Cloud Architektur

24.11.2021

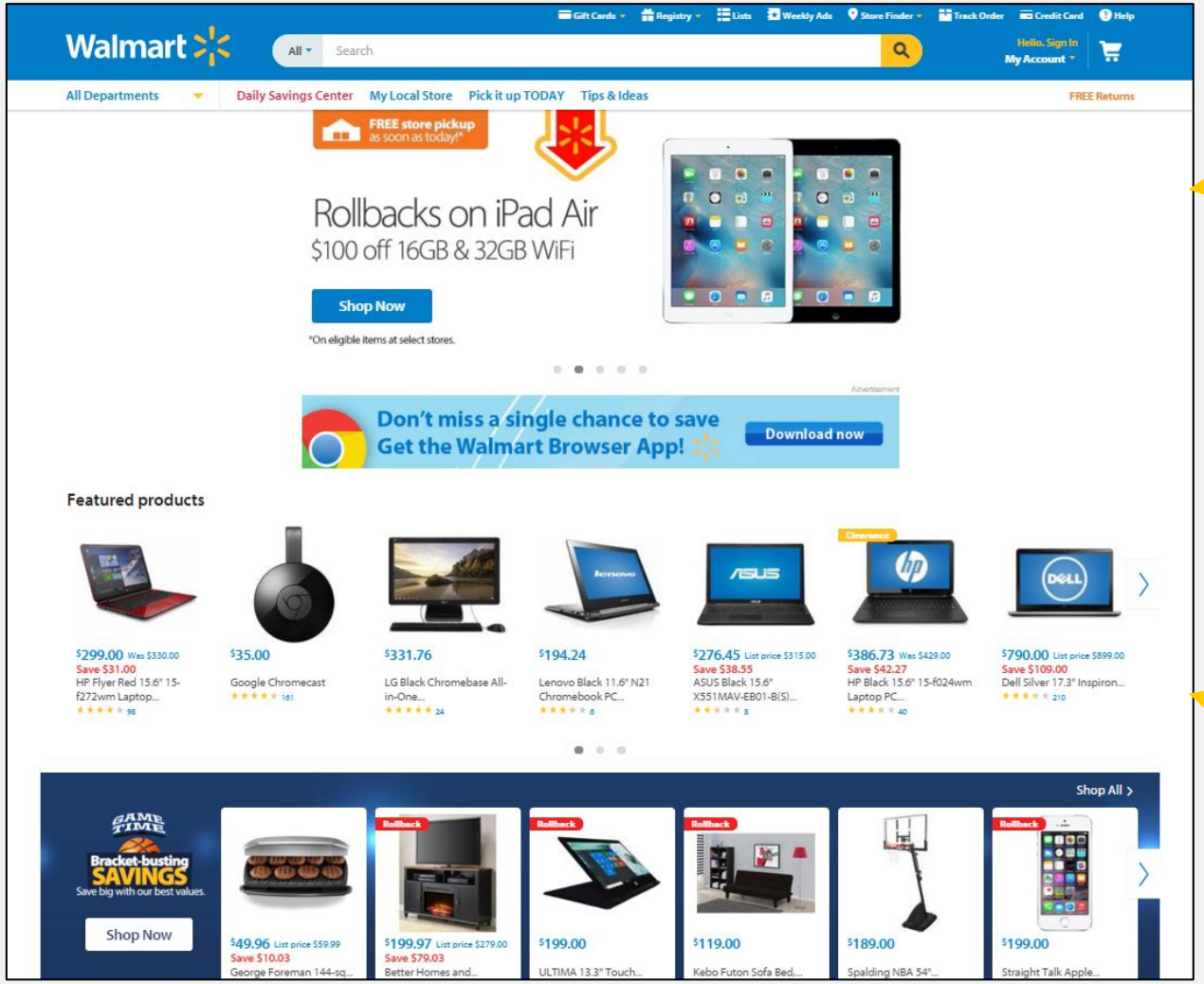
WHAT DID THEY DO?

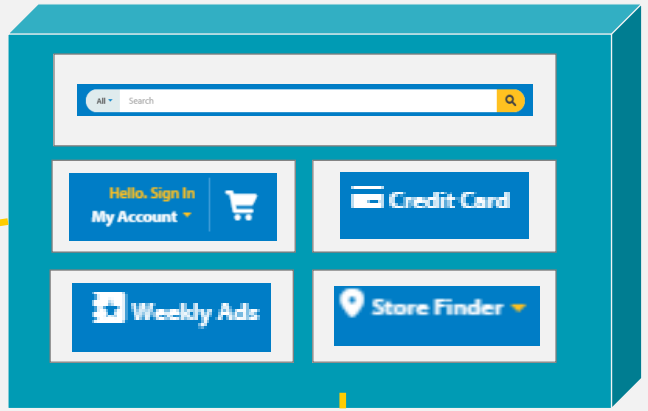
Betriebsmonolithen

Knoten

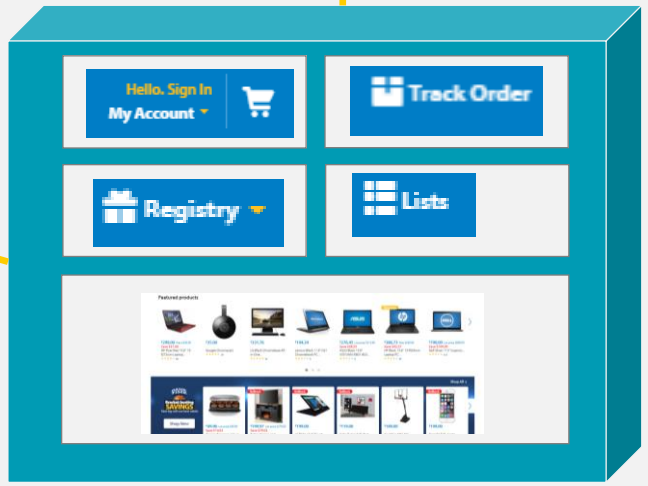
Deployment-Einheit

Betriebskomponenten





Skalieren durch
Verteilung und Klone

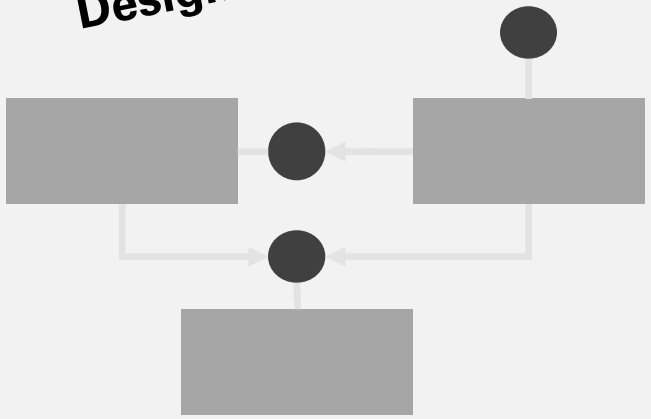


Cloud Native Application Development: Components All Along the Software Lifecycle.

DESIGN → BUILD → RUN

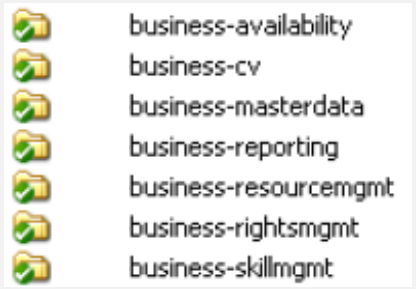
NEW!

Design Components



1:1

Dev Components



n:1

Ops Components

- Complexity unit
- Data integrity unit
- Coherent and cohesive features unit
- Decoupled unit

- Planning unit
- Team assignment unit
- Knowledge unit
- Development unit
- Integration unit

- Release unit
- Deployment unit
- Runtime unit (crash, slow-down, access)
- Scaling unit

Die Anatomie einer Betriebs-Komponente.

Diagnose Interface

Internet Protocol

Container

Application

Starting Inteface

Internet Protocol

Regel 1 für den Betrieb in der Cloud.

"Everything fails all the time."

— Werner Vogels, CTO of Amazon

Regel 2 für den Betrieb in der Cloud.

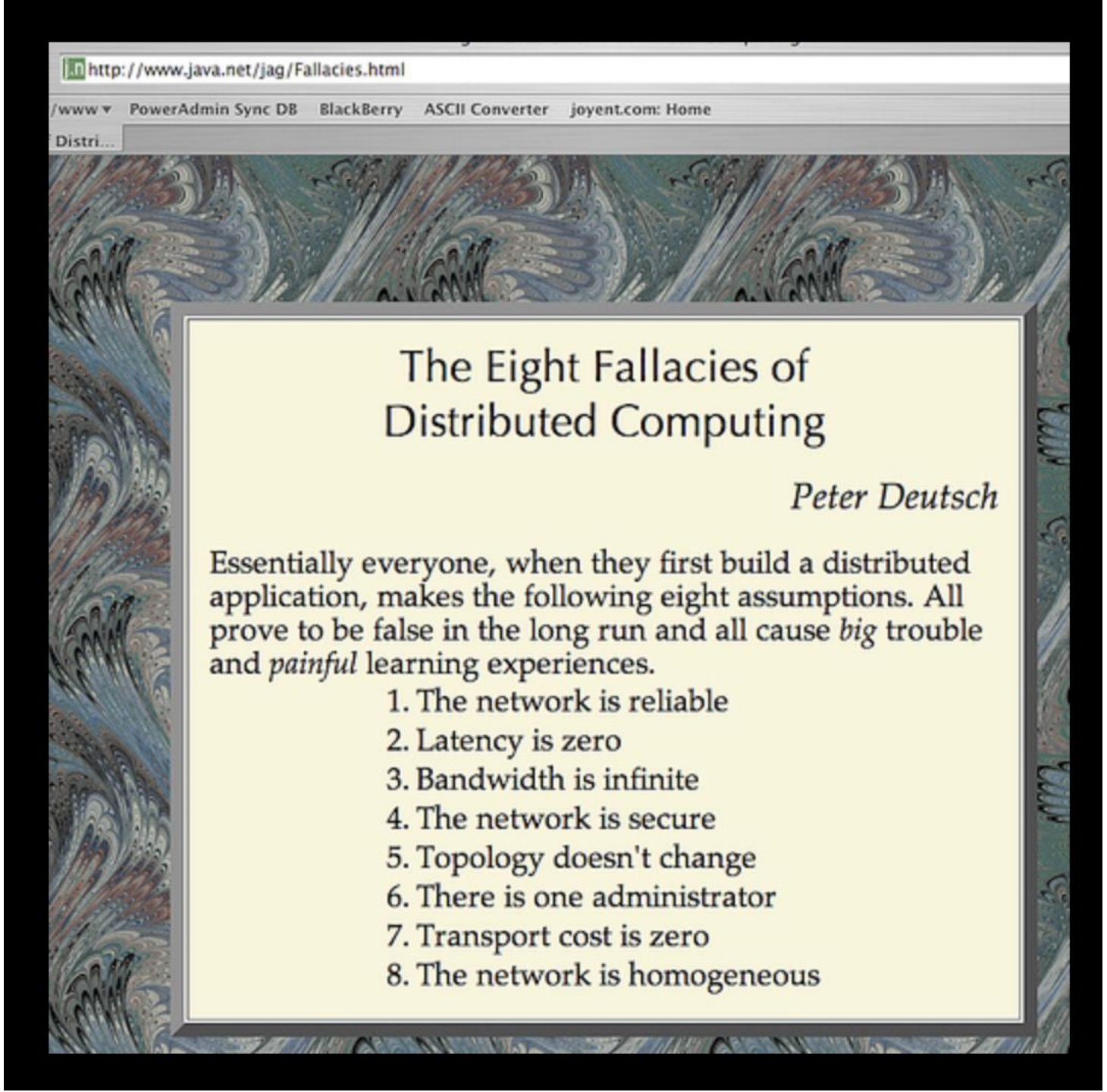
Soll nur der Himmel die Grenze sein, dann funktioniert nur horizontale Skalierung.

Regel 3 für den Betrieb in der Cloud.

Wer in die Cloud will, der sollte Cloud sprechen.

Die 5 Gebote der Cloud.

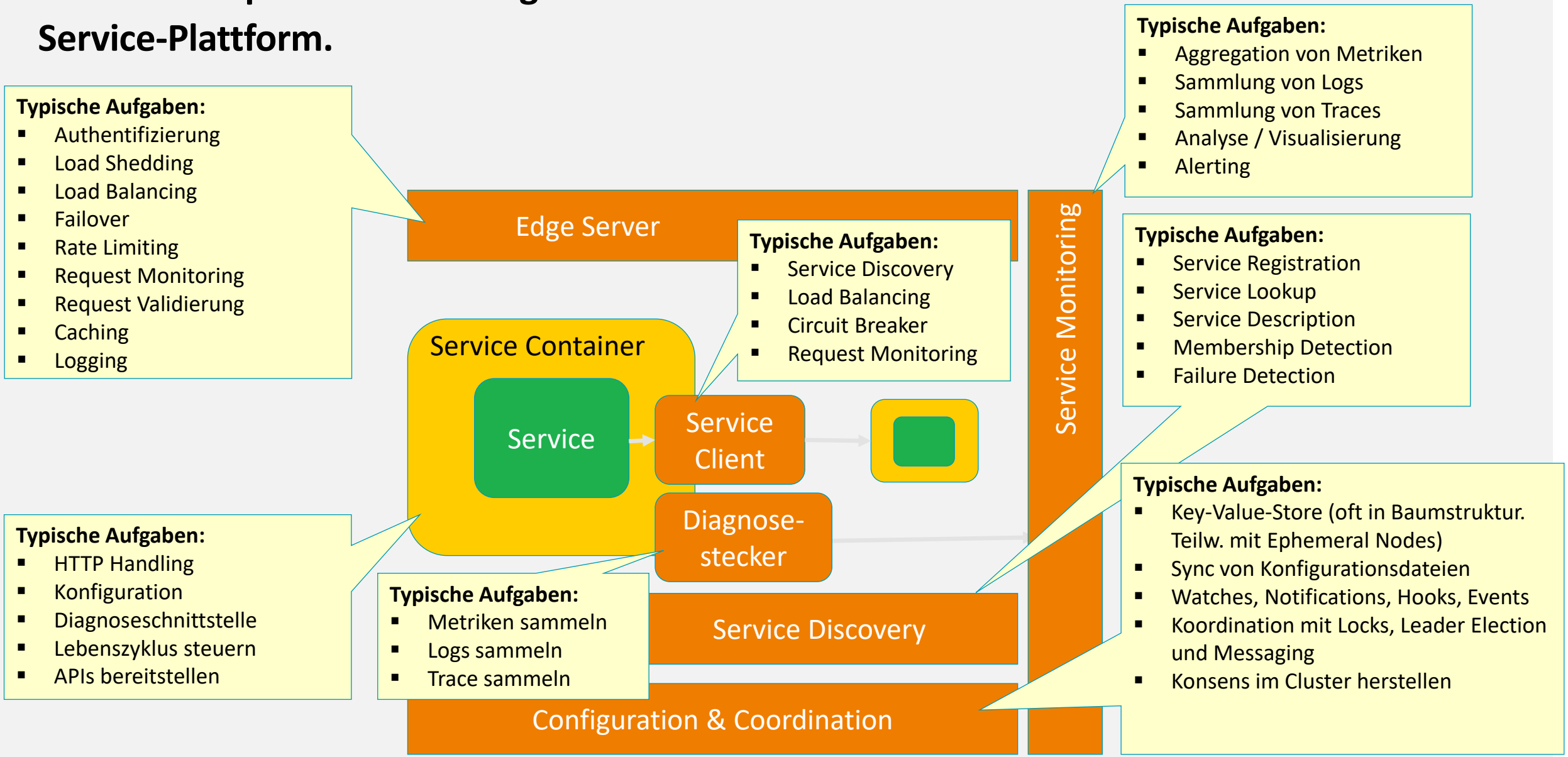
1. Everything Fails All The Time.
2. Focus on MTTR and not on MTTF.
3. Respect the Eight Fallacies of Distributed Computing.
4. Scale out, not up.
5. Treat resources as cattle, not pets.


http://www.java.net/jag/Fallacies.html
/www
PowerAdmin Sync DB
BlackBerry
ASCII Converter
joyent.com: Home
Distri...
The Eight Fallacies of Distributed Computing
Peter Deutsch
Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.
1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

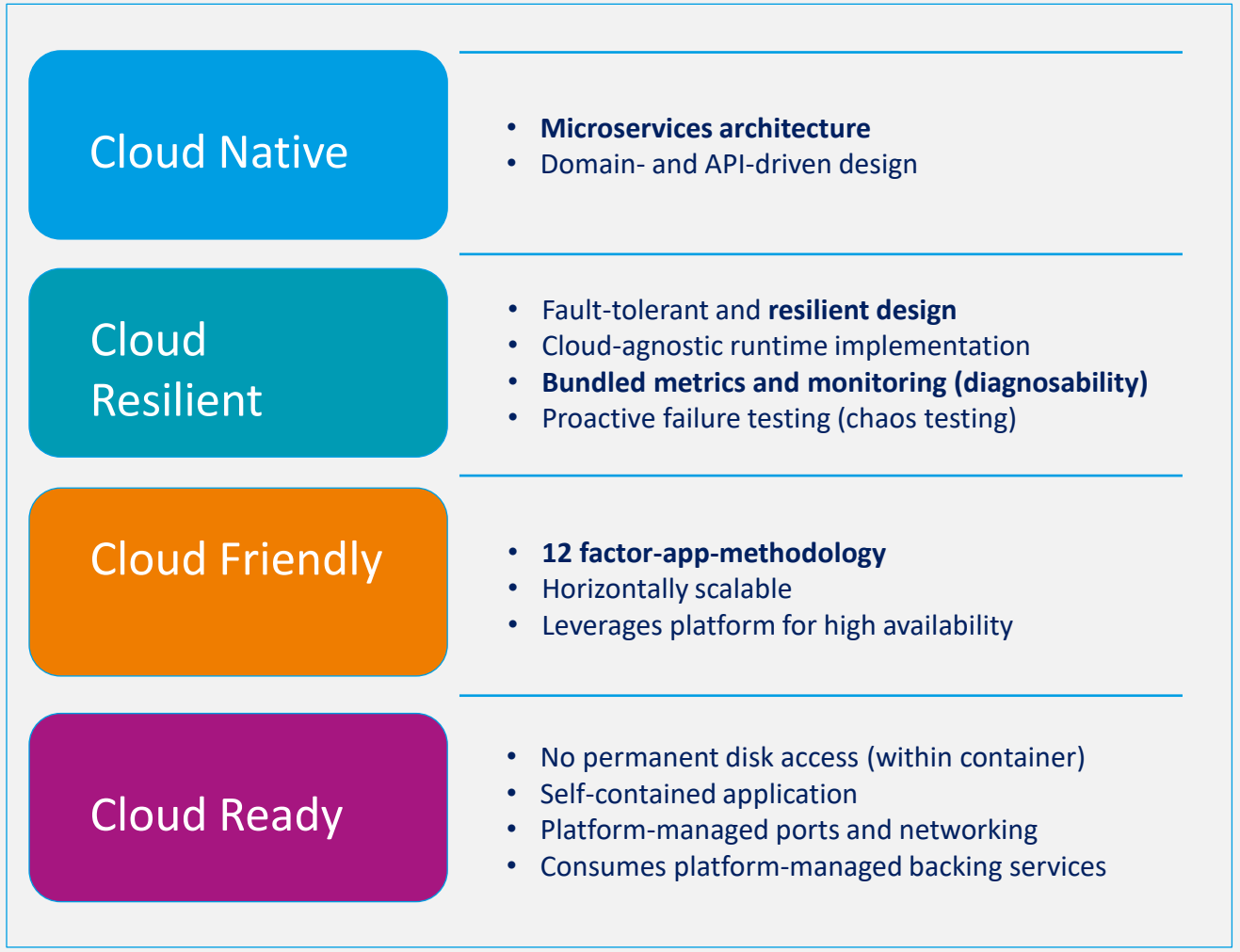
Cloud-Architektur aus Sicht der Softwarearchitektur:
Design for Failure.

1. Jede Komponente läuft eigenständig und isoliert ➔ *Betriebskomponenten*
2. Die Betriebskomponenten kommunizieren untereinander über Internet-Protokolle ➔ *HTTP, UDP, …*
3. Jede Betriebskomponente kann in mehreren Instanzen laufen und bietet damit Redundanz. Es gibt keinen „Common Point of Failure“. ➔ *Cluster Orchestrator*
4. Jede Betriebskomponente besitzt Diagnoseschnittstellen um ein fehlerhaftes Verhalten erkennen zu können
5. Jeder Microservice kann zu jeder Zeit neu gestartet und auf einem anderen Knoten in Betrieb genommen werden. Er besitzt keinen eigenen Zustand.
6. Die Implementierung hinter einem jeden Microservice kann ausgetauscht werden, ohne dass die Nutzer davon etwas bemerken.

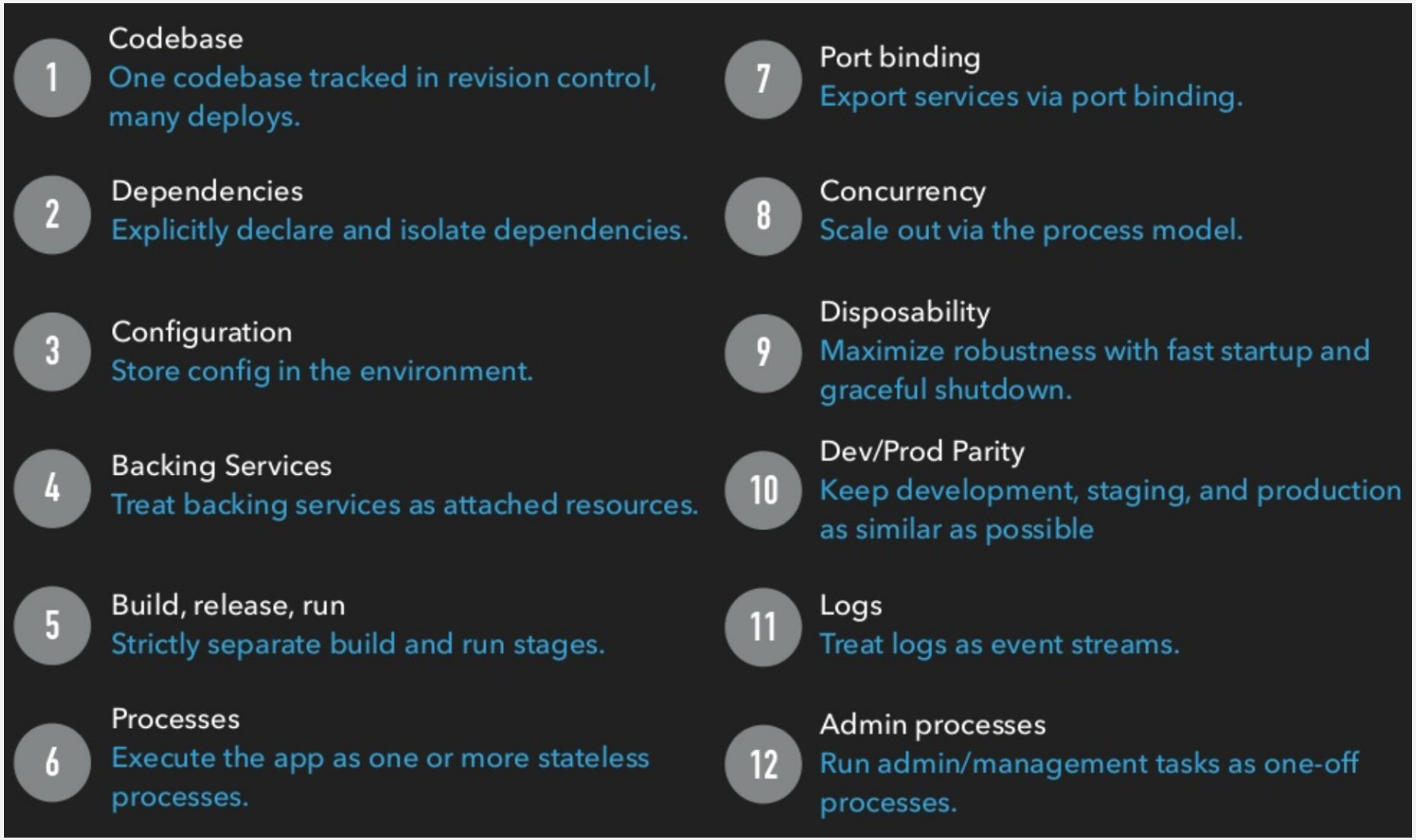
Betriebskomponenten benötigen eine Infrastruktur um sie herum: Eine Micro-Service-Plattform.



Das Cloud Native Application Reifegradmodell



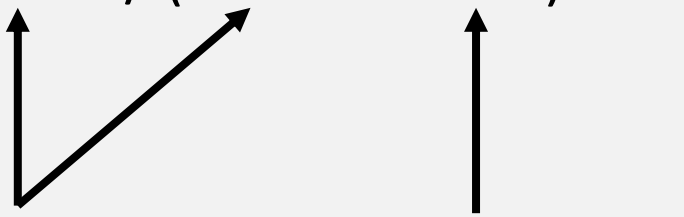
12 Factor App



https://12factor.net/de
https://www.slideshare.net/Alicanakku1/12-factor-apps

Resilienz

Verfügbarkeit = MTTF / (MTTF + MTTR)



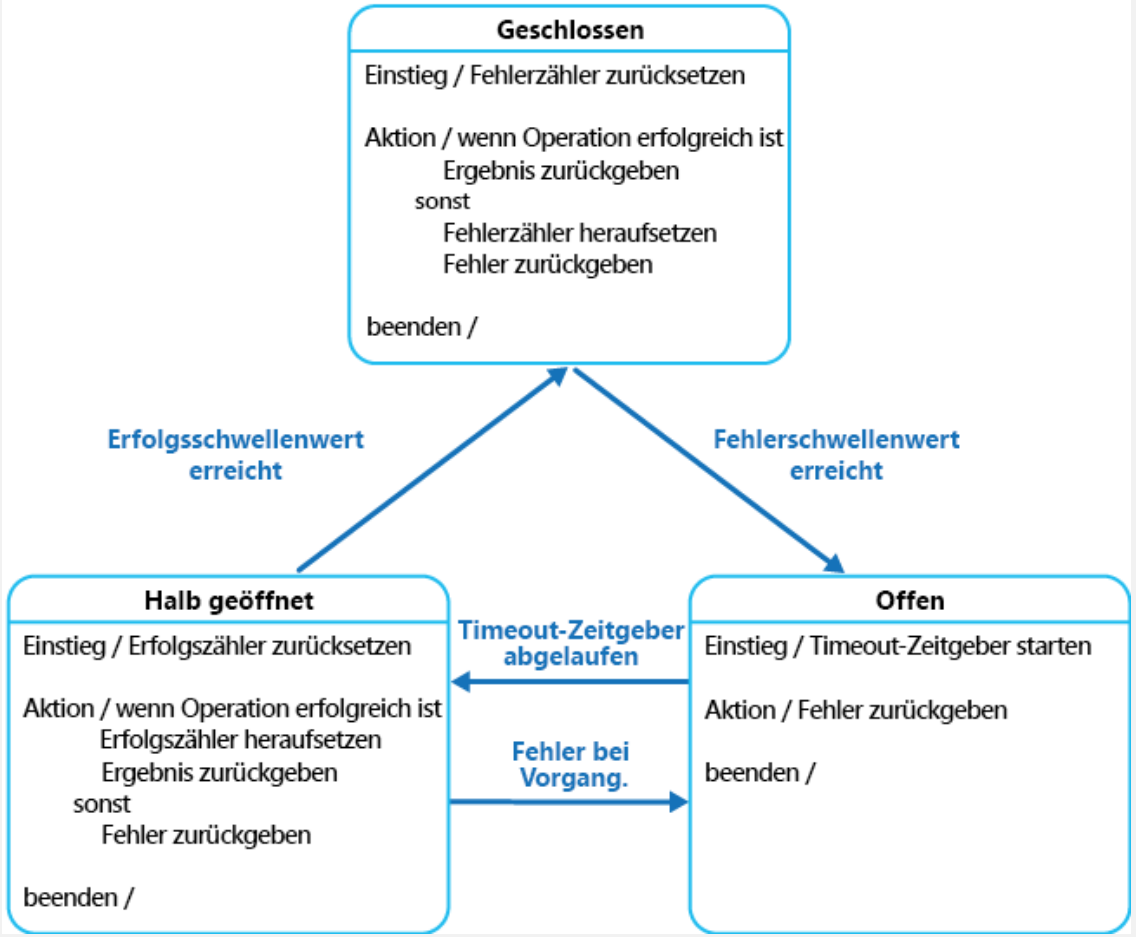
Zuverlässigkeit steigern

Resilienz & Diagnostizierbarkeit

Resilienz: Die Fähigkeit eines Systems mit unerwarteten und fehlerhaften Situationen umzugehen

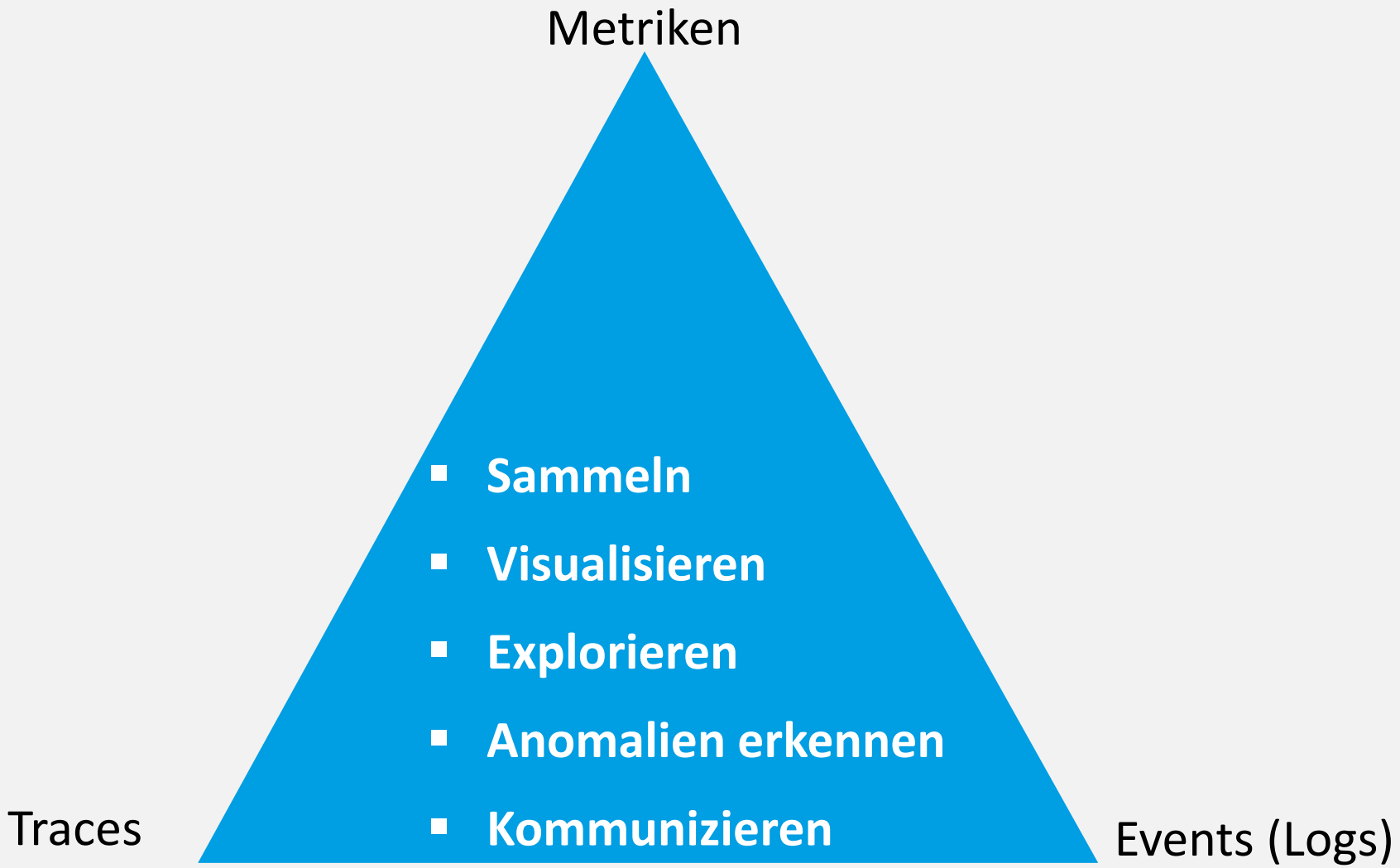
- Ohne dass es der Nutzer merkt (Bestfall)
- Mit ein einer „graceful degradation“ des Services (schlechtester Fall)

Resilienz-Pattern: Circuit Breaker

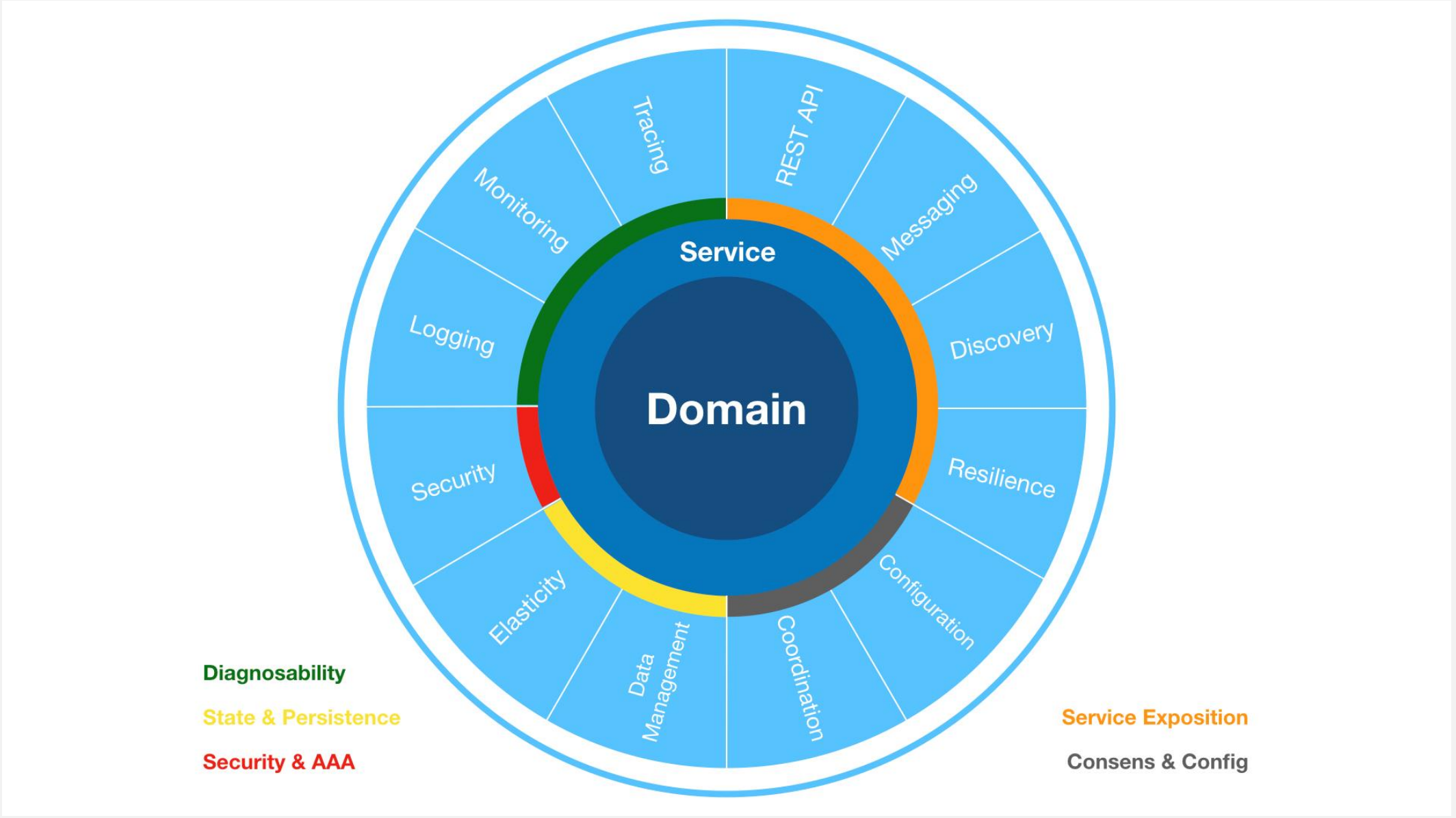


Weitere Patterns: https://docs.microsoft.com/de-de/azure/architecture/patterns/category/resiliency

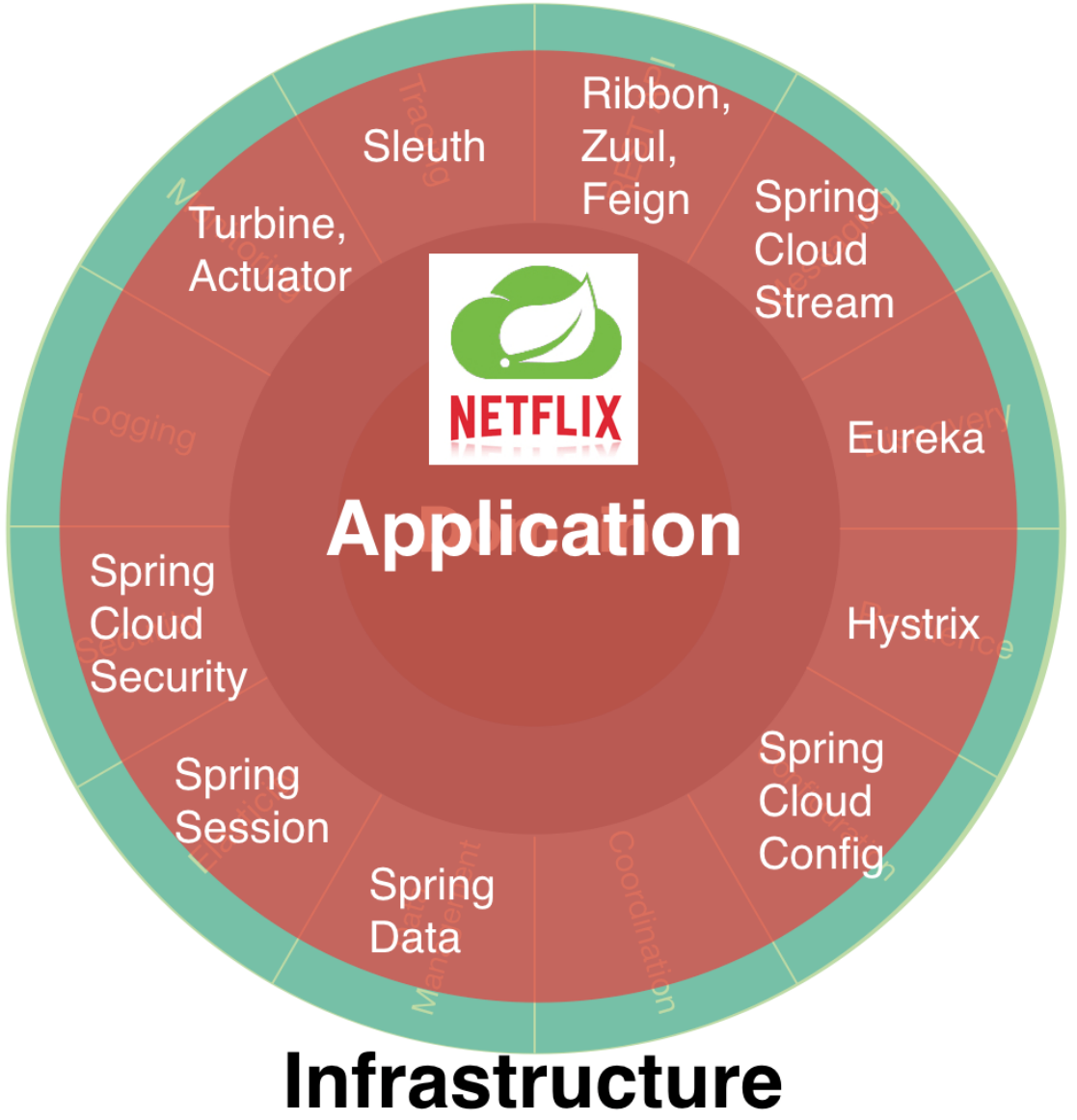
Dignostizierbarkeit



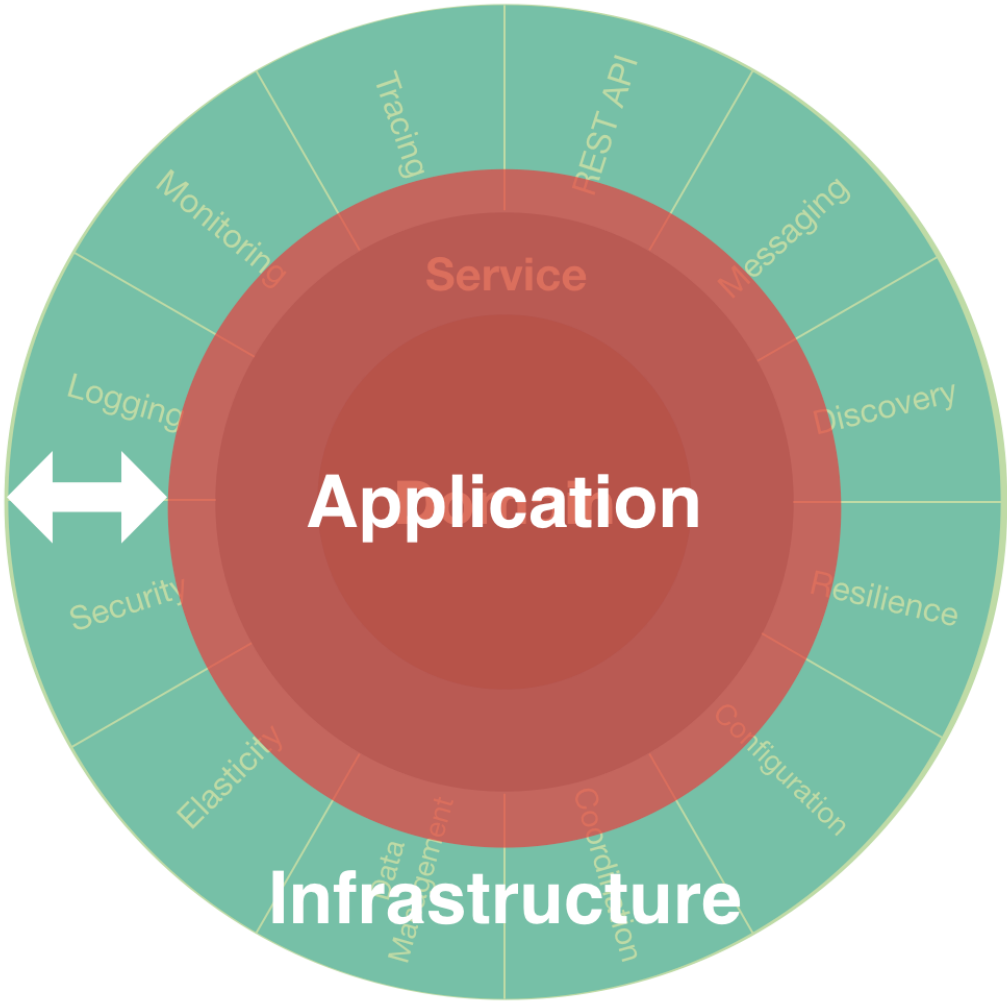
Technische Aspekte von Microservices



Technische Aspekte von Microservices: Library Lösung

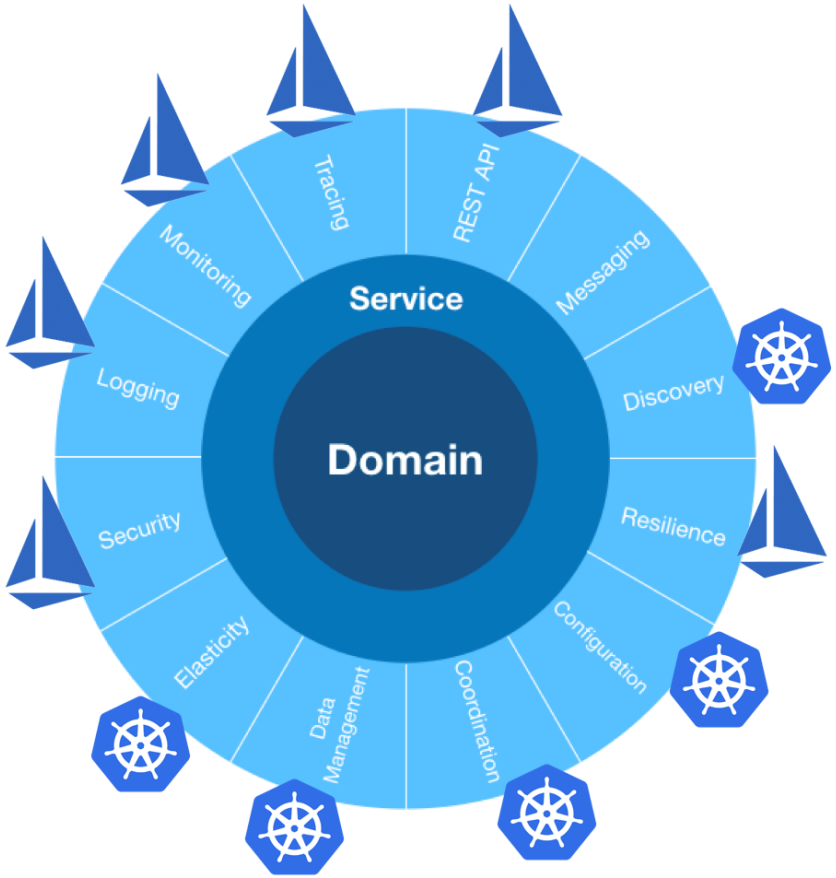


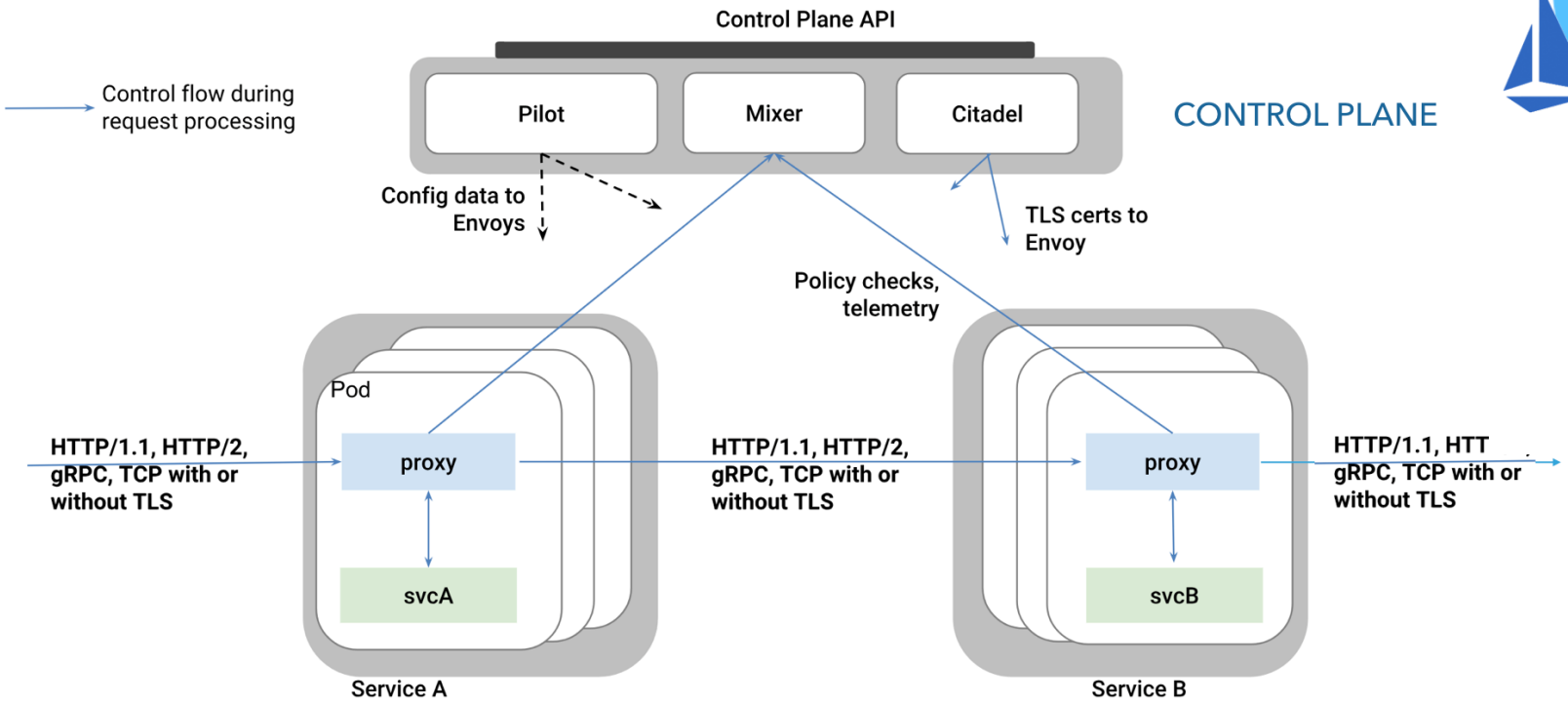
Technische Aspekte von Microservices: Infrastruktur Lösung



Istio Service Mesh

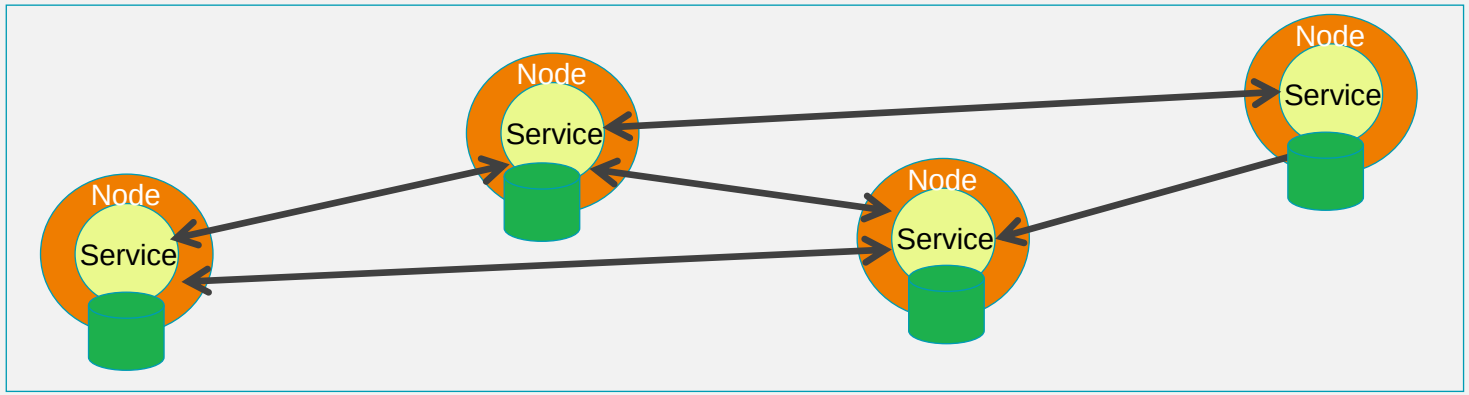
- Open-Source-Projekt von Google, IBM, Lyft, RedHat u.A.
- Aufsatz auf Kubernetes, der wichtige technische Aspekte auf Infrastruktur-Seite ergänzt





Configuration & Coordination: Verteilter Zustand & Konsens

Ein verteilter Konfigurationsspeicher.



Wie wird der Zustand des Konfigurationsspeichers im Cluster synchronisiert?

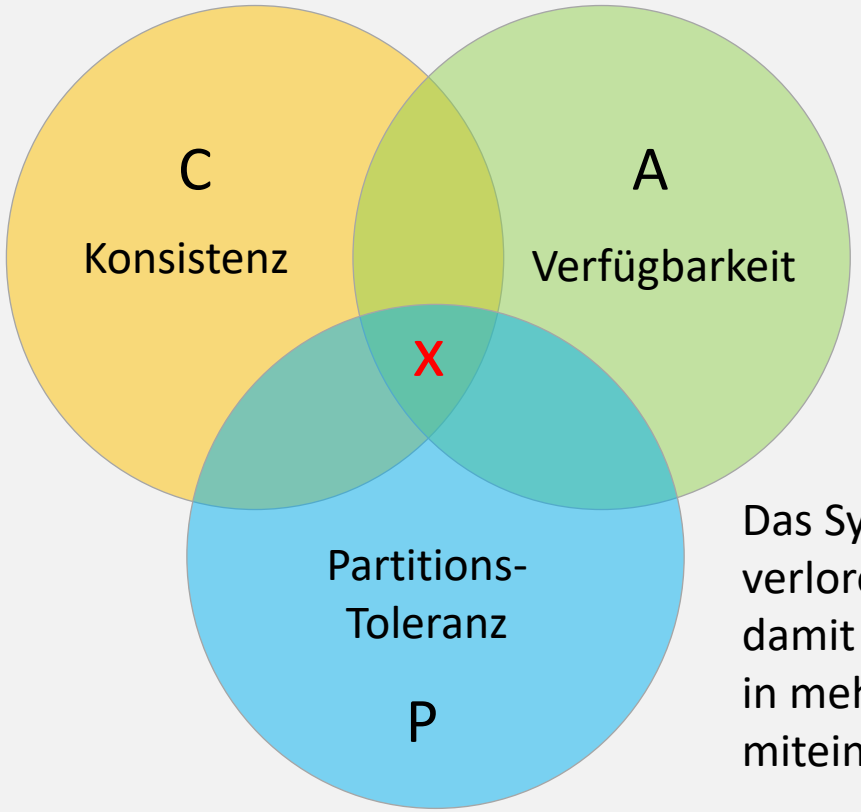
Das CAP Theorem.

Theorem von Brewer für Eigenschaften von zustandsbehafteten verteilten Systemen – mittlerweile auch formal bewiesen.

Brewer, Eric A. "Towards robust distributed systems." *PODC*. 2000.

Es gibt drei wesentliche Eigenschaften, von denen ein verteiltes System nur zwei gleichzeitig haben kann:

Alle Knoten sehen die selben Daten zur selben Zeit. Alle Kopien sind stets gleich.



Das System läuft auch, wenn einzelne Knoten ausfallen. Ausfälle von Knoten und Kanälen halten die überlebenden Knoten nicht von ihrer Funktion ab.

Das System funktioniert auch im Fall von verlorenen Nachrichten. Das System kann dabei damit umgehen, dass sich das Netzwerk an Knoten in mehrere Partitionen aufteilt, die nicht miteinander kommunizieren.

Gossip Protokolle: Inspiriert von der Verbreitung von Tratsch in sozialen Netzwerken.

Grundlage: Ein Netzwerk an Agenten mit eigenem Zustand

Agenten verteilen einen Gossip-Strom

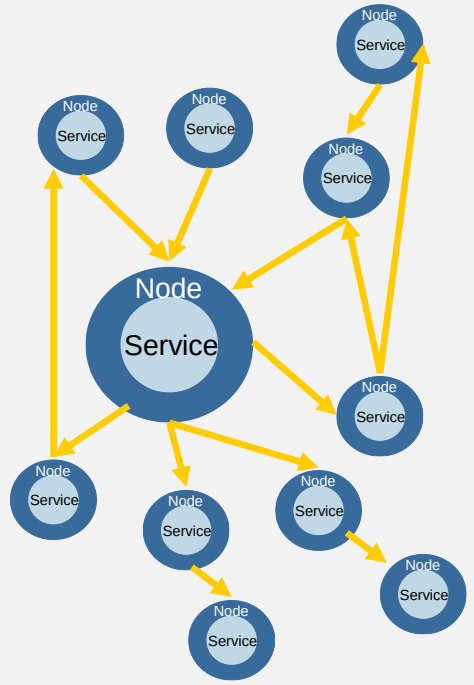
- Nachricht: Quelle, Inhalt / Zustand, Zeitstempel
- Nachrichten werden in einem festen Takt periodisch versendet an eine bestimmte Anzahl anderer Knoten (Fanout)

Virale Verbreitung des Gossip-Stroms

- Knoten, die mit mir in einer Gruppe sind, bekommen auf jeden Fall eine Nachricht
- Die Top x% an Knoten, die mir Nachrichten schicken bekommen eine Nachricht

Nachrichten, denen vertraut wird, werden in den lokalen Zustand übernommen

- Die gleiche Nachricht wurde von mehreren Seiten gehört
- Die Nachricht stammt von Knoten, denen der Agent vertraut
- Es ist keine aktuellere Nachricht vorhanden



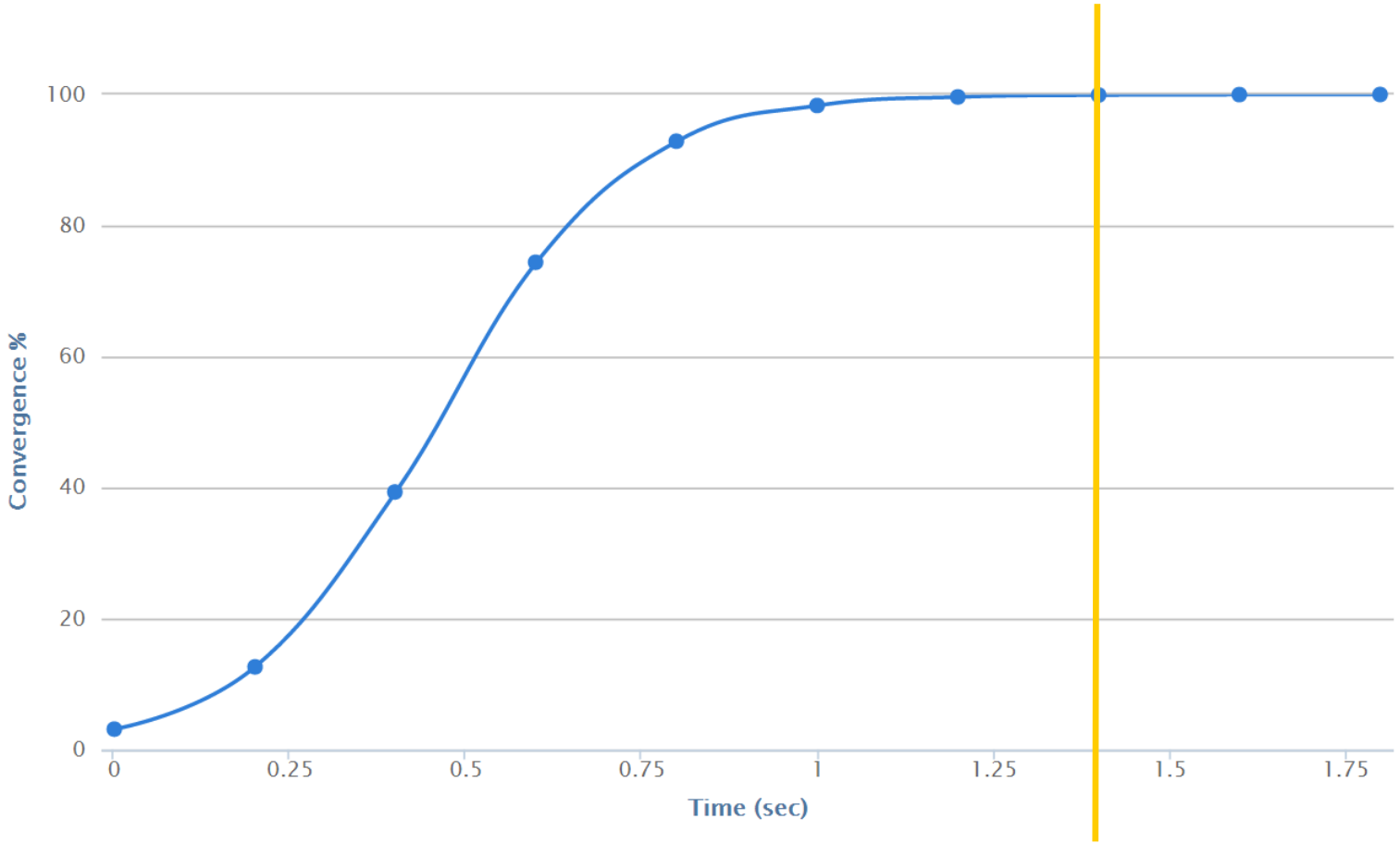
Vorteile:

- Keine zentralen Einheiten notwendig.
- Fehlerhafte Partitionen im Netzwerk werden umschifft. Die Kommunikation muss nicht verlässlich sein.

Nachteile:

- Der Zustand ist potenziell inkonsistent verteilt (konvergiert aber mit der Zeit)
- Overhead durch redundante Nachrichten.

Die Konvergenz der Daten und damit der Zeitpunkt der vollständigen Konsistenz ist berechenbar.



Vollständige Konsistenz

Estimated max bandwidth: 328.1 kbps/node

Gossip Interval

The gossip interval controls how often messages are gossiped to other nodes

0.2 seconds

Gossip Fanout

The gossip fanout controls how many nodes we gossip with

3 nodes

Nodes

This controls how many simulated nodes are in the cluster

30

Packet Loss

This controls the amount of simulated packet loss [0, 100)

0 % lost packets

Node failures

This controls what percent of simulated nodes are failed

0 % failed

https://www.serfdom.io/docs/internals/simulator.html

Protokolle für verteilten Konsens sind im Gegensatz zu Gossip-Protokollen konsistent aber nicht hoch-verfügbar.

Grundlage: Netzwerk an Agenten

Prinzip: Es reicht, wenn der Zustand auf einer einfachen Mehrheit der Knoten konsistent ist und die restlichen Knoten ihre Inkonsistenz erkennen.

Verfahren:

- Das Netzwerk einigt sich per einfacher Mehrheit auf einen Leader-Agenten – initial und falls der Leader-Agent nicht erreichbar ist. Eine Partition in der Minderheit kann keinen Leader-Agenten wählen.
- Alle Änderungen laufen über den Leader-Agenten. Dieser verteilt per Multicast Änderungsnachrichten periodisch im festen Takt an alle weiteren Agenten.
- Quittiert die einfache Mehrheit an Agenten die Änderungsnachricht, so wird die Änderung im Leader und (per Nachricht) auch in den Agenten aktiv, die quittiert haben. Ansonsten wird der Zustand als inkonsistent angenommen.

Konkrete Konsens-Protokolle: Raft, Paxos

Vorteile:

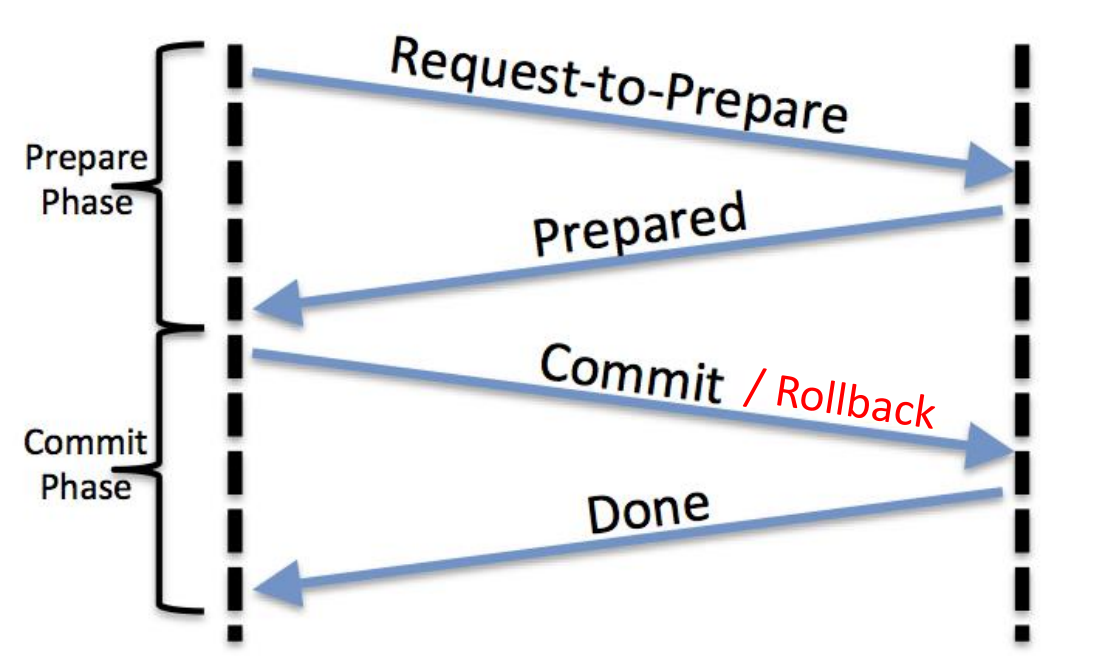
- Fehlerhafte Partitionen im Netzwerk werden toleriert und nach Behebung des Fehlers wieder automatisch konsistent.
- Streng konsistente Daten.

Nachteile:

- Der zentrale Leader-Agent limitiert den Durchsatz an Änderungen.
- Nicht hoch-verfügbar: Bei einer Netzwerk-Partition kann die kleinere Partition nicht weiterarbeiten. Ist die Mehrheit in keiner Partition, so kann insgesamt nicht weiter gearbeitet werden.

Ist strenge Konsistenz über alle Knoten notwendig, so verbleibt das 2-Phase-Commit Protokoll (2PC)

Ein Transaktionskoordinator verteilt die Änderungen und aktiviert diese erst bei Zustimmung aller. Ansonsten werden die Änderungen rückgängig gemacht.



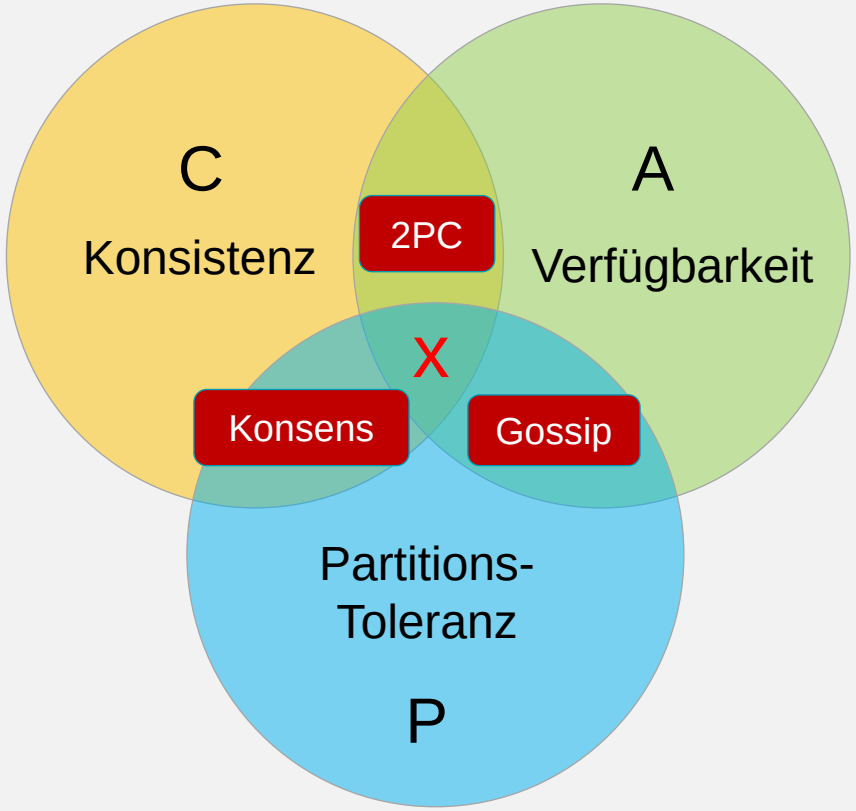
Vorteil:

- Alle Knoten sind konsistent zueinander.

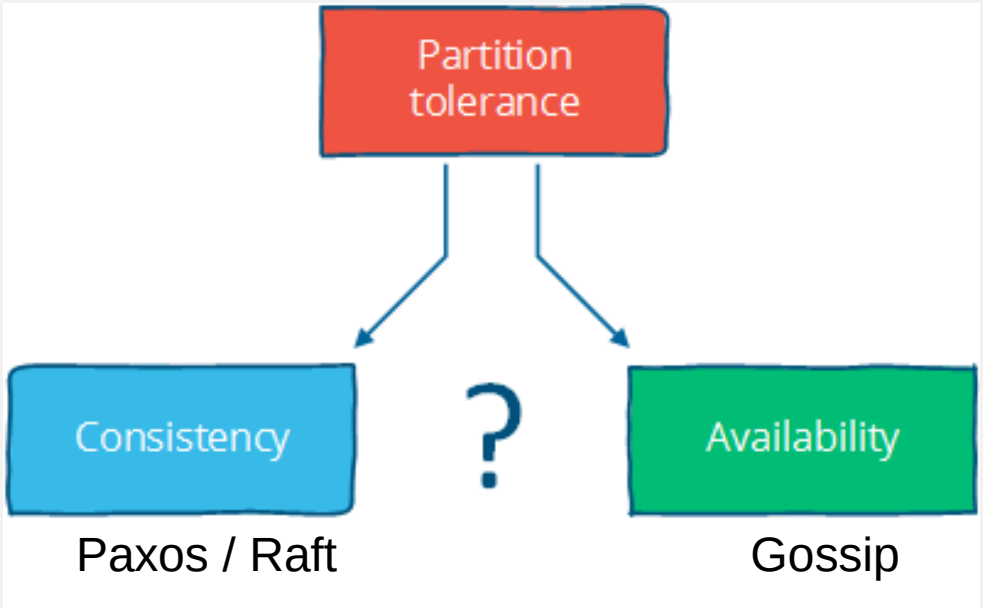
Nachteile:

- Zeitintensiv, da stets alle Knoten zustimmen müssen.
- Das System funktioniert nicht mehr, sobald das Netzwerk partitioniert ist.

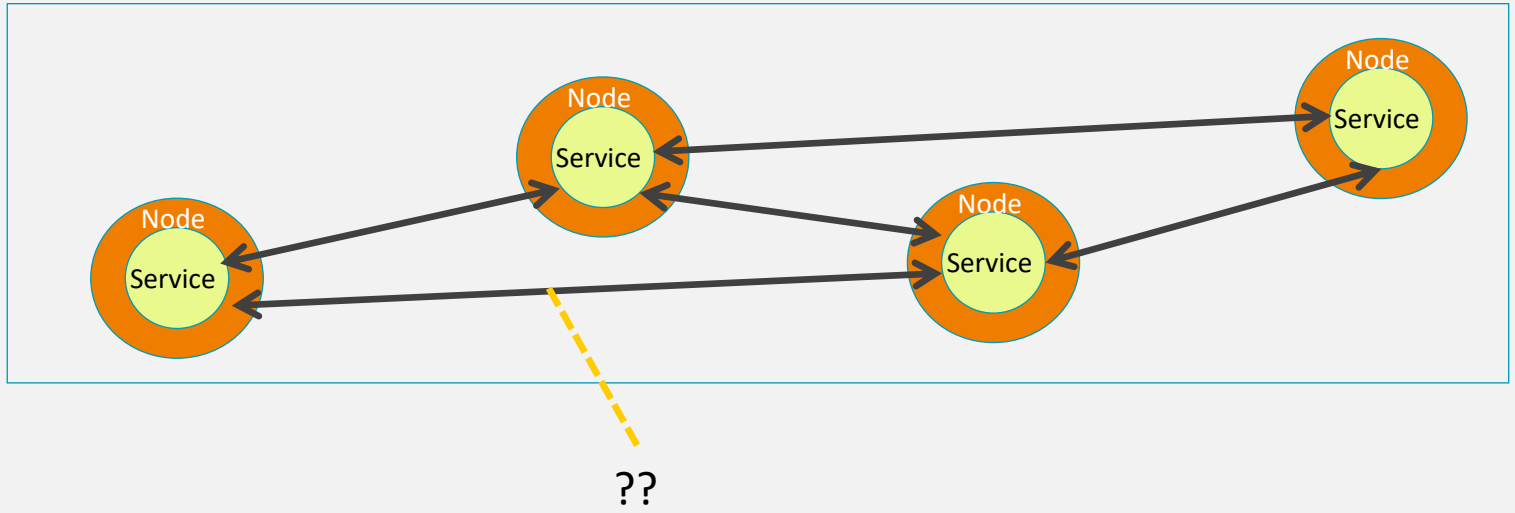
Die vorgestellten Protokolle und das CAP Theorem.



In der Cloud müssen Partitionen angenommen werden. Damit ist die Entscheidung binär zwischen Konsistenz und Verfügbarkeit.



Die Probleme einer klassischen Verknüpfung von Services in der Cloud.



Probleme:

- Mangelnde Redundanz: Jeder Service wird direkt genutzt. Er kann nicht unmittelbar in mehreren Instanzen laufen, die Redundanz schaffen.
- Mangelnde Flexibilität: Die Services können nicht ohne Seiteneffekt neu gestartet oder auf einem anderen Knoten in Betrieb genommen werden – oder sogar durch eine andere Service-Implementierung ausgetauscht werden.

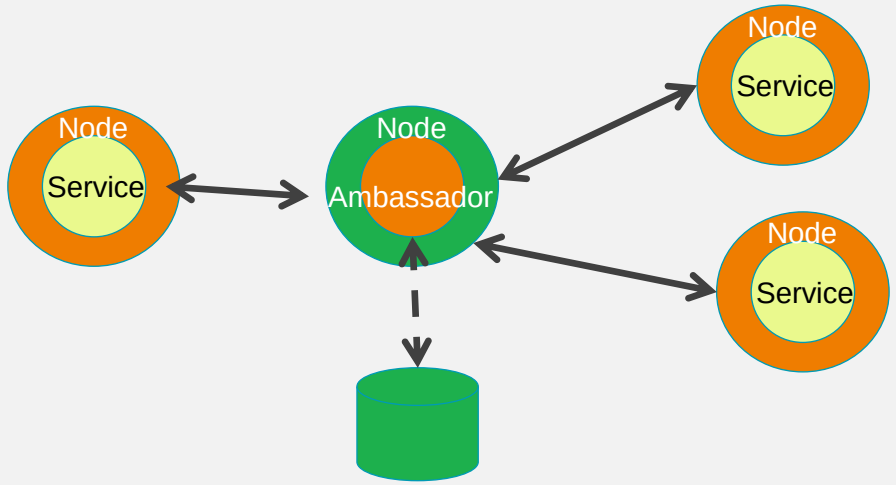
Lösungen:

- Dynamischer DNS
- Ambassador
- Dynamischer Konfigurationsdateien und Umgebungsvariablen

Das Ambassador Pattern

Ein Ambassador-Knoten für jede Knoten-Art (z.B. Webserver)

- **Service Registration**:
  - Beobachtet das Cluster und erkennt neue und kranke/tote Knoten in seiner Gruppe.
  - Hinterlegt die aktuell aktiven Knoten im Konfigurationsspeicher.
- **Service Discovery**: Der Client kommuniziert mit dem Ambassador-Knoten, der die Anfragen aber möglichst effizient an einen Knoten der Gruppe weiterreicht.



Der Ambassador-Knoten kann dabei eine Reihe an Zusatzdienste erweisen bei der Verbindung zum Service (**Service Binding**):

- Load Balancing inklusive Failover
- Service Monitoring
- Circuit Breaker Pattern
- Throttling