MINUTEN

CLOUD-NATIVE

CLOUD NATIVE COMPUTING

Unit:

Microservices

(2) Oft übersehene Randbedingungen

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

KAPITEL 12

Microservice Architekturen

12.1 Eigenschaften von Microservices

12.2 Integrationsmuster für Microservices

- Datenbank-basierte Integration
- gRPC, REST
- API Versioning
- Event-Driven Architectures

12.3 Architekturelle Sicherheit

- Circuit-Breaker, Bulkhead
- Idempotente API-Operationen

12.4 Skalierung von Microservices

- Load Balancing
- Messaging
- Scaling for Reads/Writes
- Command Query Responsibility Segregation (CQRS)
- Caching

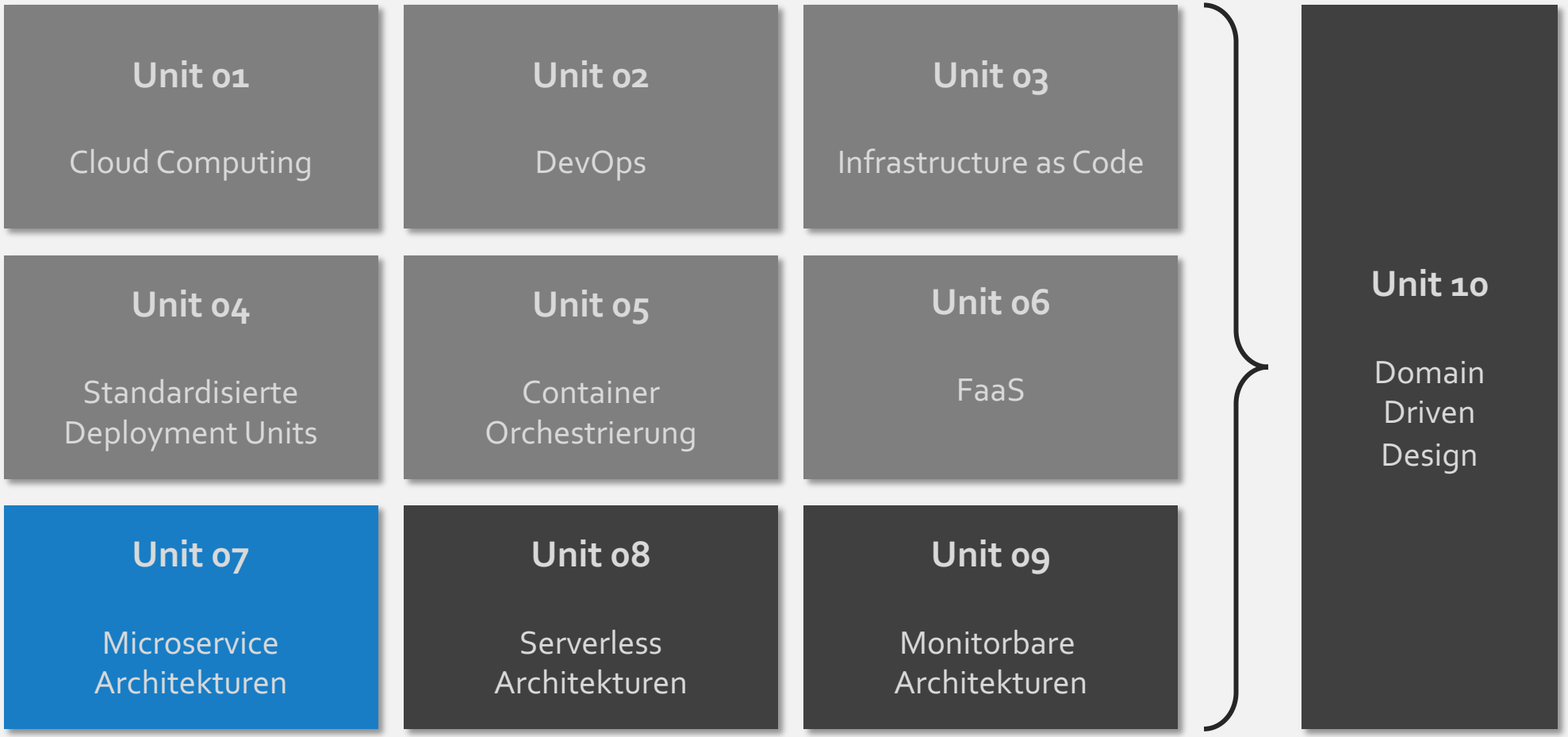
12.5 Prinzipien zur Entwicklung von Microservices

12.6 Serverless-Architekturen

- Architekturelle Konsequenzen von Serverless-Limitierungen
- Das API-Gateway Pattern
- Abgrenzung zu Microservices

INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



INHALTE

CLOUD NATIVE COMPUTING

Microservices

- Was ist das für ein Architekturstil?
- Vor- und Nachteile von Microservices

Randbedingungen

- Lose Kopplung und hohe Kohäsion
- Bounded Context und Domain Driven Design
- Conways Law
- Service Ownership und Team-Strukturen

Integration

- Shared Databases, Sync vs. Async, RPC
- REST, Services as State Machines (HATEOS)
- Versioning

Skalierung von Microservices

- Failure is Everywhere
- Architectural Safety Measures
- Scaling
- Caching

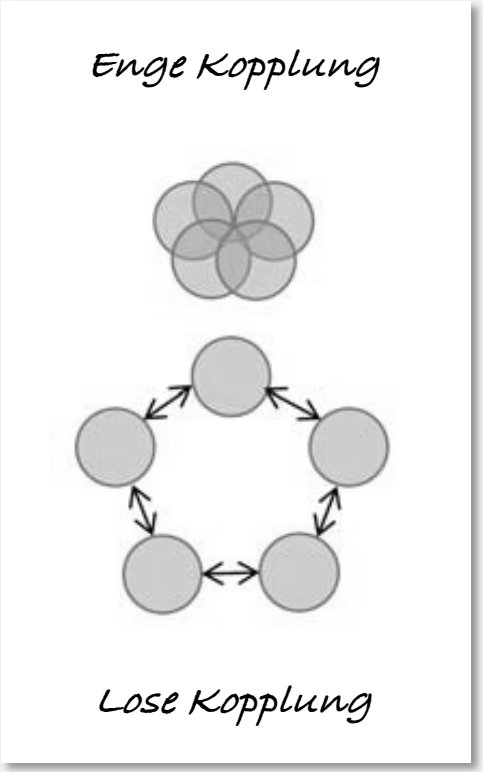
7 Prinzipien von Microservices

- Model Around Business Concepts
- Adopt a Culture of Automation
- Hide Internal Implementation Details
- Decentralize All the Things
- Independently Deployable
- Isolate Failure
- Highly Observable

WAS IST EIN „GUTER" (MICRO-)SERVICE?

Lose Kopplung

- Microservice-Ansatz strebt **unabhängige Austauschbarkeit** von Komponenten ohne Beeinflussung anderer Teile des Systems an
- Änderungen an lose gekoppelten Services erfordern weniger Änderungen an interagierenden Services
- Enge und lose Kopplung entsteht durch technische **Integrationsstile**, die Services enger oder loser aneinander binden
  - Lose gekoppelte Services sollten so wenig wie möglich über andere Services wissen
  - „Gesprächigere Kommunikation" führt oft zu engerer Kopplung und reduziert unabhängige Austauschbarkeit von Komponenten
- Begrenzung der verschiedenen Arten von Requests zwischen Services reduziert oft sowohl Leistungsprobleme als auch enge Kopplung



*Die Begrifflichkeiten **lose Kopplung** und **hohe Kohäsion** werden insbesondere in der OOAD genutzt, um „gute" (also vor allem flexibel wiederverwandbare) Domnänenmodelle zu entwickeln.*

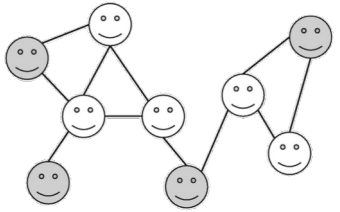
Doch diese Begriffe sind nicht dem OOAD vorbehalten und können auch für das Design von Services herangezogen werden.

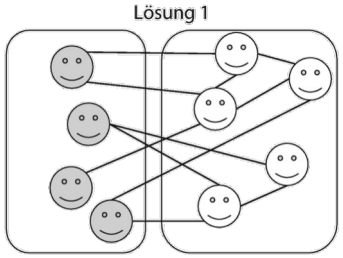
Quelle: Sam Newman, Building Microservices, O'Reilly 2015

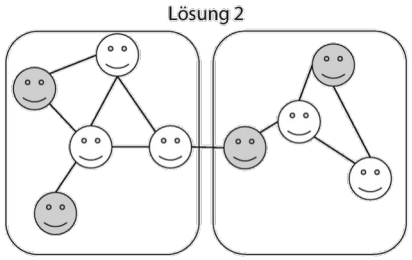
WAS IST EIN „GUTER" (MICRO-)SERVICE?

Hohe Kohäsion

- **Kohäsion (innerer Zusammenhalt)** beschreibt in der objektorientierten Programmierung die Abbildung logischer Aufgaben oder Einheiten durch eine Programmeinheit
- Ein System mit starker Kohäsion hat idealerweise Service-Komponenten, die nur für eine wohldefinierte Aufgabe zuständig sind **(Single-Responsibility)**
- Zusammengehörige Aufgaben sollten zusammen deployt und in Deployment Units voneinander isoliert werden, um schnell Änderungen ausrollen zu können
- Änderungen an vielen verschiedenen Orten im Quelltext sind langsamer und riskanter als Änderungen an einem Ort
- Architekturen mit einer hohen Kohäsion und loser Kopplung sind also eine Voraussetzung für agile DevOps-Ansätze mit vielen kleinen, schnellen und risikoarmen Updates.





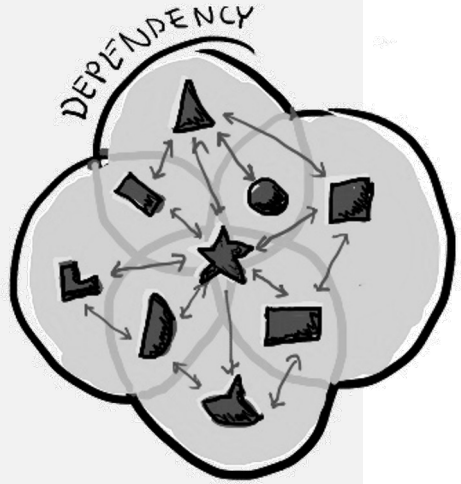


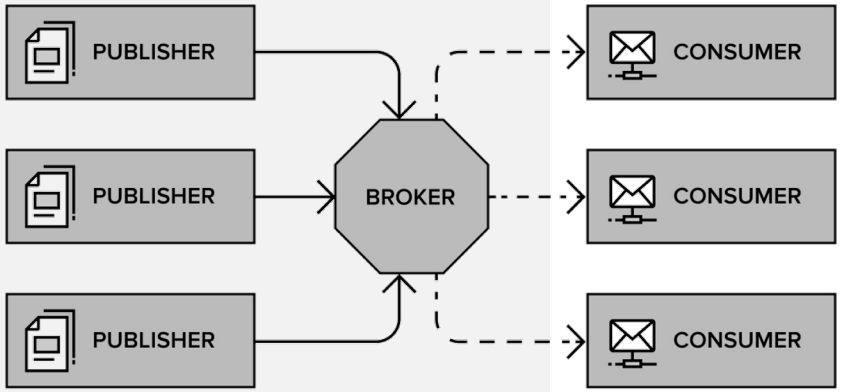
Quelle: Sam Newman, Building Microservices, O'Reilly 2015

WAS IST EIN „GUTER" (MICRO-)SERVICE?

Methoden um Kopplungen zu reduzieren

- Verbesserung der **losen Kopplung von Schnittstellen** durch Bereitstellung von Daten in einem Standardformat wie XML oder JSON
- Verbesserung der **losen Kopplung zwischen Programmkomponenten** durch Verwendung von Standarddatentypen in Parametern, um die Notwendigkeit von Kenntnissen benutzerdefinierter oder gar systemspezifischer Datendefinition zu vermeiden
- Verbesserung der **losen Kopplung von Services** durch Verwendung von Schlüsseldaten (ID) in der Service-to-Service Kommunikation, um Services von Dateninstanzen entkoppeln zu können. Aber auch Reaktives Systemdesign mittels Event-driven Architectures (Integrationsstil)



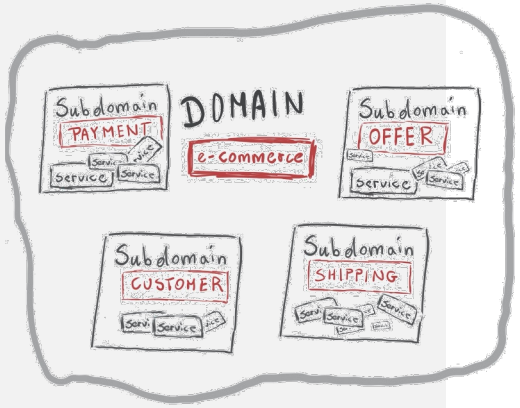


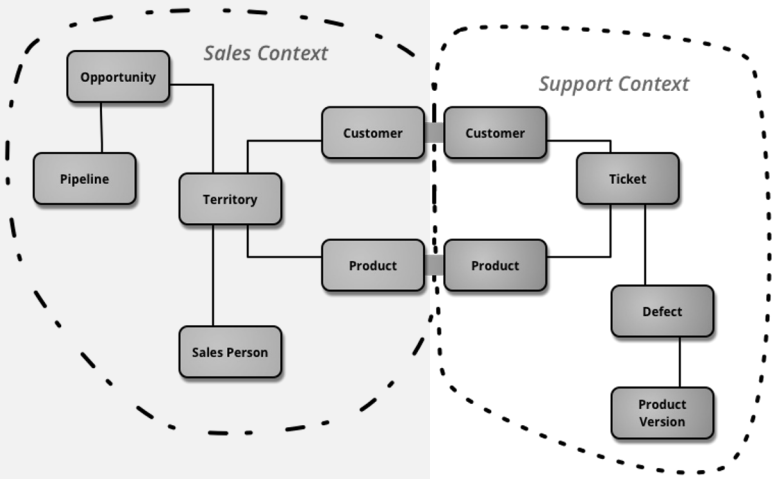
Quelle: Sam Newman, Building Microservices, O'Reilly 2015

DOMAIN DRIVEN DESIGN (DDD)

Wie kommt man zu hoher Kohäsion? Z.B. mit Bounded Contexten

- Domain-Driven Design (DDD) zielt darauf ab, Software basierend auf Modellen der zugrunde liegenden **Fachlichkeit** (Domäne) zu entwerfen.
- Ein Modell dient als konzeptionelle Grundlage für das Design der Software und unterstützt die Kommunikation zwischen Softwareentwicklern und Domänen-Experten.
- Ein Modell muss vereinheitlicht werden, um effektiv nutzbar zu sein, und darf keine Widersprüche enthalten.
- Allerdings ist eine vollständige Vereinheitlichung eines Domänenmodells für ein großes System oft weder durchführbar noch kostengünstig.
- DDD gliedert daher ein großes System in begrenzte Kontexte (Bounded Contexts), von denen jeder Kontext ein einheitliches Modell haben kann (Multiple Canonical Models).
- Jeder Bounded Context kann für sich abgeschlossen betrachtet werden (um unabhängig von andeen als **Microservice** realisiert zu werden).





Mehr zu DDD in Unit 10

Quelle: Sam Newman, Building Microservices, O'Reilly 2015, https://martinfowler.com/bliki/BoundedContext.html

CONWAY'S LAW

Und System Design

„Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.“

— Melvin E. Conway, 1968 (How do committees invent)

„ If you have four groups working on a compiler, you'll get a 4-pass compiler.“

„ If a group of N persons implements a COBOL compiler, there will be N–1 passes. Someone in the group has to be the manager.“

— Eric S. Raymond (Open Source Evangelist, 1996)


HOW DO COMMITTEES INVENT?

by MELVIN E. CONWAY

design organization criteria

Quelle: Sam Newman, Building Microservices, O'Reilly 2015, `https://martinfowler.com/bliki/BoundedContext.html`

CONWAY'S LAW

Und System Design

Wir erinnern uns:

- Das Bestreben von DevOps sind schnelle und unriskante Deployments.
- Architekturen mit einer hohen Kohäsion und loser Kopplung sind eine Voraussetzung für agile DevOps-Ansätze (viele kleine, schnelle, unriskante Updates).

Wir erfahren:

- Organisationsstrukturen prägen offenbar Architekturen

Wenn wir beides zusammen nehmen:

- Müsste man doch eigentlich Organisationsstrukturen schaffen, die Architekturen mit loser Kopplung und hoher Kohäsion erzeugen (oder zumindest wahrscheinlicher machen).

Kann das sein? Ist das nicht ein wenig extrem?

Quelle: Sam Newman, Building Microservices, O'Reilly 2015

IMPLIKATIONEN VON CONWAY'S LAW

DevOps + You build it, you run it => Service Ownership

- Paper von Nigel Bevan zu Usability-Problemen auf Webseiten (1997): Webseiten von Unternehmen spiegeln eher interne Anliegen als Benutzerbedürfnisse wider
- **Effekte des Conway-Gesetzes** wurden auch von Forschern der MIT und Harvard Business School festgestellt
  - Software lose gekoppelter Organisationen (z.B. Open Source Communities) oft modularer
  - Software stärker hierarchischer geführter Organisation (z.B. Microsoft) oft monolithischer
  - Interne Studien von Microsoft zu Windows Vista führten zu ähnlichen Erkenntnissen
- Netflix und Amazon: Entwicklungsabteilungen werden an gewünschten Software-Eigenschaften und nicht Management-Erfordernissen ausgerichtet:
  - **You build it, you run it** für Service Ownership und Verantwortung
  - **Two pizza principle** für obere Schranke von Teamgrößen

Prüfen Sie diese These einmal an der Website einer Hochschule oder Unternehmens

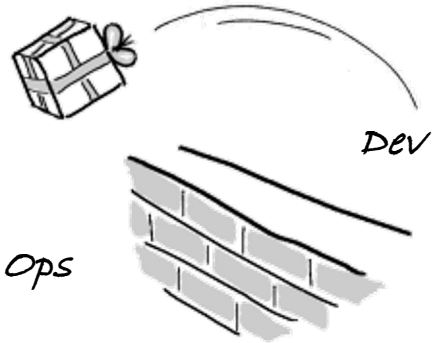
Amerikanische Pizzen natürlich ;-)

Quelle: Sam Newman, Building Microservices, O'Reilly 2015

CONWAY'S LAW

Service Ownership und Produktorientierung

- **Service Ownership** bedeutet, dass ein Entwicklerteam für Änderungen und Betrieb eines Services über dessen komplette **Lebensdauer** verantwortlich ist
- Teams haben oft auch Verantwortung für Anforderungserhebung, Erstellung, Bereitstellung und Wartung eines Services
- **Produktphilosophie** anstelle einer **Projektphilosophie**
- Anreiz für die Erstellung von einfach bereitzustellenden Services
- Vermeidung des *"Throw-over-wall"* Effekts, da Team selbst für den Betrieb verantwortlich ist
- Entscheidungen werden vom technisch versierten Team getroffen, was zu mehr **Autonomie** und aber auch mehr Verantwortung führt
- Führt zu erhöhter **Liefergeschwindigkeit**

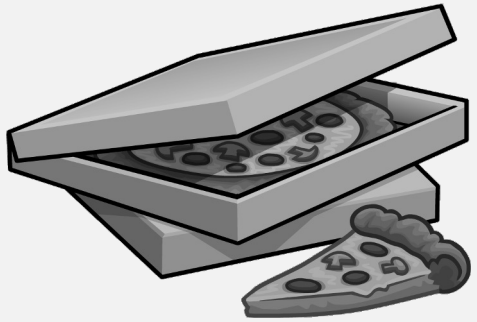


Service Ownership vermeidet den „throw-over-wall" Effekt oder die „It works on my machine"-Ausrede.

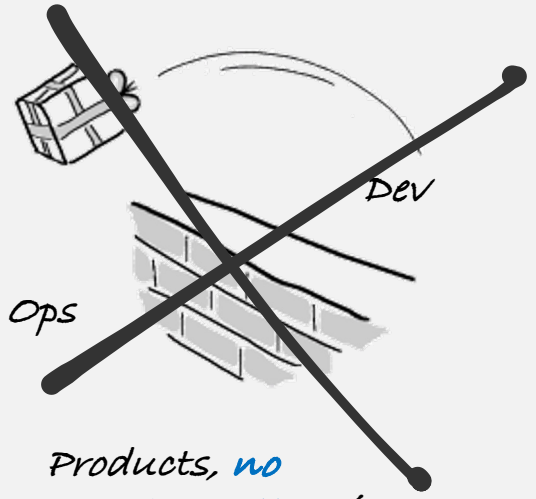
Quelle: Sam Newman, Building Microservices, O'Reilly 2015

CONWAY'S LAW

Merkregeln für das Microservice Team-Building



Two Pizza Teams



Products, no throw-over-the-wall projects

Wenn Conways Law gilt, fördern Sie mit diesen Regeln gute Microservice Architekturen.

INHALTE

CLOUD NATIVE COMPUTING

Microservices

- Was ist das für ein Architekturstil?
- Vor- und Nachteile von Microservices

Randbedingungen

- Lose Kopplung und hohe Kohäsion
- Bounded Context und Domain Driven Design
- Conways Law
- Service Ownership und Team-Strukturen

Integration

- Shared Databases, Sync vs. Async, RPC
- REST, Services as State Machines (HATEOS)
- Versioning

Skalierung von Microservices

- Failure is Everywhere
- Architectural Safety Measures
- Scaling
- Caching

7 Prinzipien von Microservices

- Model Around Business Concepts
- Adopt a Culture of Automation
- Hide Internal Implementation Details
- Decentralize All the Things
- Independently Deployable
- Isolate Failure
- Highly Observable

KONTAKT

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de