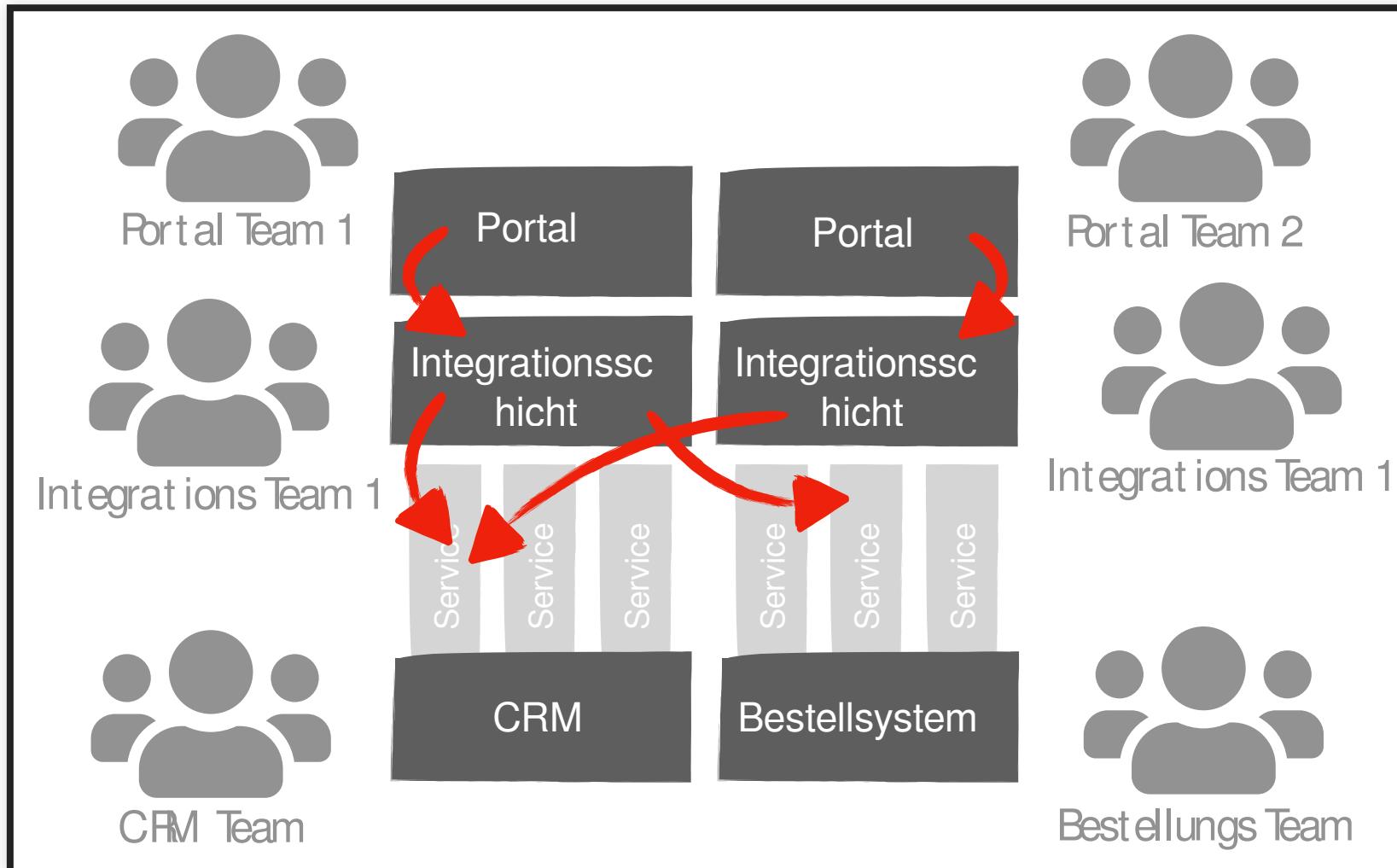
*A set of components which can be invoked, and whose interface descriptions can be published and discovered.*

<https://www.w3.org/TR/ws-gloss/>

Consumer bedienen sich Services zur Abbildung von Geschäftsprozessen, welche den Zugriff auf die Geschäftskomponenten regeln



Services können darüber hinaus in mehreren Geschäftsprozessen Anwendung finden



*SOA ist Aufteilung eines großen Systems in Services*

- Umfasst fachliche Funktionalität
- Eigenständig nutzbar
- Im Netzwerk verfügbar
- Klar definierte Schnittstelle, deren Kenntnis ist ausreichend zur Nutzung des Services
- Freie Wahl des Werkzeugs zur Anbindung der Schnittstelle
- Service ist in einem Verzeichnis registriert

# ÜBUNG: SERVICE-ORIENTIERTE ARCHITEKTUREN

*Welche Aussagen stimmen?*

- Service-orientierte Architekturen setzen Integrationsaufgaben in den Services um
- In service-orientierten Architekturen greifen Portale direkt auf Dienste zu
- Service-orientierte Architekturen umfassen die eigentlichen Dienste, Integrationsschichten, Portale und die zugrundeliegenden Backend-Systeme
- Beschreibt eine Menge von Komponenten die aufgerufen werden können und deren Schnittstellen gefunden werden können und die zugehörigen Beschreibungen veröffentlicht sind

## 3.3 MICROSERVICE ARCHITEKTUR



Monolith

[http://openphoto.net/download/index.html?image\\_id=6987](http://openphoto.net/download/index.html?image_id=6987)



Microservice

<https://pixabay.com/de/containerhafen-verladen-gestapelt-2921882/>

### *Was ist das?*

- Ein *Programm* sollte nur eine Aufgabe erfüllen, und es sollte sie gut erfüllen
- *Programme* sollten zusammenarbeiten können
- *Programme* sollten eine universelle Schnittstelle verwenden

**Ziel:** Wiederverwendbare spezialisierte *Programmen*, die letztlich eine Art Komponente sind.

- Microservices stellen eine Art Modularisierungskonzept dar
- Microservices können unabhängig voneinander in die Produktion gebracht werden
- Änderungen an einem einzelnen Microservice erfordern nur, dass dieser Microservice in Produktivbetrieb genommen werden muss
- Microservices können nicht mittels Modularisierungskonzepten realisiert werden. Diese Konzepte erfordern in der Regel, dass alle Module zusammen in einem Programm / Prozess ausgeliefert werden.

- Microservices sollten als virtuelle Maschinen, als leichtere Alternativen wie Container oder als einzelne Prozesse implementiert und betrieben werden
- Die Realisierung von einem Microservice ist nicht an eine bestimmte Technologie gebunden
- Microservices können eigene unterstützende Dienste wie Datenbanken oder andere Infrastrukturen mitbringen

## *Ergänzung zum Thema Datenbanken*

- Microservices sind eigenständig für die Datenhaltung verantworten, d.h. eine eigene Datenbank oder ein eigenes Schema in einer gemeinsamen Datenbank
- Die gemeinsame Nutzung von Datenbankschemata macht Änderungen an den Datenstrukturen praktisch unmöglich. Da dies die Änderbarkeit der Anwendung stark beeinträchtigt, sollte diese Art der Kopplung verhindert werden.
- Microservices können Daten replizieren



*Microservices müssen miteinander kommunizieren können. Dies kann auf unterschiedliche Weise erreicht werden:*

- Microservices können über ihre HTML-UI (falls vorhanden) Links zu anderen Microservices nutzen
- Microservices können in ihren HTML-Frontend (falls vorhanden) andere Microservices einbetten
- Microservices können über Request-Response-basierte Protokolle wie HTTP (z.B. auf Basis von REST) kommunizieren
- Microservices können über Messaging-Protokolle wie MQTT kommunizieren
- Grobgranularität zur Reduktion von Abhängigkeiten, SOA fokussiert eher auf größere Services

# ÜBUNG: MICROSERVICE-EIGENSCHAFTEN

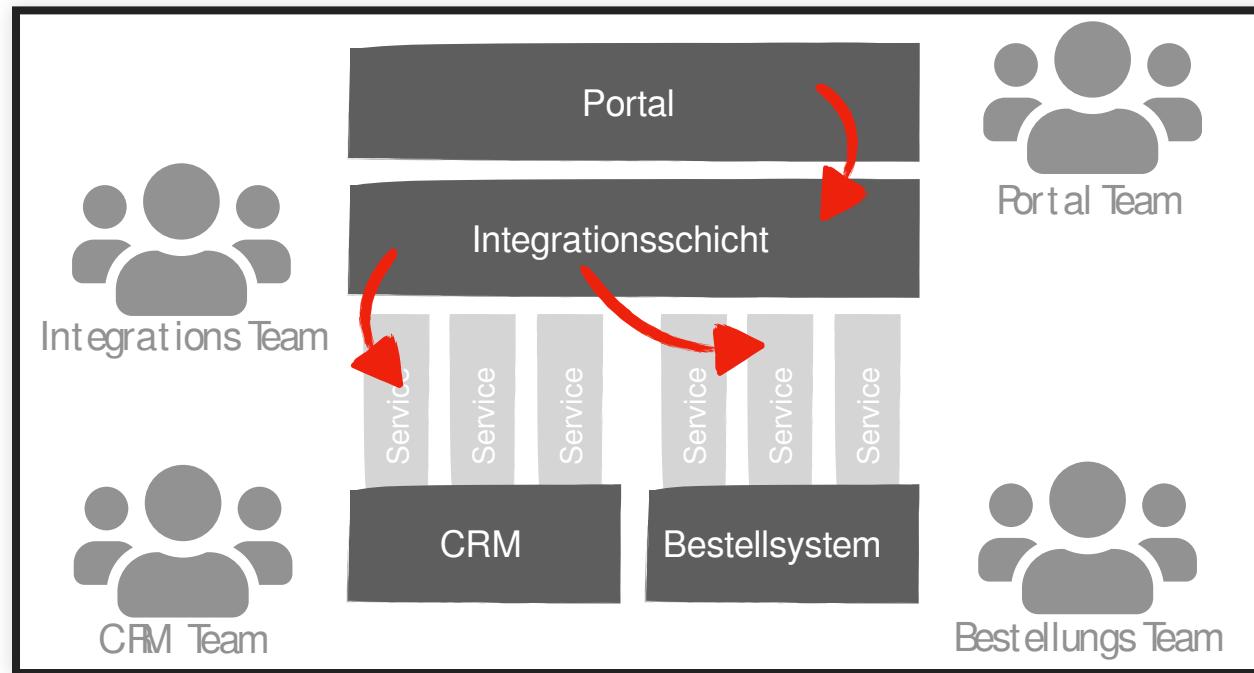
*Welche Aussagen stimmen?*

- Microservices können Änderungen unabhängig in den produktiven Betrieb bringen
- Abhängigkeiten auf ein gemeinsames (Quellcode-)Modul ist in einer Microservice-Architektur möglich
- Microservices sind unabhängige Komponenten die nur über universelle Schnittstellen kommunizieren
- Microservices können auf einer gemeinsamen Datenbasis arbeiten
- Replikation von Daten über mehrere Microservices ist ggf. notwendig



# ABGRENZUNG

## Abgrenzung zur service-orientierten Architektur

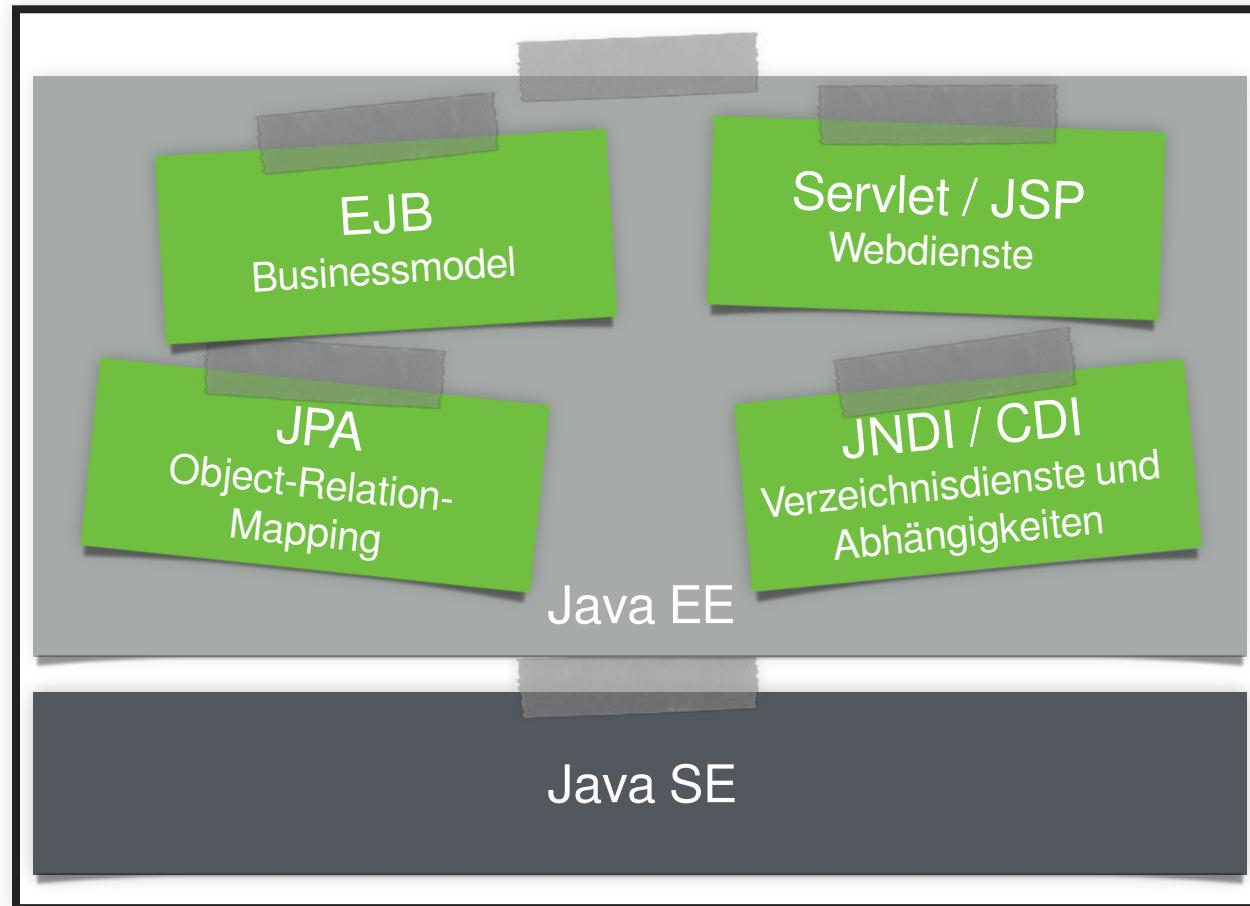


- SOA und Microservices streben nach Aufteilung
- In beiden Ansätzen:
  - werden Informationen über Netzwerke ausgetauscht
  - ist die synchrone und asynchrone Kommunikation möglich
- Unterschied in der Orchestrierung: Microservices haben keine Logik auf Integrationsebene
- Unterschiede in der Flexibilität und Architektur
- Microservices setzen auf Isolation

## *Abgrenzung zu Nanoservices*

- Amazon Lambda erlaubt den Einsatz einzelner Funktionen, die in Java, Node.js oder Python geschrieben sind.
- Jede Funktion wird automatisch überwacht
- Jeder Aufruf einer Funktion wird abgerechnet.
- Verwendung einer solchen Infrastruktur ermöglicht die Erstellung von Diensten, die nur aus wenigen Zeilen Code bestehen.
- Der Aufwand für die Bereitstellung und den Betrieb ist in der entsprechenden Infrastruktur gering.

## Abgrenzung zu modularen Konzepten wie Jakarta EE



- Modularisierungskonzept
- Komponenten (bzw. Dienste) werden in der selben Ausführungsumgebung betrieben
- Kommunikation über direkte Aufrufe
- Ressourcen zur Laufzeit sind nicht voneinander isoliert
- Kritische Fehler einer Komponente kann einen Einfluss auf das gesamtes Softwaresystem
- Ein einzelner Dienst könnte dagegen mithilfe von Jakarta EE Konzepten realisiert werden



# WIE GROSS SOLLTE EIN SERVICE SEIN?



- Es geht nicht um Lines of Code (LoC)
- Ein Microservice sollte von einem Team entwickelt werden (vgl. Amazons Two-Pizza-Teams)
- Microservices stellen einen Modularisierungskonzept dar: Entwickler sollten in der Lage sein, einzelne "Module" zu verstehen
- Microservice sollten austauschbar sein: Wenn ein Microservice nicht mehr gewartet werden kann oder beispielsweise eine leistungsfähigere Technologie eingesetzt werden soll, kann der Microservice durch eine neue Implementierung ersetzt werden.

## Herausforderung

- Verteilte **Kommunikation** zwischen Microservices über das Netzwerk ist teuer und ist mit Bedacht einzusetzen
- Der **Austausch von Code** über die Grenzen von Microservice ist schwierig
- Eine **Transaktion** innerhalb eines Microservice ist einfach zu implementieren. Über die Grenzen eines einzelnen Microservice hinaus ist dies nicht mehr trivial, da verteilte Transaktionen notwendig werden. Es ist häufig sinnvoll, sich für eine Microservice-Größe zu entscheiden, die es erlaubt, eine Transaktion vollständig in einem Dienst abzuwickeln.



- Gleiches gilt für die **Konsistenz der Daten**: Wenn beispielsweise der Kontostand mit dem Ergebnis von Einnahmen und Ausgaben konsistent sein soll, ist es empfehlenswert, dies in einem Microservice umzusetzen, anstatt mehrere zu verwenden.
- Microservices müssen eigenständig in die Produktion gebracht werden und benötigten daher eine eigene Ausführungsumgebung. Dies verbraucht **Hardware-Ressourcen** und bedeutet zudem, dass der Aufwand für die Systemadministration steigt.

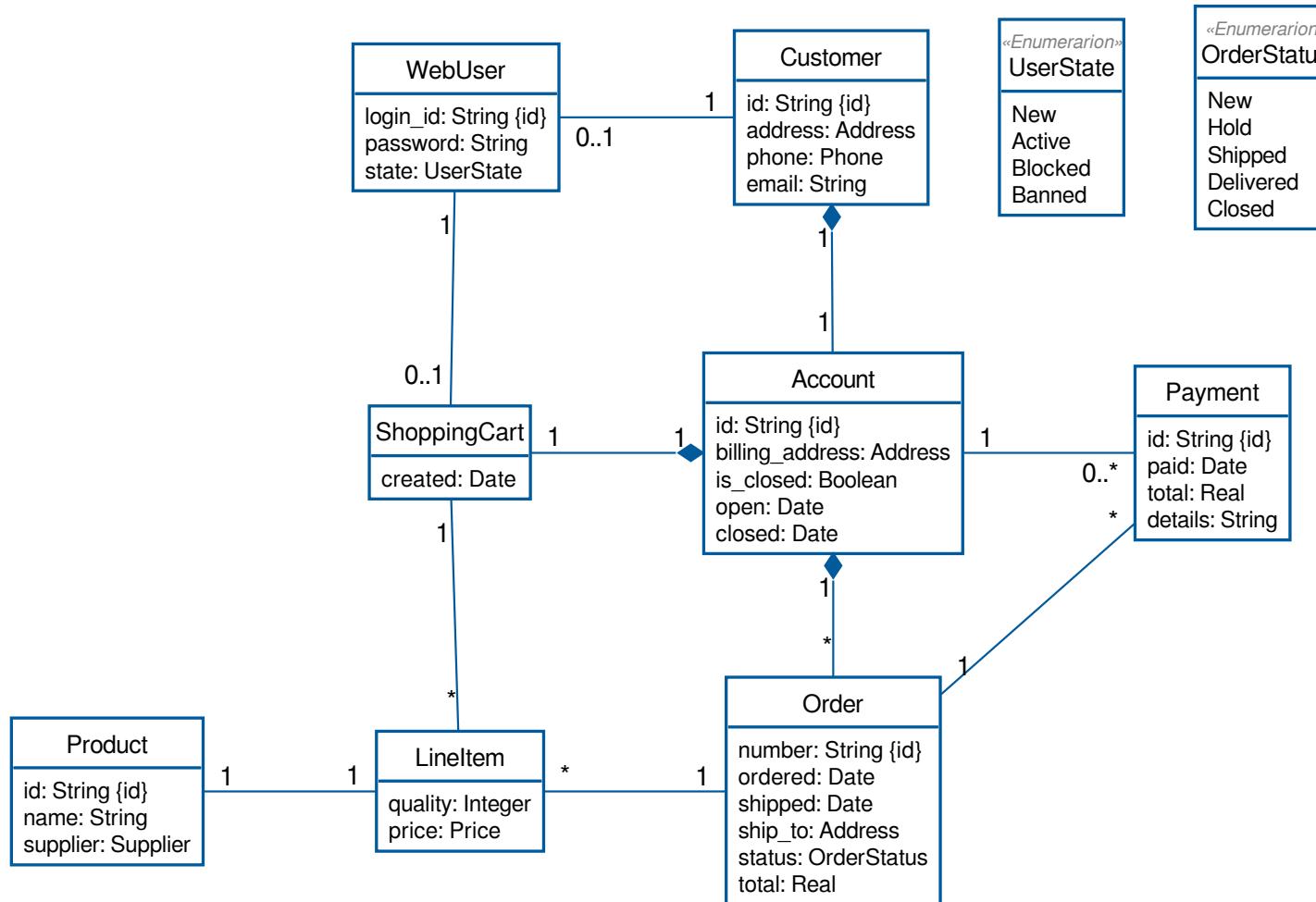


## Strategie: Bounded Context und Domain-Driven Design

Eine Möglichkeit zur Abgrenzung der Größe eines Microservices ist mittels Domain-Driven Design (DDD), welches grundlegender Muster definiert (Building Blocks):

- **Entities** sind Objekte mit Identitäten, z. B. Kunden und Waren
- **Value Objects** sind Objekte ohne Identitäten, wie z. B. Adressen welche nur im Zusammenhang mit Kunden sinnvoll sind
- **Aggregates** sind zusammengesetzte Domänenobjekte, z. B. Bestellungen als Aggregat von Bestellzeilen
- **Services** enthalten Geschäftslogik welche nicht direkt an Entities, Value Objects und Aggregates koppelbar ist
- **Repositories** beschreiben die Gesamtheit von Entities eines Typs
- **Factories** helfen bei der Erzeugung komplexer Entitätsstrukturen

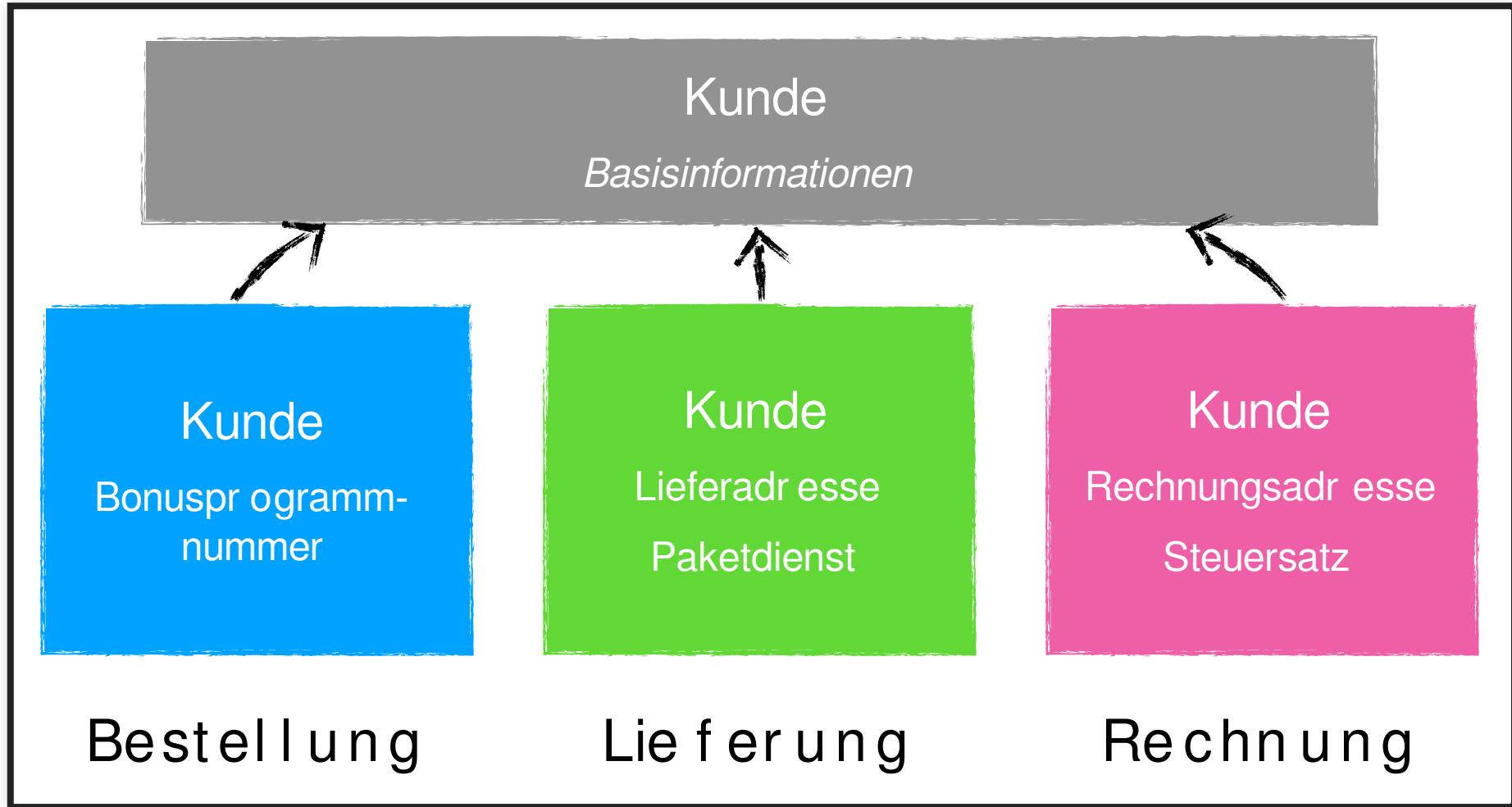
## Beispiel Shop: Ableiten von Diensten basierend auf dem Domainmodell



*Auf Basis der Strategic Designs die Beschreibung der Interaktion verschiedener Domainmodelle*

Kern des Strategic Designs ist der Bounded Context (Kontextgrenzen)

- z. B. Bestellung im Kontext der Lieferung: Größe und Gewicht
- z. B. Bestellung im Kontext der Buchhaltung: Preise und Steuersätze

*Bounded Context*



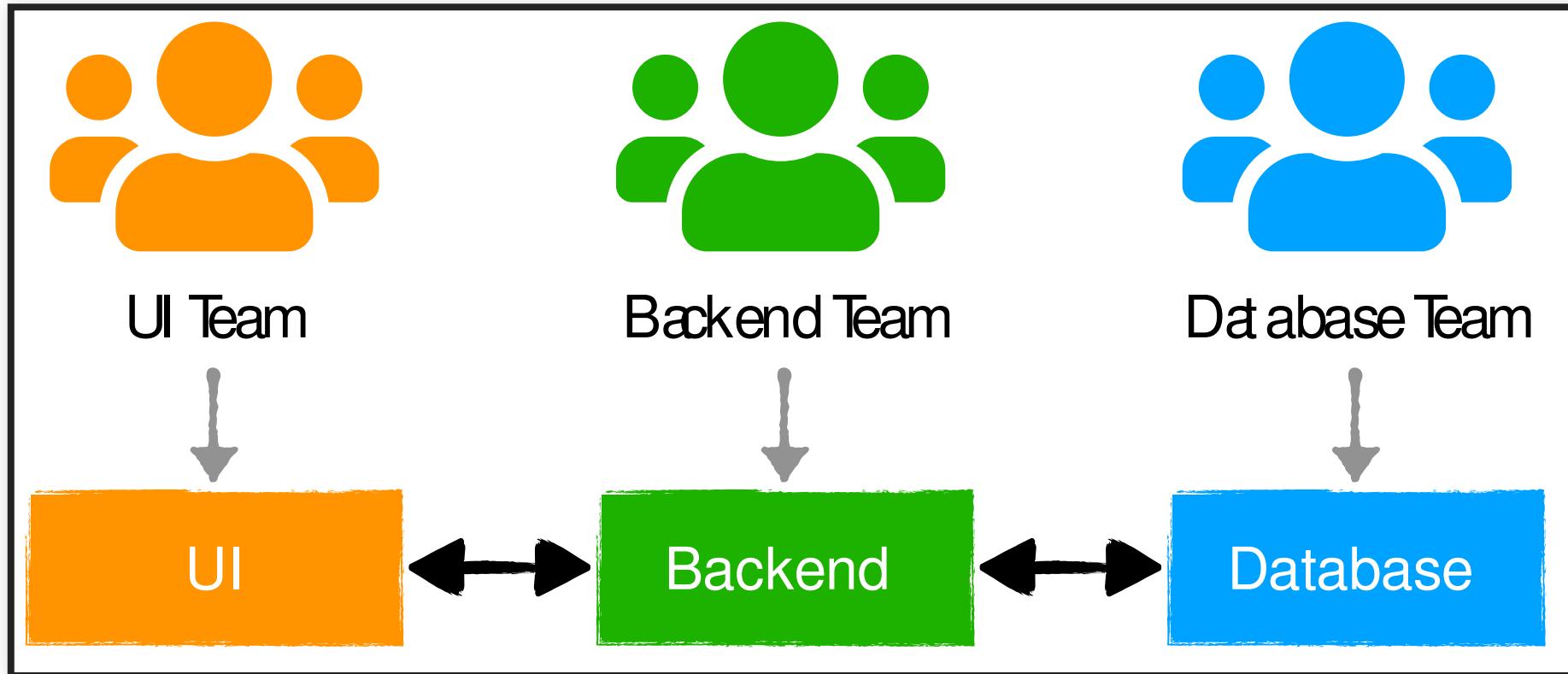
- Ein Domain Model ist nur in einem bestimmten Kontext sinnvoll - d.h. in einem Bounded Context
- Die Abgrenzung der Microservices kann einem Bounded Context je Microservice entsprechen.
- Die eigenständige Entwicklung von Features profitiert von der Aufteilung in Bounded Contexts: Wenn ein Microservice auch für einen bestimmten Teil der Daten zuständig ist, kann der Microservice Features einführen, ohne Änderungen an anderen Microservices zu verursachen.

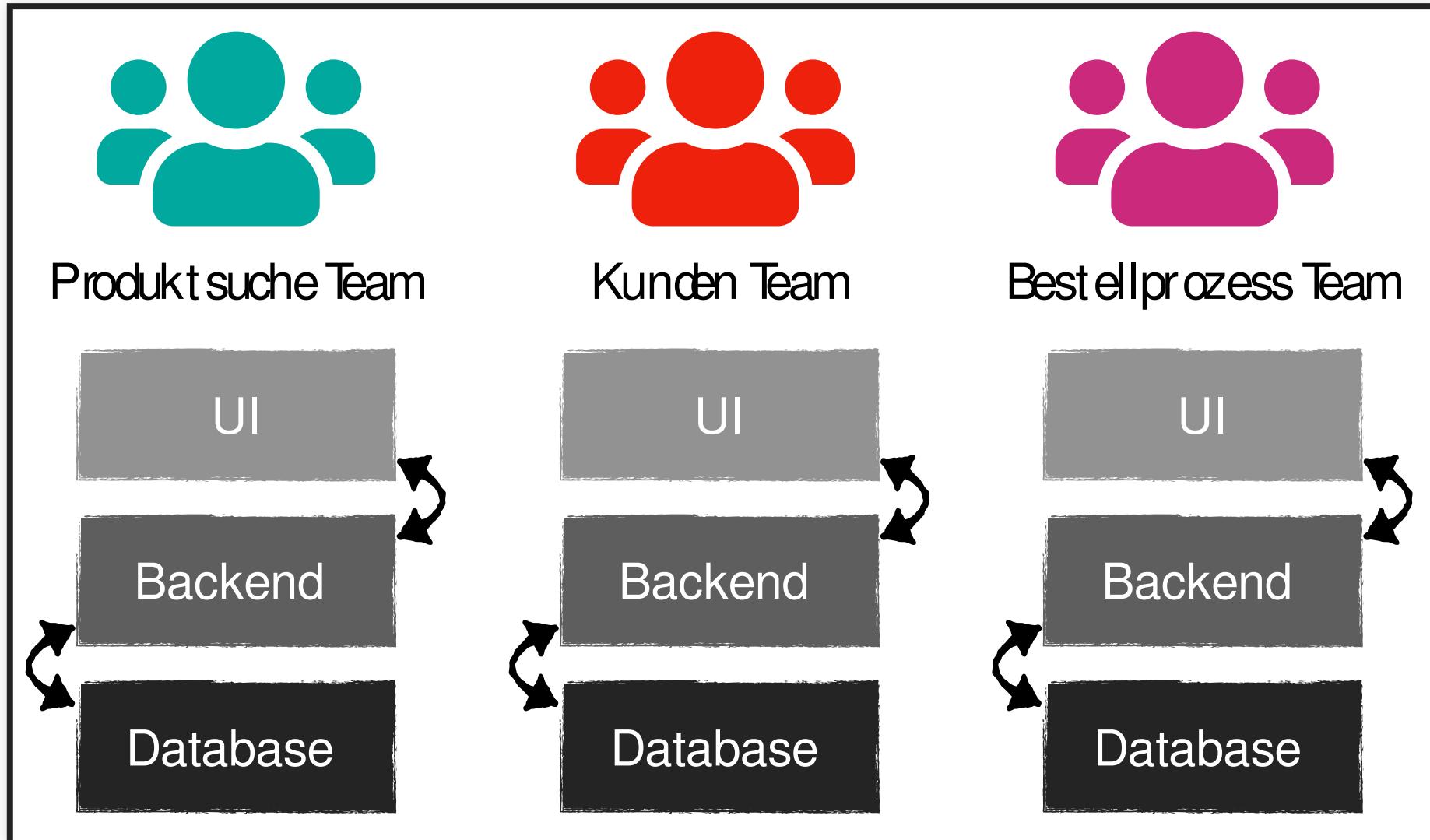


## Strategie: Conway's Law

*Organizations which design systems can only create such designs which reflect the communication structures of these organizations.*

Melvin Edward Conway







## Zusammenfassung zur Größe

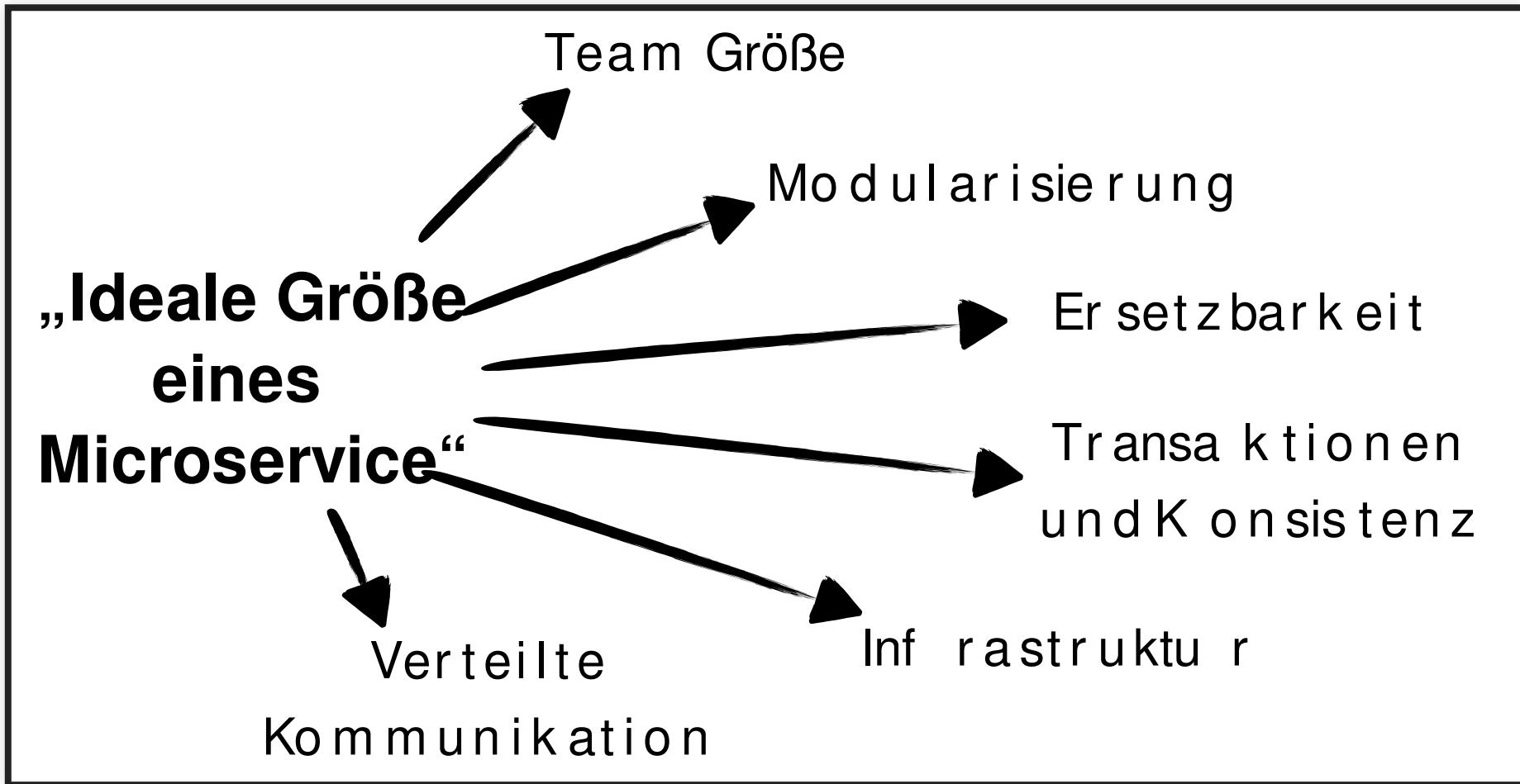
Die Frage nach der Größe von Microservices konzentriert sich auf die technische Struktur des Systems zur Definition von Microservices.

*Für die Verteilung nach Bounded Context ist die Domain Architecture ein wichtiger Aspekt der hilft Services zu identifizieren.*

*Das Conway's Law kann als enabler dienen, so dass Microservices auch Auswirkungen auf die Organisation haben.*

Zusammen ergeben diese Aspekte ein zutreffendes Bild von Microservices. Welcher dieser Aspekte der wichtigste ist, hängt vom Anwendungskontext der Microservice-basierten Architekturen ab.

## Einfluss auf Größe eines Microservices



# ÜBUNG: MICROSERVICE-GRÖSSE

*Welche Aussagen stimmen?*

- Informationen, bei denen Konsistenz wichtig ist, haben keinen Einfluss auf Größenentscheidungen
- Overhead bezieht sich (zum Beispiel) auf zusätzlichen Speicherbedarf der durch die Isolation der Dienste (z.B. virtuelle Maschinen) entsteht
- Das Verständnis der Quellcode-Basis eines Microservices sollte innerhalb des Teams gegeben sein
- Domain-Driven Design und das Konzept des Bounded Contexts kann als Einfluss auf die Microservice-Größe genutzt werden



# VORTEILE UND MÖGLICHKEITEN

Team Skalierbarkeit

Migration von Altanwendungen

Nachhaltige Entwicklung

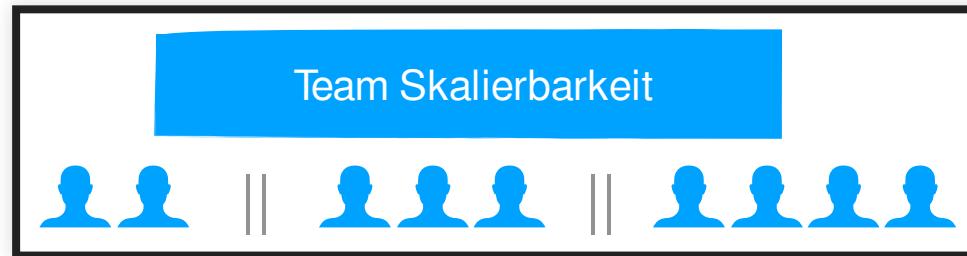
Ersetzbarkeit

Robustheit

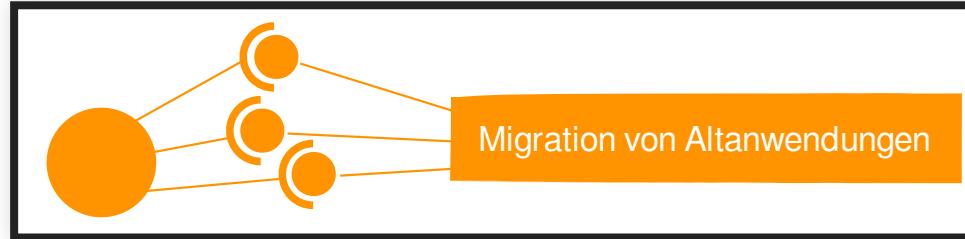
Unabhängige Skalierbarkeit

Kontinuierliche Lieferung

Technologie-Freiheit



- Microservices bieten Optionen für die Skalierung von agilen Projekten: Normalerweise müssen alle Teams koordiniert und gemeinsam an Features arbeiten.
- Wenn jedes Team seine eigenen Anforderungen hat und diese durch Änderungen an seinem eigenen Microservice umsetzen kann, können die Teams weitgehend unabhängig voneinander an Features arbeiten.



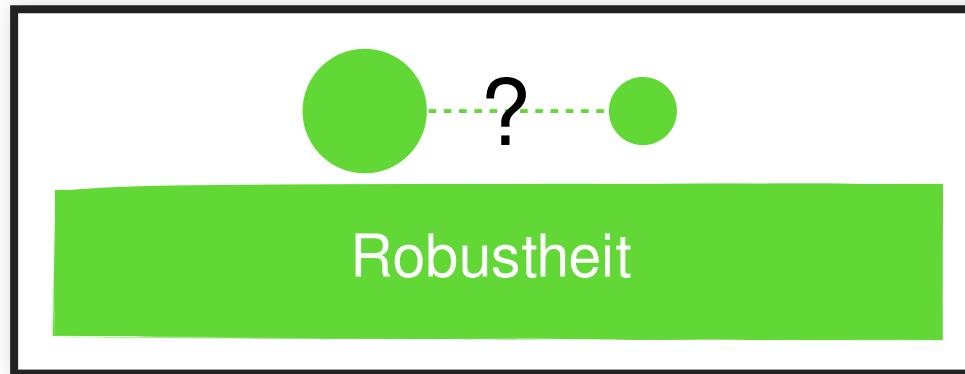
- Die Arbeit mit Altcode ist oft schwierig. Statt den Code des Altsystems zu modifizieren, wird das System an seinen externen Schnittstellen durch Microservices ergänzt oder teilweise ersetzt.
- Die einfache Integration von Microservices ist ein Grund, warum Microservices so interessant sind.
- Die Ablösung eines Altsystems durch eine Vielzahl von Microservices ist oft ein sehr nützlicher Ansatz, um in einem System schnell von Vorteilen wie Continuous Delivery zu profitieren.



- Microservice-basierte Architekturen verteilen ein System in mehrere unabhängig voneinander einsetzbare Dienste.
- Die Verteilung eines Systems in Microservices ist eine wichtige Architekturentscheidung. Er legt die Verantwortlichkeiten der Komponenten fest.
- Im Falle von Microservices ist es nicht so einfach, einen anderen Microservice einfach so zu nutzen. Microservices haben jeweils eine Schnittstelle und können nur über ihre Schnittstellen genutzt werden.
- Dies erfordert den Aufruf der Schnittstelle über Technologien wie REST oder Messaging. Dies geschieht nicht nur aus Versehen.

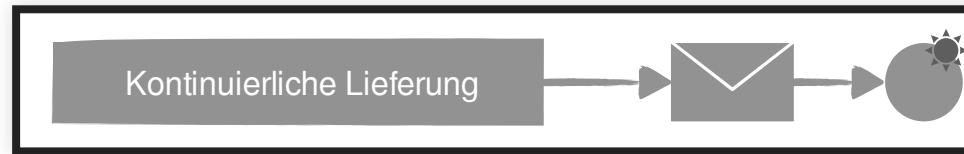


- Ohne großen Aufwand kann ein Microservice durch eine neue Implementierung ersetzt werden.
- Kommunikation ausschließlich über sync. oder async. Kommunikationsschnittstellen - korrekte Bereitstellung oder Nutzung ermöglicht u.a. beliebigen Austausch



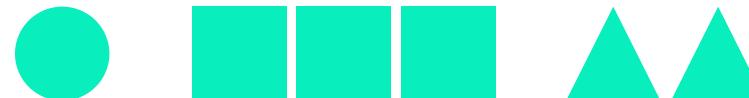
- Wenn eine Funktionalität in einem Deployment-Monolithen viel CPU oder Speicher verbraucht, sind ebenso andere Module betroffen. Wenn im schlimmsten Fall ein Modul den Ausfall des Systems verursacht, sind auch alle anderen Module nicht mehr verfügbar.
- Ein Microservice ist ein separater Prozess oder eine isolierte virtuelle Maschine. Daher hat ein Problem in einem Microservice keinen Einfluss auf einen anderen Microservice.

- Trotzdem sind Microservices verteilte Systeme. Sie laufen auf mehreren Servern und nutzen das Netzwerk. Server und Netzwerk können ausfallen. Daher müssen Microservices vor dem Ausfall anderer Microservices geschützt werden. Das nennt man Resilienz.
  - Resilienz kann auf sehr unterschiedliche Weise implementiert werden: Wenn ein Bestellvorgang nicht abgeschlossen werden kann, kann es eine Option sein, es später noch einmal zu versuchen.
  - Wenn eine Kreditkarte nicht verifiziert werden kann, besteht die Möglichkeit, die Bestellung bis zu einer bestimmten Obergrenze auszuführen. Was diese Obergrenze ist, müsste im Rahmen der Anforderungen des Systems entschieden werden.



- In der Commit-Phase werden Unit-Tests und statische Code-Analysen durchgeführt.
- Automatisierte Akzeptanztests stellen sicher, dass die Software die Funktionen korrekt implementiert.
- Kapazitätsprüfungen hingegen prüfen, ob die Leistung stimmt und ob die zu erwartende Belastung bewältigt werden kann.
- Manuelle Tests können neue Funktionen, aber auch fehleranfällige Bereiche ansprechen.
- Am Ende geht die Software in Produktion.

## Unabhängige Skalierbarkeit



- Jeder Microservice läuft als einzelner Prozess, manchmal sogar in einer separaten virtuellen Maschine.
- Dies erlaubt es, nur den betreffenden Microservice zu skalieren, wenn eine bestimmte Funktionalität besonders stark genutzt wird, während die anderen Microservices weiterhin mit der gleichen Kapazität laufen.



- Grundsätzlich kann jeder Microservice in einer jeweils anderen Technologie implementiert werden.
- Das macht das System insgesamt komplexer.
- Diese Komplexität kann durch den Einsatz von Standards für Betrieb, Überwachung, Protokollformate oder Deployment begrenzt werden.
- Damit ist zumindest eine weitgehend einheitliche Arbeitsweise gewährleistet.

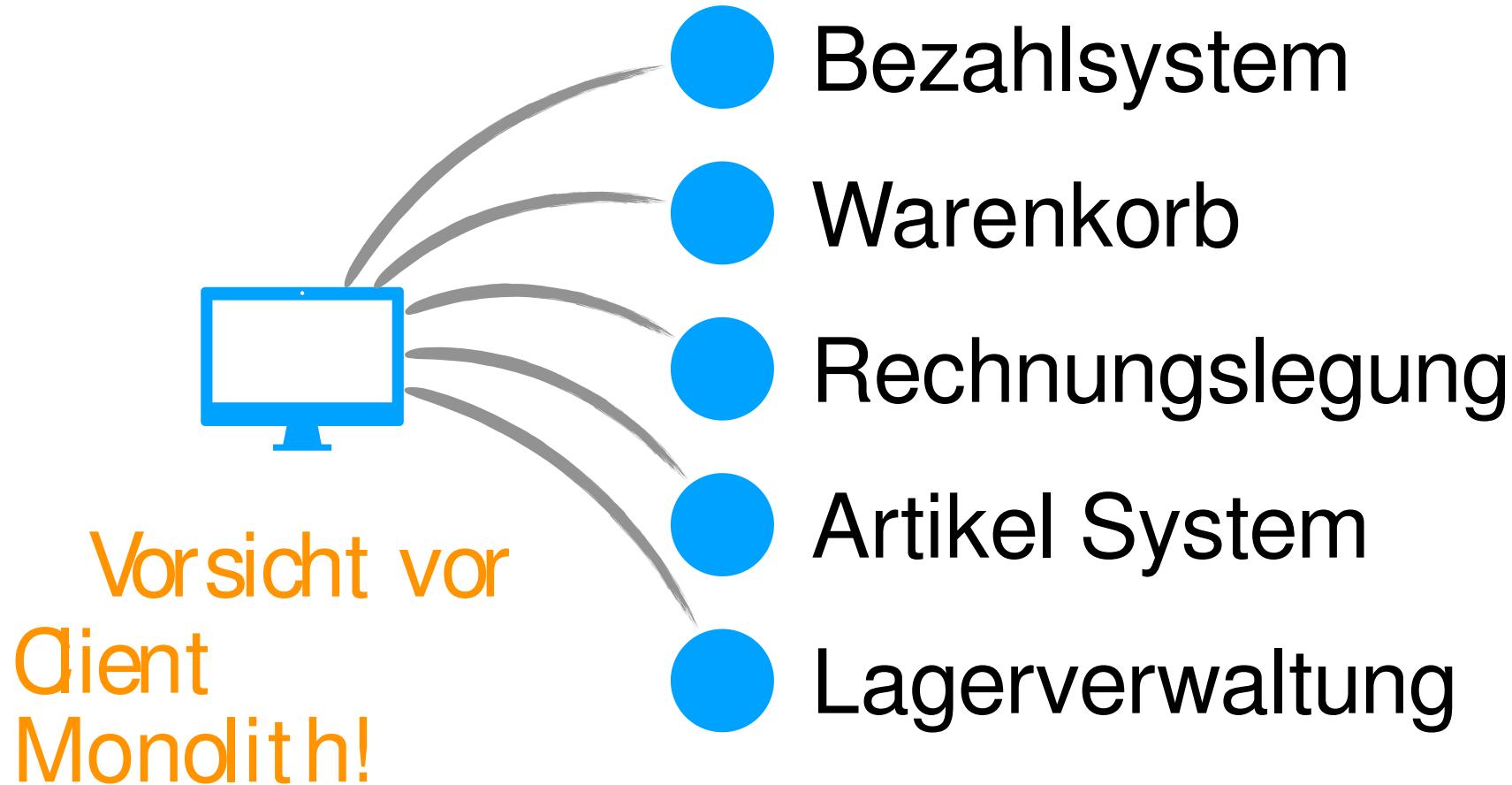


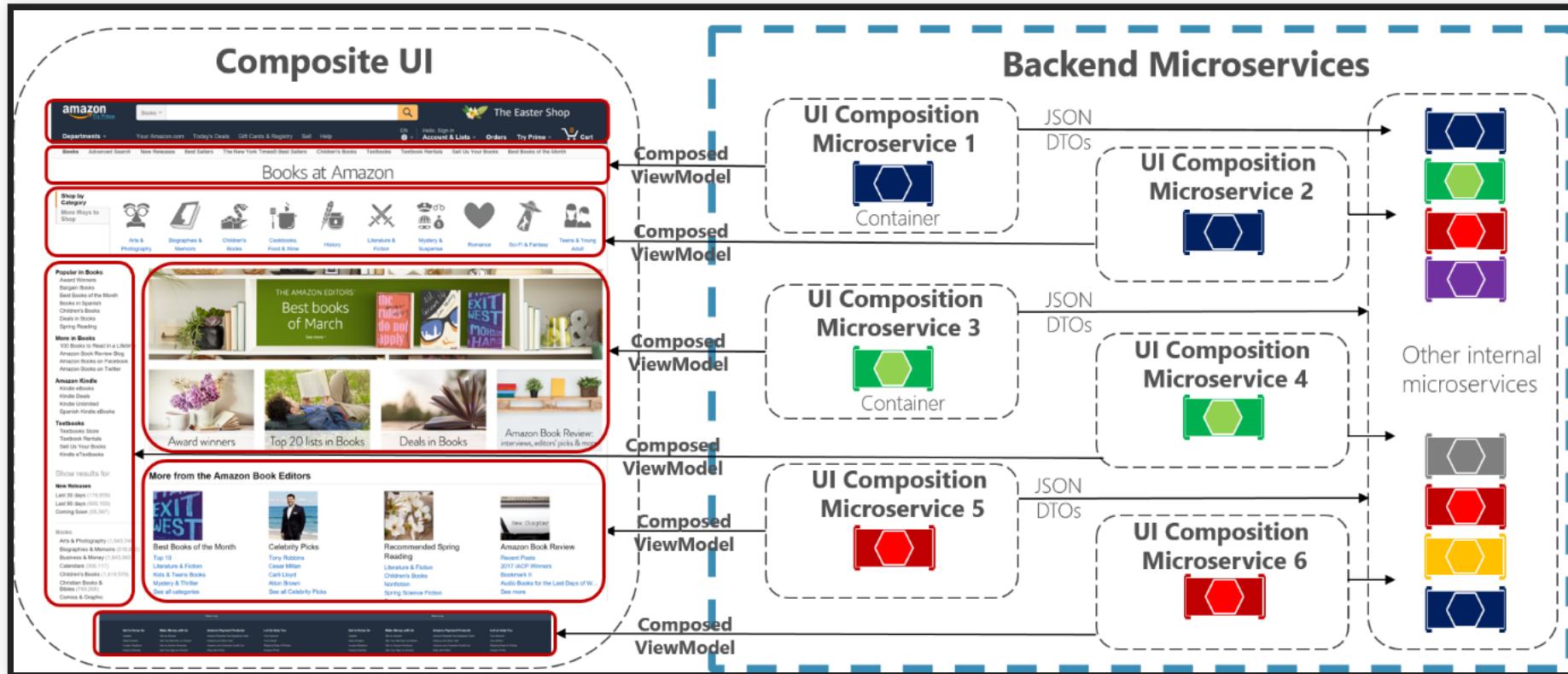
*... the **microservice architectural style** is an approach to developing a single application as a suite of **small services**, each running in its **own process** and **communicating** with **lightweight mechanisms**, often an **HTTP resource API***

Martin Fowler



## RISIKO: CLIENT MONOLITHEN





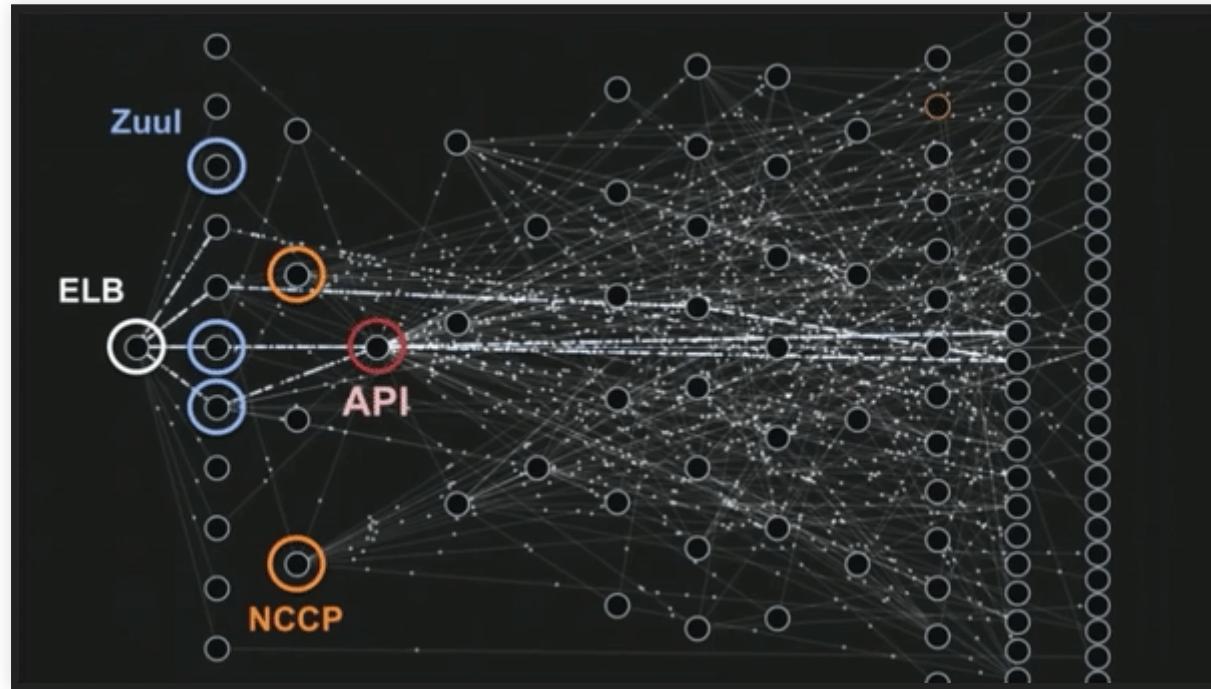
## *Strategien*

- Dienste liefern ihre eigenen UI - z.B. Integration über Verlinkung / Weiterleitung zw. Diensten
- "Page-Skeletons" und komponentengetriebene Diensteinbettung, vgl. Angular und Directives oder HTML / Web Components / Shadow DOM
- Spezialisierung von Clients mit klaren Funktionalitäten

# BEISPIEL: NETFLIX

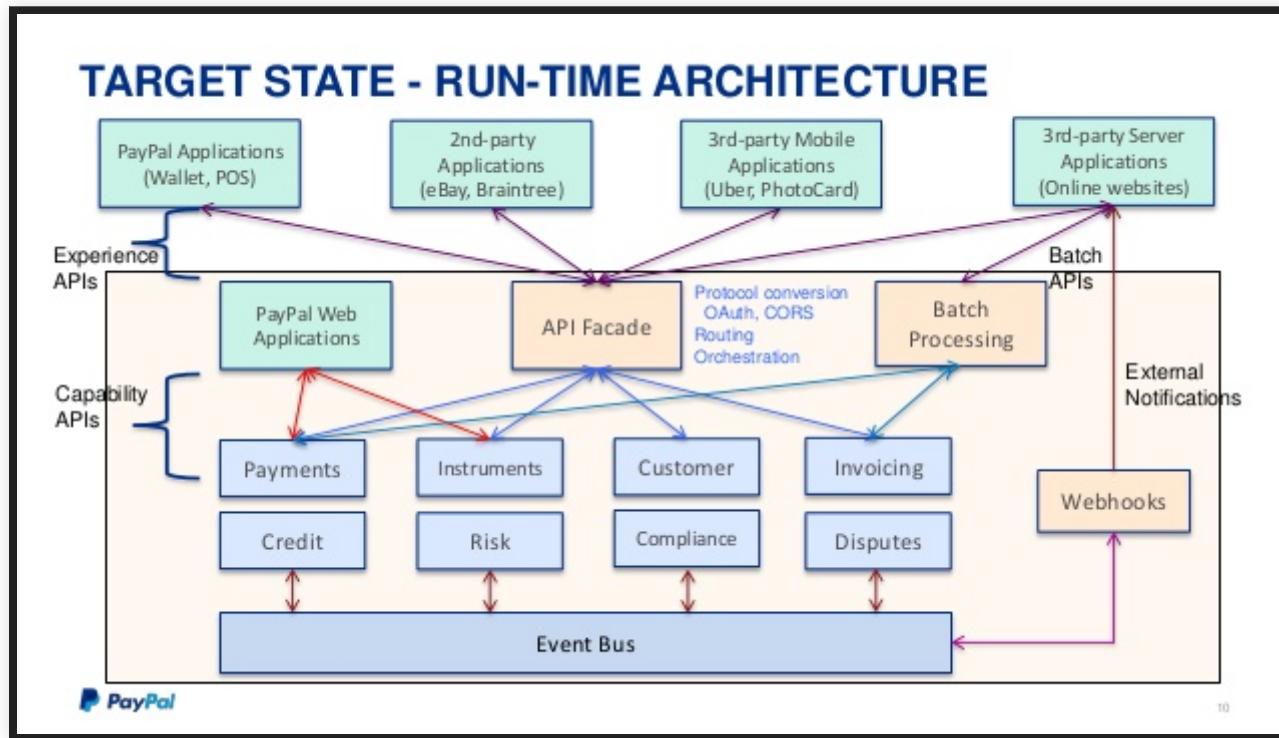


<https://www.youtube.com/watch?v=CZ3wluvmHeM>



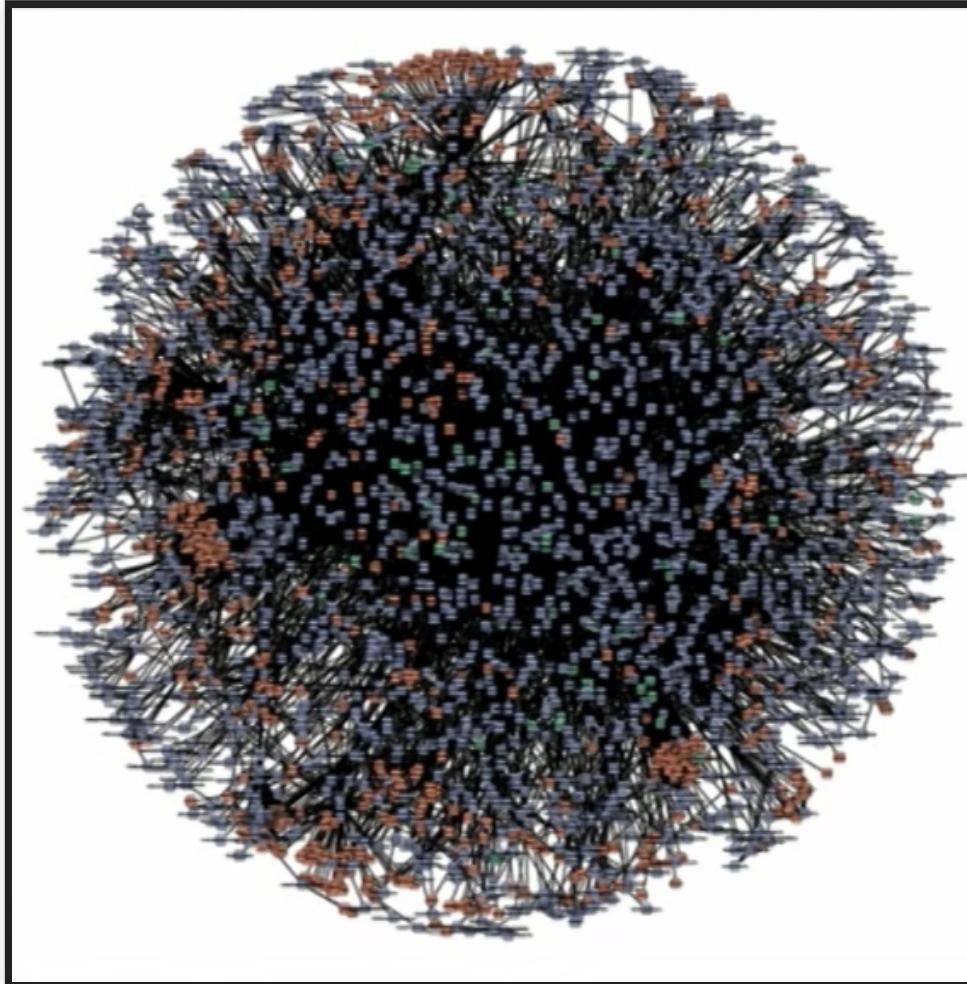
<https://www.youtube.com/watch?v=CZ3wluvmHeM>

# BEISPIEL: PAYPAL



<https://www.slideshare.net/deepaknadi/developerweek-2016-evolution-of-the-paypal-platform-journey-to-apis-microservices>

# BEISPIEL: AMAZON



Service-Oriented  
Architecture (SOA)

Single-purpose

Connected through APIs

Highly decoupled

“Microservices”

AWS re:Invent 2015: [DevOps at Amazon: A Look at Our Tools and Processes \(DVO202\)](#)

## 3.4 REST VS GRPC



*Wie könnte Kommunikation zwischen Diensten aussehen?*



# REST-API MIT HTTP

*Representational State Transfer (REST) und geht zurück auf die Dissertation von Roy Fielding aus dem Jahr 2000*

- Eindeutige Identifikation von Ressourcen (=Fachobjekte) über URLs
- Ressourcen-Verknüpfung: Ressourcen sind mit anderen Ressourcen über URLs verknüpft (z.B. Bestellung hat Kunde)
- HTTP-Standardmethoden: Server werden über HTTP mit den Grundbefehlen GET, PUT, POST, DELETE aufgerufen
- Unterschiedliche Repräsentationen: Zum Datenaustausch können unterschiedliche Repräsentationen (Text, XML, JSON, HTML) genutzt werden
- Statuslose Kommunikation: Die Kommunikation mit dem HTTP-Server ist zustandslos (stateless http server), der Server ist damit sehr gut an steigende Nutzerzahlen anpassbar (Skalierbarkeit).



*Beispiele für Ressourcen:*

URI	Interpretation
<code>http://example.com/customers/1234</code>	Kunde mit ID 1234
<code>http://example.com/orders/2007/10/776654</code>	Bestellung vom Oktober 2007 mit ID 776654
<code>http://example.com/products/4554</code>	Produkt mit ID 4554
<code>http://example.com/processes/sal-increase-234</code>	Prozess Gehaltserhöhung für Quartal 2, 3 und 4

## *Standardmethoden*

Operationen eines REST-Service beziehen sich auf eine Ressource und können mit den Mitteln, die z.B. HTTP bietet, aufgerufen.

Die typischen HTTP-Methoden GET, POST, PUT und DELETE haben in diesem Zusammenhang eine Standard-Interpretation:

<b>Methode</b>	<b>Interpretation</b>
GET	hole Information zu einer Ressource
POST	erzeuge Ressource zu übergebenen Werten
PUT	aktualisiere Ressource bzw. erzeuge Sub-Ressource oder füge Sie an
DELETE	Löschen einer Ressource



### *Beispiel mit Spring-Boot*

Nutzen Sie das bereitgestellte Projekt-Archiv `spring-rest.zip`, entpacken Sie dieses und importieren Sie den resultierenden Ordner es als "Existing Maven Project".



# VERSIONIERUNG VON REST-APIS

*Wie lassen sich unterschiedliche Versionen einer API verwalten?*

Durch die Weiterentwicklung von Diensten können sich Schnittstellen verändern. Es kann daher notwendig sein, sich Gedanken über die Unterstützung verschiedener Versionen von APIs zu machen. Im Fall von REST over HTTP lassen sich vier Strategien identifizieren: Path-, Query-, Header- oder Content-Type-basiert.



## Path- oder URL-basierte Versionierung

Über Bestandteile des Pfads in der URL wird die Version des gewünschten Endpunkts identifiziert.

```
@RestController
public class RestControllerWithVersion {
    @GetMapping("/api/v1/something/{id}")
    public Product doSomethingV1(@PathVariable String id) { /* ... */ }

    @GetMapping("/api/v2/something/{id}")
    public Product doSomethingV2(@PathVariable String id) { /* ... */ }
}
```

## Header-basierte Versionierung

Versionierung erfolgt über einen Custom-Header in der HTTP-Anfrage.

```
@RestController
public class RestControllerWithVersion {
    @GetMapping("/api/something/{id}", headers = "MY-API-VERSION=1")
    public Product doSomethingV1(@PathVariable String id) { /* ... */ }

    @GetMapping("/api/something/{id}", headers = "MY-API-VERSION=2")
    public Product doSomethingV2(@PathVariable String id) { /* ... */ }
}
```

## Query-basierte Versionierung

Versionierung erfolgt über den Query-Teil einer URI.

```
@RestController
public class RestControllerWithVersion {
    @GetMapping("/api/something/{id}", params = "version=v1")
    public Product doSomethingV1(@PathVariable String id) { /* ... */ }

    @GetMapping("/api/something/{id}", params = "version=v2")
    public Product doSomethingV2(@PathVariable String id) { /* ... */ }
}
```



## Content-Type-basierte Versionierung

Versionierung erfolgt über den gewünschten Content-Type der Antwort, was in der Anfrage über den Accept-Header-Parameter angegeben wird.

```
@RestController
public class RestControllerWithVersion {
    @GetMapping("/api/something/{id}", produces = "application/vnd.product.a
    public Product doSomethingV1(@PathVariable String id) { /* ... */ }

    @GetMapping("/api/something/{id}", produces = "application/vnd.product.a
    public Product doSomethingV2(@PathVariable String id) { /* ... */ }
}
```

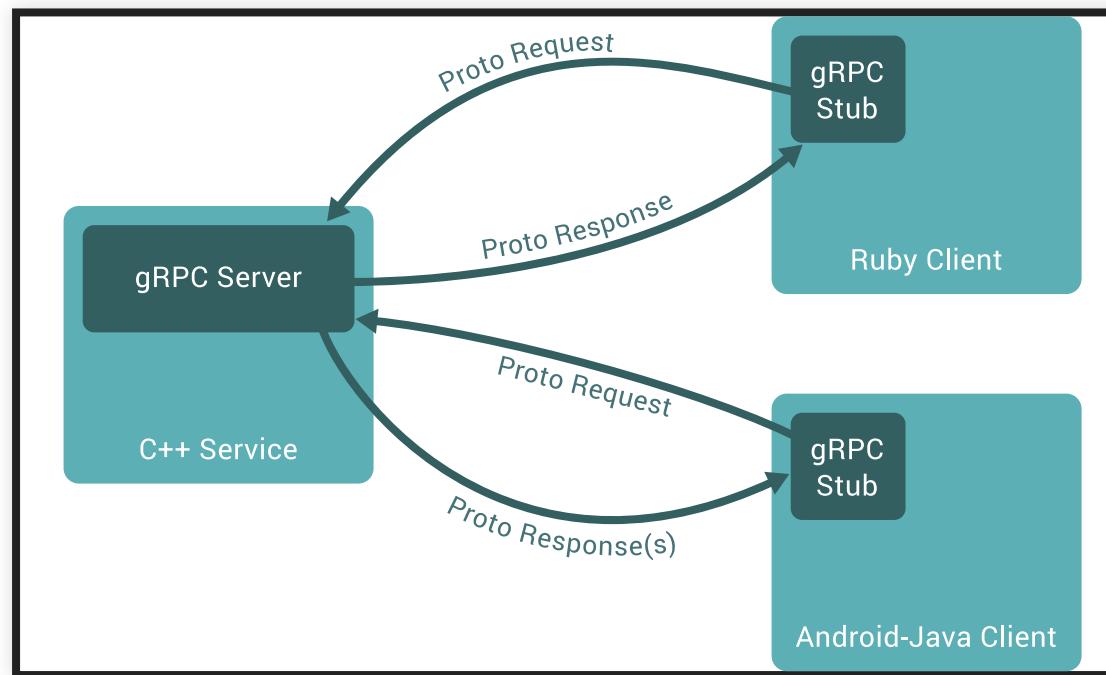


# REMOTE PROCEDURE CALL

- Innerhalb eines Prozesses können Funktionen (oder Methoden) direkt aufgerufen werden
- Sind Anwendungen verteilt ist eine Interprozesskommunikation notwendig
- Der Aufruf einer Funktion erfordert die Abbildung des Aufrufs auf ein geeignetes Kommunikationsprotokoll und entsprechende Architektur-Entscheidungen um innerhalb der Anwendung zum Beispiel über ein Client-Server-Modell Anfragen zu empfangen und auf die entsprechenden Funktionen abzubilden
- RPC-Ansätze lassen sich auch mit REST-Werkzeugen kombinieren, ist aber mit entsprechender Vorsicht zu benutzen
- Weitere Technologien: RMI in Java, SOAP und CORBA

# gRPC

*gRPC ist ein modernes Open-Source-Hochleistungs-Framework für Remote Procedure Call (RPC)*



Source: [Introduction to gRPC](#)



- gRPC ist ursprünglich von Google entwickelt wurden
- Idee war eine *general-purpose* RPC Schnittstelle für die Micro-Service Kommunikation zu entwickeln
- gRPC verwendet Protocol Buffers für die Serialisierung von Daten
- Es werden Deskriptoren zur Beschreibung der Funktionen genutzt und aus diesen die Notwendigen Werkzeuge zur Serialisierung von Daten als auch Stubs für Server und Client generiert
- Es gibt Code-Generatoren für verschiedene Sprachen: C#, C++, Dart, Go, Java, Kotlin, Node, ...

## *Beispiel für die Beschreibung eines Datenpaketes*

```
message Person {  
    string name = 1;  
    int32 id = 2;  
    bool has_ponycopter = 3;  
}
```

- Definiert eine Nachricht **Person**
- Sowie die mit der Nachricht verbundenen Attribute



*Mit dem selben Deskriptor lassen sich auch Dienste beschreiben*

```
service MyService {
    rpc SayHello (HelloRequest) returns (HelloReply) {
    }
}
message HelloRequest {
    string name = 1;
}
message HelloReply {
    string message = 1;
}
```

- Zwei Nachrichten `HelloRequest` und `HelloReply`
- Sowie ein Dienst ...
- ... mit einer Methode, welcher Parameter und Rückgabewert definiert



### *Beispiel mit gRPC*

Nutzen Sie das bereitgestellte Projekt-Archiv simple-grpc.zip, entpacken Sie dieses und importieren Sie den resultierenden Ordner es als "Existing Maven Project".



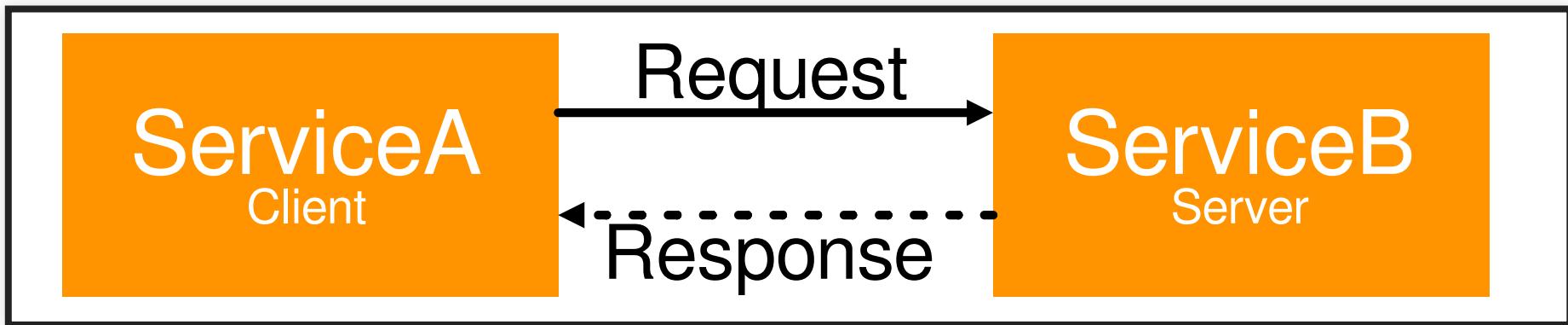
### *Beispiel mit Sprint-Boot*

Nutzen Sie das bereitgestellte Projekt-Archiv `spring-grpc.zip`, entpacken Sie dieses und importieren Sie den resultierenden Ordner es als "Existing Maven Project".

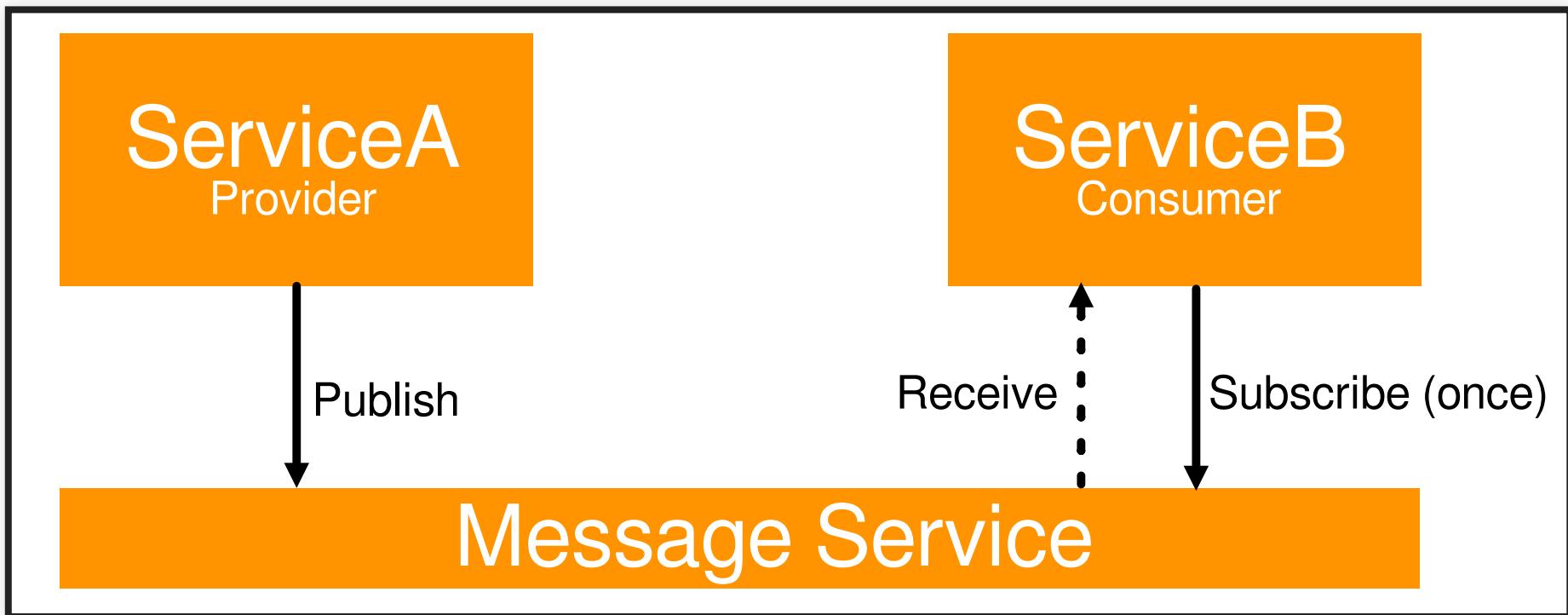
## 3.5 EVENT-DRIVEN ARCHITECTURE

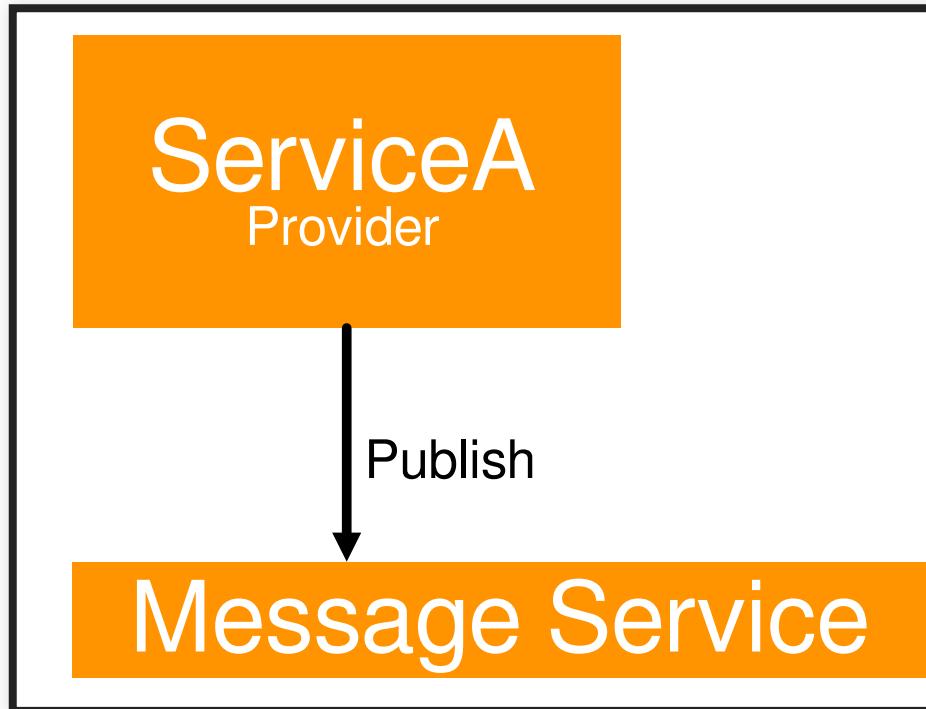
*Kommunizieren Micro-Services nur per Anfragen zu anderen Diensten, z.B. mittels  
HTTP?*

*Traditionelles Request/Response-Model*



*Alternative: Ereignisbasierte Ansätze*



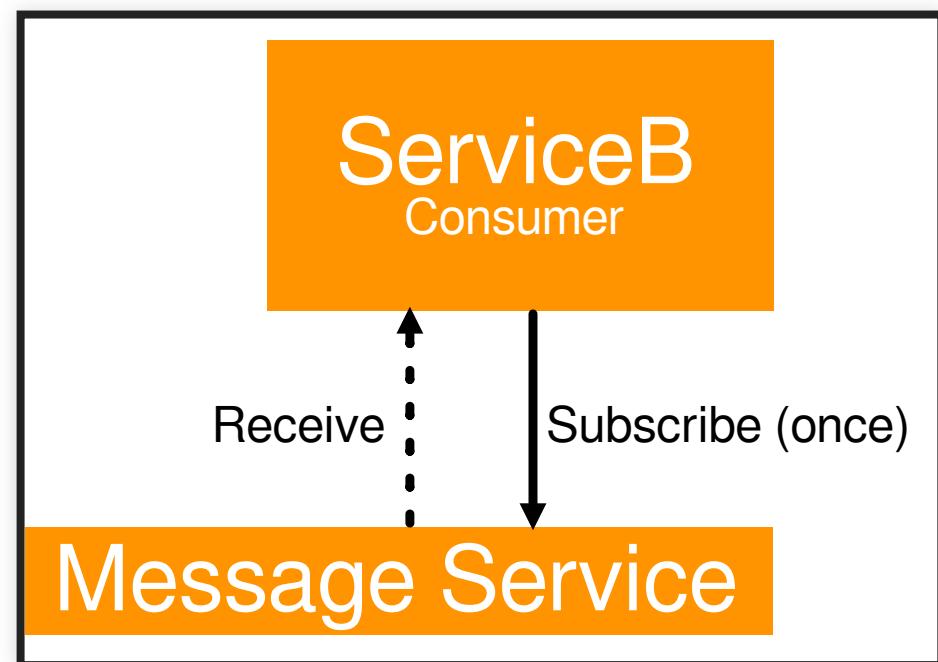


Neue Ereignisse (z.B. "Neue Bestellung eingegangen" oder "Neuer Artikel wurde verfasst"), werden durch den entsprechenden veröffentlichten Dienst abgesendet, der sich zum Beispiel um diesen Teil des Bounded Contexts kümmert (vgl. Bestellungs-Service oder Artikel-Service).

Gemeldet werden Ereignisse gegenüber einem Nachrichten-Dienst oder -Middleware (z.B. MQTT).

Konsumierende Dienste die an bestimmten Ereignissen interessiert sind, müssen sich am Nachrichten-Dienst registrieren und deren Interesse für bestimmte Ereignisse kund tun.

Anschließend wird der registrierte Dienst über Ereignisse informiert. Die Information kann der Dienst nutzen um weitere Prozesse, die mit diesem Ereignis in Verbindung stehen, anzustoßen. Dabei kann man zwischen *Queues* und *Topics* unterscheiden.





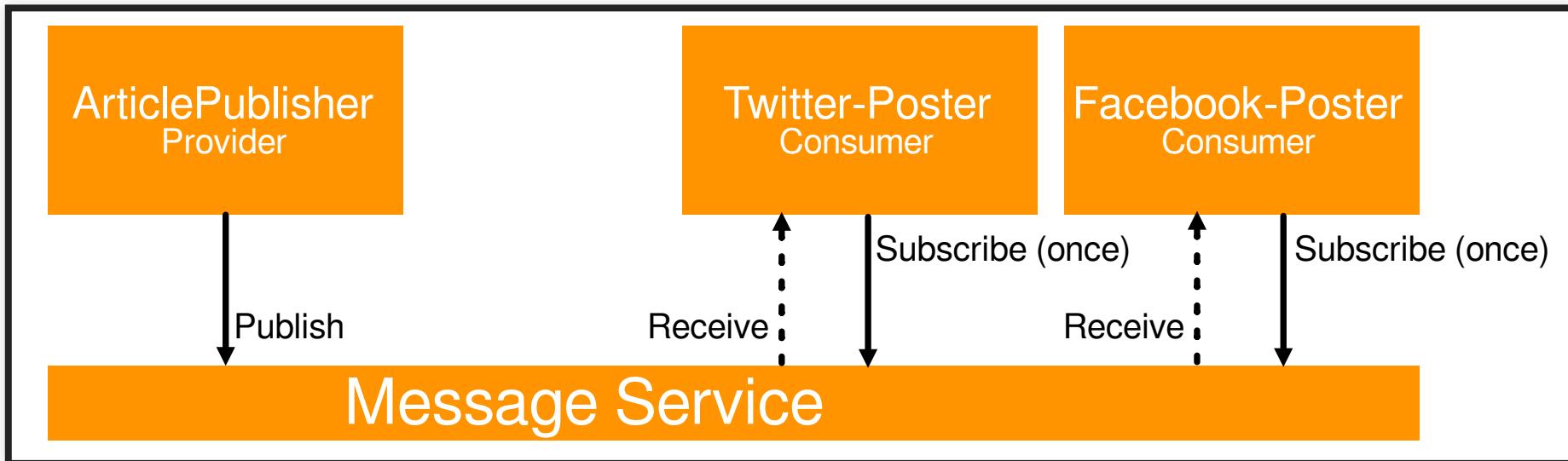
## Queue

Ereignisse werden persistiert und solang wie möglich aufgehoben, bis die Ereignisse durch einen Dienst konsumiert wurden.

## Topic

Eine Art Blackboard oder Forum in dem alle zuhörenden über Ereignisse informiert werden.

## Beispiel



Mithilfe von Nachrichten-Infrastrukturen und dem Kommunizieren werden die Dienste (ArticlePublisher, Twitter-Poster und Facebook-Poster) entkoppelt.



## Hinweise

- Implizit entsteht eine Kopplung der konsumierenden Dienste an die Nachrichtenformate und Kanäle der veröffentlichten Dienste, hier ist es wichtig geeignete Definitionsformate und Quellen der Wahrheit (Source of Truth) zu definieren (Wildwuchs vermeiden)
- Fehlersuche in ereignisbasierten Infrastrukturen ist schwierig: wo ist wann durch welches Ereignis was passiert? Nachvollziehbarkeit von Ereignisketten muss in der Regel auf Anwendungsebene sichergestellt werden (z.B. durch identifizierende Merkmale)



## MQTT MIT MOSQUITTO UND JAVA

Nutzen Sie das bereitgestellte Projekt-Archiv mqtt.zip, entpacken Sie dieses und importieren Sie den resultierenden Ordner es als "Existing Maven Project".



## MQTT IN SPRING

Nutzen Sie das bereitgestellte Projekt-Archiv `spring-mqtt.zip`, entpacken Sie dieses und importieren Sie den resultierenden Ordner es als "Existing Maven Project".

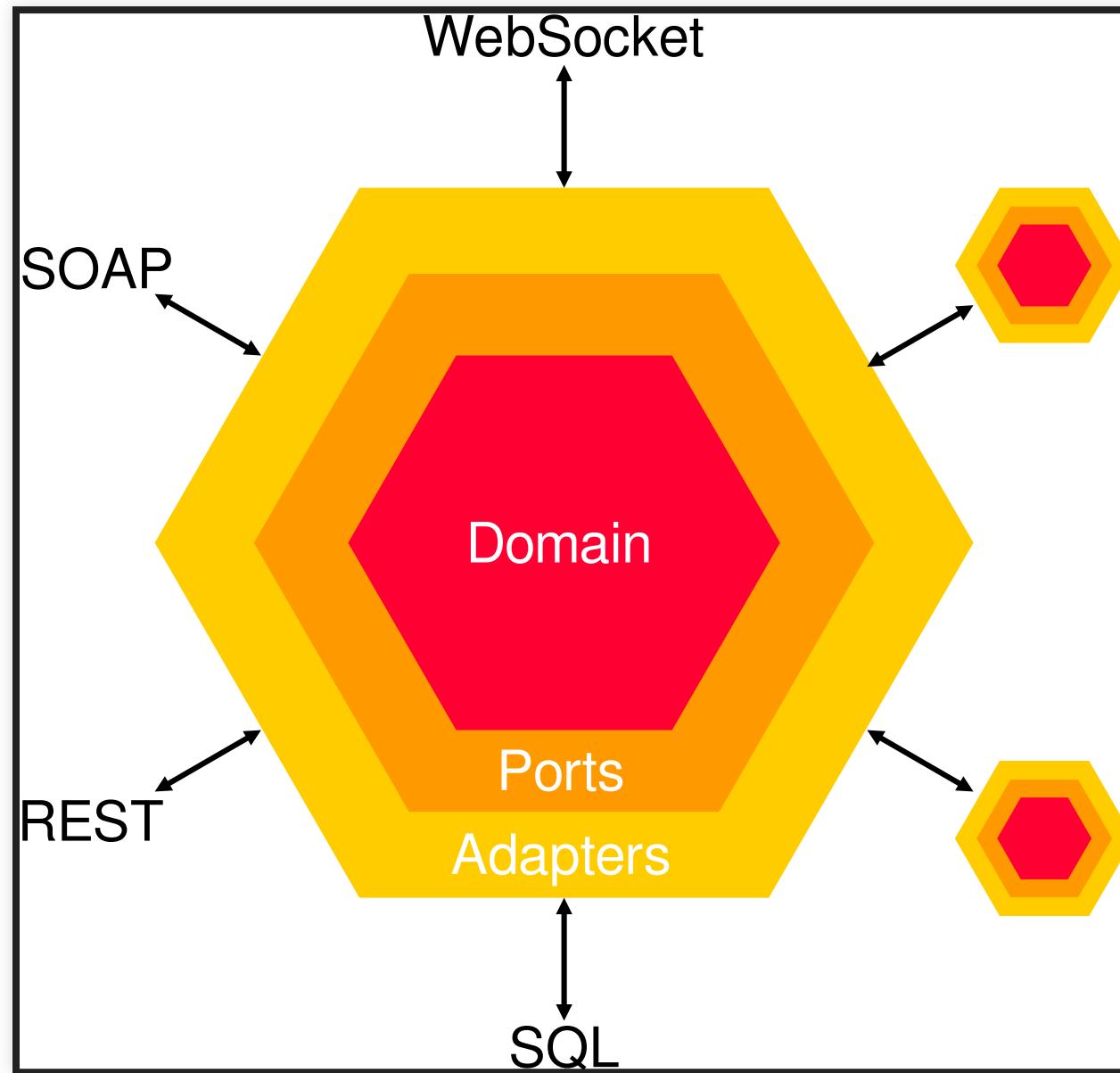
## 3.6 HEXAGONALE ARCHITEKTUR



*Wie könnte ein einzelner Micro-Service aufgebaut sein?*

## *Typische Herausforderungen beim Umsetzen von Anwendungen, auch bei Micro-Services*

- Gab es einen längerfristigen Entwicklungsplan oder war es bei der Entwicklung eher ein Short-Run?
- Lässt sich die Anwendung warten?
- Wie hoch sind die Technical Depts?
- Wie gehen wir mit vielfältigen Schnittstellentechnologien und Versionen um





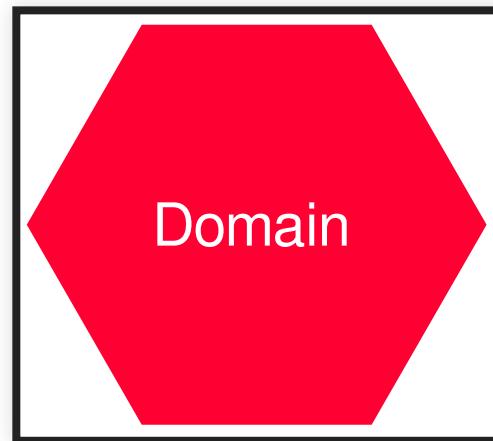


*Mit dem Ziel niedriger Technical Depts erhöhen hexagonale Architekturen die Wartbarkeit und erlauben eine langfristige Weiterentwicklung von Anwendungen*

Verschiedene Wege der Kommunikation einer Anwendungen sind entlang der äußeren Seiten beispielhaft dargestellt: REST, SOAP, WebSocket, SQL oder zu anderen (hexagonalen) Anwendungen.

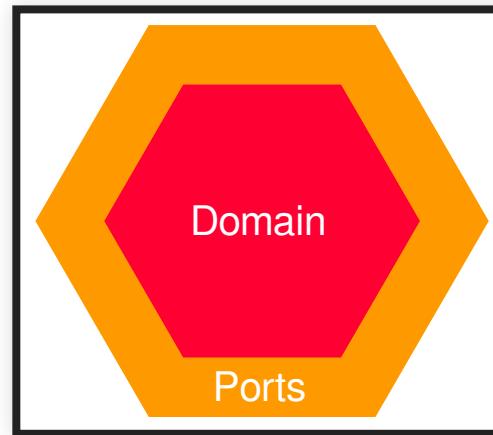
Im Inneren findet sich eine Schichtenarchitektur, ausgehend von der Domain (Bezug zu Domain Driven Design) hin zum Framework mit den verschiedenen Möglichkeiten zur Kommunikation.

## Domain



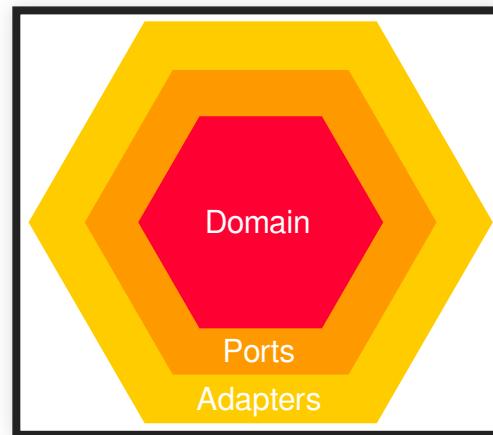
Die Domain ist das zentrale Element, welches die Geschäftslogik und entsprechende Regeln abbildet. Die Logik ist ohne Bezug zur (Kommunikations-)Technologie umgesetzt und bietet die verschiedenen Domain-Objekte, sowie Methoden zur Interaktion mit dem Model und kommuniziert über z.B. Fehler falls Regeln nicht passen.

## Ports



Der Domain (häufig auch Teil der Domain) anschließend sind eingehende und ausgehende Ports. Diese erlauben es von außen auf die Domain zuzugreifen oder der Domain nach außen zu kommunizieren (vgl. REST vs. Datenbank).

## Adapters



Außerhalb platzieren sich Adapter für mit entsprechendem Technologiebezug. Zum Beispiel werden REST-Endpunkte umgesetzt mit den entsprechenden Frameworks oder die Datenbankanbindung realisiert.



## BEISPIEL: DIENST MIT SPRING, GRPCS UND MQTT

Nutzen Sie das bereitgestellte Projekt-Archiv hexa-posts.zip, entpacken Sie dieses und importieren Sie den resultierenden Ordner es als "Existing Maven Project".

## Domain des Beispiels

*PostService*

```
+createPost(title: String, content: String, userRef: UserReference) : Post
+hidePost(id: UUID): boolean
+findPost(id: UUID): Post;
+listNewestPost(): Post[*]
+countPosts(): long
+listCommentsForPost(postRef: UUID): Comment[*]
+createCommentForPost(postRef: UUID, content: String): Comment
```

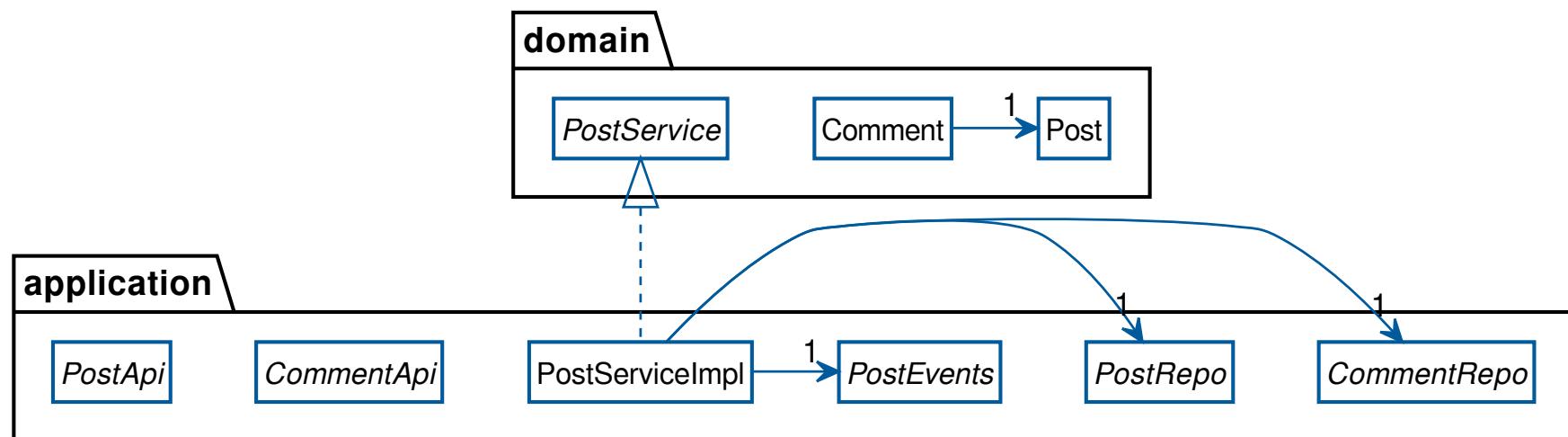




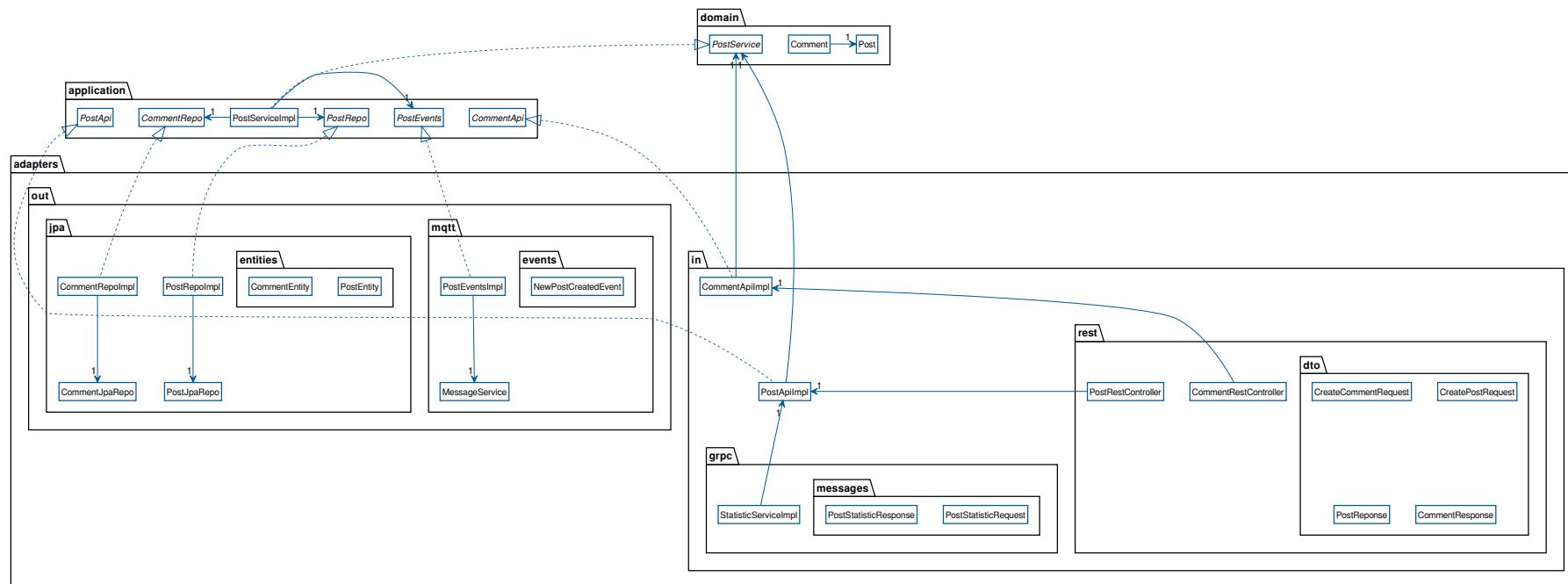
## *Erklärung*

- Die Domain umfasst die Fachobjekte des Dienstes
- Die Klassen der Domain sind *möglichst* unabhängig von spezifischen Technologien, z.B. JPA-Annotationen wären hier fehl am Platz, im Beispiel wird Lombok verwendet, um Getter/Setter zu generieren
- Innerhalb werden ein oder mehrere Schnittstellen mit den notwendigen Funktionalitäten definiert, die das Modell anbietet. Hier finden sich Synergien zum Domain Driven Design - weniger CRUD mehr fachliche Funktionalitäten

## Ports des Beispiels, sowie Implementierung des Dienstes



## Ports des Beispiels, sowie Implementierung des Dienstes

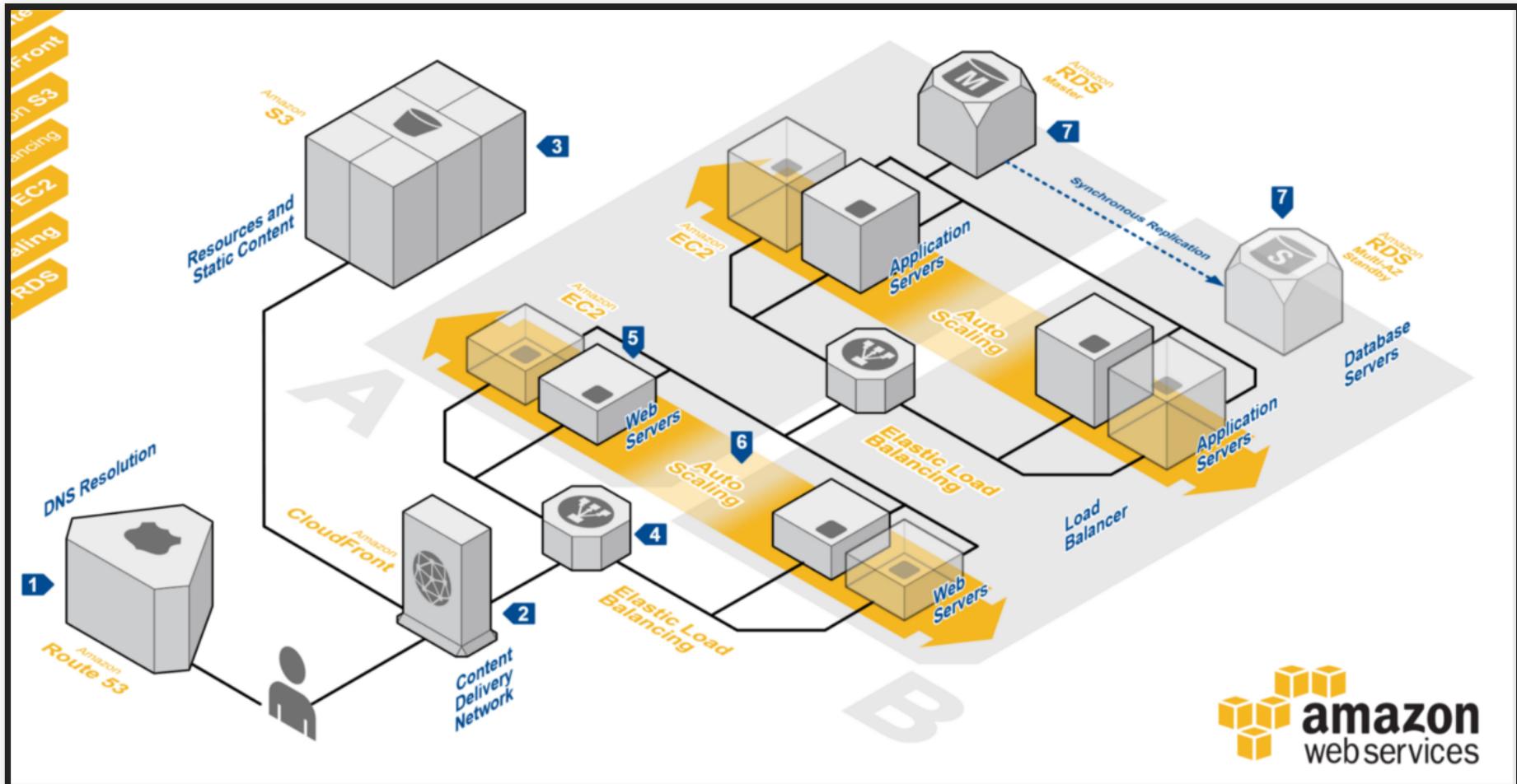


## 3.7 REFERENZARCHITEKTUREN IN DER CLOUD



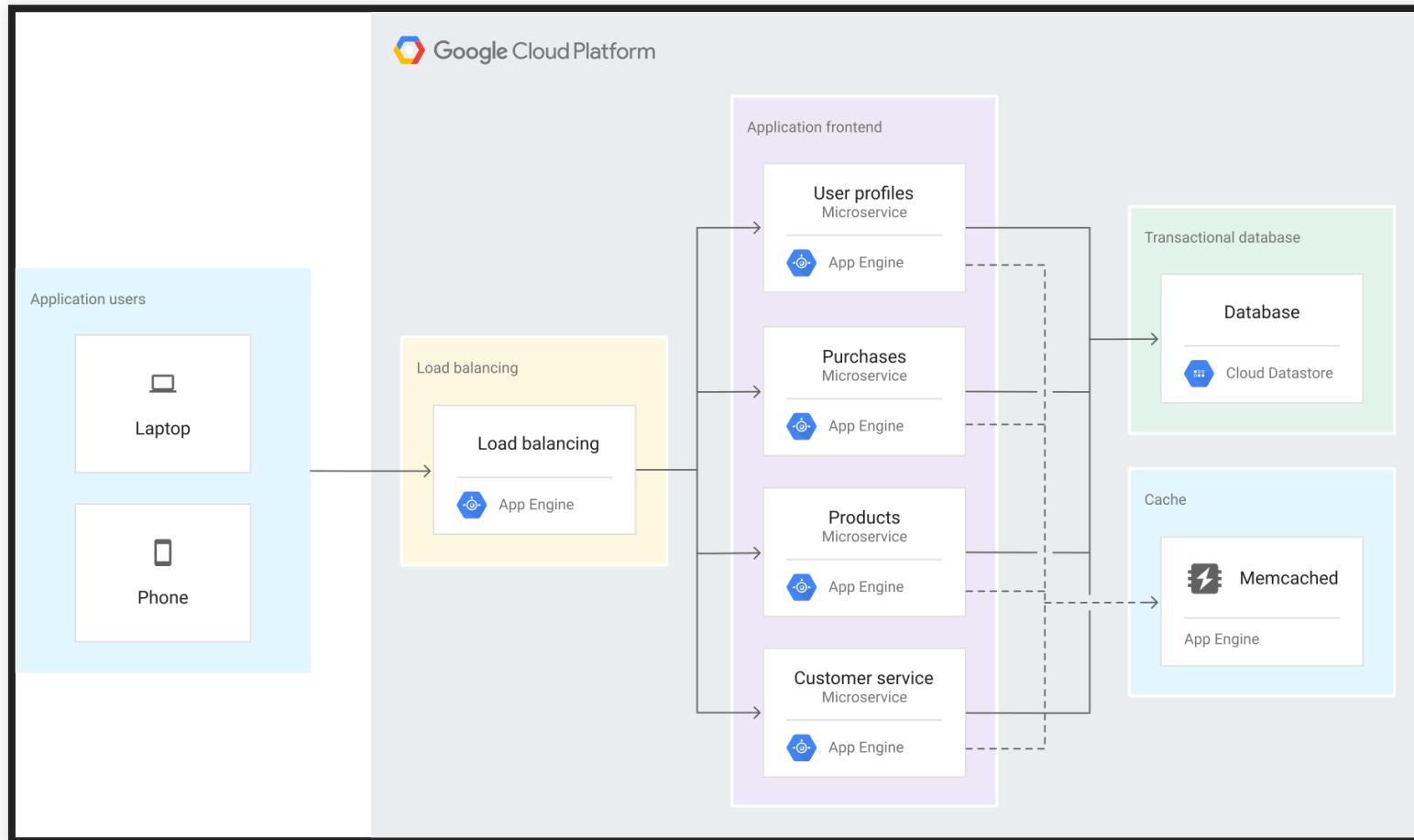
*Wie bilden sich die Architekturen in einer Cloud-Umgebung ab?*

# AWS WEBHOSTING



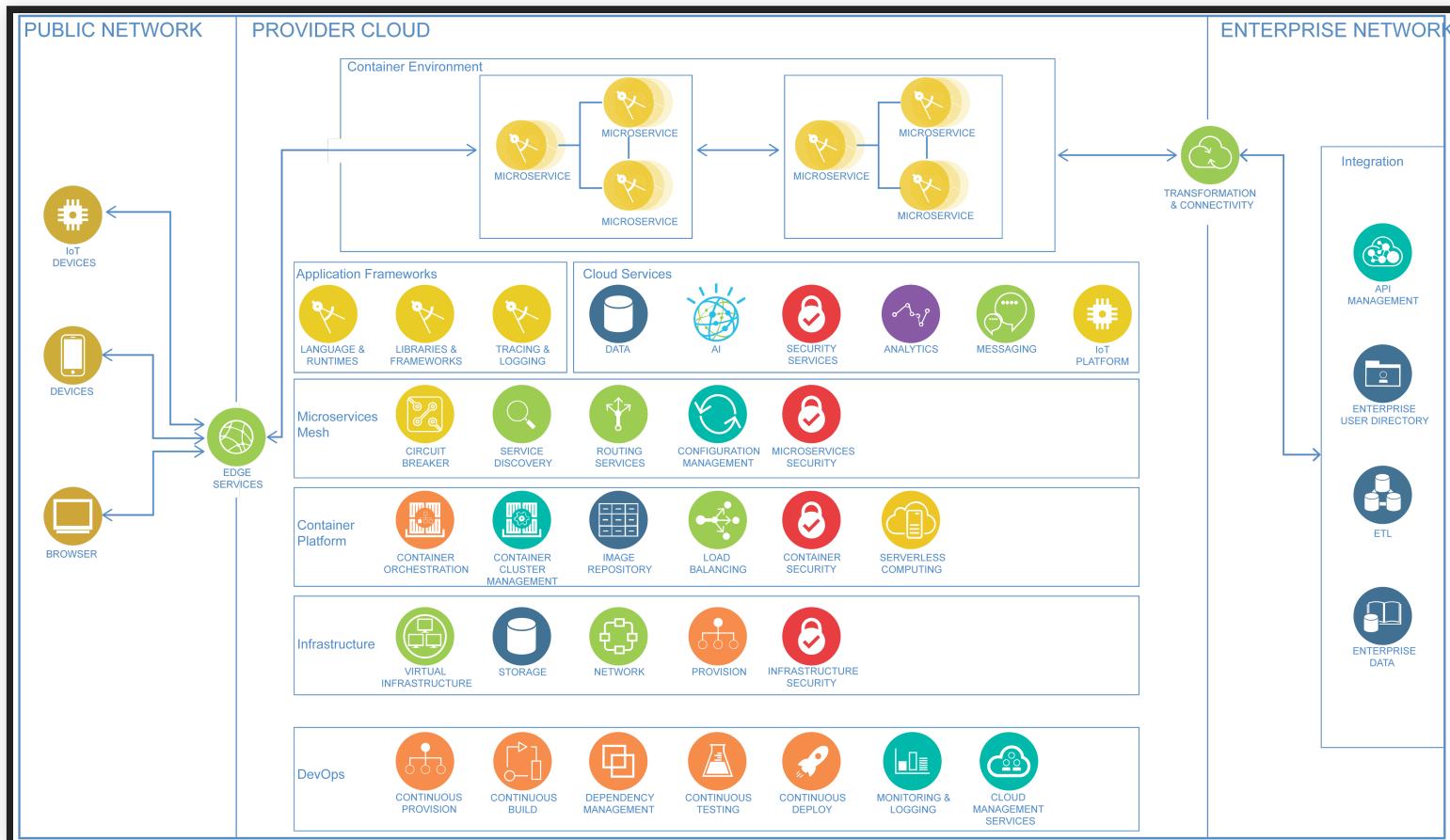
<https://docs.aws.amazon.com/whitepapers/latest/web-application-hosting-best-practices/web-application-hosting-best-practices.pdf>

# GOOLGE MICRO-SERVICE



<https://cloud.google.com/architecture/microservices-with-app-engine>

# IBM MICRO-SERVICE



<https://www.ibm.com/cloud/architecture/architectures/microservices/reference-architecture/>

## 3.8 REFERENZEN



- Eberhard Wolff: Das Microservice Praxisbuch - Grundlagen, Konzepte und Rezepte
- Eberhard Wolff: [Microservices - Flexible Software Architecture](#)
- Irakli Nadareishvili, Ronnie Mitra, Matt McLarty and Mike Amundsen: Microservice Architecture - Aligning Principles, Practices, and Culture
- Vaughn Vernon: Domain-Driven Design
- David Sprott and Lawrence Wilkes: Understanding Service-Oriented Architecture. CBDI Forum, 2004.