

Moderne Softwarearchitekturen

Dirk Leemans
Lösungsarchitekt

Mannheim, Februar 2020

Heydar-Aliyev-Kulturzentrum in Baku

Wer bin ich ?



Dirk Leemans
Freiberufler
Lösungsarchitekt

dirk.leemans@me.com

0170/874.38.93



Table of Contents

Inhaltsverzeichnis	6
1) Einleitung	8
2) Features, Szenarios und Storys.....	11
3) Die Domäne.....	42
a) Einen Sinn fürs Geschäft entwickeln	44
b) Domain Driven Design.....	46
c) Requirements Engineering	59
4) Software Qualitätsattribute.....	68
a) Qualitätsattribute	68
b) Maintainability	74
c) Usability	82
d) Availability.....	83
e) Portability	88
f) Interoperability	90
g) Testability	90
5) Resilienz.....	92
6) Softwarearchitektur Design.....	147
a) Software Architektur Design	148
b) Software Architektur Design	149
c) Attribute Driven Design.....	151
d) Microsofts Technik für Architektur und Design	158
e) Architecture Centric Design Method	167
f) Architecture Development Method	168
g) Den Fortschritt einer Software Architektur verfolgen.	176
7) Software Architektur Pattern.....	177
a) Was ist ein Software Architektur Pattern	177
b) Verschiedene Architekturen und Pattern.....	179
8) Moderne Architekturen	201
a) Monolithic Architecture	202
b) Microservice Architecture	209
c) Serverless Architecture.....	210
d) AWS Cloud-native Architekturstile.....	214
e) Microsoft Cloud Architekturstile	236
9) Microservices Architektur	295
10) Performance	357
a) Die Bedeutung von Performance	357
b) Performance Terminologien	359

c) Systematische Vorgehensweisen	365
d) Verschiedenste Strategien zur Verbesserung der Performance	370
11) Security	386
a) Zustände von Informationen	388
b) Das CIA Dreieck	389
c) Bedrohungsmodellierung	390
d) Secure by Design	392
e) Kryptografie	395
f) Identity and Access Management	396
g) Einige Risiken von Webanwendungen	400
12) DevOps und Softwarearchitektur.....	411
a) DevOps.....	412
b) DevOps Toolchain	420
c) DevOps Praktiken.....	421
d) DevOps und Architektur	424
e) DevOps und Cloud	426
13) Evolutionäre Architekturen	430
a) Änderungen sind unausweichlich	430
b) Lehmann's Gesetze der Software Evolution.....	431
c) Evolutionäre Architekturen entwerfen	436
14) Die Skills eines Softwarearchitekten	443
a) Was sind Softskills.....	443
b) Kommunikation	445
c) Führung	451
d) Verhandlungsgeschick	452
15) Wie werde ich ein guter Softwarearchitekt	453
16) Architektur und Legacy Applications.....	458
17) Anhang: Dokumentation	466

Referenzen

- Peter Reinhardt : Enterprise Architekt in Rente
 - Vorherige Version dieser Präsentation
- Gernot Starke : Effektive Software Architekturen (ISBN:978-3-446-46589-3)

1) Einleitung

Moderne Softwaresysteme sind extrem komplex

Die Rolle des Softwarearchitekten ist herausfordernd. Diese Vorlesung gibt eine Überblick über Software Architektur, über die Aufgaben und Verantwortlichkeiten eines Softwarearchitekten.

Softwarearchitekten benötigen technische und nicht-technische Skills, teilweise breit angelegtes Wissen, teilweise Wissen bis in die Details.

Die Vorlesung beginnt bei Anforderungen aus dem Business, der Arbeit in Organisationen und der Erfassung von Anforderungen.

Wir werden uns dann technischen Themen wie Software Qualität, Software Design, Best Practices, Patterns, Performance und Security widmen.

Nach der Vorlesung sollten Sie einen Überblick über die komplexe Welt der Softwarearchitektur besitzen.

Prüfung

Keine Klausur sondern

Persönliche schriftliche Vorbereitung Posterpräsentation : 15 Punkte

- 2, 3 Seiten
- Abgabetermin 28.03

Posterpräsentation : 25 Punkte

- Als Gruppe (persönliche Note)
- 6. Vorlesung

Persönliches Prüfungsgespräch : 20 Punkte

- 5 bis 10 Minuten

Mögliche Themen Posterpräsentation

Thema : eine konkrete Vorstellung von einer Architektur für ein selbstgewähltes Thema

Beispielen:

- Architektur für Web-Anwendungen
- Agile Architektur
- Architektur für Machine Learning
- Erodierte Software Architekturen modernisieren
- Domain Driven Design
- Service Oriented Architecture
- Hierarchische Architekturen

2) Features, Szenarios und Storys

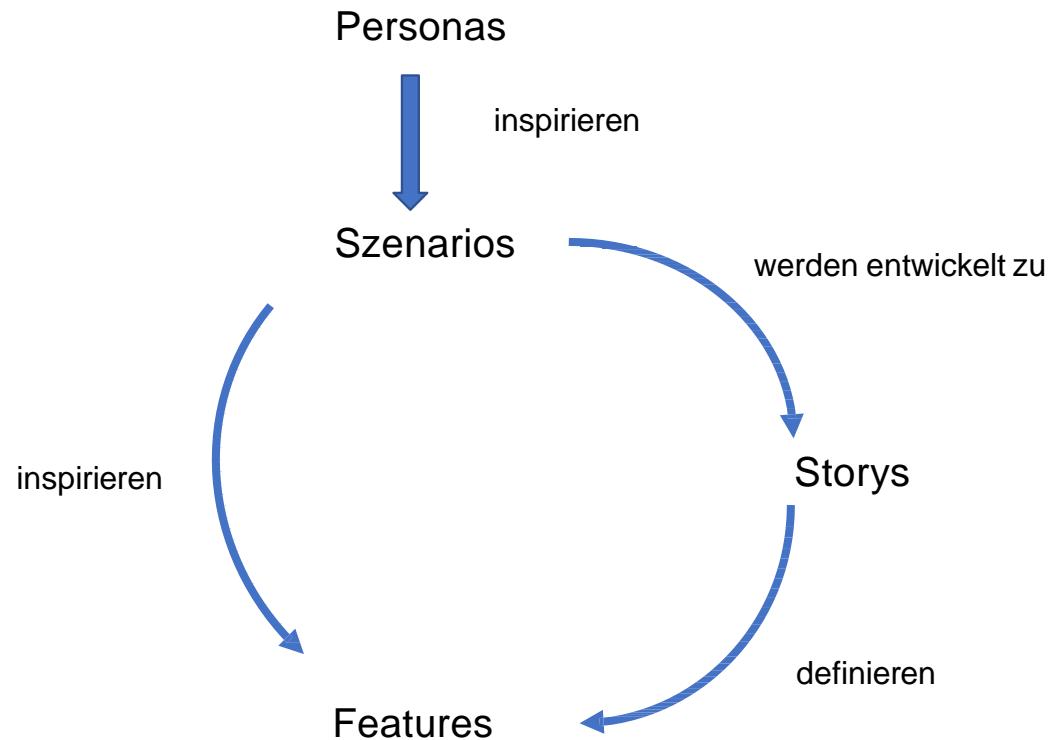
- Einige Softwareprodukte entstehen aus Visionen
- Wenige davon werden erfolgreich sein
- Die meisten erfolgreichen Produkte basieren auf dem Verständnis von Geschäftsmodellen und Benutzerproblemen

Wir werden folgendes betrachten

- a) Personas
- b) Szenarios
- c) User-Storys
- d) Features

2) Features, Szenarios und Storys

- Personas, Szenarios und User-Storys führen zu Features



2) Features, Szenarios und Storys

- Personas
 - Personas sind ein schönes und nützliches Instrument, mit dem man Zielgruppen besser verstehen und Entscheidungen für IT Projekte/Produkte treffen kann
 - Es gibt verschiedene Definitionen von „Personas“ im Internet (hier eine gebräuchliche)
 - Personas sind fiktive Charaktere
 - Sie dienen zur Darstellung von Zielgruppen mit ihren verschiedenen Benutzertypen (Archetype), die auf beobachteten Verhaltensmustern von realen Personen basieren
 - Jede Person (Persona) ist repräsentativ für ein Segment seiner Zielgruppe
 - Die zentralen Argumente für die Verwendung von Personas sind, dass sie Annahmen über Zielgruppen und/oder Zielenutzer explizit machen
 - Durch die Schaffung eines Charakters kann man mehr Interesse und Empathie gegenüber einer gesichtslosen Menge wecken
 - Eine Persona zeichnet ein Bild eines Typs von Benutzern

“It is easier to design for a specific somebody, rather than a generic everybody”

2) Features, Szenarios und Storys

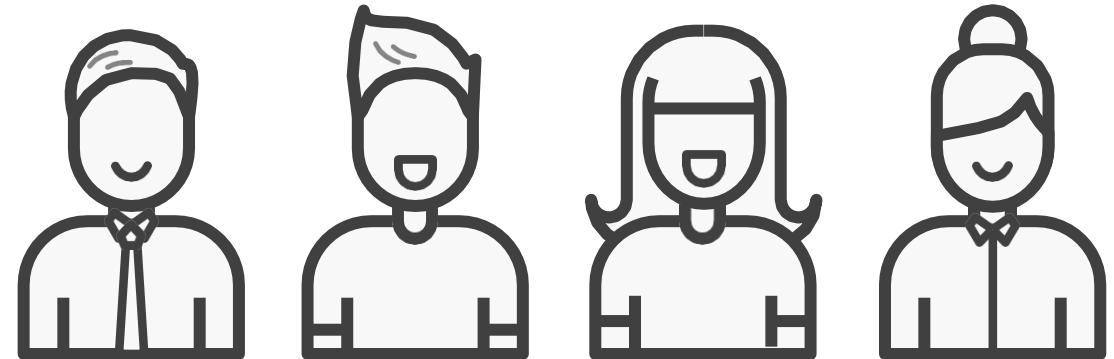
- Personas in der IT
 - Repräsentation einer Rolle in einer User Story
 - Fiktive Personen
 - Leitfrage: „Für wen entwickeln wir?“
 - Was bedeutet diese Person für meine Anwendung?
 - Personas helfen dabei, Wahrnehmung des Zielpublikums zu verstehen
 - Verschiedene Personas, die unterschiedliche Anforderungen und Wahrnehmungen haben
 - Hilfsmittel, um sich in einer Vorstellungswelt die möglichen Kunden vorzustellen und danach zu entwickeln
 - Der Benutzer ist nicht „Ich“
 - Der Benutzer hat immer recht

***You are not the user.
Neither is your boss.***

Jessyca Frederick
Director of Product Management, ThisNext

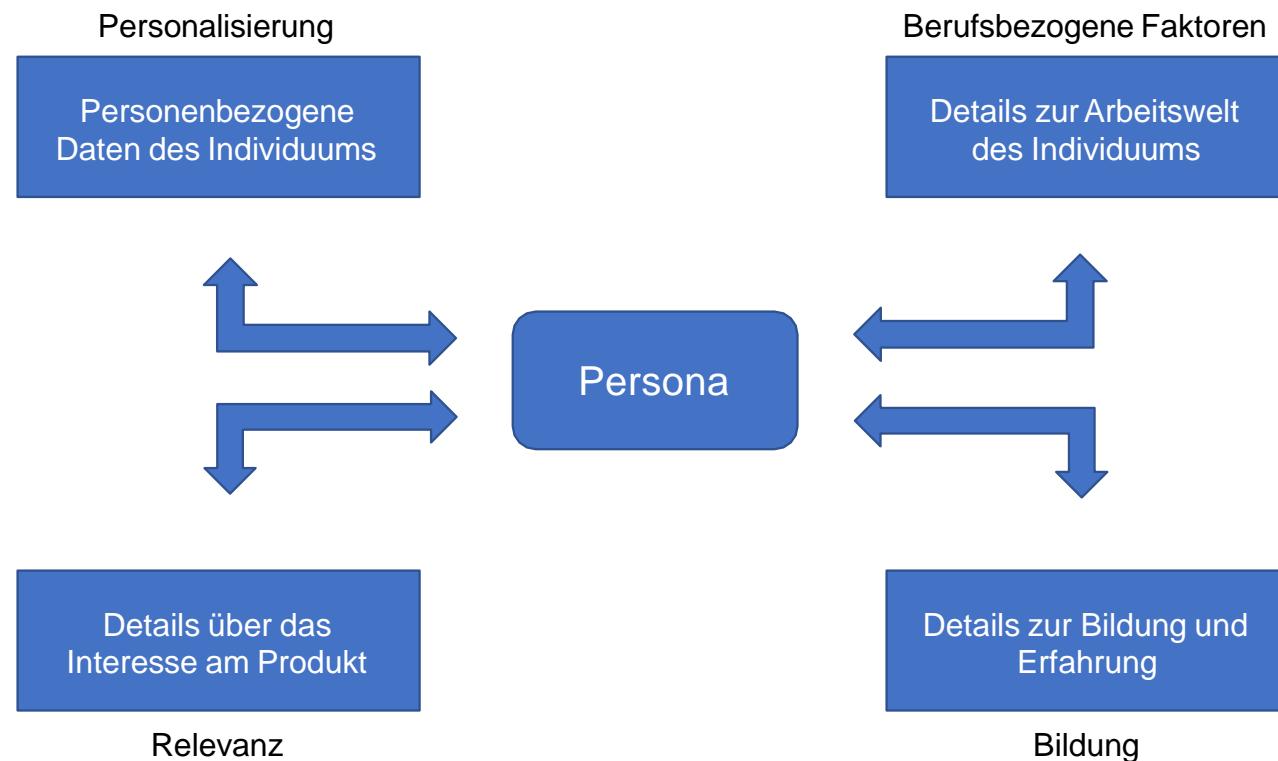
2) Features, Szenarios und Storys

- Personas (ganz konkret)
 - Potenzielle Zielgruppen einer Anwendung (ggf. auch „Gegner“)
 - Werden ganzheitlich-real beschrieben (Name, Alter, Foto, Lebensereignisse)
 - Haben Kontext / Leben (jenseits von IT/Architektur/Anwendung/Software)
 - Haben Erfahrungen, Fähigkeiten, mentale Modelle
 - Haben Ziele & Wünsche !



2) Features, Szenarios und Storys

- Personas (ganz konkret)



2) Features, Szenarios und Storys

- Personas

Bild	Name / Soziodemografische Daten	Rolle bezüglich des Kunden	Allgemeine Einstellung Einstellung zum Kunden	Nutzenerwartung an den Kunden	Internet/PC-Affinität	Präferenzen bezüglich des Kunden
	Paul Planer <ul style="list-style-type: none"> 52 Jahre Familienstand: geschieden (2 Kinder) Bildungsgrad: Hochschulabschluß (Dipl. Kfm.) Sprachen: Englisch/Französisch Berufserfahrung: 28 Jahre Marketing- und Bereichsleiter 	• Konzernbesteller	<ul style="list-style-type: none"> engagiert durchsetzungskraftig ungeuldig erwartet ein funktionierendes System beschwert sich, wenn etwas nicht seiner Erwartungen entspricht 	<ul style="list-style-type: none"> Ideen für Kampagnen bedarfsoorientierter Info-Bedarf detaillierter als Ernst Erstbesteller 	<ul style="list-style-type: none"> PC Erfahrung seit über 20 Jahren. Im Internet seit 9 Jahren Hohe Netzaffinität: kauft privat über Onlineshops und bucht Reisen online. Nutzt privat E-Mails und RSS-Feeds (Financial Times und Manager Magazin) 	<ul style="list-style-type: none"> Einheitliche Struktur Verwaltungsfunktion Effizienter Bestellprozess
	Inga Immerbesteller <ul style="list-style-type: none"> 38 Jahre Familienstand: verheiratet (2 Kinder) Bildungsgrad: mittlere Reife Sprachen: Englisch/Französisch Berufserfahrung: 20 Jahre 13 Jahre: Einkaufssachbearbeiterin 	• Konzernbestellerin mit sehr vielen Bestellungen	<ul style="list-style-type: none"> engagierte Mitarbeiterin hilfsbereit macht Verbesserungsvorschläge zur Applikation 	<ul style="list-style-type: none"> kurze Bestellwege online Schnellerfassung Bestellung aus Katalog 	<ul style="list-style-type: none"> PC-Erfahrung seit über 20 Jahren, im Internet seit 7 Jahren Kennt sich im Web aus, bestellt ab und an bei Amazon oder schaut sich auf Ebay um 	<ul style="list-style-type: none"> Auftragsstatus Bestellhistorie Bestellfunktionalität insgesamt Kontaktmöglichkeit
	Gerrit Gelegenheit <ul style="list-style-type: none"> 48 Jahre Familienstand: verheiratet (4 Kinder) Bildungsgrad: Dipl. Ing. Sprachen: Englisch / Italienisch / Französisch Berufserfahrung: 23 Jahre seit 4 Jahren Geschäftsführer 	<ul style="list-style-type: none"> Erst- oder Wenig-Besteller keine / wenig Erfahrung mit dem Kunden Preisvergleicher 	<ul style="list-style-type: none"> ruhig, ausgeglichen vielfältige Interessen 	<ul style="list-style-type: none"> Orientierung und Leitung beim Surfen durch die Applikation Günstige Konditionen Prestige 	<ul style="list-style-type: none"> Umfangreiche PC-Erfahrung Nutzt das Internet zu Informations- und Unterhaltungszwecken (Informationen sind für ihn auch Unterhaltung) 	<ul style="list-style-type: none"> Überzeugende Gestaltung Erfahrung Günstige Konditionen Klare und umfangreiche Informationen Entscheidungshilfen

2) Features, Szenarios und Storys

- Personas



2) Features, Szenarios und Storys

- Personas
 - Anleitung

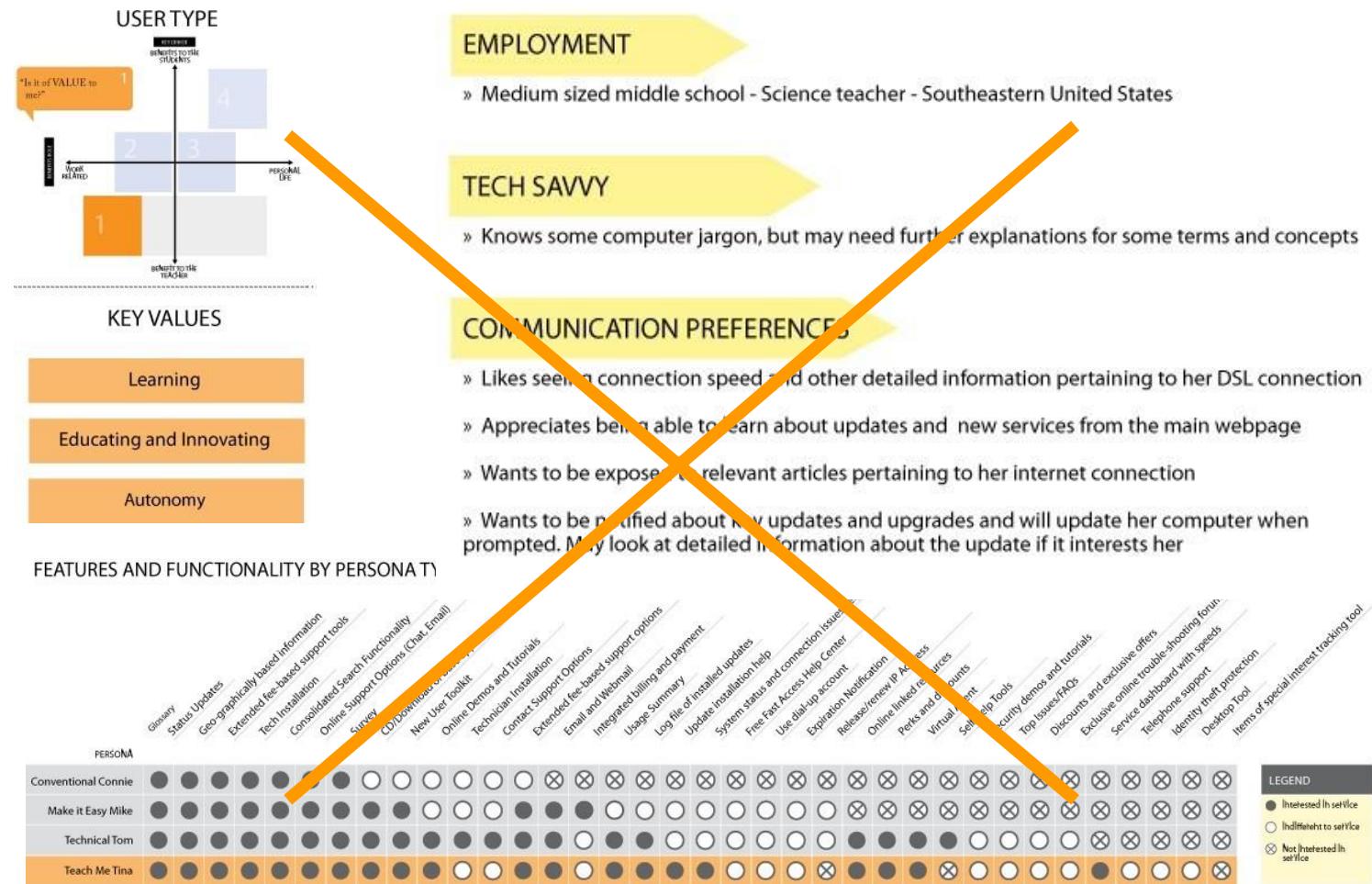
Sinnvolle Anzahl verwenden



2) Features, Szenarios und Storys

- Personas
 - Anleitung

Nicht „over-engineeren“



2) Features, Szenarios und Storys

- Personas
 - Achtung
 - Anstatt „Statistische“ Personas werden Design Personas genommen
 - Zu viele (> 8) oder zu wenige (< 3) Personas
 - Kein angemessener Detaillierungsgrad der persönlichen Beschreibungen
 - Kein „Build from Scratch“: Nie (!) Recycling von Personas aus anderen Projekten

Generieren Sie Personas im Rahmen einer Anforderungsanalyse auf Basis von quantitativen und qualitativen Daten, um potenzielle Zielgruppen zu erkennen und optimal beschreiben zu können.

Versuchen Sie nicht, auf eine ungefilterte, zu breit gestreute Zielgruppe anzupassen.

So besteht die Gefahr, ein durchschnittliche(s) Produkt/Architektur zu entwickeln.

Konzentrieren Sie sich auf wenige, exakt definierte Personas.

Vorlage: Persona-Profil [NAME + TYP]

Hintergrund zur Person: (Beruf, Karriere, Bildung, Familie)

- Wie sieht der/die typische/r Vertreter/in dieser Kundengruppe aus?
- Welchen Beruf übt der Vertreter aus?
- Wie sind die familiären Verhältnisse?
- Was ist der Person im Leben wichtig?

Demographie:

- Alter
- Geschlecht
- Wohnort
- Wohnverhältnis

Foto:

Wie sieht der/die typische/r Vertreter/in der Gruppe aus?



Identifikatoren:

- Was macht die Persona aus (z. B.: Hobbys, Interessen)?
- Wie ist ihr Auftreten?
- Welches sind ihre bevorzugten Kommunikationskanäle?
- Wie ist das Informationsverhalten (On-/Offline & welche Kanäle)?
- Wie ist das Einkaufsverhalten (On-/Offline)?
- Wer übt Einfluss auf die Persona aus (Freunde, Arbeitskollegen, Vorbilder etc.)?

Erwartungen, Ziele & Emotionen:

- Was möchte diese Persona mit dem Kauf erreichen?
- Welche Probleme will sie lösen?
- Welchen Nutzen will sie erzielen?
- Und welche Gefühle könnten dies alles begleiten?
- Welche Ängste könnte sie haben?
- Und was könnte sie ganz besonders begeistern?

Herausforderungen:

- Welche Herausforderungen treten für die Persona bei der Kaufentscheidung bzw. Anbieter- & Produktauswahl auf?
- Womit hat sie zu kämpfen?
- Was fällt ihr schwer?

Ideale Lösung:

- Wie können wir der Persona helfen, die Herausforderung zu meistern?
- Wie können wir ihre Erwartungen übertreffen?
- Mit welchen Emotionen können wir die Persona abholen?
- Wie helfen wir, dass sie ihre Ziele erreicht?

Häufige Einwände:

- Warum würde die Persona unser/e Produkt/Dienstleistung nicht kaufen?
- Welche Gegenargumente können auftreten?
- Was könnte sie stören oder verunsichern?

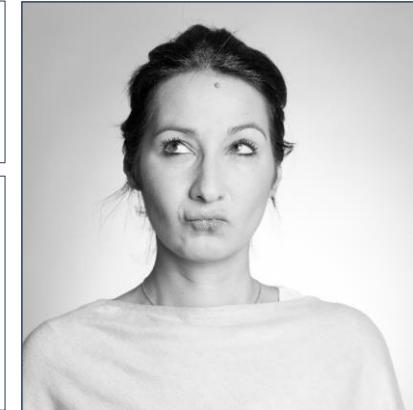
Persona-Beispiel: Sophia, Marketingleiterin

Hintergrund zur Person:

- Sophia ist eine erfolgsorientierte Marketingleiterin.
- Sie hat BWL studiert und arbeitet nach diversen Stationen im Ausland seit zwei Jahren in der Firma.
- Sie möchte erfolgreich im Beruf sein.

Demographie:

- Weiblich
- 42 Jahre alt
- Vorort von Köln
- Wohnung, sie lebt alleine



Identifikatoren:

- Sophia geht neben der vielen Arbeit boxen. Außerdem geht sie für ihr Leben gern tanzen und singen.
- Sie achtet auf ihr Äußeres und geht gerne Markenklamotten shoppen.
- Sie informiert sich intensiv über das Internet, bevorzugt Käufe aber in Geschäften bzw. nur nach direktem Gespräch mit einem Berater im Geschäft / beim Anbieter.
- Sie ist auf Facebook und Instagram aktiv.
- Große Kaufentscheidungen bespricht sie mit ihrer Familie und Freunden.

Erwartungen, Ziele & Emotionen:

- Sie braucht einen neuen Laptop für ihren Job. Ihre Firma lässt ihr bei der Auswahl freie Wahl.
- Er soll schnell sein, eine leise Tastatur und eine gute Bildschirmauflösung haben.
- Da es so eine große Auswahl im Markt gibt, ist sie ein bisschen verunsichert.

Herausforderungen:

- Es ist das nicht erste Mal, dass Sophia einen Laptop kauft. Allerdings ist es das erste Mal, dass ihr Arbeitgeber freie Hand lässt. Das Budget ist generös, gerade deswegen möchte sie keinen Fehlkauf machen.
- Sie weiß, was GBs und RAM ist, aber darüber hinaus sieht sie sich nicht als Computer-Freak. Am liebsten würde sie die Entscheidung jemand anders überlassen.
- So bald wird sie keinen neuen bekommen, also muss dieser für die nächsten fünf Jahre halten.

Ideale Lösung:

- Wir können Sophia die Sicherheit geben, dass unsere Produkte von guter Qualität sind. Allerdings sind sie auch preislich im gehobenerem Segment. Unsere guten Bewertungen auf unseren Produktseiten und allgemein im Netz werden sie hoffentlich überzeugen.
- Unsere Produktentwickler haben lange an unserer Tastatur getüftelt, damit sie leise ist. Diese Info scheint aber noch nicht im Markt angekommen zu sein. Und wie bekommen wir Sophia in unser Ladengeschäft, damit sie es erfährt?

Häufige Einwände:

- Preis, Preis, Preis. Trotz großzügigem Budget möchte Sophia sich nicht über den Tisch ziehen lassen. Off- und online gibt es manchmal große Preisunterschiede.
- Sie möchte an ihrem Feierabend nicht 20 Fachhändler besuchen, um ihre Wahl zu treffen.

Deine Persona:

Hintergrund zur Person:

Demographie:

Foto:

Identifikatoren:

Erwartungen, Ziele & Emotionen:

Herausforderungen:

Ideale Lösung:

Häufige Einwände:

Beispiel Persona

Emil Elektro



“Flexibilität im Alltag ist mir wichtig, soll aber nicht auf Kosten der Umwelt geschehen. Ich wünsche mir, dass die Automobilindustrie endlich ein Elektroauto entwirft, dass in Punkt Umweltbilanz, Reichweite und Design neue Standards setzt.”

Demografika:

- männlich, 32 Jahre alt
- Marketing-Spezialist
- wohnt in Köln

Hintergrund:

- interessiert sich für Klimaschutz
- wählt die Grünen
- hohe Kaufkraft

Hobbies und Interessen:

- geht gerne auf Messen
- macht viel Sport
- schaut gerne Serien online

Ziele:

- möchte keine Trends verpassen
- will im Alltag mobil sein
- möchte umweltbewusst leben

Probleme:

- hat Schwierigkeiten, seine Konsumbedürfnisse mit nachhaltigen Angeboten zu decken
- möchte gerne ein Auto besitzen, damit aber die Umwelt und das Klima nicht belasten

Top-Medien:

SPIEGEL ONLINE

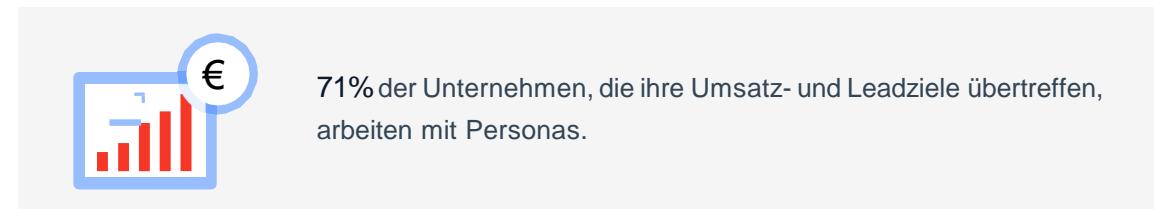
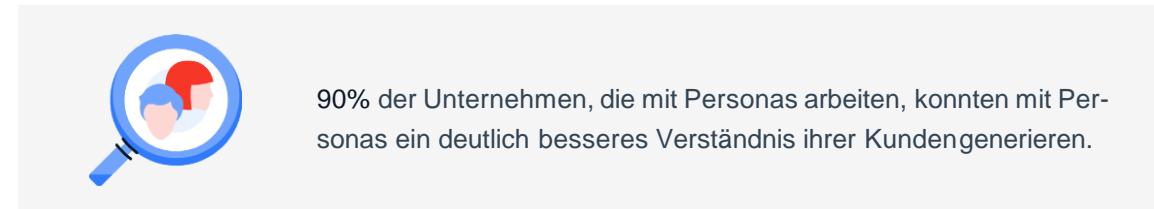


NETFLIX



2) Features, Szenarios und Storys

- Personas
 - Vorteile
 - Einsetzbar bei der Konzipierung, aber auch zur Evaluation eines Systems
 - Präsenz von Personas erleichtert die Kommunikation im Entwicklungsteam
 - Fokus auf den Nutzer wird gestärkt, seine Anforderungen werden sichtbar gemacht



2) Features, Szenarios und Storys

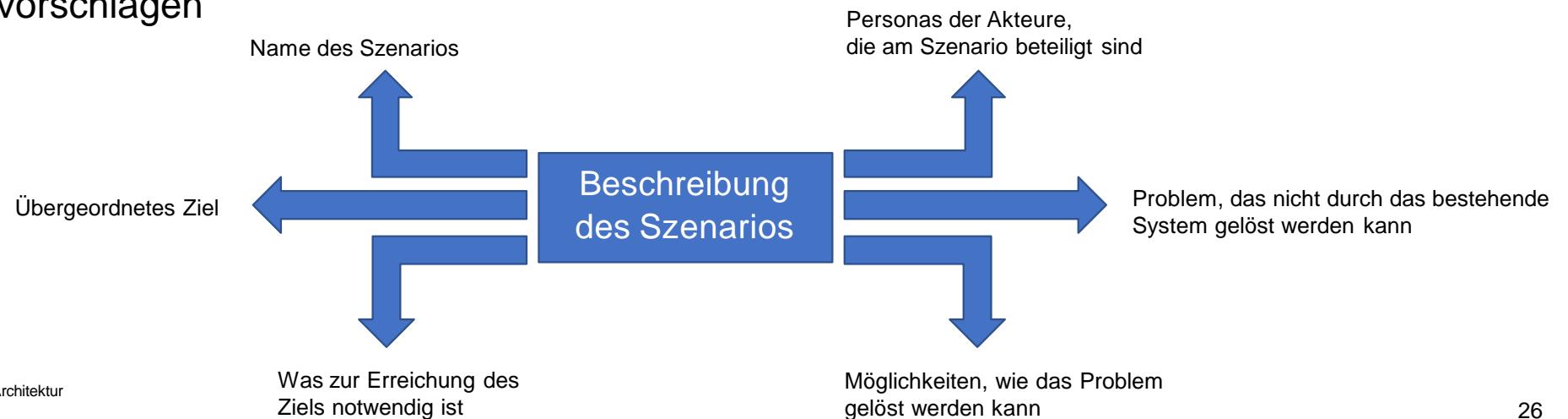
- Personas
 - Checkliste
 - Personas beschreiben keine realen Personen – sie basieren aber auf (recherchierten) Informationen über eine Zielgruppe
 - Eine Persona wird erzählend beschrieben, eine bis maximal zwei Seiten genügen
 - Vermeiden Sie, lediglich Aufgaben aufzulisten – eine Persona ist viel mehr!
 - Beschreiben Sie, wie ein typischer Tag Ihrer Persona ablaufen könnte
 - Welche Eigenschaften hat Ihre Persona, was sind Ihre Vorlieben, wer lebt und arbeitet in Ihrem Umfeld?
 - Was wäre eine typische Aussage Ihrer Persona?
 - Machen Sie Ihre Persona für alle im Team sichtbar, z.B. durch Flyer.
 - Fotos der Personas oder markante Namen für diese können helfen, die Akzeptanz von Personas im Team zu erhöhen – und damit die Wahrscheinlichkeit, dass diese verwendet werden.

2) Features, Szenarios und Storys

- Personas
 - Einen guten Einstieg in das Thema „Personas“ liefert: Pruitt, J., & Tamara, A. (2006). *The Persona Lifecycle: Keeping people in mind throughout product design*. Amsterdam: Elsevier.
 - Drei Punkte, warum sich Personas lohnen, etwas ausführlicher beschrieben:
http://www.uie.com/articles/benefits_of_personas/ [16.04.2018]
 - Eine Anleitung zur Erstellung von Personas in sechs Schritten findet sich auf:
<https://www.gruenderszene.de/allgemein/persona-personasentwickeln> [16.04.2018]
 - Auf dieser Seite finden sich einige Beispielfragen, die man sich bei der Erstellung von Personas stellen kann: <https://www.usability.gov/how-to-and-tools/methods/personas.html> [16.04.2018]
 - Eine gute Anleitung findet man unter : <https://www.netspirits.de/blog/personas-erstellen/>

2) Features, Szenarios und Storys

- Szenarios
 - Zum Definieren von Produktmerkmalen
 - Erleichtert die Auswahl und die Gestaltung von Funktionalitäten (Features)
 - Unterstützt die Vorstellung der Interaktion zwischen Benutzer (Persona) und Produkt
 - Ist eine Schilderung einer Situation, in der sich eine Anwender befindet, wenn er mit dem Produkt ein Problem lösen will
 - Ein Szenario sollte das Problem des Benutzers kurz erklären und Lösungswege vorschlagen



2) Features, Szenarios und Storys

- Szenarios
 - Die wichtigsten Elemente
 - Eine kurze Darstellung des Gesamtziels
 - Verweise auf die beteiligten Personas mit Hinweise auf Fähigkeiten und Motivationen
 - Informationen zur Durchführung der Aktivitäten
 - Erörterung der Probleme, die mit dem bestehenden System nicht ohne Weiteres behoben werden können
 - Beschreibung einer Möglichkeit das identifizierte Problem anzugehen
 - Muß nicht immer vorkommen, da zu technisch oder Informationen nicht vorliegen

2) Features, Szenarios und Storys

- Szenarios
 - Anfertigen von Szenarios
 - Ausgangspunkt sind die Personas (alle Rollen abdecken !)
 - Pro Persona sollten mehrere Szenarios angelegt werden
 - Regen eher zum Nachdenken an und liefern keine komplette Beschreibung des Systems
 - Szenarios sollten immer aus der Sicht des Anwenders verfasst sein
 - Szenarios sollten immer auf Personas oder realen Nutzern basieren
 - Szenarios sollten sich auf Ziele konzentrieren (was will der Benutzer tun) und nicht zu stark auf die Mechanismen die angewendet werden (wie)
 - Szenarios können spezifischen Details einer Interaktion beinhalten, weil die Beschreibung wichtig für die Lösung sein könnte
 - Szenarios können in (mehreren) kleinen eigenständigen Teams entwickelt werden
 - In größerer Runde werden die Szenarios besprochen und verfeinert
 - Es macht Sinn die Benutzer mit einzubinden (beim Erstellen oder Besprechen)
 - Wenn Benutzer ihre eigenen Szenarios erstellen, besteht die Gefahr, dass sie das IST beschreiben

2) Features, Szenarios und Storys

- User-Storys
 - User-Storys sind Schilderungen, die mehr ins Detail gehen
 - Stellen einzelne Aspekte, die ein Benutzer vom Softwaresystem erwartet, ausführlicher und strukturierter dar

„Als Autor brauche ich eine Möglichkeit, den Text meines Buches, in Kapitel und Abschnitte zu organisieren, um die Übersicht zu behalten.“

Typisches Standardformat einer User-Story :

„Als <Rolle>< möchte/will/muss> ich <etwas tun>, damit <Grund>.“

- Die Begründung kann helfen, alternative Wege zu sehen
- Product-Backlogs in Scrum (agile Entwicklung) sind oft eine Reihe von User-Storys
- Für den sinnvollen Einsatz von User-Storys in Backlogs sollten sich Storys auf klar definierte Funktionen innerhalb eines Sprints konzentrieren
- Komplexere Storys (über mehrere Sprints) werden in Scrum „Epic“ genannt

2) Features, Szenarios und Storys

- User-Storys
 - Sollten hilfreich sein
 - Als Möglichkeit, ein Szenario zu erweitern und Details hinzuzufügen
 - Als Teil der Beschreibung der identifizierten System-Features
- Was unterscheidet User-Storys von Szenarios (manche agile Methoden verzichten auf Szenarios)
 - Szenarios lesen sich natürlicher, weil sie beschreiben, was ein Benutzer eines Systems wirklich mit dem System macht – in User-Storys sind mehr Wünsche und Bedürfnisse beschrieben
 - Reale Nutzer sprechen nicht in der stilisierten Sprache von User-Storys. Die Schilderungen in den Szenarios wirken natürlicher
 - Szenarios liefern oft mehr Kontextinformationen (was Benutzer tun und wie sie arbeiten). Kontextinformationen würden User-Storys zu stark aufblähen

2) Features, Szenarios und Storys

- Features
 - Ziel im Entwurf ist die Erstellung einer Liste von Funktionen, die das Softwareprodukt genau beschreiben
 - Ein Feature ist eine Möglichkeit, Benutzern den Zugriff und die Nutzung der Funktionalitäten ihres Produkts zu ermöglichen
 - Die Liste der Features beschreibt die Gesamtfunktionalität des Systems/Produkts
 - Im Idealfall sind Features
 - Unabhängig – ein Feature sollte nicht davon abhängen, wie andere Features implementiert werden und sollte nicht von der Reihenfolge der Aktivierung anderer Features beeinflusst werden
 - Kohärent – Features sollten mit einem einzigen Element der Funktionalität verknüpft sein. Sie sollten nicht mehr als eine Sache tun und das ohne jegliche Seiteneffekte
 - Relevant – die Funktionen sollten die Art und Weise widerspiegeln, wie Benutzer normalerweise eine Aufgabe ausführen. Sie sollten keine Funktionalität bieten, die selten benötigt wird

2) Features, Szenarios und Storys

- Features
 - Beim Entwurf von Features sind die folgenden Faktoren wichtig

Quelle	Beschreibung
Anwenderwissen (Wissen über den Anwender)	Benutzerszenarios und User-Storys helfen zu verstehen, was Benutzer wünschen und wie sie die Funktionen nutzen können
Produktwissen	Erfahrung mit Produkten oder Rolle von Produkten in Entwicklungsprozessen
Fachbereichswissen	Kenntnis des Fach- oder Arbeitsbereichs – nur wer den Fachbereich kennt, kann sich innovative neue Wege vorstellen
Technologiewissen	Technologische Neuerungen zu Nutze machen können

2) Features, Szenarios und Storys

- Features
 - Die 6 Faktoren beim Entwurf von Features (Kompromisse oft notwendig)
 - Einfachheit (einfach zu bedienen)
 - hohe Funktionalität (kann mehr)
 - Vertrautheit (kenne mich sofort aus)
 - Neuartigkeit (oh, so geht das auch !)
 - Automatisierung (muss ich nicht tun)
 - Kontrolle (will ich aber nachvollziehen können)

2) Features, Szenarios und Storys

- Wie komme ich zu Features
 - Siehe Ausarbeitung auf den nächsten Folien
 - Ausgangspunkt : Persona
 - Beschreibung eines Szenario
 - Features im Szenario rot markiert
 - Liste der Szenarios

Jack Grundschullehrer

Hintergrund zur Person: (Beruf, Karriere, Bildung, Familie)

- Grundschullehrer
- Vater hat eine Heizölfirma, Mutter ist Krankenschwester am KliLu
- Er unterrichtet gerne und möchte in der Schule neue Technologien einführen

Demographie:

- 32 Jahre
- männlich
- Mannheim



Identifikatoren:

- Autor von Webseiten für eine Freizeitgruppe
- Freundlich und aufgeschlossen
- Nutzt Facebook und WhatsApp
- Hat keinen Festnetzanschluss, kommuniziert ausschließlich über Mobile
- Ist mit digitalen Themen vertraut
- Bestellt einiges Online, ist aber auch oft in lokalen Geschäften zum Einkaufen
- Ein Vorbild ist Bill Gates

Erwartungen, Ziele & Emotionen:

- Er ist fest davon überzeugt, dass der Einsatz digitaler Technologien zusammen mit Präsenzlehre die Lernerfahrung von Kindern verbessern kann
- Er benötigt ein Lernsystem für den projektbezogenen Unterricht
- Fächerübergreifendes Arbeiten seiner Schüler an einem gemeinsamen Thema

Herausforderungen:

- Nicht alle Kollegen sehen das genauso
- Die Schule hat wenig Budget (das meistens schon verplant ist)
- Er möchte nicht um Budget betteln müssen

Ideale Lösung:

- ?

Häufige Einwände:

- Zu teuer
- Präsenzlehre ist viel effektiver

Szenario

Jack ist Grundschullehrer in Mannheim. Er hat beschlossen, ein Klassenprojekt in der 6. Klasse zu starten, das die Fischereibranche in Mannheim zum Thema hat. Es sollen die Geschichte, die Entwicklung sowie die wirtschaftliche Bedeutung der Fischerei betrachtet werden.

Als Teil des Projekts werden die Schüler aufgefordert, Erinnerungen von Verwandten zusammenzutragen, Zeitungsarchive zu benutzen und alte Fotos zu sammeln, die mit Fischerei und Fischergemeinden der Gegend zu tun haben. Die Schüler nutzen ein Wicki, um Geschichten zur Fischerei zusammenzutragen sowie eine Webseite für historische Archive, um auf Zeitungsarchive und Bilder zuzugreifen. Jack benötigt zusätzlich eine Webseite zum Foto-Sharing, weil er möchte, dass die Schüler Fotos aufnehmen und gegenseitig kommentieren sowie Scans von alten Fotos hochladen, die sie in ihren Familien gefunden haben.

Jack will Beiträge mit Fotos moderieren, bevor sie veröffentlicht werden, da Kinder nicht die Datenschutz- und Urheberrechtsproblematik verstehen.

Kollegen empfehlen ihm ein bereits bestehendes System „Pics4Kids“. Andere Kollegen halten Moodle für die Anforderungen geeignet.

Features

Jack ist Grundschullehrer in Mannheim. Er hat beschlossen, ein Klassenprojekt in der 6. Klasse zu starten, das die Fischereibranche in Mannheim zum Thema hat. Es sollen die Geschichte, die Entwicklung sowie die wirtschaftliche Bedeutung der Fischerei betrachtet werden.

Als Teil des Projekts werden die Schüler aufgefordert, Erinnerungen von Verwandten zusammenzutragen, Zeitungsarchive zu benutzen und alte Fotos zu sammeln, die mit Fischerei und Fischergemeinden der Gegend zu tun haben. Die Schüler nutzen ein **Wicki**, um Geschichten zur Fischerei zusammenzutragen sowie eine **Webseite für historische Archive**, um auf Zeitungsarchive und Bilder zuzugreifen. Jack benötigt zusätzlich eine **Webseite zum Foto-Sharing**, weil er möchte, dass die Schüler **Fotos aufnehmen** und **gegenseitig kommentieren** sowie **Scans von alten Fotos hochladen**, die sie in ihren Familien gefunden haben.

Jack will **Beiträge mit Fotos moderieren**, bevor sie veröffentlicht werden, da Kinder nicht die Datenschutz- und Urheberrechtsproblematik verstehen.

Kollegen empfehlen ihm ein bereits bestehendes System „**Pics4Kids**“. Andere Kollegen halten **Moodle** für die Anforderungen geeignet.

Features

- Ein Wicki
- Webseite für historische Archive (oder Zugang dazu)
- Webseite zum Foto-Sharing
- Fotos aufnehmen
- Fotos gegenseitig kommentieren
- Scans von alten Fotos hochladen
- Beiträge mit Fotos moderieren
- Integration in „Pics4Kids“ oder Moodle

2) Features, Szenarios und Storys

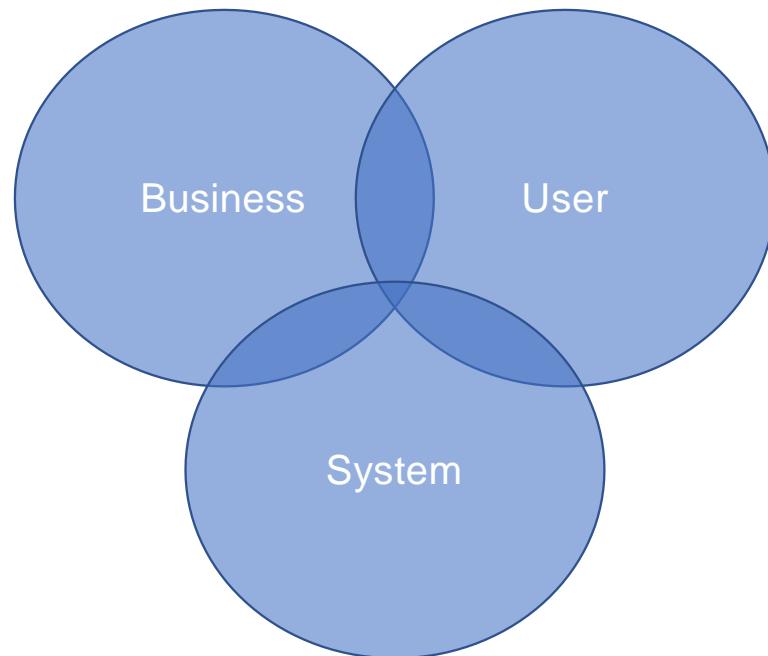
- Zusammenfassung
 - Ein Feature ist ein Fragment der Gesamtfunktionalität, die ein Benutzer benötigt
 - Features stehen in einer Liste mit kurzer Beschreibung
 - Personas sind imaginäre Nutzer – Portraits von Benutzertypen
 - Die Beschreibung einer Persona sollte ein Bild eines typischen Produktanwenders zeichnen
 - Ein Szenario ist eine Beschreibung wie die Benutzer mit dem Produkt arbeiten wollen
 - Szenarios sollten immer aus der Sicht des Benutzer geschrieben werden und auf Personas basieren
 - User-Storys sind Schilderungen, die mehr im Detail und strukturiert die Erwartungen der Benutzer beschreiben
 - User-Storys können Szenarios erweitern und Details hinzufügen
 - Die wichtigsten Einflussfaktoren bei der Identifizierung und dem Entwurf von Features sind Benutzerforschung, Fachbereichswissen, Produktwissen und Technologiewissen
 - Features lassen sich durch Markierung in Szenarios und User-Storys identifizieren

3) Die Domäne

- Es gibt Business Domänen und technische Domänen (Wirkungsraum)
- Ein Architekt muss beide beherrschen
- Das Domänenmodell hilft dabei
- Wir werden folgendes betrachten
 - a) Einen Sinn fürs Geschäft entwickeln (business acumen)
 - b) Domain Driven Design (DDD)
 - c) Anforderungs (Requirements) Engineering
 - d) Anforderungs Erhebung

a) Einen Sinn fürs Geschäft entwickeln

- Ein Software Architekt muss die Anforderungen/Ziele/Einschränkungen zusammenbringen



a) Einen Sinn fürs Geschäft entwickeln

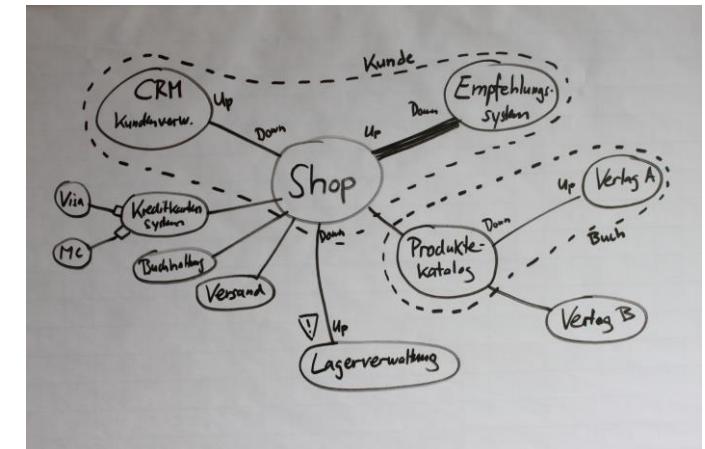
- Ein Verständnis fürs Geschäft und der Fachsprache macht den Architekten komplett
- Architekten profitieren mehr als z.B. Softwareentwickler von diesem Verständnis
- Softwarearchitekten stehen dauernd im Austausch mit dem Business und ein gemeinsames Verständnis und eine gemeinsame Sprache ist deshalb mehr als hilfreich
- Den Hut des Benutzers aufsetzen zu können, hilft bei der Umsetzung von Zielen und Anforderungen
- In größeren Unternehmen gibt es oft Architekten für Business-Bereiche wie Finanzen, Personalwesen, Produktion, Logistik u.a.
- Ein Verständnis für Entscheidungen basierend auf ROI (Return on Invest) Konzepten bringt Mehrwert in entsprechende Diskussionen
- Erreicht werden kann dieses Wissen entweder durch Mitarbeit im Business (Jobrotation), Selbststudium, Literatur oder Recherchen

a) Einen Sinn fürs Geschäft entwickeln

- Schwerpunkte
 - Produkte und Dienstleistungen
 - Womit verdient mein Unternehmen Geld ?
 - Verständnis der Business Prozesse
 - Marktumfeld
 - Konkurrenten
 - Unterschiede / Gemeinsamkeiten
 - Stärken / Schwächen
 - Kunden (wichtigster Aspekt überhaupt)
 - Business des Kunden
 - Warum nutzen die Kunden die Produkte / Dienste
 - Märkte
 - Was zeichnet die Märkte aus ?

b) Domain Driven Design

- Der Versuch Anforderungen in logische Bereiche und Unterbereiche aufzuteilen
- Methoden wie Bounded Context, Context Map, Context Mapping Patterns helfen dabei
 - Bounded context = zusammengehöriger Teilbereich im Domänenmodell
- Eine sog. „ubiquitous (universal, common) language“ dient zu Beschreibung von Domänen und Zusammenhängen/Beziehungen
 - Man beginnt am besten mit sog. Business Capabilities
 - Man definiert sog. Entitäten, Wertobjekte und Aggregate
- Man zeigt Informationsflüsse zwischen und in Bounded Contexts
- Architekturprinzipien und –muster wie „Lose Kopplung, Single Responsibility, Event Sourcing oder CQRS“ kommen zum Ansatz



b) Domain Driven Design

- Wie kann DDD helfen
 - Durch Reduktion des Übersetzungsaufwands
 - Gleiche Begriffe in Domäne und Sourcecode
 - Klare Modellierung der Fachlichkeit
 - Möglichst identische Konzepte
 - Durch Beschreiben nützlicher Methoden und Muster
 - Strategisches DDD: Analyse, Dokumentation und Abgrenzung der Domäne
 - Taktisches DDD: Umsetzung der Erkenntnisse in Sourcecode

b) Domain Driven Design

- Ubiquitous Language
 - Gleiche Begriffe in Domäne und Sourcecode
 - Jede Domäne besitzt eine eigene Fachsprache
 - Verstehen, warum diese Fachsprache gesprochen wird
 - Nicht versuchen, diese Fachsprache in „eigene“ Begriffe zu übersetzen
 - Ubiquitous Language bezeichnet die von Domänenexperten und Entwicklern gemeinsam im Projekt verwendete Sprache

b) Domain Driven Design

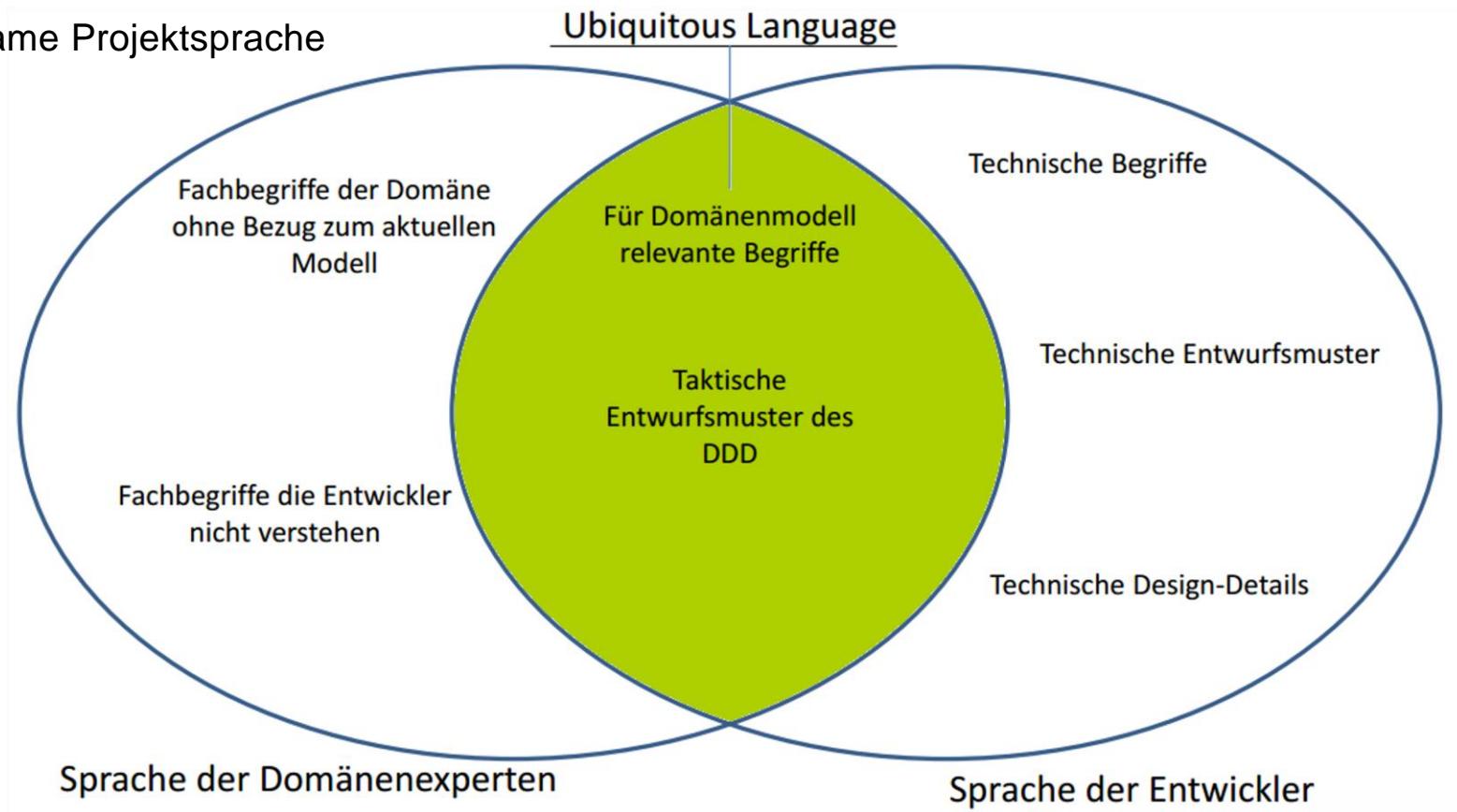
- Verständnisprobleme
 - Domänenexperten verstehen die Sprache der Entwickler meistens nur sehr begrenzt
 - Wenn ein Entwickler in seiner Sprache über das Domänenmodell spricht, hängt er den Experten ab
 - Entwickler verstehen die Sprache der Domänen-experten meistens nur sehr begrenzt
 - Wenn ein Domänenexperte in seiner Sprache über die Domäne spricht, verliert er den Entwickler schnell, weil dieser immer „übersetzen“ will/muss
 - Keine der beiden Sprachen eignet sich alleine für das Projekt

b) Domain Driven Design

- Verständnisprobleme im Code
 - Diese Kluft im gegenseitigen Verständnis setzt sich in die Software fort
 - Der Code entfernt sich von der Sprache der Domäne
 - Die Implementierung wird schwerer zu verstehen
 - Die Ubiquitous Language soll die Kluft reduzieren, indem Domänenexperten und Entwickler eine gemeinsame Sprache finden, die
 - Alle relevanten Konzepte, Prozesse und Regeln der Domäne beschreibt
 - Zusammenhänge verdeutlicht
 - Mehrdeutigkeiten und Unklarheiten beseitigt

b) Domain Driven Design

Die gemeinsame Projektsprache



b) Domain Driven Design

- Gemeinsame Projektsprache finden
 - Mit den Domänenexperten sprechen
 - Begriffe nicht „im stillen Kämmerchen“ festlegen
 - Genau zuhören und kritisch nachfragen
 - „Warum darf diese Schaltfläche nicht mehrfach betätigt werden?“
 - „Was bedeutet Mumie beim Buchzustand?“
 - Ein Glossar für die Begriffe kann helfen
 - Nie mit dem Zuhören und Nachfragen aufhören

b) Domain Driven Design

- Entitäten
 - Entities bilden identifizierbare und identitätstragende „Etwasse“ aus der Domäne ab
 - Jede Entity hat ihren eigenen Lebenszyklus
 - Die Eigenschaften der Entity können sich ändern
 - Auch Entities kümmern sich darum, die Regeln der Domäne zu forcieren
 - Der Konstruktor erzeugt nur gültige Instanzen
 - Spätere Änderungen der Eigenschaften dürfen die Entity nicht in einen ungültigen Zustand versetzen
 - „Gültiger Zustand“ ist durch die Domäne bestimmt

b) Domain Driven Design

- Grundlagen des Domänenmodells
 - Entities sind zusammen mit den Value Objects die Basis jeden Domänenmodells
 - Nicht alle Entities sind physikalisch manifestierte „Etwasse“
 - In einer Bankanwendung sind Kunden, Konten und Verträge Entities und real vorhanden
 - Kontenbewegungen sind ebenfalls Entities, aber vermutlich nicht mehr real vorhanden
 - Entities tauchen in der Sprache der Domänen-experten irgendwann auf (meist als Substantiv)

b) Domain Driven Design

- Identität von Entitäten
 - Die Identität einer Entity in der Domäne ist nicht die Objektidentität (Speicheradresse) in der Programmiersprache
 - Oft ist sogar die Verwendung von equals() und hashCode() für die Domänenidentität schwierig
 - Zwei Entities können sogar die gleiche Identität (in der Domäne) haben, obwohl sie Instanzen von unterschiedlichen Klassen/Typen in der Software sind
 - Die Identität einer Entity ist oft in der Domäne wichtig. Sie ist immer für die Entwicklung wichtig.

b) Domain Driven Design

- Beispiele für Identität von Entitäten
 - Logistik im zweiten Weltkrieg
 - Jede Kiste hat eine eindeutige Nummer – für den Heeresteil, dem sie gehört
 - Jedes Schiff hat bis zu drei Nummern: Eine von der Marine, eine für das Heer und eine für private Speditionen
 - Alle Nummern waren 6- oder 7-stellig
 - In einem Lagerhaus konnten drei Kisten „123456“ stehen (Marine, Heer, Luftwaffe)
 - Die Kiste 123456 des Heeres soll auf das Schiff 7654321 verladen werden ...

b) Domain Driven Design

- Eigenschaften von Identitäten
 - Mindestens drei Arten von Identität:
 - Kombination von Eigenschaften
 - Surrogatschlüssel (von der Anwendung selbst generierter Schlüssel : GUID oder UUID)
 - Natürlicher Schlüssel (ein bereits vorhandener Identifikator aus der Domäne : PassNr)
 - Jede Art und Vorgehensweise, Identitäten zu vergeben, hat Vorteile und Nachteile
 - Ohne klare Gründe erstmal die Vorgehensweise der Domäne verwenden
 - Es können durchaus verschiedene Vorgehens-Weisen in einer Anwendung verwendet werden

c) Requirements Engineering

- Notwendig als Grundlage zur Domänenmodellierung und Architektur
- Umfasst alle Aufgaben wie Aufnehmen, Analysieren, Dokumentieren, Validieren und Pflegen der Anforderungen (und Einschränkungen) der Stakeholder an ein Softwaresystem
- Als Software Architekt ist man in alle diese Aufgaben involviert, deshalb macht es Sinn sie zu kennen
- Typen von Software Anforderungen (Requirements)
 - Business requirements
 - Functional requirements
 - Non-functional requirements
 - Constraints (Einschränkungen)

c) Requirements Engineering

- Business requirements
 - Sind die Ziele des Business (hohes Level), die mit dem Software System erreicht werden sollen
 - Die Probleme, die gelöst werden sollen oder die Möglichkeiten, die das Software System für das Business erreichen soll
 - Business requirements können Anforderungen, die vom Markt kommen, beinhalten
 - Unterscheidungsmerkmale oder Ähnlichkeiten zu Lösungen von Wettbewerbern finden sich hier
 - Business requirements haben meistens den Fokus auf Qualitätsmerkmalen

c) Requirements Engineering

- Functional requirements
 - Beschreiben die Funktionalität des Software Systems
 - Sie beschreiben die Fähigkeiten in Bezug auf das Verhalten des Systems
 - Funktionalitäten und Fähigkeiten ermöglichen Stakeholdern ihre Aufgaben zu erfüllen
 - Beinhaltet die Interaktion des System mit der Umgebung (Input, Output, Services, Interfaces)
 - Können von unterschiedlichen Quellen kommen wie
 - Organisation (Regeln und Grundsätze des Unternehmens)
 - Gesetzgebung (Recht und Rechtsprechung)
 - Ethik (Sicherheit und Schutz)
 - Auslieferung und Verwendung
 - Standards (die befolgt werden müssen)
 - Extern (z.B. Integration mit einem externen System)

c) Requirements Engineering

- Non-functional requirements
 - Bedingungen, die zur Effektivität der Lösung beitragen
 - Einschränkungen, die bedacht werden müssen
 - Non-functional requirements werden oft nicht berücksichtigt, sind aber ein wichtiger Erfolgsfaktor
 - Können signifikante Auswirkungen auf das Design und die Architektur haben
 - Software Architekten müssen eine aktive Rolle bei der Erfassung spielen
 - Qualitätsattribute wie Wartbarkeit, Benutzbarkeit, Testbarkeit und Kompatibilität sind wichtige Themen

c) Requirements Engineering

- Constraints (Einschränkungen)
 - Technisch oder nicht-technisch
 - Funktional oder nicht-funktional
 - Entscheidungen, die schon gefallen sind und befolgt werden sollen
 - Ein Software Architekt muss sie beachten, er kann sie nicht ändern (jedenfalls nicht so einfach)
 - Beispiele
 - Abkommen mit Lieferanten
 - Bereits ausgewähltes Tool
 - Die Software muss sich im gesetzlichen Rahmen bewegen
 - Es gibt bereits festgelegte Termine oder Meilensteine
 - Ressourcen sind fest zugeordnet
 - Ein Entwicklerteam kennt hauptsächlich (nur) eine bestimmte Programmiersprache
 - Constraints sollten wie Anforderungen behandelt werden

c) Requirements Engineering

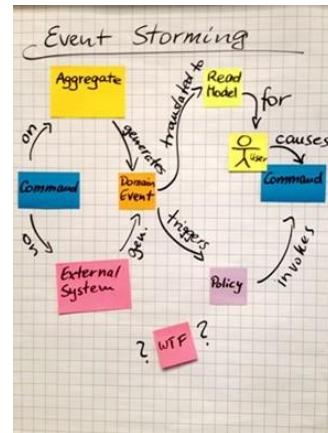
- Die Wichtigkeit kann nicht oft genug betont werden
- Wichtig für ein erfolgreiches Projekt, für eine gute Architektur, für zufriedene Kunden
- Vorteile
 - Reduzierte Nacharbeit
 - Weniger unnötige Eigenschaften
 - Geringere Erweiterungskosten
 - Schnellere Entwicklung
 - Geringere Entwicklungskosten
 - Bessere Kommunikation
 - Genauere Schätzungen beim Testen
 - Höhere Kundenzufriedenheit

c) Requirements Engineering

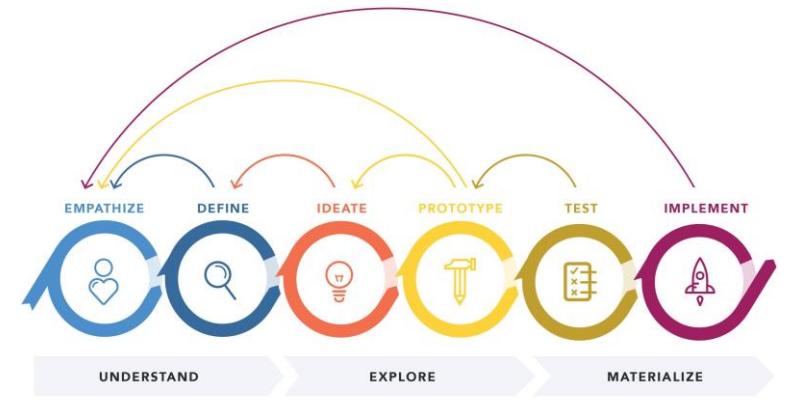
- Anforderungen müssen
 - meßbar und testfähig sein
 - Webpage ist in 2 sec bedienbar und nicht Webpage ist schnell bedienbar
 - Eindeutig und konsistent
 - Klar und nicht widersprechend
 - Verifizierbar (nachprüfbar)
- Software Architekten weisen auf Unstimmigkeiten und Unzulänglichkeiten hin
- Software Architekten sorgen für eine klare und stimmige Dokumentation ohne Mißverständnisse
- Anforderungen können (müssen nicht) die Architektur massiv beeinflussen
- Anforderungen an die Qualität beeinflussen meistens die Architektur

c) Anforderungs Erhebung

- Es gibt verschiedenste Erhebungsmethoden
 - Interviews
 - Workshops
 - Event storming
 - Design Thinking
 - Brainstorming
 - Beobachtungen
 - Umfragen
 - Analyse von vorhandener Dokumentation
 - Prototyping
 - Reverse Engineering
- Wichtig : Man muss die richtigen Stakeholder kennen !

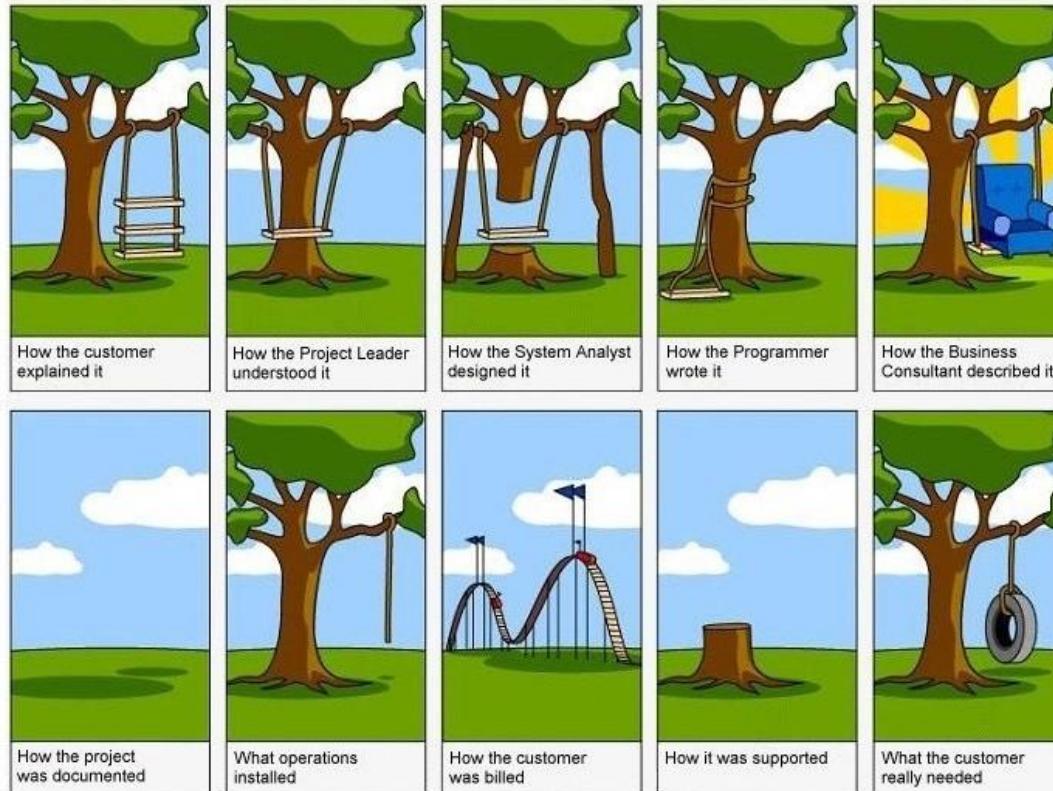


wissen was man weiß,
wissen was man nicht weiß,
nicht wissen was man weiß,
nicht wissen was man nicht weiß



DESIGN THINKING 101 NNGROUP.COM

c) Anforderungs Erhebung



**Wissen was man weiß,
wissen was man nicht weiß,
nicht wissen was man weiß,
nicht wissen was man nicht weiß**

4) Software Qualitätsattribute

- Sind für Architekten enorm wichtig, da sie Architekturentscheidungen beeinflussen
- In diesem Kapitel betrachten wir Softwareattribute und deren Bedeutung im Softwareentwicklungszyklus
- Manche Attribute lassen sich nicht einfach feststellen
- Wir werden folgendes betrachten
 - a) Was sind Qualitätsattribute
 - b) Maintainability
 - c) Usability
 - d) Availability
 - e) Portability
 - f) Interoperability
 - g) Testability

a) Qualitätsattribute

- Sind Eigenschaften eines Softwaresystems und eine Untermenge der non-functional requirements
- Wie alle Anforderungen sollten sie meßbar und testbar sein
- Sie sind ein Benchmark für die Qualität eines Software Systems und messen dessen Fitness
- Ein Software System besteht aus einer Kombination von Qualitätsattributen
- Je besser die Qualitätsattribute erfüllt werden, um so besser ist die Qualität der Software
 - Sie haben entscheidenden Einfluss auf
 - Das Design
 - Die Wartbarkeit
 - Das Laufzeitverhalten
 - Die User Experience
- Einige Attribute beeinflussen sich gegenseitig oder können sich (wenigstens teilweise) gegenseitig ausschließen (z.B. Performance mit Scalability oder Security und Usability)
- Es ist wichtig, sie zu kennen, zu verstehen und benutzen zu können
- Eine vernünftige Balance kann akzeptierbare Lösungen hervorbringen

a) Qualitätsattribute

- Können intern oder extern sein
- Interne Attribute gehören zum Softwaresystem, sind meßbar und für Entwickler sichtbar
- Beispiele
 - LOC (lines of Code)
 - Maß der Kohäsion
 - Lesbarkeit des Codes
 - Grad der Kopplung zwischen Modulen
- Sie zeigen die Komplexität eines Software Systems
- Sie beeinflussen externe Attribute, obwohl sie nicht direkt von Endanwendern gesehen werden
- Ein höherer Level von interner Qualität hat in den meisten Fällen eine höhere äußere Qualität zur Folge

a) Qualitätsattribute

- Externe Qualitätsattribute sind Eigenschaften die extern sichtbar sind
- Sie sind für Endbenutzer sichtbar/bemerkbar
- Beispiele
 - Performance
 - Reliability (Ausfallsicherheit)
 - Availability (Verfügbarkeit)
 - Usability (Benutzbarkeit)
- Qualitätsattribute spielen eine wichtige Rolle im gesamten SDLC (Software Development Life Cycle)
 - Vom Design bis zum Testen

a) Qualitätsattribute

- Müssen immer wieder überprüft werden
 - Manuell (z.B. Usability Test)
 - Durch Tools nach definierten Benchmarks
 - Durch Code Reviews
 - Durch automatisierte Unit Tests
- Jede Testmethode hat seine Stärken und Schwächen
- Oft reicht ein Test alleine nicht aus, um ein Attribut zu bestimmen
- Manchmal sind umfangreiche Tests notwendig
- Eine sinnvolle Balance zwischen Testaufwand und benötigter Zeit ist wichtig
- Automatisierung von Test, wo immer möglich, ist notwendig
- Automatisierte Test sind oft Teil eines Continuous Delivery Prozesses

a) Qualitätsattribute (Liste)

- Accessibility, accountability, accuracy, adaptability, administrability, affordability, agility, auditability, autonomy, availability
- Compatibility, composability, configurability, correctness, credibility, customizability
- Debuggability, degradability, determinability, demonstrability, dependability, deployability, discoverability, distributability, durability
- Effectiveness, efficiency, evolvability, extensibility, failure transparency, fault-tolerance, fidelity, flexibility, inspectability, installability, integrity, interchangeability, interoperability, learnability, localizability, maintainability, manageability, mobility, modifiability, modularity, observability, operability, orthogonality
- Portability, precision, predictability, process capabilities, producibility, provability, recoverability, relevance, reliability, repeatability, reproducibility
- Resilience, responsiveness, reusability, robustness, safety, scalability, seamlessness, self-sustainability, serviceability, securability, simplicity, stability, standards compliance, survivability, sustainability, tailorability, testability, timeliness, traceability, transparency, ubiquity, understandability, upgradability, usability, vulnerability

b) Maintainability

- Maß für die Wartbarkeit eines Software Systems
- Wartung muss auch nach der Inbetriebnahme einfach möglich sein
- Sichert den Wert des Software Systems über die Zeit
- Änderungen an Software Systemen sind vorprogrammiert – am besten schon vorgeplant
- Früher lag der Hauptkostenblock eines Systems in der Entwicklung
- Heute hat sich das in Richtung Wartung verschoben
 - Früher am Markt, dafür eingeschränkte Funktionalität
- Beeinflusst daher stark die Gesamtkosten
- Ein guter Entwickler entwickelt gut wartbaren Code (auch für andere Entwickler)
- Gute Wartbarkeit unterstützt die Migration von Systemen auf neue Technologien
- Gute Wartbarkeit unterstützt Reverse-Engineering

b) Maintainability

- Typen von Wartbarkeit
 - Corrective
 - Korrektur von Fehlern (Bugs), gefunden im Betrieb oder von Benutzern
 - Führt zu Incidents (Prio = Reihenfolge der Behebung, Severity = Wirkweite)
 - Wartbarkeit kann an der Zeit für Suche, Finden und Behebung gemessen werden
 - Perfective
 - Bei neuer oder erweiterter Funktionalität
 - Eine gute Architektur kann Perfective Maintainability unterstützen

b) Maintainability

- Typen von Wartbarkeit
 - Adaptive
 - Um Softwaresysteme an Änderungen der Umgebung anzupassen
 - Neues Betriebssystem oder neues Datenbanksystem
 - Preventive
 - Um Fehler, Ausfälle oder Probleme in der Zukunft zu vermeiden
 - Predictive
 - Der Versuch, mit Big Data Methoden Fehler, Probleme oder Ausfälle vorherzusagen

b) Maintainability

- Modifiability (Veränderbarkeit)
 - Die Möglichkeit Software Systeme einfach zu ändern ohne die Wartbarkeit zu verschlechtern
 - Wichtig für moderne Software in agilen Umgebungen
 - Iterationen in der Entwicklung ändern Systeme Schritt für Schritt
- Erweiterbarkeit und Flexibilität
 - Wird auch Adaptivität und Resilienz genannt
 - Werden wir in Teil 17 behandeln

b) Maintainability

- Spezielles Design zur Unterstützung von Wartbarkeit ist notwendig
 - Reduzierung der Komplexität
 - Klingt einfach – kann aber mehr Zeit und/oder Geld kosten
- Techniken zur Reduzierung der Komplexität
 - Reduzierung der Größe
 - Erhöhung der Kohesion (Zusammenhalt)
 - Reduzierung von Koppelung

Siehe Teil 6 Prinzipien und Praktiken

b) Maintainability

- Das Messen der Wartbarkeit
 - Lines of Code (LOC)
 - Vergleich zwischen 2 Systemen nur möglich bei unterschiedlichen Größenordnungen
 - Kann von Entwicklungsumgebungen gezählt werden
 - Cyclomatische Komplexität
 - Ist eine quantitative Metrik
 - Entwickelt von Thomas J. McCabe
 - Maßzahl von linear unabhängigen Pfaden durch ein Modul oder Element
 - Formel : $CC = E - N + 2P$
 - Darstellung in Graphen

E = Anzahl der Kanten im Graphen

N = Anzahl der Knoten im Graphen

P = Anzahl von Pfaden im Graphen

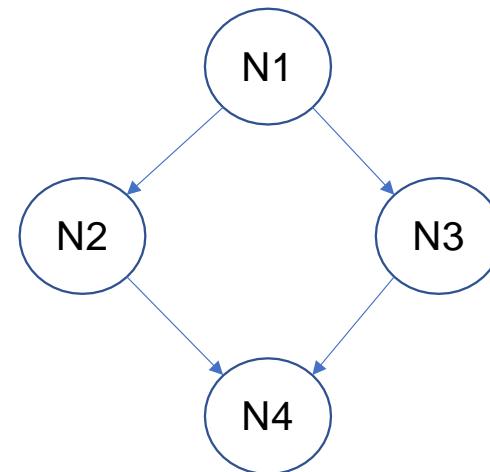
b) Maintainability

- Cyclomatische Komplexität
 - Beispiel

Pseudocode

```
If (N1)
  then N2
  else N3
End if
N4
```

Graph



$$\begin{aligned} E &= 4 \\ N &= 4 \\ P &= 1 \end{aligned}$$

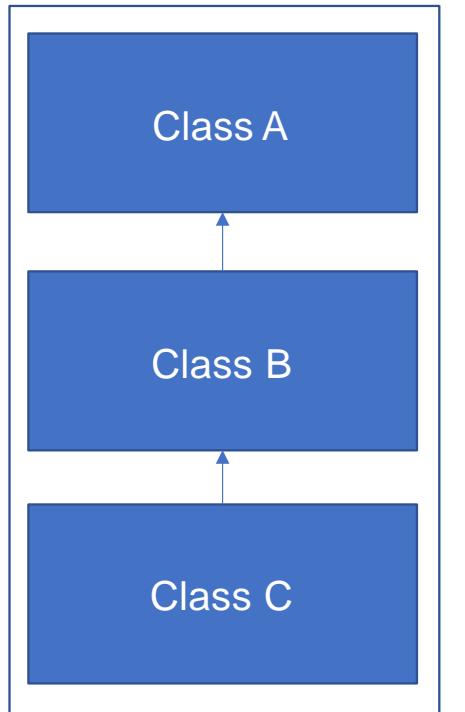
$$CC = 4 - 4 + 2 * 1 = 2$$

Geringe Komplexität

Ab $CC > 10$ spricht man von erhöhter Komplexität

b) Maintainability

- Das Messen der Wartbarkeit
 - Depth of Inheritance tree (DIT)



DIT = 0

DIT = 1

DIT = 2

Je größer DIT je höher die Komplexität

DIT > 5 deutet auf Optimierungspotential hin

c) Usability

- Wichtiger Punkt für gute Architektur
- Beschreibt wie einfach es für Enduser ist ihre Aufgaben mit dem System zu erledigen (useful)
- Ist für Enduser hauptsächlich im Fokus
- Hat Einfluss auf den Gesamteindruck des Systems
- Kann oft am einfachsten verbessert werden
- Ist enorm wichtig, da das die Produktivität der Enduser maßgeblich beeinflusst
 - Gute Usability erhöht die Produktivität
- Kann im schlechten Fall zur totalen Ablehnung oder zur Suche nach Alternativen führen
- Usability untergliedert sich in
 - Learnability
 - Die Geschwindigkeit mit der neue User das System zu bedienen lernen (ISO/IEC 25010)
 - Accessibility
 - Wie kann auf das System zugegriffen werden (Technologien, Navigation, Farben, ...)
 - Feedback
 - Nachrichten, Tooltips, lang laufende Tasks

d) Availability

- Beschreibt ein prozentuales Maß über eine definierte Zeit, an der ein System verfügbar ist
- Ist die Wahrscheinlichkeit der Verfügbarkeit eines Systems zu einem bestimmten Zeitpunkt
- Man sieht oft Maßzahlen wie 99,9%, 99,99% oder 99,999%
- Ist definiert als

$$\text{Verfügbarkeit} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

MTBF = meantime between failures (manchmal auch MTTF = meantime to failure)
 Die mittlere Zeit zwischen 2 Ausfällen

MTTR = meantime to recovery/repair
 Die mittlere Zeit die zum Wiederanlaufen/zur Reparatur benötigt wird

Wichtig : es muss ein Bezugszeitraum angegeben sein
 99,999% in Bezug auf 1 Jahr bedeutet 5 min und 15 sec

d) Availability

- Wichtig : es muss ein Bezugszeitraum angegeben sein
 - Typische Verfügbarkeiten

Verfügbarkeiten	Downtime / Jahr	Downtime / Monat	Downtime / Woche
99,0 %	3,65 Tage	7,2 Stunden	1,68 Stunden
99,9 %	8,76 Stunden	43,2 Minuten	10,1 Minuten
99,99 %	52,6 Minuten	4,32 Minuten	60,5 Sekunden
99,999 %	5,26 Minuten	25,9 Sekunden	6,05 Sekunden

- Verfügbarkeiten abhängiger Systeme werden multipliziert

d) Availability

- Fehlererkennung
 - Ping
 - Von außen anpingen (TCP/IP Ping)
 - Heartbeat
 - Periodisches Senden von Nachrichten nach außen
 - Timestamp
 - Zur Erkennung von Fehlern in Reihenfolgen
 - Voting
 - Triple modular redundancy (TMR)
 - Drei Module machen die gleiche Berechnung
 - Die Mehrheit gleicher Ergebnisse bestimmt das Gesamtergebnis
 - Sanity checking
 - Oft eine speziell implementierte Funktion (von außen aufrufbar)
 - Condition monitoring und Selbsttests

d) Availability

- Fehlerbehebung
 - Exception handling
 - geplant gesteuerte Ausnahmebehandlung
 - Retry strategien
 - Wiederholungen (in Intervallen, incrementell, sofort, zufällig)
 - Redundanz
 - Failover Mechanismen
 - Active/passive/cold Spare
 - Rollback
 - Zurückrollen zu einem definierten Punkt
 - Graceful degradation
 - Gezieltes Abschalten von Funktionalität
 - Ignorieren

d) Availability

- Fehlervermeidung
 - Removal from service (Stilllegen)
 - Transaktionen
 - Siehe Datenbanken
 - Competence Sets
 - Erhöhung der Fähigkeit eines Systems mit Fehlern umzugehen
 - Ausnahmen werden Bestandteil der Logik
 - Exception prevention
 - Prüfen von Grenzen (Arrays)
 - Überprüfen von Argumenten auf zulässige Werte

e) Portability

- Ist ein Maß der Effizienz und Effektivität ein Software System von einer Umgebung in eine andere zu migrieren. Die Faktoren, die Portability beeinflussen sind
 - Adaptability (Adaptivität, Anpassbarkeit)
 - Code, der anpassbar ist
 - SOLID Prinzipien unterstützen Adaptivität
 - Single responsibility principle
 - Open/closed principle
 - Liskov substitution principle
 - Interface segregation
 - Dependency inversion

e) Portability

- Ist ein Maß der Effizienz und Effektivität ein Software System von einer Umgebung in eine andere zu migrieren. Die Faktoren, die Portability beeinflussen sind
 - Installability
 - Grad der einfachen Installierbarkeit/Deinstallierbarkeit eines Systems
 - Der Installationsprozess sollte einfach bedienbar und verständlich sein
 - Der Installationsprozess sollte Updates verarbeiten können (Apple als Vorbild)
 - Replaceability
 - Maß der Fähigkeit, daß ein System ein anderes ersetzen kann
 - Internationalisierung und Lokalisierung
 - Maß der Anpassung an verschiedene Sprachen und kulturelle Unterschiede
- Wenn ein Software System auf Portability getrimmt ist, sollte dieser Zustand auch nach nachfolgenden Änderungen bewahrt werden

f) Interoperability

- Das Maß der Fähigkeit Informationen mit anderen Software Systemen austauschen zu können
- Um Informationen austauschen zu können, müssen Systeme kommunizieren können
 - Syntaktische Interoperabilität
- Um Informationen austauschen zu können, müssen Systeme die Informationen verstehen können
 - Semantische Interoperabilität
- Es gibt Interoperability Standards wie JSON oder HTTP(S)

g) Testability

- Das Maß der Testfähigkeit eines Systems
- Ein nicht unerheblicher Teil der Gesamtkosten eines Software Systems werden durch das Testen verursacht
- Gute Software Architektur sorgt für gute Testfähigkeit
- Ein hohes Maß an Testfähigkeit hilft auch bei der Fehlersuche
- Schnelles Auffinden von Fehlern erhöht die Gesamtqualität eines Systems

5) Resilienz

- In heutigen komplexen, verteilten und hochgradig vernetzten Systemlandschaften ist es unmöglich, die vielfältigen möglichen Fehlerquellen zu antizipieren und durch vorbeugende Maßnahmen auszuschließen
- Der immer höher werdende Grad an Verteilung, sei es wegen Cloud, Mobile oder Internet of Things sowie die immer schwerer vorhersagbaren Lastmuster werden dafür sorgen, dass Resilient Software Design in Zukunft noch viel wichtiger wird als es jetzt bereits ist
- Wir betrachten :
 - Definition
 - Herausforderungen
 - Pattern

Resilienz

- Definition

re•sil•ience (rɪ'zɪl yəns) also re•sil'ienc•y,

1. the power or ability to return to the original form, position, etc., after being bent, compressed, or stretched; elasticity.
2. ability to recover readily from illness, depression, adversity, or the like; buoyancy (Auftrieb)

<https://www.thefreedictionary.com/resilience>



All Content on TheFreeDictionary.com is for informational purposes only. Any inquiries concerning these Terms and Conditions of Use should be directed to:

Farlex, Inc.
1051 County Line Road Suite 100
Huntingdon Valley, PA 19006
USA
info@tfd.com

Resilienz

- In letzter Zeit hört man immer häufig den Begriff „Resilience“ und manchmal auch etwas vollständiger „Resilient Software Design“
- Irgendwie hat das etwas mit dem Umgang mit Fehlern zur Laufzeit in komplexen Systemlandschaften zu tun, von denen der Anwender nichts merken soll
- Aber was daran ist so neu und anders, dass man dafür einen neuen Begriff prägen muss ?
- Worum geht es ? Was ist anders ?

Resilienz

- Der primäre Zweck aller Geschäftsprozesse und der sie implementierenden und unterstützenden IT-Systeme ist, Geld zu erwirtschaften bzw. Kundenbedürfnisse zu befriedigen – am besten beides
- Das funktioniert aber nur, solange die IT-Systeme zuverlässig und – von außen betrachtet – fehlerfrei laufen
- Sind die Systeme nicht verfügbar oder fehlerhaft, sind die Kunden unzufrieden und man verdient kein Geld mit ihnen – kurzum: Sie sind wertlos
- Nur zuverlässig laufende Systeme haben Wert
- Die Verfügbarkeit von Systemen in Produktion ist also essenziell für den Wert der Software

Resilienz

- Wir kennen jetzt (hoffe ich jedenfalls), die Definition von Resilienz
- Wie kann ich Resilienz auf Softwaresysteme anwenden ?
- Was muss ich beachten ?
- **Vorsicht !!! Sollte etwas nicht verstanden werden, bitte nachfragen !**
- Die folgenden Folien sind nicht immer einfach – aber wichtig !

Resilienz

- **Die Strategien bei der Umsetzung**

- Konzentration auf den Business Case
- Fundiertes Wissen über verteilte Systeme und deren Gesetze
- Umgang mit der geforderten (100%) Hochverfügbarkeit
- DevOps richtig einführen
- Gutes (funktionales) Softwaredesign und Kenntnis der entsprechenden Prinzipien
 - Übersicht und Klassifikation von Prinzipien
- Erfahrungen aufheben/bewahren

Resilienz

- **Die Strategien bei der Umsetzung**

- Konzentration auf den Business Case

- Die erste Frage aus dem Management ist mit Sicherheit
 - Wieviel bringt uns das und was kostet es ?

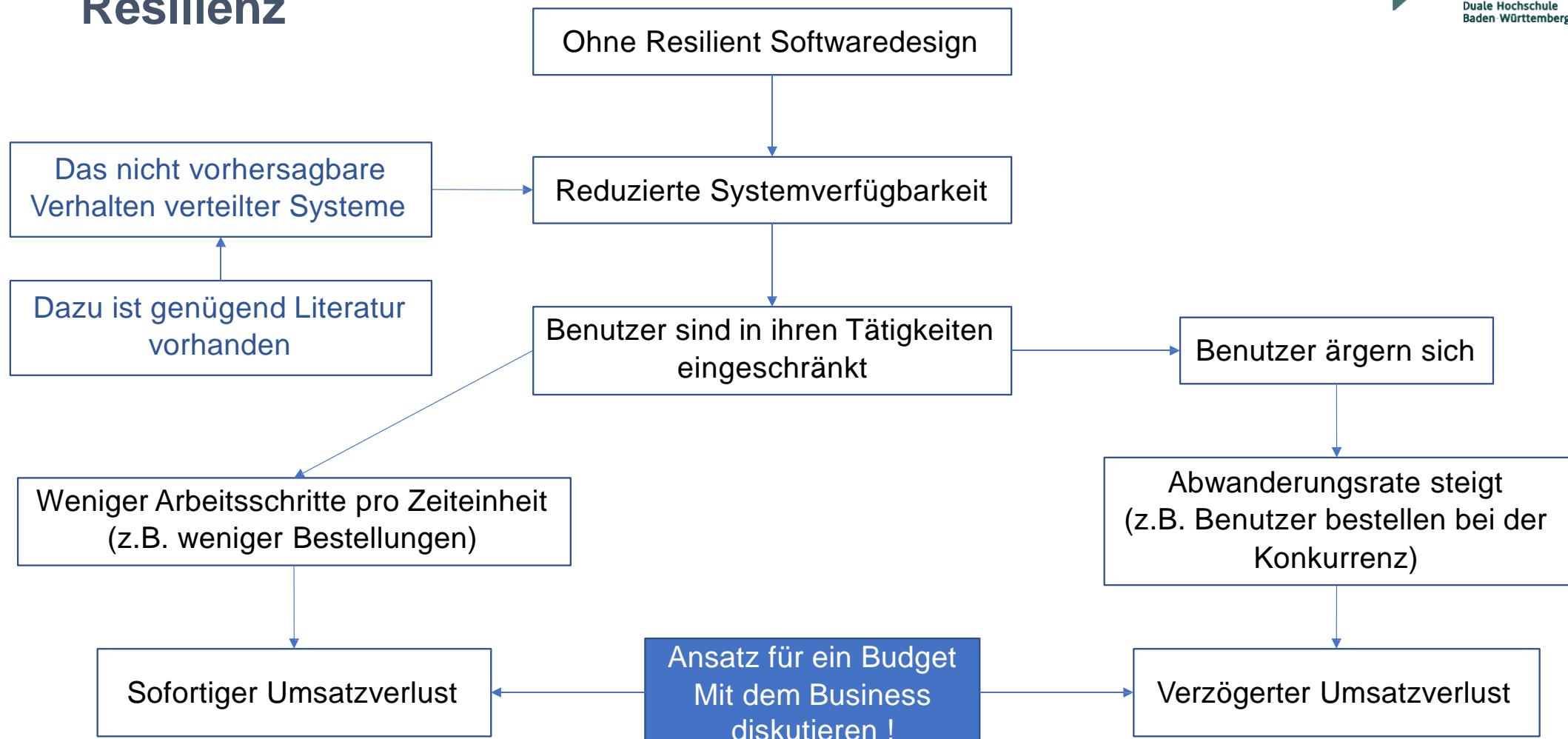
- Die erste Frage von den Product Ownern ist mit Sicherheit
 - Werden wir dadurch schneller ? (Velocity)

- Resilienz hilft nicht beim Geld verdienen, sondern verhindert Geld zu verlieren !

- Aber : Wie erkläre ich diesen wichtigen Sachverhalt den Stakeholdern ?

- Kleiner Ansatz auf der nächsten Folie

Resilienz



Resilienz

- **Die Strategien bei der Umsetzung**

- Fundiertes Wissen über verteilte Systeme und deren Gesetze

- Everything fails – all the time (Werner Vogels, VP und CTO Amazon)
 - Das muss man einfach akzeptieren
 - Ausfälle in verteilten Systemen versuchen zu verhindern – bleibt immer erfolglos !
 - Diese (erfolglosen) Versuche oder Ansätze sind immer auch extrem teuer !
 - Die IT Ausbildung lehrt aber i.A. diesen deterministischen Ansatz
 - Der deterministische Ansatz ist auch im Management verwurzelt
 - Andere Ansätze rufen die Gegner auf den Plan und haben sofort massiven Widerstand
 - Durchhaltevermögen und gute Argumente sind deshalb notwendig
 - Deshalb ein kleiner Ausflug auf den nächsten Folien



Resilienz

Was wird in der IT Ausbildung gelehrt

If X then Y

- Wenn ich dies tue, passiert das
- Vorhersagbar (deterministisch)

Gefangen im Prozessdenken

„Ursache – Wirkung“ Argumentation

Fähigkeit komplexe Systeme zu designen

Darin sind wir gut, weil unser Gehirn so funktioniert

Ausflug

Was wird für verteilte Systeme benötigt

If X then *maybe* Y

- Wenn ich dies tue, passiert vielleicht das
- Nicht vorhersagbar

Außerhalb von Prozessdenken

Wahrscheinlichkeiten werden wichtig

Fähigkeit komplexe Systeme zu designen

Können wir nicht gut,
weil unser Gehirn so nicht funktioniert

Resilienz

Klassische Physik

Gesetze

Newton
Ohm
Joule
Fourier
Maxwell
Holtzmann
Einstein

Garantien

Klassisches Weltbild

begreifbar, beschränkt, nachvollziehbar,
deterministisch

Ausflug

Quantenphysik

Wahrscheinlichkeiten

Schrödinger
Planck
Heisenberg
Hawking

Versprechungen, Annahmen

neues Weltbild

nicht begreifbar,
das klassische Weltbild ist Spezialfall,
nicht nachvollziehbar

Resilienz

- Das Ende des Determinismus ?
 - Heute ist so gut wie jedes System (in einem Unternehmen) ein verteiltes System
 - Selbst eine einfache Webanwendung besteht in der Regel aus einem Webserver, einem Application Server und einer Datenbank (Firewalls, Reverse Proxies, Load Balancer, eventuelle Cacheserver, Router und Switches nicht mitgezählt)
 - Ein CRM-System bei einem größeren Unternehmen ist in der Regel mit mehreren Dutzend anderen Systemen verbunden – vielfach online
 - Wir haben es heute also mit hochgradig komplexen, verteilten Systemlandschaften zu tun
 - Zu verteilten Systemen hat Leslie Lamport, einer der führenden Köpfe auf diesem Gebiet, einmal etwas salopp, aber treffend gesagt: *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable*
 - Was er damit ausdrücken will, ist, dass es so viele potenzielle Fehlerquellen in verteilten Systemen gibt, die das korrekte Funktionieren eines Systems compromittieren können und dass es unmöglich ist, diese alle zu antizipieren und zu vermeiden

- Das Ende des Determinismus ?

- Fehler sind in verteilten Systemen der Normalfall, nicht die Ausnahme, und es ist nicht möglich, sie vorherzusagen
- Entwicklungen wie *Cloud-Computing*, immer höhere Verfügbarkeitsanforderungen, *Mobile* und *Internet of Things* sowie Social Media sorgen für immer komplexere, immer höher vernetzte Systemlandschaften mit immer mehr beteiligten Systemen und immer weniger vorhersehbaren Zugriffs- und Lastmustern
- Der angenommene Determinismus der Stabilitätsansätze lässt sich also nicht mehr aufrechterhalten
- Es gibt keine deterministisch definierbaren Rahmenbedingungen in heutigen Systemlandschaften mehr, wie sie in den Softwarequalitätsstandards gefordert werden
- Wir haben es mit – teilweise unbekannten – Wahrscheinlichkeiten zu tun
- Der Determinismus ist einem Probabilismus gewichen

- Typische Ausfälle in verteilten Systemen
 - Die Literatur z. B. [Distributed Systems](#) von Andrew Tannenbaum unterscheidet zwischen fünf Fehlertypen
 - *Crash Failure* (Absturzfehler) – ein System antwortet permanent nicht mehr, hat bis zum Zeitpunkt des Ausfalls aber korrekt gearbeitet
 - *Omission Failure* (Auslassungsfehler) – ein System reagiert auf (einzelne) Anfragen nicht, sei es, dass es die Anfragen nicht erhält oder keine Antwort sendet
 - *Timing Failure* (Antwortzeitfehler) – die Antwortzeit eines Systems liegt außerhalb eines festgelegten Zeitintervalls
 - *Response Failure* (Antwortfehler) – die Antwort, die ein System gibt, ist falsch
 - *Byzantine Failure* (Byzantinischer/zufälliger Fehler) – ein System gibt zu zufälligen Zeiten zufällige Antworten („es läuft Amok“)
- Sehen oft einfach aus und sind in lokalen Systemen einfach zu lösen
- Führen in verteilten Systemen aber meistens zu extremen Problemen
- Deshalb gibt es dazu viel Literatur und Vorträge
(Dr. Eric Brewer [CAP Theorem] : „Towards robust distributed systems“)
Nancy Lynch : <https://www.youtube.com/watch?v=TAxnJptSWXc> (lohnt sich anzusehen ! Einigkeit = Consensus über Werte)

- Empfehlung
 - Damit umgehen lernen !
- Verteilte Systeme sind nicht vorhersagbar bei
 - Vollständigkeit der Ausführung
 - Reihenfolge von Nachrichten
 - Zeitlicher Koordination von Kommunikation
- Das wird sich auf die Anwendung auswirken
 - Die Infrastruktur kann das nicht abfangen
 - Man tut gut daran einen Plan zu haben, um sich darauf vorzubereiten zu können
 - Ansonsten wird es zu Inkonsistenzen führen



- **Die Strategien bei der Einführung**

- Umgang mit der geforderten (100%) Hochverfügbarkeit
 - Die Anforderungen nach 100% Verfügbarkeit kommen sowohl vom Business als auch von der IT
 - Das Business will seine Arbeit nicht unterbrochen sehen
 - Die IT denkt, sie kann das leisten
 - » man findet aber oft kaskadierende Fehlerketten
 - » Verantwortung wird anderen übertragen, die dann in Abhängigkeit geraten
 - ich nutze Service A und ich muss hochverfügbar sein, also muss das auch Service A sein
 - Service A war aber ganz anders geplant und ist nicht hochverfügbar umgesetzt worden
 - Fehler werden garantiert passieren, die Frage ist nur : Wann ?
 - Das muss akzeptiert werden, sonst sitzt man in der 100%-Falle oder im War-Room
 - Wenn man das akzeptiert, kann man sich darauf vorbereiten

- Kleine Anmerkung zur Verfügbarkeit
 - Annahme Verfügbarkeit eines Service : 99,5 % (inkl. geplante Downtimes)
 - Bei 10 Services, die in eine Anfrage/Aufgabe involviert sind, verringert sich die Gesamtverfügbarkeit auf 95,1%
 - Bei 50 Services entsprechend auf 77,8 %

Availability Formulas

Serial availability



$$\begin{aligned}\text{Availability} &= \text{Avail}_A \times \text{Avail}_B \times \text{Avail}_C \\ &= 99.999\% \times 99.999\% \times 99.999\% \\ &= 99.997\%\end{aligned}$$

- Verfügbarkeit und Fehlertypen

- Was ist *Verfügbarkeit*

- Die Verfügbarkeit A (für *Availability*, den englischen Begriff für Verfügbarkeit) ist definiert als $A := \text{MTTF} / (\text{MTTF} + \text{MTTR})$

Darin bedeuten:

- MTTF (*Mean Time To Failure*): die durchschnittliche Zeit vom Beginn des ordnungsgemäßen Betriebs eines Systems bis zum Auftreten eines Fehlers
 - MTTR (*Mean Time To Recovery*): die durchschnittliche Zeit vom Auftreten eines Fehlers bis zur Wiederherstellung des ordnungsgemäßen Betriebs des Systems

- Während der Nenner also die gesamte Zeit beschreibt, beschreibt der Zähler den Teil der Zeit, in dem das System ordnungsgemäß funktioniert
 - Damit kann die Verfügbarkeit Werte zwischen 0 für „gar nicht verfügbar“ und 1 für „immer verfügbar“ annehmen
 - Wenn man den Wert mit 100 multipliziert, erhält man die bekannte Darstellung als Prozentwert

Resilienz

Traditioneller Ansatz

Der traditionelle Ansatz besteht darin, das Auftreten eines Fehlers so lange wie möglich zu verschieben, d.h. die MTTF extrem groß zu machen, damit die MTTR für die Verfügbarkeit nicht signifikant wird.

Um dieses Ziel zu erreichen, sind hohe Investitionen in die Infrastruktur erforderlich:

- redundante Hardware (Cluster) und Spiegelung
- redundante Netzwerke
- Schattendatenbanken
- Zweites (entferntes) Rechenzentrum
- Businesscopies in Speichersubsystemen
- Und so weiter

$$\text{Availability} := \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Maximize MTTF through active actions to minimize the error probability

MTTF 

MTTR small compared to MTTF

Ausflug

Ziel : Availability = 1 (= 100 %)

Resilient Ansatz

Resilient Software Design versucht, Fehler zu akzeptieren, da sie weder vermieden noch vorausgesehen werden können, sondern nur auftreten.

Ausfallsicherheit ist die Fähigkeit eines Systems, unerwartete Situationen zu bewältigen, ohne den Endbenutzer zu beeinträchtigen (Best Case) oder (Worst Case) durch eine definierte und geplante „ordnungsgemäße Verschlechterung des Dienstes“

MTTF seen as given and not controllable

$$\text{Availability} := \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Minimize MTTR through active, planned and automated error corrections

MTTR 

Resilienz

- Traditionelle Stabilitätsansätze
 - Aber wie maximiert man die Verfügbarkeit?
 - Schaut man auf die Formel für Verfügbarkeit, gibt es dafür zwei Möglichkeiten
 - Entweder man versucht, den Wert für MTTF zu maximieren
 - oder man versucht, den Wert für MTTR zu minimieren
 - In beiden Fällen entwickelt sich der Wert von A, d. h. die Verfügbarkeit gegen 1, was der gewünschte Effekt ist
 - Der traditionelle Ansatz ist, das Eintreten eines Fehlers möglichst lange hinauszuzögern, d.h. den Wert für MTTF so groß zu machen, dass der Wert für MTTR unbedeutend wird
 - Bei diesen traditionellen Stabilitätsansätzen treibt man dafür meist einen großen Aufwand auf Infrastrukturebene: Es wird redundante Hardware eingesetzt, man betreibt HA-Cluster (High Availability), es werden mehrfache Netzwerkverbindungen (über verschiedene Switches) verwendet usw.

Resilienz

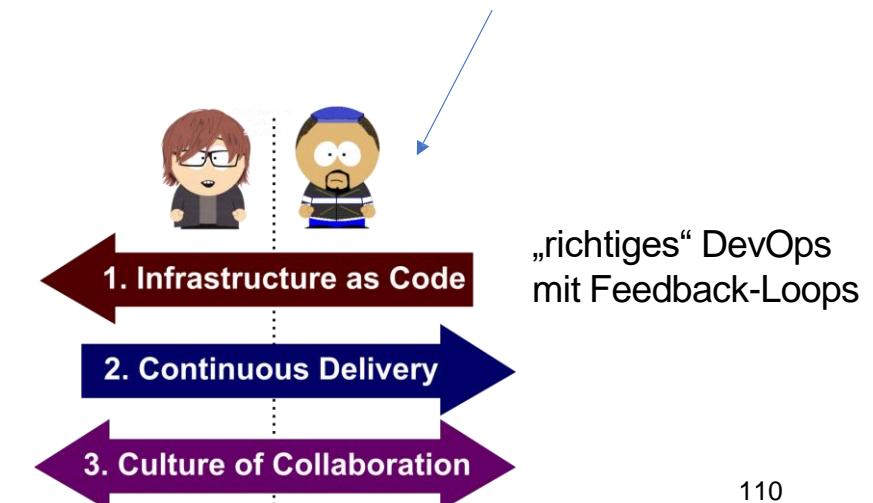
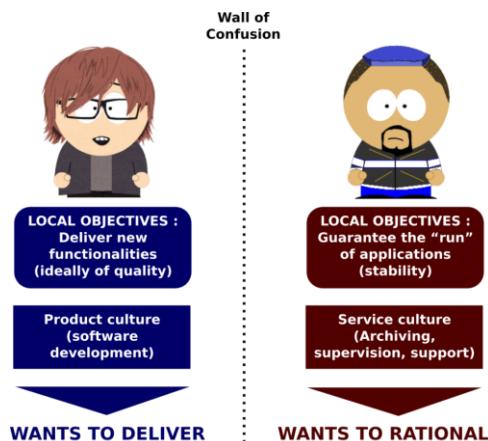
- Traditionelle Stabilitätsansätze
 - Dazu kommen aufwändige Verfahren zur Vermeidung von Fehlern auf der Softwareebene
 - Man versucht, die möglichen Fehlerquellen zu antizipieren und ergreift im Vorfeld Maßnahmen, um die Wahrscheinlichkeit des Eintretens solcher Fehler zu minimieren
 - Das ist ein durchaus valider Ansatz, solange das betrachtete System relativ isoliert läuft und es nur wenige Faktoren gibt, die die Verfügbarkeit des Systems beeinflussen
 - Dieser Ansatz spiegelt sich auch in den bekannten Softwarequalitätsstandards wieder. So findet sich z. B. im Softwarequalitätsstandard [ISO/IEC 25010:2011\(en\)](#) unter *Reliability* (Zuverlässigkeit), der Elterncharakteristik zu *Availability* (Verfügbarkeit) die folgende Definition:
Reliability: degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.
 - Wie man in der zweiten Hälfte der Definition lesen kann, ist die zugrunde liegende Annahme, dass es möglich ist, die Rahmenbedingungen für den Betrieb eines Systems deterministisch festzulegen

Resilienz

- **Die Strategien bei der Einführung**

- DevOps richtig einführen
 - Organisatorische Hürden überwinden
 - Entwickler (Dev) denken, dass Verfügbarkeit die Aufgabe von der Infrastruktur (Ops) ist
 - » In verteilten Systemlandschaften scheitert dieses Denken/Verhalten
 - DevOps Feedback wird überlebensnotwendig
 - Den „Wall of confusion“ einreisen

Nur so wird Resilienz zur Wirklichkeit



Resilienz

- **Die Strategien bei der Einführung**
 - Gutes (funktionales) Softwaredesign und Kenntnis der entsprechenden Prinzipien (Pattern)
 - Ohne ein angemessenes funktionales Design ist nichts anderes wichtig
 - Leider sind wir da nicht besonders gut unterwegs
 - Sehen Sie mal in die IT Landschaft Ihres Unternehmens

Uwe Friedrichsen



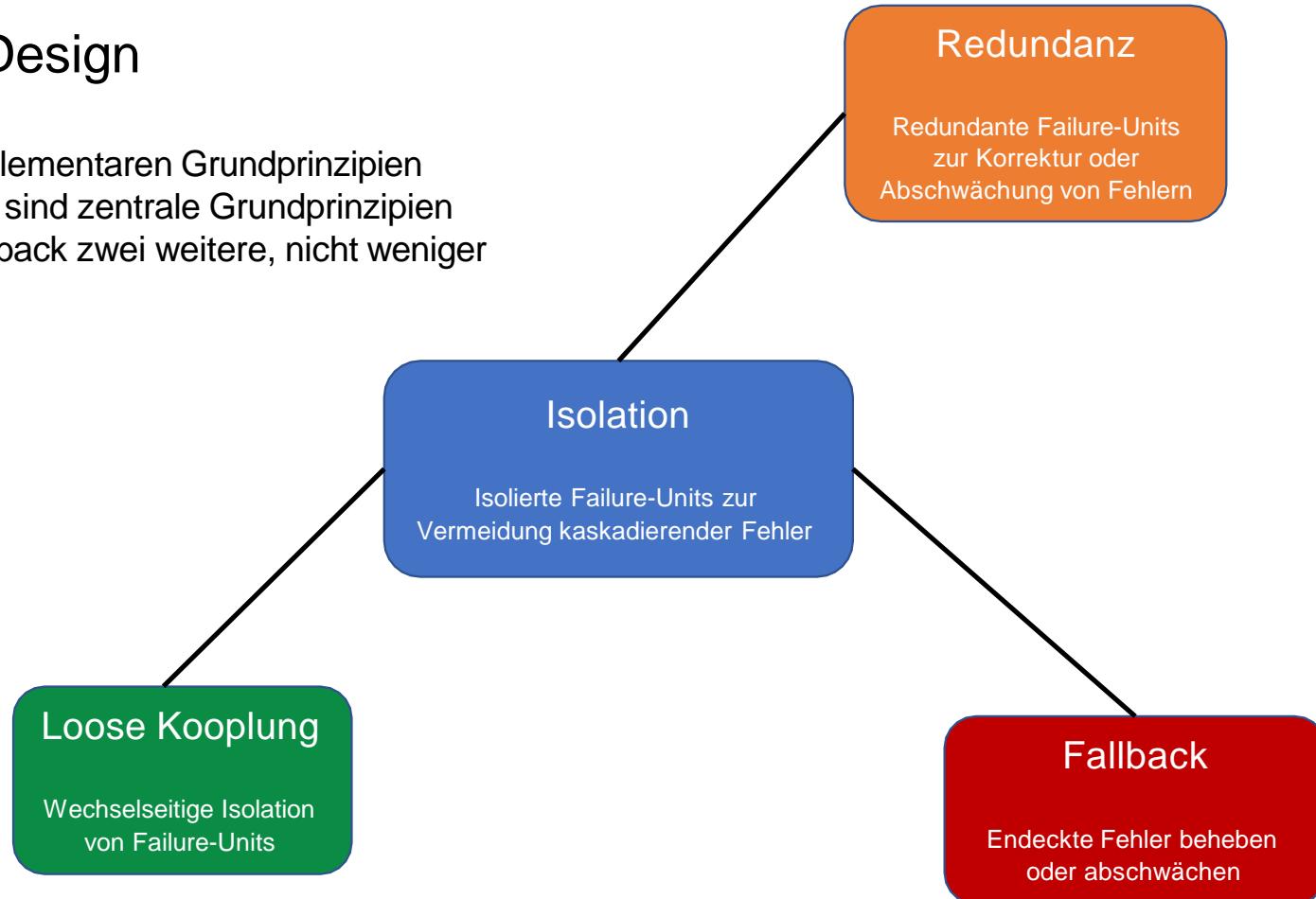
Resilienz

- Funktionales Design
 - Wie sieht das Design einer Anwendung aus, damit sie „resilient“ wird ?
 - Das ist ein weites Gebiet und es gibt unendlich viel Literatur oder Videos
 - Welchen Fragen muss sich ein Architekt stellen
 - Welche Aspekte müssen berücksichtigt werden
 - Im folgenden sehen wir uns die Grundprinzipien zum „Resilient Software Design“ etwas genauer an

Resilienz

- Funktionales Design

Resilienz besteht aus 4 elementaren Grundprinzipien
 Isolation und Redundanz sind zentrale Grundprinzipien
 Loose Kopplung und Fallback zwei weitere, nicht weniger wichtige Prinzipien



Resilienz

- Funktionales Design

Isolation

Das erste Grundprinzip von Resilienz ist Isolation.

Auf diesem Prinzip bauen fast alle anderen Resilienz-Muster auf.

Isolation bedeutet, dass ein System niemals als Ganzes kaputtgehen darf.

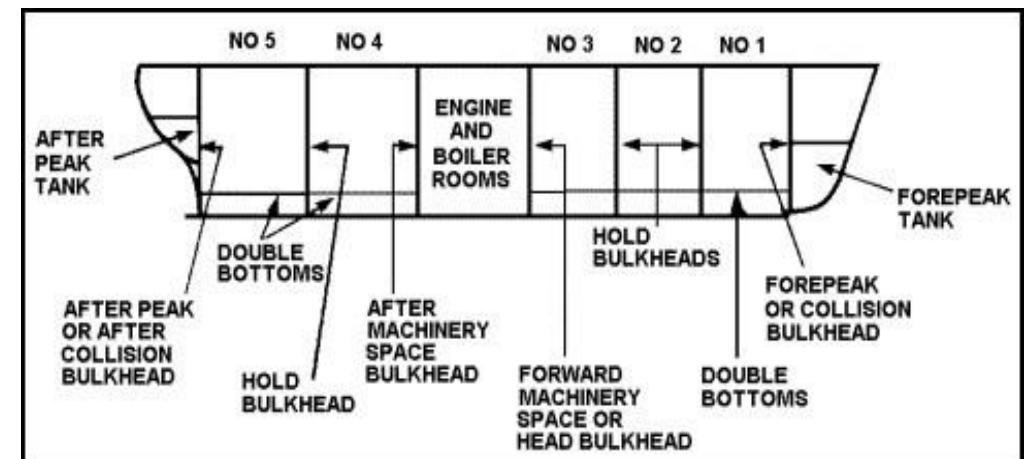
Um das zu vermeiden, teilt man das System in möglichst unabhängige Einheiten auf und isoliert diese gegeneinander (wie im Schiffsbau).

Diese unabhängigen Einheiten werden Bulkheads, Failure Units oder Units of Mitigation genannt (bulkhead = Schott).

Die Einheiten isolieren sich gegen Fehler anderer Einheiten, um kaskadierende, d. h. sich über mehrere Einheiten fortpflanzende Fehler zu vermeiden.

Isolation

Isolierte Failure-Units zur Vermeidung kaskadierender Fehler



Resilienz

- Funktionales Design
- Kleiner Test

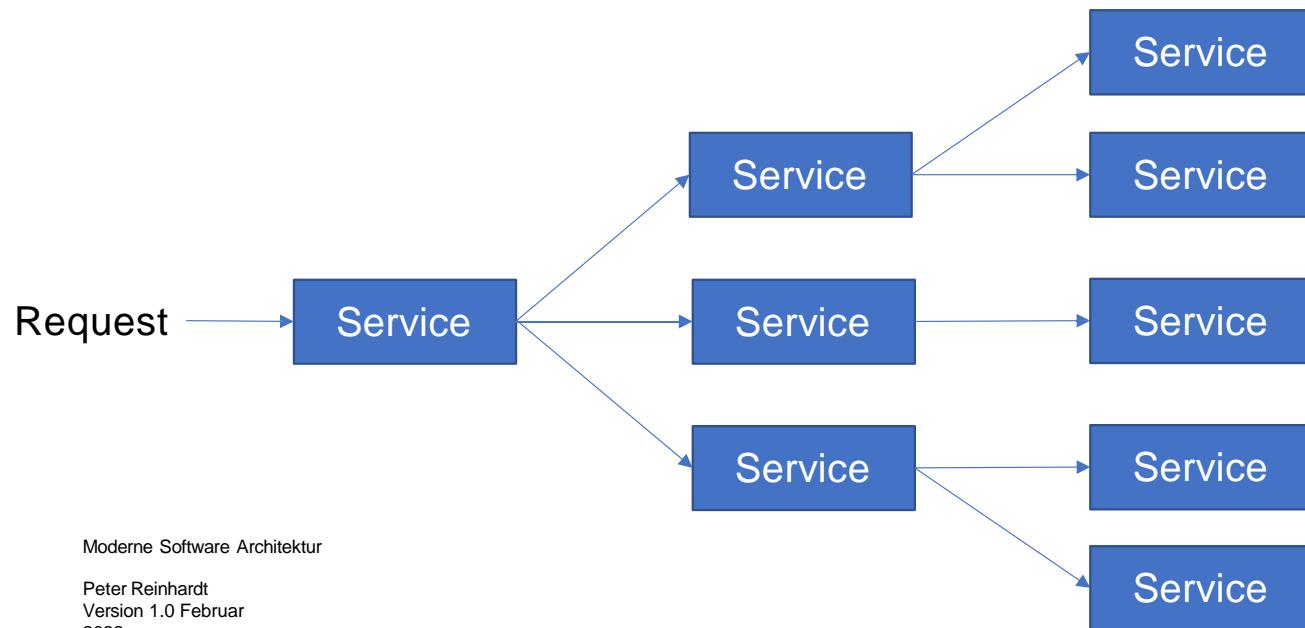
Service A benötigt Service B,
um den Request bedienen zu können

Ist das Isolation ?



Isolation

Isolierte Failure-Units zur
Vermeidung kaskadierender Fehler



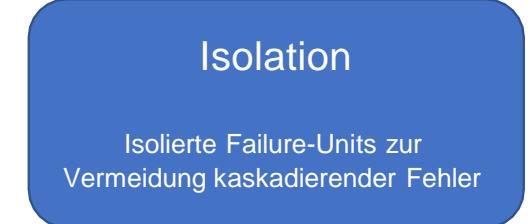
Broken Isolation by design

Und Was macht die Verfügbarkeit ?

Resilienz

- Funktionales Design
- Kleiner Test

Ist das Isolation ?



Resilienz

- Funktionales Design

- **Isolation**

- Validierung der Aufrufparameter
 - Korrekte Datentypen
 - Eingehaltene Wertebereiche
 - Erfüllte (Vor)Bedingungen
 - Ungleich NULL
 - Geprüfte Nummern (z.B. durch LUHN Algorithmus)
 - Was fällt uns noch ein ?
 - Saubere Rückgabewerte
 - Auf keinen Fall „NULL“ zurückgeben
 - Datenmengen einschränken
 - Und, und, und ...



Isolation

Isolierte Failure-Units zur
Vermeidung kaskadierender Fehler

Resilienz

- Funktionales Design

Isolation

Bulkheads in der Programmierung :

Beispiel Methodenaufrufe :

```
public int fakultaet(int n)
{
    return (n == 0) ? 1 : n * fakultaet(n-1);
}
```

Besser :

```
public int fakultaet(int n)
{
    if (n<0)
    {
        throw new IllegalArgumentException("n ist zu klein");
    }
    return (n == 0) ? 1 : n * fakultaet(n-1);
}
```

Isolation

Isolierte Failure-Units zur
Vermeidung kaskadierender Fehler

Noch besser :

```
public int fakultaet(int n)
{
    if (n<0)
    {
        throw new IllegalArgumentException("n ist zu klein");
    }
    if (n>12)
    {
        throw new IllegalArgumentException("n ist zu groß ");
    }
    return (n == 0) ? 1 : n * fakultaet(n-1);
}
```

Resilienz

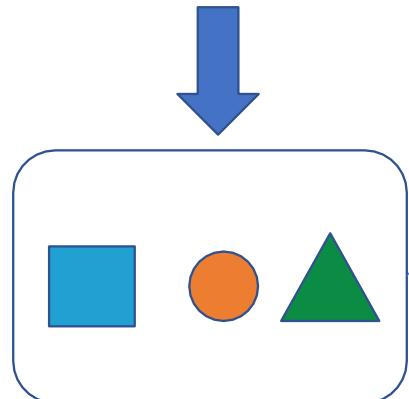
- Funktionales Design

Isolation

Bulkheads in der Architektur :



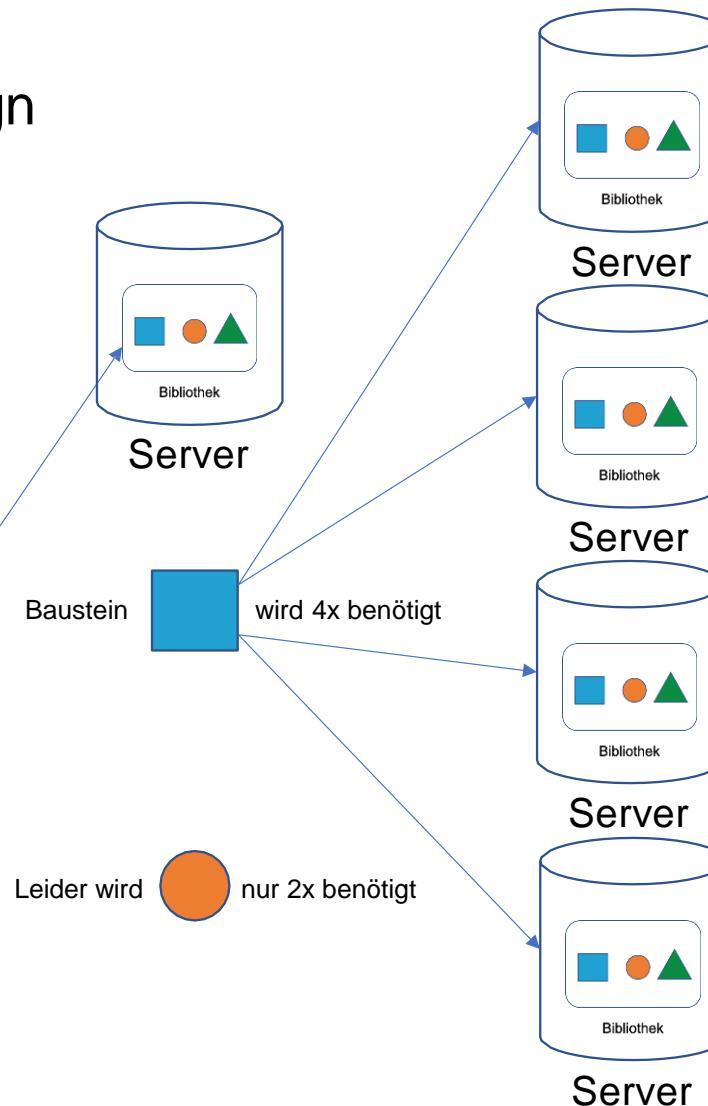
Funktionsbausteine



Bibliothek

Moderne Software Architektur

Dirk Leemans
1.0 Februar 2022



Isolation

Isolierte Failure-Units zur Vermeidung kaskadierender Fehler

Eine Änderung von  führt zu einem Redeploy von  und erhöht damit mögliche Fehlerquellen

Resilienz

- Funktionales Design

Isolation

Bulkheads in der Architektur:



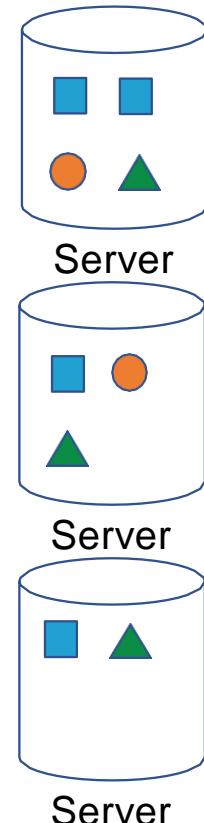
Funktionsbausteine

Microservices ?

Moderne Software Architektur

Dirk Leemans
1.0 Februar 2022

Lösung



Unabhängige Artefakte

=

Flexible Skalierung

Isolation

Isolierte Failure-Units zur
Vermeidung kaskadierender Fehler

Vorteile

Isolierte Entwicklung
Isolierte Fehler
Isoliertes Deployment
...

Resilienz

• Funktionales Design

Redundanz

Redundante Failure Units werden zur Korrektur oder Abschwächung von Fehlern genutzt.
Redundanz ist geeignet mit allen Arten von Fehlern umzugehen (nicht nur mit Absturzfehlern)

Was möchte man mit Redundanz adressieren ?

- Failover (komplett transparentes Umschalten auf eine andere Einheit)
- Geringe Latenz (Reduktion der Wahrscheinlichkeit zu langsamer Antworten)
- Erkennen von Antwortfehlern (durch Auswertung von Antworten mehrerer redundanter und unabhängiger Einheiten)
- Lastverteilung (zu zahlreiche Anfragen auf mehrere Einheiten verteilen)
- ... usw.

Abhängig vom gewählten Szenario sind weitere Aspekte zu berücksichtigen :

- Routingstrategien (Load Balancer, Round Robin)
- Master-Slave-Ansätze für Failoverszenarien
- Fan out & quickest one wins-Ansatz für geringe Latenz
- Automatisierung (Menschen unter Stress machen Fehler) mit Eingreifmöglichkeiten für Administratoren
- Verfolgung von Systemzuständen. Welche Einheit ist Master, welche ist Slave? Wie viele Einheiten laufen? Wo laufen sie? usw.

Redundanz ist ein mächtiges Prinzip, das eine Menge Architekturarbeit benötigt.

Redundanz

Redundante Failure-Units
zur Korrektur oder
Abschwächung von Fehlern

Resilienz

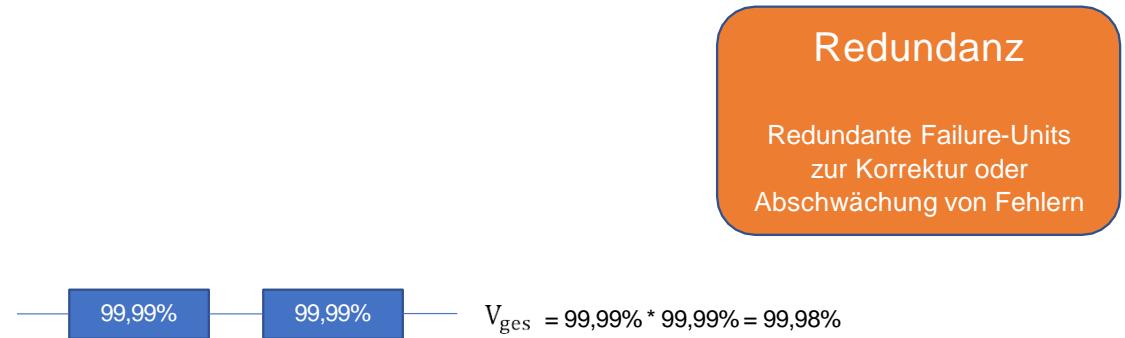
- Funktionales Design

Redundanz

$$\text{Verfügbarkeit: } V = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Für in Reihe geschaltete Systemkomponenten gilt

$$V_{\text{ges}} = \prod_{i=1}^n V_i$$



Für parallele nicht redundante Systemkomponenten gilt

$$V_{\text{ges}} = \frac{\sum_{i=1}^m V_i}{m}$$



Für parallele redundante Systemkomponenten gilt

$$V_{\text{ges}} = 1 - \prod_{i=1}^m (1 - V_i)$$



Resilienz

- Funktionales Design

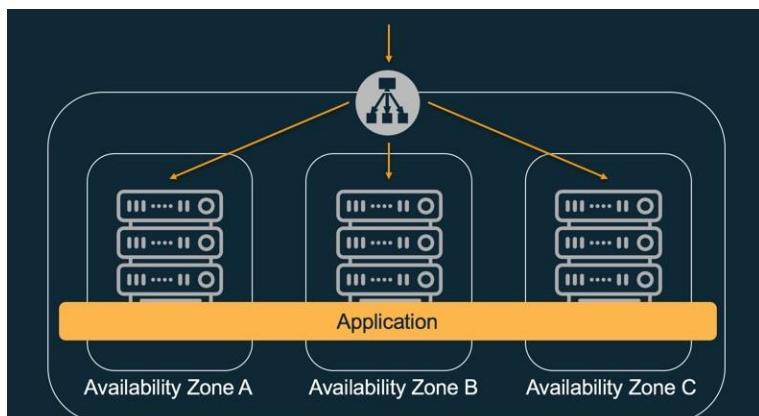
Redundanz

AM Beispiel AWS

The AWS Cloud infrastructure is built around Regions and Availability Zones ("AZs"). A Region is a physical location in the world where we have multiple Availability Zones. Availability Zones consist of one or more discrete data centers, each with redundant power, networking and connectivity, housed in separate facilities.

Redundanz

Redundante Failure-Units
zur Korrektur oder
Abschwächung von Fehlern



Verfügbarkeit

„Jede Region ist vollständig isoliert und besteht aus mehreren AZs, bei denen es sich um vollständig isolierte Partitionen unserer Infrastruktur handelt. Um Probleme besser zu isolieren und eine hohe Verfügbarkeit zu erreichen, können Sie Anwendungen auf mehrere AZs in derselben Region verteilen. Die AWS-Steuerebene und die AWS-Managementkonsole sind zudem über Regionen verteilt und umfassen regionale API-Endpunkte, die so ausgelegt sind, dass sie mindestens 24 Stunden lang sicher funktionieren, wenn sie von den Funktionen der globalen Steuerebene isoliert sind, ohne dass Kunden während einer Isolierung über externe Netzwerke auf die Region oder ihre API-Endpunkte zugreifen müssen.“

Resilienz

- Funktionales Design

Lose Kopplung

Ist ein Grundprinzip zur Vermeidung kaskadierender Fehler und unterstützt die Isolation von Failure Units.

Ein Muster zur Umsetzung von loser Kopplung ist die Verwendung asynchroner Event- oder Nachrichten-basierter Kommunikation. Dadurch wird eine maximale Entkopplung der einzelnen Einheiten erreicht und die Wahrscheinlichkeit sich fortpflanzender Fehler wird minimiert.

Asynchrone Kommunikation benötigt zusätzliche Visualisierungs- und Überwachungsmöglichkeiten für asynchrone Nachrichtennetzwerke, um den Überblick zur Laufzeit nicht zu verlieren.

Synchrone Kommunikation muß Timeout- und Latenzüberwachung betreiben, um Antwortzeitfehler erkennen und behandeln zu können. Antworten, die nicht rechtzeitig ankommen oder synchrone Handshakes machen synchrone Kommunikation ähnlich kompliziert wie asynchrone Kommunikation.

Für synchrone Kommunikation (manchmal gibt es auch dafür gute Gründe) sollte man auf jeden Fall Muster wie Timeout und Circuit Breaker einsetzen, um eine gute Latenzüberwachung zu implementieren (s.a. Bibliotheken wie Hystrix von Netflix).

Ein hilfreiches Muster bei asynchroner Kommunikation ist die sog. Idempotenz.

Loose Kooplung

Wechselseitige Isolation
von Failure-Units

- Funktionales Design

Idempotenz

- Mathematische Definition

Ein Element a einer Menge X heißt idempotent bezüglich einer n -stelligen Verknüpfung $v : X^n \rightarrow X$, $n \in \mathbb{N}$ und $n > 1$ falls gilt:
 $v(a, \dots, a) = a$

Falls $n = 2$ ist und die Verknüpfung (wie etwa bei der Multiplikation üblich) in Potenzschreibweise notiert wird, schreibt sich die Bedingung als $a^*a = a^2 = a$, woraus unmittelbar $a^n = a$ für alle $n \in \mathbb{N}$ folgt (da es für $n = 1$ und $n = 2$ gilt), was die Bezeichnung Idempotenz (lat. für gleiche Potenz) erklärt.

- Kurz : Der Begriff idempotent wird in der Mathematik verwendet, um eine Funktion zu beschreiben, die das gleiche Ergebnis erzeugt, wenn sie auf sich selbst angewendet wird, d.h. $f(x) = f(f(x))$
- Auf die Übermittlung von Nachrichten übersetzt, heißt das : eine Nachricht hat den gleichen Effekt, unabhängig davon, ob sie einmal oder mehrmals empfangen wird. Dies bedeutet, dass eine Nachricht ohne Probleme sicher erneut gesendet werden kann, selbst wenn der Empfänger Duplikate derselben Nachricht bereits mehrfach empfangen hat.

Loose Kooplung

Wechselseitige Isolation
von Failure-Units

- Funktionales Design

Idempotenz

- Ein (verändernder) Aufruf ist idempotent, wenn wiederholte Aufrufe keine zusätzlichen Seiteneffekte haben.
 - Beispiel: Der Aufruf „Addiere 1 auf Wert“ ist nicht idempotent, weil der Wert sich mit jedem Aufruf verändert. Der Aufruf „Setze Wert auf 5“ hingegen ist idempotent. Egal, wie oft ich diesen Aufruf auf einen Wert anwende, das Ergebnis wird immer gleich sein, d. h. es entstehen keine zusätzlichen Seiteneffekte.
- Bei synchronen Aufrufen mit Latenzüberwachung gibt es das Problem, dass im Fall einer zu langsamen Antwort in der Regel nicht unterschieden werden kann, ob der Aufruf gar nicht beim Empfänger angekommen ist oder ob der Empfänger nur zu langsam geantwortet hat.
 - Es ist also unklar, ob die Anfrage verarbeitet wurde oder nicht.
- Verwendet man nicht idempotente Aufrufe, steht man bei zu langsamen Antworten vor dem Problem, dass man nicht oder nur mit sehr großem Aufwand entscheiden kann, ob man den Aufruf noch einmal senden darf oder nicht.
- Verwendet man idempotente Aufrufe, stellt sich dieses Problem nicht
 - Man ruft den Empfänger einfach so oft auf (mit sinnvollen Pausen und Fehlerbehebungsmaßnahmen zwischen den Aufrufen), bis man die Rückmeldung erhält, dass der Aufruf erfolgreich verarbeitet worden ist.
 - Idempotente Aufrufe ermöglichen es, von einer Exactly Once-Kommunikation auf eine At Least Once-Kommunikation zu wechseln.
 - Diese Art der Kommunikation ist wesentlich leichter umsetzbar und hat deutlich geringere Anforderungen an die Kommunikationsinfrastruktur.

Loose Kooplung

Wechselseitige Isolation
von Failure-Units

Resilienz

- Funktionales Design

Lose Kopplung → Idempotenz

- Wie kann man Aufrufe/Nachrichten idempotent bekommen ?
 - Beispiel :

Erhöhe ab 25.01.2019 Anzahl Kinder bei Mitarbeiterin Maria um 1

Mitarbeiterin Maria hat am 25.01.2019 ein Kind bekommen
 - Was ist der Unterschied ?

Erhöhe Anzahl Kinder bei Mitarbeiterin Maria um 1 transportiert die Daten (25.01.2019, 1) und die notwendige Verarbeitungslogik (erhöhe = +) in der Nachricht

Mitarbeiterin Maria hat am 25.01.2019 ein Kind bekommen transportiert die Information – die Verarbeitungslogik liegt beim Empfänger

Diese Information kann ich beliebig oft schicken (bis der Empfänger mir mitteilt, daß er verstanden hat und ich damit aufhören kann)
 - **Der Austausch von Informationen statt Daten ist eine Möglichkeit idempotente Aufrufe/Nachrichten zu erzeugen.**

Loose Kooplung

Wechselseitige Isolation
von Failure-Units

Resilienz

- Funktionales Design

Fallback

Fallback

Endeckte Fehler beheben
oder abschwächen

Es ist notwendig eine klare Strategie zu besitzen mit Fehlern umzugehen (beheben oder abschwächen). Diese Strategie sollte sowohl die Bedürfnisse der Nutzer als auch der Administratoren so gut wie möglich unterstützen. Die Benutzer sollten im Idealfall überhaupt nicht bemerken, dass ein Fehler aufgetreten ist. Die Administratoren sollten einfach Fehlerbehebung durchführen können. Das Mittel der Wahl ist eine vollständig automatisierte Fehlerbehandlung.

Beispiel : einfache Webanwendung

In einer Webanwendung führt eine Anfrage an die Datenbank nach einer Sekunde in einen (erkannten) Timeout. Alle weiteren Anfrage ebenfalls.

Wie könnte die Anwendung reagieren :

- Fehlerseite mit Hinweis auf das Problem (bitte später noch einmal versuchen)
- Nutzerdaten in Caches und Leseanfragen aus den Caches bedienen -Schreibanfragen mit Hinweis
- Schreibanfragen in eine Queue, die asynchron abgearbeitet wird, sobald die Verbindung zur Datenbank wieder verfügbar ist, mit Hinweis, dass der Auftrag angenommen wurde und später verarbeitet wird
- Oder ... ?

Das sind alles Varianten einer **Graceful Degradation Of Service**.

- Funktionales Design
- **Graceful Degradation Of Service**

- Ist die kontrollierte Herabsetzung der Servicequalität.
- Jede der vorher genannten Reaktionen kann je nach Anwendungsfall valide sein.
- Eine Variante ist nie valide → der Anwender sieht die Sanduhr, bis sein Browser nach 5 Minuten einen Netzwerktmeout meldet.
- Was genau gewählt wird, muss geplant sein und ist keine Entscheidung, die ein Entwickler „on the fly“ treffen kann, während er gerade die entsprechende Fehlerbehandlung implementiert.
- Fachanforderungen bestimmen die gewählte Variante, deshalb müssen diese Fragen wie die User Stories und die Basisarchitektur geklärt sein, bevor ein Entwickler den zugehörigen Code schreibt.
- In einem agilen Umfeld klärt und beantwortet das ein Product Owner. Gute Architekten entwickeln mit allen Produkt Ownern gemeinsam einheitliche Strategien über alle Produkte.
- Ist die grundsätzliche Fallback-Strategie festgelegt, kann man die Strategie bei der Umsetzung noch durch zusätzliche Muster wie Escalation Strategy oder Error Handler, für das Implementieren einer automatischen Fehlerbehebung, unterstützen (s.a. Patterns for Fault Tolerant Software Oktober 2007 von Robert Hanmer).
- Graceful Degradation of Service ist also immer **eine geplante Strategie** - aus Fachanforderungen entwickelt.

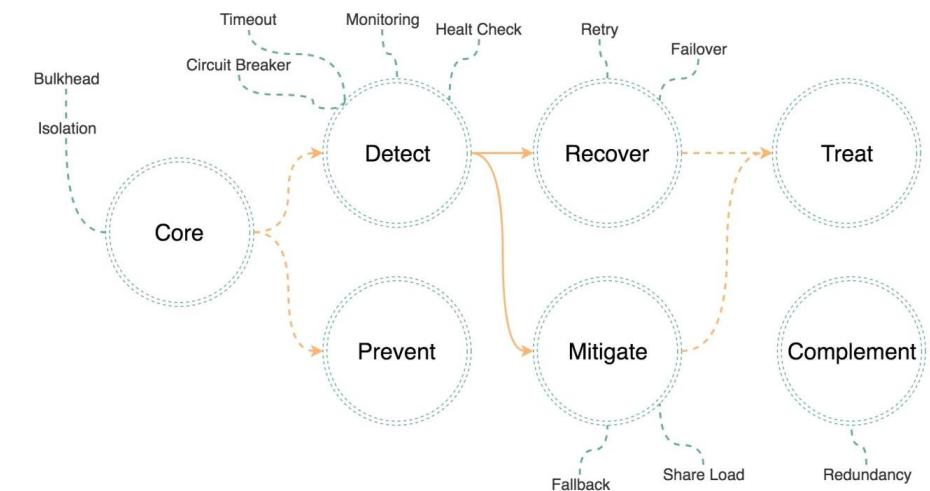
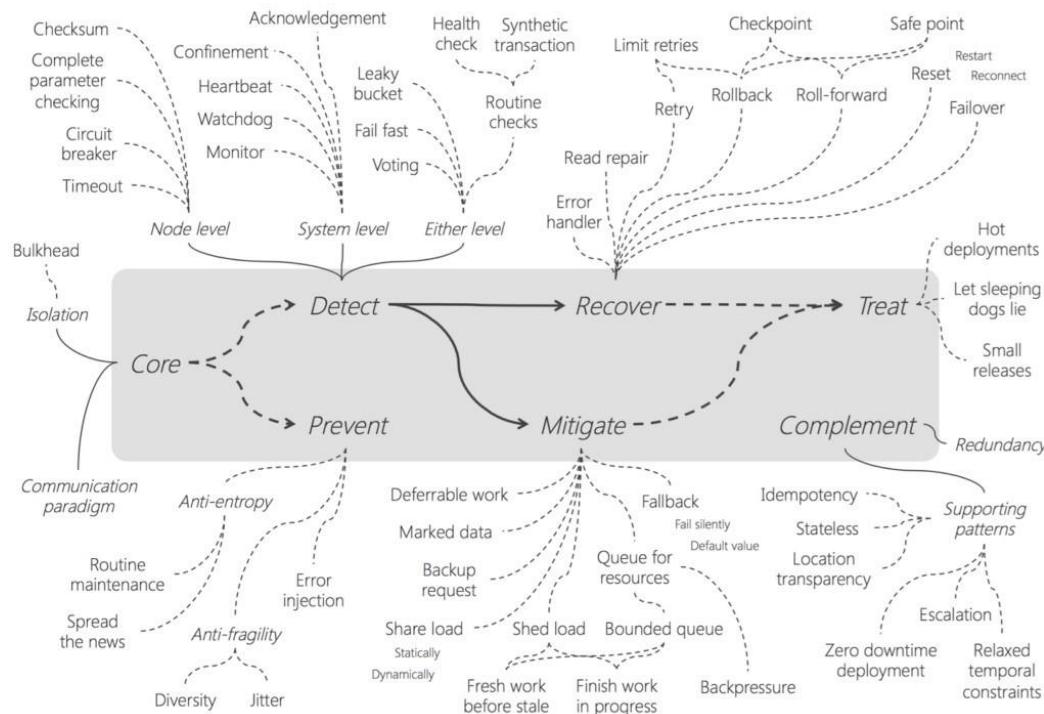
Resilienz

- Funktionales Design
- Es gibt kein allgemeingültiges Rezept



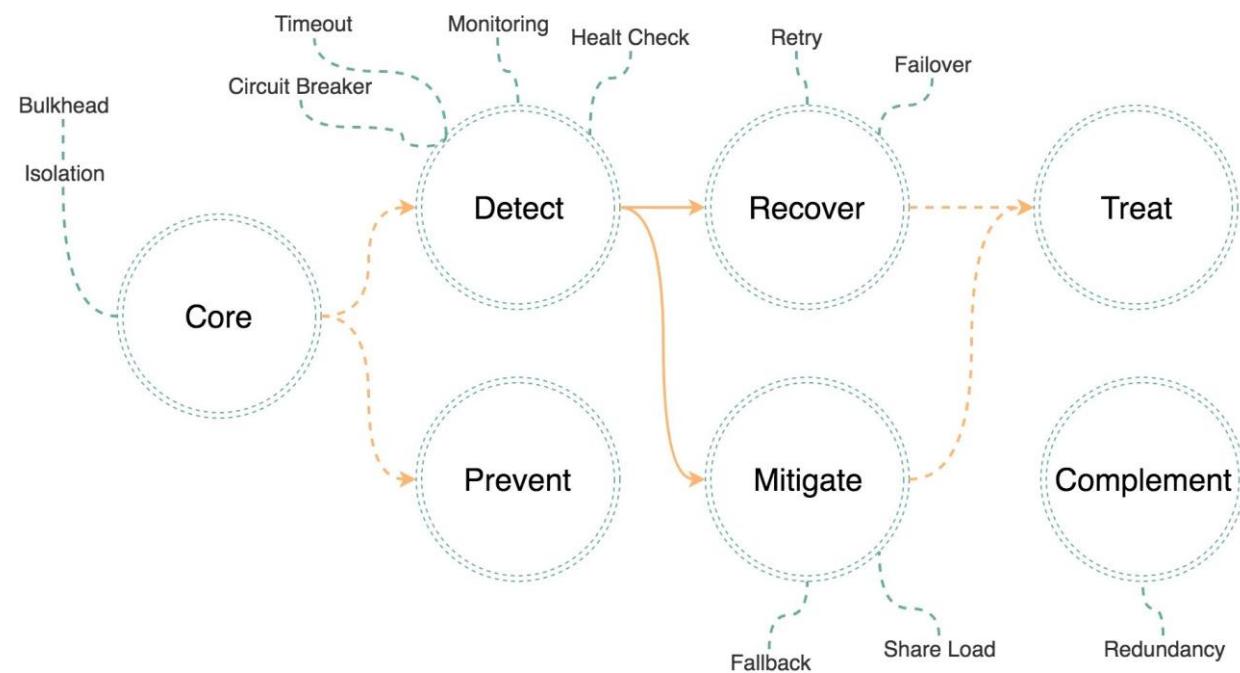
Resilienz

- Die Strategien der Umsetzung
 - Übersicht und Klassifikation von Prinzipien



Resilienz

- Klassifikation
 - Core (zentral)
 - Detect (erkennen)
 - Recover (retten)
 - Mitigate (abmildern)
 - Treat (behandeln)
 - Complement (ergänzen)



Resilienz

- Klassifikation
 - Core

Isolation

Isolation hilft uns dabei, dass entstehende Fehler nicht das gesamte System zum Absturz bringen. Kaskadierende Fehler werden verhindert, es werden Failure Units definiert, in denen diese Fehler auftreten können und behandelt werden.

Bulkhead

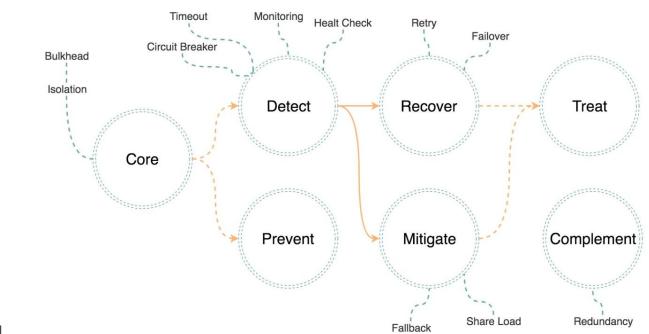
Das Isolation Pattern wird während der Design-Phase angewandt, die Anwendung wird so geschnitten, dass dabei die sogenannten Bulkheads entstehen.

Parameter Checking

Das wird oft unterschätzt und deshalb nicht implementiert. Parameter Checking schützt vor abgebrochenen/fehlerhaften Aufrufen und Rückgabewerten. Die Auswahl der Datentypen ist wichtig, nicht alle Datentypen sind immer geeignet. Postel's Law beachten !

Location Transparency

Entkoppelt den Sender vom Empfänger, da der Sender den konkreten Standort des Empfängers nicht kennen muss. Wird für die Implementierung von transparentem Failover und Redundanz genutzt. Das geschieht mit Hilfe von Dispatchern oder Mappern.



Loose Coupling

Unterstützt die Isolation und verhindert kaskadierende Fehler. Bulkheads sollten immer lose gekoppelt sein. Lose Kopplung kann auf vielfältige Weise erreicht werden.

Asynchronous Communication

Diese Art der Kommunikation entkoppelt den Sender vom Empfänger. Der Sender muss nicht auf die Antwort des Empfängers warten. Sehr hilfreich, um kaskadierende Fehler bei nicht verfügbaren Ressourcen zu vermeiden. Man muss sich allerdings von dem gewohnten Call-Stack verabschieden.

Event Driven Communication

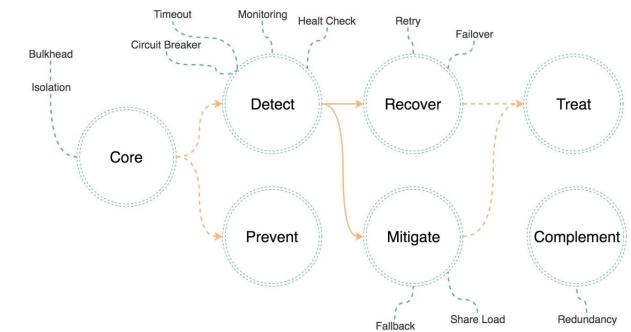
Sehr populäre asynchrone Kommunikation. Geschehnisse (Events) werden dann kommuniziert, wenn sie geschehen. Interessierte Empfänger können sich für Events anmelden.

Resilienz

- Klassifikation
 - Detect

Circuit Breaker

Der Circuit Breaker kontrolliert und überwacht die Aufrufe eines Remote-Service und kann die Aufrufe im Fehlerfall oder bei einem Timeout in einen bereitgestellten Fallback umleiten. Dabei kontrolliert er über einen bestimmten Zeitraum alle Aufrufe. Sollten mehrere Aufrufe scheitern, öffnet sich der Circuit Breaker und lässt für eine definierte Zeit keine weiteren Requests mehr durch. Der Circuit Breaker bedient sich des **Timeout Patterns** und des **Fail Fast Pattern**, die ebenfalls aus dem Resilience Pattern-Baukasten stammen.



Monitoring

Eine verteilte Anwendung, muss zu jeder Zeit über den Status ihrer einzelnen Komponenten informiert sein und ist im besten Fall selbst in der Lage, auf Fehler zu reagieren und entsprechende Fallback-Szenarien bereit zu stellen.

Health Check

Beim Health Check Pattern wird die Verfügbarkeit und Response-Zeit eines Remote Service überwacht. So können im Vorfeld Aufrufe auf eine andere Instanz des Service umgeleitet und Fehler verhindert werden.

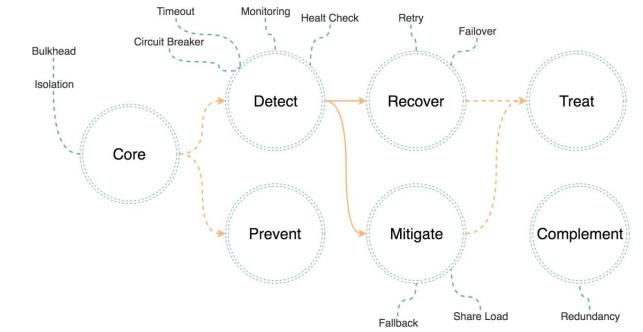
Resilienz

- Klassifikation
 - Prevent

Simian-Army und Chaos Monkeys

Die Simian-Army besteht aus Diensten (Chaos Mokeys), mit denen verschiedene Arten von Fehlern generiert, abnormale Zustände erkannt und die Überlebensfähigkeit eines Systems getestet werden kann. Ziel ist es, das System sicher und hoch verfügbar zu halten.

Quelle : Netflix

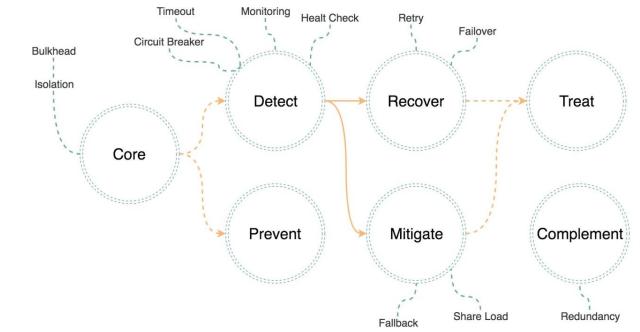


Resilienz

- Klassifikation
 - Recover

Retry

Beim Retry Pattern werden im Fehlerfall ein oder mehrere erneute Aufrufe durchgeführt, in Abhängigkeit des Fehlers.



Failover

Im Failover Pattern wird eine andere Instanz des Service im Fehlerfall aufgerufen, dies setzt natürlich die Redundanz eines Service voraus. Wurden hier Fehler im Design der Anwendung gemacht, werden diese bei der Implementierung eines Failovers sichtbar – wenn auch zu spät.

Resilienz

- Klassifikation
 - Mitigate

Fallback

Das Fallback Pattern gehört wohl zu den schwierigsten und aufwändigsten Pattern, nicht weil es sehr herausfordernd ist, es lässt sich technisch schnell abhandeln und versetzt das System in einen stabilen Zustand. Das Ziel dieses Patterns ist es, den Benutzer nicht merken zu lassen, dass das Backend in Trümmern liegt.

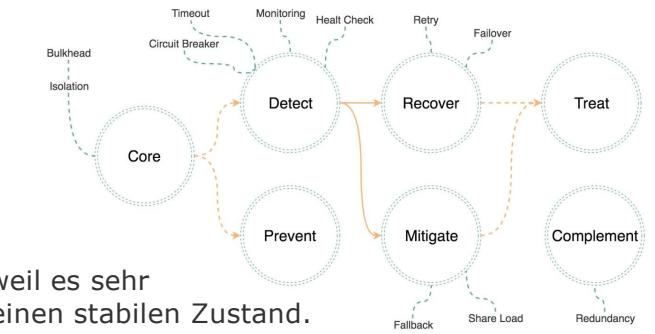
Daher ist es wichtig, bei der Definition von Fallbacks eng mit dem Fachbereich zusammenzuarbeiten und genau zu definieren, wie der Fallback auszusehen hat.

- Sollen und können wir auf geachte Ergebnisse zurückgreifen?
- Sollen wir im Fallback von der synchronen auf eine asynchrone Verarbeitung wechseln?
- Sollen wir den Request zwischenspeichern und später verarbeiten?

Diese und andere Fragen gilt es zu beantworten. Die Antwort im Fallback muss **fachlich** definiert werden. Hier gilt es, die richtigen Entscheidungen zu treffen und sich auch die notwendige Zeit dafür einzuräumen.

Share Load

Beim Share Load Pattern wird die Last im System verteilt, dies kann durch eine statische, im Code implementierte Logik oder dynamisch erfolgen. Ein Client-Side-Load-Balancer bietet sich in Verbindung mit einer Service Discovery an. Die Service Discovery liefert dem Load-Balancer die wichtigen Informationen über verfügbare Instanzen eines Service, die dann vom Load-Balancer via Round-Robin-Verfahren mit Requests versorgt werden. So wird die Last im System verteilt und es kann auf ausgefallene Instanzen reagiert werden.



Resilienz

- Klassifikation
 - Complement

Redundancy

Bei der Redundanz lassen sich die verschiedensten Strategien entwickeln, wie

- Clustering betreiben
- einen Load-Balancer einbauen oder
- eine Service Discovery in Verbindung mit einem Client-Side-Load-Balancer.

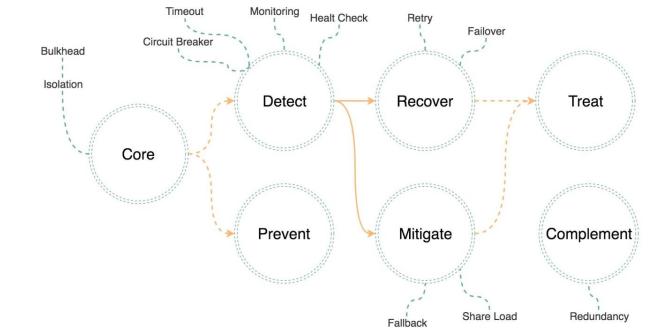
Das Ziel bleibt dabei immer gleich, die Applikation durch Redundanz einzelner Instanzen abzusichern.

Itempotency

siehe Folien zu 100% Verfügbarkeit

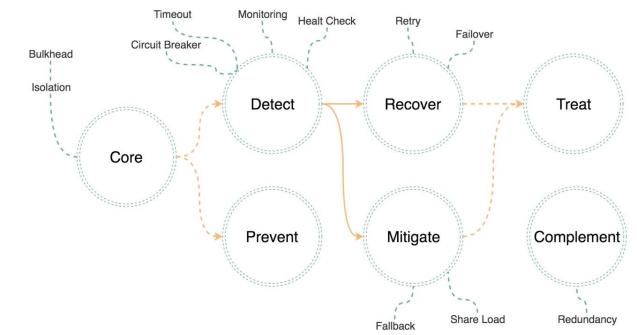
Stateless

Stateless unterstützt auch Location Transparency (Core). Failover ist sehr schwer, wenn Stateless nicht implementiert ist. Das Verschieben von Services auch. Dieses Prinzip ist fundamental für resiliente Systeme und für die Skalierbarkeit von Systemen.



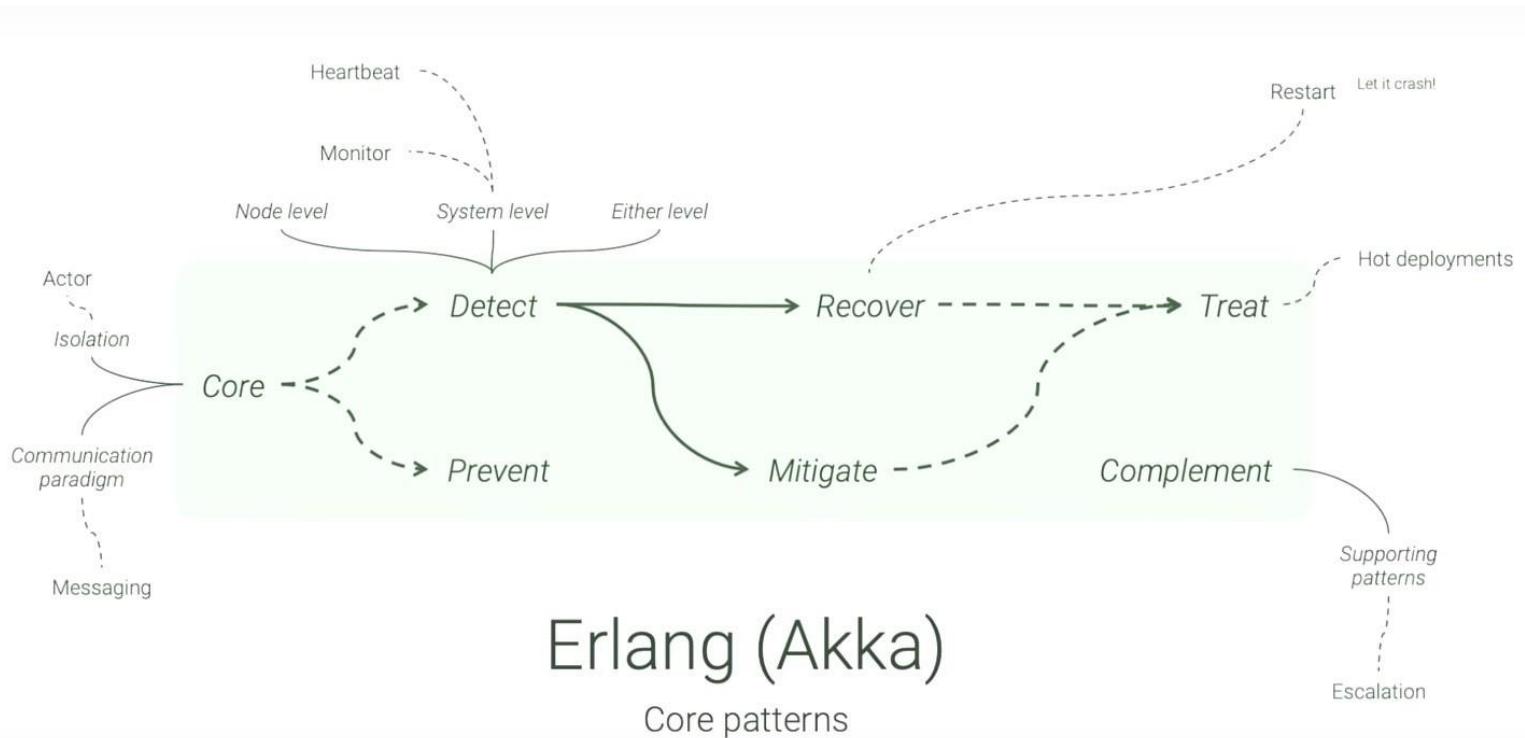
Resilienz

- Hilfsmittel
 - Resilience Patterns
 - Regeln
 - Patterns sind optional, nicht verpflichtend
 - Nicht zu viele Patterns nutzen
 - Jedes Pattern erhöht die Komplexität
 - Komplexität ist der Feind von Robustheit
 - Jedes Pattern kostet Geld in Dev & Ops
 - Das Resilienz Budget ist i.a. begrenzt
 - Suche nach ergänzenden Pattern



Resilienz

- Hilfsmittel
 - Resilience Patterns
 - Beispiel



Telefon Switches,
die im Betrieb gewechselt
werden können.

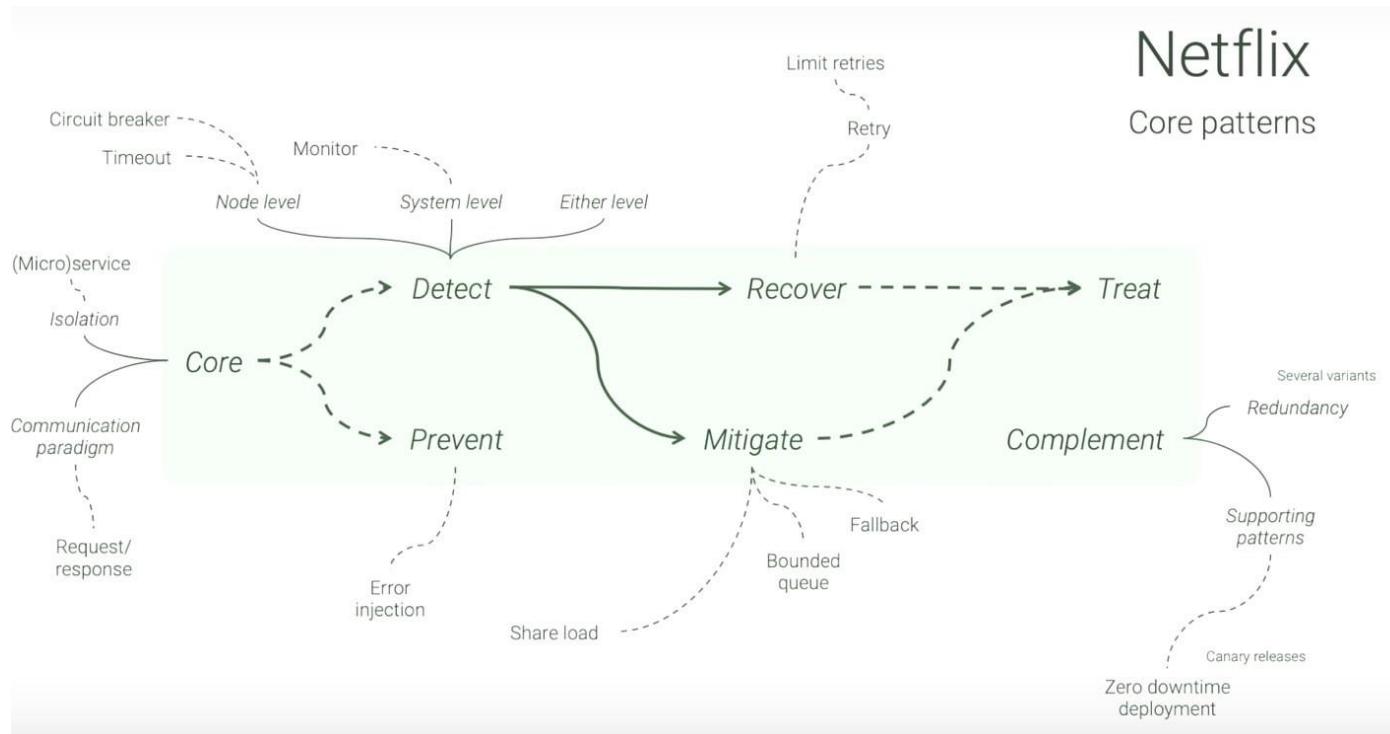
NoSQL Datenbank

Ein paar einfache Patterns !

Quelle : Uwe Friedrichsen

Resilienz

- Resilience Patterns
 - Beispiel



Netflix

Core patterns

Hystrix Bibliothek

Ein paar einfache Patterns !

Zu Spitzenzeiten werden mehr als 100.000 Server benutzt !
Generiert ca. die Hälfte des amerikanischen Downstream Volumens !

Quelle : Uwe Friedrichsen

Resilienz

• Die Strategien bei der Einführung

– Erfahrungen aufheben/bewahren

- Alle 5 Jahre werden immer wieder alte/bekannte Themen neu diskutiert (Sau durch das Dorf getrieben)
- Das ist speziell in der IT der Fall
- Warum ist das so ?
 - Wir lieben technologische Hypes (new & shiny stuff)
 - Was nicht neu ist, halten wir für nutzlos



- **Die 7 Herausforderungen bei der Einführung**

7. Die gesammelten Erfahrungen aufheben/bewahren
 - Wir sollten unser Wissen/unsere Erfahrungen alle 5 Jahre neu entdecken
 - Resilientes Software Design gibt es seit 30 Jahren, das ist nicht neu !
 - Wir sollten nicht stolz darauf sein, dauernd alles zu vergessen/vergessen zu wollen
 - Sie  können das ändern !



6) Softwarearchitektur Design

- Ist einer der wichtigsten Schritte erfolgreiche Software Systeme zu entwickeln
- Es gibt 2 wichtige Ansätze für ein Software Design
 - Top-down
 - Bottom-up
- Software Design ist ein komplexes Thema, dafür wurden Design Prinzipien und Lösungen zur Unterstützung entwickelt
- Design Prozesse leiten Architekten bei Erstellen von Architekturen, die den Anforderungen, Qualitätsattributen und Einschränkungen genügen

6) Softwarearchitektur Design

- Wir werden folgendes betrachten
 - a) Software Architektur Design
 - b) Typen von Software Architektur Design
 - c) Attribute Driven Design (ADD)
 - d) Microsofts Technik für Architektur und Design
 - e) Architecture Centric Design Method (ACDM)
 - f) Architecture Development Method (ADM)
 - g) Den Fortschritt einer Software Architektur verfolgen

6a) Software Architektur Design

- Ist die notwendige Grundlage für Entscheidungen zu Software Systemen
- Basiert auf functional und non-functional requirements, Qualitätsattributen und Einschränkungen
- Definiert die Strukturen, die Elemente und die Beziehungen zwischen Elementen eines Software Systems und dokumentiert sie
 - Liefert die Beschreibung der Funktionalitäten und Interaktionen zwischen Elementen
- Ist signifikant für die Qualität und den langfristigen Erfolg eines Software Systems
- Ist die technische Hilfestellung für die Entwicklung
 - Kann sich während der Entwicklung ändern,
wenn sich Anforderungen oder Qualitätsattribute ändern
- Ist ein kreativer Prozess
 - Teilweise hoher Spaßfaktor
- Ist ein kollaborativer Prozess
 - Je mehr Wissensträger dazu beitragen, um so besser
- „Perfect is the enemy of good“

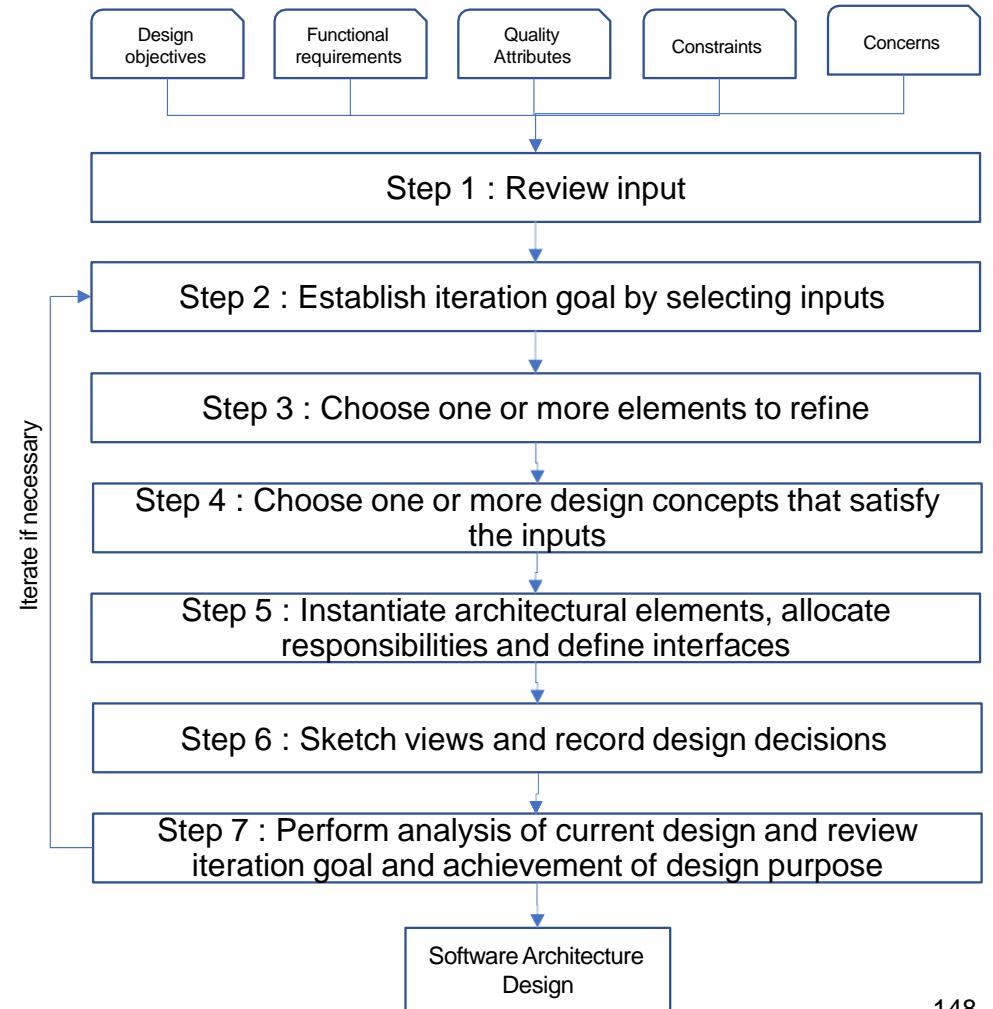
6b) Software Architektur Design

- Typen
 - Top-down
 - Bei großen und komplexen Projekten, bei großen Teams, bei Enterprise Software, die Business Domäne ist bekannt
 - Bottom-up
 - Punkte von top-down negiert
 - Greenfield
 - Grüne Wiese Ansatz
 - Brownfield
 - Aufbauend auf einem bereits bestehenden System

6c) Attribute Driven Design

- Schritt 1 Review Input

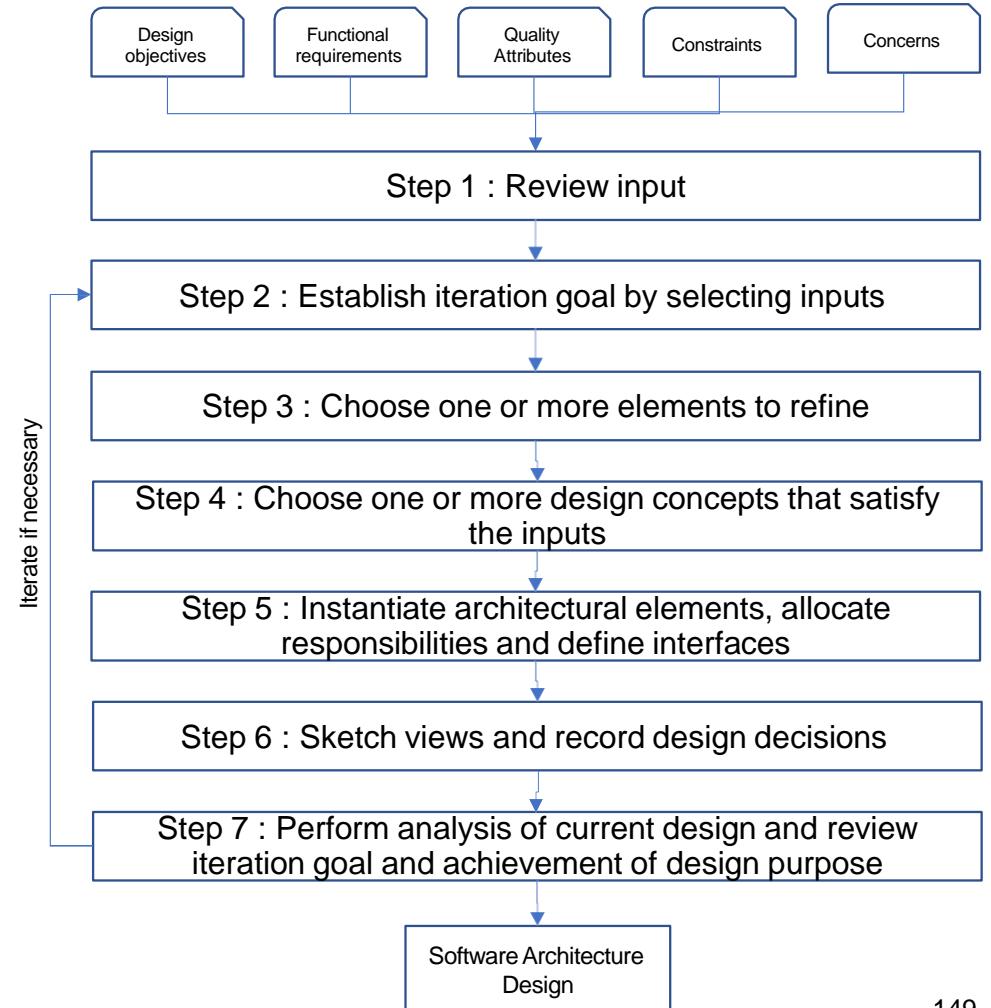
- Im Schritt 1 wird überprüft, ob alle notwendigen Unterlagen/Eingaben vorhanden und vollständig sind
- Der Input sind die folgenden Treiber
 - Design Ziele/Vorgaben
 - Functional Requirements
 - Qualitätsattribute
 - Einschränkungen
 - Belange
- Oft liegt auch schon eine Architektur vor, oder zumindest Teile davon



6c) Attribute Driven Design

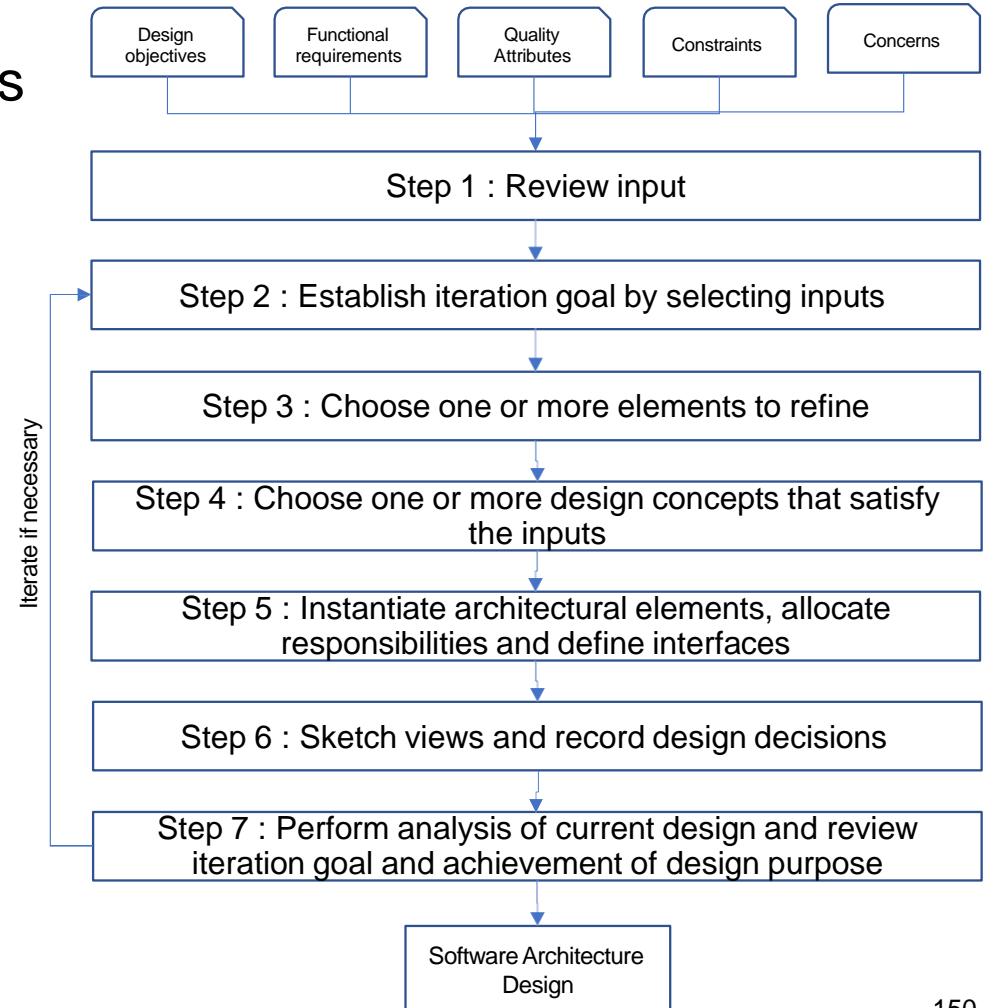
- **Schritt 2 Establish iteration goals**

- Nach Schritt 1 werden 1 – N Iterationen durchgeführt werden, die immer mit Schritt 2 beginnen
- Mit einer agilen Entwicklungsmethode werden immer mehrere Iterationen durchgeführt (Sprints)
- Das Designziel jeder Iteration wird zu Beginn der Iteration festgelegt
- Wichtige Frage : welches Designziel wollen wir in dieser Iteration lösen
- Jedes Designziel muss mit einem oder mehreren Eingaben in Beziehung stehen und stehen in dieser Iteration im Fokus



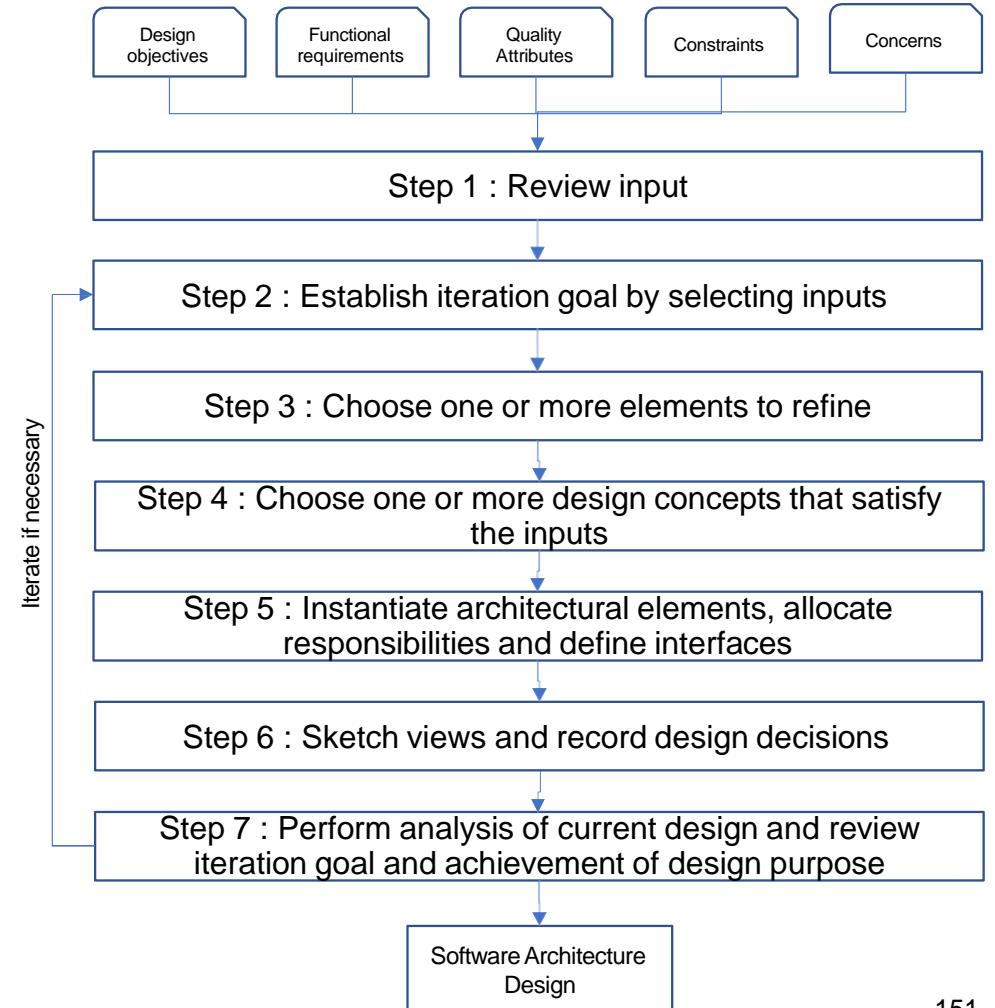
6c) Attribute Driven Design

- Schritt 3 Choose one or more elements
 - Die Teilelemente des Gesamtsystems, auf die in dieser Iteration der Fokus liegen soll, werden ausgewählt
 - Diese Elemente werden detaillierter betrachtet



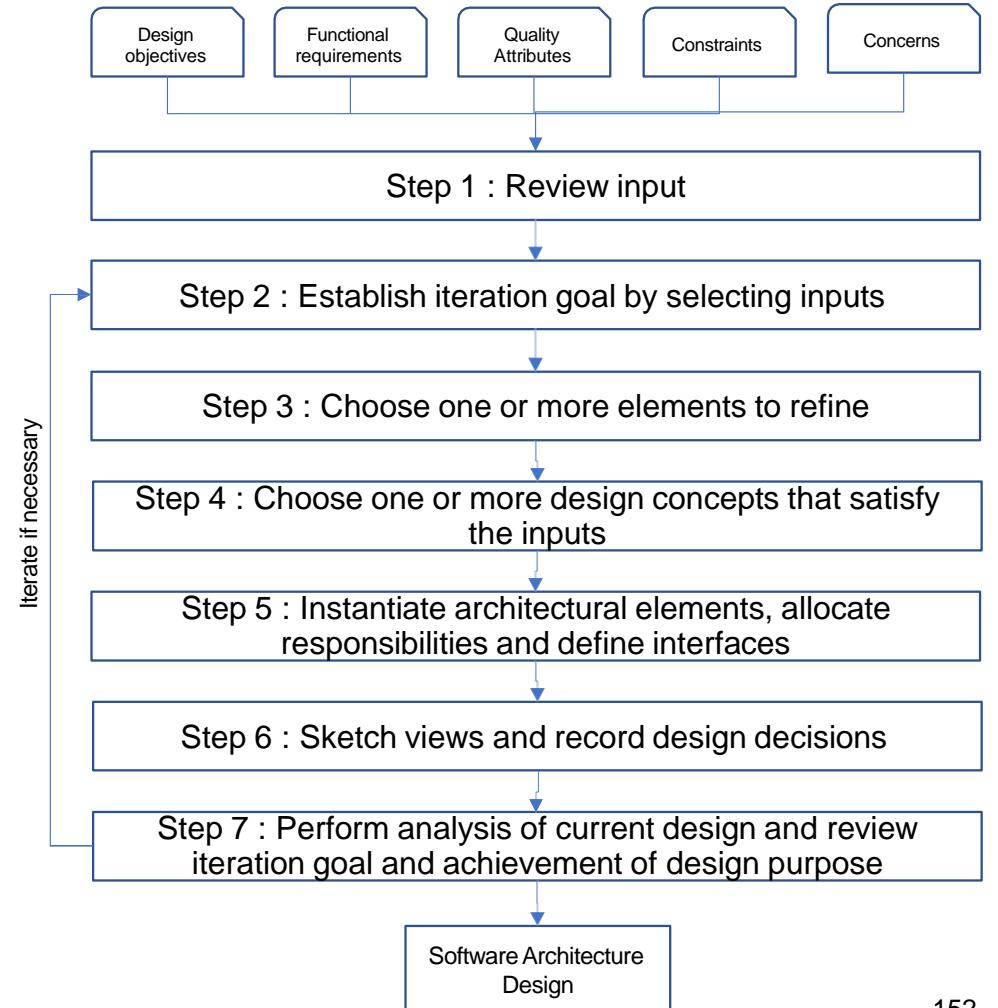
6c) Attribute Driven Design

- Schritt 4 Choose one or more elements to refine Design Konzepte zum Erreichen des Iterationsziels werden auf der Basis, der in Schritt 3 gewählten Elementen und der Eingaben, bestimmt
- Design Konzepte sind Design Prinzipien, Architektur Pattern, Referenzarchitekturen, taktische Maßnahmen und extern entwickelte (zugekaufte) Software/Lösungen



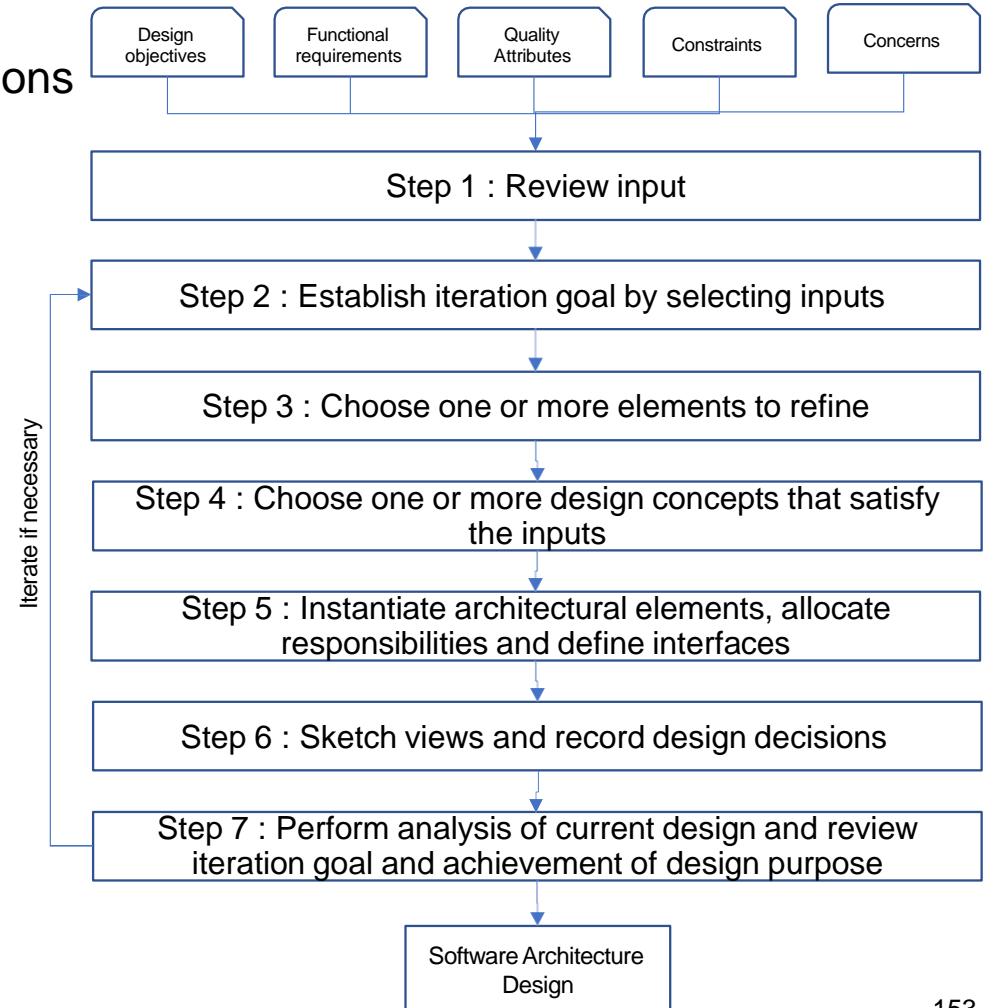
6c) Attribute Driven Design

- Schritt 5 Instantiating architectural elements
 - Die Verantwortlichkeiten und Schnittstellen der Elemente werden auf Basis der gewählten Designkonzepte analysiert



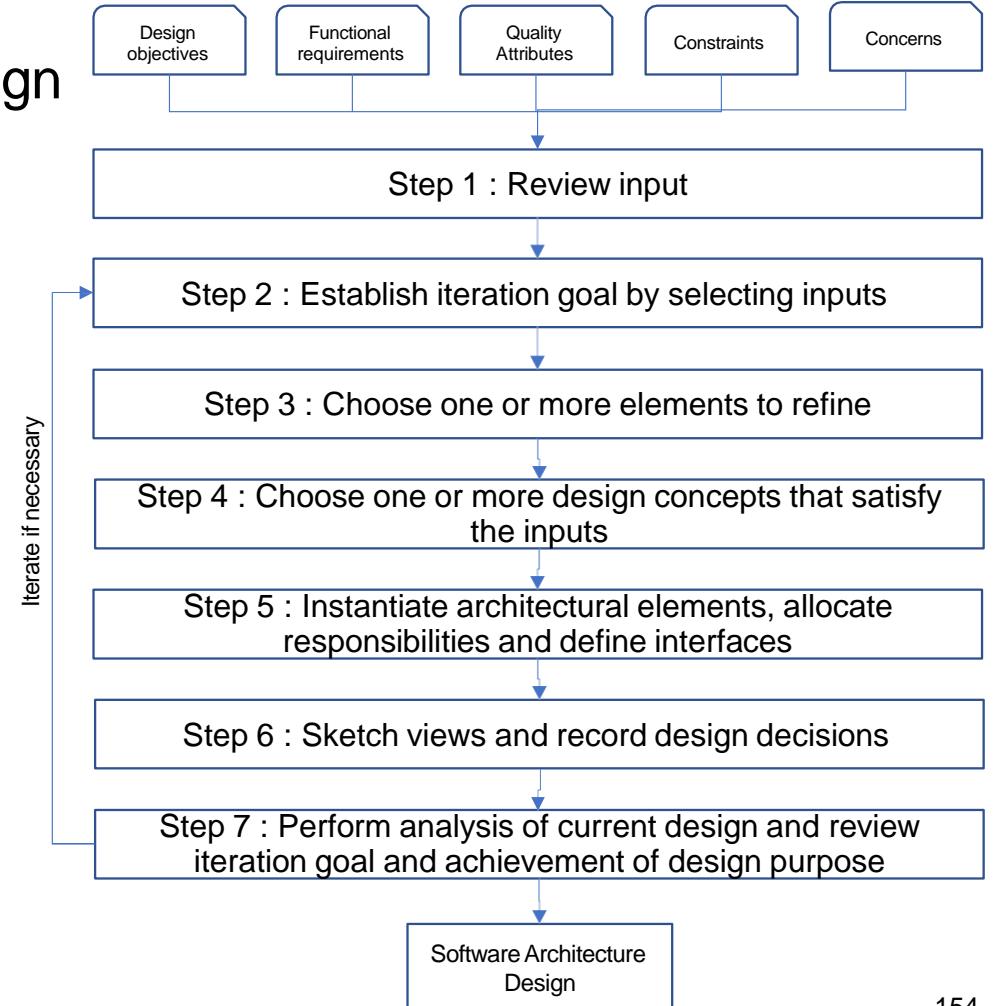
6c) Attribute Driven Design

- Schritt 6 Sketch views and record design decisions
 - Sichten auf die Lösung müssen für die verschiedenen Stakeholder skizziert und dokumentiert werden
 - Die Begründung des gewählten Designs wird auch dokumentiert
 - Die erstellten Skizzen beinhalten alle Design Entscheidungen
 - Die Skizzen sollten nicht zu formal und detailliert sein



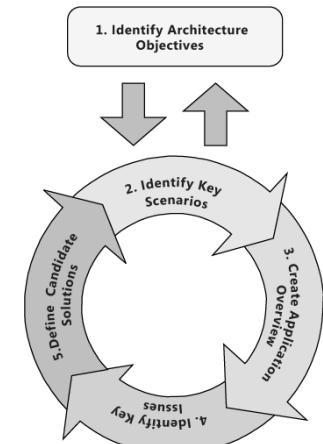
6c) Attribute Driven Design

- Schritt 7 Perform analysis of current design
 - Der Architekt und evtl. weitere Teammitglieder analysieren das Design
 - Sie prüfen auf Korrektheit und Übereinstimmung mit den Zielen der Iteration
 - In diesem Schritt wird überprüft und festgelegt, ob weitere Iterationen notwendig sind
 - Sind keine Iterationen mehr notwendig, ist das Software Architektur Design vollständig



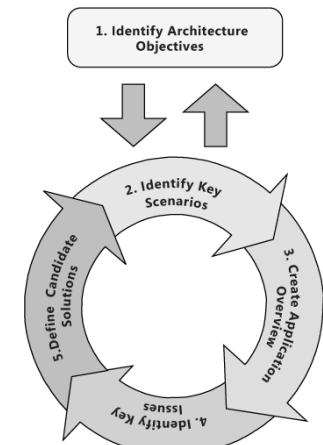
6d) Microsofts Technik für Architektur und Design

- Microsofts Agile Architecture Methode ist ein iterativer und incrementeller Ansatz für ein Software Architektur Design
- Zusammengefasst es ist eine Technik
 - Setzt den Rahmen und fokussiert die Architekturarbeit
 - Benutzt Szenarien um das Design zu lenken und potentielle Lösungen zu beurteilen
 - Hilft bei der Auswahl von Anwendungstypen, Plattformen, Architekturstile und Technologien
 - Hilft beim schnellen Betrachten möglicher Lösungen
 - Hilft beim Aussuchen von möglichen Pattern



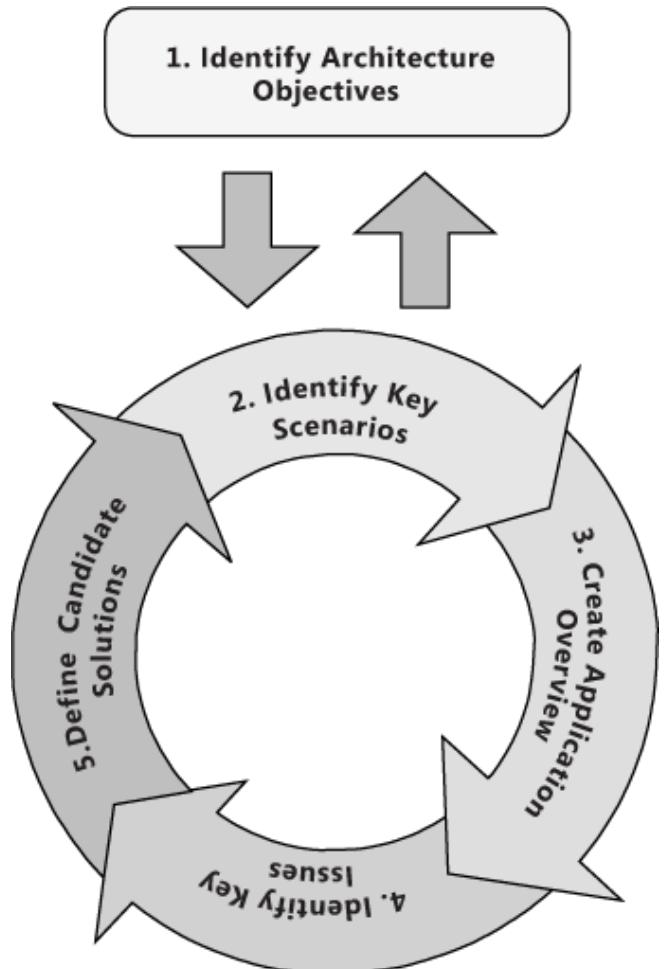
6d) Microsofts Technik für Architektur und Design

- Wichtigste Eingaben (Input) in den Prozess
 - Use cases and usage scenarios
 - Functional requirements
 - Non-functional requirements
(quality attributes such as performance, security, and reliability)
 - Technological requirements
 - Target deployment environment
 - Constraints
- Ergebnis des Prozesses
 - Architecturally significant use cases
 - Architecture hot spots
 - Candidate architectures
 - Architectural spikes



6d) Microsofts Technik für Architektur und Design

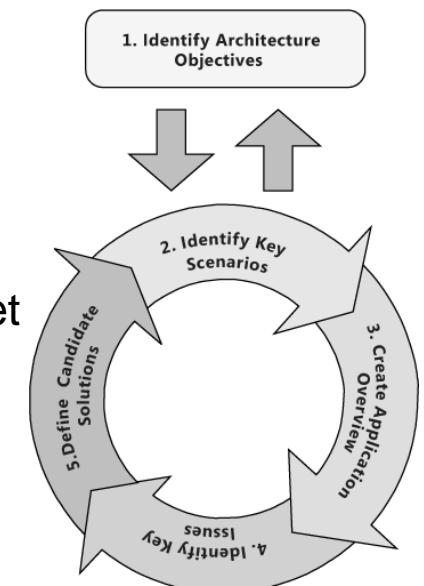
- Zusammenfassung der Schritte
 - Step 1. Identify Architecture Objectives.
 - Step 2. Identify Key Scenarios.
 - Step 3. Create an Application Overview.
 - Step 4. Analyze Key Hot Spots.
 - Step 5. Create Candidate Solutions.



6d) Microsofts Technik für Architektur und Design

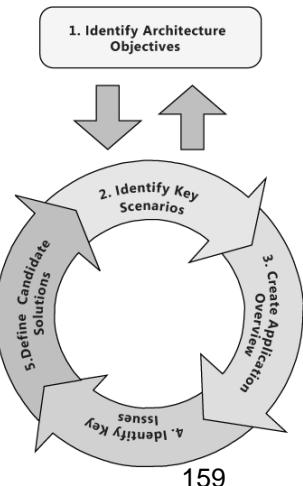
- Schritt 1 Identify Architecture Objectives

- In diesem Schritt werden die Ziele, die mit der Architektur erreicht werden sollen, festgelegt
- Die Festlegung der Ziele sichert die Fokusierung auf die richtigen Problemstellungen
- Mögliche Ziele könnten sein
 - Prototyperstellung
 - Identifizierung von wichtigen technischen Risiken
 - Testen von möglichen Abläufen
 - Modell- und Verständnisabgleich
- Die notwendige Zeit und die benötigten Ressourcen werden hier berechnet



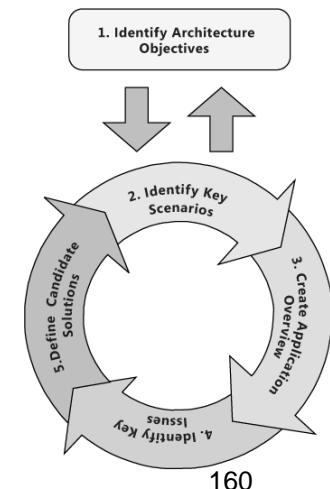
6d) Microsofts Technik für Architektur und Design

- Schritt 2 Identify Key Scenarios
 - Identifikation relevanter Szenarien
 - um das Design auf die wichtigsten Punkte zu konzentrieren
 - um geeignete Lösungen zu bestimmen
 - Identifikation für die Architektur signifikanter Use Cases, die folgenden Kriterien genügen
 - Sie sind wichtig für den Erfolg und die Akzeptanz der zu erstellenden Lösung
 - Sie wenden an und prüfen genug vom Design, um den Nutzen für die Architektur feststellen zu können
 - Schlüsselszenarien von User Stories, Business Stories und System Storys werden aufgezeichnet



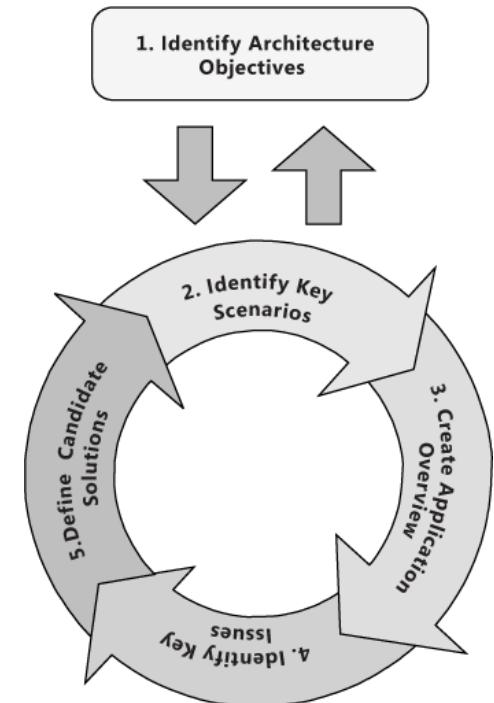
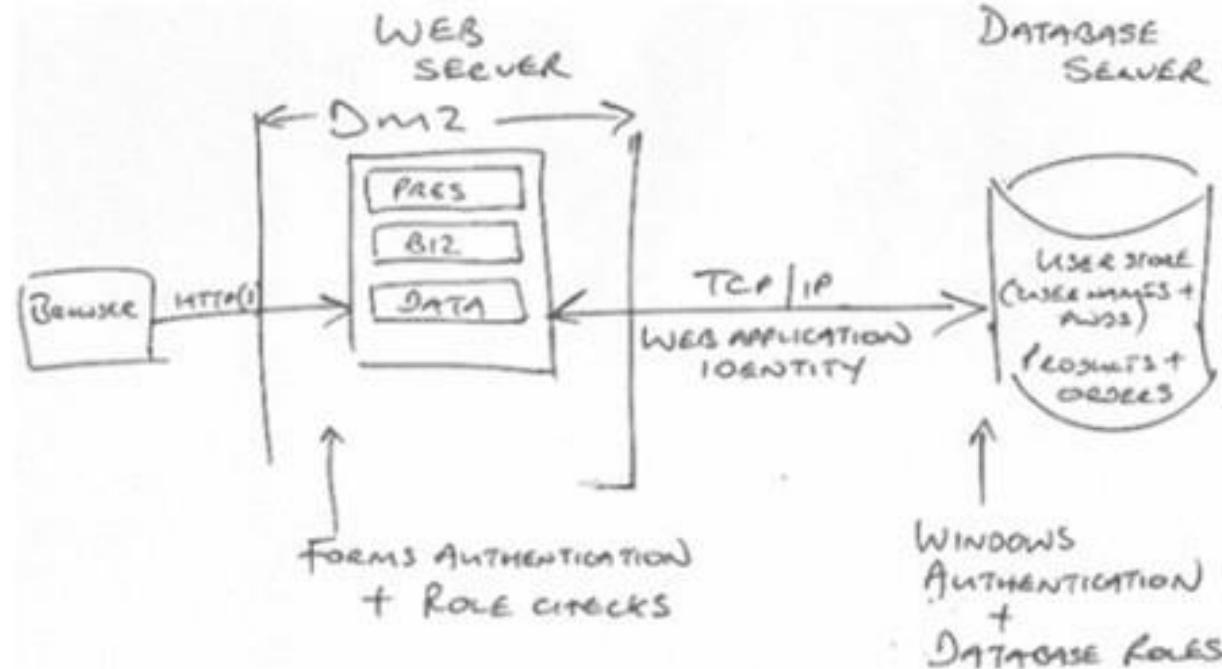
6d) Microsofts Technik für Architektur und Design

- Schritt 3 Create an Application Overview
- Die Übersicht über die Anwendung macht eine Architektur mehr real, indem man sie mit echten Einschränkungen und Entscheidungen verbindet
- Eine Anwendungsübersicht wird wie folgt erstellt
 - Bestimme den Anwendungstyp (Mobile, Rich Client, Internet application, Service, Web application, oder Kombinationen davon)
 - Verstehe die Einschränkungen beim Ausrollen
 - Verstehe die Zielumgebung und erkenne den Einfluss auf die Architektur
 - Identifiziere wichtige Architekturstile (SOA, client/server, layered, message bus, Kombinationen)
 - Bestimme relevante Technologien basierend auf dem Anwendungstyp, dem Architekturstil und den Einschränkungen
- Ein guter Test für das Verständnis und einen Überblick liefert ein sog. Whiteboard



6d) Microsofts Technik für Architektur und Design

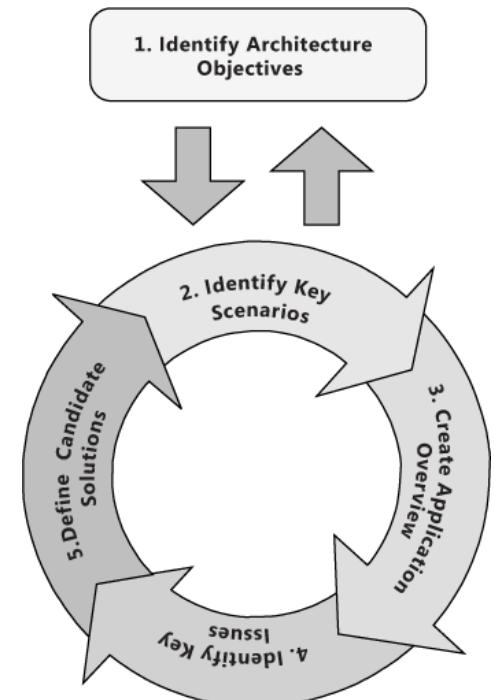
- Schritt 3 Create an Application Overview
 - Whiteboard



6d) Microsofts Technik für Architektur und Design

- Schritt 4 Analyze Key Issues

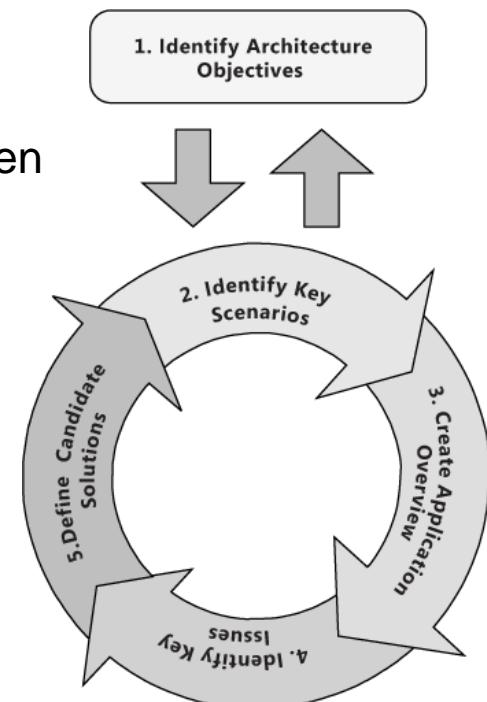
- Identifiziere die wichtigsten Hindernisse der (in der) Architektur
- Hindernisse lassen sich typischerweise auf Qualitätsattribute abbilden
- Gibt Hinweise, wo typische Fehler bei der Anwendungsarchitektur gemacht werden können



6d) Microsofts Technik für Architektur und Design

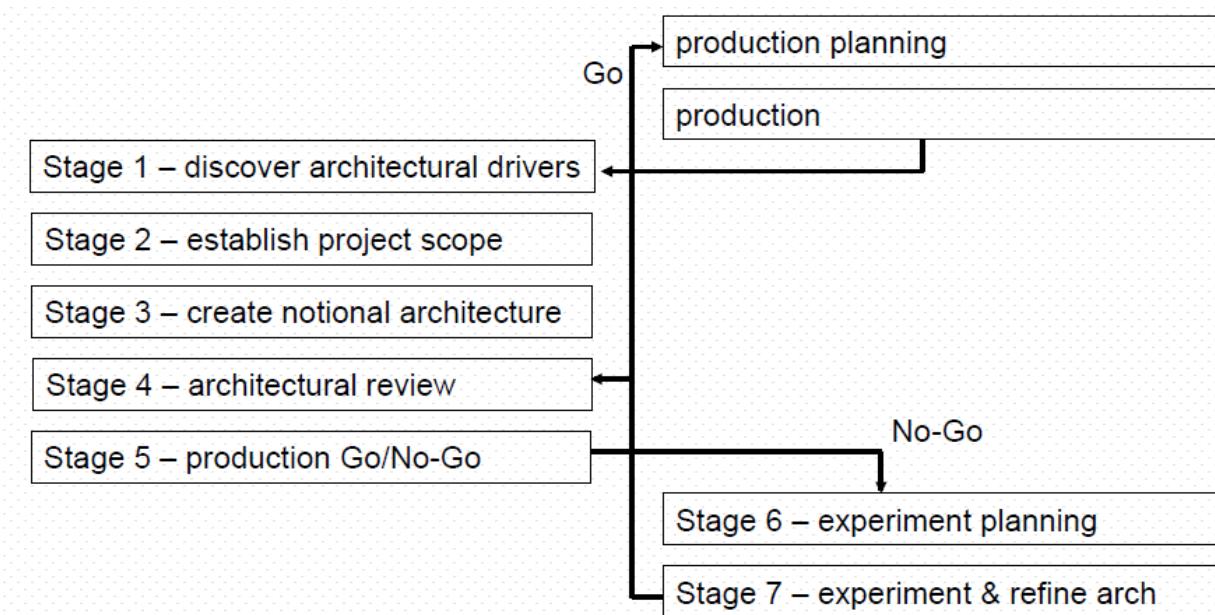
- Schritt 5 Define Candidate Solutions

- Lösungskandidaten werden definiert
- In der ersten Iteration entsteht eine neue Architektur in weiteren Iterationen wird die Architektur angepasst/erweitert/verbessert
- Nachdem geeignete Kandidaten in die Gesamtarchitektur aufgenommen wurden, kann die Architektur geprüft und ausgewertet werden
- Wird in diesem Schritt festgestellt, daß weitere Architekturarbeiten notwendig sind, geht es in die nächste Iteration (mit Schritt 2)



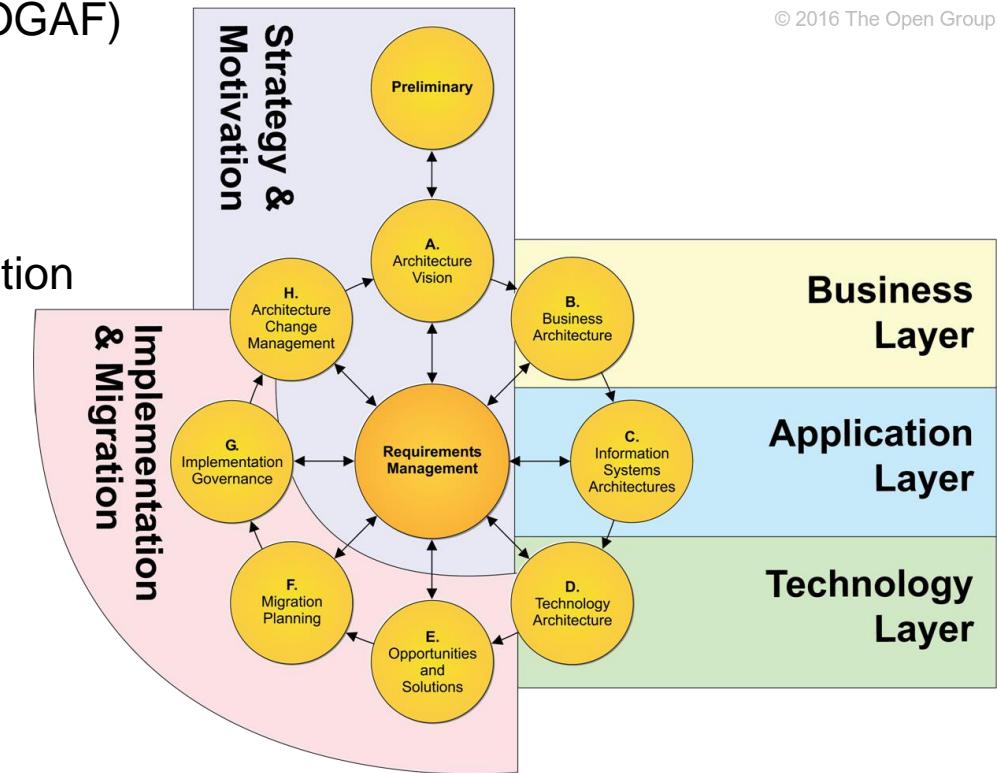
6e) Architecture Centric Design Method

- Methode mit Fokus auf das Produkt und den Betrieb
- Die Architektur kann in Schritt 4 bei jeder Iteration geändert werden und ist deshalb in der ersten Iteration i.a. nicht vollständig



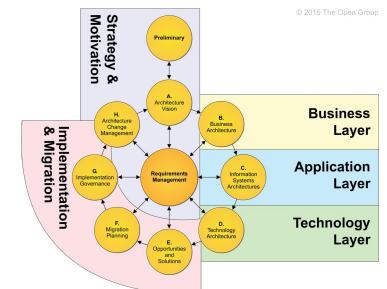
6f) Architecture Development Method

- ADM ist Teil eines größeren Frameworks
 - The Open Group Architecture Framework (TOGAF)
 - Framework für Enterprise Architektur
 - ADM besteht aus acht Phasen A-H und einer sog. Preliminary
 - In der Preliminary bereitet sich eine Organisation auf die Einführung von ADM vor (Regeln, Frameworks u.ä.)
 - Jede Phase betrachtet kontinuierlich die Anforderungen



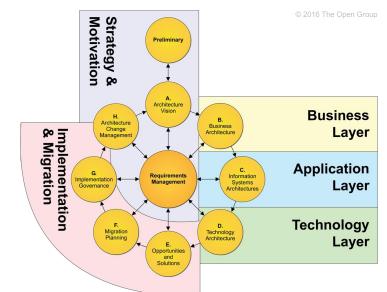
6f) Architecture Development Method

- Phase AArchitecture Vision
 - Hier wird die Vision der Architektur mit ihren Fähigkeiten und ihrem Wert entwickelt
 - Teil dieser Phase ist das Festlegen von Umfang, Ziele des Business, Treibern, Einschränkungen, Anforderungen, Rollen, Verantwortlichkeiten und Zeitplänen
 - Alles wird in einem sog. Statement of Architecture Work (SAW) festgehalten
 - Dieses Dokument enthält typischerweise
 - Projektantrag mit Hintergrundinformationen
 - Projektbeschreibung mit Projektumfang (Scope)
 - Übersicht der Architekturvision
 - Rollen, Verantwortlichkeiten, Lieferungen
 - Projekt- und Zeitplan



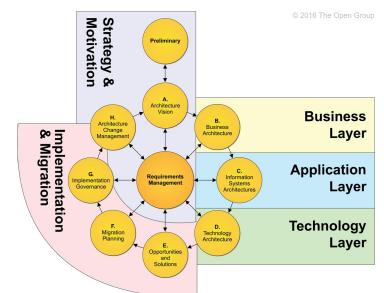
6f) Architecture Development Method

- Phase B Business Architecture
 - Eine der TOGAF Domänen
 - Daten, Anwendungen, Business und Technologie sind die 4 maßgeblichen Domänen
 - Hier wird die Business/Service Strategy der Organisation beschrieben
 - Hier wird das Business und das zugehörige Umfeld beschrieben
 - Hier wird die Ziel-Businessarchitektur festgelegt
 - Um das Ziel zu erreichen und eine Roadmap dahin zu erstellen gibt es die Schritte
 - Die aktuelle (Business)Architektur verstehen
 - Die zukünftige (Business)Architektur aufstellen
 - Die Lücken zwischen beiden Architekturen bestimmen
 - Eine Roadmap für eine Überführung erstellen



6f) Architecture Development Method

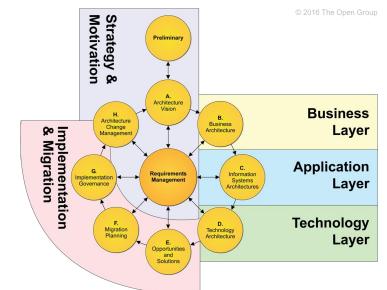
- Phase C Information Systems Architecture
 - Eine der TOGAF Domänen
 - Die Datenarchitektur beschreibt die Daten einer Organisation und wie sie verarbeitet werden
 - Phase A und Phase B bestimmen mögliche Änderungen in Datenmodellen
 - Auch in dieser Phase wird der Unterschied voher/nachher ausgearbeitet



© 2016 The Open Group

6f) Architecture Development Method

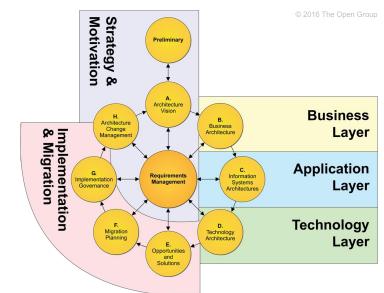
- Phase D Technology Architecture
 - Eine der TOGAF Domänen
 - Bezieht die Infrastrukturkomponenten in die Architekturbetrachtung mit ein
 - Hardware und Software
 - Auch hier wird eine Gap Betrachtung gemacht
 - Eventuelle Auswirkungen oder notwendige Änderungen auf die Infrastruktur wird in eigenen Projekten berücksichtigt



© 2016 The Open Group

6f) Architecture Development Method

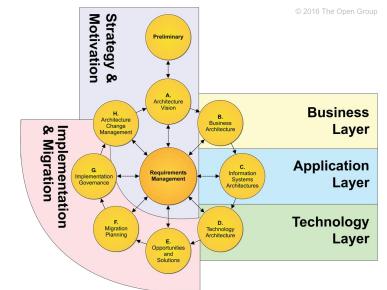
- Phase E Opportunities and solutions
 - Der Output von Phase A,B,C,D wird in eine gemeinsame Roadmap konsolidiert
 - Mögliche Komponenten werden identifiziert
 - Lösungen werden konzipiert
 - Inkrementelle Ansätze können definiert werden



© 2016 The Open Group

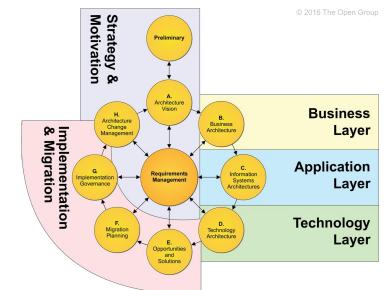
6f) Architecture Development Method

- Phase F Migration Planning
 - Die Implementierung wird geplant
 - Projekte werden definiert
 - Change Management und Projekt Management werden eingebunden



6f) Architecture Development Method

- Phase G Implementation Governance
 - Architekten begleiten und steuern den Entwicklungsprozess
- Phase H Architecture Change Management
 - Architekten begleiten und steuern den Change Management Prozess im Unternehmen



© 2016 The Open Group

6g) Den Fortschritt einer Software Architektur verfolgen

- Während eines Entwicklungsprozesses will man zu jeder Zeit über den Fortschritt informiert sein
- Dazu gibt es seit der Entwicklung agiler Entwicklungsmethoden folgende Hilfsmittel
 - (Product)Backlog
 - Liste aller Bugs und Features eines Softwareprodukts
 - Während der Entwicklungszyklen (Sprints) ist diese Liste die Grundlage aller Entwicklungen
 - Dazu können einzelne Einträge von den agilen Teams priorisiert werden
 - DIVE Kriterien
 - » Dependencies
 - » Insure against risks
 - » Business Value
 - » Estimated effort
 - Backlogs verändern sich dynamisch

7) Software Architektur Pattern

- Pattern spielen in der Softwarearchitektur eine große Rolle
- Viele Probleme im Software Design haben bereits erprobte Lösungen
- Erfahrene Architekten erkennen Einsatzmöglichkeiten von verfügbaren Patterns
- Wir betrachten
 - a) Was ist ein Software Architektur Pattern
 - b) Verschiedene Architekturen und Pattern

7a) Was ist ein Software Architektur Pattern

- Ein Software Architektur Pattern ist eine Lösung zu einem wiederkehrenden Problem, das in einem bestimmten Kontext, gut verstanden ist
- Jedes Pattern besteht aus einem Kontext, einem Problem und einer Lösung
- Das Problem könnte eine Herausforderung, eine vorteilhafte Gelegenheit oder die Antwort auf die Anforderungen von Qualitätsattributen sein
- Pattern kodieren Wissen und Erfahrung in Lösungen, die wir wiederverwenden können
- Pattern vereinfachen Design und Architektur durch die Nutzung erprobter Lösungen
- Architekturen pattern sind übergreifender, Designpattern spezieller, detaillierter
- Jedes Pattern hat Ausprägungen, Stärken und Schwächen
- Pattern helfen die Komplexität zu verringern
- Pattern können kombiniert werden
- Pattern können übermäßig benutzt werden
- Architektur Stile und Pattern werden manchmal getrennt aufgelistet und unterschieden (hier nicht)
- Architekten sind mit Pattern vertraut

7b) Verschiedene Architekturen und Pattern

- Architekturen oder Architekturmuster
 - Architectural patterns
 - Hoher Abstraktionsgrad
 - Architektur eines Systems
 - Layers, verteilte Systeme, MVC
- Entwurfsmuster oder Pattern
 - Design patterns (GoF = Gang of Four / Gama, Helm, Johnson, Vlissides)
 - Niedrigerer Anstraktionsgrad
 - Problemstellung innerhalb eines Subsystems / Konstruktionsprinzipien im Kleinen (Mikroarchitekturen)
 - Beeinflusst i.a. nicht die Architektur eines Systems
- Idiome
 - Muster in einer bestimmten Programmiersprache

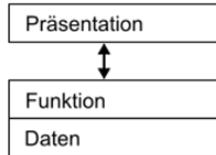
7b) Verschiedene Architekturen und Pattern

- Schema zur Darstellung
 - Name
 - Problembeschreibung
 - Lösungsbeschreibung
 - Teilnehmer (Rollenbeschreibungen)
 - Diagramme (Klassen, Beziehungen)
 - Verhalten (Sequenzdiagramm)
 - Beispiel (Code)
 - Bewertung
 - Vorteile
 - Nachteile
 - Einsatzgebiete
 - Ähnliche Muster (Gemeinsamkeiten und Unterschiede)

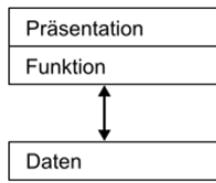
7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Schichtenarchitektur
 - Horizontale Schichten

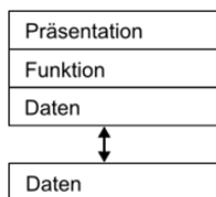
- unterschiedliche Aufteilung



THIN-CLIENT, AKTIVER SERVER
 + zentrale Administration & Wartung
 + Kostensparnis
 + Sicherheit (nur 1 Server muss geschützt werden)
 + Flexibilität
 - hohe Belastung für Server

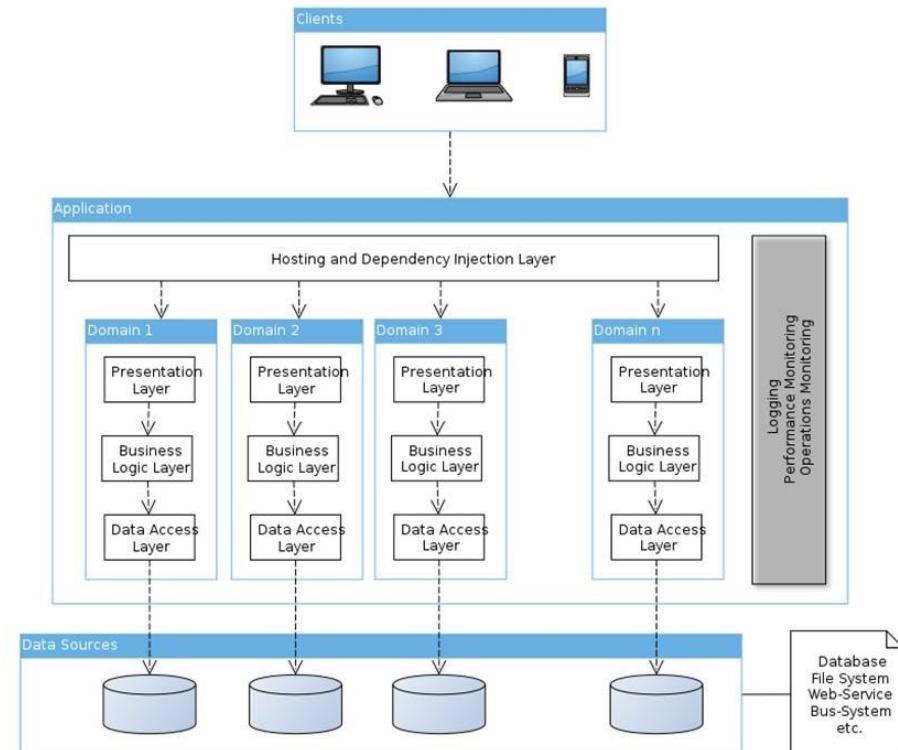


DATEN-SERVER
 → Server liefert nur benötigte Daten
 + jeder Client kann selber rechnen → höhere Performance
 + zentrale Datenhaltung bleibt bestehen → Sicherheit
 - hoher Installationsaufwand, da jeder Client alle Applikationen braucht



FAT-CLIENT
 + Entlastung Server & Datenleitung durch Plug-Ins
 + Weiterarbeiten möglich, wenn Kommunikation zum Server gestört
 - lange Dauer, bis alle Clients Updates haben
 - hohe Wartungskosten

Moderne Software Architektur



7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Schichtenarchitektur
 - Es gibt offenen und geschlossene Schichten
 - Geschlossene Layerarchitekturen verhindern das Umgehen (Bypass) von Schichten
 - Tiers und Layers
 - Layers sind die logische, Tiers die physikalische Aufteilung in Schichten
 - Vorteile
 - Separation of concern wird direkt unterstützt oder lässt sich einfach implementieren
 - Abhängigkeiten zwischen Layern lassen sich reduzieren
 - Wiederverwendbarkeit wird unterstützt
 - Skalierbarkeit wird unterstützt (jeder Layer auf eigener Hardware)
 - Security kann höher sein

7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Schichtenarchitektur
 - Nachteile
 - Änderungen (ein neues Feld) an einem Layer verursachen oft Änderungen in anderen Layern
 - Die Interfaces zwischen den Layern verursachen mehr Code und damit mehr Komplexität
 - Oft landet Code im falschen Layer (Business Logik im Presentation Layer oder Datenzugriffslogik im Business Layer)
 - Stark verteilte Anwendungen mit hoher Anforderung an Performance und Verfügbarkeit bringen solche Architekturen an ihre Grenzen
 - » Bottlenecks
 - » Laufzeiten durch die Layer
 - » Überlast von Ressourcen

7b) Verschiedene Architekturen und Pattern

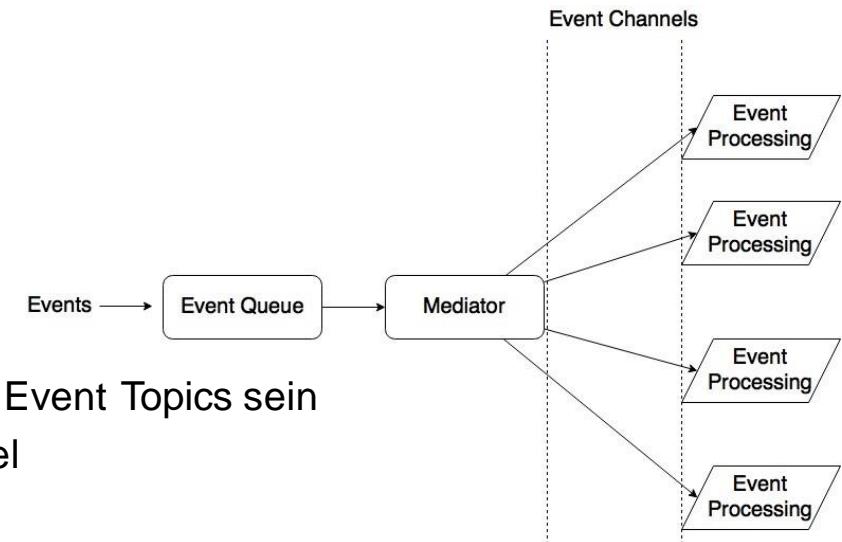
- Architekturmuster
 - Event-driven Architektur
 - Ist eine verteiltes, asynchrones Architektur Pattern, das Anwendungen und Komponenten durch das Erzeugen und Verwalten von Events verbindet
 - Ist eine lose gekoppelte Architektur
 - Der Erzeuger eines Events hat keine Ahnung vom Empfänger/Konsumenten des Events
 - Kann eine sehr komplexe Architektur sein (mit Mediatoren und Brokern)
 - Kann sehr mächtig sein (wenn richtig geplant und implementiert)

7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Event-driven Architektur
 - Die beiden wichtigsten Event Topologien benutzen
 - Event channels (= Ströme von Event Nachrichten) zwischen Event Prozessoren (Verarbeitern)
 - Event channels sind typischerweise als Message Queues implementiert, die
 - » Point-to-point Verbindungen (genau ein Empfänger)
 - » Message Topics (mehrere Empfänger)
 - » Publish-subscribe Pattern (Broadcast an Interessierte, Empfänger melden Interesse an) implementieren

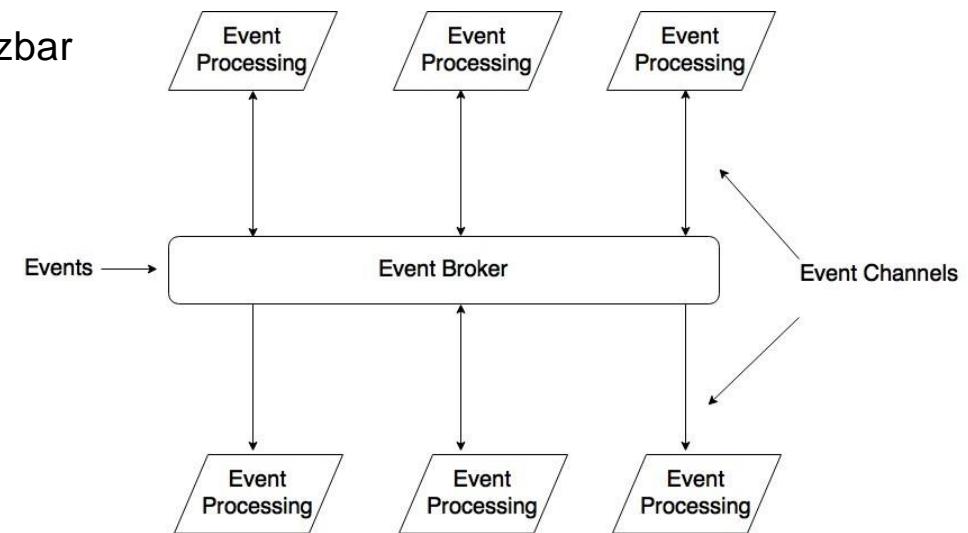
7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Event-driven Architektur
 - Die Mediator Topologie
 - Alle Event gehen durch einen sog. Mediator
 - Der Mediator orchestriert die Events, generiert einen oder mehrere Events zur folgenden Eventverarbeitung
 - Event Channels können Event Queues oder Event Topics sein
 - Event Prozessoren warten an einem Channel auf einen Event zur Verarbeitung
 - Implementierungen dieser Art findet man oft in Business Prozess oder Workflow Umgebungen



7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Event-driven Architektur
 - Die Broker Topologie
 - Ohne Event Queue
 - Event Prozessoren holen sich Events vom Broker, verarbeiten sie und erzeugen nach Bedarf neue Events, die an den Broker übergeben werden
 - Bei einfachen Event Flüssen besser einsetzbar



7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Event-driven Architektur
 - Event Verarbeitung
 - Simple Event processing
 - » Events werden zur Verarbeitung sofort weitergeleitet
 - » Kann für Real-Time Workflows verwendet werden
 - Event Stream processing
 - » Analyisiert Event Streams und steuert danach die weitere Verarbeitung
 - » Beispiel : Kauf von Aktien, die unter eine definierte Schwelle gefallen sind
 - Complex Event processing
 - » Analysiert Events Streams auf wiederkehrende Muster
 - » Läuft i.a. über längere Zeiträume
 - » Beispiel : Kreditlimit

7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Event-driven Architektur
 - Event Verarbeitung
 - Simple Event processing
 - » Events werden zur Verarbeitung sofort weitergeleitet
 - » Kann für Real-Time Workflows verwendet werden
 - Event Stream processing
 - » Analyisiert Event Streams und steuert danach die weitere Verarbeitung
 - » Beispiel : Kauf von Aktien, die unter eine definierte Schwelle gefallen sind
 - Complex Event processing
 - » Analysiert Events Streams auf wiederkehrende Muster
 - » Läuft i.a. über längere Zeiträume
 - » Beispiel : Kreditlimit

7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Event-driven Architektur
 - Typen von Event-driven Funktionalität
 - Event notification
 - » Eine Nachricht (Event) wird generiert/gesendet, wenn ein Ereignis auftritt
 - » Mediator und Broker verarbeiten/steuern i.a. genau solche Events
 - » Es existiert eine lose Kopplung zwischen Sender und Empfänger/Konsument
 - » Event Prozessoren haben eine genau definierte Aufgabe
 - » Die lose Kopplung macht die Event Verfolgung (Fehlerverfolgung) schwieriger

7b) Verschiedene Architekturen und Pattern

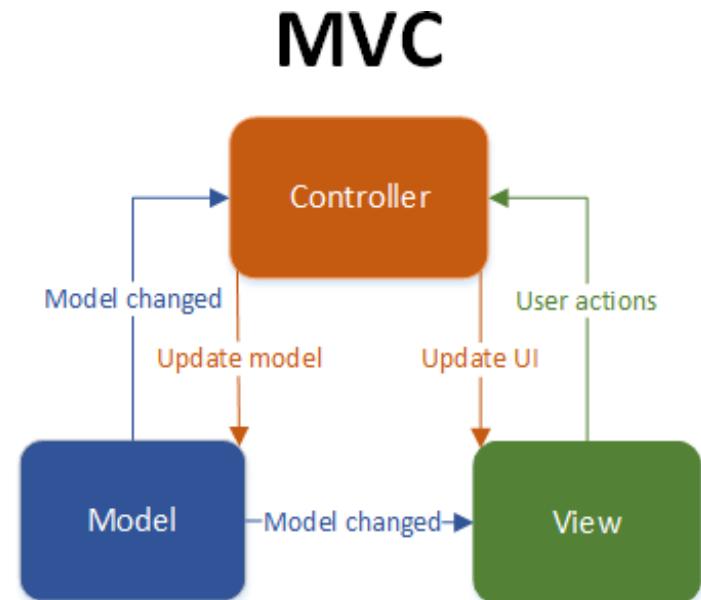
- Architekturmuster
 - Event-driven Architektur
 - Typen von Event-driven Funktionalität
 - Event-carried state transfer
 - » Eine Variante von Event notification
 - » Nach dem Empfang eines Events könnte der Empfänger weitere Informationen benötigen/anfordern, um den Event verarbeiten zu können
 - » Statusinformationen erhöhen die Nutzbarkeit von Events und werden deshalb mitgeliefert
 - » es werden mehr Daten übermittelt und Datenkonsistenz gerät in Gefahr

7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Event-driven Architektur
 - Typen von Event-driven Funktionalität
 - Event sourcing
 - » Events werden in einem Event Store persistiert, um Änderungen (z.B. am Status) nachvollziehen zu können
 - » Events können aus dem Store “gespielt” werden
 - » Roll back Szenarien sind möglich
 - » ähnlich einem Transaction Log bei Datenbanken
 - » bringt zusätzliche Komplexität ins Gesamtsystem
 - » wenn sich Event Processing ändert, müssen evtl. ältere Events angepasst werden
 - » es macht Sinn, externe Systeme über Events anzubinden (um sie speichern zu können)

7b) Verschiedene Architekturen und Pattern

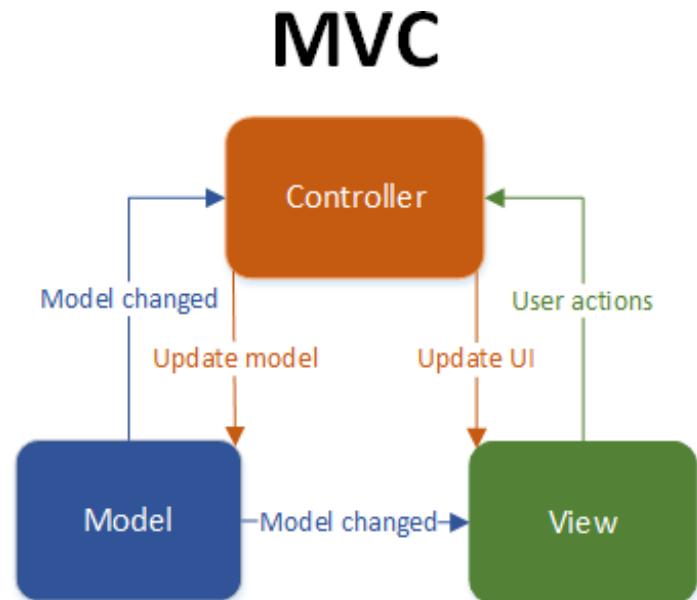
- Architekturmuster
 - Das Model-View-Controller Pattern (MVC)
 - weit verbreitet (IOS Apps)
 - trennt Verantwortlichkeiten der Schnittstellen zum Nutzer
 - Model
 - ist verantwortlich für Daten und Status
 - auch für die Verarbeitung von Daten von und zu einer Datenbank
 - unabhängig von Controller und View
 - empfängt Anforderungen vom Controller
 - hat eine eher passive Rolle



7b) Verschiedene Architekturen und Pattern

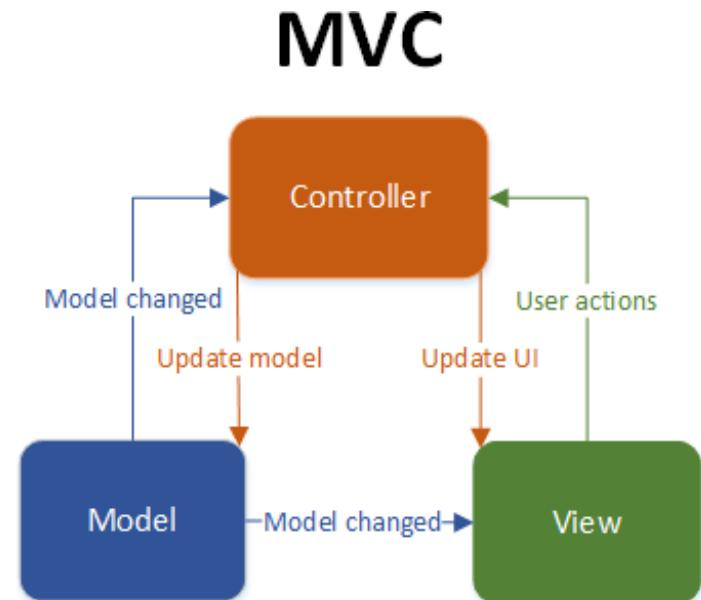
- Architekturmuster
 - Das Model-View-Controller Pattern (MVC)

- View
 - ist verantwortlich für die Anzeige der Anwendung
 - ist für den Benutzer sichtbar
 - zeigt Daten an und nimmt Eingaben vom Benutzer auf
 - Gibt Eingaben an den Controller weiter



7b) Verschiedene Architekturen und Pattern

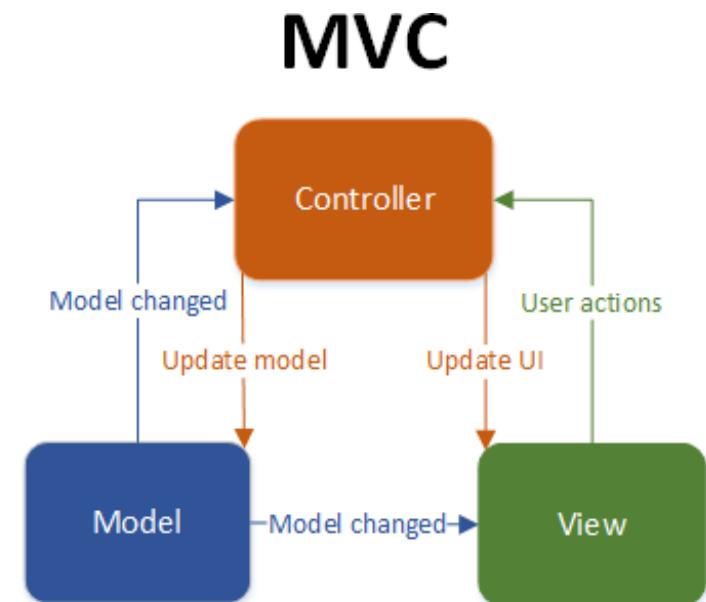
- Architekturmuster
 - Das Model-View-Controller Pattern (MVC)
 - Controller
 - ist verantwortlich für die Logik der Anwendung
 - ist Mittler zwischen Daten (Model) und Anzeige (View)
 - gibt Anfragen/Updates an das Model und empfängt von dort Daten
 - wählt geeignete Views zur Anzeige aus



7b) Verschiedene Architekturen und Pattern

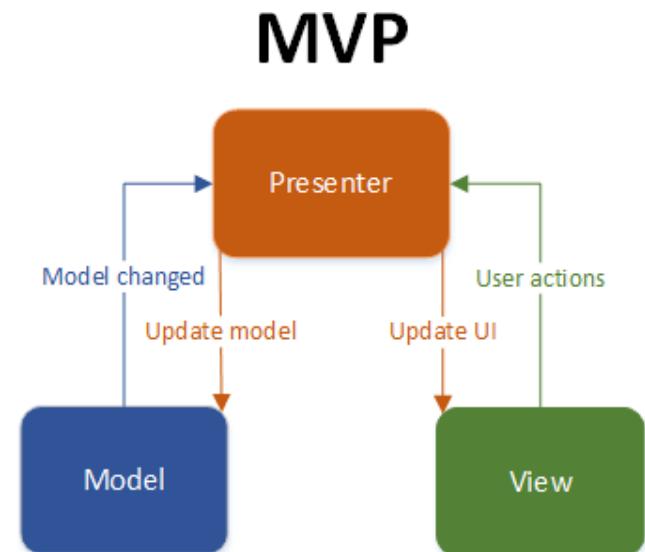
- Architekturmuster
 - Das Model-View-Controller Pattern (MVC)

- Vorteile
 - direkte Unterstützung von “Separation of concerns”
 - einfacheres Testen der Einzelteile
 - keine ganz lose Kopplung erreichbar (neues Feld)
 - dient hauptsächlich zum Abtrennen von Benutzeroberflächen (dynam. Ändern zur Laufzeit)



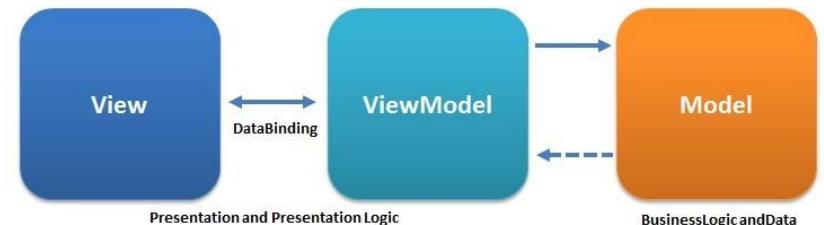
7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Das Model-View-Presenter Pattern (MVP)
 - eine Variante des MVC Pattern
 - der Presenter ersetzt den Controller
 - jede View hat ein spezielles Interface (view interface), der Presenter ist an die View interfaces gekoppelt und steuert diese
 - MVP hat eine stärkere lose Kopplung als MVC (speziell zwischen View und Model)
 - im Gegensatz zum MVC verwaltet ein Presenter eine View
 - es kann mehrere Presenter geben



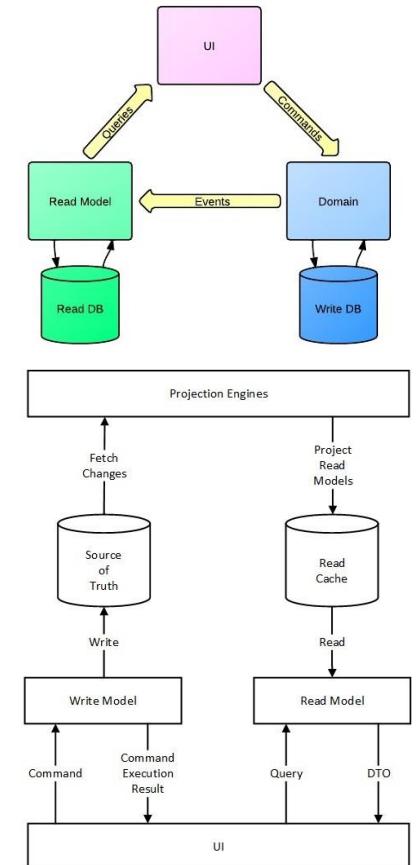
7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Das Model-View-ViewModel Pattern
 - eine Mischung von MVC und MVP
 - Windows Presentation Foundation nutzt dieses Modell
 - Model besitzt die Logik und die Anbindung der Daten
 - View ist das für den Anwender sichtbare Interface
 - ViewModel spielt die Rolle des Controllers oder Presenters
 - ViewModel besitzt die Logik der Navigation
 - klares Schichtenmodell, gut geeignet für Rich Clients



7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Das Command Query Responsibility Segregation Pattern (CQRS)
 - Schreibkanal getrennt vom Lesekanal
 - unterschiedliche Workloads zwischen Lese- und Schreiboperationen können ideal abgebildet werden
 - Sperren von Datensätzen wird vermieden
 - Lesedaten können für die Anzeige optimiert abgelegt sein (Performance)
 - Optimierte und unterschiedliche Datenpools können benutzt werden
 - Updates der Datenpools über Events (z.B. Event sourcing)
 - Manche NoSQL Datenbanken unterstützen CQRS out-of-the-box
 - Das Trennen von Schreib- und Lesemodell kann die Security erhöhen
 - CQRS erhöht die Komplexität des Gesamtsystems
 - Lesepool „hinkt“ hinterher



7b) Verschiedene Architekturen und Pattern

- Architekturmuster
 - Service Oriented Architecture (SOA)
 - lose gekoppelte voneinander unabhängige Services, die zusammen die Anwendung bilden
 - teilt ein Software System in kleinere Einheiten auf, die jeweils eine andere Aufgabe erfüllen
 - kann auch Business Logik in kleinere Einheiten aufteilen
 - ein Service kann aus mehreren anderen Services bestehen oder sie nutzen
 - verbindet die Logik mit der Technik
 - unterstützt die Nutzung verschiedener Serviceanbieter
 - unterstützt stark agile Entwicklungsmethoden
 - benötigt gute Architekten
 - bedingt gute Architektur von Services und eine vollständige Beschreibung
 - besitzt oft sog. Service Register als Verzeichnis der vorhandenen Services

8) Moderne Architekturen

- Moderne Anwendungen in der Cloud haben ganz andere Erwartungen und Anforderungen an Architekturen im Vergleich zu früher
- Um diesem zu genügen, wurden neue Software Architektur Patterns entwickelt
- Wir betrachten
 - a) Monolithic Architecture
 - b) Microservice Architecture
 - c) Serverless Architecture
 - d) AWS Cloud-native Architecturstile
 - e) Microsoft Cloud Architekturstile

a) Monolithic Architecture

- einzelne sich selbst beinhaltende Anwendung
- sehr weit verbreitet
- hohe interne Koppelung der Module
- keine Trennung von Logik und Anzeige oder Logik und Datenhaltung
- Vorteile
 - bessere Performance (speziell bei kleinen Anwendungen)
 - einfacher zu verteilen und installieren
 - einfacher zu testen und debuggen
- Nachteile
 - skaliert schlechter (speziell in verteilten Umgebungen)
 - niedrige Wartbarkeit
 - wenig/schlecht erweiterbar
 - unterstützt agile Entwicklungsmethoden und Teamstrukturen schlecht



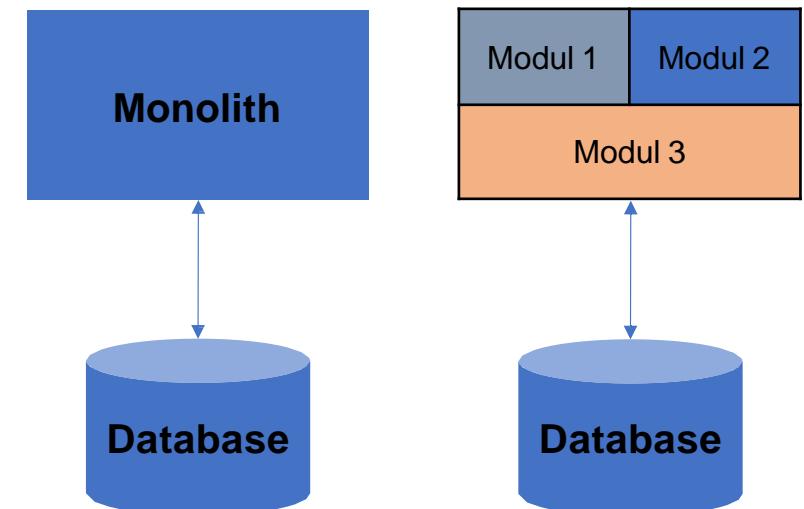
a) Monolithic Architecture

- Ein Monolith ist ein System innerhalb einer Bereitstellungseinheit (Deployment unit)
 - Wenn alle Funktionen eines Systems zusammen bereitgestellt werden müssen, handelt es sich um einen Monolithen
- Es gibt mindestens drei Arten von monolithischen Systemen
 - der Monolith mit einem Prozess
 - der verteilte Monolith
 - Black-Box-Systeme von Drittanbietern



a) Monolithic Architecture

- Arten von Monolithen
 - Der Einzelprozess-Monolith
 - Ein System, in dem der gesamte Code als Einzelprozess mit einer bis mehreren Instanzen bereitgestellt wird. In der Regel werden Daten aus einer Datenbank gelesen oder in einer Datenbank gespeichert
 - Der modulare Monolith ist eine Variation des Einzelprozess-Monolithen
 - Einzelprozess Monolithen stellen die überwiegende Mehrheit der monolithischen Systeme dar
 - Wenn der Begriff "Monolith" verwendet wird, bezieht er sich auf diese Art von Monolithen



a) Monolithic Architecture

- Arten von Monolithen
 - Der verteilte Monolith
 - Ein verteilter Monolith ist ein System, das aus mehreren Diensten besteht. Aus irgendeinem Grund muss das gesamte System zusammen bereitgestellt werden
 - Verteilte Monolithen haben alle Nachteile eines verteilten Systems und die Nachteile eines Einzelprozessmonolithen
 - Verteilte Monolithen typischerweise entstehen, wo nicht genug Fokus auf Konzepte wie Information Hiding und die Kohäsion von Geschäftsfunktionen gelegt wurde
 - Verteilte Monolithen haben stark gekoppelte Architekturen, in denen Änderungen über Dienstgrenzen hinweg auftreten und scheinbar einfache Änderungen, deren Umfang lokal zu sein scheint, beeinflussen stark andere Teile des Systems

a) Monolithic Architecture

- Arten von Monolithen
 - Black-Box-Systeme von Drittanbietern
 - Einige Softwaresysteme von Drittanbietern können auch als Monolithen betrachtet werden
 - Dazu gehören beispielsweise Lohn- und Gehaltsabrechnungssysteme, CRM-Systeme und HR-Systeme
 - Der gemeinsame Faktor ist, dass es sich um Software handelt, die von anderen Personen entwickelt wurde und man nicht die Fähigkeit oder die Erlaubnis hat, den Code zu ändern
 - Es kann sich um Standardsoftware handeln, die in der eigenen Infrastruktur bereitgestellt wird, oder um ein verwendetes Software-as-a-Service-Produkt (SaaS)

a) Monolithic Architecture

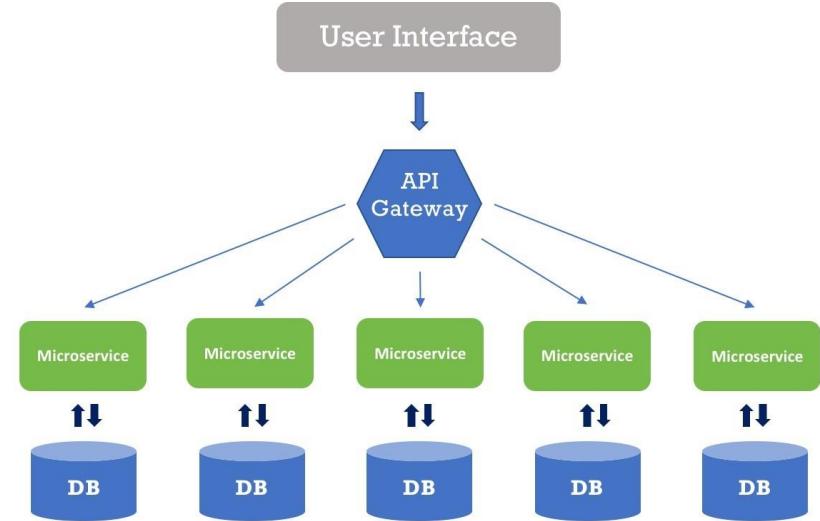
- Herausforderungen
 - Der Monolith ist häufig anfälliger für die Gefahren der Kopplung - insbesondere bei der Implementierung und der Bereitstellung
 - Verschiedene Entwickler möchten denselben Code ändern
 - Verschiedene Teams möchten die Funktionalität zu unterschiedlichen Zeiten in Produktion bringen (oder Bereitstellungen verschieben)
 - Verwirrung darüber, wem was gehört und wer Entscheidungen trifft

a) Monolithic Architecture

- Vorteile
 - Der Einzelprozessmonolith hat eine Reihe von Vorteilen
 - einfache Bereitstellungstopologie
 - einfache Entwickler-Workflows
 - Einfachere Überwachung, Fehlerbehebung und End-to-End-Tests
 - Monolithe können auch die Wiederverwendung von Code innerhalb des Monolithen selbst vereinfachen
 - Eine monolithische Architektur ist möglicherweise nicht unter allen Umständen die richtige Wahl, genauso wenig wie Microservices - aber es ist trotzdem eine Wahl

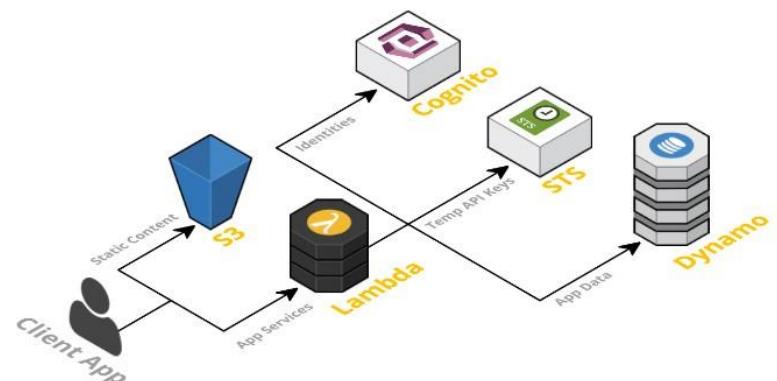
b) Microservice Architecture

- kleine, autonome, unabhängig versionierte Services
- genau definiertes Interface zur Kommunikation
- eingehende Requests gehen über ein sog. API Gateway
- eine Variante von SOA (SOA done right)
- Charakteristiken
 - kleine, fokussierte Services
 - genau spezifizierte Interfaces
 - autonom und unabhängig ausrollbar
 - unabhängiger Datenspeicher
 - einfache Kommunikationsprotokolle
(JSON, Advanced Message Queueing Protocol (AMQP))
 - sehr gute Isolation
 - Bulkhead fähig
 - Recovery fähig
 - Polyglot



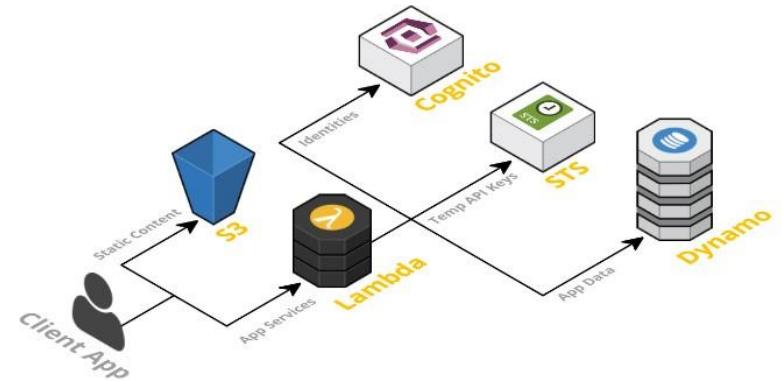
c) Serverless Architecture

- Services ohne Berücksichtigung von Hardware (Server)
- Code wird auf einer Plattform dann ausgeführt, wenn benötigt
- ist oft in einem Container implementiert, der zur Laufzeit hochgefahren und wieder beendet wird
- besteht oft aus statischen (Speicher) und dynamischen (Funktionen) Anteilen
- benutzt FaaS (function as a service) oder BaaS (Backend as a service) Modelle
- Jede Funktion folgt dem Single Responsibility Principle und dient genau einem definierten Zweck
- Funktionen sollten idempotent sein
- Funktionen können synchron oder asynchron ablaufen
- ein wichtiger Teil einer Serverless Architektur ist das API Gateway
- Beispiele Amazon Lambda, Azure Services, Google Cloud Functions



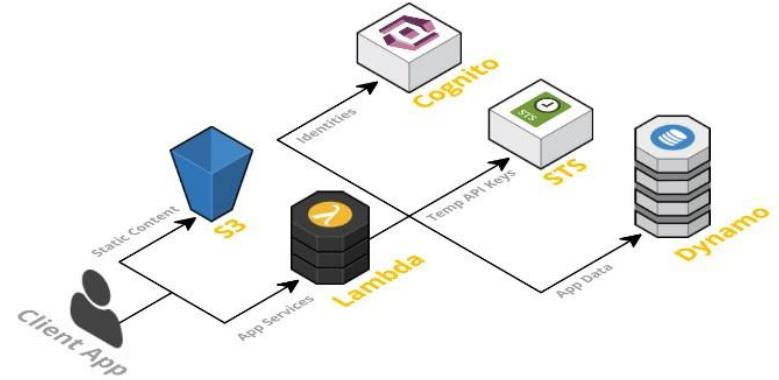
c) Serverless Architecture

- Vorteile
 - Kosten
 - Code läuft nur, wenn benötigt
 - abgerechnet werden benutzte Ressourcen (pay per usage)
 - keine Hardwarekosten (Bestellung, Installation, Betrieb, Austausch)
 - keine Betriebsmannschaften
 - Skalierbar und flexible
 - es steht immer genügend Kapazität zur Verfügung
 - keine Verschwendungen von Kapazitäten, keine Freistände
 - Fokussiert
 - auf die Lösung des Problems
 - keine Infrastruktur notwendig



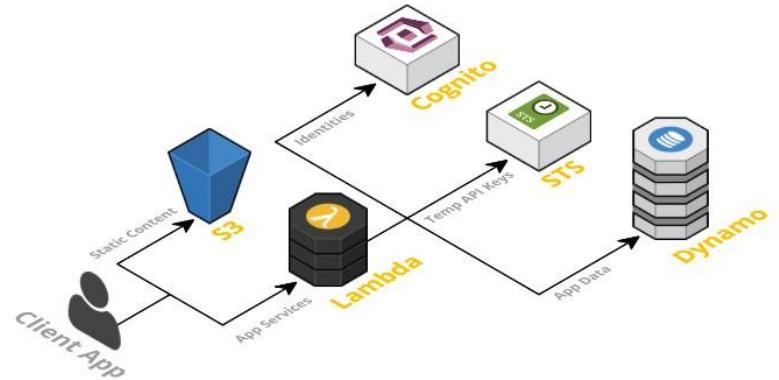
c) Serverless Architecture

- Vorteile
 - Polyglot
 - best-of-breed Programmiersprachen einsetzbar
 - Cloud Plattformen beliebig mischbar
- Nachteile
 - aufwändigeres Monitoring und Debugging
 - verteilt, Event getrieben, stateless
 - wenige verfügbare Pattern
 - verfügbare Pattern noch nicht lange im Einsatz
 - geringerer Reifegrad
 - Maintenance (Performance & Security)
 - Vendor lock-in
 - Komplexität
 - Optimierungsmöglichkeiten eingeschränkt



c) Serverless Architecture

- Hybride Ansätze möglich
 - teilweise on-premise, teilweise in der Cloud
- Aufrufmethoden von Funktionen
 - Synchron
 - asynchron
 - Message stream
 - Batch job



d) AWS Cloud-native Architecturstile

- Was bedeutet Cloud Native ?
 - Cloud Native (cloud-native) beinhaltet folgende Konzepte
 - „Wegwerf“-Infrastruktur als Triebfeder
 - Einsatz von funktional eingeschränkten, isolierten Komponenten
 - Globale Skalierung
 - Eine Architektur, die sich dauernd verändert
 - Wirksamer Einsatz von wertbringenden Cloud Services
 - Vielsprachigkeit (Programmiersprachen)
 - Unabhängige, voll selbstständige Teamstrukturen
 - Kulturwechsel in Organisationen

d) AWS Cloud-native Architecturstile

- „Wegwerf“-Infrastruktur als Triebfeder
 - Erzeugen und Vernichten von Ressourcen zu jeder Zeit
 - Automatisierung
 - Geschwindigkeit, Schnelligkeit
 - Infrastructure as Code
 - Sicherheit
 - Optimierung von Teamgröße und Effizienz

d) AWS Cloud-native Architecturstile

- Die Cloud ist die Datenbank
- Reactive Architectur (www.reactivemanifesto.org)
 - Responsiv, Resilient, Elastic, Message driven
- Die Datenbank nach außen stülpen (inside out)
 - Weg vom Monolithen zur Replikation
- Bulkheads
- Event streaming
- Vielschichtige Persistenz
- Cloud Native Datenbanken
- Cloud Native Patterns
- Bounded Isolated components



Adobe Acrobat
Document

d) AWS Cloud-native Architecturstile

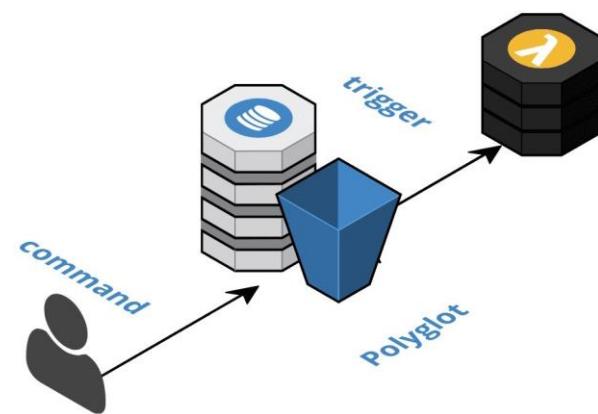
- Cloud Native Patterns
 - Foundation Patterns
 - Boundary patterns
 - Control patterns

d) AWS Cloud-native Architecturstile

- Cloud Native Foundation Patterns
 - Cloud Native Database per Component
 - Event Streaming
 - Event Sourcing
 - Data Lake
 - Stream Circuit Breaker
 - Trilateral API

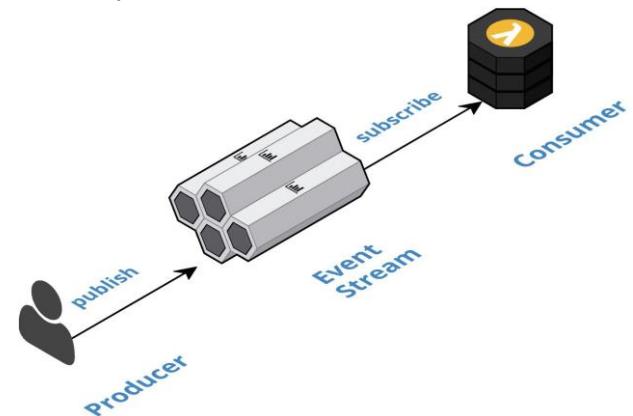
d) AWS Cloud-native Architecturstile

- Cloud Native Database per Component
 - Nutzung von einer oder mehreren vollständig verwalteten Cloud-native Datenbanken, die nicht komponentenübergreifend gemeinsam genutzt werden
 - Ereignisse lösen eine komponenteninterne Verarbeitungslogik aus
 - Nutzung der vollständig verwalteten Cloud-native Datenbankdienste des Cloud-Anbieters
 - Mehrere Datenbanktypen können je nach Bedarf innerhalb einer Komponente verwendet werden, um Workloads besser abfangen zu können
 - Wählen Sie Datenbanktypen wie Blob Store, Document Store o.ä., je nach Anforderung
 - Jede Datenbank ist einer bestimmten Komponente zugeordnet und wird nicht von mehreren Komponenten gemeinsam genutzt



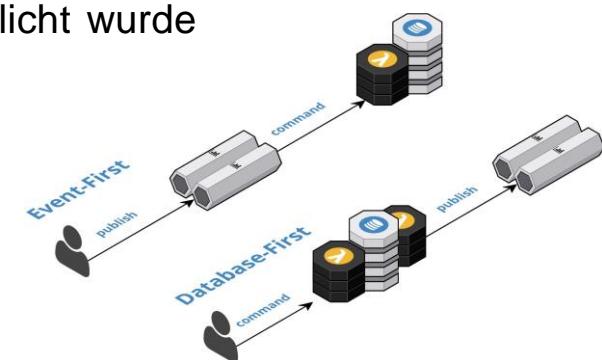
d) AWS Cloud-native Architecturstile

- Event streaming
 - Nutzung eines vollständig verwalteten Streaming-Service, um die gesamte Kommunikation zwischen Komponenten asynchron zu implementieren, wobei Upstream-Komponenten die Verarbeitung an Downstream-Komponenten weiterreichen, indem sie Domänenereignisse veröffentlichen, die von Downstream-Komponenten konsumiert werden
 - Konsumenten/Verbraucher abonnieren je nach Bedarf einen oder mehrere Streams
 - Um eine einheitliche Ereignisbehandlung sicherzustellen, definiert man i.a. ein Standardereignisformat für alle Ereignisse, um eine einheitliche Ereignisbehandlung sicherzustellen (alle kenne das gemeinsam definierte Format)



d) AWS Cloud-native Architecturstile

- Event sourcing
 - Kommunikation und Statusänderung von Domänenentitäten geschieht als eine Reihe von atomar erzeugten unveränderlichen Domänenereignissen unter Verwendung von Event-First- oder Database-First-Techniken, um die asynchrone Kommunikation zwischen Komponenten sicher zu stellen und die Verarbeitungslogik zu vereinfachen
 - Die Richtlinie, nach der ein Befehl nur eine atomare Schreiboperation gegen eine einzelne Ressource oder einen Stream oder eine Cloud-native Datenbank ausführen darf, muss strikt eingehalten werden und nur der asynchronen Mechanismus der Ressource kann Befehle verketten
 - Event-First: Daten von einem Domänenereignis, das in einem Stream veröffentlicht wurde
 - Database-First: Daten, die in eine Cloud-native Datenbank geschrieben wurden und ein Domänenereignis auslösen, das in einem Stream veröffentlicht wurde



d) AWS Cloud-native Architecturstile

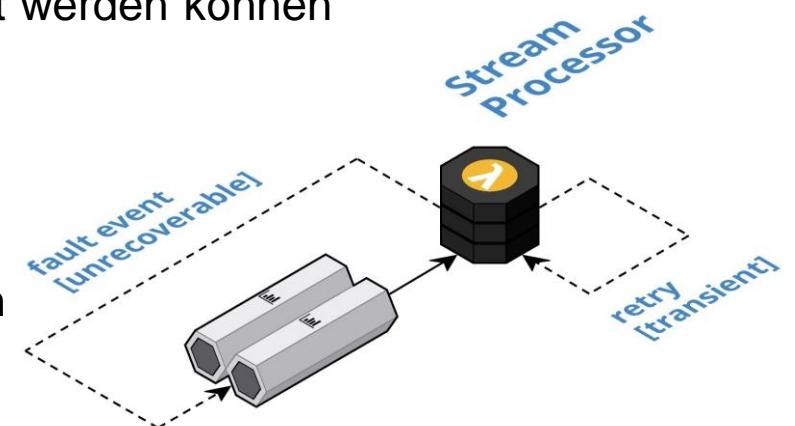
- Data Lake
 - Sicherer Sammeln, Speichern und Indizieren aller Ereignisse (im Rohformat), um Auditing, Wiederabspielen und Analysen zu unterstützen
 - Einer oder mehrere Konsumenten konsumieren alle Ereignisse aus allen Streams und speichern die unveränderten Ereignisse im Rohformat in einem dauerhaften Blob-Speicher
 - Optionale Konsumenten speichern Ereignisse in einer Suchmaschine wie Elasticsearch für Indizierungen und Zeitreihenanalysen



d) AWS Cloud-native Architecturstile

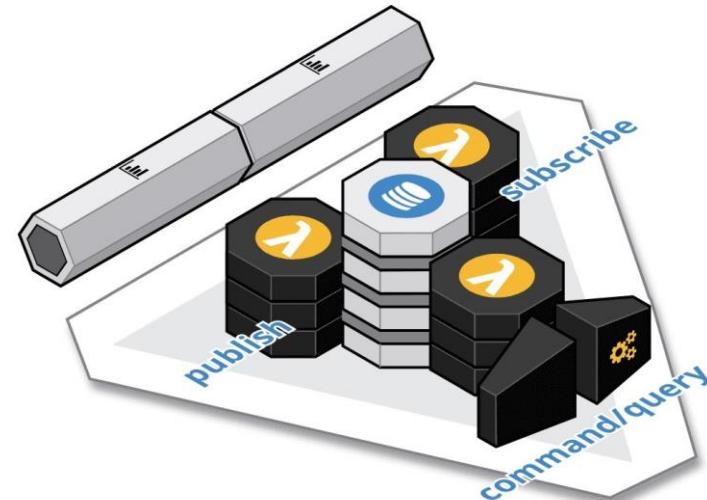
- Stream Circuit Breaker

- Kontrolle von Ereignisflüssen in Stream-Prozessoren, indem die Behandlung nicht behebbarer Fehler an Fehlerereignisse weiter gereicht werden, damit Fehler den Durchsatz nicht unangemessen stören
- Nicht behebbare Fehler werden zur Behandlung an eine andere Komponente weiter gegeben, damit sie die Verarbeitung der fehlerfreien Ereignisse nicht blockieren
- Diese Fehler werden als fehlerhafte Ereignisse zusammen mit den betroffenen Ereignissen und den Stream-Prozessor-Informationen abgelegt (DataLake), damit die Ereignisse bei Bedarf erneut abgespielt/übermittelt werden können
- Fehlerereignisse werden überwacht und wenn notwendig Supportteams benachrichtigt
- Dienstprogramme senden die betroffenen Ereignisse vom DataLake erneut an die Komponenten, die das Fehlerereignis ausgegeben haben



d) AWS Cloud-native Architecturstile

- Trilateral API
 - Jede Komponente besitzt mehrere Schnittstellen
 - Eine synchrone API zum Verarbeiten von Befehlen und Abfragen
 - Eine asynchrone API zum Veröffentlichen von Ereignissen, bei Statusänderungen von Komponenten
 - Eine asynchrone API zum Konsumieren der von anderen Komponenten ausgegebenen Ereignisse



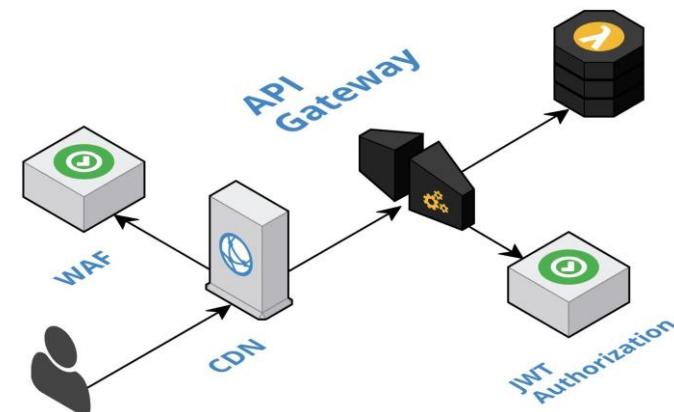
d) AWS Cloud-native Architecturstile

- Cloud Native Boundary Patterns
 - API Gateway
 - Command Query Responsibility Segregation (CQRS)
 - Offline-First Database
 - Backend For Frontend
 - External Service Gateway

d) AWS Cloud-native Architecturstile

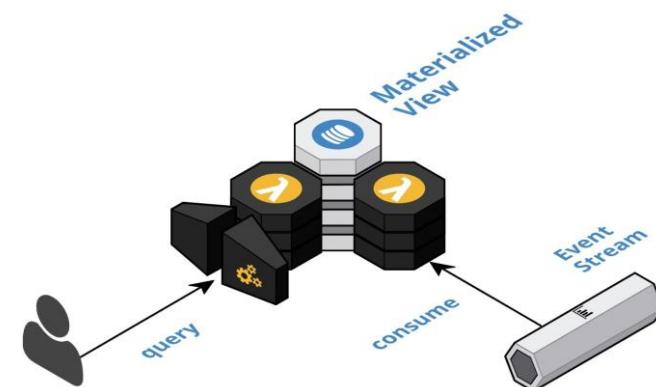
- API Gateway

- Als Barriere an den Grenzen eines Cloud-native Systems dient API-Gateway des Cloud-Anbieters (fully managed)
- Querschnittsthemen wie Sicherheit und Caching werden an den Rand der Cloud verschoben und absorbieren dort Last absorbiert, bevor Sie in das Innere des Systems gelangen
 - CDN = Content Delivery Network
 - WAF = Web Application Firewall
- Die Barriere fungiert als Bulkhead (Schutzschild), um die Innenteile des Systems schon an den Systemgrenzen vor Angriffen zu schützen



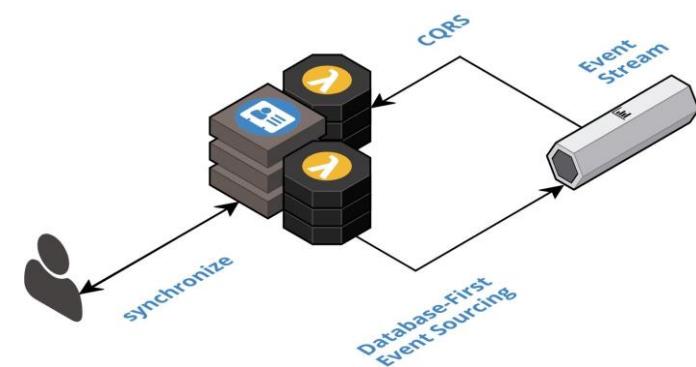
d) AWS Cloud-native Architeurstile

- Command Query Responsibility Segregation (CQRS)
 - Trennen von Statusänderungen/Befehlen von Abfragen
 - Sog. materialisierte Ansichten unterstützen Abfragen von Komponenten
 - Downstream (Command) und Upstream (Query) Komponenten sind getrennt und damit widerstandsfähiger
 - Query-Nichtverfügbarkeiten haben keine Auswirkungen auf Status/Befehl Downstream-Komponenten
 - Abfragen geben weiterhin Ergebnisse aus den materialisierten Ansichten zurück.
 - Eine materialisierte Ansicht fungiert als lokaler Cache, der immer auf dem neuesten Stand ist
 - Es handelt sich um eine lokale Ressource, deren Eigentümer die Komponente ist und daher keine unnötigen Ebenen durchlaufen werden müssen, um die Daten abzurufen



d) AWS Cloud-native Architecturstile

- Offline-First Database
 - Benutzerdaten im lokalen Speicher werden bei Verbindungsauftakt über Events in die Cloud synchronisiert
 - Umgekehrt werden Cloud-seitige Daten aus materialisierten Ansichten abgerufen
 - Der Hauptvorteil dieser Lösung ist eine extrem hohe Verfügbarkeit und Reaktionsfähigkeit für wichtige Benutzerinteraktionen
 - Benutzer empfinden das System als reaktionsschnell und verfügbar, selbst wenn die Konnektivität unterbrochen ist
 - Wenn die Konnektivität wiederhergestellt ist, werden die Daten mit dem Cloud-Speicher und damit geräteübergreifend synchronisiert
 - Die Lösung wird so ausfallsicherer



d) AWS Cloud-native Architecturstile

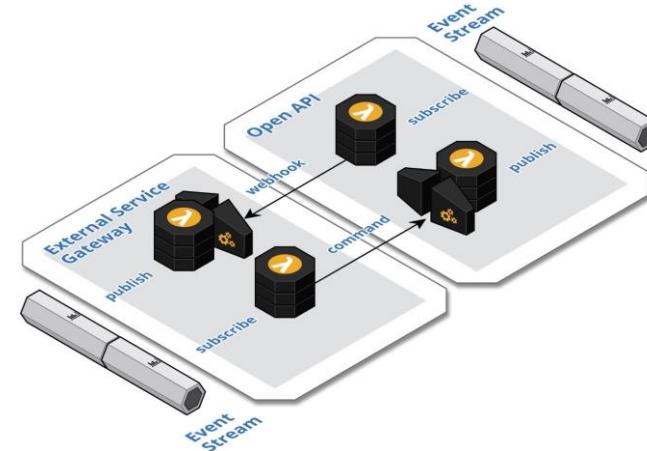
- Backend For Frontend

- Dedizierte und autarke Backend-Komponenten unterstützen die Funktionen benutzerorientierter Frontend-Anwendungen
- Architektur einer Reihe unabhängiger User-Experiences von denen jede eine unterschiedliche Benutzerbasis und eine unterschiedliche und zusammenhängende Reihe von Funktionen unterstützt
- Die Lösung hilft das Conway-Gesetz zu beachten : „Organisationen sind gezwungen, Anwendungsdesigns zu erstellen, die Kopien ihrer Kommunikationsstrukturen sind“
- Die Benutzererfahrungen sind entkoppelt und werden jeweils von einem einzelnen Entwicklungsteam verwaltet (getrennte und unabhängige Entwicklung in verschiedenen Teams)
- Erhöht die Anzahl der Frontends und vermindert gleichzeitig die Komplexität



d) AWS Cloud-native Architecturstile

- External Service Gateway
 - Integration externer Systeme durch Kapselung von eingehender und ausgehender systemübergreifender Kommunikation in einer begrenzten isolierten Komponente
 - Für jedes externe System existiert eine Komponente, die als bidirektionale Brücke für die Übertragung von Ereignissen zwischen den Systemen fungiert
 - Das Aufrufen von Befehlen auf dem externen System folgt der Event-First-Variante des Event-Sourcing-Musters, wobei das externe System die Datenbank darstellt
 - Das Konsumieren von Ereignissen aus dem externen System folgt der Database-First-Variante des Event-Sourcing-Musters, wobei das externe System einen Ereignisstrom darstellt
 - Ereignisse werden im internen Ereignisstrom veröffentlicht, um die Verarbeitung an nachgeschaltete Komponenten weiter leiten zu können

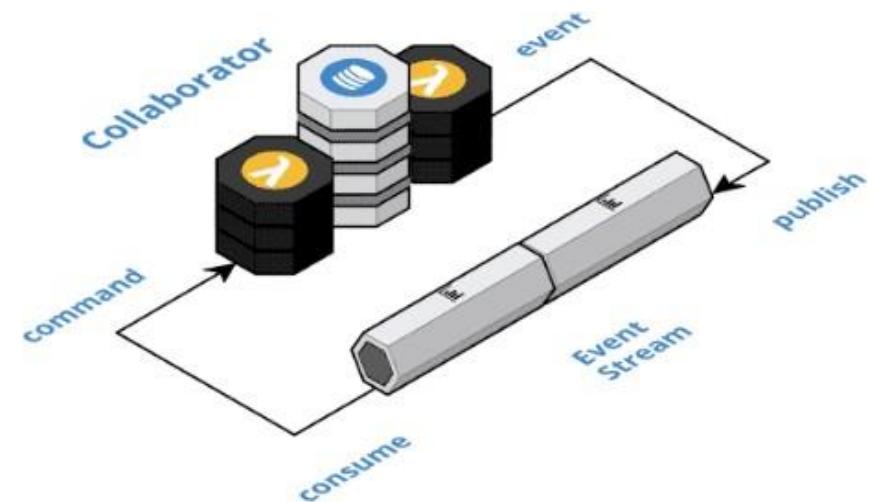


d) AWS Cloud-native Architecturstile

- Cloud Native Control Patterns
 - Event Collaboration
 - Event Orchestration
 - Saga

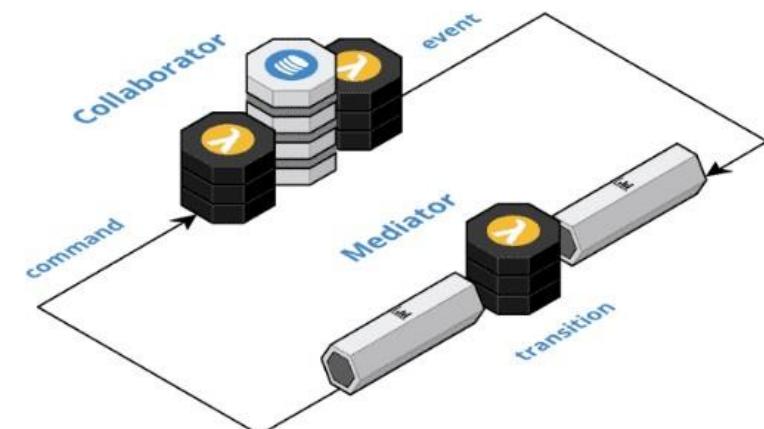
d) AWS Cloud-native Architecturstile

- Event Collaboration
 - Domäneneignisse nutzen Events, um nachgelagerte Befehle auszulösen oder um die Zusammenarbeit zwischen mehreren Komponenten zu gewährleisten
 - Sychrone Kommunikation zwischen Komponenten wird durch asynchrone Kommunikation zwischen Komponenten ersetzt



d) AWS Cloud-native Architecturstile

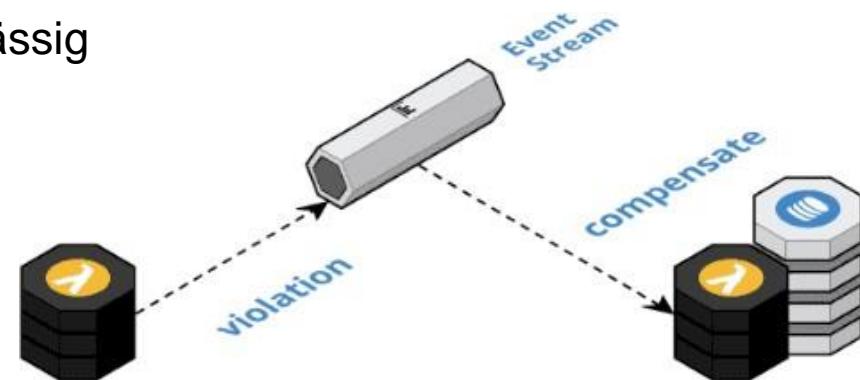
- Event Orchestration
 - Eine Mediatorkomponente koordiniert die Zusammenarbeit zwischen Komponenten, die nicht über Events gekoppelt sind
 - Eine Komponente pro Geschäftsprozess dient als Vermittler zwischen den zusammen arbeitenden Komponenten und koordiniert den Arbeitsfluss zwischen diesen Komponenten
 - Der Mediator erfasst und übersetzt die Ereignisse von Upstream-Komponenten für die Downstream-Komponenten. Die Zuordnungen und Übersetzungen sind im Mediator als Regelwerk zusammengefasst
 - Alle beteiligten Komponenten sind vollständig voneinander und vom Mediator entkoppelt
 - Der Mediator definiert den Geschäftsprozess, wenn der Prozess weiterentwickelt wird, muss nur der Mediator angepasst werden



d) AWS Cloud-native Architecturstile

- **Saga**

- Wenn Geschäftsregeln verletzt werden, können Ausgleichstransaktionen ausgelöst werden, um Änderungen rückgängig zu machen
- Verstöße gegen Geschäftsregeln werden als zusätzliche Ereignisse implementiert
- Die Rückgängigmachung entspricht nicht einem Rollback
- Es ist nicht gewollt die Ausführung eines vorherigen Schritts zu löschen
- Der Schritt wird gezielt abgeschlossen und später durch einen weiteren Schritt, der die Aktion aufhebt, rückgängig gemacht
- In einigen Bereichen, wie z. B. der öffentlichen Buchhaltung, ist dies notwendig
- Undo einer Buchung ist rechtlich nicht zulässig



d) AWS Cloud-native Architecturstile

- Saga (Sage oder Geschichte)
 - Das Wort Saga wurde von Hector Garcia-Molina und Kenneth Salem von der Princeton University 1987 geprägt
 - Langlebige Transaktionen (LLTs) halten die Datenbankressourcen für relativ lange Zeiträume fest und verzögern die Beendigung kürzerer und häufiger verwendeter Transaktionen erheblich
 - Um diese Probleme zu lösen, schlagen wir den Begriff einer Saga vor
 - Ein LLT ist eine Saga, wenn es als eine Folge von Transaktionen geschrieben werden kann, die mit anderen Transaktionen verschachtelt werden können
 - Das Datenbankverwaltungssystem garantiert, dass entweder alle Transaktionen in einer Saga erfolgreich abgeschlossen werden oder kompensiert werden
 - Transaktionen werden ausgeführt, um eine teilweise Ausführung zu ändern
 - Sowohl das Konzept der Saga als auch ihre Umsetzung sind relativ einfach, aber sie haben das Potenzial, die Performance erheblich zu verbessern

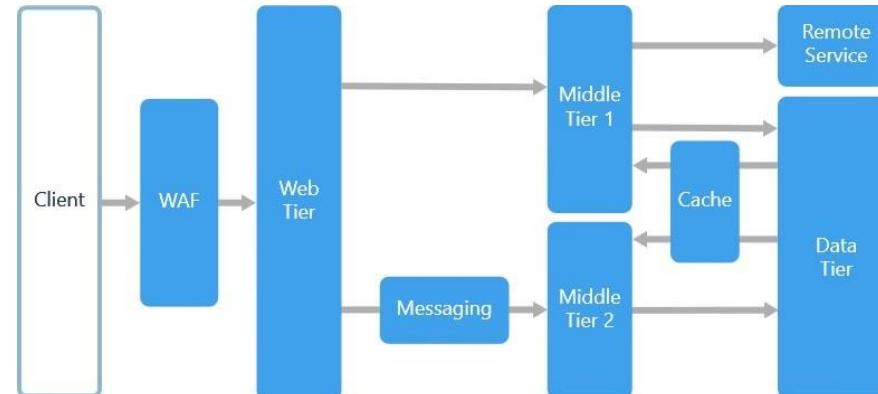
e) Microsoft Cloud Architekturstile

Architekturstil	Abhängigkeitsmanagement	Domänentyp
N-Schichten	Horizontale Ebenen, unterteilt durch Subnetz	Traditionelle Business-Domäne. Die Frequenz der Aktualisierungen ist niedrig
Web-Queue-Worker	Front- und Back-End-Aufgaben, entkoppelt durch asynchrones Messaging	Relativ einfache Domäne mit einigen ressourcenintensiven Aufgaben
Mikroservices	Vertikale (funktional) zerlegte Services, die sich gegenseitig über APIs aufrufen	Komplizierte Domäne. Häufige Updates
Command and Query Responsibility Segregation (CQRS)	Lesen/Schreiben Aufgabentrennung. Schema und Maßstab werden separat optimiert	Kollaborative Domäne, in der viele Benutzer auf dieselben Daten zugreifen
Ereignisgesteuerte Architektur	Produzent/Konsument. Unabhängige Ansicht pro Subsystem	IoT- und Echtzeitsysteme
Big Data	Aufteilen eines großen Datensatzes in kleine Blöcke. Parallele Verarbeitung in lokalen Datensätzen	Datenanalyse im Stapel und in Echtzeit. Prädiktive Analyse mit ML
Big Compute	Datenzuordnung an Tausende von Kernen	Berechnungsintensive Domänen wie Simulationen

e) Microsoft Cloud Architekturstile

- **N-Schichten (N-tier)**

- ist eine traditionelle Architektur für Unternehmensanwendungen. Abhängigkeiten werden verwaltet, indem die Anwendung in Ebenen unterteilt wird, die logische Funktionen wie Präsentation, Geschäftslogik und Datenzugriff ausführen. Eine Ebene kann nur Ebenen aufrufen, die darunter liegen. Diese horizontale Schichtung kann jedoch ein Problem darstellen. Es kann schwierig sein, Änderungen in einem Teil der Anwendung einzufügen, ohne den Rest der Anwendung zu berühren. Das macht häufige Aktualisierungen zu einer Herausforderung und begrenzt die Geschwindigkeit, mit der neue Funktionen hinzugefügt werden können.
- N-Schichten eignet sich hervorragend für die Migration vorhandener Anwendungen, die bereits eine geschichtete Architektur aufweisen. Aus diesem Grund wird N-Schichten am häufigsten in IaaS-Lösungen (Infrastructure as a Service) verwendet oder in Anwendungen, die eine Mischung aus IaaS und verwaltete Services sind.

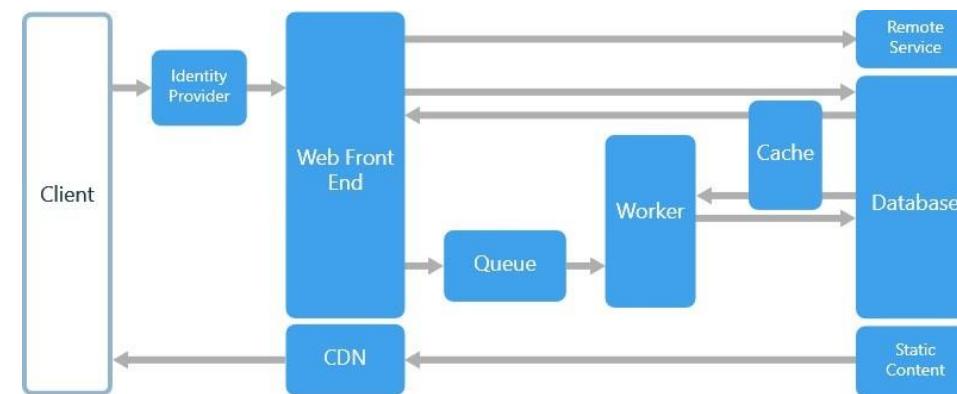


WAF = Web Application Firewall

e) Microsoft Cloud Architekturstile

- Web-Queue-Worker

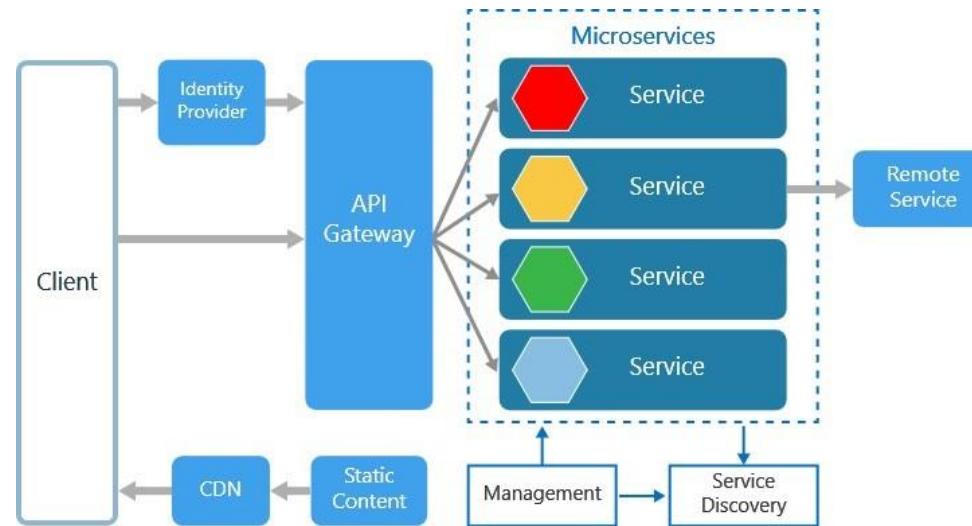
- Erwägen Sie für eine reine PaaS-Lösung eine Web-Queue-Worker-Architektur. In diesem Stil verfügt die Anwendung über ein Web-Front-End, das HTTP-Anforderungen verarbeitet, und einen Back-End-Worker, der CPU-intensive Aufgaben oder lang andauernde Vorgänge ausführt. Das Front-End kommuniziert über eine asynchrone Nachrichtenwarteschlange mit dem Auftragnehmer.
 - Web-Queue-Worker eignet sich für relativ einfache Domänen mit einigen ressourcenintensiven Aufgaben. Wie bei N-Schichten ist die Architektur leicht zu verstehen. Die Verwendung von verwalteten Services vereinfacht die Bereitstellung und den Betrieb. Bei komplexen Domänen kann es jedoch schwierig sein, die Abhängigkeiten zu verwalten.
 - Das Front-End und der Worker können leicht zu großen, monolithischen Komponenten werden, die schwierig zu warten und zu aktualisieren sind.
 - Wie bei N-Schichten kann dies die Häufigkeit von Aktualisierungen reduzieren und die Innovation einschränken.



e) Microsoft Cloud Architekturstile

- Mikro-Services

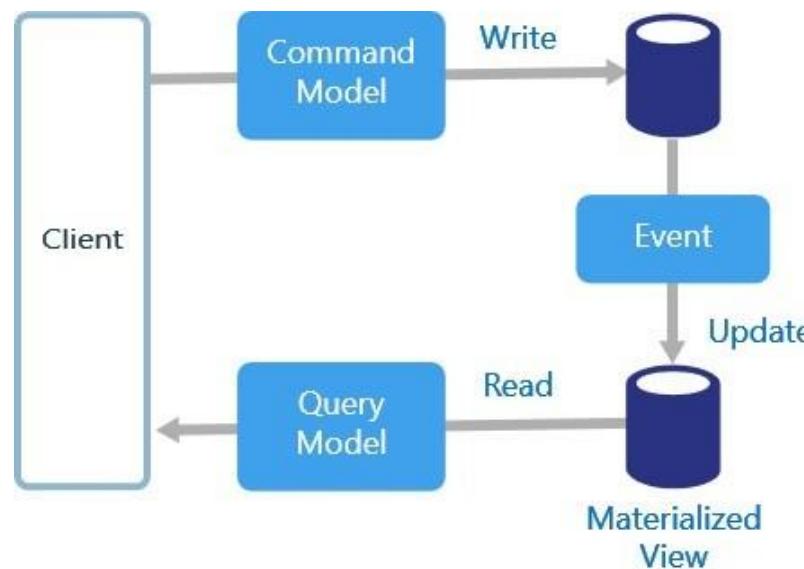
- Wenn Ihre Anwendung eine komplexere Domäne hat, überlegen Sie einen Wechsel auf eine Microservices-Architektur. Eine Microservices-Anwendung besteht aus vielen kleinen, unabhängigen Services. Jeder Dienst implementiert eine einzelne Business-Fähigkeit. Die Services sind lose gekoppelt und kommunizieren über API-Verträge. Jeder Service kann von einem kleinen, fokussierten Entwicklerteam erstellt werden.
- Einzelne Services können ohne große Koordinierung zwischen den Teams bereitgestellt werden, was die Möglichkeit häufiger Aktualisierungen fördert. Eine Microservice-Architektur ist komplexer zu entwickeln und als N-Schichten oder Web-Queue-Worker schwerer zu verwalten. Das erfordert eine ausgereifte Entwicklung und DevOps-Kultur. Aber richtig gemacht, kann dieser Stil zu höherer Veröffentlichungsgeschwindigkeit, schnellerer Innovation und einer widerstandsfähigeren Architektur führen.



e) Microsoft Cloud Architekturstile

- CQRS

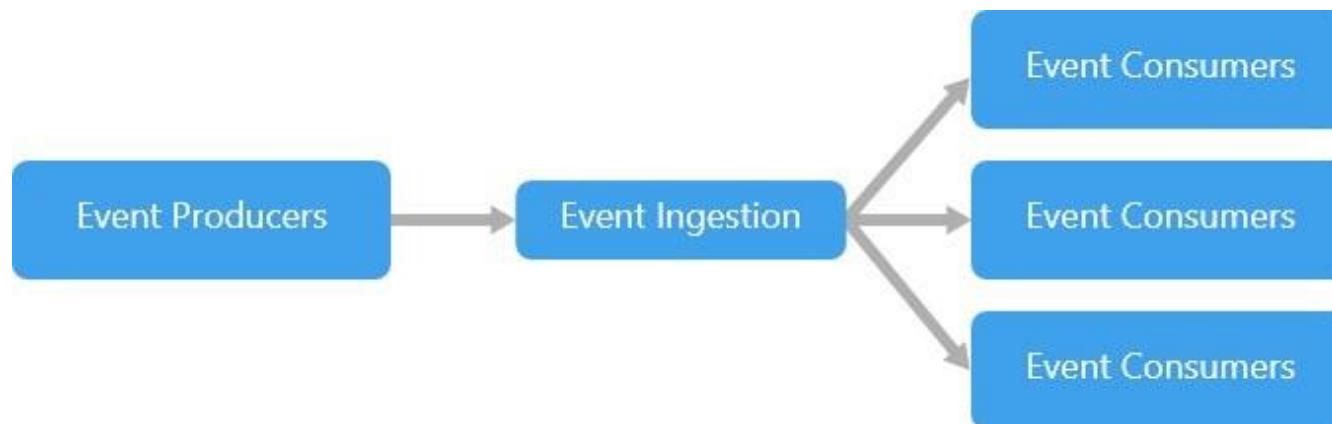
- Der CQRS-Stil – Command and Query Responsibility Segregation (CQRS) Pattern – trennt Lese- und Schreibvorgänge in separate Modelle. Dies isoliert die Teile des Systems, die Daten von den Teilen aktualisieren, von denen die Daten lesen. Darüber hinaus können Lesevorgänge für eine materialisierte Ansicht ausgeführt werden, die physisch von der Schreibdatenbank getrennt ist. Dadurch können Sie die Workloads für das Lesen und Schreiben unabhängig skalieren und die materialisierte Ansicht für Abfragen optimieren.
- CQRS ergibt vor allem dann Sinn, wenn es auf ein Subsystem einer größeren Architektur angewendet wird.
- CQRS sollte nicht für eine gesamte Anwendung genutzt werden, da dies nur unnötige Komplexität verursacht. Die Verwendung ist für kollaborative Domänen, in denen viele Benutzer auf dieselben Daten zugreifen, optimal.



e) Microsoft Cloud Architekturstile

- Ereignisgesteuerte Architektur

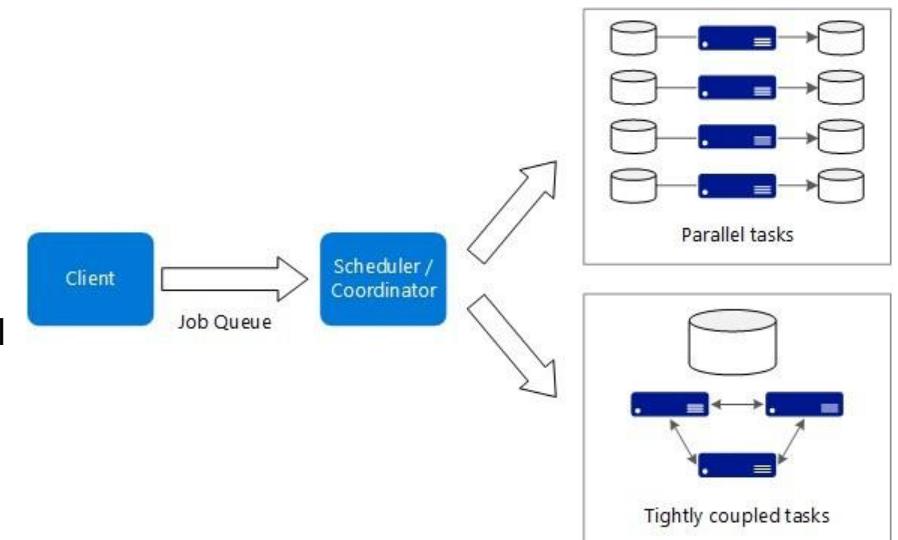
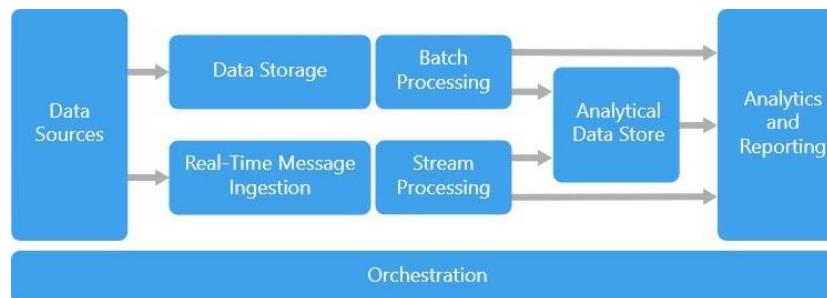
- Ereignisgesteuerte Architekturen verwenden ein Publish-Subscribe-Modell (Pub-Sub-Modell), bei dem die Produzenten Ereignisse veröffentlichen und Kunden sie abonnieren.
- Die Produzenten sind unabhängig von den Konsumenten und die Konsumenten sind untereinander auch unabhängig.
- Hauptsächliche Verwendung bei ereignisgesteuerter Architektur für Anwendungen, die große Datenmengen mit sehr geringer Latenz verarbeiten, z. B. IoT-Lösungen. Der Stil ist auch nützlich, wenn verschiedene Subsysteme unterschiedliche Verarbeitungstypen für dieselben Ereignisdaten ausführen müssen.



e) Microsoft Cloud Architekturstile

- Big Data und Big Compute

- Big Data und Big Compute sind spezialisierte Architekturstile für Workloads, die bestimmten spezifischen Profilen entsprechen.
- Big Data unterteilt sehr große Daten-Sets in Chunks und führt die parallele Verarbeitung über das gesamte Set hinweg für Analysen und Berichte durch.



- Big Compute, auch High-Performance Computing (HPC) genannt, führt parallele Berechnungen über eine große Anzahl von (tausenden) Kernen durch.
- Zu den Domänen gehören Simulationen, Modellierungen und 3D-Rendering.

e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Selbstheilende Anwendungen
 - In verteilten Systemen gibt es immer Fehler. Anwendungen sollten Fehler annehmen und entsprechend darauf reagieren (selbstheilend)
 - Redundanz
 - Um SPOF (Single point of failure) Situationen zu vermeiden
 - Abhängigkeit (Koordination)
 - Abhängigkeiten verhindern Skalierbarkeit
 - Horizontale Skalierbarkeit
 - Wegnehmen und Hinzufügen von Instanzen ermöglicht horizontale Skalierung
 - Partitionierung
 - Partitionierung umgeht Einschränkungen bei Datenbanken, Netzwerken und Rechenleistung
 - Betriebsorientierung
 - Anwendungen sind in die Betriebsabläufe und deren Tools integriert
 - Managed Services
 - Wann immer möglich Platform as a Service (PaaS) anstatt Infrastructure as a Service (IaaS) nutzen
 - Aufgabenorientierte Datenablage
 - Aufgabenorientierte Auswahl der geeigneten Technologie zur Ablage der Daten
 - Evolutionäre Architektur
 - Ein auf Änderungen vorbereitetes Anwendungsdesign
 - Anforderungsorientierung
 - Jeder Designentscheidung steht eine Anforderung aus dem Business gegenüber

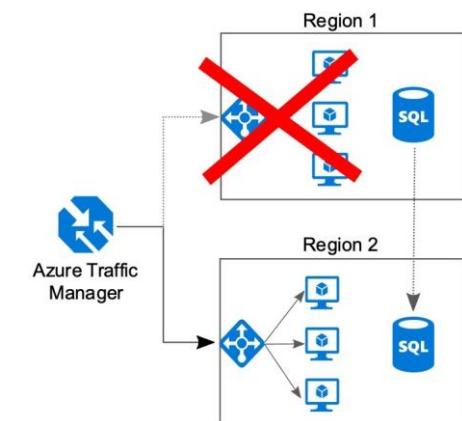
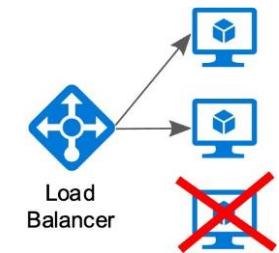
e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Selbstheilende Anwendungen
 - In verteilten Systemen gibt es immer Fehler. Anwendungen sollten Fehler annehmen und entsprechend darauf reagieren (selbstheilend)
 - Strategie
 - Fehler entdecken
 - Auf Fehler geeignet reagieren
 - Fehler festhalten (Logging) und verfolgen (Monitoring), um einen reibungslosen Betrieb zu gewährleisten
 - Empfehlungen
 - Retry pattern
 - Circuit breaker pattern
 - Isolation (Bulkheads)
 - Queue based load leveling pattern
 - Fail over
 - Failed Transaction compensating
 - Checkpoints (Wiederanlauf lang laufender Aktionen an geeigneten Punkten)
 - Graceful Degregation
 - Drosselung (Throtteling)
 - Blocken von Usern (nach mehrfachem Fehlverhalten)
 - Nutzung von Tools wie Apache Zookeeper
 - Fehlersimulationen ausgiebig testen
 - Chaos engineering (Fehlererzeugung im Produktionssystem)



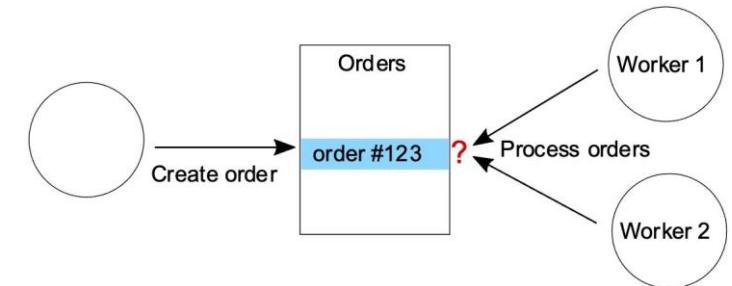
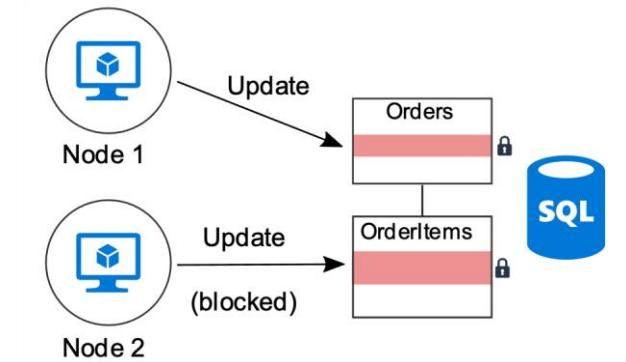
e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Redundanz
 - Um SPOF (Single point of failure) Situationen zu vermeiden
 - Strategie
 - Bestimmung des kritischen Pfads
 - Ist im kritischen Pfad der Anwendung Redundanz vorhanden
 - Kann die Anwendung mit Ausfällen umgehen
 - Empfehlungen
 - Business Anforderungen betrachten (Kosten-, Nutzenanalyse)
 - VM's hinter einem Loadbalancer
 - Datenbank Replizierung
 - Geo Replikation (Replikation über Regionen)
 - Datenbank Partitionierung
 - Verteilung über Regionen
 - Nutzung vom Azure Traffic Manager für Failover Szenarien



e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Abhängigkeit (Koordination)
 - Abhängigkeiten verhindern Skalierbarkeit
 - Strategie
 - Datenbank Locks vermeiden (ACID Garantie)
 - Exactly once Anforderungen vermeiden
 - Empfehlungen
 - Eventuelle Konsistenz akzeptieren
 - CQRS pattern
 - Event Sourcing pattern
 - Daten Partitionierung
 - Idempotente Operationen
 - Asynchrone parallele Verarbeitung
 - Optimistische Concurrency nutzen
 - MapReduce und andere parallele Algorithmen nutzen (Big compute)
 - Leader election pattern



e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Horizontale Skalierbarkeit
 - Wegnehmen und Hinzufügen von Instanzen ermöglicht horizontale Skalierung
 - Strategie
 - Design lässt das flexibles Ändern von Instanzen zu
 - Empfehlungen
 - Abhängigkeiten zwischen Klients und Servern vermeiden (Request vom Client geht immer zum selben Server)
 - Bottlenecks identifizieren
 - Workloads entzerren (unterschiedliche Clients für unterschiedliche Aufgaben)
 - Ressourcen intensive Aufgaben verteilen
 - Eingebaute Autoscaling Features der Plattform nutzen
 - Aggressives Autoscaling anwenden (schnelles hochfahren zusätzlicher Instanzen trotz kurzfristiger zusätzlicher Last)
 - Herunterfahren, Stilllegen von Instanzen planen (im Anwendungsdesign)

e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Partitionierung
 - Partitionierung umgeht Einschränkungen bei Datenbanken, Netzwerken und Rechenleistung
 - Strategie
 - Datenbankpartitionierung
 - Horizontal (Sharding), vertikal (nach Nutzungsgrad), funktional (nach Kontext)
 - Empfehlungen
 - Nicht nur Datenbanken sondern auch Storage, Caches, Queues und Compute Instanzen partitionieren
 - Hotspots vermeiden (geeignete Partitionierungsmethode wählen)

e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Betriebsorientierung
 - Anwendungen sind in die Betriebsabläufe und deren Tools integriert
 - Strategie
 - Deployment, Monitoring, Escalation, Incident management, Security auditing betrachten
 - Robustes logging und tracing
 - Das Betriebsteam in der Design- und Planungsphase einbinden
 - Empfehlungen
 - Alles überwachbar machen
 - Verteiltes Tracing (Möglichkeit der Zusammenfassung über sog. Correlation ID's)
 - Standardisierung von Logs und Metriken
 - Automatisierung
 - Konfigurationsdaten als Code ansehen (und im Version Control System halten)

e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Managed Services
 - Wann immer möglich Platform as a Service (PaaS) anstatt Infrastructure as a Service (IaaS) nutzen
 - Strategie
 - Plattform Services nutzen
 - Empfehlungen

Anstatt	Besser
Active Directory	Azure Active Directory
Elasticsearch	Azure Search
Hadoop	HDInsight
IIS	App Service
MongoDB	Cosmos DB
Redis	Azure Cache for Redis
SQL Server	Azure SQL Database
File share	Azure NetApp Files

e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Aufgabenorientierte Datenablage
 - Aufgabenorientierte Auswahl der geeigneten Technologie zur Ablage der Daten
 - Strategie
 - ACID kritisch betrachten
 - Normalisierung nicht übertreiben (komplizierte Joins vermeiden)
 - Locks vermeiden
 - Empfehlungen
 - Nicht immer und für alles RDBMS nutzen
 - Gemischte Technologien nutzen (NoSQL)
 - Die Art der Daten betrachten (transaktionsorientiert, Dokumente, Telemetriedaten, Zeit, Logs, Blobs)
 - Wichte Verfügbarkeit über strenge Konsistenz
 - Skills des Entwicklungsteams berücksichtigen
 - SAGA pattern (compensating transactions)
 - Bounded Context betrachten

e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Evolutionäre Architektur
 - Ein auf Änderungen vorbereitetes Anwendungsdesign
 - Strategie
 - Nicht nur ein Problem von Monolithen
 - Microservices in Erwägung ziehen
 - Empfehlungen
 - Hohe Kohäsion
 - Lose Kopplung
 - Kapselung von Domain Wissen in Modulen
 - Asynchrone Kommunikation
 - Kein Domain Wissen in Gateways
 - Offene Interfaces anbieten
 - Gut definierte API's einsetzen (entwickeln und testen einfacher)
 - Domain Logik und Infrastruktur strikt trennen
 - Querschnittsthemen in eigenen Services auslagern
 - Unabhängiges Deployment aller Services

e) Microsoft Cloud Architekturstile

- Microsoft Design Prinzipien für Azure Anwendungen
 - Anforderungsorientierung
 - Jeder Designentscheidung steht eine Anforderung aus dem Business gegenüber
 - Strategie
 - Anforderungen zusammen mit dem Anforderer betrachten und Entscheidungen daraus ableiten
 - Empfehlungen
 - recovery time objective (RTO), recovery point objective (RPO), and maximum tolerable outage (MTO) betrachten
 - Service Level Agreements (SLA) und Service Level Objectives (SLO) genau dokumentieren
 - Domain Driven Design anwenden
 - Functional von non-functional Requirements trennen
 - Workloads planen und voneinander trennen
 - Wachstum einplanen
 - Kostenbetrachtung vornehmen

e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Herausforderungen
 - Verfügbarkeit
 - Daten Management
 - Design und Implementierung
 - Messaging
 - Management und Monitoring
 - Performance und Skalierbarkeit
 - Resilienz
 - Security
 - Pattern geben Lösungsvorschläge

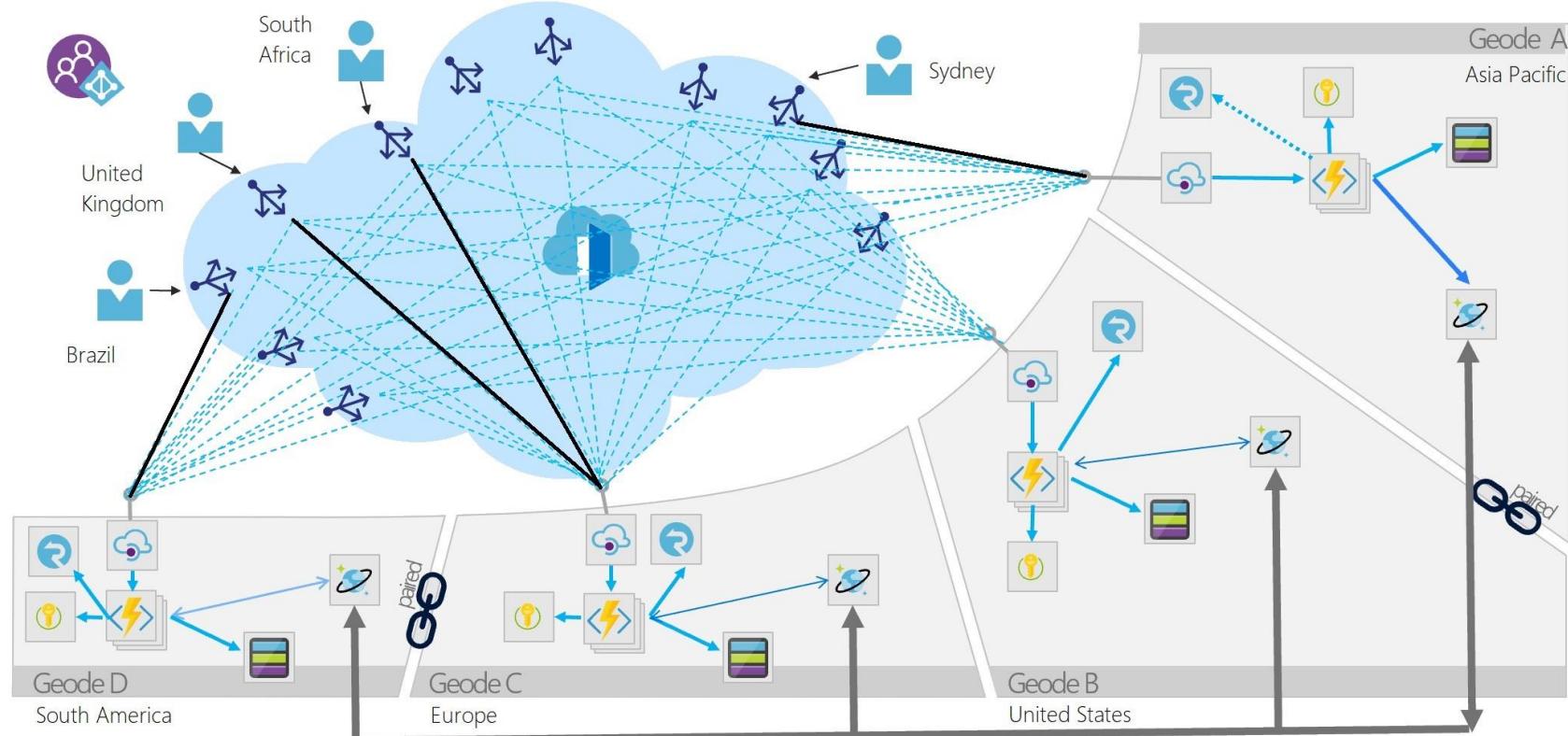
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Verfügbarkeit

Pattern	Überblick
Deployment Stamps	Deployment mehrerer unabhängiger Kopien von Anwendungskomponenten incl. Datenablagen
Geodes	Deployment von backend services in sog. geographical nodes, die Anfragen von Klients in deren Regionen bedienen
Health Endpoint Monitoring	Implementierung von funktionalen Checks in Anwendungen, die von externen Tools von Zeit zu Zeit getestet werden können
Queue-Based Load Leveling	Queue als Puffer zwischen Tasks und Services um Last abzufedern
Throttling	Kontrollierte Drosselung der Benutzung von Ressourcen

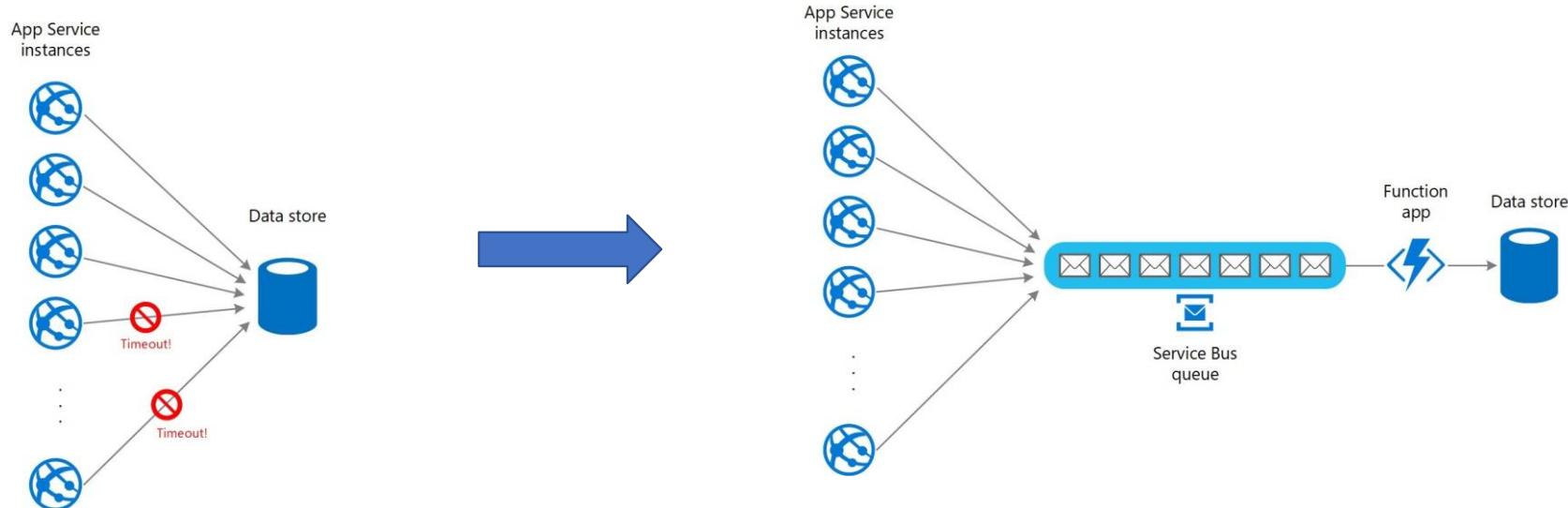
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Verfügbarkeit (Geodes)



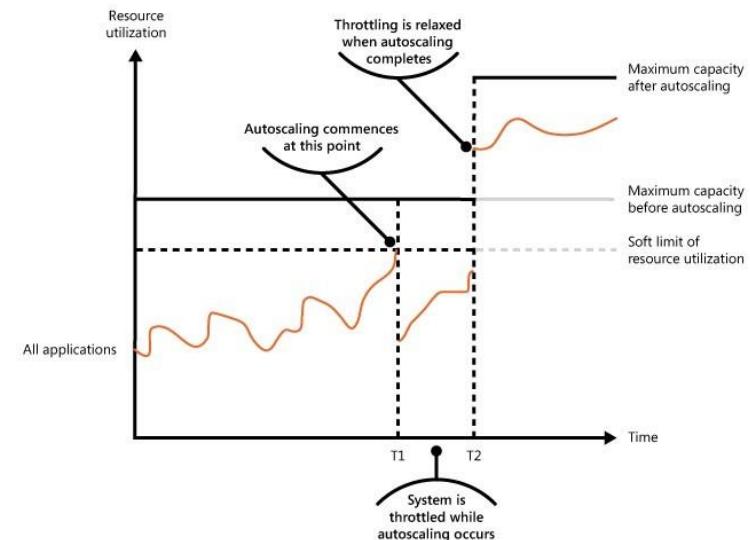
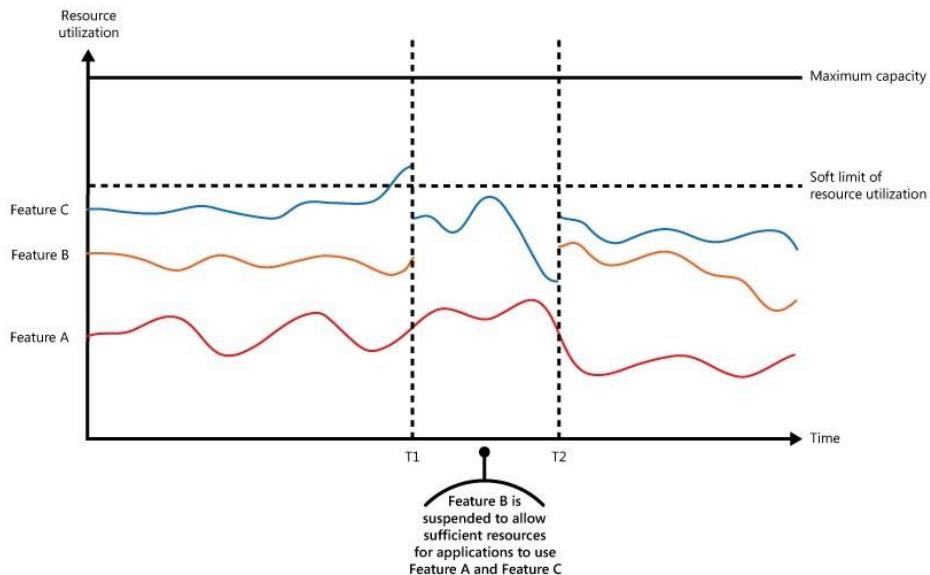
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Verfügbarkeit (Queue-based load-leveling)



e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Verfügbarkeit (Throttling)



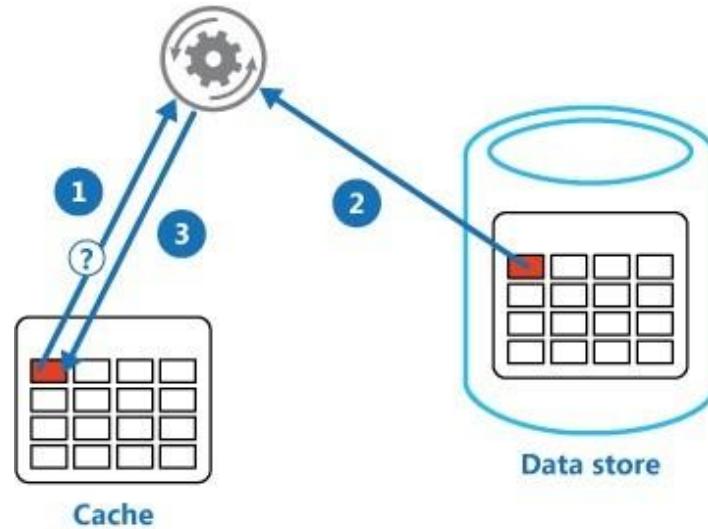
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Daten Management

Pattern	Überblick
Cache-Aside	Zusammen mit einer Datenbank einen Cache nutzen
CQRS	Trennung von Schreib- und Lesekanal
Event Sourcing	Append-only Queue, um die gesamte Reihe von Ereignissen aufzuzeichnen, die Aktionen beschreiben, die für Daten in einer Domäne ausgeführt wurden
Index Table	Indizes über die Felder, auf die häufig von Abfragen verwiesen wird (gilt nicht nur für Datenbanktabellen)
Materialized View	Vorab ausgefüllte Ansichten über Daten von einem oder mehreren Datenspeichern, wenn die Daten für die erforderlichen Abfragen nicht ideal formatiert sind
Sharding	Aufteilung eines Datenspeichers in eine Reihe horizontaler Partitionen oder Shards
Static Content Hosting	Statische Inhalte in einem Cloud Speicher halten, Links darauf können direkt an den Client gesendet werden
Valet Key	Token oder einen Schlüssel, für den eingeschränkten direkten Zugriff auf eine bestimmte Ressource oder einen bestimmten Dienst

e) Microsoft Cloud Architekturstile

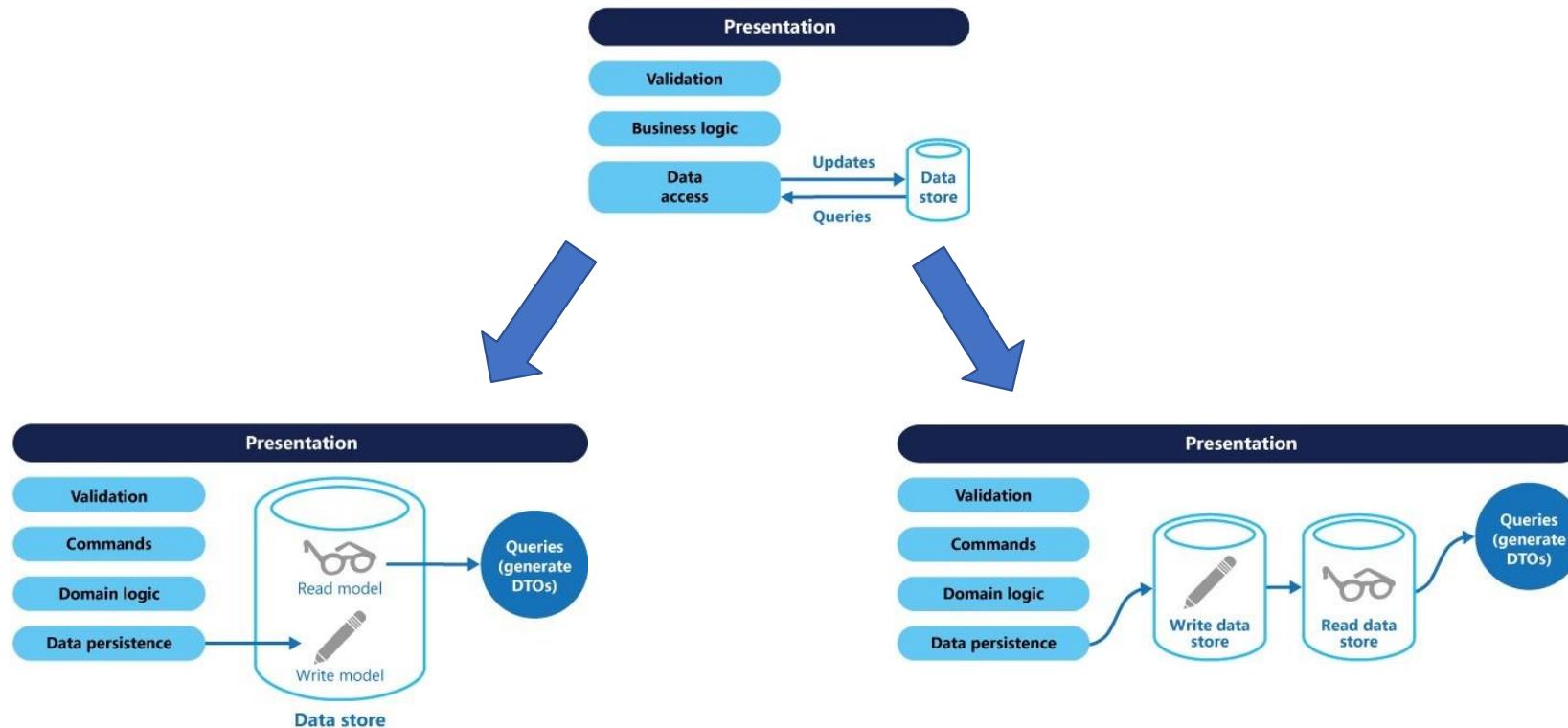
- Microsoft Cloud Design Pattern
 - Daten Management (Cache aside)



- 1: Determine whether the item is currently held in the cache.
- 2: If the item is not currently in the cache, read the item from the data store.
- 3: Store a copy of the item in the cache.

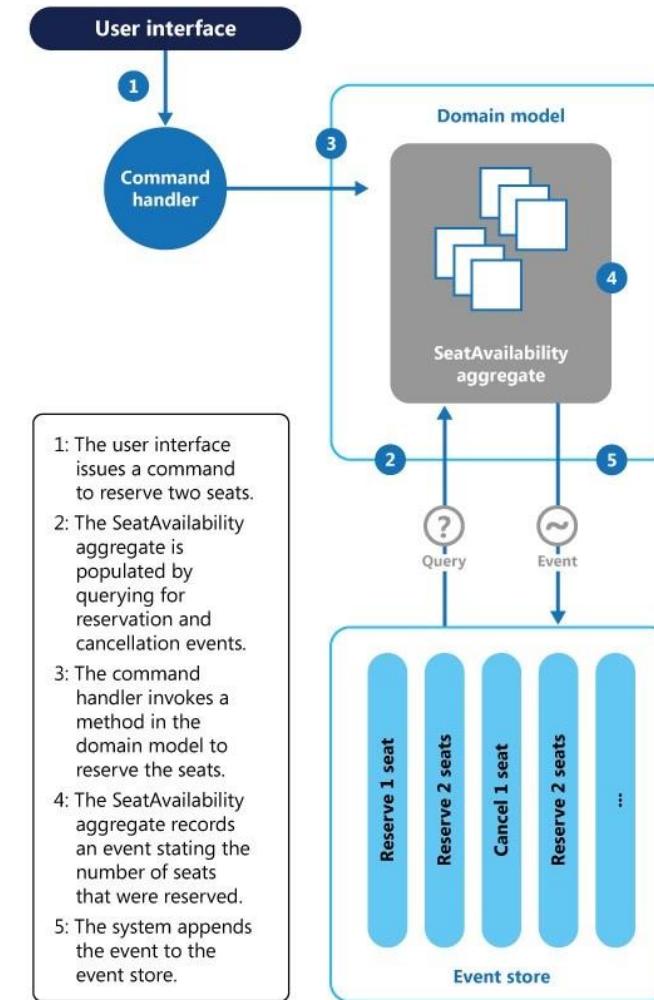
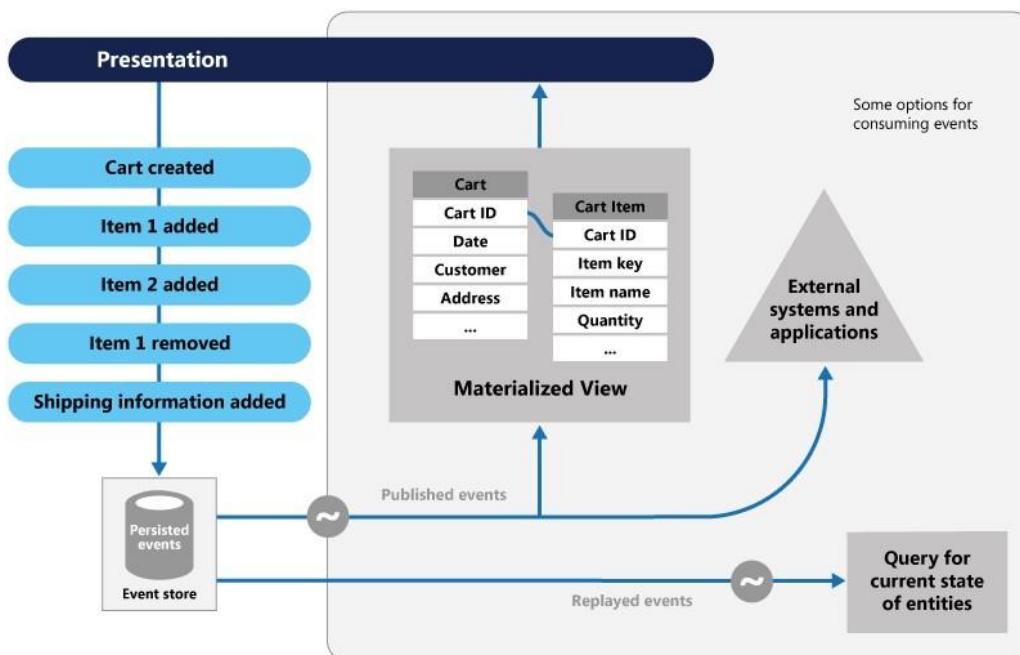
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Daten Management (CQRS)



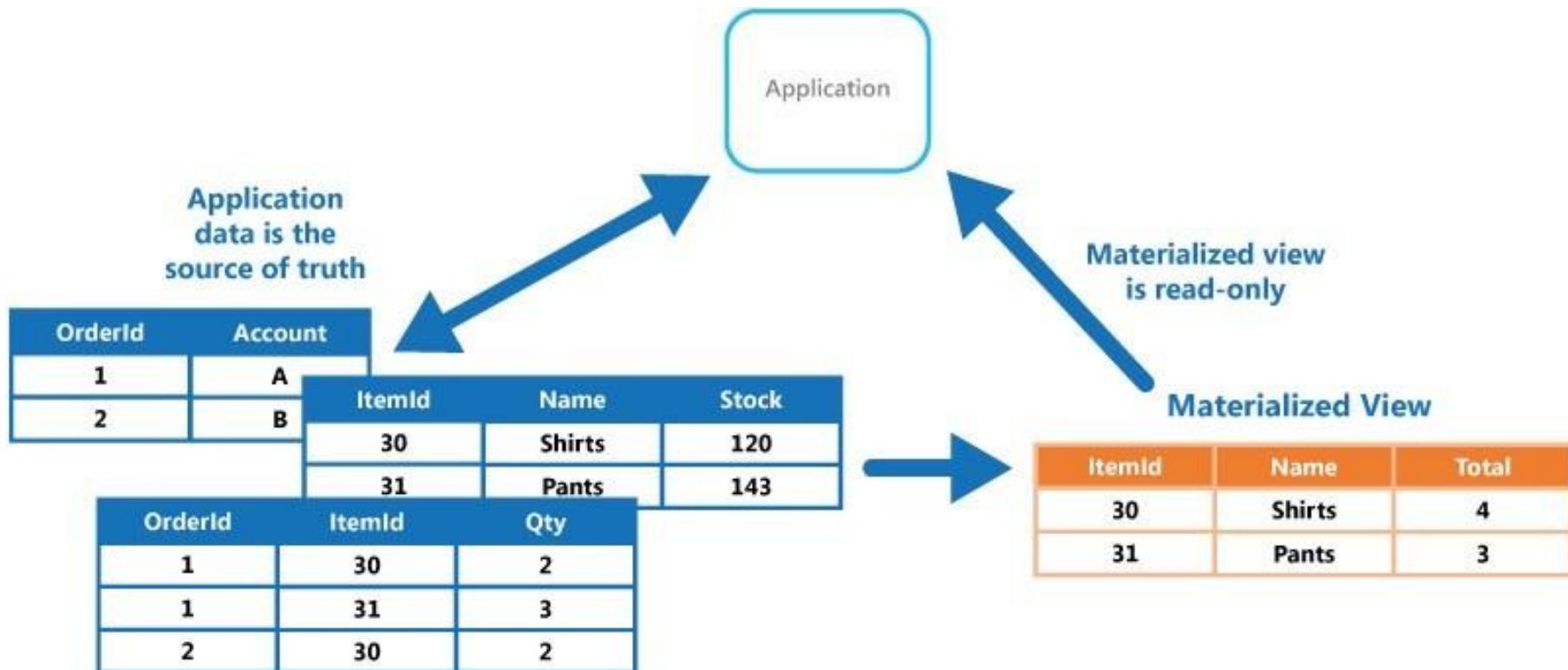
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Daten Management (Event sourcing)



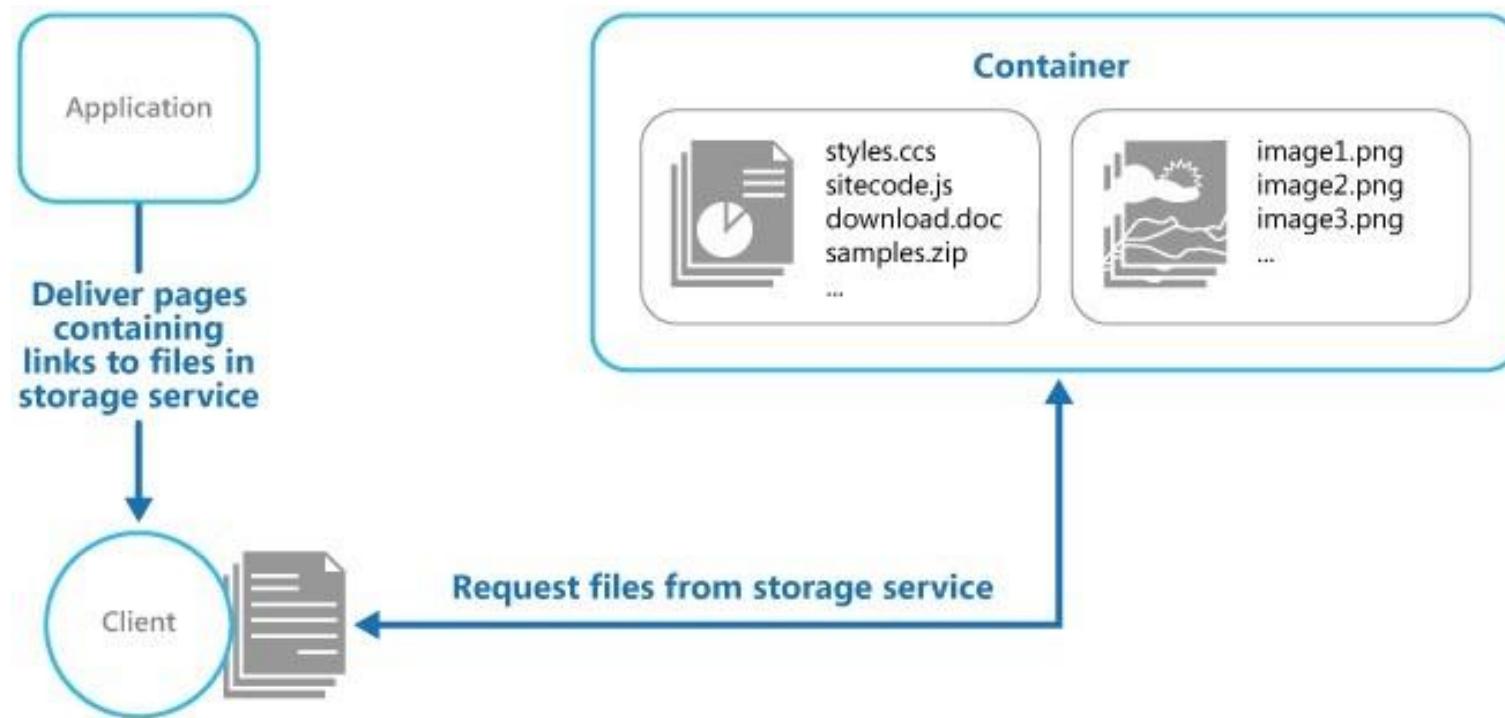
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Daten Management (Materialized view)



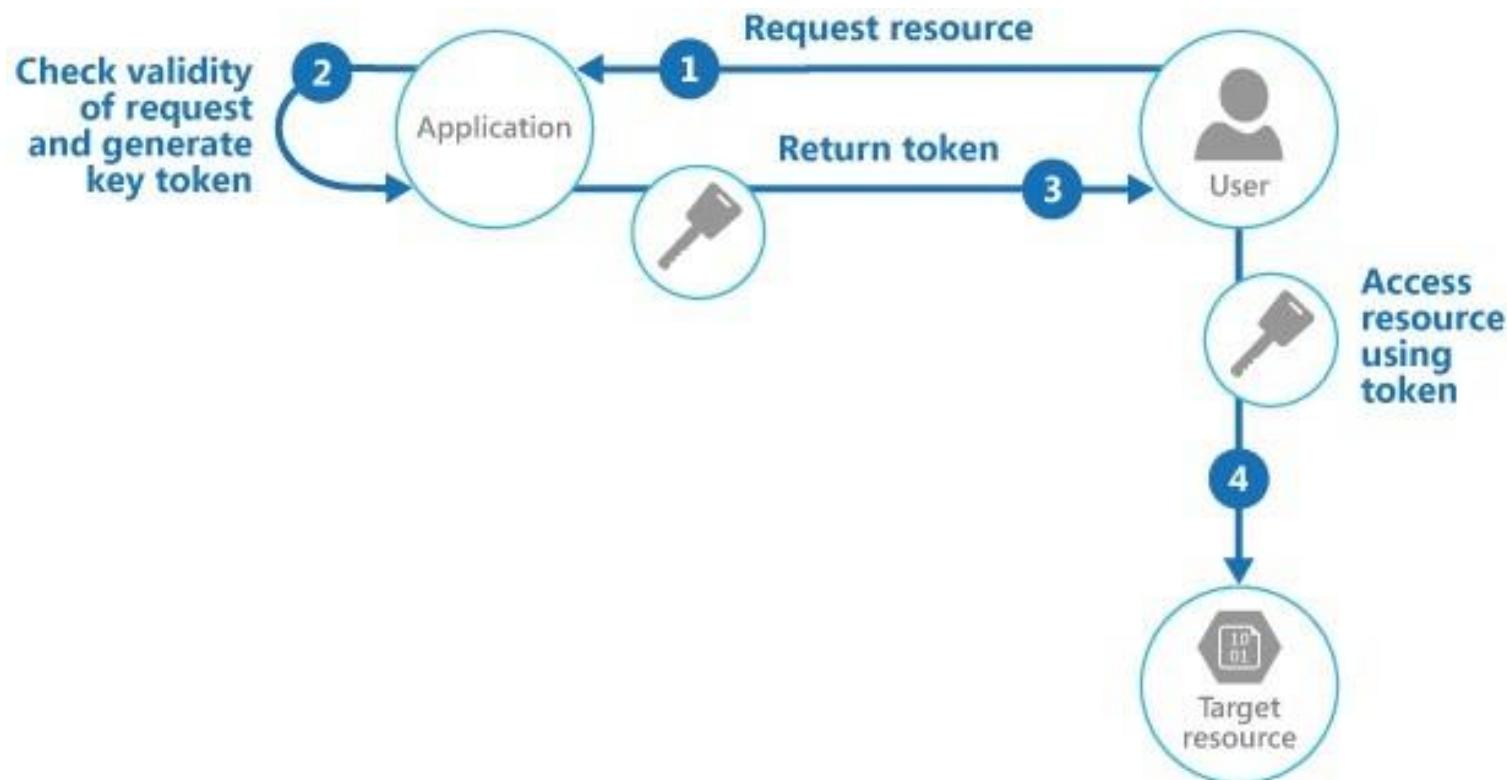
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Daten Management (Static content hosting)



e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Daten Management (Valet key)



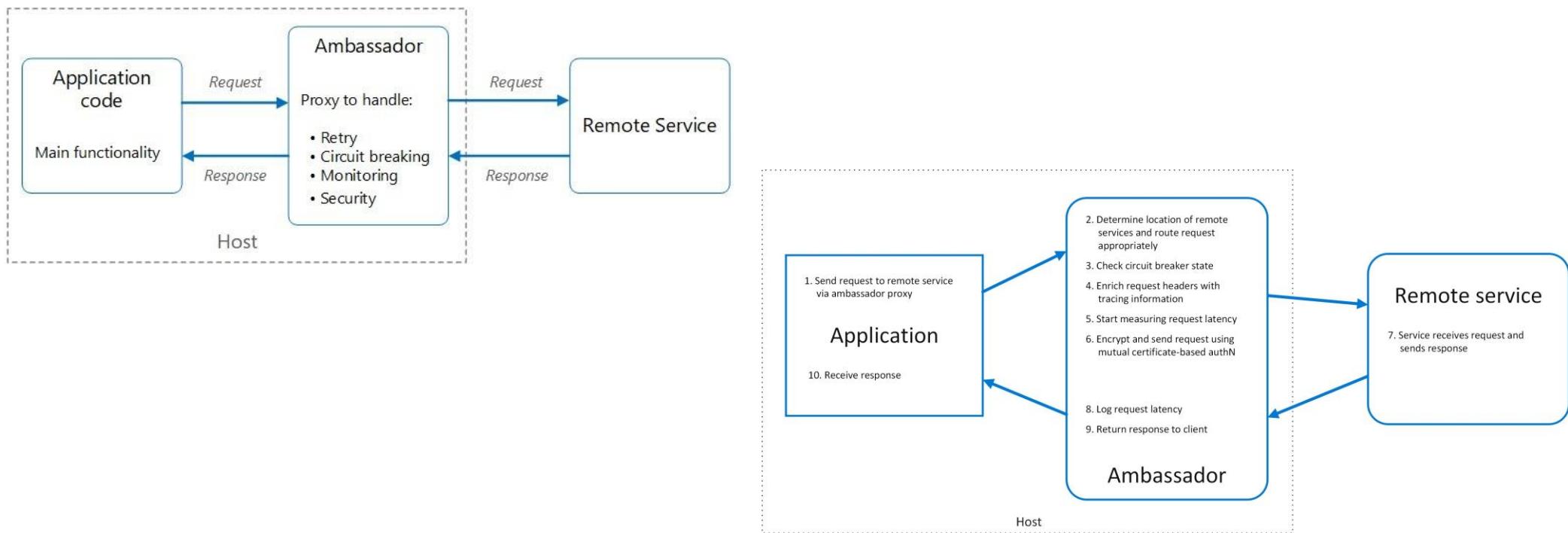
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung

Pattern	Summary
Ambassador	Hilfsdienste für Netzwerkanforderungen von Verbraucherdiensten oder Anwendungen
Anti-Corruption Layer	Fassade- oder Adapterschicht zwischen einer modernen Anwendung und einem Altsystem
Backends for Frontends	Separate Backend-Services für bestimmte Frontend-Anwendungen oder -Schnittstellen
CQRS	Kennen wir schon
Compute Resource Consolidation	Konsolidieren von mehreren Aufgaben oder Vorgängen in einer einzigen Recheneinheit
External Configuration Store	Zentrale Zusammenfassung von Konfigurationsinformationen (nicht in der Anwendung)
Gateway Aggregation	Gateways zur Zusammenfassung von einzelnen Anforderungen
Gateway Offloading	Gemeinsam genutzte oder spezialisierte Servicefunktionen auf einen Gateway-Proxy laden
Gateway Routing	Anforderungen für mehrere Services über ein Gateway leiten
Leader Election	Koordinierung von Aktionen von zusammenarbeitenden Aufgabeninstanzen über eine Instanz, die als Leiter die Verantwortung für die Verwaltung der anderen Instanzen übernimmt
Pipes and Filters	Aufteilung einer komplexen Verarbeitung in eine Reihe separater, wiederverwendbarer Einzelemente
Sidecar	Isolation und Kapselung von Komponenten einer Anwendung in einem separaten Prozess oder Container
Static Content Hosting	Kennen wir schon
Strangler	Schrittweise Migration eines Legacy-Systems durch neue Anwendungen und Dienste

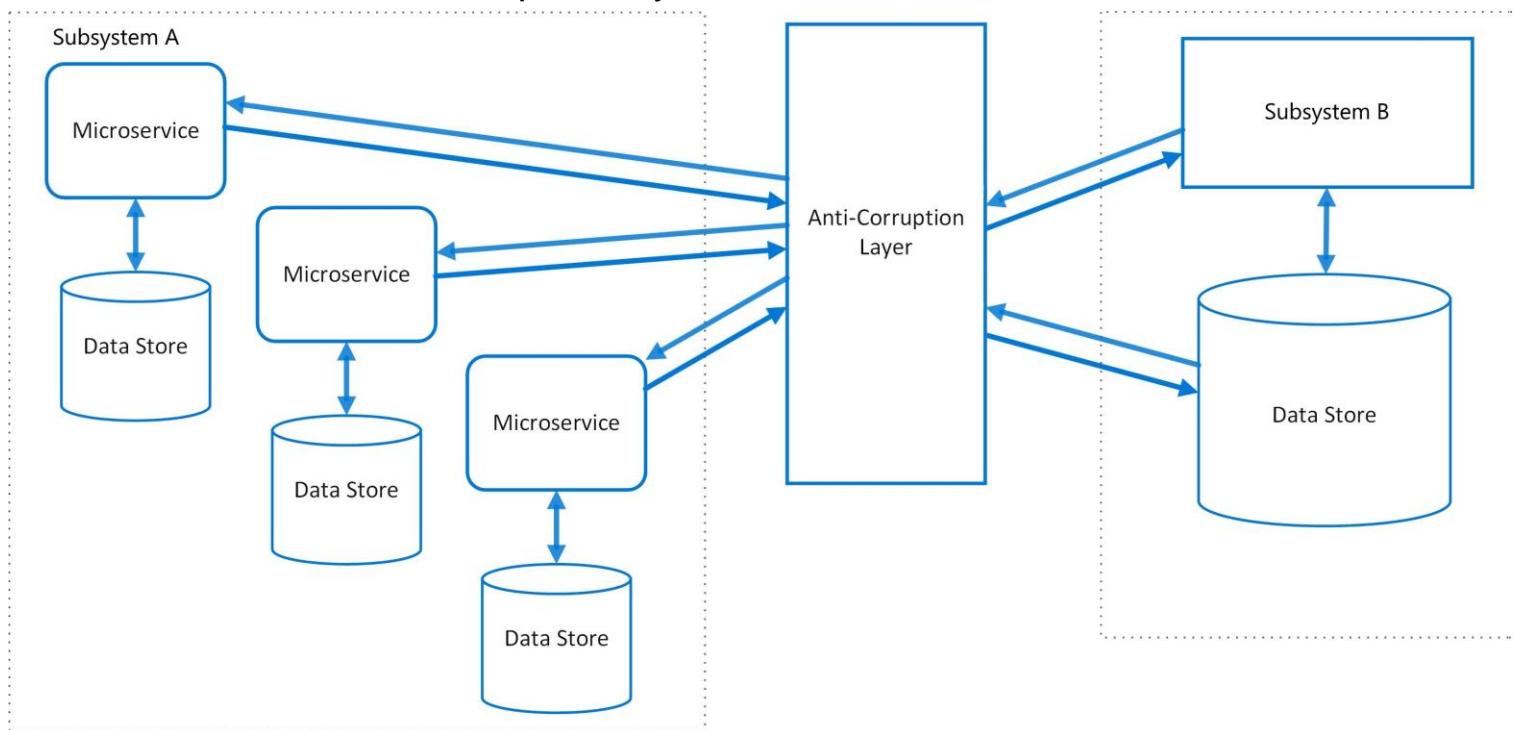
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Ambassador



e) Microsoft Cloud Architekturstile

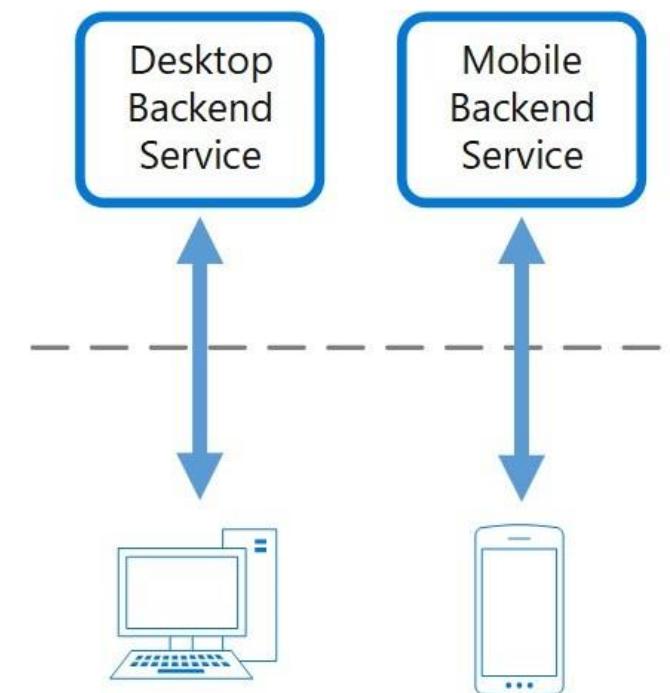
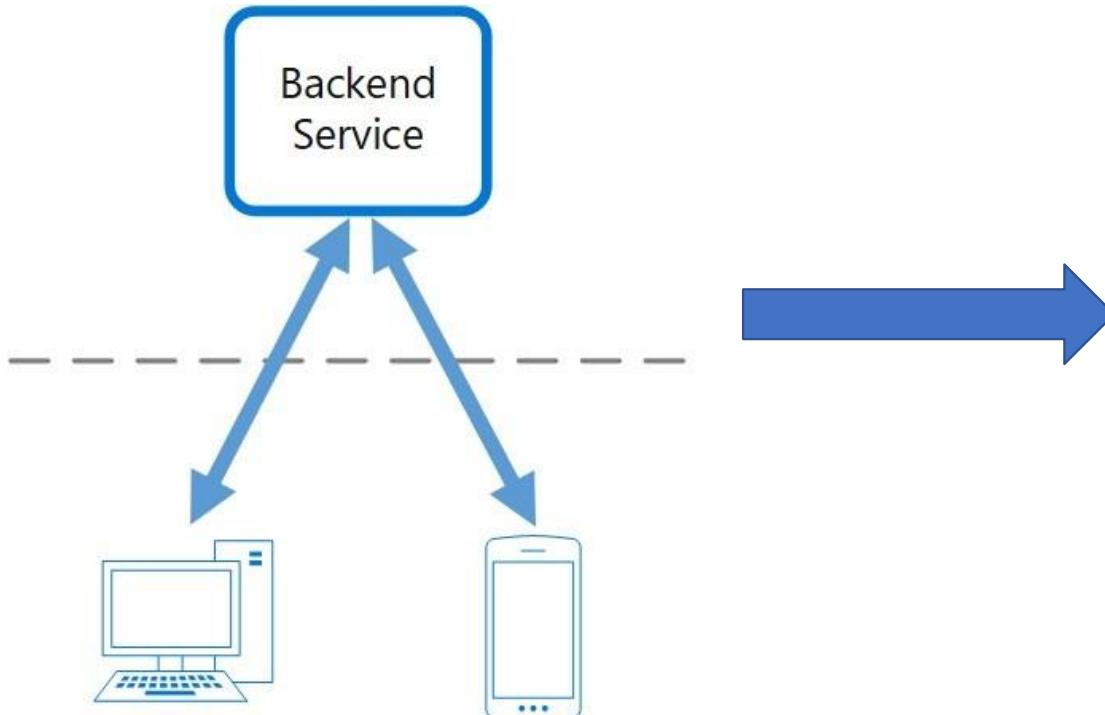
- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Anti-corruption-layer



Isolation von verschiedenen Subsystemen. Subsysteme können unabhängig voneinander geändert werden.
 Kann auch zur Absicherung genutzt werden.

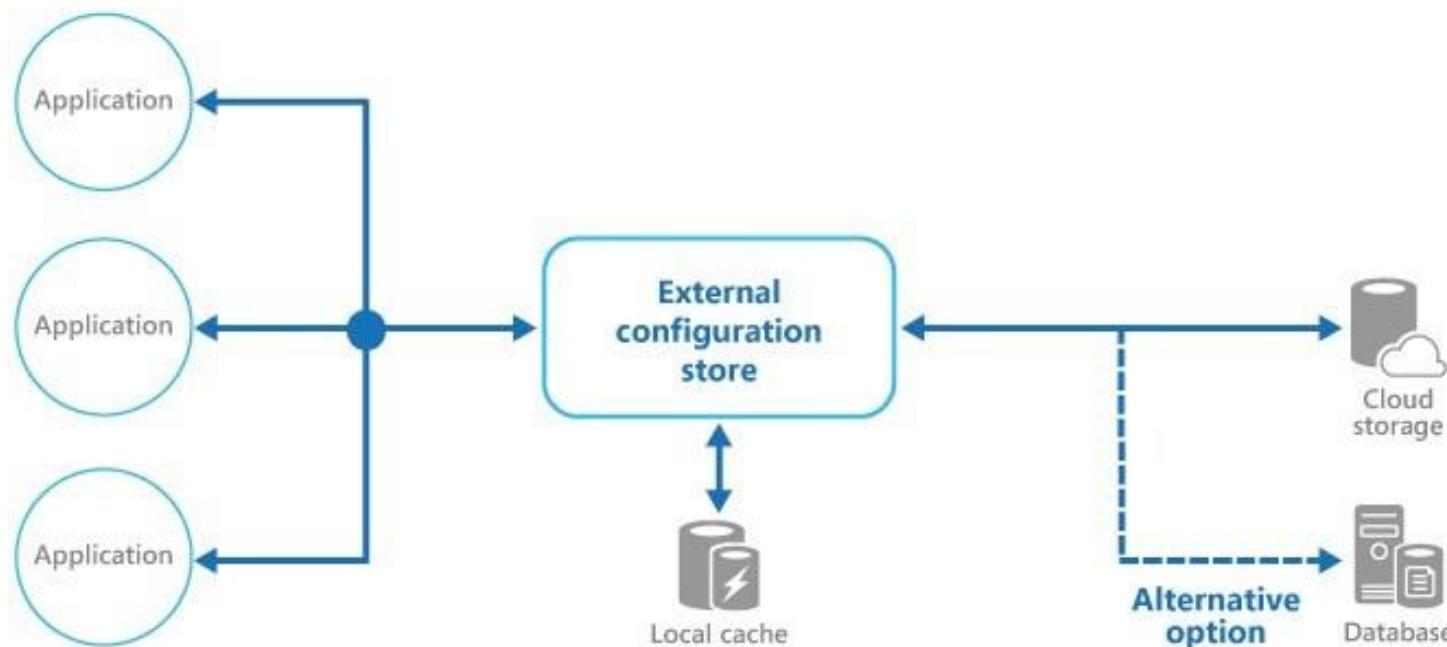
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Backends for Frontends



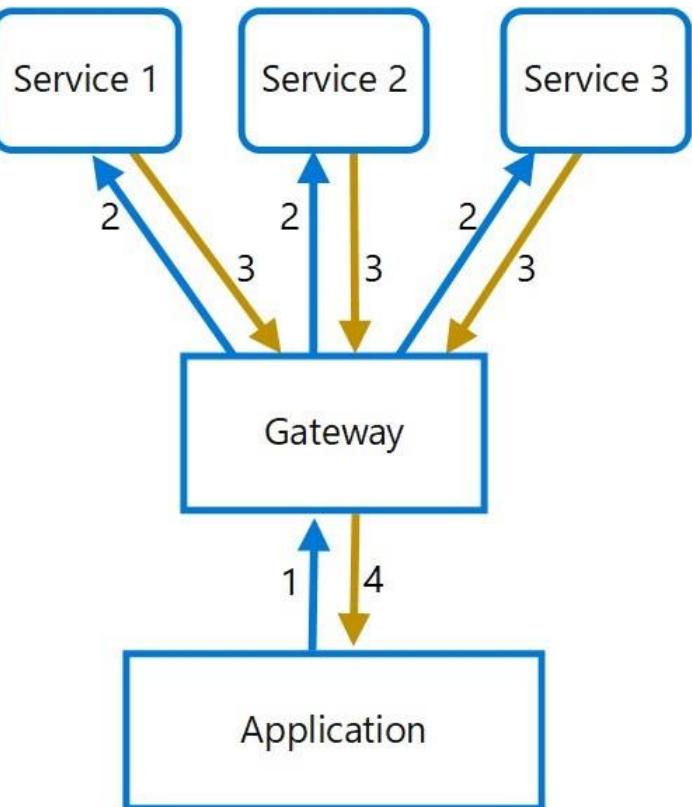
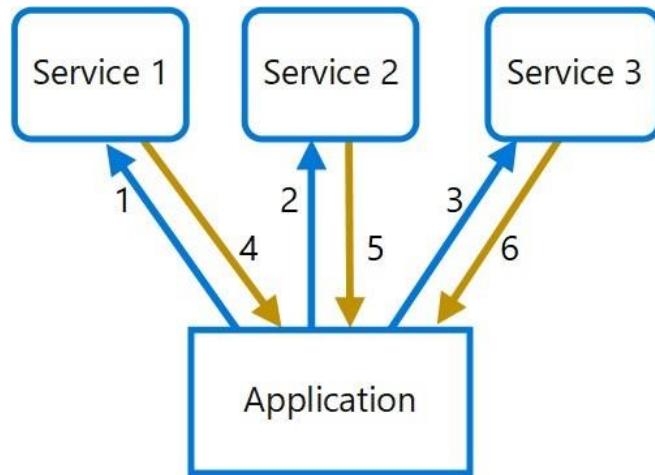
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - External configuration store



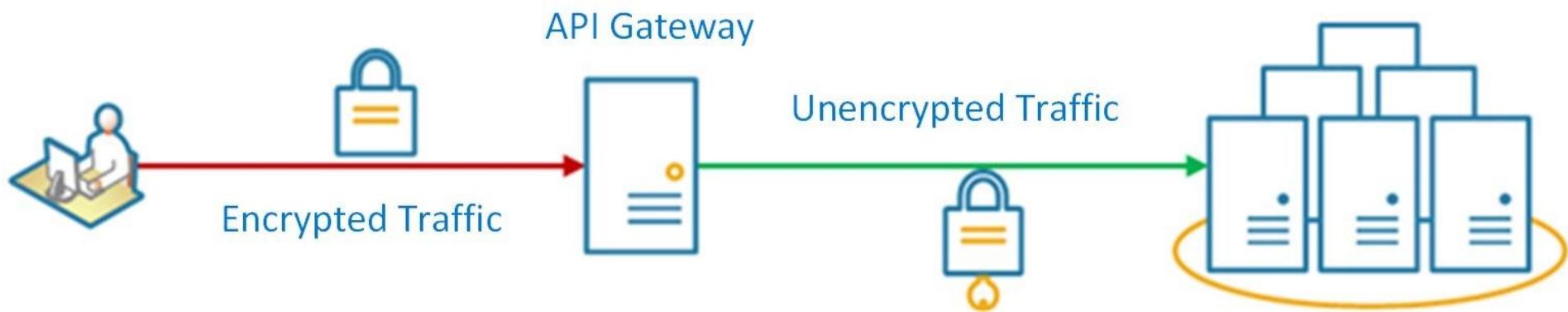
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Gateway aggregation



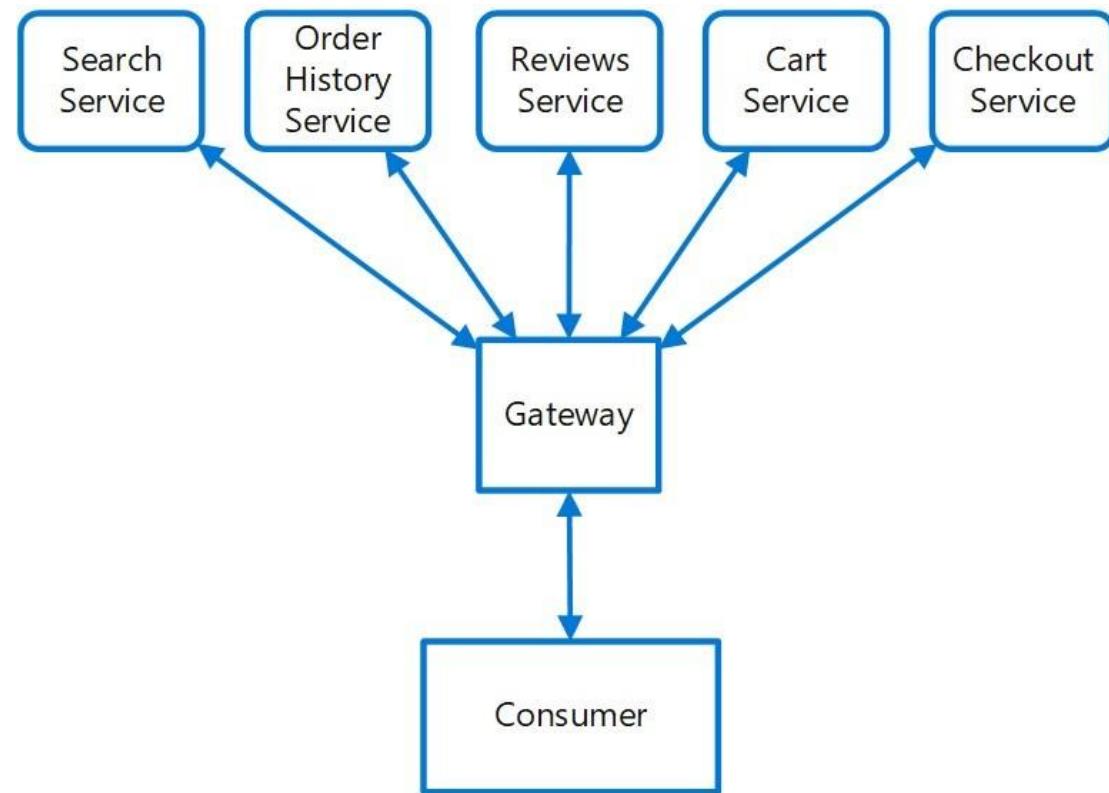
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Gateway offloading



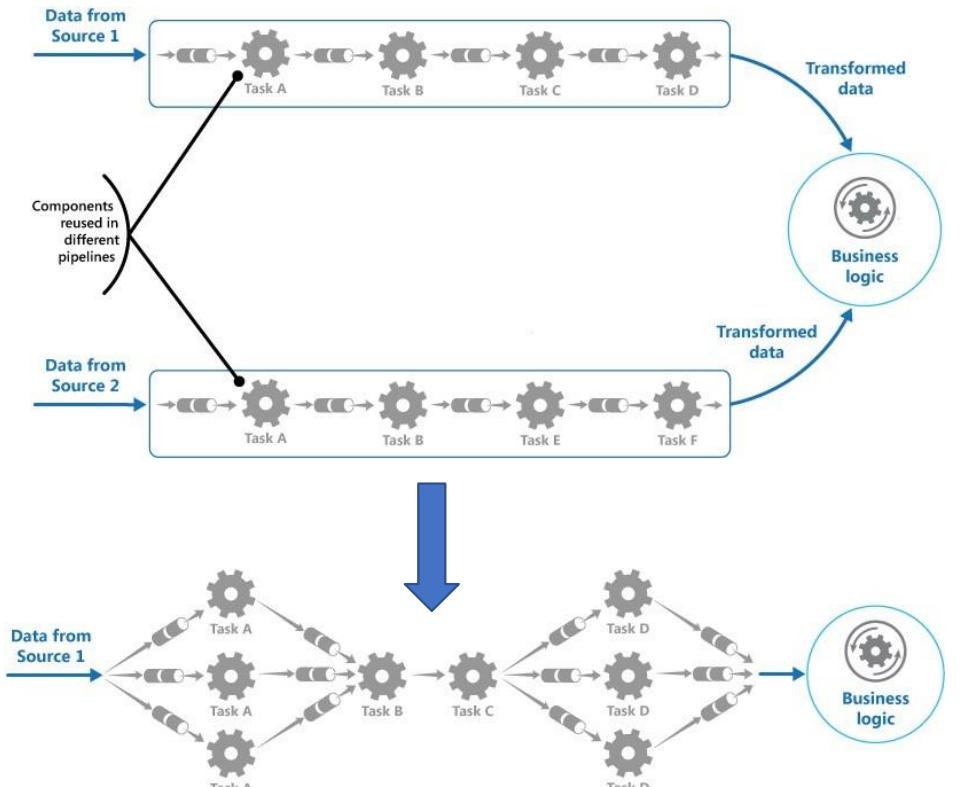
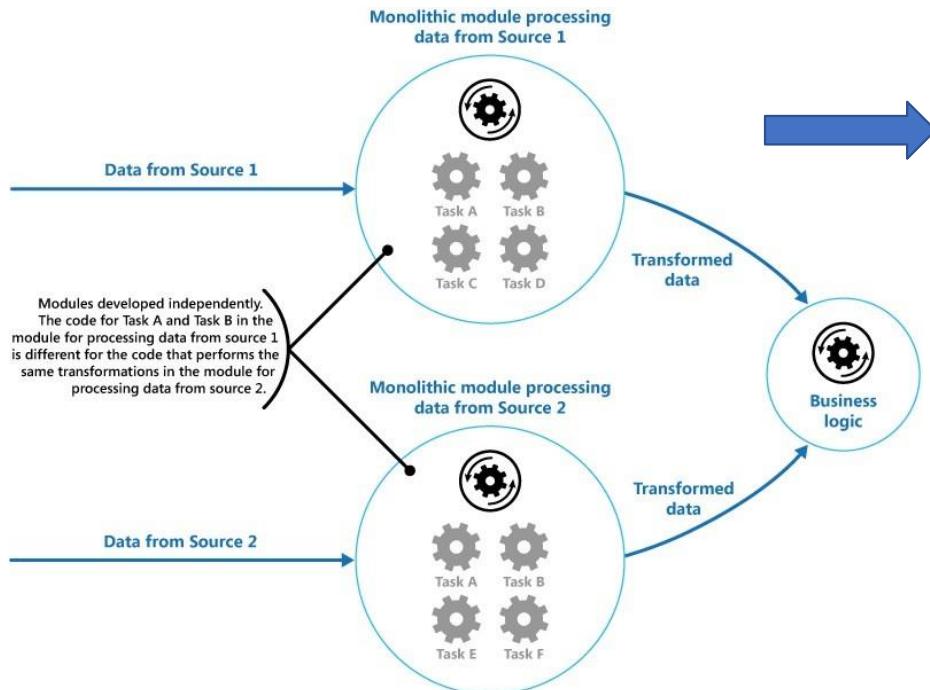
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Gateway routing



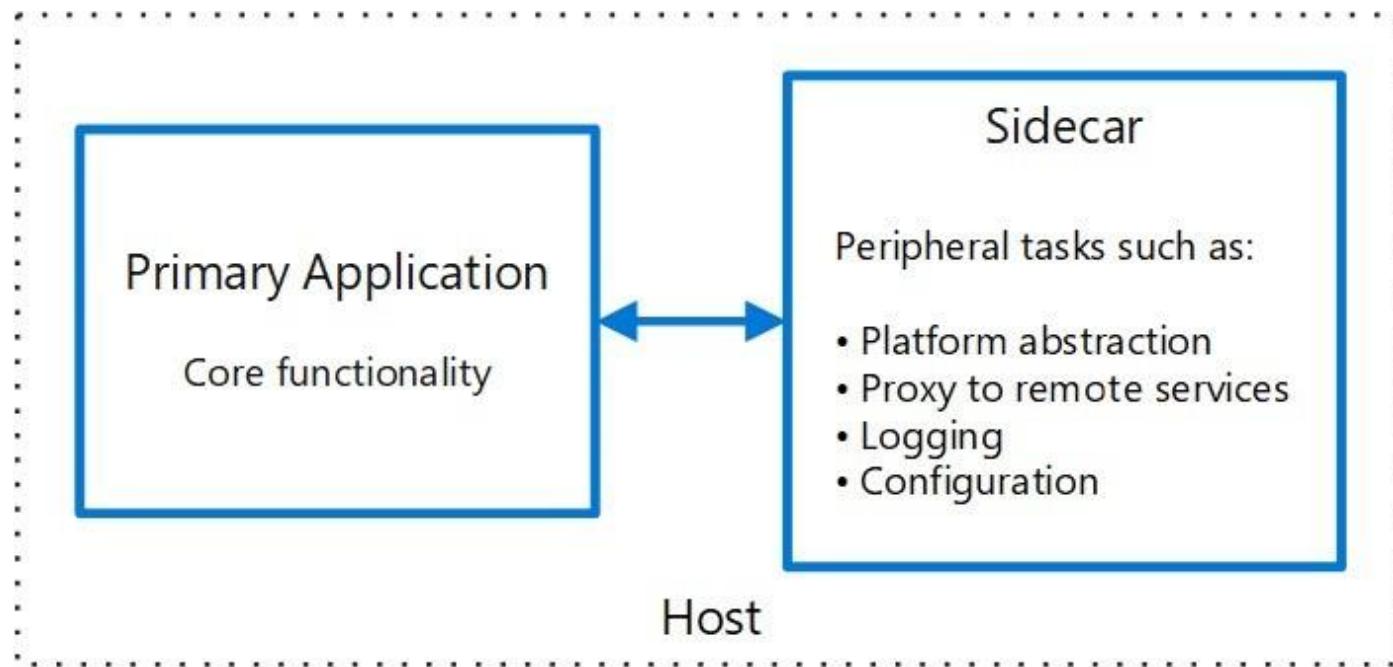
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Pipes and filters



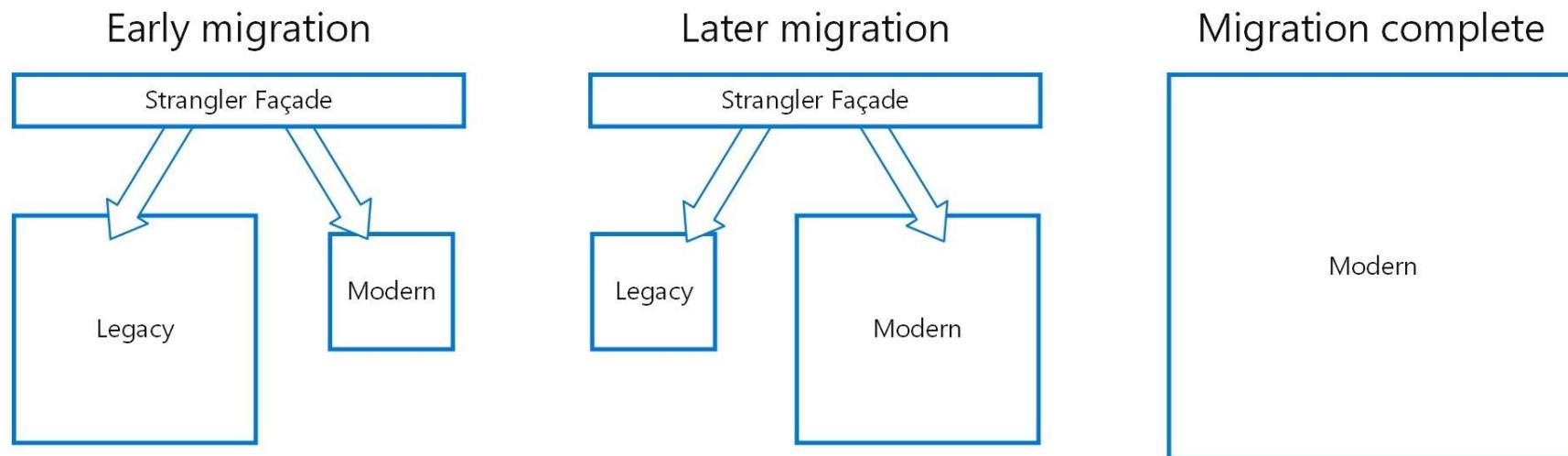
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Sidecar



e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Design und Implementierung
 - Strangler



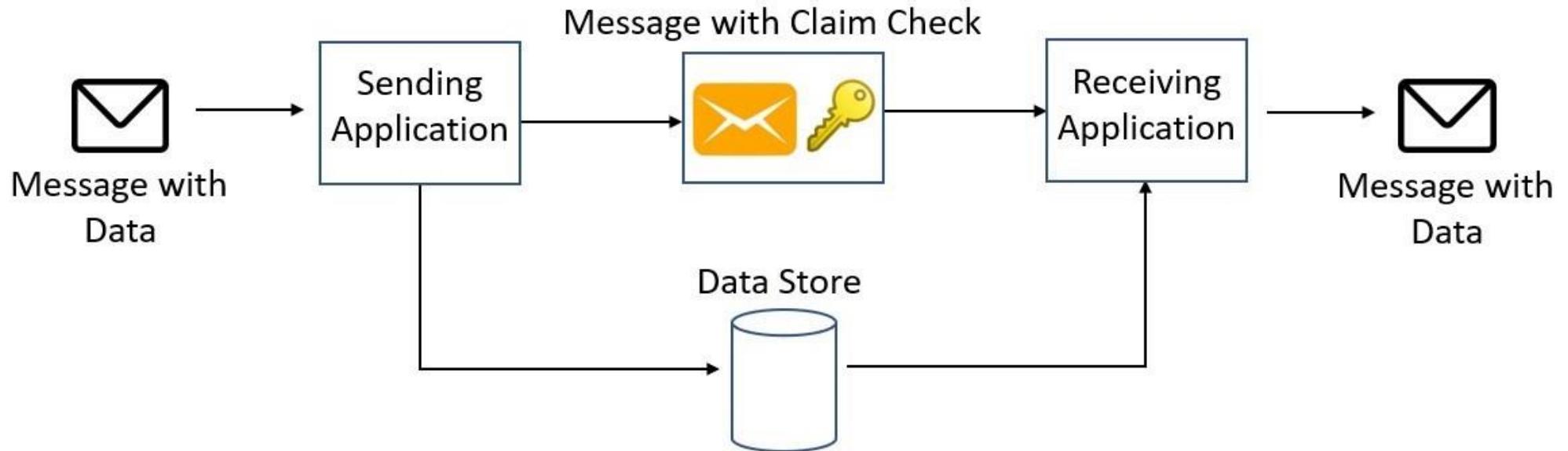
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Messaging

Pattern	Summary
Asynchronous Request-Reply	Synchrone Frontend Verarbeitung von asynchroner Backend Verarbeitung entkoppeln
Claim Check	Aufteilung von großen Nachrichten in Mitteilung und Inhalt
Choreography	Aufteilung eines Geschäftsprozesses in unabhängige Komponenten
Competing Consumers	Parallele Verarbeitung von Nachrichten
Pipes and Filters	Kennen wir schon
Priority Queue	Priorisieren von Anforderungen
Publisher-Subscriber	Asynchrone Ereignisse für mehrere interessierte Verbraucher bereitstellen
Queue-Based Load Leveling	Kennen wir schon
Scheduler Agent Supervisor	Koordination einer Reihe von Aktionen über einen Satz von verteilten Diensten oder anderer Ressourcen
Sequential Convoy	Gemeinsame Verarbeitung von einem Satz verwandter Nachrichten ohne die Verarbeitung anderer Nachrichten zu blockieren

e) Microsoft Cloud Architekturstile

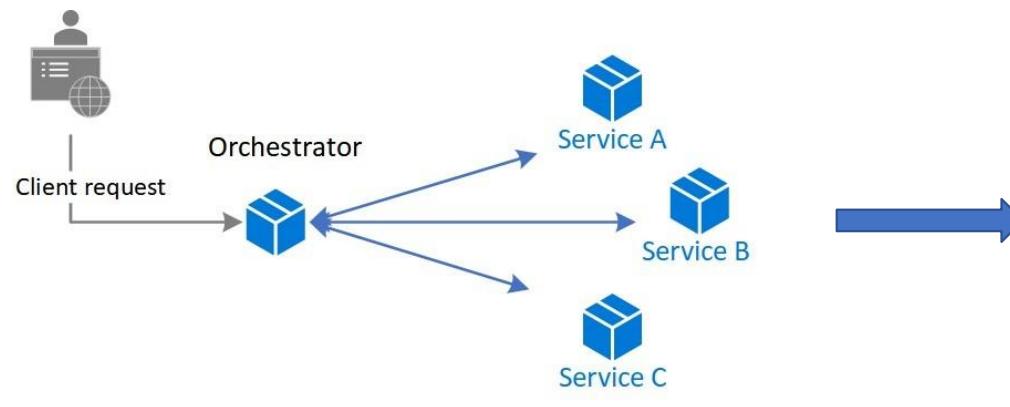
- Microsoft Cloud Design Pattern
 - Messaging
 - Claim Check



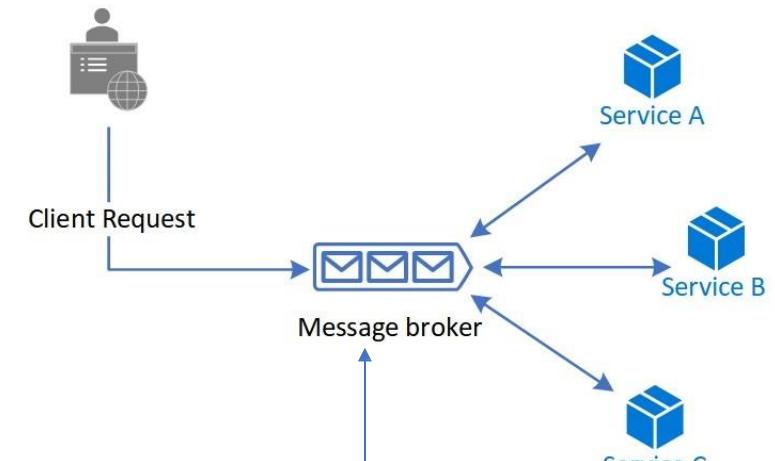
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Messaging
 - Choreography

Lose (asynchrone) Kopplung



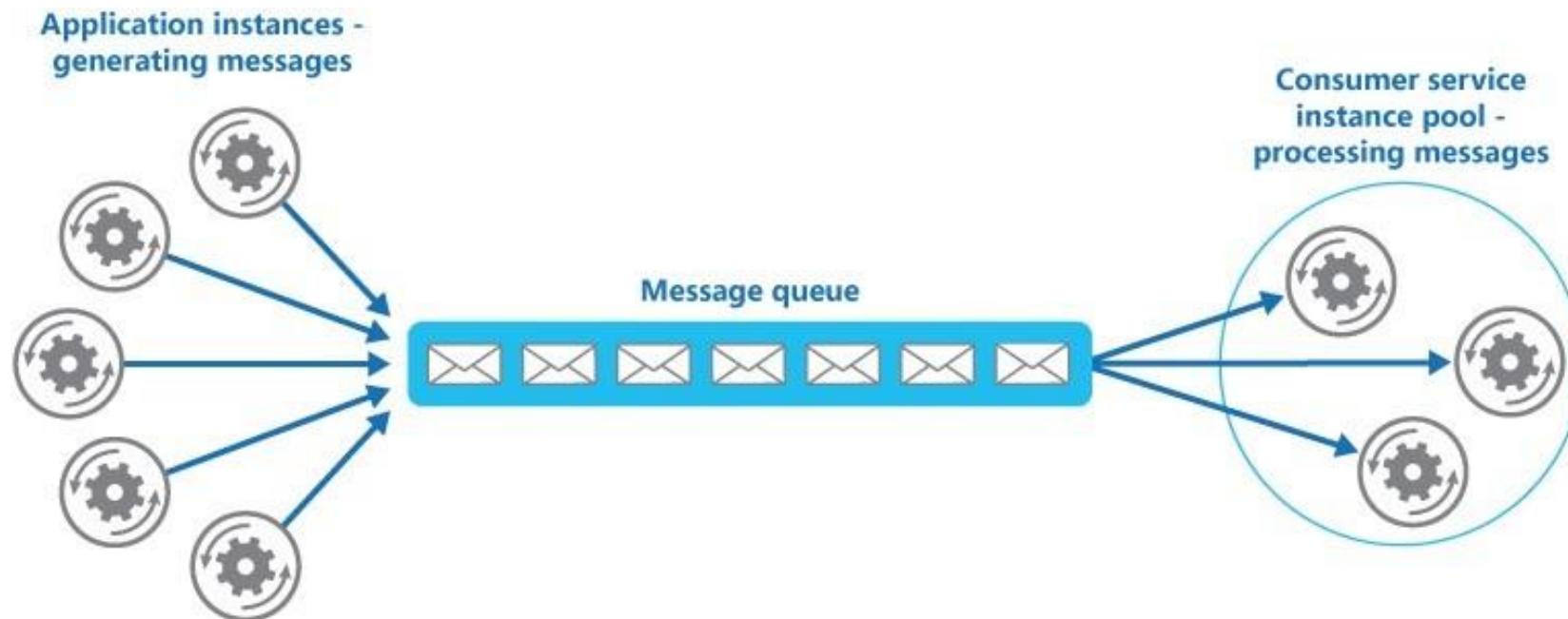
Enge Kopplung zwischen Orchestrator und Services



Publisher/Subscriber pattern

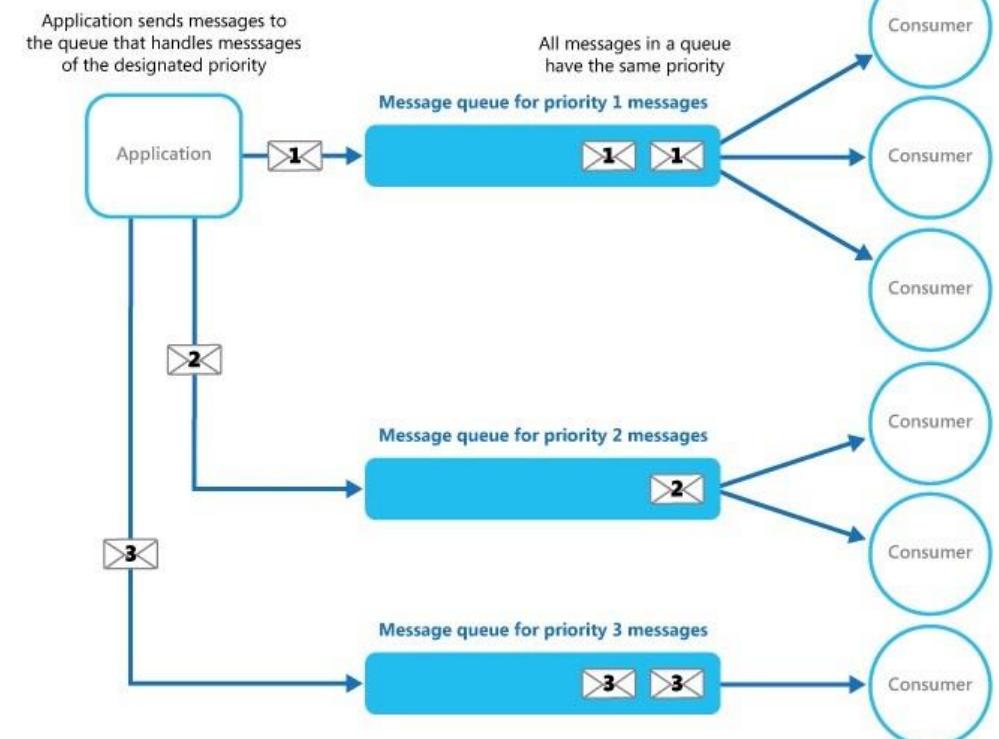
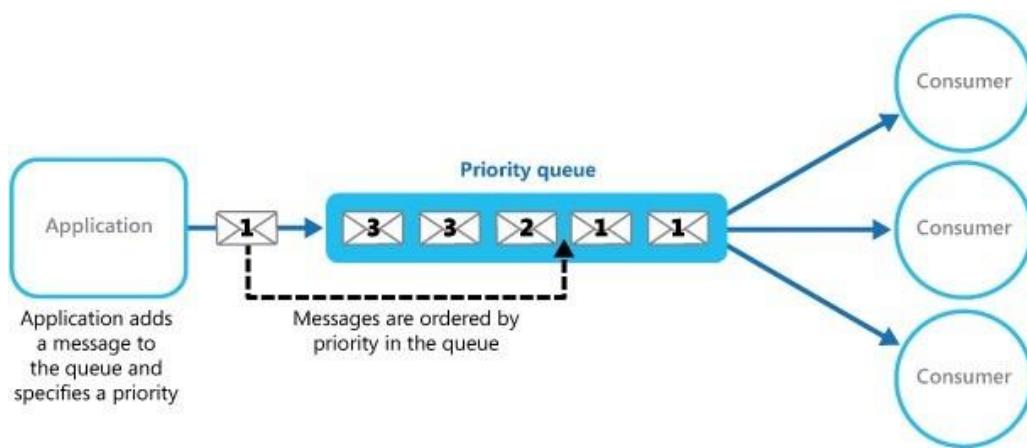
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Messaging
 - Competing consumers



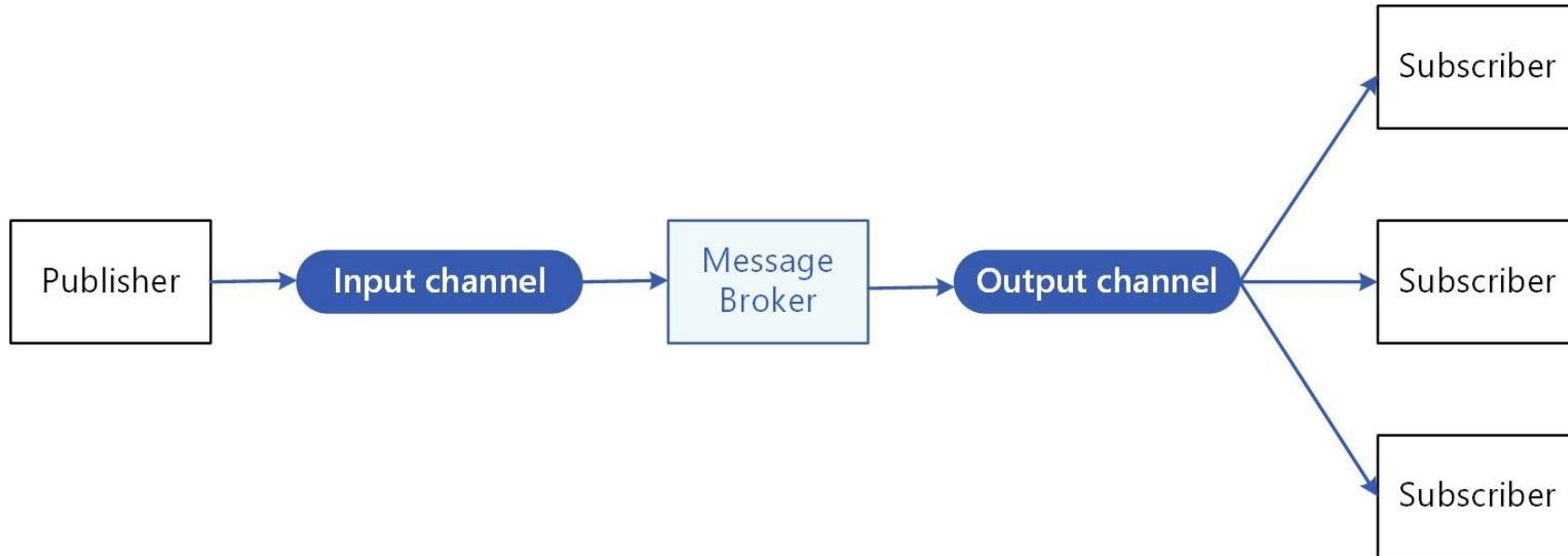
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Messaging
 - Priority queue



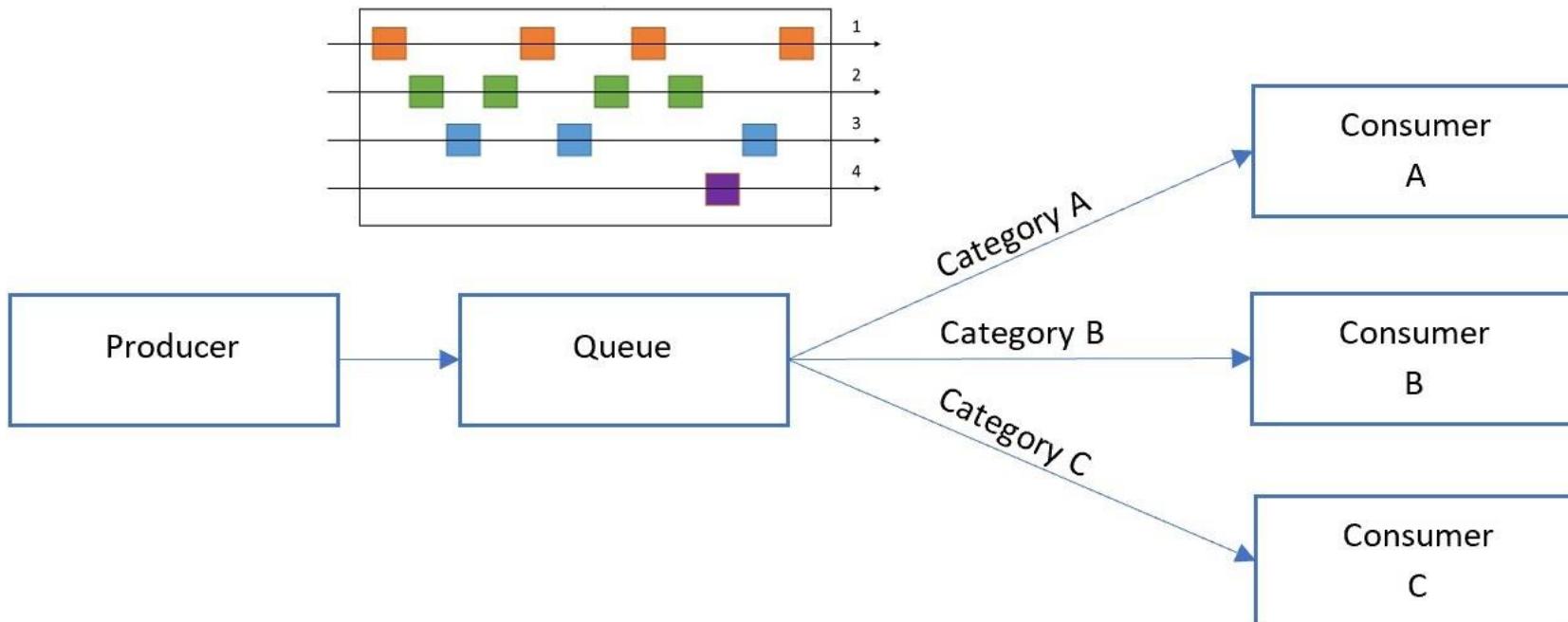
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Messaging
 - Publisher/Subscriber



e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Messaging
 - Sequential convoy



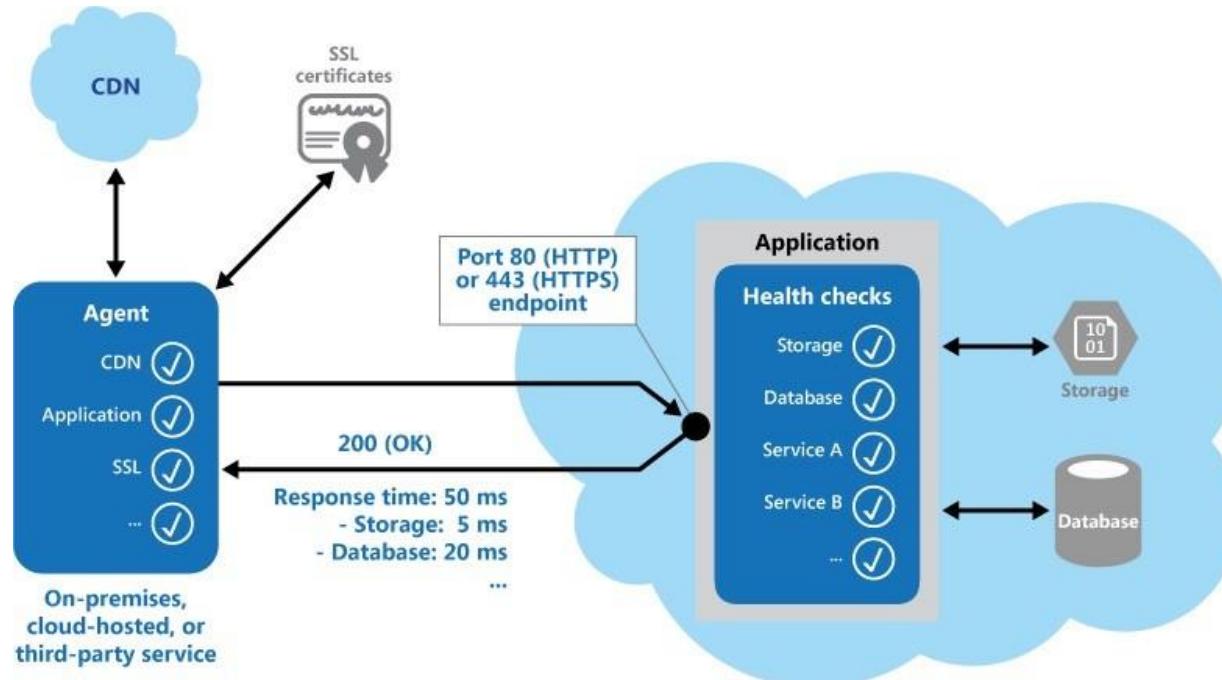
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Management und Monitoring

Pattern	Summary
Ambassador	Kennen wir schon
Anti-Corruption Layer	Kennen wir schon
External Configuration Store	Kennen wir schon
Gateway Aggregation	Kennen wir schon
Gateway Offloading	Kennen wir schon
Gateway Routing	Kennen wir schon
Health Endpoint Monitoring	Funktionsprüfungen einer Anwendung, auf die externe Tools in regelmäßigen Abständen zugreifen können
Sidecar	Kennen wir schon
Strangler	Kennen wir schon

e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Management und Monitoring
 - Health endpoint monitoring



e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Performance und Skalierbarkeit

Pattern	Summary
Cache-Aside	Kennen wir schon
Choreography	Kennen wir schon
CQRS	Kennen wir schon
Event Sourcing	Kennen wir schon
Deployment Stamps	Kennen wir schon
Geodes	Kennen wir schon
Index Table	Kennen wir schon
Materialized View	Kennen wir schon
Priority Queue	Kennen wir schon
Queue-Based Load Leveling	Kennen wir schon
Sharding	Kennen wir schon
Static Content Hosting	Kennen wir schon
Throttling	Kennen wir schon

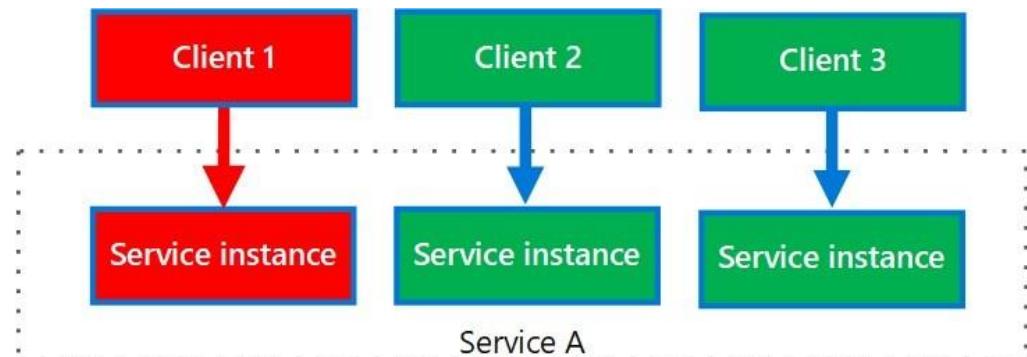
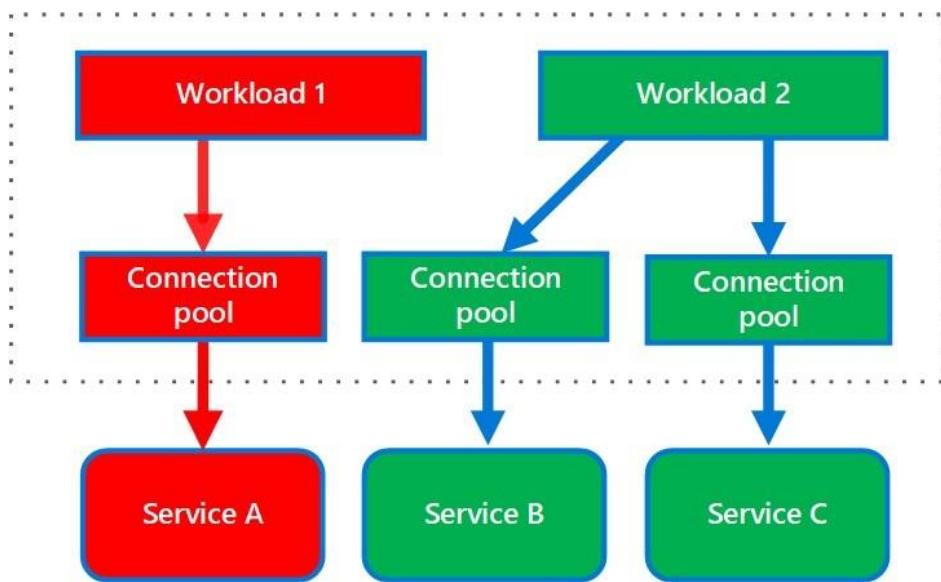
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Resilienz

Pattern	Summary
Bulkhead	Isolierte Failure-Units zur Vermeidung kaskadierender Fehler
Circuit Breaker	Behandlung von Fehlern beim Herstellen einer Verbindung zu einem Remote-Dienst oder einer Remote-Ressource
Compensating Transaction	Machen Sie die Arbeit rückgängig, die durch eine Reihe von Schritten ausgeführt wird, die zusammen eine eventuell konsistente Operation definieren.
Health Endpoint Monitoring	Kennen wir schon
Leader Election	Kennen wir schon
Queue-Based Load Leveling	Kennen wir schon
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed.
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources.

e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Resilienz
 - Bulkhead

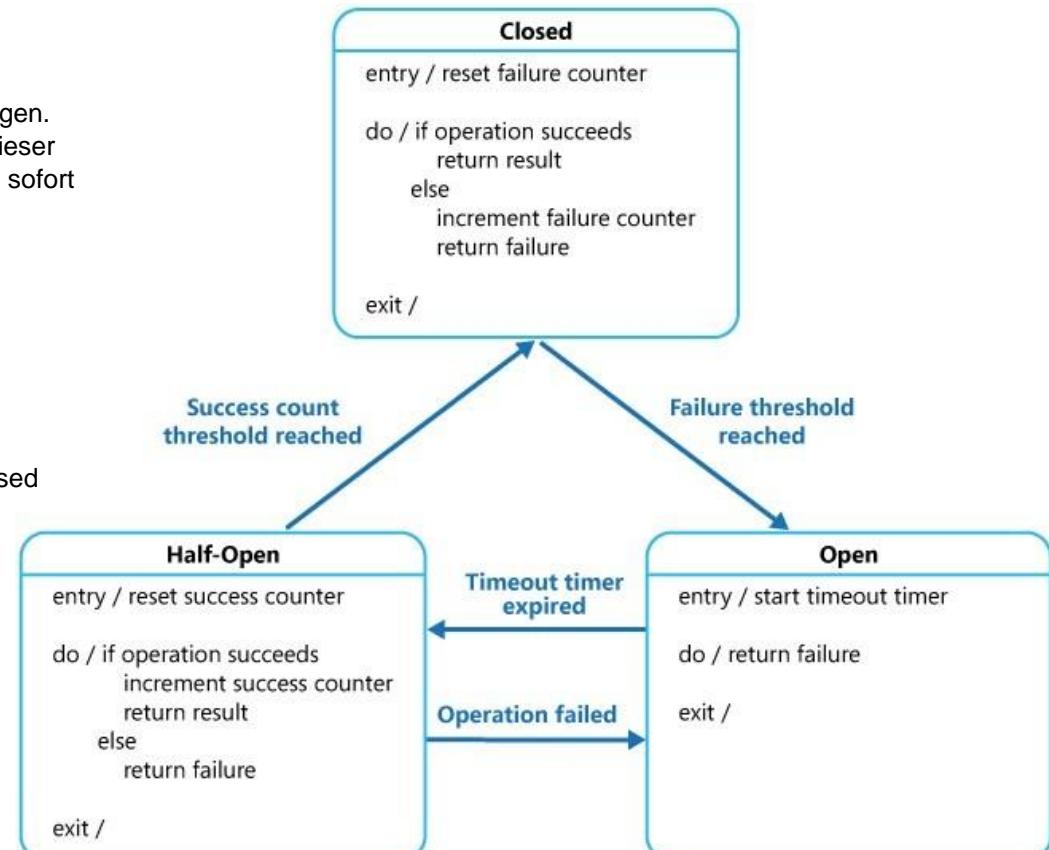


e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Resilienz
 - Circuit breaker

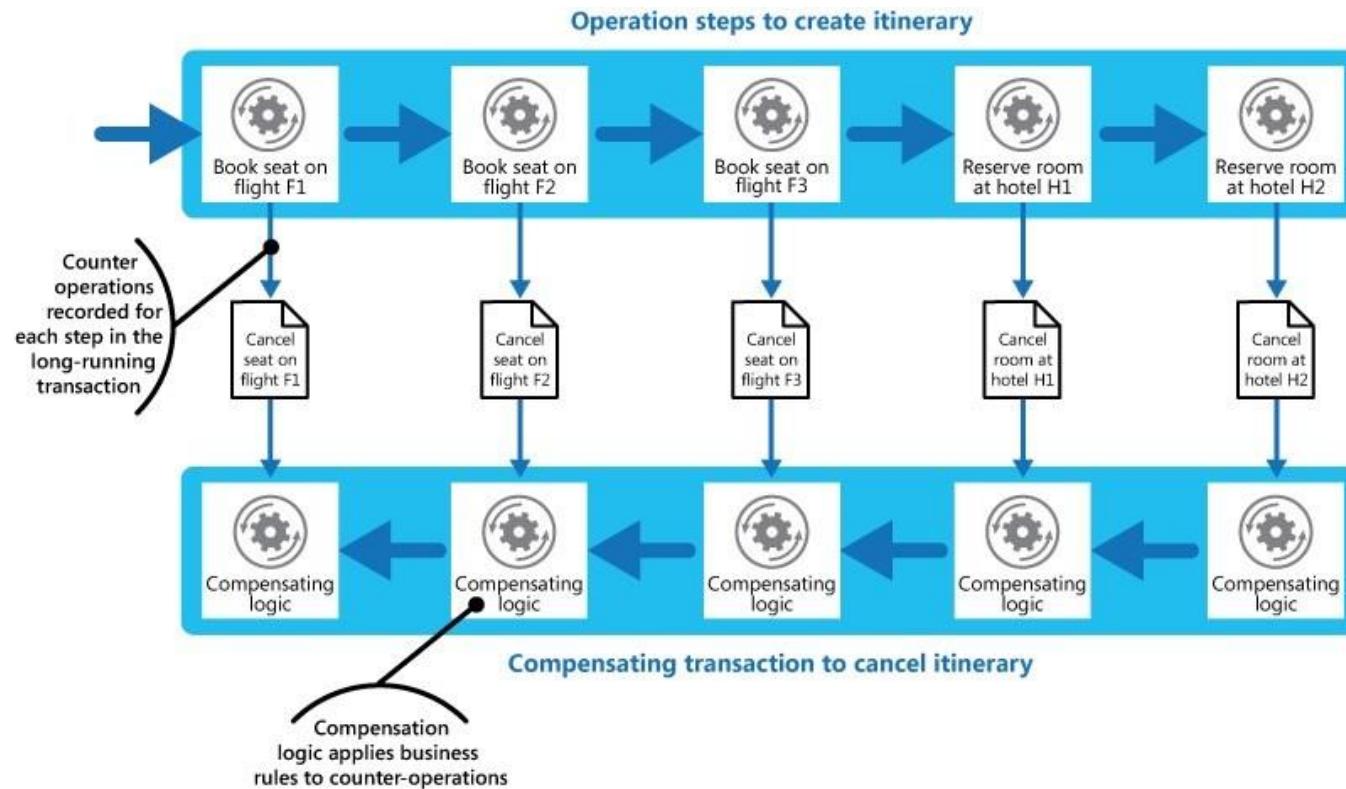
Ein Circuit Breaker fungiert als Proxy für Anfragen, die möglicherweise fehlgeschlagen. Der Proxy sollte die Anzahl der aufgetretenen Fehler überwachen und anhand dieser Informationen entscheiden, ob der Vorgang fortgesetzt werden soll, oder einfach sofort eine Ausnahme zurückgeben. Der Proxy verwaltet 3 Status :

- Closed
 - Anfrage kann durchgehen
 - Fehlgeschlagene Anfragen werden gezählt
 - Wird ein Grenzwert erreicht, wird ein Timeout Timer gestartet
 - Wird das Ende desTimeouts erreicht geht der Proxy in den Status
- Half-open
 - Eine begrenzte Anzahl von Anfragen wird durchgehen
 - Sind diese Anfragen erfolgreich fällt der Proxy zurück in den Status Closed
 - Wenn eine Anfrage fehlgeschlägt, geht der Proxy in den Status
- Open
 - Alle Anfragen werden sofort zurückgewiesen



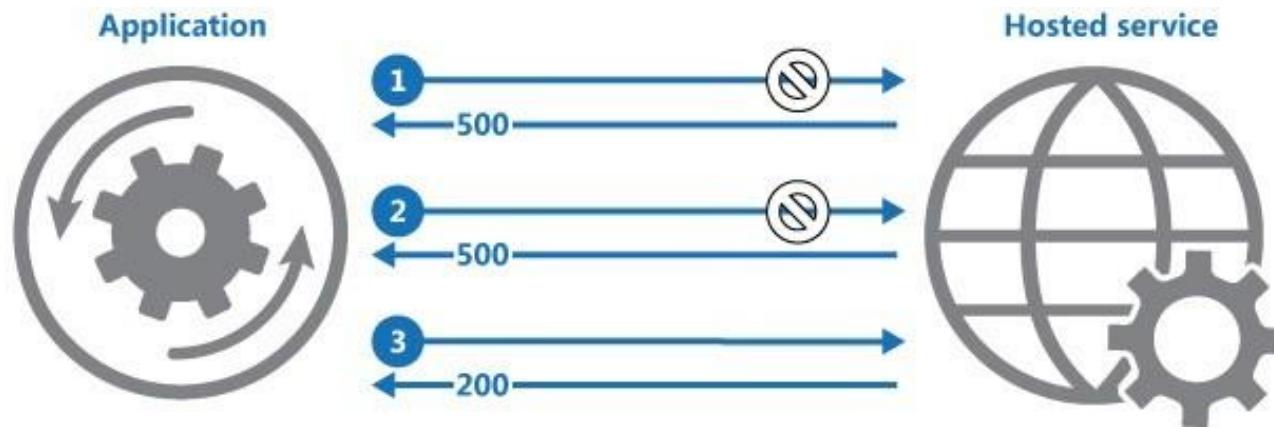
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Resilienz
 - Compensating Transaction



e) Microsoft Cloud Architekturstile

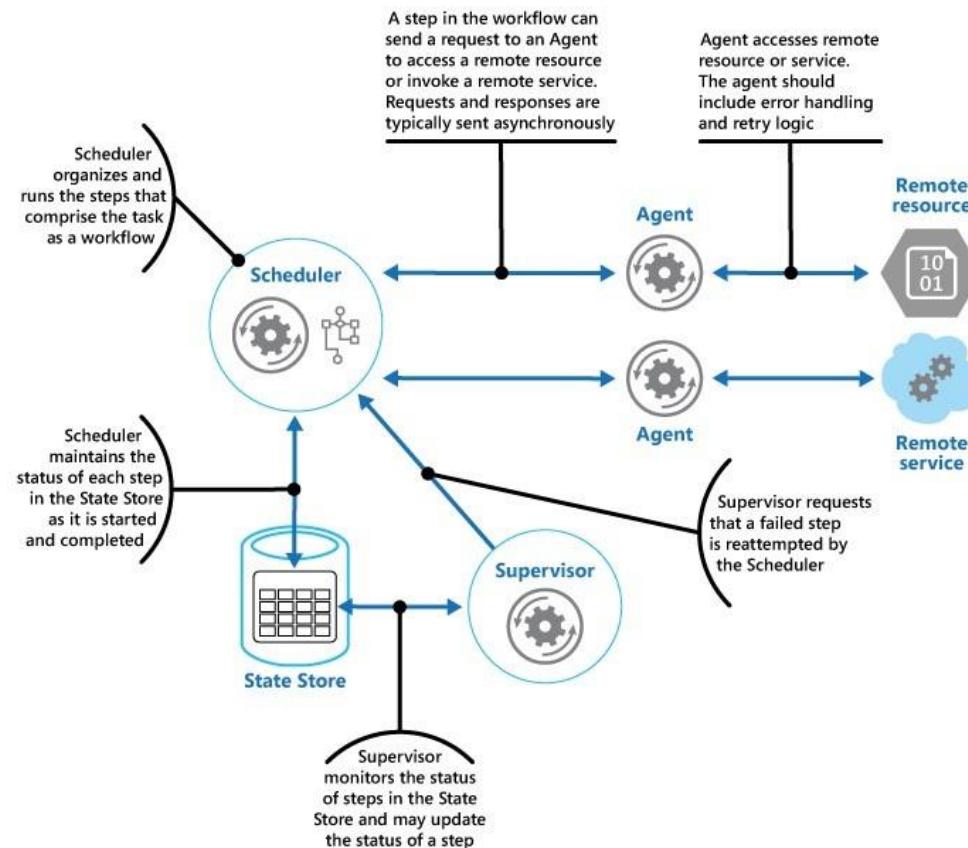
- Microsoft Cloud Design Pattern
 - Resilienz
 - Retry



- 1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
- 2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
- 3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Resilienz
 - Scheduler Agent Supervisor



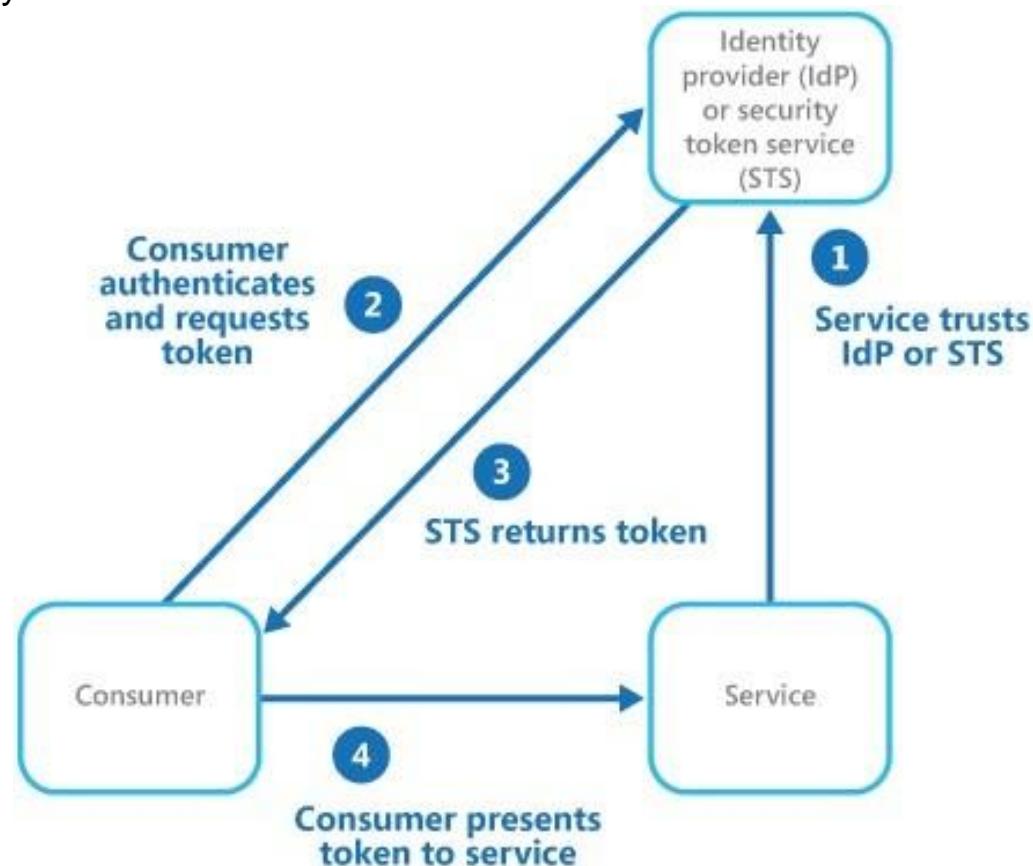
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Security

Pattern	Summary
Federated Identity	Authentifizierung an einen externen Identitätsanbieter delegieren
Gatekeeper	Dedizierter Broker zwischen Clients und Anwendungen zum Schützen von Anwendungen und Diensten
Valet Key	Kennen wir schon

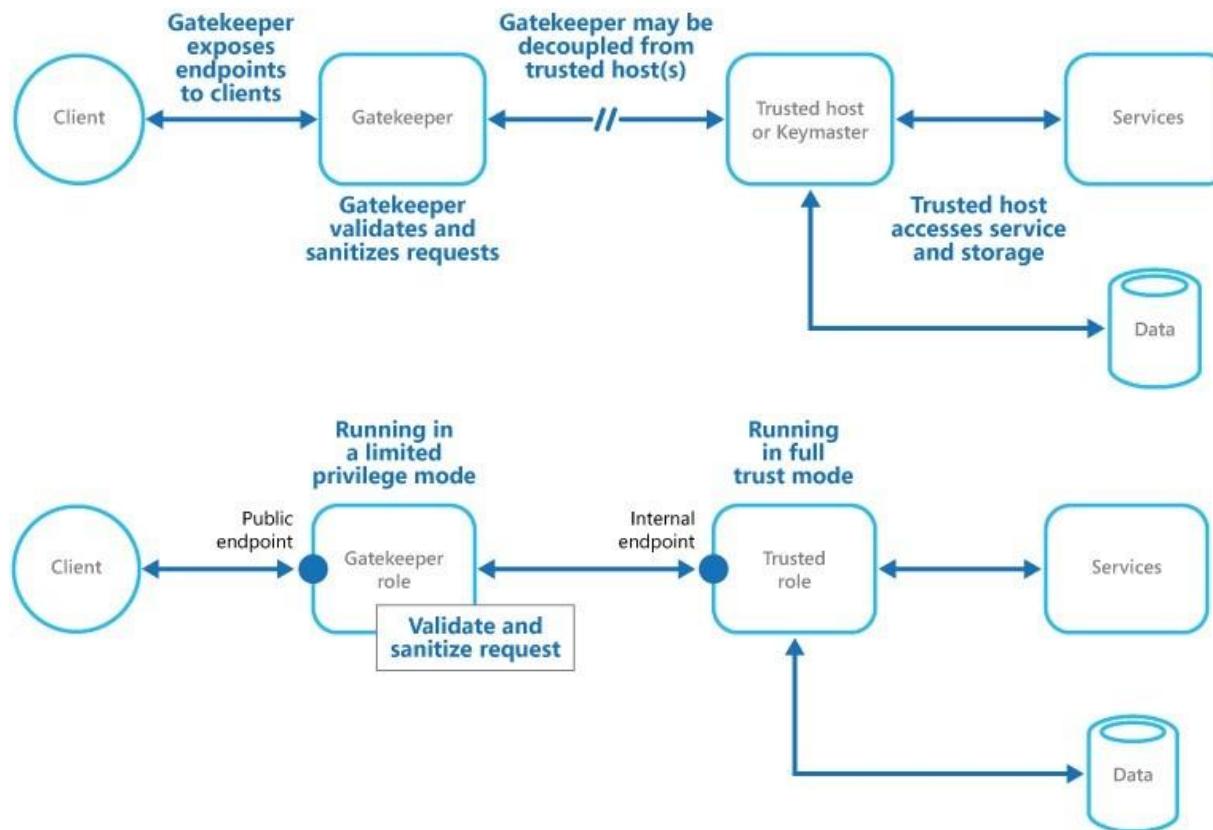
e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Security
 - Federated Identity



e) Microsoft Cloud Architekturstile

- Microsoft Cloud Design Pattern
 - Security
 - Gatekeeper



9) Microservices Architektur

- Viele Entwicklungsteams haben festgestellt, dass der Architekturstil von Microservices ein überlegener Ansatz gegenüber einer monolithischen Architektur ist
- Andere haben jedoch festgestellt, dass sie die Produktivität belasten können
- Microservices bringen wie jeder Architekturstil Kosten und Nutzen
- Um eine vernünftige Wahl zu treffen, müssen Sie diese verstehen und auf Ihren spezifischen Kontext anwenden

Wir betrachten

- Was sind Microservices
- Wie kann man Microservices anwenden

Microservices Architektur

- Was sagt Martin Fowler

„Das Microservice-Architekturmuster ist ein Ansatz zur Entwicklung einer einzelnen Anwendung als Suite **kleiner Dienste**, die jeweils in einem eigenen Prozess ausgeführt werden und **mit einfachen Mechanismen kommunizieren**, häufig einer HTTP-Ressourcen-API. Diese Services **basieren auf Geschäftsfunktionen** und können **unabhängig voneinander** von einem vollautomatisierten Deployment bereitgestellt werden. Es gibt ein Minimum an **zentraler Verwaltung** dieser Dienste, die in **verschiedenen Programmiersprachen** geschrieben sein und **unterschiedliche Datenspeichertechnologien** verwenden können. Während ihre Vorteile sie in den letzten Jahren sehr in Mode gebracht haben, sind sie **erhöhten Kosten bei der Verteilung**, einer **verminderten (eventuellen) Konsistenz** und der Notwendigkeit einer **ausgereiften Betriebsführung** verbunden.“.

Microservices Architektur

- Was sagt Martin Fowler

- Vorteile

- Starke Modulgrenzen: Microservices verstärken den modularen Aufbau, was besonders für größere Teams von Vorteil ist
 - Unabhängige Bereitstellung: Einfache Dienste sind einfacher bereitzustellen und da sie autonom sind, verursachen sie nicht so häufig Systemfehler, wenn sie schief gehen
 - Technologische Vielfalt: Mit Microservices können mehrere Sprachen, Entwicklungsframeworks und Datenspeichertechnologien gemischt werden

- Nachteile

- Verteilung: Verteilte Systeme sind schwieriger zu programmieren, da Remote Calls langsam sind und immer das Risiko eines Ausfalls besteht
 - Eventuelle Konsistenz: Die Aufrechterhaltung einer starken Konsistenz ist für ein verteiltes System äußerst schwierig und muss speziell verwaltet werden
 - Betriebskomplexität: Sie benötigen ein sehr gutes Betriebsteam, um die vielen Dienste zu verwalten und regelmäßig neu bereitzustellen

Microservices Architektur

- Software Services
 - Ein Softwareservice ist eine Softwarekomponente, auf die von Remote-computern über das Internet zugegriffen werden kann. Bei einer Eingabe erzeugt ein Service eine entsprechende Ausgabe (ohne Seiteneffekte)
 - Auf den Service wird über seine veröffentlichte Schnittstelle zugegriffen, alle Details der Serviceimplementierung werden ausgeblendet.
 - Services halten keinen internen Status. Statusinformationen werden entweder in einer Datenbank gespeichert oder vom Serviceanforderer verwaltet.
 - Wenn eine Serviceanforderung gestellt wird, können die Statusinformationen als Teil der Anforderung enthalten sein und die aktualisierten Statusinformationen werden als Teil des Serviceergebnisses zurückgegeben.
 - Da es keinen lokalen Status gibt, können Services dynamisch von einem virtuellen Server auf einen anderen umverteilt und auf mehrere Server repliziert werden...

Microservices Architektur

- Moderne Web Services
 - Nach verschiedenen Experimenten in den 1990er Jahren mit serviceorientiertem Computing entstand Anfang der 2000er Jahre die Idee der „großen“ Webdienste.
 - Diese basierten auf XML-basierten Protokollen und Standards wie SOAP für die Dienstinteraktion und WSDL für die Schnittstellenbeschreibung.
 - Die meisten Softwaredienste benötigen nicht die Allgemeingültigkeit, die dem Entwurf von Webdienstprotokollen innewohnt.
 - Infolgedessen verwenden moderne serviceorientierte Systeme einfachere, leichtere Service-Interaktionsprotokolle, die einen geringeren Overhead und folglich eine schnellere Ausführung aufweisen.

Microservices Architektur

- Microservices
 - Microservices sind kleine, zustandslose Dienste, die eine einzige Verantwortung tragen. Sie werden kombiniert, um Anwendungen zu erstellen.
 - Sie sind mit ihrem eigenen Datenbank- und UI-Verwaltungscode völlig unabhängig.
 - Softwareprodukte, die Microservices verwenden, verfügen über eine Microservices-Architektur.
 - Wenn Sie Cloud-basierte Softwareprodukte erstellen müssen, die anpassbar, skalierbar und belastbar sind, sollten sie auf der Grundlage einer Microservices-Architektur entworfen werden.

Microservices Architektur

- Ein Microservices Beispiel
 - Systemauthentifizierung
 - Benutzerregistrierung, bei der Benutzer Informationen zu ihrer Identität, Sicherheitsinformationen, Mobiltelefonnummer und E-Mail-Adresse angeben.
 - Authentifizierung mit UID / Passwort.
 - Zwei-Faktor-Authentifizierung mit Code, der an das Mobiltelefon gesendet wird.
 - Verwaltung der Benutzerinformationen, z. Passwort oder Handynummer ändern.
 - Vergessenes Passwort zurücksetzen.
 - Jede dieser Funktionen kann als separater Dienst implementiert werden, der eine zentrale gemeinsam genutzte Datenbank zum Speichern von Authentifizierungsinformationen verwendet.
 - Diese Merkmale sind jedoch zu groß, um Mikrodienste zu sein. Um die Mikrodienste zu identifizieren, die möglicherweise im Authentifizierungssystem verwendet werden, müssen Sie die grobkörnigen Merkmale in detailliertere Funktionen aufteilen.

Microservices Architektur

- Ein Microservices Beispiel

User registration

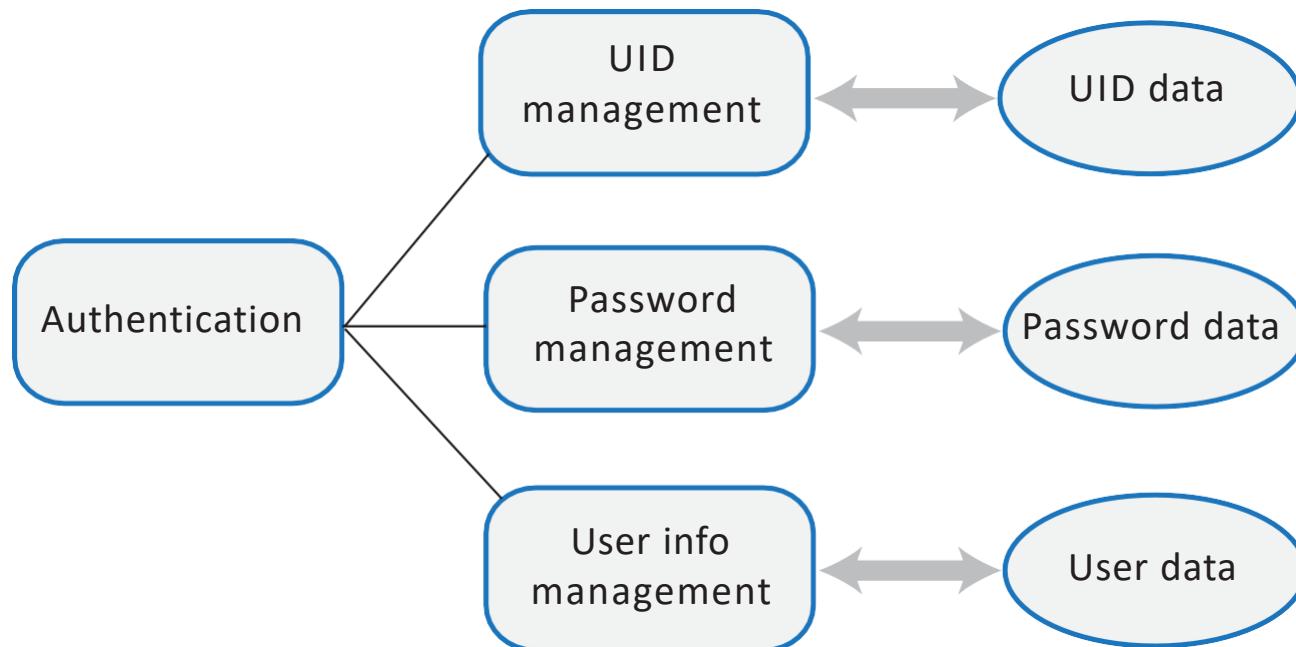
```
Setup new login id
Setup new password
Setup password recovery information
Setup two-factor authentication
Confirm registration
```

Authenticate using UID/password

```
Get login id
Get password
Check credentials
Confirm authentication
```

Microservices Architektur

- Ein Microservices Beispiel



Microservices Architektur

- Eigenschaften von Microservices

- In sich geschlossen

- Microservices haben keine externen Abhängigkeiten. Sie verwalten ihre eigenen Daten und implementieren ihre eigene Benutzeroberfläche.

- Leichtgewichtig

- Microservices kommunizieren mit Lightweight-Protokollen, sodass der Overhead für die Servicekommunikation gering ist.

- Implementierungsunabhängig

- Microservices können unter Verwendung verschiedener Programmiersprachen implementiert werden und können unterschiedliche Technologien (z. B. verschiedene Arten von Datenbanken) in ihrer Implementierung verwenden.

- Unabhängig bereitstellbar

- Jeder Microservice wird in einem eigenen Prozess ausgeführt und kann mithilfe automatisierter Systeme unabhängig voneinander bereitgestellt werden.

- Geschäftsorientiert

- Microservices sollten Geschäftsfähigkeiten und -anforderungen implementieren und nicht nur einen technischen Service bereitstellen.

Microservices Architektur

- Kommunikation von Microservices
 - Microservices kommunizieren durch den Austausch von Nachrichten.
 - Eine Nachricht, die zwischen Diensten gesendet wird, enthält Verwaltungsinformationen, einen Servicerequest (Dienstanforderung) und die Daten, die zur Bereitstellung des angeforderten Dienstes erforderlich sind.
 - Dienste geben eine Antwort auf Servicerequests zurück.
 - Ein Authentifizierungsdienst kann eine Nachricht an einen Anmeldedienst senden, die den vom Benutzer eingegebenen Namen enthält.
 - Die Antwort kann ein Token sein, das einem gültigen Benutzernamen zugeordnet ist, oder ein Fehler, der besagt, dass kein registrierter Benutzer vorhanden ist.

Microservices Architektur

- Eigenschaften von Microservices
 - Ein gut konzipierter Mikroservice sollte eine hohe Kohäsion und eine geringe Kopplung aufweisen.
 - Kohäsion ist ein Maß für die Anzahl der Beziehungen, die Teile einer Komponente zueinander haben. Hohe Kohäsion bedeutet, dass alle Teile, die zur Bereitstellung der Funktionalität der Komponente benötigt werden, in der Komponente enthalten sind.
 - Die Kopplung ist ein Maß für die Anzahl der Beziehungen, die eine Komponente zu anderen Komponenten im System hat. Geringe Kopplung bedeutet, dass Komponenten nicht viele Beziehungen zu anderen Komponenten haben.
 - Jeder Microservice sollte eine einzige Verantwortung haben, d. H. er sollte nur eines tun und das sollte gut funktionieren.
 - Es ist jedoch schwierig, „nur eine Sache“ so zu definieren, dass sie für alle Dienste gilt.
 - Verantwortung bedeutet nicht immer eine einzige funktionale Aktivität.

Microservices Architektur

- Am Microservices Beispiel

Passwort Management

User functions	Supporting functions
Create password	Check password validity
Change password	Delete password
Check password	Backup password database
Recover password	Recover password database
	Check database integrity
	Repair password DB

Support Code

Microservice X	
Service functionality	
Message management	Failure management
UI implementation	Data consistency management

Microservices Architektur

- Die Architektur
 - Eine Microservices-Architektur ist ein Architekturstil - eine bewährte Methode zur Implementierung einer logischen Softwarearchitektur.
 - Dieser Architekturstil befasst sich mit oder löst zwei Probleme(n) bei monolithischen Anwendungen
 - Das gesamte System muss neu erstellt, erneut getestet und erneut bereitgestellt werden, wenn Änderungen vorgenommen werden. Dies kann ein langwieriger Prozess sein, da Änderungen an einem Teil des Systems andere Komponenten nachteilig beeinflussen können.
 - Wenn die Anforderungen an das System steigen, muss das gesamte System skaliert werden, auch wenn die Anforderungen auf eine kleine Anzahl von Systemkomponenten beschränkt sind, die die meist benutzten Systemfunktionen implementieren.

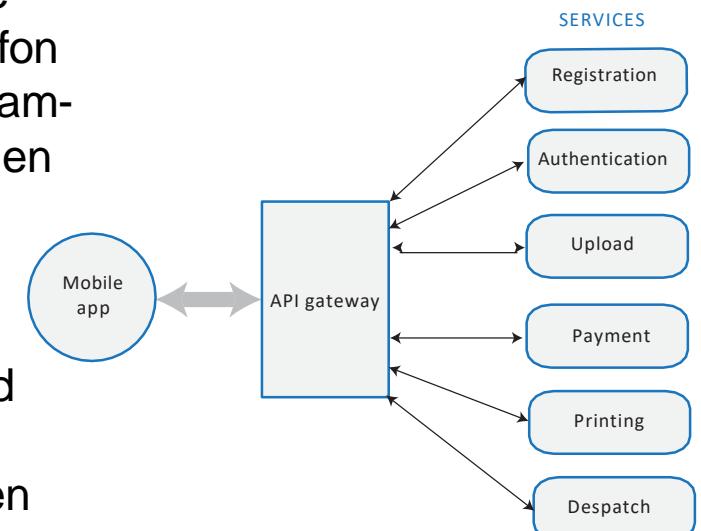
Microservices Architektur

- Vorteile
 - Microservices sind in sich geschlossen und werden in separaten Prozessen ausgeführt.
 - In Cloud-basierten Systemen kann jeder Mikrodienst in einem eigenen Container bereitgestellt werden. Dies bedeutet, dass ein Mikrodienst gestoppt und neu gestartet werden kann, ohne andere Teile des Systems zu beeinträchtigen.
 - Wenn die Nachfrage nach einem Service steigt, können Service-Replikate schnell erstellt und bereitgestellt werden. Für diese ist kein leistungsfähigerer Server erforderlich, sodass das „scale-out“ in der Regel viel billiger ist als das „scale-up“.

Microservices Architektur

- Beispiel (Fotodruckservice)

- Stellen Sie sich vor, Sie entwickeln einen Fotodruckservice für mobile Geräte. Benutzer können Fotos von ihrem Telefon auf einen Server hochladen oder Fotos aus ihrem Instagram-Konto angeben, die gedruckt werden sollen. Drucke können in verschiedenen Größen und auf verschiedenen Medien erstellt werden.
- Benutzer können die Druckgröße und das Druckmedium auswählen. Zum Beispiel können sie beschließen, ein Bild auf einen Becher oder ein T-Shirt zu drucken. Die Drucke oder andere Medien werden vorbereitet und dann zu Ihnen nach Hause geschickt. Sie bezahlen Drucke entweder über einen Zahlungsdienst wie Android oder Apple Pay oder durch die Registrierung einer Kreditkarte beim Druckdienstleister.



Microservices Architektur

- Schlüsselfragen bei der Microservices Architektur



Microservices Architektur

- Hinweise zur Aufteilung von Microservices
 - Balance zwischen feinkörniger Funktionalität und Systemleistung
 - Single-function Services bedeuten, dass Änderungen auf weniger Services beschränkt sind, jedoch Service-Kommunikation erfordern, um Benutzerfunktionen zu implementieren. Dies verlangsamt ein System, da jeder Dienst von anderen Diensten gesendete Nachrichten bündeln und entbündeln muss.
 - Befolgen Sie das „Prinzip der gemeinsamen Geschlossenheit“
 - Elemente eines Systems, die wahrscheinlich zur gleichen Zeit geändert werden, sollten sich innerhalb desselben Services befinden. Die meisten neuen und geänderten Anforderungen sollten daher nur einen einzigen Service betreffen.
 - Verknüpfen Sie Services mit Geschäftsfunktionen
 - Eine Geschäftsfähigkeit ist ein diskreter Bereich der Geschäftsfunktionalität, für den eine Einzelperson oder eine Gruppe verantwortlich ist. Sie sollten die Services identifizieren, die zur Unterstützung der einzelnen Geschäftsfunktionen erforderlich sind.
 - Entwerfen Sie Services so, dass sie nur auf die Daten zugreifen können, die sie benötigen
 - Wenn es eine Überlappung zwischen den von verschiedenen Services verwendeten Daten gibt, benötigen Sie einen Mechanismus, um Datenänderungen mit denselben Daten an alle Services weiterzugeben.

Microservices Architektur

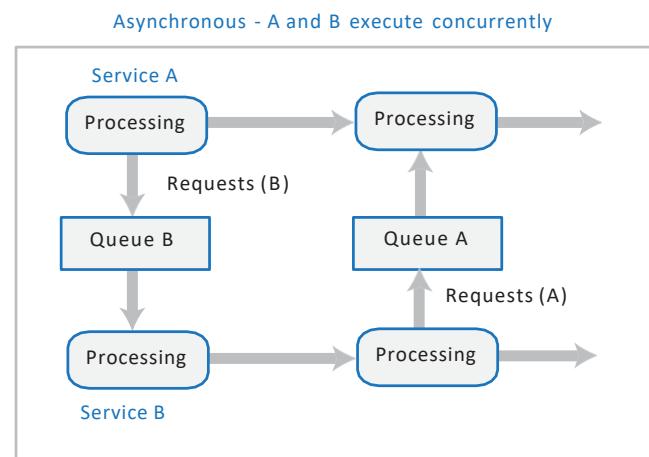
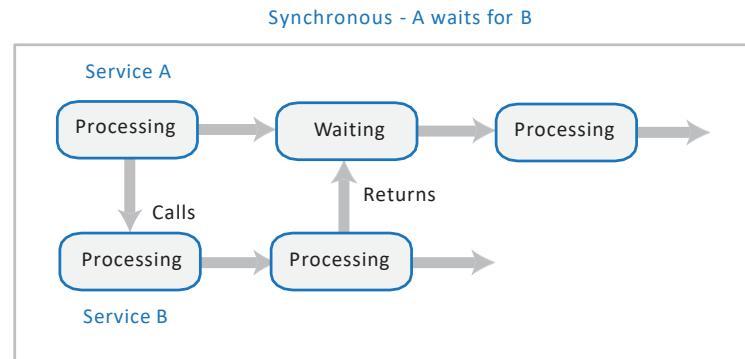
- Hinweise zur Service Kommunikation
 - Services kommunizieren durch den Austausch von Nachrichten, die Informationen über den Absender der Nachricht sowie die Daten enthalten, die als Eingabe für oder Ausgabe von der Anforderung dienen.
 - Beim Entwurf einer Microservices-Architektur, muss ein Standard für die Kommunikation festlegt werden, dem alle Microservices folgen sollten. Einige der wichtigsten Entscheidungen sind
 - Synchrone oder asynchrone Service-Interaktion ?
 - Direkte Kommunikation oder über Message Broker Middleware ?
 - Welches Protokoll sollte für Nachrichten verwendet werden, die zwischen Services ausgetauscht werden ?

Microservices Architektur

- Hinweise zur Service Kommunikation
 - Synchron
 - In einer synchronen Interaktion gibt Service A eine Anforderung an Service B ab. Service A unterbricht dann die Verarbeitung, während B die Anforderung verarbeitet.
 - Service A wartet, bis Service B die erforderlichen Informationen zurückgegeben hat, bevor die Ausführung fortgesetzt wird.
 - Asynchron
 - In einer asynchronen Interaktion gibt Service A die Anforderung aus, die zur Verarbeitung durch Service B in die Warteschlange gestellt wird. A setzt dann die Verarbeitung fort, ohne darauf zu warten, dass B seine Berechnungen beendet.
 - Einige Zeit später schließt Service B die frühere Anforderung von Service A ab und stellt das von A abzurufende Ergebnis in die Warteschlange.
 - Service A muss daher seine Warteschlange regelmäßig überprüfen, um festzustellen, ob ein Ergebnis verfügbar ist.

Microservices Architektur

- Hinweise zur Service Kommunikation



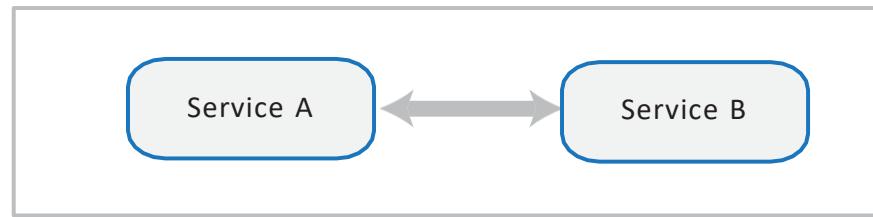
Microservices Architektur

- Hinweise zur Service Kommunikation
 - Direkt
 - Für die Kommunikation mit einem direkten Dienst müssen die interagierenden Dienste die Adresse des anderen kennen.
 - Die Dienste interagieren, indem sie Anforderungen direkt an diese Adressen senden.
 - Indirekt über einen Message Broker
 - Bei der indirekten Kommunikation wird der erforderliche Service benannt und diese Anforderung an einen Message Broker (manchmal auch als Message Bus bezeichnet) gesendet.
 - Der Message Broker ist dann dafür verantwortlich, den Service zu finden, der die Serviceanforderung erfüllen kann.

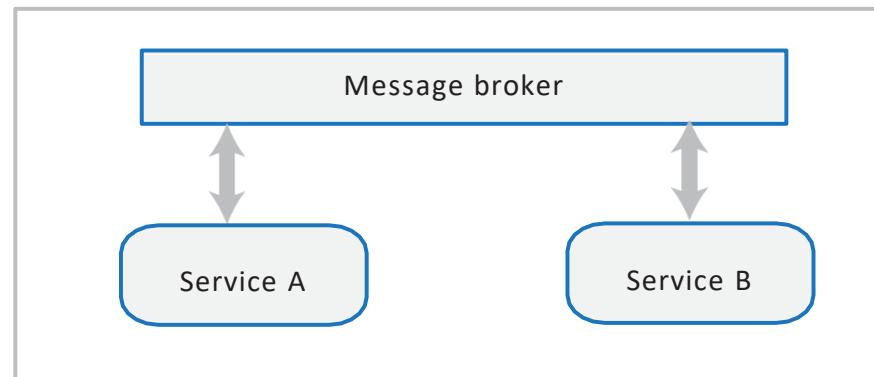
Microservices Architektur

- Hinweise zur Service Kommunikation

Direct communication - A and B send messages to each other



Indirect communication - A and B communicate through a message broker



Microservices Architektur

- Hinweise zum Datendesign
 - Daten innerhalb jedes Services sollten so wenig wie möglich mit anderen Services geteilt sondern von anderen isoliert werden
 - Sollte die gemeinsame Nutzung von Daten unvermeidbar sein, sollten Microservices so gestaltet sein, dass die meisten gemeinsamen Nutzungen schreibgeschützt sein und nur eine minimale Anzahl von Services für Datenaktualisierungen verantwortlich ist.
 - Wenn Services in einem System repliziert werden, muss das einen Mechanismus einschließen, mit dem die Datenbankkopien der replizierten Services konsistent bleiben.

Microservices Architektur

- Behandlung von Inkonsistenzen

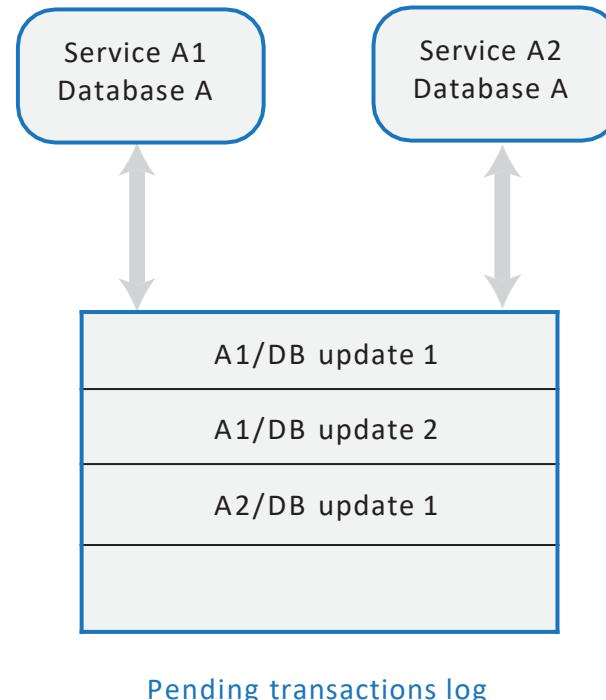
- Eine ACID-Transaktion bündelt eine Reihe von Datenaktualisierungen in einer einzigen Einheit, sodass entweder alle Aktualisierungen abgeschlossen sind oder keine. ACID-Transaktionen sind in einer Microservices-Architektur unpraktisch.
- Die von verschiedenen Microservices oder Microservice-Replikaten verwendeten Datenbanken müssen nicht immer vollständig konsistent sein.
- Abhängige Dateninkonsistenz
 - Die Aktionen oder Fehler eines Dienstes können dazu führen, dass die von einem anderen Service verwalteten Daten inkonsistent werden.
- Replikatinkonsistenz
 - Es gibt mehrere Replikate desselben Services, die gleichzeitig ausgeführt werden. Diese haben alle ihre eigene Datenbankkopie und jede aktualisiert ihre eigene Kopie der Servicedaten. Sie benötigen eine Möglichkeit, diese Datenbanken "schließlich konsistent" zu machen, damit alle Replikate mit denselben Daten arbeiten.

Microservices Architektur

- Eventuelle Konsistenz
 - Eine eventuelle Konsistenz ist eine Situation, in der das System garantiert, dass die Datenbanken schließlich konsistent werden.
 - Eine eventuelle Konsistenz kann implementiert, indem ein Transaktionsprotokoll verwaltet wird.
 - Wenn eine Datenbankänderung vorgenommen wird, wird dies in einem Protokoll "Ausstehende Aktualisierungen" aufgezeichnet.
 - Andere Serviceinstanzen überprüfen dieses Protokoll, aktualisieren ihre eigene Datenbank und geben an, dass sie die Änderung vorgenommen haben.

Microservices Architektur

- Eventuelle Konsistenz

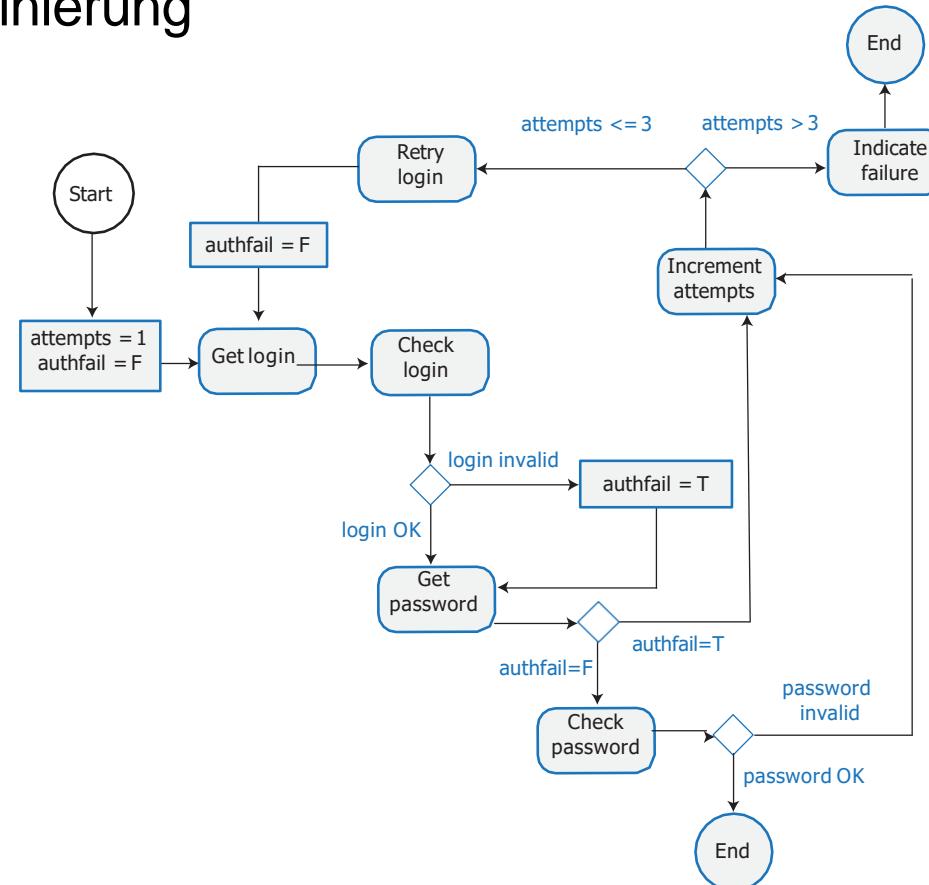


Microservices Architektur

- Service Koordinierung
 - Die meisten User Sessions umfassen eine Reihe von Interaktionen, bei denen Vorgänge in einer bestimmten Reihenfolge ausgeführt werden müssen.
 - Dies wird als Workflow bezeichnet.
 - Ein Authentifizierungsworkflow für die UID- / Kennwortauthentifizierung zeigt die Schritte zur Authentifizierung eines Benutzers.
 - In diesem Beispiel sind dem Benutzer 3 Anmeldeversuche gestattet, bevor das System anzeigt, dass die Anmeldung fehlgeschlagen ist.

Microservices Architektur

- Service Koordinierung



Microservices Architektur

- Service Orchestration vs. Service Choreography

- Orchestration

- In einem Orchester haben wir einen zentralen Mann namens Dirigent ("Orchestrator"). Er ist dafür verantwortlich, alle Musikinstrumente zu „dirigieren“.
 - In ähnlicher Weise verarbeitet der Orchestrator (zentraler Controller) in der Microservice-Orchestrierung alle Microservice-Interaktionen. Er überträgt die Ereignisse und reagiert darauf.
 - Die Microservice-Orchestrierung ähnelt eher einem zentralisierten Service. Es ruft einen Dienst auf und wartet auf die Antwort, bevor es den nächsten Dienst anruft.
Dies folgt einem Paradigma vom Typ „Request / Response“.



322

Microservices Architektur

- Service Orchestration vs. Service Choreography
 - Orchestration
 - Vorteile
 - Einfach zu warten und zu verwalten, da der Geschäftsprozess im Orchestrator konzentriert ist.
 - Bei der synchronen Verarbeitung wird der Ablauf der Anwendung effizient koordiniert.
 - Einschränkungen
 - Abhängigkeiten aufgrund gekoppelter Services. Wenn beispielsweise Service A nicht verfügbar ist, wird Service B niemals aufgerufen.
 - Der Orchestrator trägt die alleinige Verantwortung. Wenn er ausfällt, wird die Verarbeitung gestoppt und die Anwendung schlägt fehl.
 - Verlust an Sichtbarkeit, da die Geschäftsprozesse nicht verfolgt werden.



Microservices Architektur

- Service Orchestration vs. Service Choreography

- Choreographie

- Dies ist der andere Weg, um eine Microservice-Interaktion zu erreichen.
 - Abhängigkeiten in einer Microservice-Architektur werden vermieden, damit jeder Service unabhängig arbeiten kann. Die Choreografie löst dieses Problem der Abhängigkeit, die größte Herausforderung beim Orchestrierungsansatz.
 - Man kann sich Microservice Choreography als einen Tanz vorstellen (wie im Bild gezeigt). Darin führt jeder Einzelne Schritte unabhängig aus.
 - In ähnlicher Weise führt in der Microservice-Choreografie jeder Microservice seine Aktionen unabhängig aus. Es sind keine Anweisungen erforderlich. Es ist wie die dezentrale Art der Übertragung von Daten, die als Ereignisse bezeichnet werden.

Die Dienste, die an diesen Ereignissen interessiert sind, werden sie verwenden und Aktionen ausführen. Dies wird auch als reaktive Architektur bezeichnet.

Die Dienste wissen, worauf sie reagieren müssen und wie sie vorgehen müssen, was eher einem asynchronen Ansatz ähnelt.



324

Microservices Architektur

- Service Orchestration vs. Service Choreography

- Choreographie

- Vorteile
 - Ermöglicht eine schnelle Verarbeitung. Da keine Abhängigkeit von einer zentralen Steuerung.
 - Einfach zu erweitern und zu aktualisieren. Services können jederzeit aus dem Eventstream entfernt oder hinzugefügt werden.
 - Da die Steuerung verteilt ist, gibt es keinen Single Point of Failure.
 - Funktioniert gut mit agilen Modellen, da Teams an bestimmten Services und nicht an der gesamten Anwendung arbeiten.
 - Es können bekannte Muster verwendet werden wie z.B. Event Sourcing oder CQRS (Command-Query Responsibility Segregation)
 - Einschränkungen
 - Komplexität. Jeder Dienst ist unabhängig, um die Logik zu identifizieren und basierend auf der vom Eventstream bereitgestellten Logik zu reagieren.
 - Der Geschäftsprozess ist verteilt, was es schwierig macht, den Gesamtprozess zu warten und zu verwalten.
 - Eine Änderung der Denkweise zum asynchronen Ansatz ist notwendig



Microservices Architektur

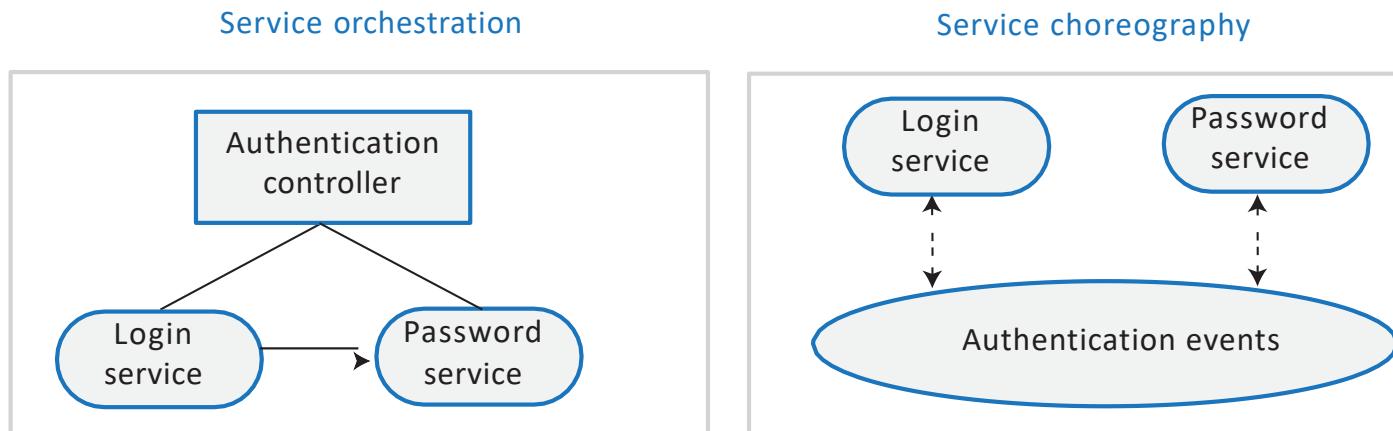
- Service Orchestration vs. Service Choreography
 - In den meisten Fällen funktionieren diese Ansätze in der Architektur nicht gut. Was ist die Lösung in diesen Anwendungsfällen ?
 - Der **hybride Ansatz** kann das Problem lösen. Zum Beispiel eine Mischung aus synchronen und asynchronen Aktivitätsblöcken.
 - Hybrid ist die Kombination aus Orchestrationsansatz und Choreografie. Bei diesem Ansatz wird Orchestrierung innerhalb der Services und Choreografie zwischen den Services verwendet.
 - Ein hybrider Ansatz hat natürlich auch Vor- und Nachteile
 - Vorteile
 - Der Gesamtfluss ist verteilt. Jeder Dienst enthält seine Ablauflogik.
 - Serviceleistungen sind entkoppelt (bis zu einem gewissen Grad).
 - Einschränkungen
 - Der Koordinator ist mit den Services gekoppelt.
 - Wenn der Koordinator ausfällt, wirkt sich dies auf das gesamte System aus.



326

Microservices Architektur

- Service Orchestration vs. Service Choreography



Microservices Architektur

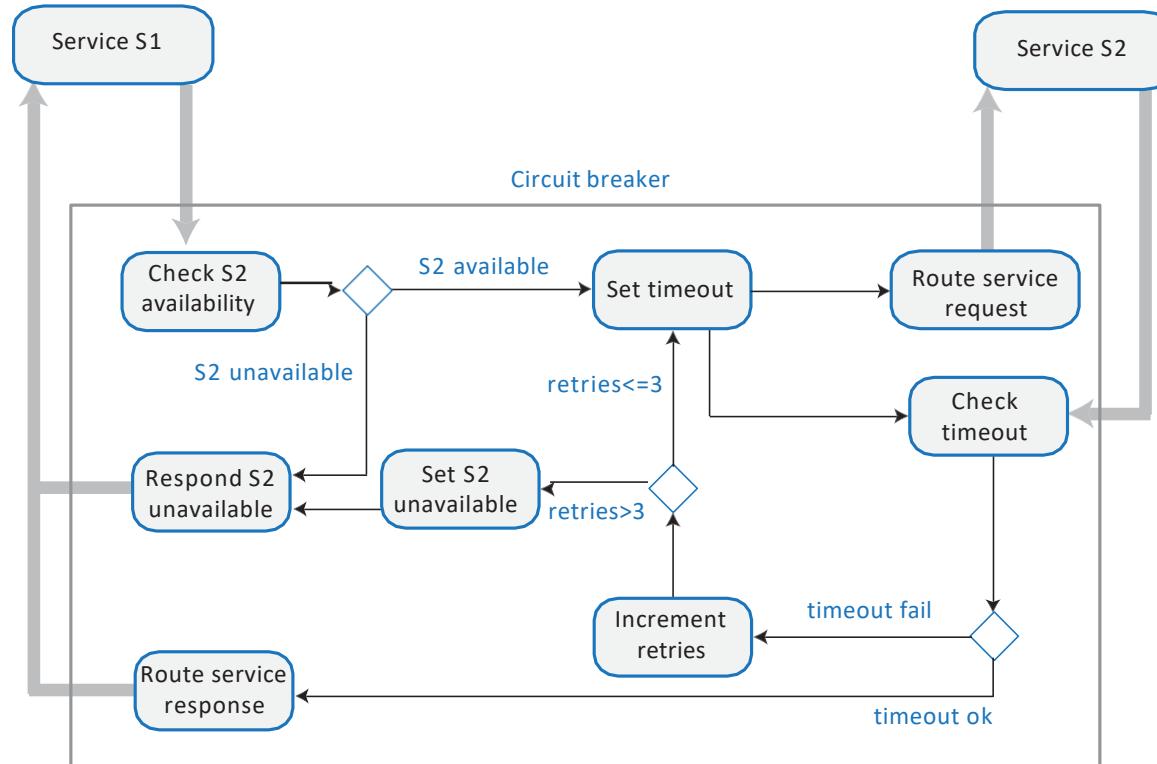
- Typische Fehler in einem Microservices System
 - Interner Servicefehler
 - Dies sind Bedingungen, die vom Service erkannt werden und in einer Fehlermeldung an den Client gemeldet werden können. Ein Beispiel für diese Art von Fehler ist ein Service, der eine URL als Eingabe verwendet und feststellt, dass es sich um einen ungültigen Link handelt.
 - Externer Servicefehler
 - Diese Fehler haben eine externe Ursache, die sich auf die Verfügbarkeit eines Services auswirkt. Ein Fehler kann dazu führen, dass der Service nicht mehr reagiert und Maßnahmen ergriffen werden müssen, um den Service neu zu starten.
 - Serviceperformance Fehler
 - Die Leistung des Services verschlechtert sich auf ein nicht akzeptables Maß. Dies kann auf eine hohe Last oder ein internes Problem des Services zurückzuführen sein. Eine externe Serviceüberwachung kann verwendet werden, um Leistungsfehler und nicht reagierende Services zu erkennen.

Microservices Architektur

- Timeouts und Circuit Breaker
 - Ein Timeout ist ein Zeitzähler, der mit den Service Requests verknüpft ist und ausgeführt wird, wenn der Request gestellt wird.
 - Sobald der Zähler einen vordefinierten Wert erreicht hat, z. B. 10 Sekunden, geht der anrufende Service davon aus, dass der Service Request fehlgeschlagen ist, und handelt entsprechend.
 - Das Problem beim Timeout-Ansatz besteht darin, dass jeder Serviceaufruf an einen "fehlgeschlagenen Service" um den Timeout-Wert verzögert wird, sodass das gesamte System langsamer wird.
 - Anstatt Timeouts kann man Circuit Breaker verwenden. Wie bei einem elektrischen Schalter wird der Zugriff auf einen ausgefallenen Service verweigert ohne die damit verbundenen Timeouts und daraus resultierenden Verzögerungen.

Microservices Architektur

- Timeouts und Circuit Breaker



Microservices Architektur

- RESTful Services
 - Der Architekturstil REST (REpresentational State Transfer) basiert auf der Idee, Darstellungen digitaler Ressourcen von einem Server auf einen Client zu übertragen.
 - Eine Ressource repräsentiert einen Datenblock wie Kreditkartendaten, Krankenakte einer Person, eine Zeitschrift oder Zeitung, einen Bibliothekskatalog usw..
 - Der Zugriff auf Ressourcen erfolgt über ihre eindeutige URI, und RESTful-Services arbeiten mit diesen Ressourcen.
 - Eine Ressource ist i.a im Web eine Seite, die im Browser des Benutzers angezeigt werden soll.
 - Eine HTML-Darstellung wird vom Server als Antwort auf eine HTTP-GET-Anforderung generiert und zur Anzeige durch einen Browser oder eine Spezialanwendung an den Client übertragen.

Microservices Architektur

- RESTful Service Prinzipien
 - HTTP-Verben verwenden
 - Die im HTTP-Protokoll definierten grundlegenden Methoden (GET, PUT, POST, DELETE) müssen verwendet werden, um auf die vom Dienst bereitgestellten Vorgänge zuzugreifen.
 - Zustandslose Services
 - Services dürfen niemals einen internen Status beibehalten. Microservices sind zustandslos und passen daher zu diesem Prinzip.
 - URI adressierbar
 - Alle Ressourcen müssen einen URI mit einer hierarchischen Struktur haben, der für den Zugriff auf Unterressourcen verwendet wird.
 - XML oder JSON verwenden
 - Ressourcen sollten normalerweise in JSON oder XML oder in beidem dargestellt werden. Gegebenenfalls können andere Darstellungen wie Audio- und Videodarstellungen verwendet werden.

Microservices Architektur

- RESTful Service Operationen
 - Create
 - Implementiert mit HTTP POST, wodurch die Ressource mit dem angegebenen URI erstellt wird. Wenn die Ressource bereits erstellt wurde, wird ein Fehler zurückgegeben.
 - Read
 - Implementiert mit HTTP GET, das die Ressource liest und ihren Wert zurückgibt. GET-Operationen sollten niemals eine Ressource aktualisieren, damit aufeinanderfolgende GET-Operationen ohne dazwischenliegende PUT-Operationen immer den gleichen Wert zurückgeben.
 - Update
 - Implementiert mit HTTP PUT, das eine vorhandene Ressource ändert. PUT sollte nicht für die Ressourcenerstellung verwendet werden.
 - Delete
 - Implementiert mit HTTP DELETE, wodurch auf die Ressource mit dem angegebenen URI nicht zugegriffen werden kann. Die Ressource kann physisch gelöscht werden oder nicht.

Microservices Architektur

- Beispiel : Straßeninformationssystem
 - Stellen Sie sich ein System vor, das Informationen über Vorfälle wie Verkehrsverzögerungen, Straßenarbeiten und Unfälle in einem nationalen Straßennetz verwaltet. Auf dieses System kann über einen Browser unter folgender URL zugegriffen werden: <https://trafficinfo.net/incidents/>
 - Benutzer können das System abfragen, um Vorfälle auf den Straßen zu erkennen, auf denen sie fahren möchten.
 - Bei der Implementierung als RESTful-Webdienst müssen Sie die Ressourcenstruktur so gestalten, dass Vorfälle hierarchisch organisiert sind.
 - Beispielsweise können Vorfälle gemäß der Straßenkennung (z. B. A90), dem Ort (z. B. Stonehaven), der Fahrbahnrichtung (z. B. Norden) und einer Vorfallsnummer (z. B. 1) aufgezeichnet werden.
Daher kann auf jeden Vorfall über seine URI zugegriffen werden:
<https://trafficinfo.net/incidents/A90/stonehaven/north/1>

Microservices Architektur

- Beispiel : Strasseninformationssystem

- Die Antwort könnte so aussehen

Incident ID: A90N17061714391

Date: 17 June 2017

Time reported: 1439

Severity: Significant

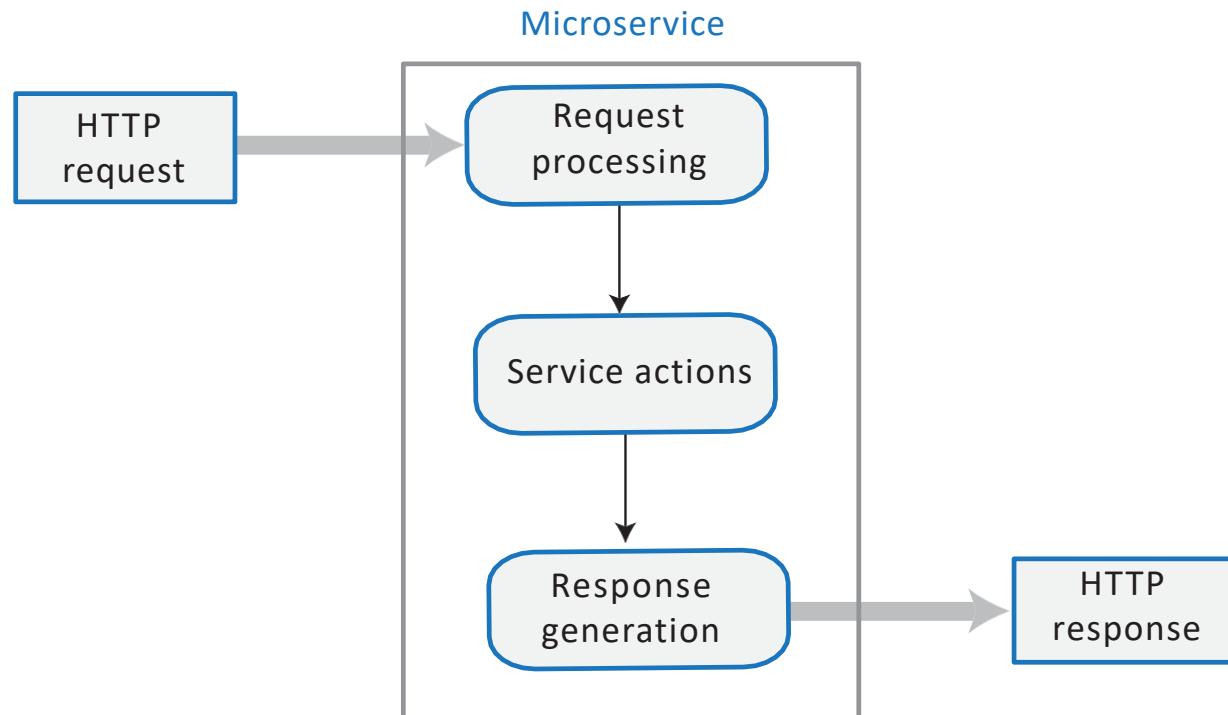
Description: Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes

Microservices Architektur

- Service Operationen
 - Retrieve
 - Gibt Informationen zu einem oder mehreren gemeldeten Vorfällen zurück.
Zugriff über das GET-Verb.
 - Add
 - Fügt Informationen zu einem neuen Vorfall hinzu. Zugriff über das POST-Verb.
 - Update
 - Aktualisiert die Informationen zu einem gemeldeten Vorfall. Zugriff über das PUT-Verb.
 - Delete
 - Löscht einen Vorfall. Das Verb DELETE wird verwendet, wenn ein Vorfall gelöscht wurde.

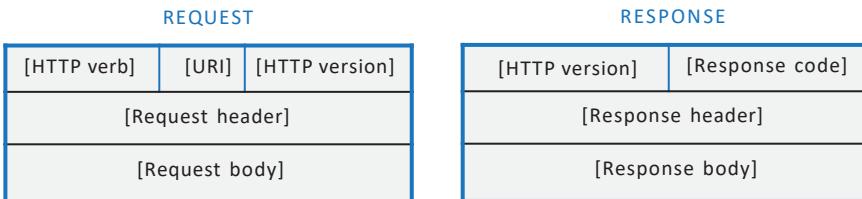
Microservices Architektur

- Service Operationen



Microservices Architektur

- Nachrichtenaufbau



REQUEST	RESPONSE
<pre>GET incidents/A90/stonehaven/ HTTP/1.1 Host: trafficinfo.net ... Accept: text/json, text/xml, text/plain Content-Length: 0</pre>	<pre>HTTP/1.1 200 ... Content-Length: 461 Content-Type: text/json { "number": "A90N17061714391", "date": "20170617", "time": "1437", "road_id": "A90", "place": "Stonehaven", "direction": "north", "severity": "significant", "description": "Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes." } { "number": "A90S17061713001", "date": "20170617", "time": "1300", "road_id": "A90", "place": "Stonehaven", "direction": "south", "severity": "minor", "description": "Grass cutting on verge. Minor delays" }</pre>

Microservices Architektur

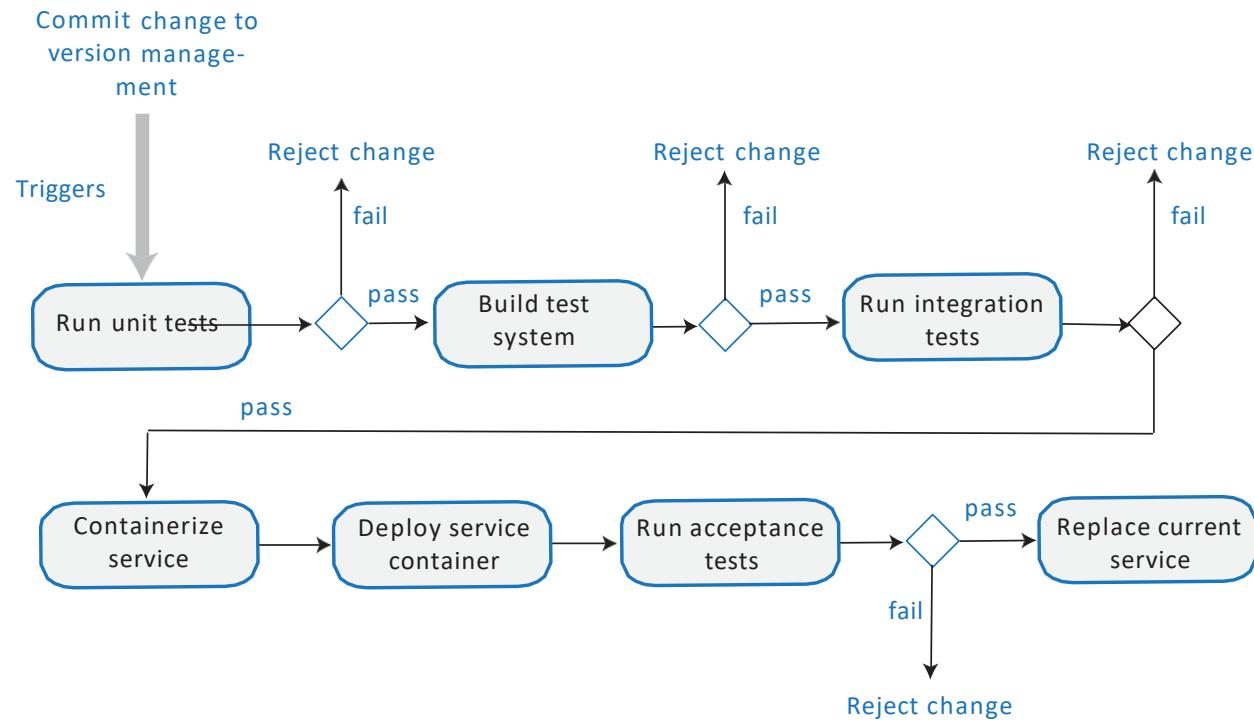
- Service Bereitstellung (Deployment)
 - Nachdem ein System entwickelt und bereitgestellt wurde, muss es auf Servern bereitgestellt, auf Probleme überwacht und aktualisiert werden, sobald neue Versionen verfügbar sind.
 - Wenn ein System aus zehn oder sogar Hunderten von Mikrodiensten besteht, ist die Bereitstellung des Systems komplexer als bei monolithischen Systemen.
 - Die Serviceentwicklungsteams entscheiden, welche Programmiersprache, Datenbank, Bibliotheken und andere Support-Software zur Implementierung ihres Service verwendet werden sollen. Folglich gibt es keine Standardbereitstellungskonfiguration für alle Dienste.
 - Es ist heutzutage üblich, dass Microservice-Entwicklungsteams für die Bereitstellung und das Service-Management sowie die Softwareentwicklung verantwortlich sind und eine kontinuierliche Bereitstellung verwenden.
 - Kontinuierliche Bereitstellung bedeutet, dass der geänderte Dienst erneut bereitgestellt wird, sobald eine Änderung an einem Dienst vorgenommen und validiert wurde.

Microservices Architektur

- Kontinuierliche Service Bereitstellung (Automatisierung)
 - Die kontinuierliche Bereitstellung hängt von der Automatisierung ab. Sobald eine Änderung festgeschrieben wird, wird eine Reihe automatisierter Aktivitäten zum Testen der Software ausgelöst.
 - Wenn die Software diese Tests , ‘ passiert hat, geht es dann in eine weitere Automatisierungs-Pipeline zum Erzeugen von Paketen und zur Installation.
 - Die Bereitstellung einer neuen Serviceversion beginnt damit, dass der Programmierer die Codeänderungen an ein Codeverwaltungssystem wie Git überträgt.
 - Dies löst eine Reihe automatisierter Tests aus, die mit dem geänderten Service ausgeführt werden. Wenn alle Servicetests erfolgreich ausgeführt werden, wird eine neue Version des Systems erstellt, die den geänderten Service enthält.
 - Ein weiterer Satz automatisierter Systemtests wird dann ausgeführt. Wenn diese erfolgreich ausgeführt wurden, ist der Service für die Bereitstellung bereit.

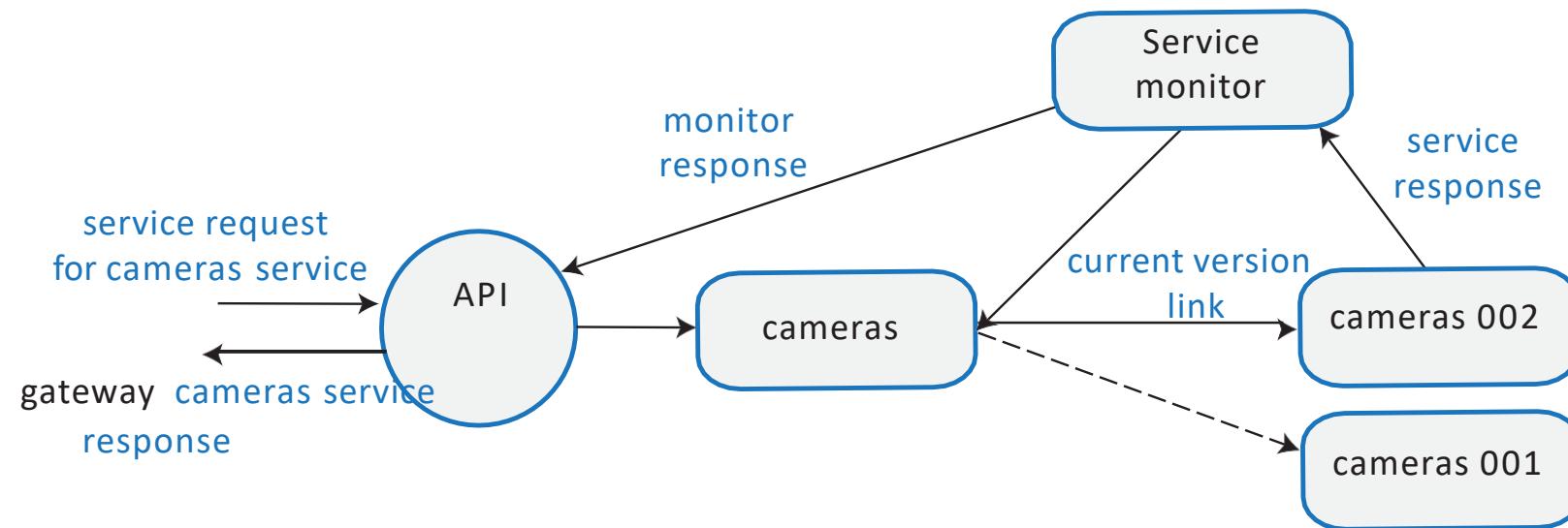
Microservices Architektur

- Kontinuierliche Service Bereitstellung (Automatisierung)



Microservices Architektur

- Versionierung bei Services



Microservices Architektur

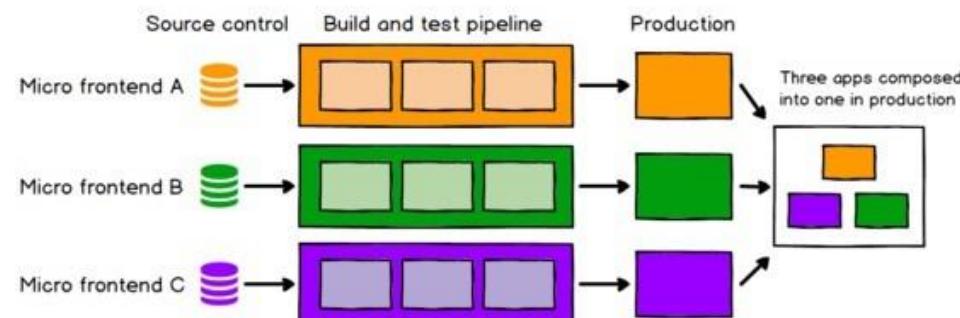
- **Zusammenfassung**
 - Ein Microservice ist eine unabhängige und in sich geschlossene Softwarekomponente, die in einem eigenen Prozess ausgeführt wird und über einfache Protokolle mit anderen Microservices kommuniziert.
 - Microservices in einem System können mit verschiedenen Programmiersprachen und Datenbanktechnologien implementiert werden.
 - Microservices haben eine einzige Verantwortung und sollten so konzipiert sein, dass sie leicht geändert werden können, ohne dass andere Microservices im System geändert werden müssen.
 - Microservices-Architektur ist ein Architekturstil, bei dem das System aus kommunizierenden Microservices aufgebaut ist. Es eignet sich gut für Cloud-basierte Systeme, bei denen jeder Mikroservice in einem eigenen Container ausgeführt werden kann.
 - Die beiden wichtigsten Aufgaben der Architekten eines Microservices-Systems bestehen darin, zu entscheiden, wie das System in Microservices strukturiert werden soll und wie Microservices kommunizieren und koordiniert werden sollen.

Microservices Architektur

- Zusammenfassung
 - Zu den zu treffenden Entscheidungen gehören die Entscheidung über Kommunikationsprotokolle für Mikroservices, die gemeinsame Nutzung von Daten, die zentrale Koordinierung von Services und das Fehlermanagement.
 - Der RESTful-Architekturstil wird häufig in Mikroservice-basierten Systemen verwendet. Dienste sind so konzipiert, dass die HTTP-Verben GET, POST, PUT und DELETE den Dienstvorgängen zugeordnet werden.
 - Der RESTful-Stil basiert auf digitalen Ressourcen, die in einer Microservices-Architektur mithilfe von XML oder häufiger JSON dargestellt werden können.
 - Continuous Delivery ist ein Prozess, bei dem neue Versionen eines Dienstes in Produktion gehen, sobald eine Serviceänderung vorgenommen wurde. Es ist ein vollständig automatisierter Prozess, der auf automatisierten Tests beruht, um zu überprüfen, ob die neue Version von „Produktionsqualität“ ist.
 - Wenn Continuous Delivery verwendet wird, müssen Sie möglicherweise mehrere Versionen der bereitgestellten Services verwalten, damit Sie zu einer älteren Version wechseln können, wenn bei einem neu bereitgestellten Service Probleme festgestellt werden.

Microservices Architektur

- Microfrontends
 - Eine gute Frontend-Entwicklung ist nicht einfach
 - Noch schwieriger ist es, die Frontend-Entwicklung so zu skalieren, dass viele Teams gleichzeitig an einem großen und komplexen Produkt arbeiten können
 - Wir betrachten noch einen aktuellen Trend, Frontend-Monolithen in viele kleinere, besser verwaltbare Teile zu zerlegen, und wie diese Architektur die Effektivität und Effizienz von Teams steigern kann, die an Frontend-Code arbeiten
 - Auch da gibt es Vorteile und Nachteile und Kosten und verschiedene Optionen der Implementierung



Microservices Architektur

- Microfrontends

- Definition

"An architectural style where independently deliverable frontend applications are composed into a greater whole"

Einige der wichtigsten Vorteile von Micro-Frontends sind

- kleinere, zusammenhängendere und wartbarere Codebasen
- Skalierbarere Organisationen mit entkoppelten, autonomen Teams
 - Teams können eigene Technologie- und Architekturentscheidungen treffen
- Separates Deployment
 - Die Möglichkeit, Teile des Frontends inkrementeller als bisher zu verbessern (upgrade), zu aktualisieren (update) oder sogar neu zu schreiben (rewrite)
- Es ist kein Zufall, dass diese Hauptvorteile die gleichen sind, wie die von Microservices

Microservices Architektur

- Microfrontends

Einige der Nachteile von Micro-Frontends sind

- Keine Framework Unterstützung
 - Aktuelle Frameworks sind nicht darauf ausgelegt
 - Teams müssen sich diese Unterstützung selbst bauen
 - Bedingt eine tiefe Kenntnis der benutzten Frameworks und von der Javascript-Plattform
 - Abhängig von der gewählten Implementierungsvariante können sich größere Bundles ergeben
 - Da jedes Micro-Frontend seine eigenen Frameworks mitbringen kann

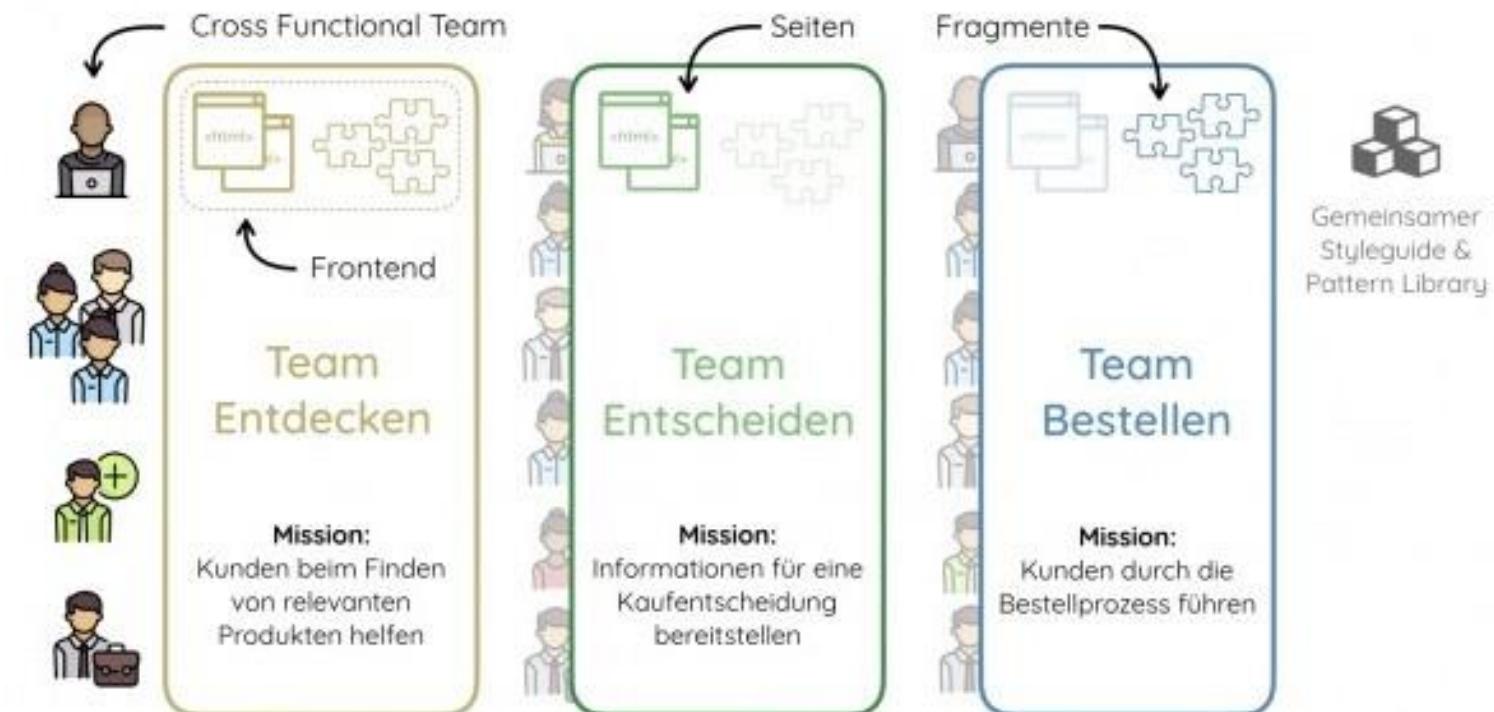
Microservices Architektur

- Microfrontends
 - Autonome Teams
 - Durch die Entkopplung der Codebasis als auch der Veröffentlichungszyklen erhalten wir einen langen Weg zu vollständig unabhängigen Teams, die einen Teil eines Produkts von der Idee bis zur Produktion und darüber hinaus besitzen können
 - Teams können die volle Verantwortung für alles haben, was sie benötigen, um ihren Kunden einen Mehrwert zu bieten, wodurch sie schnell und effektiv reagieren können
 - Damit dies funktioniert, müssen sich diese Teams eher um vertikale Bereiche der Geschäftsfunktionalität als um technische Fähigkeiten kümmern
 - Eine einfache Möglichkeit, dies zu tun, besteht darin, das Produkt basierend auf den Endbenutzern zu zerlegen, sodass jedes Micro-Frontend eine einzelne Seite (SPA = Single Page Application) der Anwendung kapselt und von Ende zu Ende einem einzelnen Team gehört
 - Dies führt zu einem höheren Zusammenhalt der Teamarbeit, als wenn Teams um technische oder „horizontale“ Belange wie Styling, Formulare oder Validierung gebildet würden.

Microservices Architektur

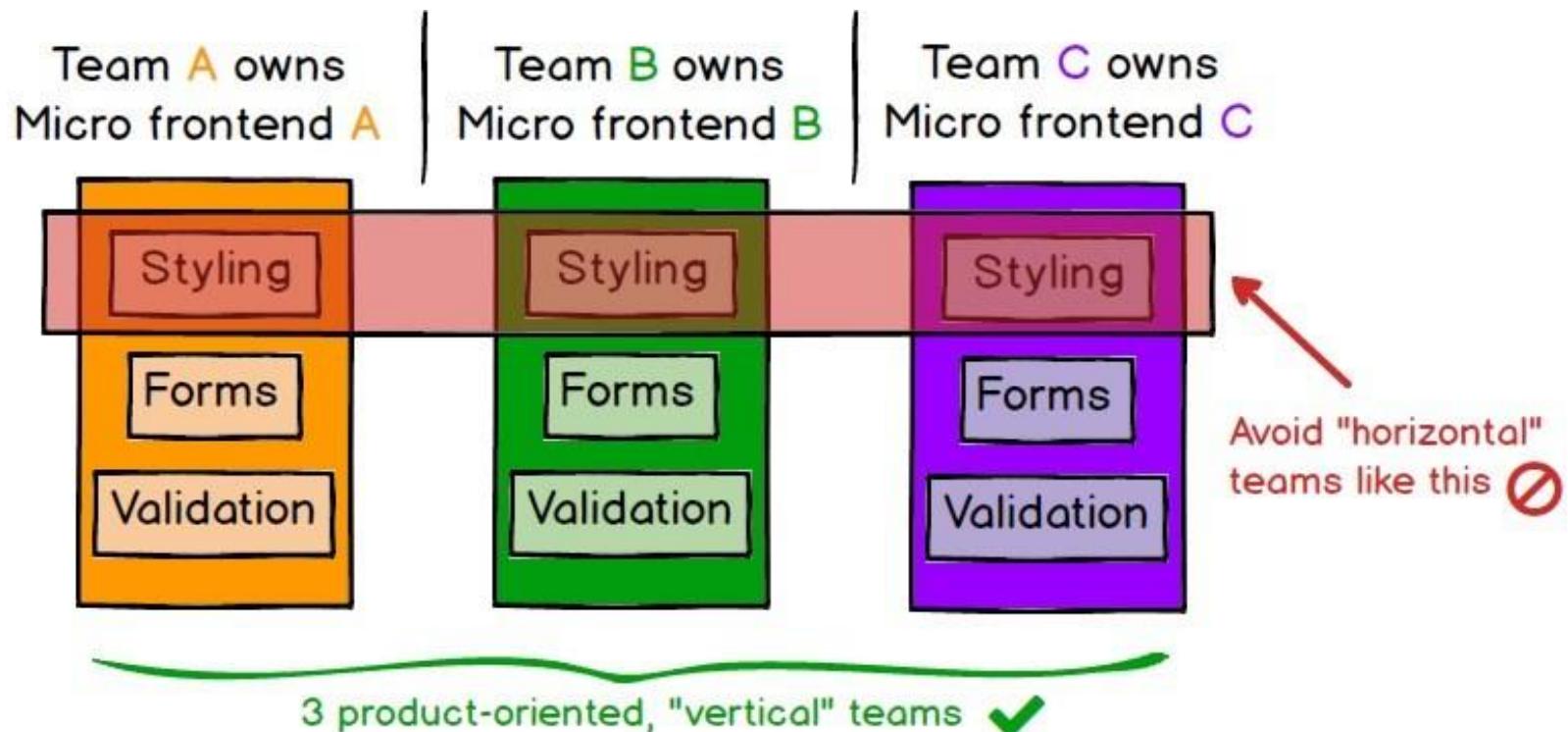
- Microfrontends
 - Autonome Teams

Micro Frontends



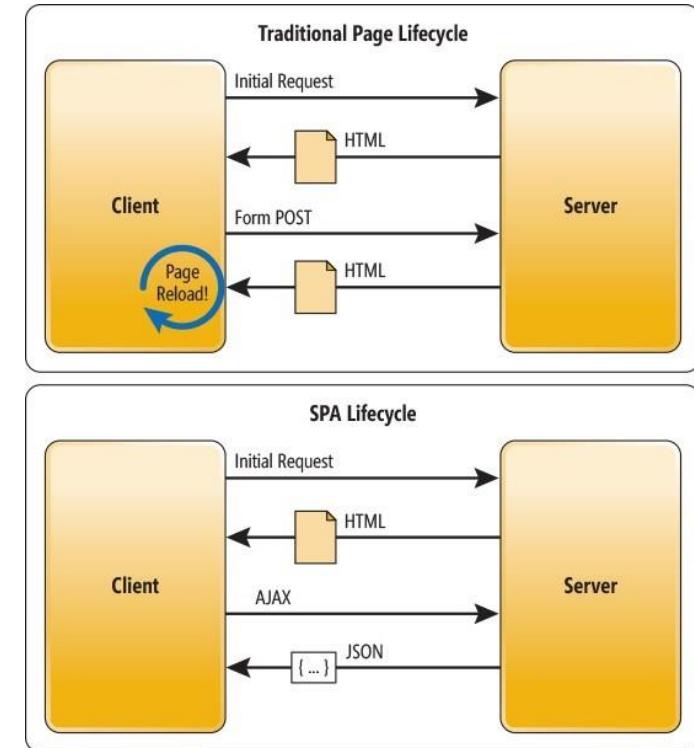
Microservices Architektur

- Microfrontends
 - Autonome Teams



Microservices Architektur

- Microfrontends
 - Single Page Applications
 - Eine Single-Page-Anwendung (SPA) ist eine Webanwendung oder Website, die mit dem Benutzer interagiert, indem sie die aktuelle Seite dynamisch neu schreibt, anstatt ganze neue Seiten von einem Server zu laden
 - Dieser Ansatz vermeidet eine Unterbrechung der User Experience zwischen aufeinanderfolgenden Seiten, wodurch sich die Anwendung eher wie eine Desktop-Anwendung verhält



Quelle : Wikipedia

Microservices Architektur

- Microfrontends
 - Voraussetzung
 - Man kann die Anwendung in verschiedene möglichst autarke Teile zerlegen
 - Domain Driven Design bezeichnet diese Teile als Domänen oder Subdomänen
 - Ein weit bekanntes Beispiel ist Office365



DAZN MICRO-FRONTENDS MANIFESTO



Independent implementation,
avoiding sharing logic



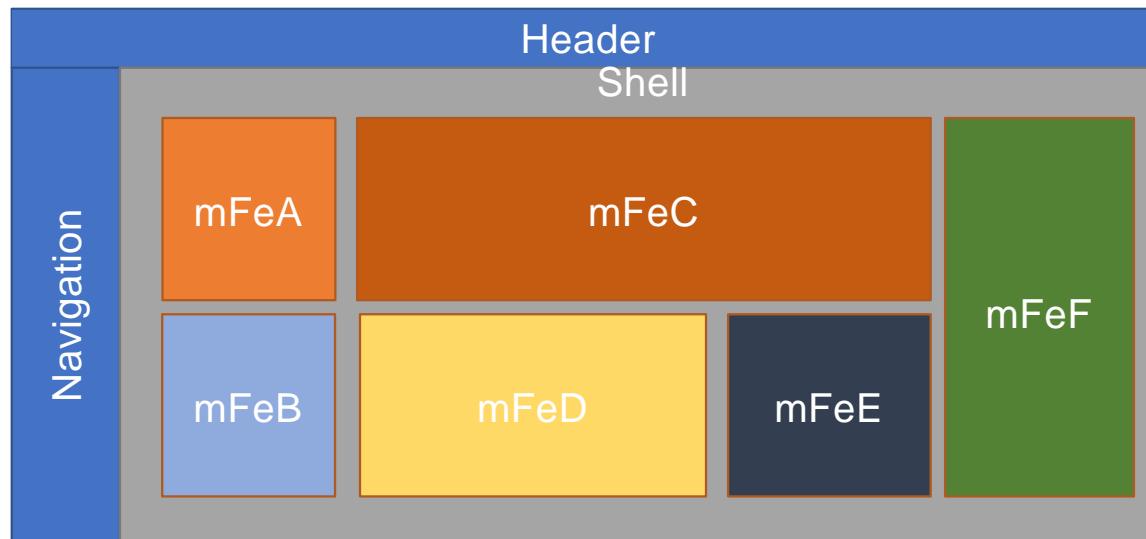
Modelled around
a Business Domain



Own by a single team

Microservices Architektur

- Microfrontends
 - Die Schritte zu Micro-Frontends
 - Der einfachste Ansatz sind Hyperlinks (wie bei Office365)
 - Ein komplexerer Ansatz ist eine Shell, die die einzelnen Micro-Frontends bei Bedarf lädt
 - Ist noch manuell zu realisieren, es gibt dafür leider noch keine geeigneten Frameworks



10) Performance

- Benutzer besitzen i.a. hohe Anforderungen an die Performance von Anwendungen
- Performance Anforderungen finden sich in den Qualitätsattributen und sind Teile einer guten Architektur
- Architekten und Software Entwickler müssen sich mit Performance Themen beschäftigen und die Verantwortung dafür übernehmen
- Wir betrachten
 - a) die Bedeutung von Performance
 - b) Performance Terminologien
 - c) Systematische Vorgehensweisen
 - d) Verschiedenste Strategien zur Verbesserung der Performance

a) Die Bedeutung von Performance

- Die Performance einer Anwendung ist ein Maß der Reaktionsfähigkeit ihrer Tätigkeiten
- Benutzer haben heute höhere Anforderungen als früher
- Sie erwarten gute bis sehr gute Antwortzeiten unabhängig vom Ort und des benutzten Devices
- Sollte eine Anwendung allerdings den Anforderungen nach Benutzbarkeit und funktionalen Anforderungen nicht Gerecht werden, dann spielt Performance auch keine Rolle mehr
- Erfüllt hingegen eine Anwendung alle Anforderungen, dann ist gute Performance um so wichtiger
 - in Webanwendungen sind z.B. die Ladezeiten von Seiten enorm wichtig
- Gute Performance erhöht die Produktivität der Anwender
- Gute Performance von Webseiten erhöht das Ranking in Suchmaschinen
- Performance ist ein Requirement und deshalb in den Qualitätsattributen

b) Performance Terminologien

- Bounce rate
 - manchmal auch Exit rate genannt
 - ein bounce eines Benutzer ist der Besuch und das Verlassen einer Seite ohne andere Seiten einer Anwendung besucht zu haben

$$\text{Bounce rate} = \frac{\text{total number of bounces}}{\text{total entries to a page}}$$

- wenn eine Seite langsam lädt, erhöht sich zwangsläufig die Bounce rate
- bounces sind jegliche Art von Verlassen einer Seite
(exit, back, neue Url, Browserfenster schließen, ...)

b) Performance Terminologien

- Conversion rate
 - ist der Prozentsatz von Benutzern, die eine Seite besuchen und die gewünschte Aktion durchführen
 - eine Bestellung machen, Registrierung als Mitglied, Download einer Datei, Bestellung eines Newsletters

$$\text{Conversion rate} = \frac{\text{number of goal achievements}}{\text{number of visitors}}$$

- Anwendungen mit schlechter Performance haben i.a. eine niedrige Conversion rate

b) Performance Terminologien

- Latency
 - die Zeit (oder die Wartezeit), die benötigt wird, um Informationen von einer Quelle zu einem Ziel zu schicken
 - manchmal auch beschrieben als die Zeit im Kabel oder im Netz
 - Normalerweise in Millisekunden gemessen
 - Einflußfaktoren sind Hardware, Netzwerkkomponenten wie Router u.ä., der Verbindungstyp, die Entfernung und der Netzwerkverkehr
 - Oft besteht ein Großteil der gesamten Latency aus der Latency zwischen dem Endbenutzer und dem Internet Service Provider (ISP) – dann redet man von last-mile latency

b) Performance Terminologien

- Throughput (Durchsatz)
 - eine Menge von Arbeitseinheiten pro Zeiteinheit
 - im Kontext eines Netzwerks ist das die Menge der übertragenen Daten pro Zeiteinheit
 - Typische Maßeinheiten sind
 - bits per second (bps)
 - megabits per second (Mbps)
 - gigabits per second (Gbps)
 - Im Kontext einer Anwendung ist Durchsatz die Anzahl von durchgeführten Aktionen pro Zeiteinheit
 - z.B. die Anzahl der Transaktionen pro Sekunde

b) Performance Terminologien

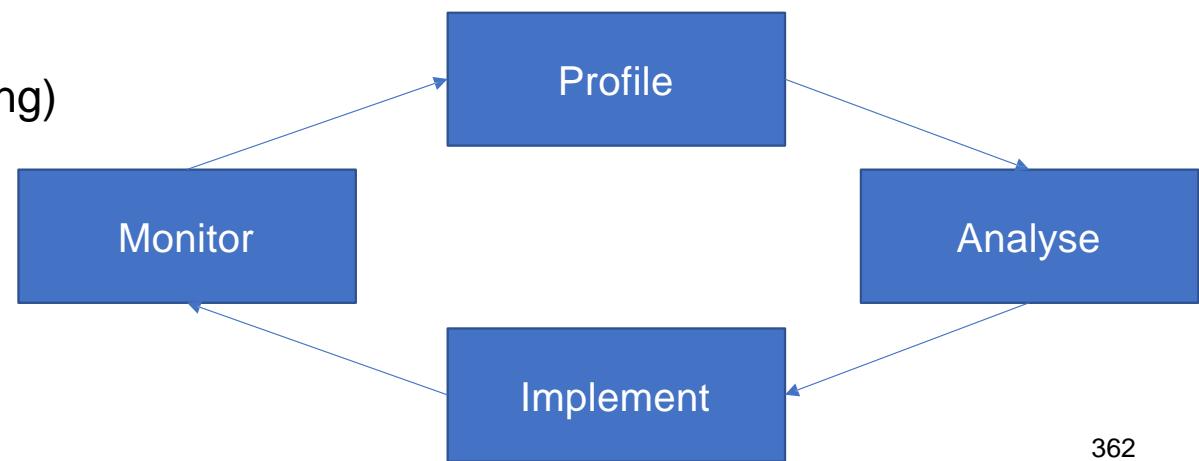
- Bandwidth (Bandbreite)
 - ist der maximale Durchsatz eines logischen oder physikalischen Kommunikationsweges
- Processing time
 - die Zeitdauer, die eine Anwendung benötigt, um eine Anforderung abzuarbeiten
 - Latency wird dabei nicht berücksichtigt
 - manchmal wird zwischen Server Processing time und Client Processing time unterschieden
 - es gibt einige Faktoren, die die Processing time beeinflussen können
(Hardware, Software, Architektur)
- Response time (Antwortzeit)
 - die Zeitdauer zwischen der Anforderung eines Benutzers und der dazugehörigen Antwort des Systems
 - Latency wird dabei nicht berücksichtigt (kann aber oft vom Benutzer nicht getrennt werden)

b) Performance Terminologien

- Workload (Last)
 - die Menge von Verarbeitungsschritten, die ein System zu einer Zeit zu bewältigen hat
 - Verarbeitungsschritte benutzen Anteile von Prozessorkapazitäten und lassen damit weniger für andere Verarbeitungen übrig
 - Typische Workloads sind CPU-, Memory-, I/O- und Datenbank-Workloads
 - Regelmäßige Messungen von Workloads helfen Spitzen Belastungen vorherzusagen und die Performance einer Anwendung bei unterschiedlichen Belastungen zu messen und zu optimieren
- Utilization (Auslastung)
 - die Prozentzahl der Zeit, in der eine Resource benutzt wird, verglichen mit der Gesamtverfügbarkeit einer Resource
 - Beispiel : eine CPU ist innerhalb einer Minute 45 Sekunden mit einer Transaktion beschäftigt dann ist die Utilization 75%
 - Utilization von CPU, Memory und Platten spielen eine wichtige Rolle bei der Performancebetrachtung von Anwendungen

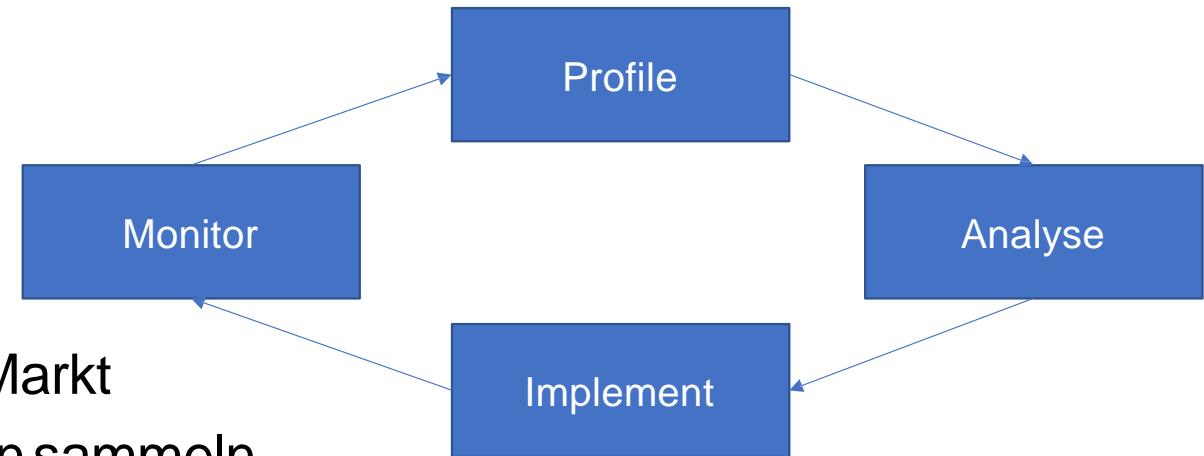
c) Systematische Vorgehensweisen

- Bei der Performancebetrachtung muss das Entwicklungsteam involviert sein
- Das Entwicklungsteam muss die Verantwortung für die Performance einer Anwendung übernehmen
- Um die Performance zu verbessern, ist es sinnvoll systematisch vorzugehen
- Eine beispielhafte Vorgehensweise benutzt einen iterativen Ansatz
 - eine Anwendung messen (Profiling)
 - die Ergebnisse auswerten
 - Änderungen implementieren
 - Änderungen überwachen (Monitoring)



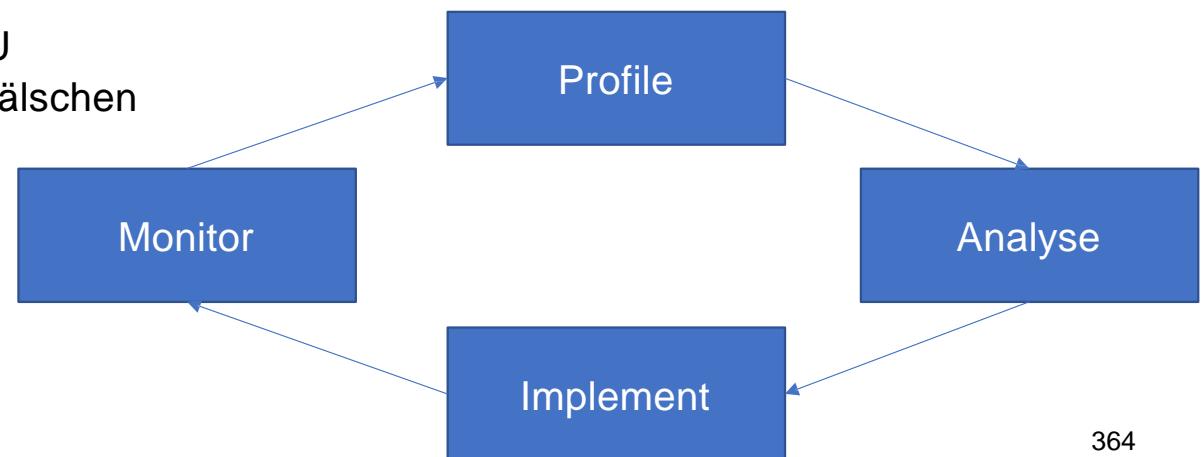
c) Systematische Vorgehensweisen

- eine Anwendung messen (Profiling)
 - ist eine Analyse einer Anwendung, die die Ausführung einer Anwendung misst
 - überall in einer Anwendung werden dazu Meßpunkte/Meßstellen implementiert
 - sie erfassen Daten wie
 - Laufzeit von Modulen
 - Anzahl Aufrufe von Modulen
 - Durchsatz
 - Antwortzeiten
 - Auslastung
- Es gibt dazu Profiling Tools am Markt
- Es gibt 2 Arten wie Profiler Daten sammeln
 - Instrumentation
 - Statistical



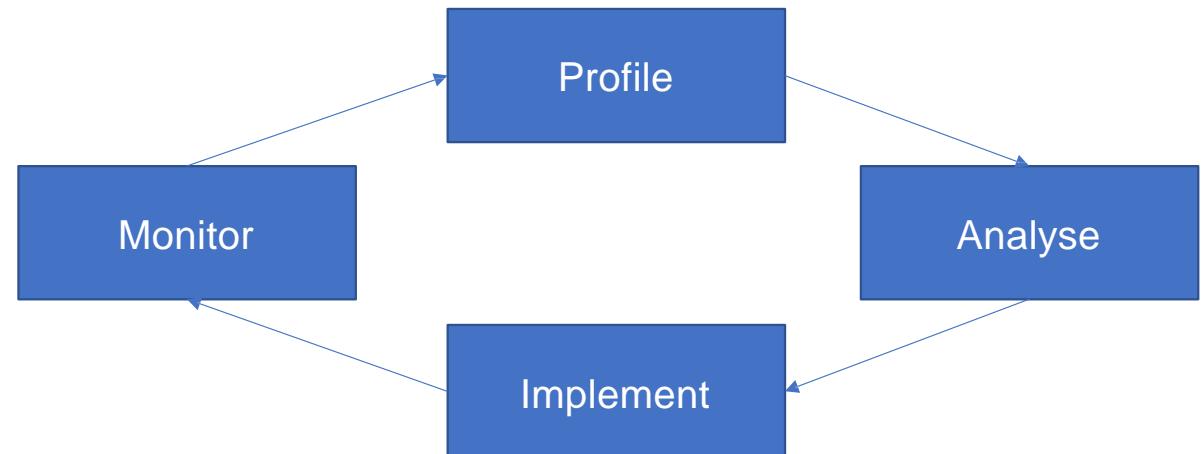
c) Systematische Vorgehensweisen

- Arten von Profilern
 - Instrumentation
 - Code wird zur Anwendung hinzugefügt
 - ein Call in den Profiler am Anfang und Ende jeder Funktion
 - es gibt Profiler, die Source Code modifizieren können
 - der eingefügte Code kann die Meßergebnisse beeinflussen
 - gute Profiler berücksichtigen das
 - eingefügter Code kann aber auch die Optimierungsstrategie der CPU beeinflussen und damit Werte verfälschen



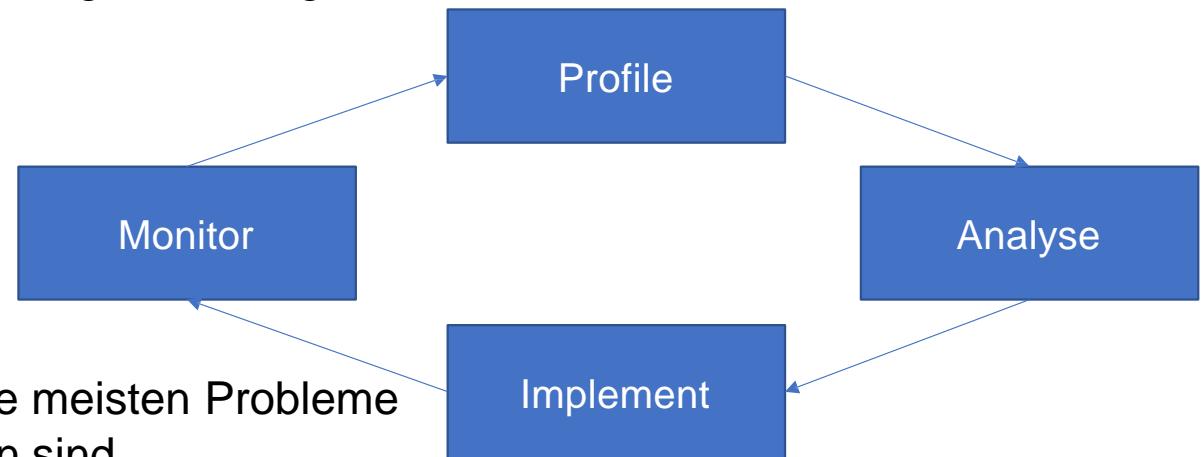
c) Systematische Vorgehensweisen

- Arten von Profilern
 - Statistical
 - sammelt Messwerte außerhalb einer Anwendung
 - ohne Code einzufügen
 - benutztes Verfahren : Sampling
 - Sampling nutzt die Interrupts des Betriebssystems zum Sammeln von Messwerten
 - Messergebnisse sind deshalb oft nur Näherungswerte
 - Instrumentation liefert genauere Werte



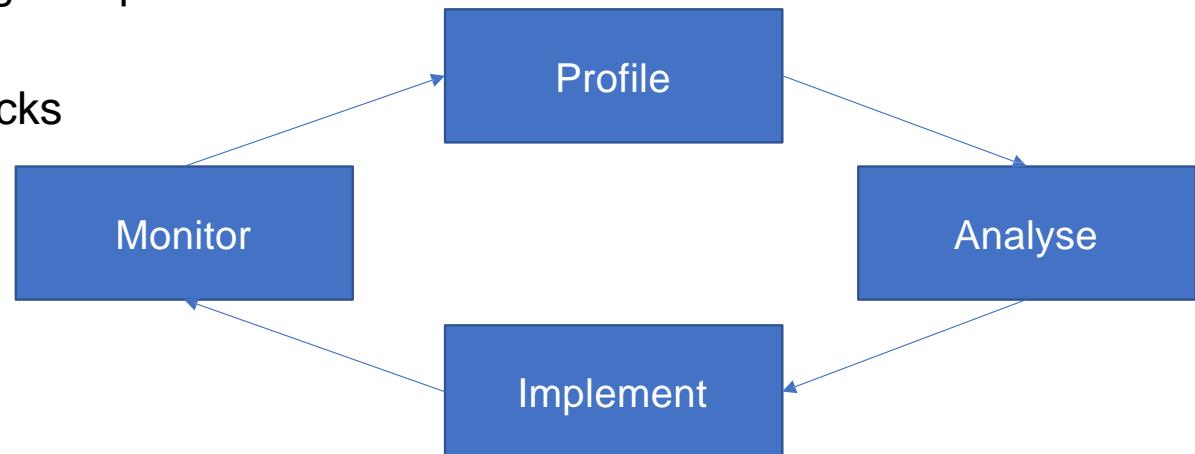
c) Systematische Vorgehensweisen

- Analyse der Daten
 - zum Aufspüren von Bottlenecks
 - Bottlenecks sind die Teile einer Anwendung, die die Performance negativ beeinflussen
 - der Fokus liegt also auf der Optimierung/Behebung der Bottlenecks
 - Bottlenecks könnten sein
 - CPU
 - Memory
 - Netzwerk
 - Datenbanken
 - I/O
 - die (meine) Erfahrung zeigt, daß die meisten Probleme in der Anwendung selbst verborgen sind
 - Entwickler wollen das i.a. nicht wahrhaben



c) Systematische Vorgehensweisen

- Implementieren der Änderungen
 - basierend auf den Messergebnissen
- Überwachen der Änderungen
 - die Änderungen müssen auf Wirkung überprüft und überwacht werden
 - es kann sein, daß sich jetzt Bottlenecks in andere Bereiche der Anwendung verschoben haben
 - Monitoring von Anwendungen ist immer angebracht
 - der Zyklus kann von vorne beginnen



d) Verschiedenste Strategien zur Verbesserung der Performance

- Caching
 - Caching ist ein gängiges Hilsmittel zur Erhöhung der Performance
 - benötigt genaue Planung und muss in die Architektur aufgenommen werden
 - Caching kann den Zugriff auf langsame Systeme/Medien optimieren
 - Caching bedeutet immer das Anlegen von Kopien auf anderen (schnelleren) Medien
 - der Cache sollte nah bei der Anwendung liegen, um Latency zu vermeiden
 - Caching erzeugt zusätzlichen Verwaltungsaufwand

d) Verschiedenste Strategien zur Verbesserung der Performance

- Caching
 - In verteilten Anwendungen findet man
 - privater (lokaler) Cache auf dem Rechner des Endusers
 - schneller Cache im Memory
 - Einstellungen, Credentials
 - gemeinsam genutzter Cache am Server
 - Cache Service oder spezielle Implementierung
 - genutzt von mehreren Instanzen oder Cluster der Anwendung
 - kann in Storagesystemen eingebaut sein
 - kann in spezieller Hardware/Software/Systemen im Netz verteilt sein

d) Verschiedenste Strategien zur Verbesserung der Performance

- Caching
 - Priming
 - vorfüllen eines Caches beim Starten einer Anwendung
 - wegschreiben des Caches beim Beenden einer Anwendung
 - Nicht mehr benötigte Cache-Inhalte (Invalidating) löschen oder ersetzen
 - kann extrem kompliziert sein
 - es gibt 2 gebräuchliche Strategien
 - Expiring
 - Evicting

d) Verschiedenste Strategien zur Verbesserung der Performance

- Caching
 - Cache Strategien
 - Expiring
 - » nach Datum oder Zeitspanne (zum 31.12.2019, 1 Tag)
 - Evicting
 - » Least recently used (die am wenigsten benutzten Daten werden gelöscht)
 - » most recently used (gerade benutzt, in absehbarer Zeit nicht nochmal benutzt)
 - » First-in First-out (zuerst eingestellte (älteste) Daten werden gelöscht)
 - » Last-in First-out (selten benutzt)
 - » explicitly evicting (z.B. aus dem Cache gelöscht, wenn aus der Datenbank gelöscht)

d) Verschiedenste Strategien zur Verbesserung der Performance

- Caching
 - Cache Pattern
 - es gibt für Anwendungen 2 Wege Caches zu nutzen
 - die Anwendung verwaltet den Cache selbst neben dem eigentlichen Datenpool
 - » cache aside
 - die Anwendung sieht den Cache als den Datenpool, der dann unabhängig von der Anwendung den dahinterliegenden echten Datenpool verwaltet
 - » read through (Cache im Lesepfad)
 - » write through (Cache im Schreibpfad)
 - » write behind (Puffern von Schreibanweisungen, verzögertes Schreiben)

d) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)
 - Vermeidung von Datentransfer über das Netzwerk
 - Browercaching von Daten (z.B. große Bilddateien)
 - Sharing von CSS oder Javascript Dateien über mehrere Webseiten
 - Benutzung von entsprechenden Direktiven in den HTTP Headern
 - Validation token (ETag header)
 - Cache-control directives

d) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)
 - Benutzung von entsprechenden Direktiven in den HTTP Headern
 - Validation token (**ETag** header)
 - der ETag HTTP response header ist ein Identifier für eine bestimmte Version einer Resource. Er trägt zur Effizienzsteigerung von Caches bei und spart Bandwidth, weil ein Webserver nur die Teile schicken muss, die sich geändert haben
 - Zusätzlich helfen Etags zu verhindern, daß sich simultane Updates auf Ressourcen gegenseitig überschreiben ("mid-air collisions")
 - ETags sind Fingerabdrücken ähnlich und können in Vergleichen verwendet werden
 - Beispiel:
 - » ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4" ist der Hash einer Webseite
 - » Wenn Änderungen dieser Seite gespeichert werden sollen, enthält der POST request einen If-Match header mit dem Inhalt des ETag zur Überprüfung auf Änderungen (If-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4")
 - » Sind die Hashes ungleich, wurde die Seite inzwischen geändert und ein 412 Precondition Failed error wird erzeugt

d) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)
 - Benutzung von entsprechenden Direktiven in den HTTP Headern
 - Cache Control directives
 - steuern das Cachen von Responses (Antworten), die Bedingungen zum Cachen und die Zeitspannen
 - für das Verhindern der Ablage von sensitiven Daten in Caches zu verhindern, wird die Direktive **no-store** verwendet (gilt für den Browercache und alle Caches auf dem Weg zum Browser)
 - die **private** Direktive erlaubt das Cachen im Browser des Benutzers, verhindert aber die Ablage in Caches auf dem Weg
 - die **non-cache** Direktive sorgt im Zusammenspiel mit der If-match Direktive, ob aus dem Cache gelesen wird oder nicht
 - die **max-age** Direktive gibt an wie lange (in Sekunden und relativ zur Zeit der Antwort) eine Antwort aus dem Cache gelesen werden kann
 - Caches können vom Benutzer im Browser gelöscht werden
 - Eine oft benutzte Variante zum gesteuerten Nachladen von Ressourcen ist die Veränderung der URL, was ein Nachladen auslöst
 - Bei URL's auf Dateien kann ein Fingerprint eingefügt werden, was eine Art der Versionierung direkt unterstützt kann

d) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)
 - Kompression
 - eine wichtige Technik zur Performancverbesserung
 - Nutzung eines Algorithmus zur Verkleinerung von Ressourcen
 - Verbessert Transferzeiten und Bandbreiten Nutzung
 - Webserver und Browser haben entsprechende Funktionen bereits implementiert
 - Webserver und Browser müssen sich über den benutzten Algorithmus einig sein
 - die 2 Haupttypen von Kompression sind
 - file compression
 - content-encoding (end-to-end) compression

d) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)

- Kompression

- file compression

- » Images, Videos und Audiodateien haben eine hohe Rate von Redundanz
 - » Images machen in modernen Webseiten den Großteil der Bytes, die geladen werden, aus
 - » diese Ressourcen sollten unbedingt komprimiert werden
 - » dafür gibt es verschiedene Tools und Algorithmen
 - » man unterscheidet 2 Kompressionsalgorithmen
 - lossless compression (verlustlose Komprimierung)
 - alle Bytes einer Ressource können 1:1 wiederhergestellt werden
 - GIF (Graphics Interchange File) und PNG (Portable Network Graphics) sind Dateiformate, die lossless compression beinhalten
 - Images hoher Qualität und der Anforderung nach verlustfreiem Transfer werden deshalb in PNG gespeichert
 - lossy compression
 - nicht verlustfrei bei der Komprimierung
 - funktioniert gut bei Images, Video und Audio Dateien, bei denen es nicht auf hohe Qualität ankommt
 - es gibt verschiedene Stufen der Komprimierung
 - JPEG (Joint Graphic Expert Group) ist ein Beispiel

d) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)
 - Kompression
 - Content-encoding(end-to-end) compression
 - » signifikante Performancesteigerungen möglich
 - » der komplette Body-Teil einer Nachricht wird komprimiert
 - » nur der Client (Browser) dekomprimiert die Nachricht, keine Stellen dazwischen
 - » Server und Client müssen den Algorithmus via **content negotiation** abstimmen
 - » es gibt die gebräuchlichen Algorithmen **content-encoding gzip** und **content-encoding br** (Brotli)
 - » bereits komprimierte Ressourcen werden bei nochmaliger Komprimierung nicht kleiner, eher größer

d) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)
 - Minifying Resourcen
 - Minification nennt man das Entfernen aller unnötigen Zeichen aus Dateien
 - Leerzeichen aus Javascript, HTML und CSS
 - Bundling Resourcen
 - das Zusammenführen mehrerer Files mit gleicher Aufgabe in ein File (foo.css und bar.css in styles.css)

d) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP/2
 - rückwärtskompatibel zu HTTP/1.1
 - ist ein binäres Übertragungsprotokoll (HTTP/1.1 ist textorientiert)
 - kompakter, einfacher zu parsen und weniger fehleranfällig
 - kann Multiplexing
 - paralleles und asynchrones Senden und Empfangen von Nachrichten über eine TCP Verbindung
 - macht Bundling überflüssig und die Nutzung von Caches werden optimiert
 - kann Server Push
 - proaktives Senden von benötigten Ressourcen vom Server an den Client
 - vorsichtig einzusetzen, kann zu zuviel Übertragungen führen
 - Header Kompression
 - HPACK Kompressionsformat (Huffman lossless)

d) Verschiedenste Strategien zur Verbesserung der Performance

- Content Delivery Networks
 - Benutzer von Resourcen können weltweit verteilt sein
 - um Latency zu vermindern gibt es sog. Content Delivery Networks (CDN)
 - geografisch verteilte Servergruppen, um Content local Benutzern zur Verfügung zu stellen
 - CDN unterstützen zusätzlich Load Balancing, Caching, Minification und Dateikomprimierung
- Optimierte Webfonts
 - spezielle Fonts werden in den Browser geladen (wenn unterstützt)
 - durch geschickten Einsatz von Webfonts kann man Ladezeiten von Webpages verringern
 - Leider gibt es noch keinen einheitlichen Standard
- Optimierung des kritischen Rendering Paths (critical rendering path CRP)
 - Wenn man die Art und Weise, wie ein Browser eine Site renderd, kennt, dann kann man die Seiten entsprechend aufbauen, daß die interne Erzeugung des Document Object Model (DOM) und des CSS Object Model schnell abläuft

d) Verschiedenste Strategien zur Verbesserung der Performance

- Datenbank Performance
 - Effizientes Datenbankschema
 - Normalisierung
 - Manchmal muß aus Performancegründen (teilweise) denormalisiert werden
 - Nutzung von Schlüsseln (Primary, Foreign)
 - Auswahl von geeigneten Datentypen (Konvertierung in andere Typen kostet Zeit)
 - Nutzung von Indizes
 - Skalierung von Datenbankservern
 - Nutzung von Transaktionen
 - CAP Theorem
 - Consistency, Availability, Partition intolerance
 - nur 2 der 3 Anforderungen können in verteilten Systemen erfüllt werden
 - manche NoSQL Datenbanken stellen Performance über Consistency

10) Security

- Software Systeme zu designen und entwickeln, die eine hohe Sicherheit besitzen, wird im wichtiger
- Die Anzahl von Attacken auf Firmennetzwerke und Anwendungen nimmt dramatisch zu
- Der Schaden durch solche Attacken kann für Firmen existenzbedrohend sein
- Wir betrachten
 - a) Zustände von Informationen
 - b) das CIA Dreieck
 - c) Bedrohungsmodellierung
 - d) Secure by Design
 - e) Kryptografie
 - f) Identity und Access Management (mit Authentifizierung und Authorisierung)
 - g) Einige Risiken von Webanwendungen

Security

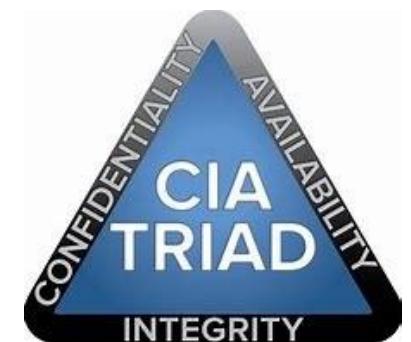
- Security ist die Fähigkeit einer Software Anwendung heimtückische Angriffe zu vermeiden und abzuwehren und die nicht genehmigte Benutzung der Anwendung und deren Daten zu verhindern
- sie schützt das wichtigste Kapital einer Firma : die Informationen
- Software Architekten müssen Security Themen in ihre Arbeit integrieren und von Anfang an beachten
- Security ist eines der Qualitätsattribute und wird aus entsprechenden Anforderungen abgeleitet
- Security ist ein Teil der Architektur und muss bei der Erfassung der Anforderungen, beim Design, in der Entwicklungsphase und beim Testen berücksichtigt werden

a) Zustände von Informationen

- Informationen, die abgesichert werden sollen, können in 3 Zuständen vorliegen
 - im Ruhezustand (at rest)
 - in Gebrauch (in use)
 - auf dem Transportweg (in transit)
- in allen 3 Zuständen müssen Informationen abgesichert werden
 - sichere Ablage
 - sichere Verarbeitung
 - sichere Übertragung

b) Das CIA Dreieck

- CIA Triad
 - Confidentiality (Vertraulichkeit)
 - unberechtigter Zugriff auf Informationen vermeiden
 - Integrity (Integrität)
 - unberechtigtes Verändern oder Löschen von Informationen vermeiden
 - Availability (Verfügbarkeit)
 - Informationen Berechtigten zeitnah zur Verfügung stellen



c) Bedrohungsmodellierung

- Threat modelling
 - strukturierte Vorgehensweise, um die Sicherheit von Anwendungen zu analysieren
 - kommt aus der Sicht des Angreifers, greift die Anwendung von außen an
 - identifiziert und priorisiert potentielle Sicherheitsbedrohungen, damit Entwicklerteams erkennen können, wo ihre Software am meisten angreifbar ist
 - nach der Analyse hilft ein daraus resultierender Plan die Sicherheitslücken zu verkleinern oder zu schließen
 - betrachtet die Anwendung in Teilen (decomposing)
 - wo ist das wertvollste Gut
 - was könnten mögliche Ziele von Attacken sein
 - betrachtet die Angreifer
 - interne und externe Angreifer
 - betrachtet alle Schnittstellen zu anderen Systemen und Benutzern

c) Bedrohungsmodellierung

- Potentielle Bedrohungen
 - Threat Kategorisierungsmodell STRIDE
 - Spoofing Identity (Verschleierung)
 - sich als jemanden anderen ausgeben
 - Tampering with data (Verfälschung)
 - modifizieren, verfälschen von Daten
 - Repudiation (Zurückweisung)
 - zurückweisen von Verantwortlichkeit (z.B. da nicht protokolliert)
 - Information disclosure (Offenlegung)
 - fehlende Absicherung von Informationen gegen unberechtigten Zugriff
 - Denial-of-service (Abweisen, Stören)
 - stören und/oder ausschalten von Services
 - Elevation of privilege (Erhöhung)
 - unberechtigtes Erlangen höherer Berechtigungen

Threat	Definition
Spoofing	An attacker tries to be something or someone he/she isn't
Tampering	An attacker attempts to modify data that's exchanged between your application and a legitimate user
Repudiation	An attacker or actor can perform an action with your application that is not attributable
Information Disclosure	An attacker can read the private data that your application is transmitting or storing
Denial of Service	An attacker can prevent your legitimate users from accessing your application or service
Elevation of Privilege	An attacker is able to gain elevated access rights through unauthorized means

c) Bedrohungsmodellierung

- Potentielle Bedrohungen
 - Risk assessment model DREAD (Priorisierungsmodell)
 - Damage potential
 - potentieller Schaden
 - Reproducibility
 - Reproduzierbarkeit einer Attacke
 - Exploitability
 - Nachvollziehbarkeit
 - Affected users
 - Anzahl betroffener Benutzer
 - Discoverability
 - Grad der Erkennbarkeit

DREAD: Example Ratings

Rating	High (3)	Medium (2)	Low (1)
Damage potential	Attacker can subvert security system, get full trust authorization, run as admin, upload content	Leaking sensitive information	Leaking trivial information
Reproducibility	Attack can be reproduced every time and does not require a timing window	Attack can be reproduced but only with timing window and particular race situation	Attack is difficult to reproduce, even with knowledge of the security hole
Exploitability	A novice programmer could make the attack in a short time	A skilled programmer could make the attack, then repeat the steps.	Attack requires extremely skilled person and in-depth knowledge every time to exploit
Affected users	All users, default configuration, key customers	Some users, non-default configuration	Small percentage of users, obscure feature; affects anonymous users
Discoverability	Published information explains the attack; Vulnerability is in most commonly used feature	Vulnerability is in seldom-used part of product	The bug is obscure; unlikely that users will work out damage potential



d) Secure by Design

- Prinzipien und Praktiken Software zu erstellen, die sicher ist
 - Minimierung der Oberfläche für Attacken (attack surface)
 - alle Punkte, die attackiert werden können
 - Defense in Depth
 - mehrere Hindernisse hintereinander
 - Principle of least privilege (PoLP)
 - gerade soviel Berechtigung wie nötig
 - ist oft schwierig zu bestimmen, Definition von Rollen helfen beim Design
 - Simplicity (KISS)
 - einfachere Systeme sind einfacher zu sichern
 - Secure by default
 - alle Security Mechanismen sind default auf „an“
 - Default deny
 - Zugriffe nur auf ... anstatt auf alles Zugriff außer ...

d) Secure by Design

- Prinzipien und Praktiken Software zu erstellen, die sicher ist
 - Validating input
 - alle Eingaben prüfen, egal woher sie kommen (User, Schnittstelle, Kommunikation)
 - Secure the weakest link
 - eine Kette ist nur so stark wie das schwächste Glied in der Kette
 - Angreifer suchen zuerst nach solchen Schwachstellen
 - Security must be usable
 - dürfen die Benutzbarkeit nicht herabsetzen, sonst gehen Benutzer andere (unsichere) Wege
 - Fail securely
 - hohe Sicherheit auch im Fehlerfall (und Fehler passieren)

e) Kryptografie

- Verschlüsselung
 - symetrisch (geheimer Schlüssel)
 - benutzt einen Schlüssel zum Verschlüsseln
 - asymetrisch (öffentlicher Schlüssel)
 - benutzt 2 Schlüssel zum Verschlüsseln (1 x geheim, 1 x öffentlich)
 - cryptographic hash functions
 - ein fixer Wert, der einen beliebigen Input repräsentiert
 - es gibt verschiedenste Algorithmen
 - MD5, SHA-256, SHA-512
 - kann zum Vergleich von 2 Dateien, Texten, Datenblöcken benutzt werden
 - werden genutzt für
 - digitale Signaturen
 - HTTPS Zertifikate
 - SSL/TLS oder SSH Protokolle

f) Identity und Access Management

- Identity und Access Management (IAM)
 - 2 der fundamentalen Konzepte von IAM sind
 - Authentication (Authentifizierung)
 - ist jemand der, den er vorgibt zu sein
 - Validierung der Identität
 - Multi-factor authentication (MFA)
 - » zusätzlicher Level von Sicherheit
 - » Sicherheit wird über mehrere sog. Faktoren sichergestellt
 - » eine Variante ist 2FA (two-factor identification)
 - » Typen von Faktoren
 - Knowledge factor (etwas, was der Benutzer wissen sollte oder weiß)
 - Possession factor (etwas, was die Person hat, wie ein Mobile Phone, das einen Code empfangen kann)
 - Inherence factor (Nutzung von Finger- oder Augen-Scannern oder anderer biometrischer Daten)

f) Identity und Access Management

- Identity und Access Management (IAM)
 - 2 der fundamentalen Konzepte von IAM sind
 - Authorization
 - was einem Benutzer erlaubt ist zu tun
 - Zugriffssteuerung auf Systeme oder Teile von Systemen
 - die Granularität von Zgriffsrechten ist ein wichtiger Aspekt in Software Architekturen
 - » zu klein = zu grobe Rechtevergabe
 - » zu groß = zu komplizierte Vergabe und Anwendung
 - Passwörter
 - im Klartext (unbedingt vermeiden !)
 - verschlüsselt (Verschlüsselungsalgorithmus muss geschützt sein)
 - hashed (dictionary attacks versuchen Verzeichnisse zu hacken, rainbow tables dienen als Vergleich)

f) Identity und Access Management

- Identity und Access Management (IAM)
 - Domain authentication
 - Benutzung von sog. Domain Controllern zusammen mit einem Directory Service wie Active Directory (AD)
 - zentralisierter Identity Provider (IdP)
 - manche Anwendungen müssen mit API's interagieren, die nicht in der Gleichen Domäne sind, manchmal sogar weltweit öffentlich sein müssen
 - Domain Controller können das nicht leisten
 - Zugriffsrechte über Domänengrenzen hinweg können mit IdP's realisiert werden
 - IdP's können auch die Verantwortung der Authentifizierung von der Anwendung nehmen und selbst verwalten
 - all Funktionen wie Benutzeregistrierung, Passwort Policies, Passwort Änderungen und die Behandlung von ausgesperrten Benutzern kann von IdP's übernommen werden
 - Einmal implementiert, kann die Funktionalität von allen Anwendung genutzt werden
 - Erniedrigt die Komplexität des Gesamtsystems

10f) Identity und Access Management

- Identity und Access Management (IAM)
 - OAuth 2/OpenID Connect (OIDC)
 - offener Standard für Authentifizierung
 - unterstützt eine Anwendung beim Zugriff auf Resourcen anderer Anwendungen und gewährt Zugriff auf eigene Resourcen für andere Anwendungen
 - OIDC ist eine Identity Layer oberhalb von OAuth 2 und wird zur Verifizierung von Benutzeridentitäten benutzt
 - OAuth 2 definiert 4 Rollen
 - Resource owner : Person oder Anwendung, der die Resourcen gehören, auf die Zugriff benötigt wird
 - Resource server : der Server, der die Resourcen hosted
 - Client : die Anwendung, die die Resource nutzen will
 - Authorization server : Server, der den Client für die Nutzung der Resource autorisiert
 - Authorization Server und Resource Server können der gleiche Server sein

g) Einige Risiken von Webanwendungen

- Das Open Web Application Security Project (OWASP) stellt auf Ihrer Homepage (<https://www.owasp.org>) jede Menge nützlicher Informationen zur Verfügung – unter anderem eine Liste der Top Sicherheitsrisiken von Webanwendungen
- Risiken von Webanwendungen
 - Injection
 - nicht vertrauenswürdige Daten werden einem Interpreter geschickt und unvorhergesehene Befehle werden ausgeführt
 - bekanntestes Beispiel : SQL injection (SQLi)
 - SQL Befehle in Benutzerdaten versteckt, die dann aus Versehen am Server ausgeführt werden
 - ein Web Applications Firewall (WAF) kann das anhand von Signaturen größtenteils (nicht alle) erkennen und ausfiltern
 - Methoden wie Validating input, SQL parameter und das Prinzip “least privilege” kann die Gefahr reduzieren

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - Broken authentication
 - eine Schwachstelle im Authentifizierungs- oder Sessionmanagement kann die Security eines Software Systems gefährden
 - Angreifer versuchen zuerst manuell Schwachstellen zu finden, um sie anschließend mit Toolunterstützung auszunutzen
 - hashing, multi-factor Authentifizierung, Secure by default und Installationen ohne Nutzung von Defaultrechten helfen das Risiko dieser Art von Attacken zu minimieren
 - Passwort Policies und das zyklische Erneuern von Passwörter erhöhen die Sicherheit
 - Anwendung sollten gezielte Log-outs unterstützen, um den Einfall von Hackern bei Timeouts von Sessions noch nicht abgemeldeter Benutzer zu verhindern
 - Session ID's sollten nicht in der URL erkennbar sein und nicht zwischen Sessions rotieren
 - Informationen wie Passwörter, Tokens, Session ID's und andere Berechtigungen sollten immer über sichere Verbindungen ausgetauscht werden

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - Sensitive data exposure
 - nicht richtig abgesicherte vertrauliche Informationen wie Sozialversicherungs- oder Kreditkarten-Nummer, Berechtigungen u.ä.
 - Sensitive Daten nur speichern, wenn notwendig und sobald als möglich entfernen
 - Sensitive Daten, die längerfristig gespeichert sind, sollten verschlüsselt sein (auch alle Backups)
 - Sensitive Daten, die übertragen werden, müssen während der Übertragung verschlüsselt sein
 - Starke und moderne Algorithmen sollten zusammen mit einem entsprechenden Schlüsselmanagement benutzt werden
 - Browser Direktiven und Header können die Sicherheit erhöhen
 - vermeiden von Cachen von sensitiven Daten

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - XML external entity (XXE) attack
 - greift Parser, die XML parsen an
 - wenn XML Input Referenzen auf externe Ressourcen enthält und der Parser nicht entsprechend konfiguriert, kann dies zu Sicherheitslücken führen
 - Denial of Service (DoS) und Server-Side Request Forgery (SSRF) (Verfälschung) können mit einer XXE Attacke ausgelöst werden
 - eine der effektivsten Methoden XXE Angriffe zu vermeiden, ist die Wahl anderer Datenformate wie JSON
 - komplette Deaktivierung von document type definitions (DTD) ist auch effektiv
 - ist das nicht möglich, müssen die Nutzung externer Entities oder Entity expansion verhindert werden
 - die letzten gültigen Security patches und fixes sollten eingespielt sein

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - XML external entity (XXE) attack, Beispiel (billion laughs attack oder XML bomb)

```
<?xml version=„1.0“?>
<!DOCTYPE lolz [
  <!ENTITY lol „lol“>
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol1 „&lol;&lol; &lol;&lol; &lol;&lol; &lol;&lol; &lol;&lol;“>
  <!ENTITY lol2 „&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;“>
  <!ENTITY lol3 „&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;“>
  <!ENTITY lol4 „&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;“>
  <!ENTITY lol5 „&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;“>
  <!ENTITY lol6 „&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;“>
  <!ENTITY lol7 „&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;“>
  <!ENTITY lol8 „&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;“>
  <!ENTITY lol9 „&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;“>
]>
<lolz&lol9;</lolz>
```

- wird zu einer Milliarde Entitäten aufgebläht und sprengt alle Puffer und erzeugt damit einen DoS

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - Broken access control
 - eine weit verbreitete Sicherheitsbedrohung
 - kann manuell oder mit Tools entdeckt werden und kann zur Benutzung von nicht autorisierten Benutzungsrechten führen
 - auch wenn die Benutzeroberfläche in allen Punkten sicher ist, können Angreifer versuchen die Server zu attackieren, indem sie die URL, den Anwendungsstatus oder Tokens ändern oder Requests gezielt verändern, um Zugriff auf nicht autorisierte Funktionen zu bekommen
 - Auf Client- und Serverseite zu prüfen
 - Zugriffstokens so schnell wie möglich ungültig machen
 - Deny by Default Ansatz anwenden
 - Access Control Fehler protokollieren und bei erhöhter Anzahl unverzüglich und automatisch melden
 - Softwaretests sollten solche Attacken beinhalten

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - Security misconfiguration
 - oft der Fall
 - je größer und komplexer die Anwendung, um so höher die Wahrscheinlichkeit
 - die Anwendung muss vor der Installation auf Security geprüft sein
 - Default Werte vermeiden
 - Teile von Anwendungen, die nicht benötigt oder nicht benutzt werden, sollten nicht installiert sein
 - Beispiele : Accounts, Berechtigungen, Ports, Services
 - Alle Passwörter von Default Accounts sollten geändert oder die Accounts gelöscht sein
 - Viele Anwendungen nutzen eine Vielzahl von Frameworks und Tools
 - alle diese Hilfsmittel müssen beherrscht werden und richtig konfiguriert sein
 - Die Fehlerbehandlung sollte nicht zu detailliert sein und vor allem keine Informationen ausgeben, die zu einem Angriff genutzt werden können (z.B. detaillierter Stacktrace)
 - Aktuelle Patches und Fixes sind eingespielt
 - oft sind Probleme in Patches gelöst, aber nicht installiert

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - Cross-site scripting (XSS)
 - erlaubt Angreifern das ausführen von Scripts im Browser
 - Scripts können User Sessions übernehmen, Content ersetzen oder Benutzer umlenken (redirect)
 - sehr geläufige Methode
 - 3 Haupttypen
 - Reflected XSS
 - » eine Anwendung übernimmt nicht verauenswürdige Daten und schickt sie ohne Überprüfung an den Browser
 - Stored XSS
 - » nicht überprüfte Daten werden zur späteren Anzeige gespeichert
 - DOM XSS
 - » Daten oder Code kontrolliert durch einen Angreifer wird dynamisch durch ein Script eingefügt
 - Überprüfungen vermeiden nicht validierte und unsichere Eingaben
 - Server erkennen (untrusted HTTP requests) und beenden entsprechende Attacken

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - Unsecure deserialization
 - Attacken gegen serialized Objekte
 - kann zur Ausführung von nicht gewolltem Code führen
 - Datenstrukturen mit Zugriffsrechten können verändert werden
 - Serialized Objekte nur von vertrauenswürdigen Quellen akzeptieren
 - alle Exceptions loggen

g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
 - Komponenten nutzen, die bekannt unsicher sind
 - externe Bibliotheken oder Systeme
 - auf Wartung und Support achten
 - nicht ausreichendes Logging und Monitoring
 - ein wichtiges Risiko bei der Security Betrachtung
 - Angreifer wollen solange wie möglich unentdeckt bleiben
 - gutes Logging und Monitoring verhindert das
 - typische Fragestellung : wer/was/wann
 - kontrolliertes Log-Management notwendig
 - Auswertungen
 - Information an entsprechende Verantwortliche
 - auch Logs müssen gesichert sein
 - Angreifer könnten Spuren verwischen oder Informationen abgreifen

g) Einige Risiken von Webanwendungen

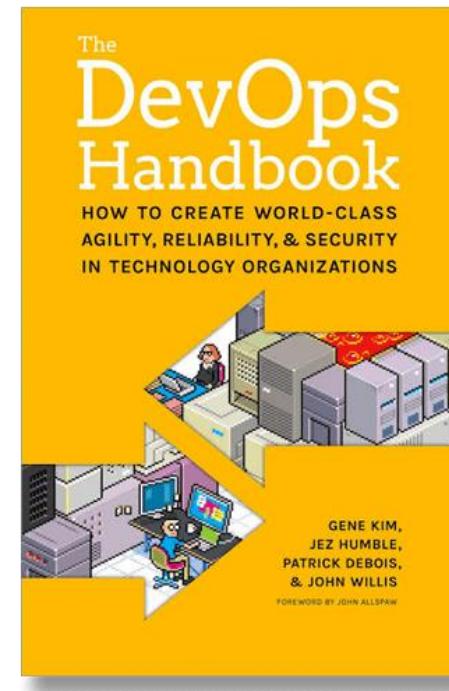
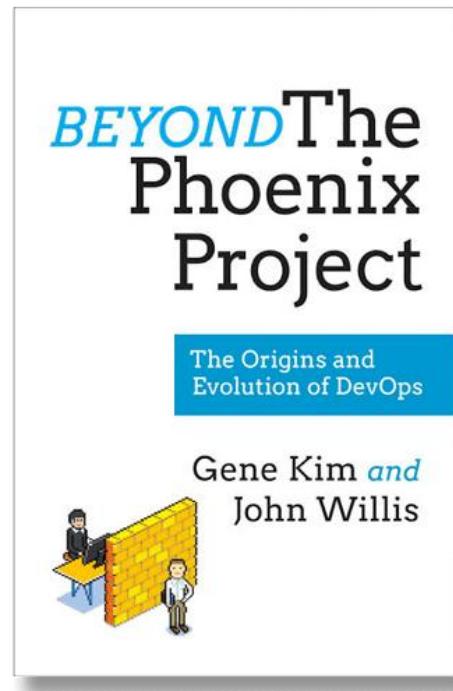
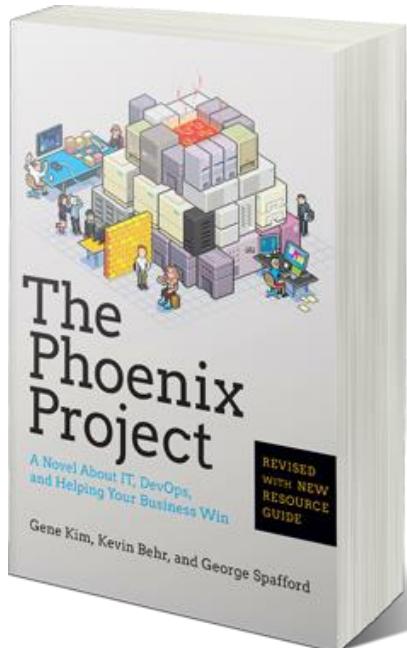
- Risiken von Webanwendungen
 - Unvalidated redirects and forwards
 - Weiterleitungen können zum Weiterleiten auf unsichere Webseiten benutzt werden
 - Redirects und Forwards nicht nutzen
 - Alle Redirects sollten über eine Seite gehen, die den Benutzer darüber informiert – der Benutzer muss die Weiterleitung bestätigen

12) DevOps und Softwarearchitektur

- DevOps ist die Kombination von kulturellen Werten, Praktiken und Tools für eine schnelle Softwareentwicklung
- Software Architekten sollten damit vertraut sein
- DevOps Praktiken kann Architekturen beeinflussen
- Wir betrachten
 - a) DevOps
 - b) DevOps Toolkette
 - c) DevOps Praktiken
 - d) DevOps und Architektur
 - e) DevOps und Cloud

a) DevOps

- DevOps ist ein Satz von Tools, Praktiken und zugehöriger Kultur, um Softwareentwicklerteams mit Betriebsteams während des ganzen Lebenszyklus eines Software Systems zusammen zu bringen
- Weiterführende Literatur



a) DevOps

- DevOps verbindet die Collaboration zwischen Teams mit der Automatisierung von Prozessen, um das gemeinsame Ziel, zusätzlichen Mehrwert für die User Experience zu erzeugen, zu erreichen indem Software schnell und mit hoher Qualität erstellt und geliefert wird
- DevOps setzt eine kulturelle Veränderung innerhalb von Organisationen voraus, im Zusammenspiel mit der Nutzung neuer Technologien
- *Shift left* ist ein bekannter Ausdruck bei DevOps und meint die Idee Aufgaben früher im Lebenszyklus durchzuführen oder sie nach links auf der Zeitschiene zu schieben
- DevOps hat sich die *shift left* Mentalität zu eigen gemacht und versucht in jeder Phase Aktivitäten nach links zu schieben
- z.B. sollte die Betriebsmannschaft *shift left* von Anfang an in alle Entwicklungsschritte eingebunden sein
- Weitere Beispiele für *shift left* sind Continuous Integration, bei dem die Integration und das Erstellen von Changes nach links geschoben wird und Continuous Delivery, bei dem das Ausrollen von Änderungen in Produktion viel schneller beim Endkunden ankommt

a) DevOps

- CALMS
 - repräsentiert die Kernwerte von DevOps
 - Culture
 - Automation
 - Lean
 - Measurement
 - Sharing

a) DevOps

- CALMS
 - Culture
 - im Inneren ist DevOps Kultur und Philosophie
 - die Änderung der Kultur, das Aufbrechen von Barrieren zwischen Teams in einer Organisation ist die Grundlage
 - DevOps führt sog. cross-funktionale Teams ein und bringt Mitarbeiter mit unterschiedlichen Skills zusammen
 - die DevOps Kultur schätzt Lernen (auch aus Fehlern) und jede Art von Wissen für Verbesserungen zu nutzen
 - Verantwortlichkeit ist auch Teil der DevOps Kultur. Wenn Fehler gemacht werden, übernehmen Teams die Verantwortung und arbeiten an einer Lösung, ohne zuerst Schuldige zu suchen
 - Qualität steht immer im Fokus von DevOps und deshalb Teil der Kultur
 - Mitarbeiter werden zu eigenen Entscheidungen befähigt und ermächtigt und in Entscheidungen einbezogen

a) DevOps

- CALMS
 - Automation
 - Automatisierung bei DevOps bedeutet, daß alle manuellen Schritte automatisiert werden und damit nachvollziehbar immer wieder gleich ablaufen können (wiederholbar)
 - automatische Prozesse garantieren höhere Konsistenz und Genauigkeit, im Vergleich zu manuellen Prozessen
 - automatische Prozesse können zu jeder Zeit ablaufen (auch zu Zeiten an denen niemand anwesend ist)
 - kann für eine bessere Auslastung von Resourcen sorgen
 - automatische Prozesse laufen i.a. schneller ab und sind weniger fehleranfällig
 - automatische Prozesse liefern schneller Feedback über Probleme oder notwendige Änderungen

a) DevOps

- CALMS
 - Lean
 - Lean Software Development kommt vom Lean Manufacturing und hat einige “best practices” von dort übernommen
 - das Ziel ist die Optimierung von Prozessen und die Minimierung von unnötiger Arbeit
 - die 7 Lean Prinzipien
 - eliminate waste
 - build quality in
 - create knowledge
 - defer commitment
 - deliver fast
 - respect people
 - optimize the whole
 - passen gut mit den Zielen von DevOps zusammen

a) DevOps

- CALMS
 - Measurement
 - Meßbarkeit ist wichtig, denn ohne Meßbarkeit ist kein Fortschritt nachweisbar
 - Architekten sind von Maßzahlen abhängig, ansonsten hätten sie nur ihr Bauchgefühl übrig ☺
 - Maßzahlen oder gemessene Werte müssen transparent für alle Beteiligten verfügbar sein
 - Sharing
 - alle Ressourcen wie Tools oder auch Methoden, Ideen, Wissen und Erfahrungen müssen mit allen anderen geteilt werden
 - nur so kann man von Fehlern oder von anderen lernen
 - meistens der erste Schritt zur Änderung der Unternehmenskultur

a) DevOps

- Warum DevOps ?
 - Als Software Architekt will man immer Software schneller, öfter und mit wenigen Fehler ausliefern
 - Andere Ansätze als DevOps haben nicht so den Fokus auf Continuous Delivery und benötigen deshalb länger bis der Wert der erstellten Lösungen beim Kunden ankommt
 - Wenn der Betrieb nicht in die Entwicklung integriert wird, dann gibt es oft zusätzliche Wartezeiten bei der Bestellung und Installation der notwendigen Umgebungen
 - die Software ist schon fertig, aber die Umgebung steht noch nicht
 - die Bereitstellung der Umgebung ist nicht automatisiert
 - oft werden Hardware Ressourcen zu groß bestellt, weil das Entwicklungsteam nur einmal den hinderlichen Bestellprozess durchlaufen will (ein späteres Upgrade will man sich sparen)
 - um heute wettbewerbsfähig zu bleiben, müssen Lösungen schnell am Markt verfügbar sein, das gilt auch (oder ganz speziell) für Softwarelösungen
 - das geht nur, wenn man bereit ist, dem Kunden Teillösungen, die später erweitert werden, anzubieten
 - in das Design der Erweiterungen wird der Kunde eingebunden

a) DevOps

- Warum DevOps ?
 - die Einbindung des Kunden in den Entwicklungsprozess sorgt für besseres Verständnis des Anforderungen und für eine höhere Zufriedenheit beim Kunden
 - Kunden haben heute weniger Verständnis für Ausfälle als früher
 - Deshalb wird es immer wichtiger, daß Fehler schnell entdeckt und behoben und neue Versionen oder Releases sofort automatisiert ausgerollt werden können
 - Softwarelösungen werden immer größer und komplexer. DevOps, zusammen mit entsprechenden Prinzipien und Pattern, kann für eine Reduzierung der Komplexität sorgen
 - DevOps sorgt für einen Wandel in der Firmenkultur und sorgt für die Zusammenarbeit unterschiedlichster Teams (Entwicklung, Betrieb, Security) bei der Erstellung von Kundenlösungen
 - Wissen, Erfahrung und Begeisterung wird ausgetauscht – Informationen sind transparent verfügbar

b) DevOps Toolchain

- Der DevOps Toolchain
 - Toolchains sind ein Satz von Tools, die in Kombination zur Erstellung von Lösungen verwendet werden
 - Ein DevOps Toolchain ist auf die Entwicklung und das Ausrollen von Softwarelösungen fokussiert
 - Software Architekten sorgen für die Auswahl und den konsistenten Einsatz von entsprechenden Tools
 - es gibt immer noch die Microsoft und Java Fraktionen mit unterschiedlichen Glaubensrichtungen
 - der Betrieb einer Lösung ist in den Toolchain eingebunden
 - Automatisierungsskripte oder Software sollte in den Entwicklungszyklus eingebunden sein (Sourcecontrol u.ä.), ist also Teil der Entwicklung (Infrastructure as Code)

c) DevOps Praktiken

- 3 wichtige DevOps Praktiken
 - Continuous Integration (CI)
 - Source Control
 - Entwickler legen alle Entwicklungen und Änderungen so oft wie möglich in einem gemeinsamen Source Control System ab
 - Ein Art der Versionierung ist dabei unbedingt notwendig
 - » Semantic Versioning : Major.Minor.Patch = größere Änderungen.Erweiterungen.Bugfixes
 - Entwickler sollten in bestimmten Zyklen ihre Änderungen bestätigen (commiten)
 - dies erlaubt Konflikte früh zu erkennen
 - automatisierte Builds
 - alle Commits werden automatisch zur benutzbaren Gesamtlösung zusammengefügt und erzeugt
 - Schritte dahin könnten sein : Prüfen von Abhängigkeiten, Kompilieren von Sources, Paketierung, Erzeugen von Installern, Erzeugen/Update von Datenbank Schemas, Aufruf von Skripts zur Konfiguration, automatisierte Tests
 - die Laufzeit von Builds sollte 20 min nicht überschreiten
 - Architekten sollten für eine CI Umgebung sorgen
 - automatisierte Tests sorgen für das Erkennen von Problemen während der Builds

c) DevOps Praktiken

- 3 wichtige DevOps Praktiken
 - Continuous Delivery (CD)
 - ist die Fähigkeit, Änderungen schnell, wiederholbar und nachvollziehbar auszurollen
 - Releasezyklen sind kurz und reduzieren Kosten, Risiken und Aufwände
 - ist die Fähigkeit, eine Lösung jederzeit in einem produktionsfähigen Status verteilen zu können
 - fasst CI, automatisches Testen (technisch und user acceptance) und das Ausrollen zusammen (manuell)
 - Continuous Deployment
 - CD mit zusätzlichem automatiserten Ausrollen der Lösung
 - verschiedene Staging Areas wie Test, Consolidation, Production werden unterstützt

d) DevOps und Architektur

- Software Architekten müssen DevOps in ihrer Architektur berücksichtigen
 - DevOps beeinflusst die Qualitätsattribute
 - Testability
 - ganz wichtig, da hoch automatisiert
 - Maintainability
 - um Zwischenstände schnell ausrollen zu können, muß das Softwaresystem einfach wartbar sein
 - Reduzierung von Komplexität erfordert eine hohe Aufmerksamkeit
 - geringer Komplexität erlaubt kürzere Zyklen
 - Deployability
 - die verschiedenen Stages sollten sehr ähnlich aufgebaut oder gleich sein
 - Konfigurationsdaten sollten von der Anwendung getrennt vorliegen

d) DevOps und Architektur

- Architektur Pattern, die DevOps erleichtern
 - Microservices
 - schmale kleine Services mit definiertem Interface
 - jeder Microservice sollte einzeln austauschbar und ausrollbar sein
 - jeder Microservice hat seinen eigenen Datenpool
 - Microservices unterstützen hohe Fehlerisolierung

e) DevOps und Cloud

- viele der Vorteile von DevOps werden auf Cloudplattformen direkt unterstützt
- Cloudtypen können sein
 - Public Cloud (AWS, MS Azure)
 - hohe Verfügbarkeit und unbegrenzte Skalierbarkeit
 - Datensicherheit muss beachtet werden (Sensitive Daten)
 - Private Cloud
 - ausschließlich von einer Organisation benutzt
 - kann bei einem Provider oder im eigenen Rechenzentrum sein
 - höhere Kosten und eingeschränkte Skalierbarkeit
 - höhere Datensicherheit realisierbar

e) DevOps und Cloud

- Cloudtypen können sein
 - Hybrid Cloud
 - Kombination von Public und Private Cloud
 - Nutzung der jeweiligen Vorteile
 - Übergangsphasen (Transitions) sind einfacher
 - Notfallszenarien können geplant werden

e) DevOps und Cloud

- Cloudmodelle
 - Infrastructure as a Service (IaaS)
 - Betrieb von Hardware in der Cloud (gängiger Start oder Eintritt in Cloudplattformen)
 - Containers as a Service (CaaS)
 - Betrieb von virtuellen Umgebungen (VM oder Container) in der Cloud
 - Kubernetes, Docker Swarm, Apache Mesos
 - Platform as a Service (PaaS)
 - komplette Plattform zur Entwicklung und zum Ausrollen von Lösungen
 - beliebige Programmierumgebungen und –sprachen werden unterstützt
 - Kontrolle über das Betriebssystem geht verloren

e) DevOps und Cloud

- Cloudmodelle
 - Serverless Function as a Service (FaaS)
 - ähnlich PaaS
 - Verrechnungsmodell verschieden (nach Ausführung nicht nach Hosting)
 - Software as a Service (SaaS)
 - Cloud Software über das Internet (MS Email, MS Office, Salesforce)

13) Evolutionäre Architekturen

- Viele Softwareanwendungen stehen unter dauerndem Druck geändert und angepasst zu werden
- Software Architekten müssen Anwendungen planen, die diesen Anforderungen genügen
- Lange Planungszeiten vor der Realisierung von Anwendungen gehören der Vergangenheit an
- Moderne (evolutionäre) Anwendungen müssen für die später (sicher) kommenden Änderungen vorbereitet sein
- Wir betrachten
 - a) Änderungen sind unausweichlich
 - b) Lehmann's Gesetze der Software Evolution
 - c) Evolutionäre Architekturen entwerfen

a) Änderungen sind unausweichlich

- mit (Ver)Änderungen muss von Anfang an geplant werden
- Softwaresysteme sind keine statischen Systeme mehr und können nicht geplant kontrolliert werden
- Änderungen können kaum vorhergesehen werden
- moderne Softwaresysteme richten sich an realen Bedingungen aus und sind damit deren Veränderungen unterworfen
- Änderungsanforderungen können aus technischen Gründen entstehen oder aus Anforderungen, die vom Business kommen
- die Vielzahl an aktuellen Technologien lassen mehr Möglichkeiten zu und können deshalb den Wunsch nach Änderung zusätzlich treiben
- Änderungen im Business wie Zukäufe, Verkäufe, neue Märkte, neue Konkurrenten, neue Produkte geschehen in immer kürzeren Zeiträumen
- Erwartungen/Anforderungen von Benutzern werden größer und ändern sich öfter
- Architekturen müssen anpassbare und jederzeit änderbare Lösungen zum Ziel haben

b) Lehmann's Gesetze der Software Evolution

- In seinem Papier “Programs, Life Cycles, and Laws of Software Evolution” beschreibt Lehmann 3 verschiedene Typen von Systemen
 - **S-type** Systeme
 - unterscheidbar mit genauer Spezifikation
 - kann formal beschrieben werden und die Lösung ist einfach verständlich
 - vollkommen zutreffende Lösungen können bereitgestellt werden
 - Anforderungen ändern sich selten und entwickeln sich nicht weiter
 - Lehmann's Gesetz ist für S-Type Systeme nicht anwendbar
 - **P-type** Systeme
 - Probleme sind genau festgesetzt
 - das Resultat ist bekannt und eine Spezifikation ist erstellbar
 - im Gegensatz zu S-type Sytemen ist aber die Lösung nicht verstanden oder unpraktisch umzusetzen
 - die Komplexität der Logik ist so hoch, daß eine Umsetzung keinen Sinn macht
 - Lehmann's Gesetz ist für P-Type Systeme nicht anwendbar

b) Lehmann's Gesetze der Software Evolution

- In seinem Papier “Programs, Life Cycles, and Laws of Software Evolution” beschreibt Lehmann 3 verschiedene Typen von Systemen
 - E-type Systeme
 - System ist realitätsnah (Prozesse und Menschen) modelliert
 - die meisten Softwaresysteme sind E-Type Systeme
 - E-type Systeme beeinflussen ihre Umgebung, in die sie eingebettet sind
 - die Umgebung löst Veränderungsdruck auf E-Type Systeme aus
 - E-type Systeme müssen sich verändern, um benutzbar zu bleiben
 - Lehmann's Gesetz ist für E-Type Systeme anwendbar

b) Lehmann's Gesetze der Software Evolution

- Gesetz I : Continuing change
 - Softwaresysteme müssen durch einen andauernden Veränderungsprozess gehen, sonst werden sie unbrauchbar
 - die Zufriedenheit von Endusers wird abnehmen, wenn das System die sich verändernden Anforderungen nicht mehr abdecken kann
- Gesetz II : Increasing Complexity
 - mit der Zeit erhöht sich durch die vorgenommenen Änderungen die Komplexität des Systems, obwohl Anstrengungen zur Reduzierung der Komplexität unternommen wurden
 - Software Entropy (abgeleitet aus dem 2. Gesetz der Thermodynamik) sorgt für Unordnung durch die Erhöhung von Modifikationen in einem System
- Gesetz III : Self-regulation
 - die Entwicklung eines Systems ist selbst regulierend
 - es gibt strukturelle und orgab'nisatorische Faktoren, die für eine Regulierung sorgen
 - je größer und komplexer eine System ist, um so schwieriger können Änderungen umgesetzt werden
 - komplizierte Abstimmungen in Organisationen behindern zusätzlich

b) Lehmann's Gesetze der Software Evolution

- Gesetz IV : Conservation of organizational stability
 - über den Lebenszyklus eines Systems ist die Weiterentwicklungsrate nahezu constant und unabhängig von den eingesetzten Ressourcen
- Gesetz V : Conservation of familiarity
 - über den Lebenszyklus eines Systems muss immer der gleiche Level an Know-How vorhanden sein
 - Wissen über technische und funktionale Aspekte muss erhalten werden
- Gesetz VI : Continuing growth
 - wenn sich ein Softwaresystem weiter entwickelt, wird es wachsen
 - die Anzahl der Funktionen wird wachsen und damit die technischen Implementierungen
 - steht in Beziehung zu Gesetz II, da Größe in Beziehung zu Komplexität steht
- Gesetz VII : Declining quality
 - wenn sich ein Softwaresystem weiterentwickelt, wird seine Qualität abnehmen wenn nicht zusätzliche Aufwände in die Qualitätsverbesserung investiert werden
 - mehr Code erhöht die Wahrscheinlichkeit von Defekten und Ausfällen

b) Lehmann's Gesetze der Software Evolution

- Gesetz VIII : Feedback system
 - Software Weiterentwicklung ist ein komplexer Prozess und benötigt Rückmeldungen von verschiedenen Stakeholdern, um sicher zu stellen, daß die Weiterentwicklung in die richtige Richtung geht und Mehrwert liefert
 - Änderungen werden aus Rückmeldungen abgeleitet

c) Evolutionäre Architekturen entwerfen

- Wie können Software Architekten ihre Architekturen so entwickeln, daß sie auf Änderungen vorbereitet sind
- Evolutionäre Architekturen lassen sich selbst einfach ändern und erweitern
- Evolutionäre Architekturen unterstützen Modifikationen am System ohne die Architektur verändern zu müssen

“An evolutionary architecture supports guided incremental change across multiple dimensions.”

- aus dem Buch : Building Evolutionary Architectures (Rebecca Parsons, Neal Ford, Patrick Kua)

c) Evolutionäre Architekturen entwerfen

- Software Architekten sollten alle Änderungen an Software Systemen führen, damit die Charakteristiken der Architektur erhalten bleiben
 - Alle Design Entscheidungen und alle Qualitätsattribute
- Wenn sich eine Architektur, ausgelöst durch technische Änderungen oder Änderungen im Business, anpassen muss, dann müssen die Konsequenzen dieser Anpassungen berücksichtigt werden und in eine neue, angepasste Architektur einfließen
- Architekturen müssen vielleicht in einem Maß verändert werden, das nicht verhersehbar war
- Architekten müssen deshalb aktiv durch diesen Änderungsprozess führen
- Architekten nutzen sog. Fitness functions um die Auswirkungen von Modifikationen an Software Systemen zu bewerten

c) Evolutionäre Architekturen entwerfen

- Fitness functions
 - eine Funktion, um festzustellen, wie nah sich eine Lösung am gewünschten Ziel befindet
 - sie bestimmen die Fitness einer Lösung
 - Fitness functions können auch zur Überprüfung der Designcharakteristik von Software Architekturen verwendet werden
 - Fitness functions helfen die Fitness einer Lösung vor und nach der Realisierung zu messen und zu bestimmen
 - Sie können auch bei der Auswahl geeigneter Lösungen helfen

c) Evolutionäre Architekturen entwerfen

- Kategorien von Fitness functions
 - atomare (fokussiert auf eine Architekturcharakteristik)
 - ganzheitliche (auf mehrere Charakteristika fokussiert)
 - ausgelöste (ausgelöst durch einen Event)
 - kontinuierliche (dauernd laufende)
 - statische (der Wert einer Bedingung bleibt constant)
 - dynamische (sich verändernde Bedingungen)
 - automatische (automatisch ausgelöst, z.B. in einem automatischen Build Prozess)
 - manuelle (manuell ausgelöst)
 - zeitlich begrenzte
 - bewusste (schon früh bekannte)
 - plötzlich auftretende (z.B. während der Entwicklung)
 - domänen-spezifische (in Beziehung zu einer Business Domäne, z.B. Regeln oder Sicherheit)

c) Evolutionäre Architekturen entwerfen

- Beispiele von Fitness functions
 - Fitness functions können in Form von Tests entstehen, aber nicht alle Tests sind Fitness functions, sondern nur diejenigen, die Architekturcharakteristika betrachten
 - andere Fitness functions können z.B. die Wartbarkeit betrachten
 - Security Test können die Fitness eines Systems in Bezug auf Sicherheit prüfen
 - Andere Beispiele für Fitness functions kommen aus dem Chaos Engineering (Chaos Monkey)
 - um systematisch die Stärke (oder Schwäche) eines System zu messen
 - ein Tool, das von Netflix entwickelt wurde, um die Auswirkung von Ausfällen auf das Gesamtsystem zu messen
-

c) Evolutionäre Architekturen entwerfen

- Evolutionäre Architekturen unterstützen Änderungen über mehrere Dimensionen
 - Programmiersprachen
 - Frameworks
 - Datenbanken
 - Security
 - Performance
- lose Kopplung als wichtiger Pfeiler für evolutionäre Architekturen
- das Design erweiterbarer API's ist ein weiterer Aspekt evolutionärer Architekturen
 - Postels Gesetz (das Prinzip der Robustheit)
 - konservativ, was man selbst tut und liberal, was man von anderen akzeptiert
- in evolutionären Architekturen liegt ein Schwerpunkt auf der Nutzung von Standards
- Verlagern von Entscheidungen auf den letztmöglichen Zeitpunkt (last responsible moment)
 - schwierig zu bestimmen
 - Kosten der Verschiebung dürfen Kosten der Entscheidung nicht übertreffen

14) Die Skills eines Softwarearchitekten

- Neben technischen Skills sollte jeder Architekt auch eine Reihe von Softskills besitzen
- Softskills sind für den Erfolg eines Architekten wichtiger als technische Skills
- Softskills machen den Unterschied zwischen Architekten und besonderen Architekten aus
- Wir betrachten
 - a) was sind Softskills
 - b) Kommunikation
 - c) Führung
 - d) Verhandlungsgeschick

a) Was sind Softskills

- HardSkills sind konkret und können definiert und gemessen werden
- sie sind typischerweise aufgaben- oder job-spezifisch
- sie können mit Training (intern oder extern) und durch Lernen und Lesen verbessert werden
- bei Software Architekten kann das die Beherrschung einer Programmiersprache oder den geeigneten Einsatz von Frameworks beinhalten
- Softskills sind schwerer zu fassen, schlecht zu definieren und kaum zu messen
- Softskills beziehen sich mehr auf Beziehungen zu anderen, wie Führung, Kommunikation, Zuhören, Empathie, Gesprächsführung und Geduld
- Auch wenn man Softskills trainieren kann, sind sie doch mehr angeboren oder in der Person verwurzelt
- Software Architekten profitieren vom Besitz, der Anwendung und dem Auf- und Ausbau von Softskills
- Softskills spielen eine große Rolle bei der Arbeit von Architekten
- die wichtigsten Softskills werden wir betrachten

b) Kommunikation

- Kommunikationsfähigkeit ist wahrscheinlich eine der wichtigsten Fähigkeiten eines Architekten
- Stakeholdern die Architektur oder Architekturvorschläge vermitteln zu können, ist überlebensnotwendig
- um etwas kommunizieren zu können, muß man die verstehen können und das Zielpublikum genau kennen
- dies hilft bei der Auswahl der geeigneten Hilfsmittel und Methoden zur Weitergabe von Informationen
- meistens ist die Art, wie man etwas sagt, wichtiger als was man sagt
- manchmal geschieht die Kommunikation auf rein technischer Ebene, manchmal auf fachlicher Ebene
- ein guter Architekt richtet seine Kommunikation am Zielpublikum aus
- Architekturen müssen an die Entwickler, die Projektteilnehmer und an die Nutzer kommuniziert werden
 - technische Aspekte, Programmervorgaben, Designentscheidungen u.ä. and die Programmierer
 - Qualitätsattribute wie Performance, Usability, Security, Entscheidungen u.ä. and die Anwender
 - Resourceplanung, Zeitplanung u.ä. an das Projekt
- zusätzlich wird evtl. das Management dauernd über den Fortschritt informiert werden
- gute Kommunikation zwischen allen Beteiligten erhöht die Erfolgsaussichten und verhindert Überraschungen

b) Kommunikation

- die 7 C's der Kommunikation (Tips für effective Kommunikation)
 - Klarheit (Clarity)
 - Klarheit sorgt für Verständnis und effective Kommunikation
 - Einbeziehung der Zuhörer und die Wahl der Sprache unterstützen Klarheit
 - Prägnanz (Conciseness)
 - Weniger ist oft mehr, Zuhörer lieben Prägnanz
 - zu viele Worte beeinträchtigen Klarheit und den Inhalt
 - Prägnanz leitet auf den eigentlichen Inhalt, indem unnötige Dinge weggelassen werden
 - Prägnanz spart Zeit und Geld
 - Konkretheit (Concreteness)
 - spezifische statt vage Aussagen
 - bringt Klarheit in Aussagen und verhindert Mißverständnisse
 - macht Aussagen für Zuhörer interessanter
 - lebhafte, active Sprache fesselt Zuhörer

b) Kommunikation

- die 7 C's der Kommunikation (Tips für effective Kommunikation)
 - Höflichkeit (Courteousness)
 - zeigt Respekt gegenüber den Zuhörern
 - unterstützt bei der Auf-/Annahme der Nachricht durch andere
 - verstärkt Beziehungen
 - berücksichtigt kulturelle, religiöse und ethnische Unterschiede
 - Rücksicht (Consideration)
 - hat die Empfänger der Nachricht im Sinn
 - legt den Schwerpunkt auf "Du" nicht "Wir" und "Ich"
 - berücksichtigt die Blickpunkte der Empfänger
 - Richtigkeit (Correctness)
 - Aussagen müssen richtig sein (keine fake news)
 - die Wortwahl muss das unterstützen
 - unterstützt das Vertrauen der Zuhörer in die Nachricht

b) Kommunikation

- die 7 C's der Kommunikation (Tips für effective Kommunikation)
 - Höflichkeit (Courteousness)
 - zeigt Respekt gegenüber den Zuhörern
 - unterstützt bei der Auf-/Annahme der Nachricht durch andere
 - verstärkt Beziehungen
 - berücksichtigt kulturelle, religiöse und ethnische Unterschiede
 - Rücksicht (Consideration)
 - hat die Empfänger der Nachricht im Sinn
 - legt den Schwerpunkt auf "Du" nicht "Wir" und "Ich"
 - berücksichtigt die Blickpunkte der Empfänger
 - Richtigkeit (Correctness)
 - Aussagen müssen richtig sein (keine fake news)
 - die Wortwahl muss das unterstützen
 - unterstützt das Vertrauen der Zuhörer in die Nachricht

b) Kommunikation

- die 7 C's der Kommunikation (Tips für effective Kommunikation)
 - Vollständigkeit (Completeness)
 - alle notwendigen Informationen sind enthalten
 - nichts wird verschwiegen
 - Hilfreich dabei die 5 W-Fragen (Who, What, Where, When, Why)
 - in der Nachricht sind alle relevanten Antworten zu den Fragen
 - Vollständigkeit ist die Grundlage für gute Entscheidungen
 - spart Zeit und Kosten

b) Kommunikation

- Zuhören
 - richtig zuhören zu können ist enorm wichtig und die Grundlage für Verstehen
 - Kommunikation geht immer in 2 Richtungen
 - Nachrichten können verloren gehen, wenn nicht richtig zugehört wird
 - Hören ist nicht Zuhören, Aufmerksamkeit ist notwendig
 - Einfühlungsvermögen in die Person gegenüber unterstützt das Verstehen
 - Mehr zuhören, weniger reden und Verständnisfragen stellen, zeigen Einfühlungsvermögen
 - die Methode “Effective Listening” kann geübt werden
- Präsentationen halten
 - diese Fähigkeit muss stark ausgeprägt sein und immer wieder geübt werden

b) Kommunikation

- Präsentationen halten
 - diese Fähigkeit muss stark ausgeprägt sein und immer wieder geübt werden
 - die 4 P's von Präsentationen
 - Plan
 - gut geplant ist halb gewonnen
 - Ziele, Zuhörer, Inhalt, Art, Dauer
 - Prepare
 - Aufteilung, Humor, Inhalt, Art, Aufteilung, Design, Methodik
 - Practice
 - Üben, Test
 - Present
 - Dress, erster Eindruck, Augenkontakt, Begeisterung, Lautstärke, keine Panik, positive Aussage am Schluß

c) Führung

- Andere mitreißen, inspirieren oder positiv beeinflussen können
- Respekt, Vertrauen und Glaubwürdigkeit erlangen
- immer hilfsbereit sein
- mit Herausforderungen umgehen können
- technisches Vorbild sein können
- Visionen haben und vermitteln können
- Innovationen vorantreiben
- Verantwortung übernehmen und andere mit einbeziehen
- andere unterstützen und voranbringen
- delegieren können
- Änderungen nicht fürchten sondern vorantreiben
- Experimentierfreude zeigen
- richtig kommunizieren
- ander anleiten (Mentoring)
- Walk the talk
- mit gutem Beispiel vorangehen
- andere einbeziehen

d) Verhandlungsgeschick

- bei Verhandlungen aktiv unterstützen
- Meinungsverschiedenheiten ausräumen
- Einigungen anstreben und erarbeiten
- kommt oft mit der Erfahrung
- Alternativen kennen und betrachten
- auch auf andere Alternativen vorbereitet sein

15) Wie werde ich ein guter Softwarearchitekt

- Softwareentwicklung und Software Architektur ist ein sich dauernd änderndes Tätigkeitsfeld
- Software Architekten müssen immer auf der Höhe der Zeit sein, aktuelle Trends kenn und sich entsprechend weiterbilden
- Wir betrachten
 - a) Continuous learning
 - b) Teilnahme an Open Source Projekten
 - c) den eigenen Blog betreiben
 - d) sich Zeit nehmen, andere zu schulen
 - e) neue Technologien ausprobieren
 - f) selbst entwickeln
 - g) an User Groups und Konferenzen teilnehmen

Wie werde ich ein guter Softwarearchitekt

- Continuous learning
 - Ein guter Architekt erweitert dauernd sein Wissensportfolio
 - Architekten müssen sich wie Softwaresysteme an Veränderungen anpassen
 - Architekten erhalten ihren Wert, indem sie Ihr Wissen immer auf dem aktuellen Stand halten
 - sie sind unzufrieden mit ihrem Wissen und begierig darauf, neues zu lernen
 - sie vertiefen ihr Wissen in Bereichen, in denen sie noch nicht so gut sind
 - die Technik, Methoden und Konzepte ändern sich – ein guter Architekt kennt sich aus
 - gute Architekten lösen sich von Detailwissen hin zu mehr Überblick und Wissen in der Breite
 - wenn Architekten zu sehr auf einzelne Technologien fokussiert sind, besteht die Gefahr, daß diese Technologien zu sehr in den Fokus rücken und immer wieder empfohlen werden oder Grundlage von Architekturen sind
 - Law of the instrument oder law of the hammer oder Maslow's hammer
 - “*wenn das einzige Werkzeug, das Du hast, ein Hammer ist, sieht alles aus wie ein Nagel!*”
 - gute Architekten finden die Zeit und nehmen sich die Zeit, um zu lernen
 - Bücher sind immer eine gute Quelle

Wie werde ich ein guter Softwarearchitekt

- Open Source Projekt
 - die Teilnahme an Open Source Projekten kann der Firma und dem Architekten weiter helfen
 - Wissen und Ansehen (Marktwert)
- den eigenen Blog betreiben
 - über technische Themen in Blogs zu schreiben, kann das Wissen und die Fähigkeit, komplexe Themen darzustellen, erweitern
 - Blogs helfen dabei, die “Sichtbarkeit” nach außen zu erhöhen (Marktwert)
- sich Zeit nehmen, andere zu schulen
 - interne Kurse anbieten
 - Vorlesungen halten
 - Mentor für andere sein
 - was man schulen kann, hat man selbst auch verstanden
- neue Technologien ausprobieren
 - Vorsicht – nicht dem Spieltrieb fröhnen

Wie werde ich ein guter Softwarearchitekt

- selbst entwickeln
 - ein guter Softwarearchitekt kann auch selbst entwickeln und kennt die gängigen Methoden und Programmiersprachen
 - er kann mit Programmierern mitreden und kennt die gängigen Fachbegriffe
 - manchmal macht es Sinn in einem Projekt eine Teilaufgabe als Entwickler zu übernehmen
 - Code von anderen Programmierern lessen und verstehen lernen
- an Usergroups und Konferenzen teilnehmen
 - aktive Teilnahme
 - Vorträge halten
 - an Arbeitsgruppen teilnehmen
 - sich mit anderen austauschen
 - neue Leute kennenlernen (Marktwert)

16) Architektur und Legacy Applications

- Auch wenn es viele neue Plattformen, Umgebungen und Technologien gibt und jeden Tag neue dazu kommen, wird sich jeder Architekt mindestens einmal im Leben mit einer Legacy Application beschäftigen müssen
- Software Architekten müssen Legacy Applications deshalb beherrschen können
- Wir betrachten
 - a) Was ist eine Legacy Application
 - b) Ändern von Legacy Applications
 - c) Migration in die Cloud
 - d) Anwenden von agilen Methoden
 - e) Build und Deployment modernisieren
 - f) Legacy Applications integrieren (in andere Lösungen)

Architektur und Legacy Applications

- Was ist eine Legacy Application
 - eine Legacy Application ist eine Anwendung, die noch in Benutzung aber schwer zu warten ist
 - auch wenn es mehr Spaß macht, sich mit neuen Technologien zu beschäftigen und neue Systeme auf der grünen Wiese zu planen (Greenfield Ansatz), wird die eine oder andere bereits existierende (Legacy) Anwendung irgendwann auf dem Tisch des Architekten landen
 - die neuen Anwendungen von heute sind die Legacy Anwendungen von morgen
 - Probleme mit Legacy Anwendungen
 - keine Architektur, nicht mehr zeitgemäß, schlecht zu warten, keine Dokumentation vorhanden, Programmierer in Rente, Designentscheidungen nicht bekannt, Anforderungen unbekannt
 - Software Entropie ist weit fortgeschritten, Spaghetti code, alte (nicht mehr unterstützte) Versionen von Software und Betriebssystem
 - fachliche Anforderungen haben sich verändert
 - einzige noch lauffähige Anwendung, die ein wichtiges Problem löst
 - Anwendung, die andere Anwendungen mit Daten versorgt oder Grundlage für andere Anwendungen ist

Architektur und Legacy Applications

- Ändern von Legacy Applications
 - klassisches Buch : *Refactoring: Improving the Design of Existing Code* von Martin Fowler
 - *“...the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure”*
 - die Business Logik und die Funktionalität bleiben erhalten
 - folgende Schritte sind notwendig
 - den Legacy Code testbar machen
 - unit Test einführen
 - redundanten Code entfernen
 - unerreichbar, tot, auskommentiert, doppelt
 - Tools benutzen
 - IDE kann oft dabei helfen
 - kleine, inkrementelle Änderungen planen und vornehmen
 - Monolithen in Microservices umarbeiten

Architektur und Legacy Applications

- Migration in die Cloud
 - um Legacy Applications zu modernisieren, könnte auch die Migration in die Cloud ein geeignetes Mittel sein
 - Es gibt Gründe, warum eine Migration Sinn machen könnte wie z.B.
 - Kosteneinsparung bei HW/SW, Betrieb, Patches
 - höhere Verfügbarkeit und Skalierbarkeit
 - die 6 R der Migration
 - Remove (oder Retire)
 - bei jeder Migration fallen immer Anwendungen an, die eingestellt werden können
 - Retain
 - es gibt immer Anwendungen, die in ihrer Umgebung bleiben müssen (techn., organisator., finanziell, politisch)
 - Replatform
 - manche Anwendungen werden zu IaaS Providern migriert
 - Rehost
 - ein externer Provider hostet die Anwendung

Architektur und Legacy Applications

- Migration in die Cloud
 - die 6 R der Migration
 - Repurchase
 - manchmal macht es Sinn die Anwendung durch eine neue Standardanwendung zu ersetzen
 - Refactor (Rearchitect)
 - das ist die anspruchsvollste Variante, hier können aber alle Vorteile der neuen Plattform genutzt werden
 - » Verfügbarkeit, Resilienz, Skalierbarkeit, Performance, weltweite Verteilung, Kosteneinsparungen, neue Verrechnungsmodelle (pay per usage)
 - in den meisten Fällen ist eine neue Architektur notwendig

Architektur und Legacy Applications

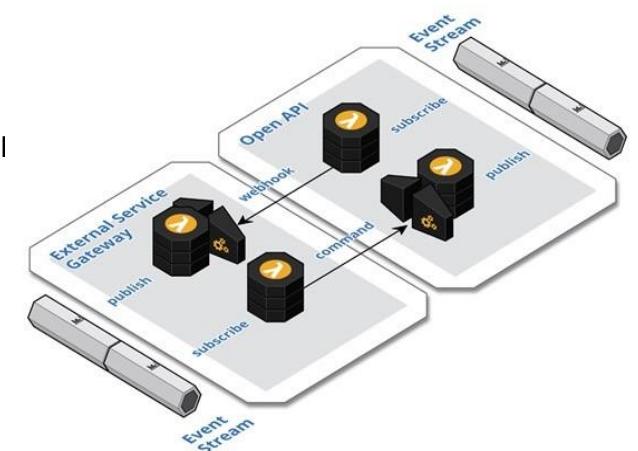
- Anwenden von agilen Methoden
 - ein Teil der Modernisierung einer Legacy Application könnte auch der Einsatz agiler Entwicklungsmethoden sein
 - agile Methoden können immer dann von Vorteil sein, wenn der Schwerpunkt bei der Anwendung mehr auf Marktorientierung und schnelle/dauernde Änderungen liegt, statt auf Optimierung von längerfristig benutzten Funktionalitäten
 - der Einsatz von agilen Methoden bedingt allerdings ein Umdenken in den Entwicklungsteams
 - der Einsatz von Microservices unterstützt den Ansatz agiler Methoden

Architektur und Legacy Applications

- Build und Deployment modernisieren
 - oft werden Legacy Applications mit veralteten und manuell gesteuerten Build- und Deployment Prozessen betrieben
 - manchmal steuern Skripte diesen Prozess, die entweder plattformspezifisch oder unübersichtlich und ohne Dokumentation sind
 - die Umstellung auf moderne Prozesse und Tools bringt immer Vorteile
 - eine Schwerpunkt sollte auf die Automatisierung dieser Prozesse gelegt werden (s. DevOps Kapitel)
 - die schrittweise Einführung von Continuous Integration und Continuous Delivery haben weiteres Optimierungspotential

Architektur und Legacy Applications

- Legacy Applications integrieren
 - Anwendungen haben oft verschiedenste Integrationen mit anderen Anwendungen
 - können Legacy Applications nicht komplett neu aufgebaut werden (z.B. Refactor), kommen erprobte Patterns für die Integration von Anwendungen zum Einsatz
 - wir hatten in Kapitel 8 das Pattern “External Service Gateway”
 - Integration mit externen Systemen über die Kapselung der ein- und ausgehenden Kommunikation zu anderen Systemen über eine isolierte Komponente als Brücke zum Austausch von Events
 - Das erlaubt die Einbeziehung von Legacy Systemen in cloud-basierte, asynchrone, message-orientierte Umgebungen
 - weitere Integrationen könnten über gemeinsam genutzte Daten und/oder Funktionen realisiert werden





Dokumentation

1. Einführung und Ziele
2. Randbedingungen
3. Kontextabgrenzung
4. Lösungsstrategie
5. Bausteinsicht
6. Laufzeitsicht
7. Verteilungssicht
8. Querschnittliche Konzepte
9. Entwurfsentscheidungen
10. Qualitätsszenarien
11. Risiken und technische Schulden
12. Glossar



Anforderungen
Technik
Strukturen/Sichten
Entscheidungen

Einführung und Ziele

- Beschreibt die wesentliche Anforderungen und treibenden Kräfte, die Softwarearchitekten und Entwicklungsteams berücksichtigen müssen.
Dazu gehören die
 - zugrunde liegenden Geschäftsziele, wesentliche Aufgabenstellung und essenzielle fachliche Anforderungen an das System
 - Qualitätsziele für die Architektur
 - relevante Stakeholder und deren Erwartungshaltung



Dokumentation

Einführung und Ziele



Geschäftsziele

Aufgabenstellungen

Fachl.
Anforderungen

- Inhalt
 - Kurzbeschreibung der Geschäftsziele und der fachlichen Aufgabenstellung, treibenden Kräfte, Extrakt (oder Abstract) der Anforderungen. Verweis auf vielleicht vorhandene Anforderungsdokumente (mit Versionsbezeichnungen und Ablageorten).
- Motivation
 - Aus Sicht der späteren Nutzer ist die Unterstützung einer fachlichen Aufgabe oder Verbesserung der Qualität der eigentliche Beweggrund, ein neues System zu schaffen oder ein bestehendes zu modifizieren.
- Form
 - Kurze textuelle Beschreibung, eventuell in tabellarischer Use-Case Form. Sofern vorhanden sollte die Aufgabenstellung Verweise auf die entsprechenden Anforderungsdokumente enthalten.
 - Halten Sie diese Auszüge so knapp wie möglich und wägen Sie Lesbarkeit und Redundanzfreiheit gegeneinander ab.

Einführung und Ziele



Qualitätsziele

- Inhalt
 - Die Top-3 bis Top-5 der Qualitätsziele für die Architektur, deren Erfüllung oder Einhaltung den maßgeblichen Stakeholdern besonders wichtig sind. Gemeint sind hier wirklich Qualitätsziele, die nicht unbedingt mit den Zielen des Projekts übereinstimmen. Beachten Sie den Unterschied.
- Motivation
 - Weil Qualitätsziele grundlegende Architekturentscheidungen oft maßgeblich beeinflussen, sollten Sie die für Ihre Stakeholder relevanten Qualitätsziele kennen, möglichst konkret und operationalisierbar.
 - Wenn Sie als Architekt nicht wissen, woran Ihre Arbeit gemessen wird,
- Form
 - Tabellarische Darstellung der Qualitätsziele mit möglichst konkreten Szenarien, geordnet nach Prioritäten.

Einführung und Ziele



Stakeholder

- Inhalt
 - Expliziter Überblick über die Stakeholder des Systems, d.h. über alle Personen, Rollen oder Organisationen, die die Architektur kennen sollten oder von der Architektur überzeugt werden müssen, mit Architektur oder Code arbeiten (z.B. Schnittstellen nutzen), Dokumentation der Architektur für ihre eigene Arbeit benötigen, Entscheidungen über das System und dessen Entwicklung treffen.
- Motivation
 - Sie sollten die Projektbeteiligten und -betroffenen kennen, sonst erleben Sie später im Entwicklungsprozess Überraschungen. Diese Stakeholder bestimmen unter anderem Umfang und Detaillierungsgrad der von Ihnen zu leistenden Arbeit und Ergebnisse.
- Form
 - Tabelle mit Rollen- oder Personennamen, sowie deren Erwartungshaltung bezüglich der Architektur und deren Dokumentation.

Rolle	Kontakt	Erwartungshaltung
<Rolle-1>	<Kontakt-1>	<Erwartung-1>
<Rolle-2>	<Kontakt-2>	<Erwartung-2>

Randbedingungen



- Inhalt
 - Fesseln und Vorgaben, die ihre Freiheiten bezüglich Entwurf, Implementierung oder Ihres Entwicklungsprozesses einschränken. Diese Randbedingungen gelten manchmal organisations- oder firmenweit über die Grenzen einzelner Systeme hinweg.
- Motivation
 - Als Architekt sollten Sie explizit wissen, wo Ihre Freiheitsgrade bezüglich Entwurfsentscheidungen liegen und wo Sie Randbedingungen beachten müssen. Sie können Randbedingungen vielleicht noch verhandeln, zunächst sind sie aber da.
- Form
 - Einfache Tabellen der Randbedingungen mit Erläuterungen. Bei Bedarf unterscheiden Sie technische, organisatorische und politische Randbedingungen oder übergreifende Konventionen (beispielsweise Programmier- oder Versionierungsrichtlinien, Dokumentation- oder Namenskonvention).

Dokumentation

Kontextabgrenzung



- Inhalt
 - Die Kontextabgrenzung grenzt das System von allen Kommunikationspartnern (Nachbarsystemen und Benutzerrollen) ab. Sie legt damit die externen Schnittstellen fest.
 - Differenzieren Sie fachlichen Kontext (fachliche Ein- und Ausgaben) und technischen Kontext (Kanäle, Protokolle, Hardware), falls nötig.
- Motivation
 - Die fachlichen und technischen Schnittstellen zu Kommunikationspartnern gehören zu den kritischsten Aspekten eines Systems. Stellen Sie sicher, dass Sie diese komplett verstanden haben.
- Form
 - Verschiedene Optionen:
 - Diverse Kontextdiagramme
 - Listen von Kommunikationspartnern mit deren Schnittstellen

Kontextabgrenzung fachlich



- Inhalt
 - Festlegung aller Kommunikationspartner (Nutzer, IT-Systeme, ...) mit Erklärung der fachlichen Ein- und Ausgabedaten oder Schnittstellen. Zusätzlich bei Bedarf fachliche Datenformate oder Protokolle der Kommunikation mit den Nachbarsystemen.
- Motivation
 - Alle Beteiligten müssen verstehen, welche fachlichen Informationen mit der Umgebung ausgetauscht werden.
- Form
 - Alle Diagrammarten, die das System als Black Box darstellen und die fachlichen Schnittstellen zu den Nachbarn beschreiben.
 - Alternativ oder ergänzend können Sie eine Tabelle verwenden. Der Titel gibt den Namen Ihres Systems wieder; die drei Spalten sind:
Kommunikationspartner, Eingabe, Ausgabe.
<Diagramm und/oder Tabelle>
<optional: Erläuterung der externen fachlichen Schnittstellen>

Kontextabgrenzung technisch



- Inhalt
 - Technische Schnittstellen (Kanäle, Übertragungsmedien) zwischen dem System und seiner Umwelt. Zusätzlich eine Erklärung (mapping), welche fachlichen Ein- und Ausgaben über welche technischen Kanäle fließen.
- Motivation
 - Viele Stakeholder treffen Architekturentscheidungen auf Basis der technischen Schnittstellen des Systems zu seinem Kontext.
 - Insbesondere Infrastruktur- oder Hardwareentwickler entscheiden auch über diese technischen Schnittstellen.
- Form
 - Beispielsweise UML Deployment-Diagramme mit den Kanälen zu Nachbarsystemen, begleitet von einer Tabelle, die Kanäle auf Ein-/Ausgaben abbildet.
<Diagramm oder Tabelle>
<optional: Erläuterung der externen technischen Schnittstellen>
<Mapping fachliche auf technische Schnittstellen>

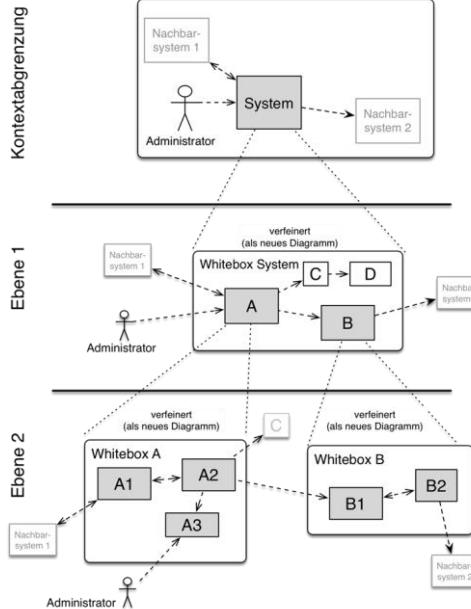
Lösungsstrategie



- Inhalt
 - Kurzer Überblick über die grundlegenden Entscheidungen und Lösungsansätze, die Entwurf und Implementierung des Systems prägen. Hierzu gehören:
 - Technologieentscheidungen
 - Entscheidungen über die Top-Level-Zerlegung des Systems, beispielsweise die Verwendung prägender Entwurfs- oder Architekturmuster
 - Entscheidungen zur Erreichung der wichtigsten Qualitätsanforderungen
 - relevante organisatorische Entscheidungen (Entwicklungsprozesse oder Delegation bestimmter Aufgaben an andere Stakeholder).
- Motivation
 - Diese Entscheidungen bilden wesentliche „Eckpfeiler“ der Architektur. Von ihnen hängen meistens viele weitere Entscheidungen oder Implementierungsregeln ab.
- Form
 - Fassen Sie die zentralen Entwurfsentscheidungen kurz zusammen. Motivieren Sie ausgehend von Aufgabenstellung, Qualitätszielen und Randbedingungen, was Sie entschieden haben und warum Sie so entschieden haben.

Dokumentation

Bausteinsicht



- **Inhalt**
 - Diese Sicht zeigt die statische Zerlegung des Systems in Bausteine (Module, Komponenten, Subsysteme, Klassen, Interfaces, Pakete, Bibliotheken, Frameworks, Schichten, Partitionen, Tiers, Funktionen, Makros, Operationen, Datenstrukturen...) sowie deren Beziehungen.
 - Diese Sicht sollte in jeder Architekturdokumentation vorhanden sein. In der Analogie zum Hausbau bildet die Bausteinsicht den Grundrissplan.
- **Motivation**
 - Behalten Sie den Überblick über den Quellcode, indem Sie die statische Struktur des Systems durch Abstraktion verständlich machen.
 - Damit ermöglichen Sie Kommunikation auf abstrakterer Ebene, ohne zu viele Implementierungsdetails offenlegen zu müssen.
- **Form**
 - Die Bausteinsicht ist eine hierarchische Sammlung von Blackboxen und Whiteboxen (siehe Abbildung unten) und deren Beschreibungen.

Laufzeitsicht



- Inhalt
 - Diese Sicht erklärt konkrete Abläufe und Beziehungen zwischen Bausteinen in Form von Szenarien aus folgenden Bereichen:
 - Wichtige Abläufe oder Features
 - Interaktionen an kritischen externen Schnittstellen
 - Betrieb und Administration: Inbetriebnahme, Start, Stop.
 - Fehler- und Ausnahmeszenarien
- Motivation
 - Verständnis der Aufgabenerfüllung und Kommunikation von Bausteinen des Systems
 - Bessere Kommunikation mit Stakeholdern (verständlicher als z.B. Bausteinsicht, Verteilungssicht)
- Form
 - Folgende Ausdrucksmöglichkeiten
 - Nummerierte Schrittfolgen oder Aufzählungen in Umgangssprache
 - Aktivitäts- oder Flussdiagramme
 - Sequenzdiagramme
 - BPMN oder EPKs (Ereignis-Prozessketten)
 - Zustandsautomaten
 - ...

Anmerkung: Kriterium für die Auswahl der möglichen Szenarien (d.h. Abläufe) des Systems ist deren Architekturrelevanz. Es geht nicht darum, möglichst viele Abläufe darzustellen, sondern eine angemessene Auswahl zu dokumentieren.

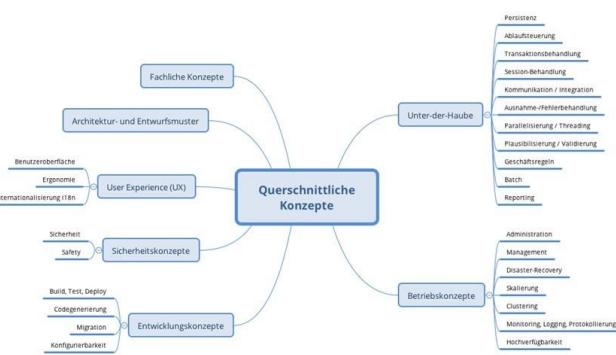
Verteilungssicht



- Inhalt
 - die technische Infrastruktur, auf der Ihr System ausgeführt (Standorte, Umgebungen, Rechnern, Prozessoren, Kanälen und Netztopologien, ...)
 - die Abbildung von (Software-)Bausteinen auf diese Infrastruktur
 - Darstellung von relevanten Entwicklung-/Test-/Produktionsumgebungen
 - Verteilung darstellen (Rechner, Prozessor, Server oder Container)
 - Schwerpunkt auf Softwaresicht
- Motivation
 - Die Infrastruktur sollte allen bekannt sein
 - Kombination von „Black-box“ mit weiteren Detailsichten
- Form
 - Diagramme
 - Sichten

Querschnittliche Konzepte

- Inhalt
 - übergreifende, prinzipielle Regelungen und Lösungsansätze
 - Themen
 - fachliche Modelle,
 - eingesetzte Architektur- oder Entwurfsmuster,
 - Regeln für den konkreten Einsatz von Technologien,
 - prinzipielle, meist technische, Festlegungen übergreifender Art,
 - Implementierungsregeln
- Motivation
 - Grundlage für konzeptionelle Integrität (Konsistenz, Homogenität) der Architektur
 - wesentliche Grundlage für die innere Qualität Ihrer Systeme
 - Platz zum behandeln spezieller Themen (wie z.B. "Sicherheit")
- Form
 - Konzeptpapiere mit beliebiger Gliederung, Modelle/Szenarien
 - Verweise auf Nutzung von Standardframeworks (z.B. Hibernate als Object/Relational Mapper)



Entwurfsentscheidungen



- Inhalt
 - Wichtige Architektur- oder Entwurfsentscheidungen mit Begründungen
 - Auswahl einer von mehreren Alternativen unter vorgegebenen Kriterien
 - Vermeiden Sie Redundanz
- Motivation
 - Stakeholder des Systems sollten wichtige Entscheidungen verstehen und nachvollziehen können.
- Form.
 - Liste oder Tabelle, nach Wichtigkeit und Tragweite der Entscheidungen geordnet
 - einzelne Unterkapitel je Entscheidung
 - ADR (Architecture Decision Record) für jede wichtige Entscheidung

Qualitätsszenarien



- Inhalt
 - Konkretisierung der Qualitätsanforderungen durch (Qualitäts-)Szenarien
 - Diese Szenarien beschreiben, was beim Eintreffen eines Ereignisses auf ein System in bestimmten Situationen geschieht
 - Nutzungsszenarien oder auch Anwendungs- oder Anwendungsfall-szenarien (Reaktionen zur Laufzeit, Effizienz oder Performance)
 - Änderungsszenarien (Modifikation des Systems oder seiner unmittelbaren Umgebung)
- Motivation
 - Szenarien operationalisieren Qualitätsanforderungen und machen deren Erfüllung mess- oder entscheidbar
 - Qualitätsszenarien müssen diskutierbar und nachprüfbar sein
- Form.
 - Entweder tabellarisch oder als Freitext

Risiken und technische Schulden



- Inhalt
 - Eine nach Prioritäten geordnete Liste der erkannten Architekturrisiken und/oder technischen Schulden
- Motivation
 - "Risikomanagement ist Projektmanagement für Erwachsene" (Tim Lister, Atlantic Systems Guild.)
 - Unter diesem Motto sollten Sie Architekturrisiken und/oder technische Schulden gezielt ermitteln, bewerten und Ihren Management-Stakeholdern (z.B. Projektleitung, Product-Owner) transparent machen
- Form
 - Liste oder Tabelle von Risiko und/oder technischen Schulden, eventuell mit vorgeschlagenen Maßnahmen zur Risikovermeidung, Risikominimierung oder dem Abbau der technischen Schulden.

Glossar



- Inhalt
 - Die wesentlichen fachlichen und technischen Begriffe, die Stakeholder im Zusammenhang mit dem System verwenden
 - Nutzen Sie das Glossar ebenfalls als Übersetzungsreferenz, falls Sie in mehrsprachigen Teams arbeiten
- Motivation
 - Sie sollten relevante Begriffe klar definieren, so dass alle Beteiligten
 - diese Begriffe identisch verstehen, und
 - vermeiden, mehrere Begriffe für die gleiche Sache zu haben
- Form
 - Zweispaltige Tabelle mit <Begriff> und <Definition>
 - Eventuell weitere Spalten mit Übersetzungen, falls notwendig

Anhang





The Digital Transformation “Book of Dreams”

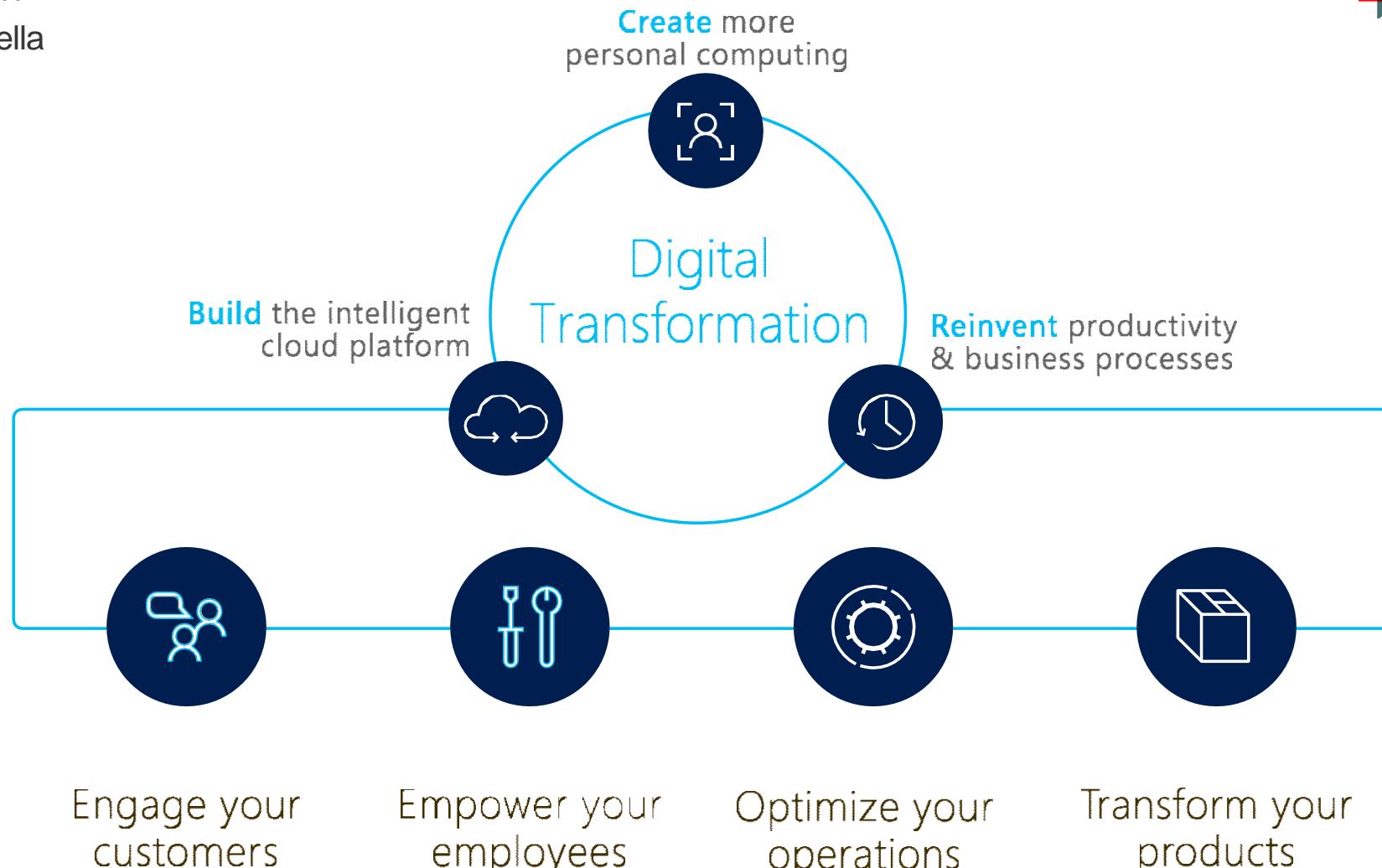
Digital Business Transformation Patterns
for a Mobile-First, Cloud-First World

Microsoft Internal – 12/19/2014



"At Microsoft, we're proud to partner with and empower all our customers around the world with the leading technology to seize the vast opportunities ahead."

– Satya Nadella



Imagine if...

...you could enable customers to connect with your business in ways that they choose, and deliver personalized experiences anywhere, anytime.

...if your employees could efficiently collaborate to meet rapidly changing customer needs and desires.

...you could broadly share information throughout the business, manage resources with agility, and better coordinate processes.

...you could use many channels to expand the reach of your business, better understand how customers use your products, and innovate quickly.



Vision for Digital Transformation

Imagine if you could...

Engage Your Customers

Deliver personalized, rich, connected experiences in journeys your customers choose.

Empower Your Employees

Keep up with your fast-moving customers, efficiently collaborating to anticipate and meet customer demands.

Optimize Your Operations

Increase the flow of information across your entire business operations, better manage your resources, and keep your business processes synchronized across all boundaries.

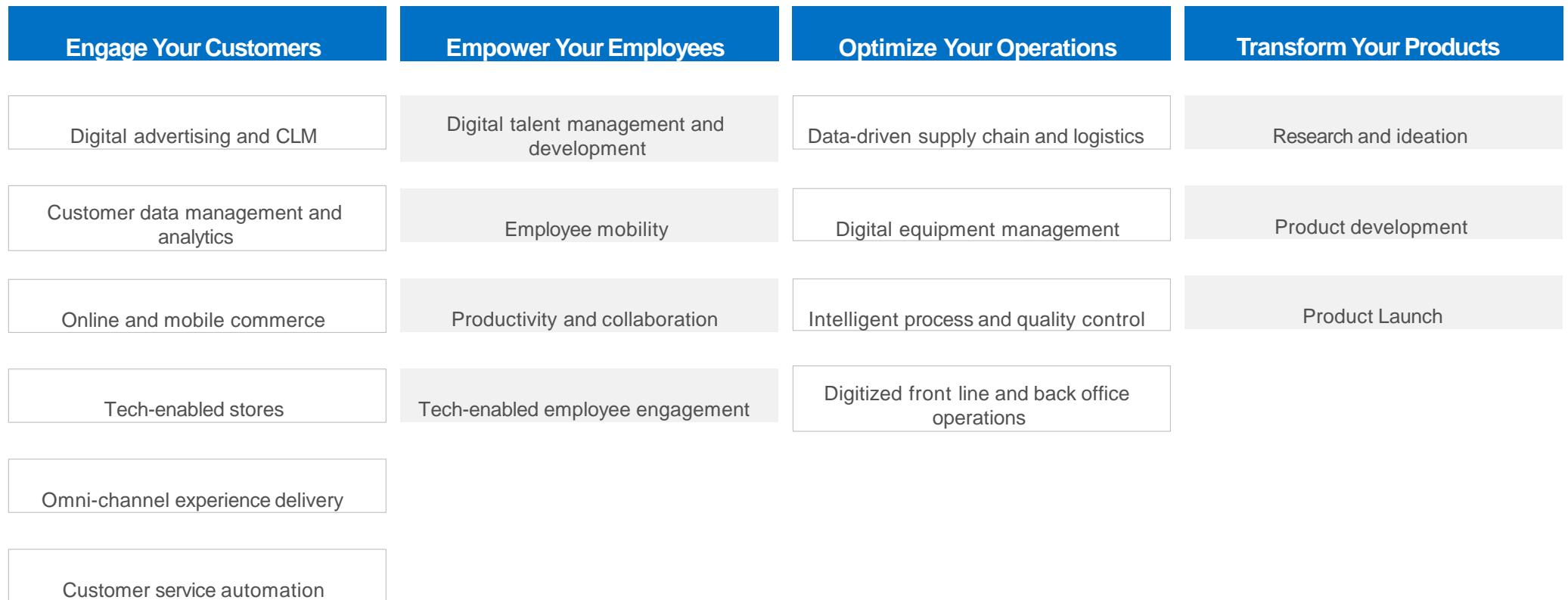
Transform Your Products

Expand the reach of your business using digital channels, anticipate customer needs, understand how your products are used, and quickly develop and improve products and services.

Business Drivers for Digital Transformation



Digital Transformation Capability Model



Capability Map

Provide Strategy & Direction	Develop Products & Services	Market & Sell Products & Services	Manage the Total Supply Chain	Provide Customer Service
Focus the Customer Value Proposition	Continually Assess Viability and Strategic Fit and Adapt the Development Portfolio	Gather and analyze data to Identify Markets, Segment Customers, and Leverage the Customer Base	Forecast Finished Goods and Raw Materials	Develop, Deploy, and Deliver the Customer Experience
Collect, communicate, and Understand the Customer, Market, Competitive and Regulatory Environment	Manage the Full Product Service Life Cycle	Communicate the Brand	Plan Materials and Production Demand and Capacity	Collaborate with Customers to Design/Build Products
Understand Customer Needs and Map to Strategic Direction	Design and Prototype New Products	Continually Adapt Product Packaging and Pricing	Acquire Materials, Services, and Products	Analyze Service Issues and Implement Process Improvements
Communicate and Manage to Goals and Measures	Pilot New Products and Product Refinements	Target Markets, Customers, and Segments and Execute Plans for Attraction, Acquisition, and Conversion	Develop and Operate and/or Acquire Manufacturing Capacity	Measure and Report Service and product Quality Delivery Performance
Develop and Direct Risk Mitigation and Management	Continually Improve Production Capacity and Performance		Optimize Supply Chain Performance	Assess Customer Satisfaction and Report Net Promoter Score
Develop and Direct Lean/Continuous Improvement <small>Moderne Software Architektur</small>	Manage Iterative Release/Certification Processes		Assure Product and Service Quality	Proactively Assure Product Satisfaction (Save Sales)
			Provide On-Site Assembly and Configuration	Communicate with Customers via any communication channel

Internal organization	Suppliers	Distribution channels	Partners	API interfaces
Structure the organization to include the new approach	Agree on the business objectives and respective contributions	Agree on the omni-channel framework	Develop partner eco-system strategy	Drive towards agile business strategy and business model
Assess organization culture and implement management of change	Define the development models and benefit sharing	Update the operational framework and objectives (interfaces)	Define the key topics (e.g. open innovations)	Include in company strategy as 2 ways (consume and expose)
Assess skill and address gaps	Refresh the supplier landscape	Define updated KPI	Identify key strategies with clients	Include API into the product and service development considerations
Update KPI, department and individual objectives to drive change	Measure value	Include consumers interactions	Define the boundaries	Explore opportunities
Ensure continuity and coexistence of the two models	Include consumer interactions	Measure value	Agree on the reward and benefit sharing	Measure value
			Measure value	Leverage a partnership eco-system

Digital Transformation Approach

Engage Your
Customers



Empower Your
Employees



Optimize Your
Operations



Transform Your
Products



- Am Beispiel
 - Engage your customers

Digital Transformation Book of Dreams

Engage Your Customers

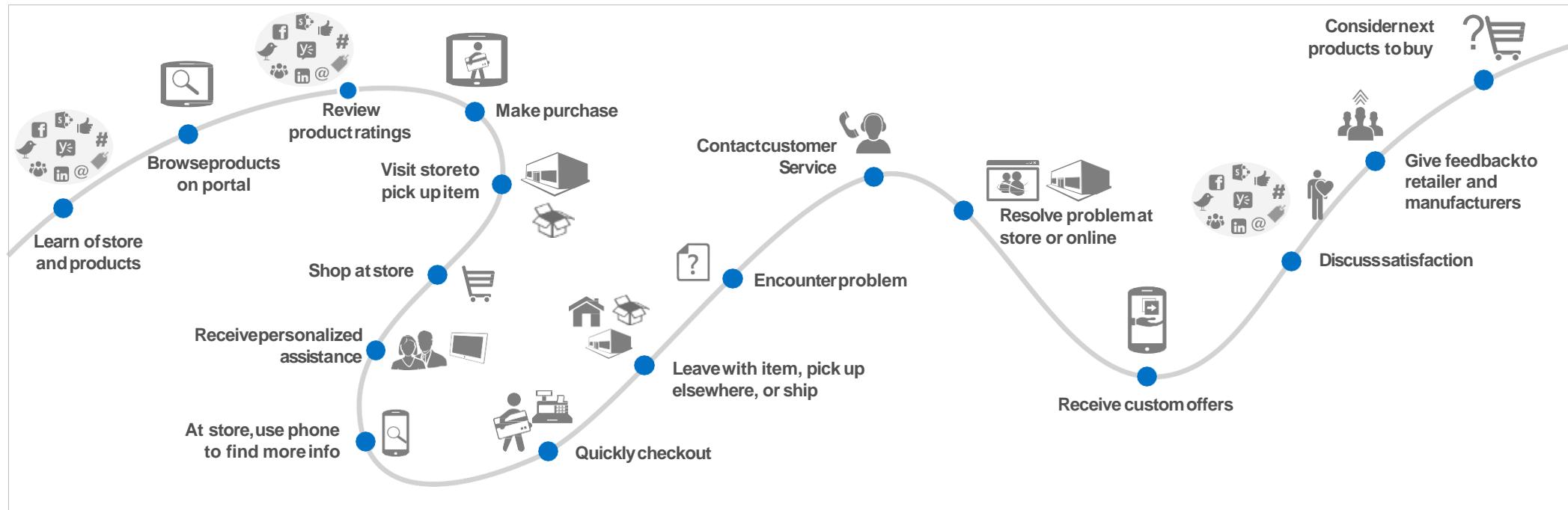


Imagine if...

...you could enable customers
to connect with your business
in ways that they choose, and
deliver personalized
experiences anywhere,
anytime.



Customer Experience Journey Map



Digital Hotspots

Pre-Purchase

- Learning of products and store
- Researching product online
- Talking to sales associates at store
- Use smartphone while at store to learn more

Moderne Software Architektur

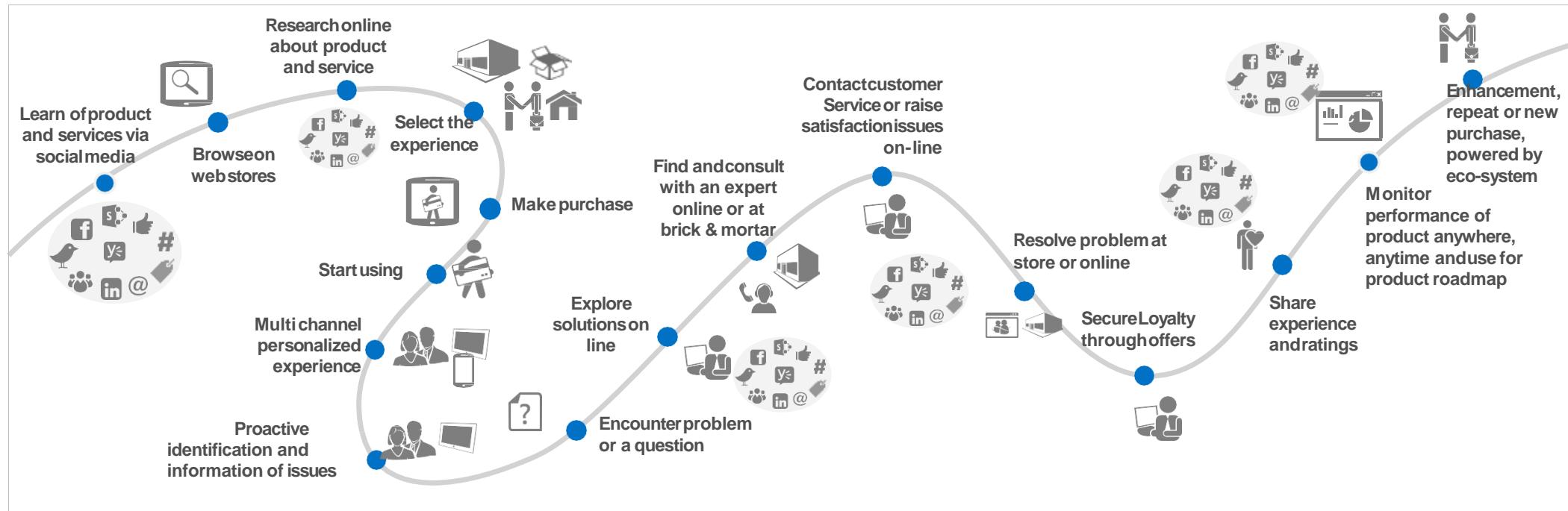
Purchase

- Ordering online
- Purchasing product in store
- Arranging payment
- Tracking shipment

Post-Purchase

- Receiving customer service
- Resolving problems anywhere
- Providing ideas to store and manufacturers
- Sharing experience with others

Customer Experience Journey Map



Digital Hotspots

Pre-Purchase

- Digital presence for products and store awareness
- Omni channel purchase
- Multi channel distribution and interactivity

Moderne Software Architektur

Peter Reinhardt
Version 1.0 Februar 2022

Purchase

- Omni channel, multi device solution for purchase
- Customer support
- Arranging payment
- Tracking shipment

Microsoft Internal – 12/19/2014

Post-Purchase

- Social network presence and management
- Receiving customer service
- Resolving problems anywhere
- Continuoud the experience
- Developing eco-system
- Product life cycle and organisations interlocks

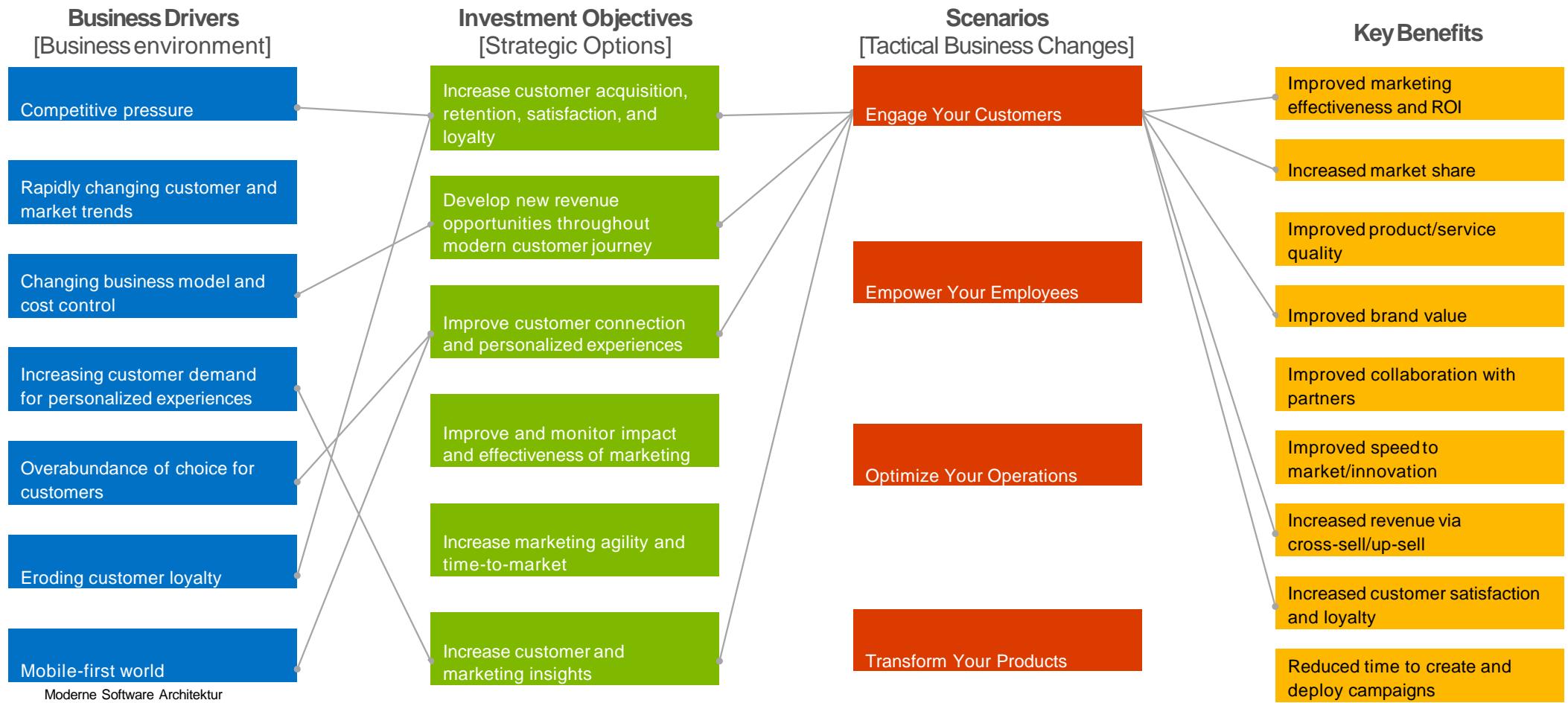
493

13

Engage Your Customers

Enterprise Scenarios Mapped to Benefits

Scenario Chain



Engage Your Customers

Understand and engage with customers to drive their buying decisions

Current State

We have difficulty finding ways to maximize reach and deliver personalized experiences that resonate with our customers.

Our customers are using more devices in more locations than ever before.

Our lack of ongoing engagement with our customers is affecting their loyalty.

We need to find interactive solutions that allow us to be a part of the moments that matter and modernize the customer experience.

Desired Future State

We engage with our customers in personalized and interactive ways, and deliver experiences and relevant content on any devices.

We react to real-time insights about each customer.

Our connection with each customer is inviting throughout the entire journey chosen by the customer

We have the solutions to connect to customers in the moments that matter.

Chief Marketing Officer

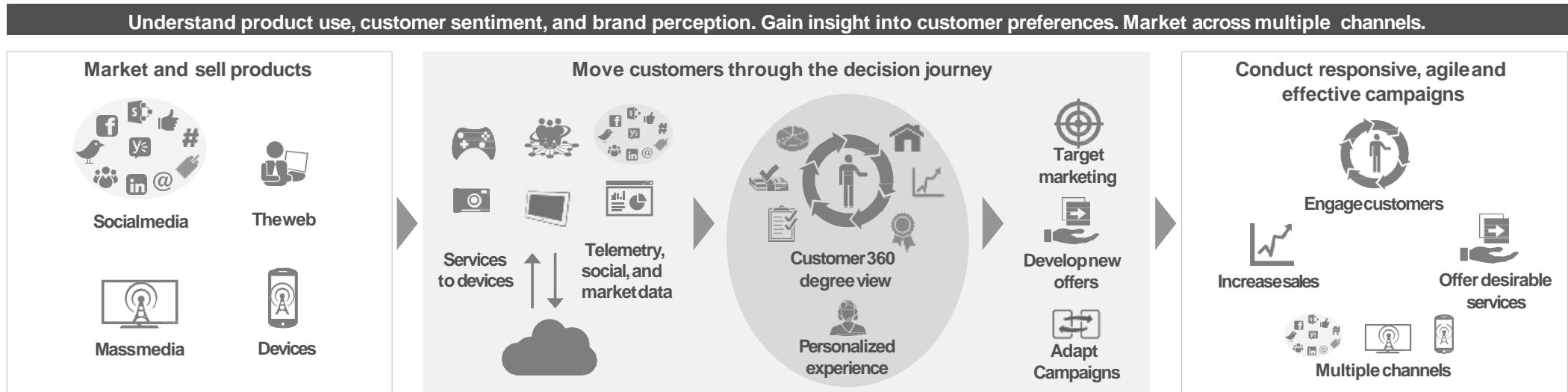


We are always looking to...

- Improve our marketing and sales effectiveness
- Reduce the cost of gaining new customers and maintaining customer loyalty
- Control our message in the market at all times and in all situations
- Drive each customer to buy across a range of products through cross-selling and up-selling

Engage Your Customers

Solution Storyboard



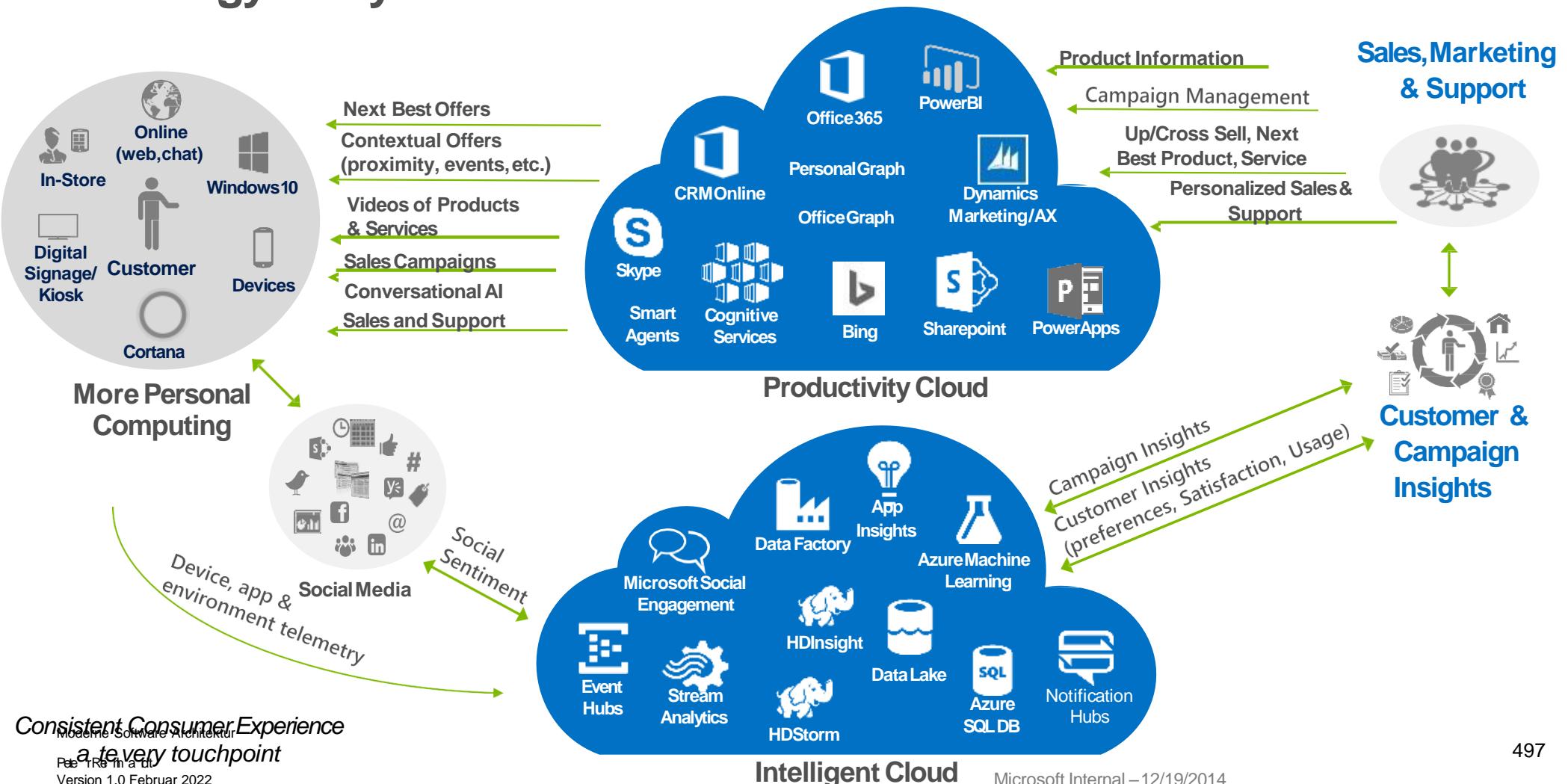
Top-line Impact

Enhance and improve customer engagement and generate ongoing revenue with personalized, interactive experiences throughout a customer's chosen journey.

Benefits	Start	Stop	Continue
<ul style="list-style-type: none"> Improved marketing, sales, and service effectiveness Increased market share Improved product and service quality Improved brand and shareholder satisfaction Improved speed to market 	<ul style="list-style-type: none"> Personalized marketing created with help of social listening analytics Evaluate device usage to identify new marketing opportunities Using customer analytics to connect customers with the most pertinent products and services Personalizing mobile marketing 	<ul style="list-style-type: none"> Restrict listening to only brand and not beyond (Competitors) Restrict sharing sentiment analysis data throughout the organization 	<ul style="list-style-type: none"> Perform customer and market intelligence analysis Develop and manage social media channels Develop and manage promotional activities

Engage Your Customers

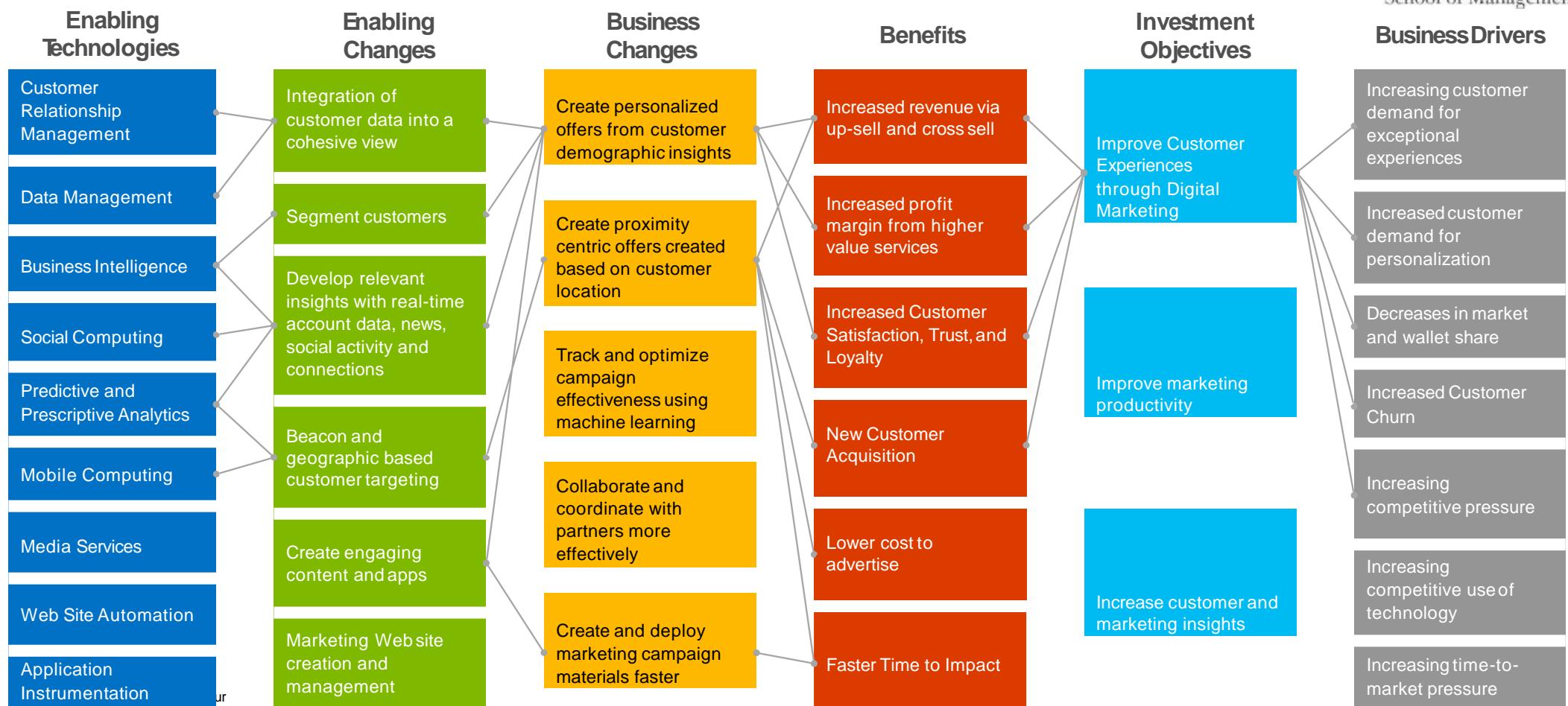
Technology Storyboard



Engage Your Customers

Benefits Dependency Network

Business Drivers

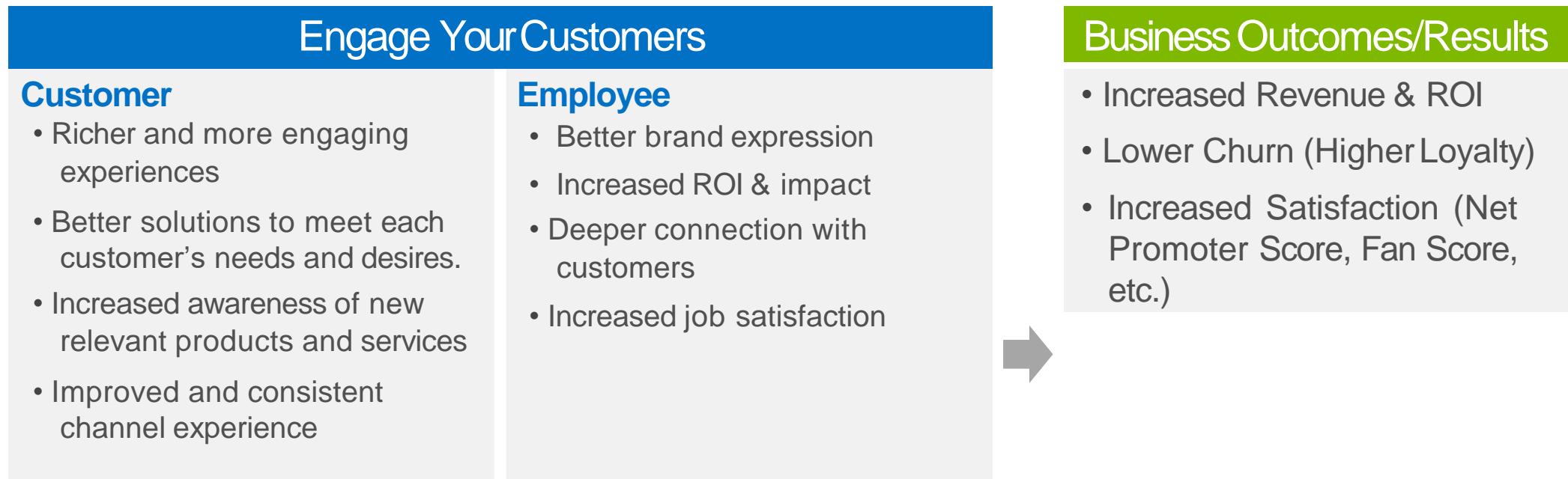


Engage Your Customers

Value Touchpoints

	Value for Customers	Value for Business Leaders	Value for IT Leaders
Strategic	<ul style="list-style-type: none"> Personalized solutions to meet each customer's needs and desires. Increased respect and relevance to the company leading to preferential treatment (loyalty programs) 	<ul style="list-style-type: none"> Increased understanding of customer's wants and needs Increased brand recognition and recall Increased customer engagement and loyalty 	<ul style="list-style-type: none"> Increase technology innovations Amplify the customer experience Improve channel integration
Time to Value	<ul style="list-style-type: none"> Geographic based offers provides awareness to customer and timeliness Reduced time researching 	<ul style="list-style-type: none"> Reduced time to sales conversion Reduced time to increase customer awareness Reduced time for offer/pricing experiment results Rapid offer experimentation to test ideas. 	<ul style="list-style-type: none"> Reduced time to gather, transform and process customer data Reduced time to create, manage and deploy campaign sites
Increase Revenue		<ul style="list-style-type: none"> Increased cross-sell/up-sell Increased customer wallet share Increased geographic proximity sale Acquisition of new customers 	
Decrease Cost	<ul style="list-style-type: none"> Loyalty based discounts Improve satisfaction levels 	<ul style="list-style-type: none"> Reduced capital expense Reduced customer acquisition costs 	<ul style="list-style-type: none"> Reduce cost of data analytics Reduce cost of data acquisition, storage, and management Reduced cost of creating, managing and deploying campaign sites
Decrease Risk	<ul style="list-style-type: none"> Understanding what's available with lower personal expense and avoiding a purchase made without full knowledge (i.e. research, etc.) 	<ul style="list-style-type: none"> Reduced failure of new product and service offerings by leveraging insights about customer preferences 	<ul style="list-style-type: none"> Increased automation, less risk for process failures

Continuously managing & measuring clear value levers, success factors, and performance measures



Engage Your Customers

Roadmap

Business Benefits

- Improved Up/Cross Sell
- Improved Customer/Fan Satisfaction
- Improved Market/Wallet Share
- Reduced Customer Churn

- Better understand and serve customers
- Improve business insight and workflow
- Increase sales

- Improve customer experience
- Improved employee collaboration
- Raise customer satisfaction
- Reduce operating costs

Phase 3: Enable predictive experiences and offers

- Predictive modelling for customers preferences and desires
- Real-time business and customer analytics, predictive models
- Customer specific offers driven by customer insights
- Prescriptive analytics
- Create engaging experiences and media

Phase 2: Gain Insight

- Determine customer journey and needs
- Model and analyse data for customer patterns and insights
- Collaborative and shared workspaces

Phase 1: Shared 360 degree view of customer

- Gather internal and external data about customers from all marketing channels and partners
- Enable gathering, storage, transforming, and distributing of Big Data
- Provide a centralized data warehouse and accompanying data marts that aggregate data from systems across the enterprise for reporting and analysis purposes

Time

Value

504

IT Projects

- Azure ML
- Data Modelling
- Big Data
- Universal Application
- Azure Media Services

- Azure ML
- Dynamics CRM
- PowerBI
- Office365

- ETL
- Data Management
- Data rationalization
- Big Data
- PowerBI

Volvo is reimagining the car-buying experience

Volvo is bringing digital elements of car buying to life to engage and inform customers, and to help them understand the features and options of the automobiles.



Customer Story

“Volvo is really a human-centric company, that’s the core focus of everything we’ve done in terms of the products we develop, but also in the way we interact with our customers.”

—Bjorn Annvall
SVP Marketing, Sales, & Services

Challenge

Create a more immersive car-buying experience to help customers choose and configure a vehicle.

Differentiate the Volvo brand, highlight innovation, and keep up with customer expectations.

Strategy

Volvo is using HoloLens to create an augmented reality interface for customers, helping them learn about and configure cars in three dimensions.

HoloLens enables customers to digitally interact with their automobiles in more immersive experiences.

Results

- Build reputation as an innovative and customer-focused company
- Increased customer satisfaction
- Increased sales, upsell and cross-sell
- Fast time-to-market for more immersive experiences

RealMadrid CF.

Improve fan engagement and experience

As the leading sports franchise in the world, Real Madrid has more than 100 championships and a passionate fan base. But what the football club didn't have was a way to directly engage with its 450 million global supporters.



Challenge

Engage with fans in more personal ways to increase revenue, customize marketing initiatives, and reinforce the club's leadership position in the worldwide sports industry.

Strategy

Implement a comprehensive platform-as-a-service solution to collect the data Real Madrid needs and to create personalized fan experiences and thrive

Customer Story

"Using the Microsoft Cloud, we are building a way of understanding who our fans are, where they are, and what they want from us."

— José Ángel Sánchez, CEO,
Real Madrid C.F.

Results

- One-to-one relationships with fans increases engagement
- Targeted, customized marketing
- Near-real-time campaigns
- Transparent user interactions with content and advertising

Leveraging technology to give people what they want

Metro Bank is focused on offering the best customer experience, how, when and where the customer wants it. The bank's motto is "no stupid bank rules" and its focus is on bringing back traditional retail banking, supported by flexible technology.



Challenge

Provide personalized customer service.
Enable agile operations, applying best practices from one branch to all.
Differentiate services and brand.

Strategy

Use Microsoft Dynamics CRM, Office 365, and Azure capabilities to provide tools to employees for identifying and responding to problems before they affect the customer relationship.
Enable employees to connect with experts and access knowledge base of answers.

Customer Story

"We've embraced the software of a company that helps us treat people like people — Office 365, [Microsoft Dynamics] CRM, and Yammer from Microsoft. With the technology working for us, we can focus on the really important stuff."

— Paul Marriott-Clarke
Commercial Director Metro Bank

Results

- Reduced customer wait times
- Improved consistency with every customer interaction
- Streamlined systems create more time for customers
- Ongoing connection with customers

Jean Coutu Group (JCG) is a large Canadian pharmacy that successfully balances the advantages of a large chain with a relentless focus on personalized service offered by a local personnel.



Challenge

- Deliver more timely, meaningful, and efficient service to customers
- Help store personnel be more active on the floor
- Provide mobile point-of-sale in store

Strategy

- Provide point-of-sale functionality to thousands of terminals and devices
- Streamline supply, warehouse, and delivery of products to the franchise network
- Enhance in store experiences for customers and employees

"As a franchiser, we're responsive to the demands of our customers as well as our stores. Microsoft Dynamics for Retail offers the flexibility and the technology to meet those demands."

—Alain Boudreault

Vice President and Chief Information Officer

Results

- Enhanced customer service
- Seamless shopping experience
- Reduce waiting times
- Platform for the future

Fujitsu is the leading Japanese information and communication technology company offering a full range of technology products, solutions and services to customers in over 100 countries.



"Leveraging the Fujitsu Eco-Management Dashboard solution alongside Microsoft Azure and the Fujitsu IoT/M2M platform, we are able to deliver real-time visualization of the engineering process for big data analytics to improve the entire production process and inform decision-making"

— **Hiroyuki Sakai**
Corporate Executive Officer, Executive Vice President,
Head of Global Marketing at Fujitsu Limited

Challenge

Fujitsu needed to optimize processing by both machines and humans in a way that could enable managers, engineers and scientists to improve product quality, streamline systems, and enhance functionality while reducing costs.

Strategy

Fujitsu brought together its Eco-Management Dashboard and IoT/M2M platform with Microsoft cloud services and Windows tablets to enable managers, engineers and scientists to simultaneously manage product quality, process efficiency, and equipment performance.

Results

- Operational excellence
- Reduced energy costs
- Higher quality products
- Enhanced functionality
- Greater productivity

Ecolab is a leading global provider of water, hygiene, and energy technologies and services.



Challenge

To help customers reduce water usage, Ecolab must capture and analyze real-time information from thousands of sensors in thousands of plants around the world.

As Ecolab grows their business, they need a way to deliver water management solutions on a much larger scale and at a much deeper level, and to demonstrate value to customers.

Strategy

Using the Azure cloud computing platform and advanced analytical capabilities, Ecolab helps customers improve production processes and better understand value.

Ecolab employees around the world use Microsoft productivity tools to collaborate and take action based on insight, and support field activities.

Customer Story

“Suddenly, complexity and size are no longer barriers...We can now harness the power of this platform to serve many more customers, measuring many more flows at many more plants than we could even conceive of in the past.”

— Christophe Beck
President, Nalco Water,
an Ecolab company

Results

- Reduced water usage by customers
- Improved insight about customer processes and performance
- Improved customer engagement and guidance
- Scalability to meet growing business needs

As the largest airline in the Middle East, Saudi Airlines offers flights to over 90 destinations, with global operations and more than 40,000 employees.

Customer Story



Challenge

Saudi Airlines needed a consolidated approach to critical business information spanning multiple systems across functions, necessitating a data warehouse and business intelligence solution for an integrated view across operations.

Strategy

Create an Azure-based solution to provide real-time, self-service insight and enable executives to make more timely business decisions.

Results

- Increased operational efficiencies by 2%
- Enabled self-service insights for executives with intuitive reports and dashboards
- Anticipated millions in savings

Rolls-Royce provides engines to nearly 13,000 commercial aircraft in service around the world. For two decades, Rolls-Royce has provided limited, but valuable services, extending comprehensive maintenance services to airlines using its engines.



“By working with Microsoft we can really transform our digital services, supporting customers right across engine-related aircraft operations to make a real difference to performance.”

– Tom Palmer, Senior Vice President Services, Civil Aerospace,

Challenge

Flight delays are disruptive and costly for passengers and freight, and have a very large impact on airlines. Rolls-Royce aims to minimize the cost and disruption of airline maintenance, and to expand the services it offers to develop new revenue streams.

Strategy

To better serve customers, Rolls-Royce is leveraging Azure to transform the way it uses data. Rolls-Royce collects and aggregates a very large amount of data from many locations, and uses analytical tools and models to predict maintenance needs. To help customers understand the data and make better decisions, the Rolls-Royce solution provides dashboards and visualization.

Results

- Minimize cost and disruption of flights.
- Enable airlines to achieve safer, more efficient, and more profitable operations.
- Increase revenue through new business models and compelling services.

Optoplexia was founded by Swedish researchers to help schools identify students at risk for dyslexia earlier than is possible with current screening methods.



Challenge

Create a portable tool that is fast, easy-to-use, scalable, and indicates when early intervention is necessary to treat dyslexia.

Strategy

Using the Azure platform, analytics, and machine learning, Optoplexia built a portable screening tool to easily assess children for dyslexia.

The solution sends data from eye-tracking equipment to the Azure cloud, where an analysis engine based on Azure Machine Learning evaluates the data against a large repository of eye-tracking data.

Customer Story

“The flexibility and ease of use of the Azure Machine Learning analytics platform makes it a perfect foundation for expanding our existing solution into new areas.”

– Fredrik Wetterhall
Chief Executive Officer

Results

- Early identification of students at risk for dyslexia
- Early intervention and treatment
- Higher accuracy and reduced costs of screening
- Scalable platform to enable screening for other diseases and conditions

Developing innovative products and services to meet customer demand

ABB is a leader in power and automation technologies that enable utility and industry customers to improve performance while lowering environmental impact. ABB is now expanding its infrastructure to enter the market for vehicle charging.



Challenge

ABB needs to innovate quickly to provide the equipment and value-added services needed to take an early lead in the vehicle charging market.

As part of innovating improved equipment and services, ABB needs to gather and analyze telemetry and customer behavior.

Strategy

ABB is developing charging equipment and stations using detailed data and new insight from stations, equipment, and customers.

ABB equipment will be connected to Azure and take advantage of the machine learning and predictive analytics that Azure supports.

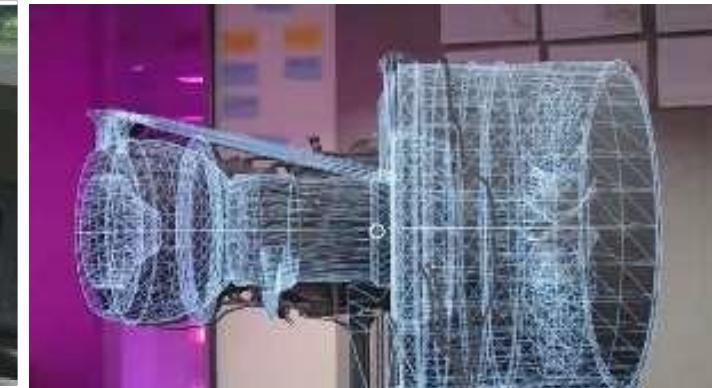
"This partnership gives us the solutions, scalability and global agility to support expanding demand for EVcharging infrastructure in the world's major automotive markets, which is a key focus of our Next Level growth strategy."

—Pekka Tiitinen
President
Discrete Automation and Motion Division

Results

- Innovative and scalable products and services
- Digital transformation of electric vehicle infrastructure
- Agility to quickly improve existing features and develop new features.

Japan Airlines operations include international and domestic passenger and cargo services to 220 destinations in 35 countries worldwide. The group has a fleet of 279 aircraft, and transports millions of passengers each year.



Challenge

Support safe operations of flights
Increase productivity of mechanics
Improve quality of flight training and maintenance
Build brand and innovate in operations and customer service

Strategy

Using the Microsoft HoloLens, users can see a version of a plane's engine and other equipment placed in the real world. Users can manipulate equipment, find specific parts, and simulate cockpit layouts and operations as though sitting in the cockpit seat. Japan Airlines is also adopting Microsoft's Surface Hub to streamline their development processes.

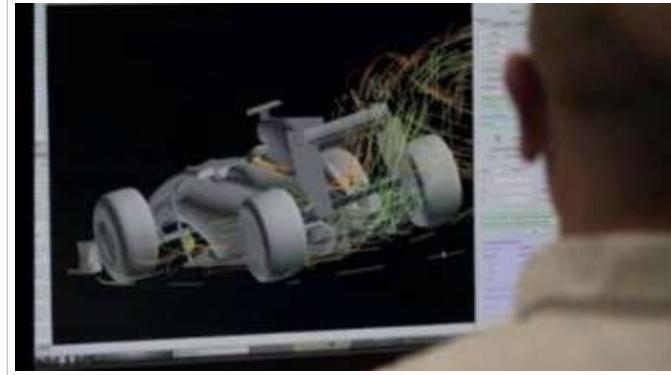
Results

- Improve training and quality of performance for maintenance crews
- Enable supplementary training for flight crews, and trainees seeking promotion to copilot status
- Optimize operations

Lotus F1

Insight and collaboration enable rapid advances in technology, design, and production

The Lotus F1 car is constantly evolving, with most of the components having limited life spans. The parts team is in perpetual motion, making improvements to parts within each season, and even between races.



Customer Story 
Duale Hochschule
Baden Württemberg

"We tied Dynamics AX in with everything from design release to consumption of parts on the car at the track, this included project-based manufacturing and design engineering—the production of the car."

—Thomas Mayer
COO, Lotus F1 Team

Challenge

Need better insight and collaboration to stay in constant motion and support the rapid pace of technological changes in the Formula 1 sport.

Strategy

The Lotus team leverages data coming from the car telemetry system, and using predictive analytics and machine learning, apply the insight to quickly and collaboratively innovate and produce better components.

Results

- Increased agility when developing and integrating design changes
- Improved sharing of information
- Increased cost awareness at all levels
- Support just-in-time manufacturing with improved insight into parts lifecycle