CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Microservices

(7) Die Sieben Prinzipien des Microservice Designs

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

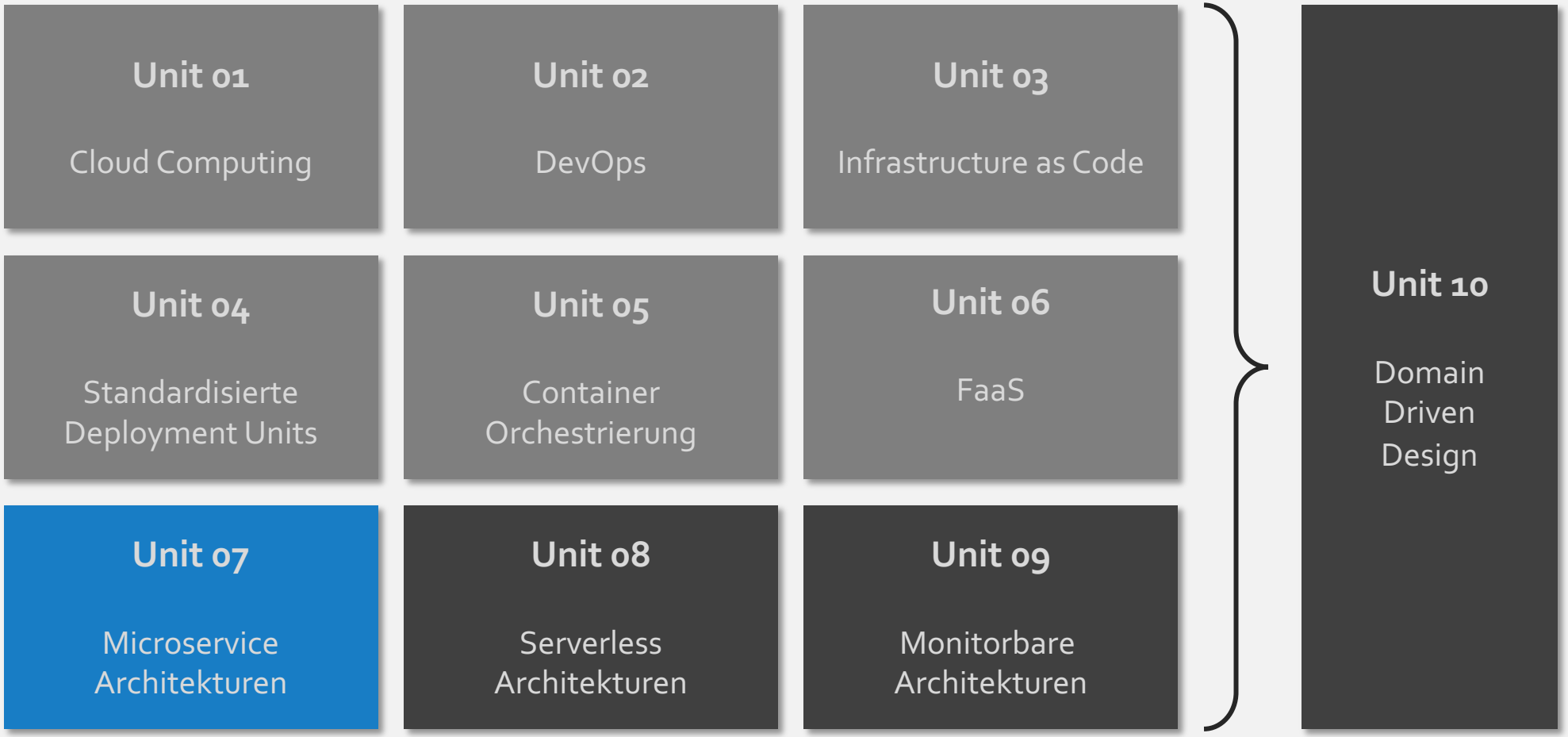
12.5 Prinzipien zur Entwicklung von Microservices

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

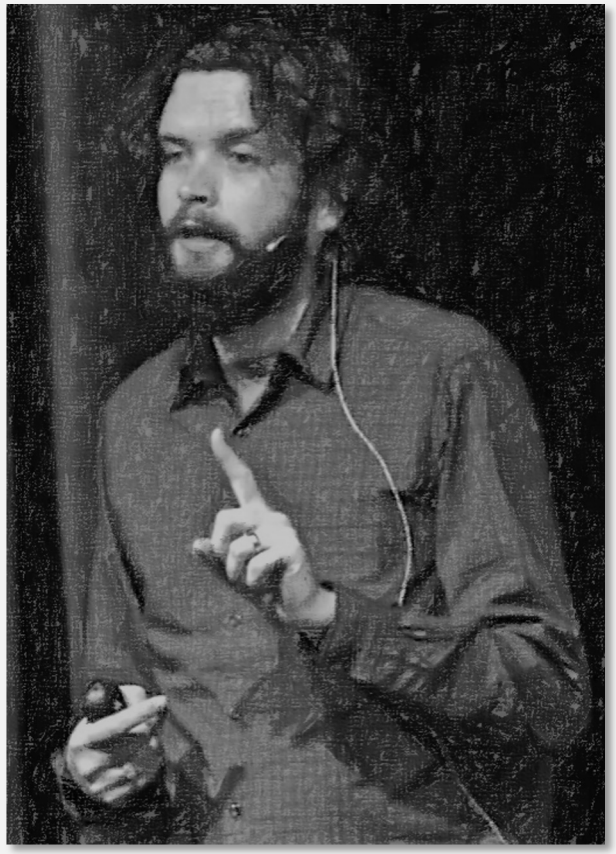
Skalierung von Microservices

- 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

7 PRINZIPIEN VON MICROSERVICES

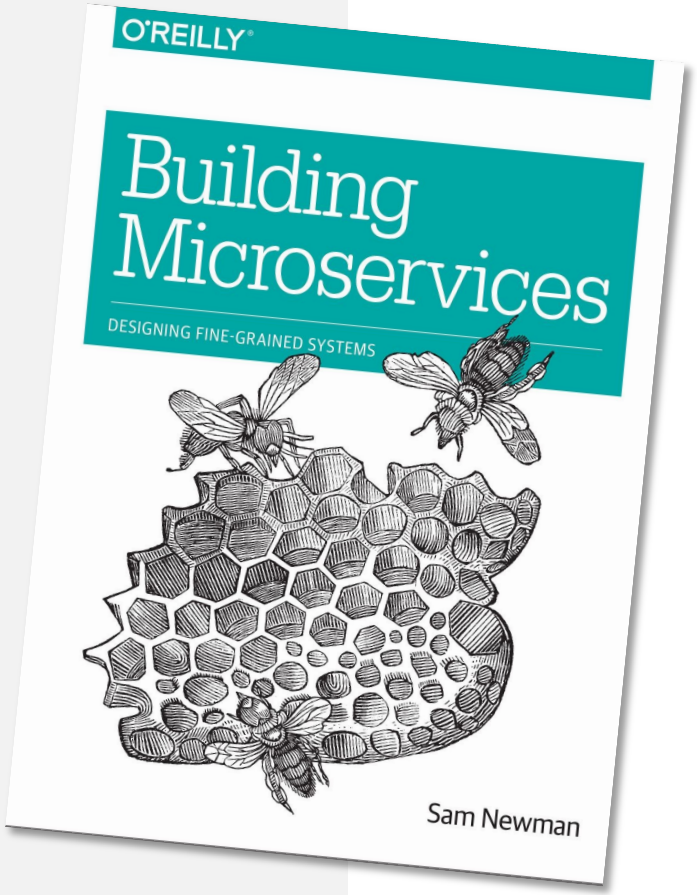


Sam Newmann, 2014

Sam Newman ist **einer der prägendsten Experten im Bereich Microservices**.

Er ist als unabhängiger Berater tätig und hat zuvor unter anderem bei ThoughtWorks (Martin Fowler) und einigen Start-ups gearbeitet. Heute ist er als selbstständiger Berater und Autor tätig.

Er ist Autor des „Standardwerks“ Building Microservices und empfiehlt Microservices anhand von **7 Prinzipien** zu entwickeln.



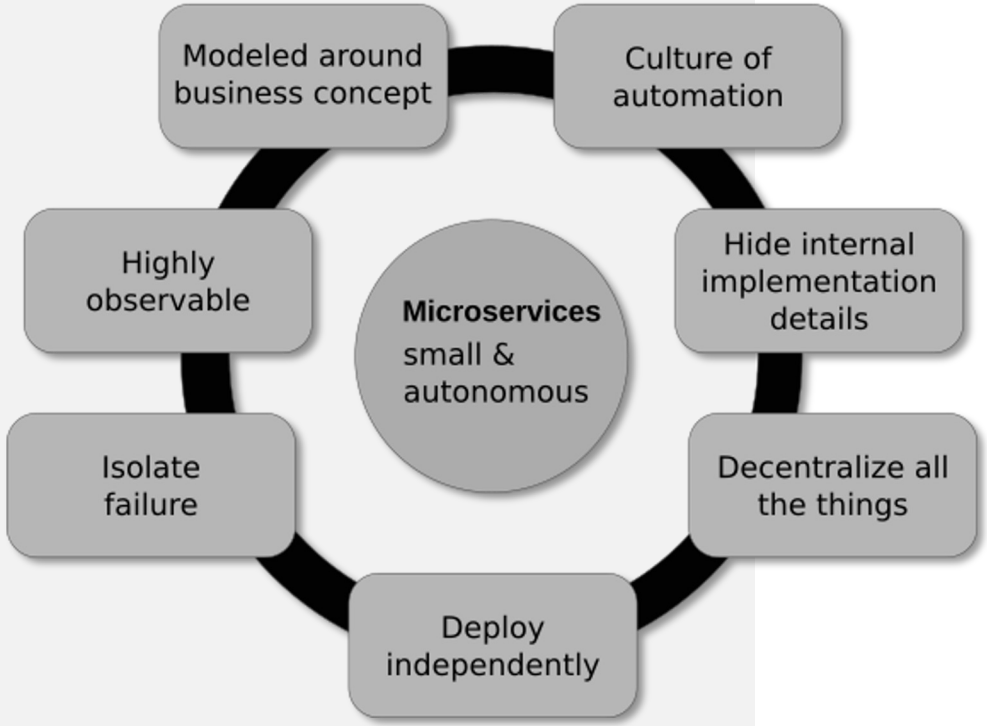
7 PRINZIPIEN VON MICROSERVICES

CLOUD NATIVE COMPUTING

Zusammenfassung dieser (und vorhergehender) Units

- Bilde Modelle um **Geschäftskonzepte**
- Erschaffe eine Kultur der **Automatisierung**
- Blende interne Implementierungsdetails aus
- **Dezentralisiere**
- Definiere **unabhängig aktualisierbare Einheiten**
- **Isoliere Fehler**
- Baue gut **beobachtbare Services**

- Wann man keine Microservices nutzen sollte



Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Prinzip 1: Bilde Modelle um Geschäftskonzepte

- Die Erfahrung lehrt folgendes:
- Schnittstellen, die **Domain-driven** (also aus der Fachlichkeit) abgeleitet werden, sind meist stabiler als solche, die primär technischen Erfordernissen folgen.
- Die Modellierung der **System-Domäne** fördert:
  - **stabilere Schnittstellen**
  - bessere Abbildbarkeit von **Geschäftsprozess-Änderungen**
- Die Methodik der **Bounded Contexts** (DDD) ermöglicht es potenzielle Domänengrenzen zu definieren

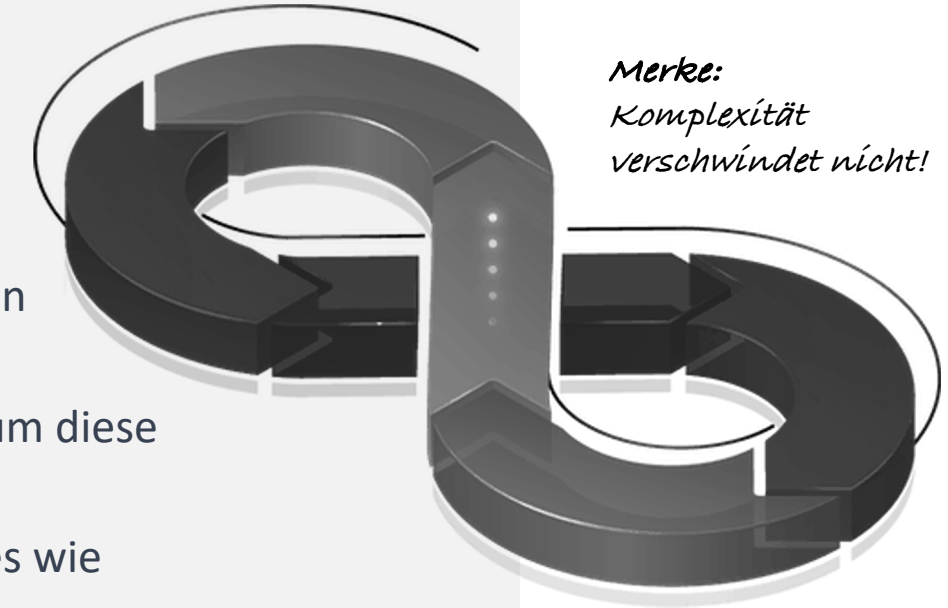
Die Methodik des Domain-Driven-Designs (DDD) werden wir in Unit 10 behandeln.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Prinzip 2: Erschaffe eine Kultur der Automatisierung

- Einzelne Microservices sind **einfach**.
- Eine **Architektur** von Microservices ist im Allgemeinen **komplex**.
- Ein wesentlicher Grund für diese Komplexität ist die steigende Anzahl von **Wechselwirkungen eines Gesamtsystems**.
- Die Einführung einer Kultur der Automatisierung ist ein wichtiger Weg, um diese **Komplexitätsverschiebung** zu beherrschen.
  - **Automatisierte Tests** stellen sicher, dass Services auch nach Updates wie erwartet funktionieren.
  - **Automatisierte Deployments** ermöglichen schnelles Feedback zur Produktionsqualität.
- Die Definition von **Environments** (z.B. Test, Staging, Production) ermöglicht unterschiedliche Ausbaustufen eines Systems bereitstellen und testen zu können.
- **Container** Images können die Bereitstellung von Komponenten beschleunigen und deren **Betrieb zu standardisieren** und mittels **Orchestrierungsplattformen** zu automatisieren.



Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

CLOUD NATIVE COMPUTING

Merke:
Komplexität verschwindet nicht!

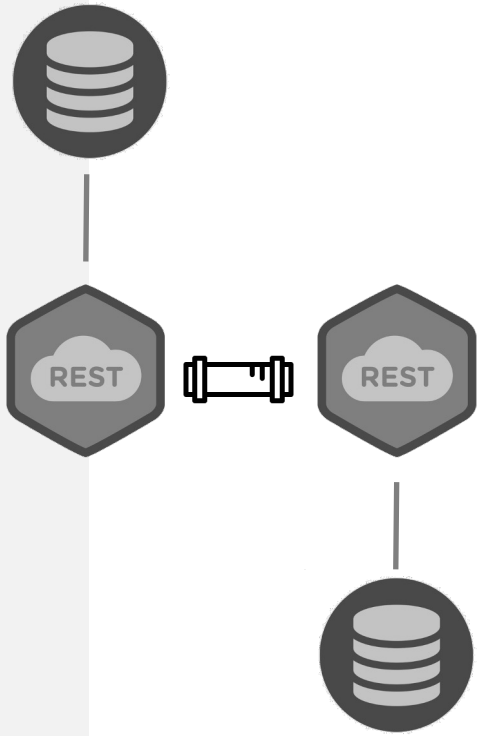
Deploy Environments können Sie bspw. auch in Gitlab-CI Pipelines definieren (vgl. Unit 02, DevOps).

Container (Images) haben wir in Unit 04+05 behandelt.

7 PRINZIPIEN VON MICROSERVICES

Prinzip 3: Blende interne Implementierungsdetails aus

- Implementierungsdetails müssen verborgen werden, um Abhängigkeiten zu vermeiden
- **Bounded Contexts** helfen bei der Identifizierung von gemeinsam genutzten Schnittstellen
- Services sollten ihre Datenbanken verbergen, um **Datenkopplung** zu **vermeiden**
- **(Event) Data Pumps** können genutzt werden, um über mehrere Services hinweg zu konsolidieren
- **Technologieunabhängige APIs** ermöglichen Flexibilität bei der Verwendung verschiedener Technologie-Stacks
- Die **REST-Philosophie** kann zur Trennung von internen und externen Implementierungsdetails genutzt werden (auch bei der Verwendung von Remote Procedure Calls, RPCs)



Bounded Contexts und Domain Driven Design werden wir in Unit 10 behandeln.

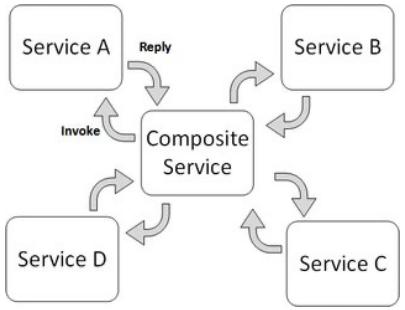
Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

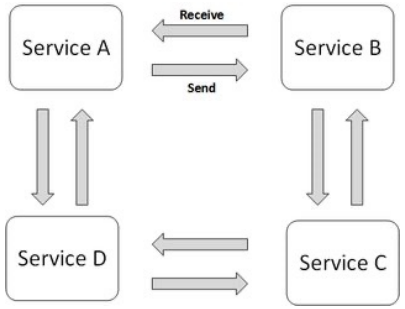
Prinzip 4: Dezentralisiere

- Nutzung von **Self-Service Lösungen** zur Bereitstellung von Software
- Teams sollten **Servicebesitzer** sein (Service Ownership)
- **Verantwortlichkeit** der Teams für vorgenommene Änderungen
- Ausrichtung der Organisation an dem zu erstellenden System, um das **Gesetz von Conway** zu berücksichtigen
- Entwicklung von übergreifenden Richtlinien in einem **Shared-Governance-Modell**
- **Vermeidung von zentralisierenden Ansätzen** wie Enterprise Service Bus oder Orchestrierungssystemen
- Ansätze wie **Prefer-Choreography-over-Orchestration** und **Dumb-Middleware-with-Smart-Endpoints** zur Erhaltung von Service-Kohäsion nutzen

Service Orchestration
(zentral)



Service Choreography
(dezentral)



Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Prinzip 5: Definiere unabhängig aktualisierbare Einheiten

- Änderungen an einem Service sollten **unabhängig** von anderen Services durchgeführt werden
- Entscheidung zur Aktualisierung sollte auf Seite des **konsumierenden Service**s, nicht bei bereitstellenden Services (Service-Ownership) getroffen werden
- Parallele Verwendung von **versionierten Endpunkten** bei Breaking-Changes für schnellere Releases neuer Funktionen ohne Alt-Funktionen sofort anpassen zu müssen
- **Vermeidung** eng gebundener **Client/Server-Stub-Generierung** bei RPC-basierter Integration
- **One-Service-per-Host/Pod/Container-Modell** zur Reduzierung von Auswirkungen bei Aktualisierungen
- Einsatz von **Blue/Green** oder **Canary-Release-Techniken** zur Risikominimierung von Deployments

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

CLOUD NATIVE COMPUTING

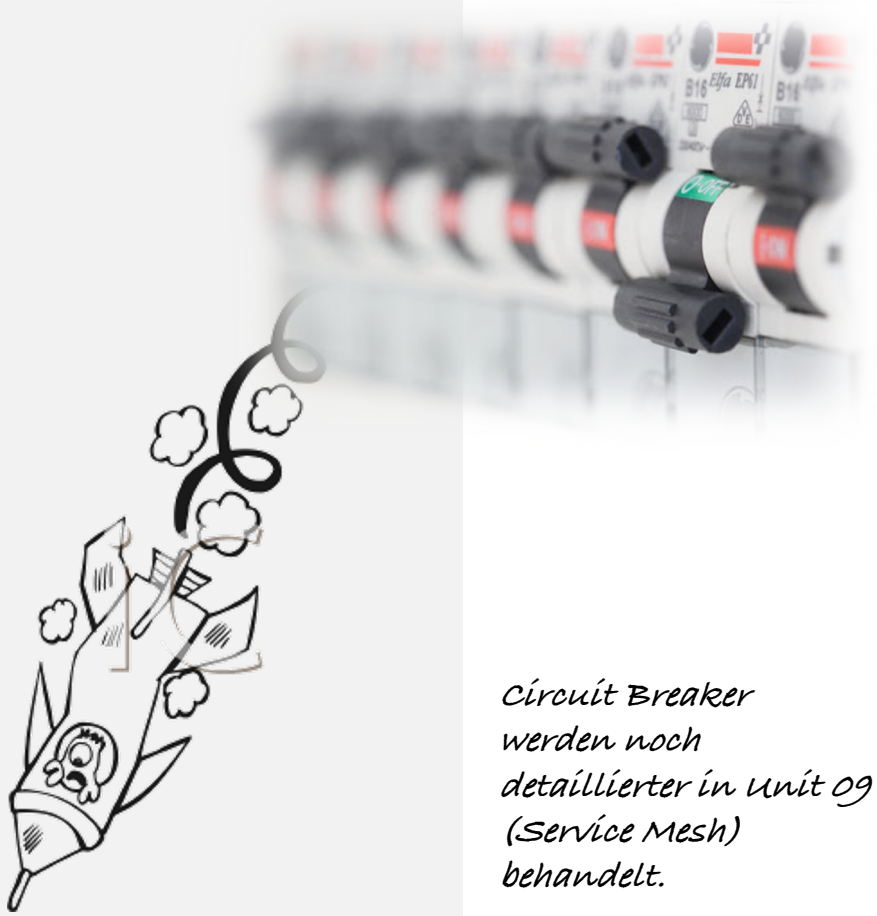
Release Pattern hatten wir bereits in Unit 02 (DevOps) kennen gelernt und werden auch noch mal in Unit 09 (Service Meshes behandelt werden).

7 PRINZIPIEN VON MICROSERVICES

CLOUD NATIVE COMPUTING

Prinzip 6: Isoliere Fehler

- Eine Microservice-Architektur kann widerstandsfähiger sein als ein monolithisches System, aber nur bei **fehlertolerantem System-Design**
- RPC Remote Calls sollten nicht wie Local Calls behandelt werden, um Auswirkungen von Netzwerkfehlern einzudämmen
- **Timeouts** entsprechend einstellen
- **Circuit Breaker** verwenden, um Auswirkungen einer fehlerhaften Komponente zu begrenzen
- Auswirkungen für Nutzer verstehen, wenn sich ein Teil des Systems fehlerhaft verhält
- Auswirkungen von **Netzwerkpartitionen** berücksichtigen und Entscheidung über Opferung von Verfügbarkeit oder Konsistenz treffen (CAP-Theorem)



Circuit Breaker werden noch detaillierter in Unit 09 (Service Mesh) behandelt.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

7 PRINZIPIEN VON MICROSERVICES

Prinzip 7: Baue gut beobachtbare Services

- Bei Microservices reicht das Beobachten einer einzelnen Service-Instanz nicht aus, um feststellen zu können, ob ein verteiltes **System als Ganzes** funktioniert
- Eine **integrierte Sicht** auf das Systemverhalten ist erforderlich
- Semantic Monitoring kann das Verhalten des Systems überwachen
- **Synthetische Transaktionen** können in Systeme eingespielt werden, um so das Verhalten realer Benutzer zu simulieren
- **Logs** und **Metriken** sollten so aggregiert werden, das man Probleme lokalisieren und einen Drilldown zur Quelle durchführen kann
- Korrelations-Ids ermöglichen Tracing von Transaktionen in verteilten Systemen (**Distributed Tracing**)

Tracing und Observability werden wir in Unit 09 behandeln.

Quelle: Sam Newman, Building Microservices, O'Reilly, 2014

WANN SOLLTEN MICROSERVICES NICHT VERWENDET WERDEN?

Microservices are not a free lunch

- Netflix hat mit über 700 Microservices den Microservice-Architekturansatz populär gemacht
- Viele Unternehmen haben den Ansatz daraufhin adaptiert
- Erfolgsgeschichten kommen allerdings hauptsächlich von produktbasierten Unternehmen
- Microservice-Architekturen ermöglichen agile Entwicklungen erfolgreicher Produkte für Unternehmen, die aufgrund ihres Fokus viele Iterationen und Modelle durchlaufen können
- Microservices sind daher kein Garant für Erfolg
- Erfolgreiche Produkte in der Cloud-basierten Geschäftswelt sind zwar oft Microservice-basiert
- aber nicht alle Microservice-basierten Produkte sind automatisch erfolgreich

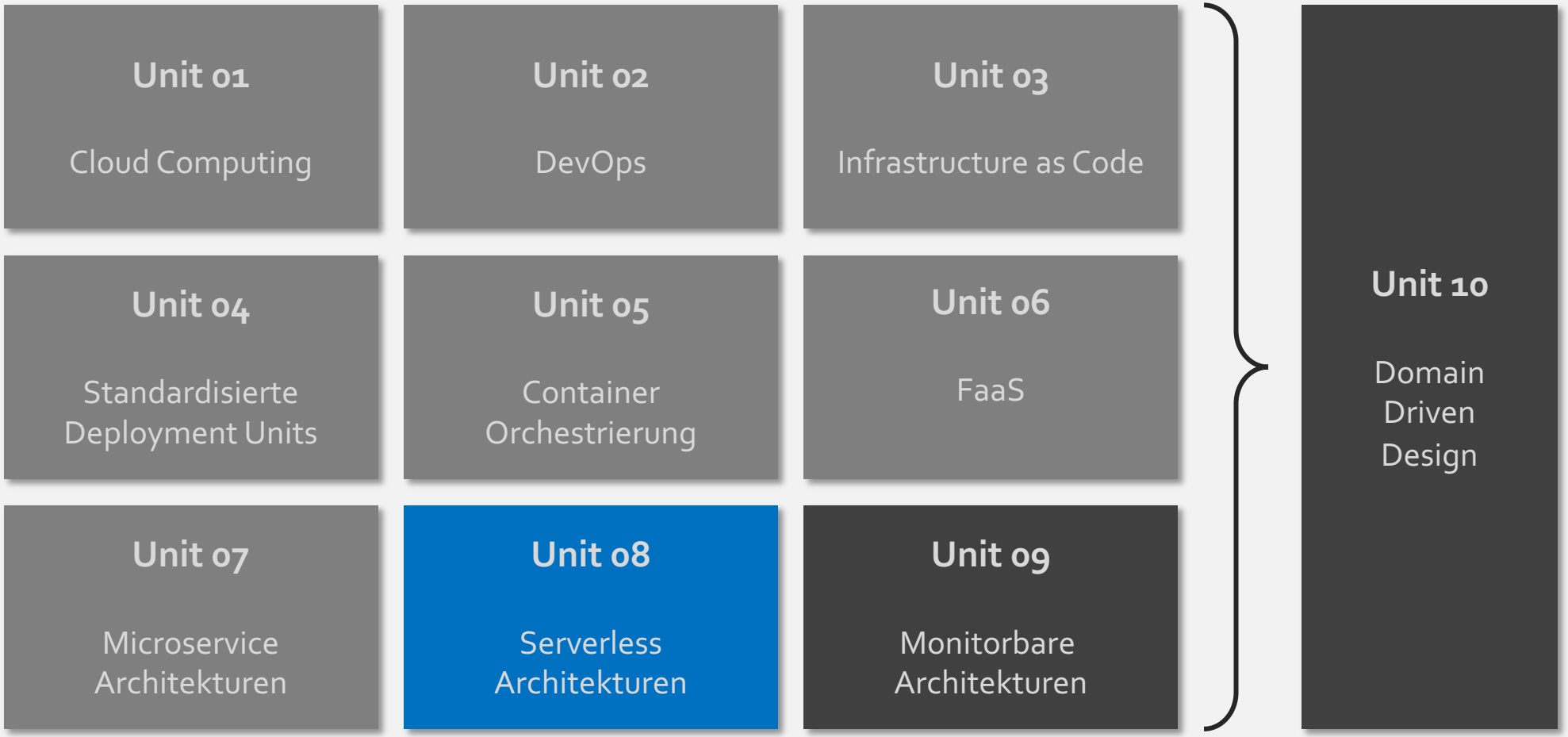
- **Microservices sind Lösungen für komplexe Probleme**
  Ohne komplexes Problem, gibt es keinen wirklichen Grund für Microservices
- **Unterschätzen Sie nicht die Anzahl erforderlicher Teams (und deren Größen, 1-12 Personen)**
  Netflix braucht gem. dem One-Service-Per-Team Ansatz somit etwa 700 Teams!
- **Wenn Sie nicht skalieren müssen, bringen Microservices häufig wenig Mehrwert**
  Die THL hat bspw. etwa 5.000 Studierende, das werden nicht über Nacht plötzlich mehr.

CLOUD NATIVE COMPUTING

NETFLIX

AUSBLICK

Überblick über Units



KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de