# ANFORDERUNGEN AN CLOUD NATIVE APPLIKATIONEN

FWP AKTUELLE TECHNOLOGIEN ZUR ENTWICKLUNG VERTEILTER JAVA ANWENDUNGEN

LUZIE MERTINGK

# INHALTSVERZEICHNIS

# CLOUD UNTERNEHMEN





# 1. MOTIVATION FÜR CLOUD-COMPUTING

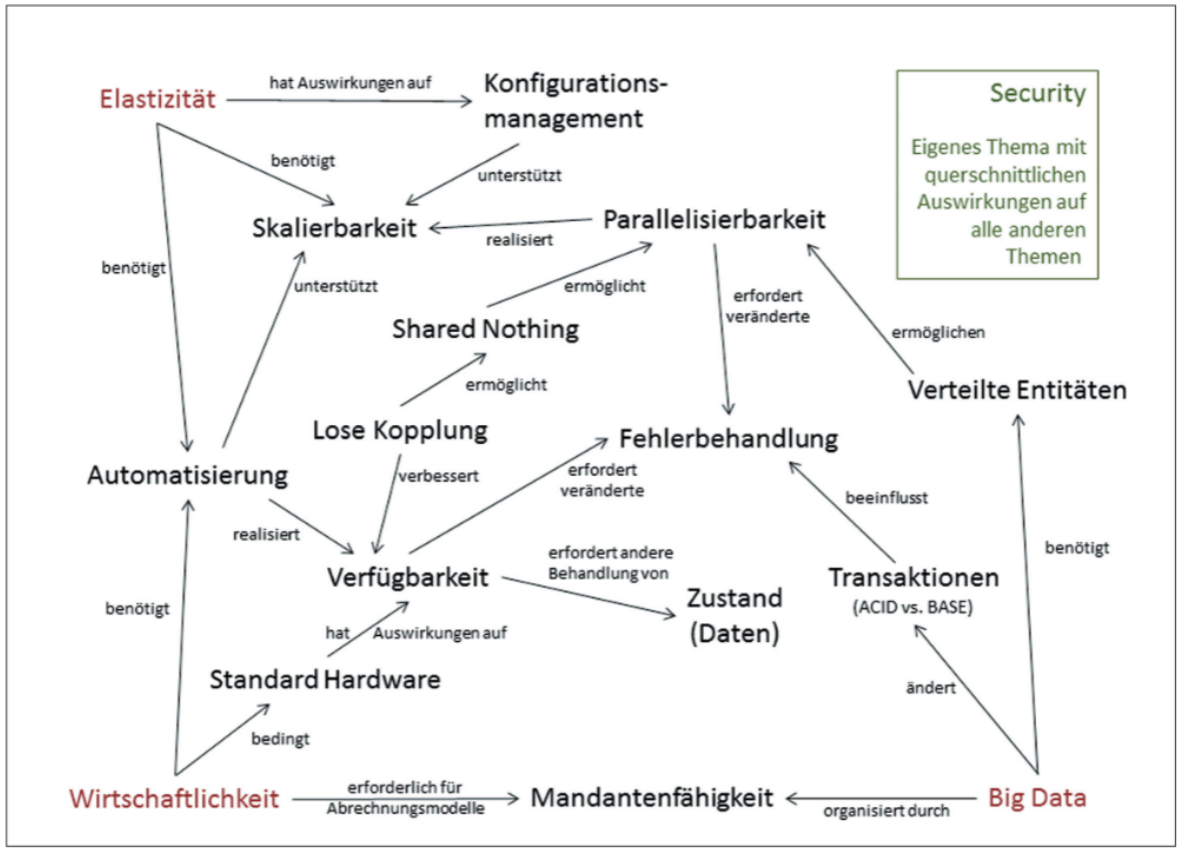
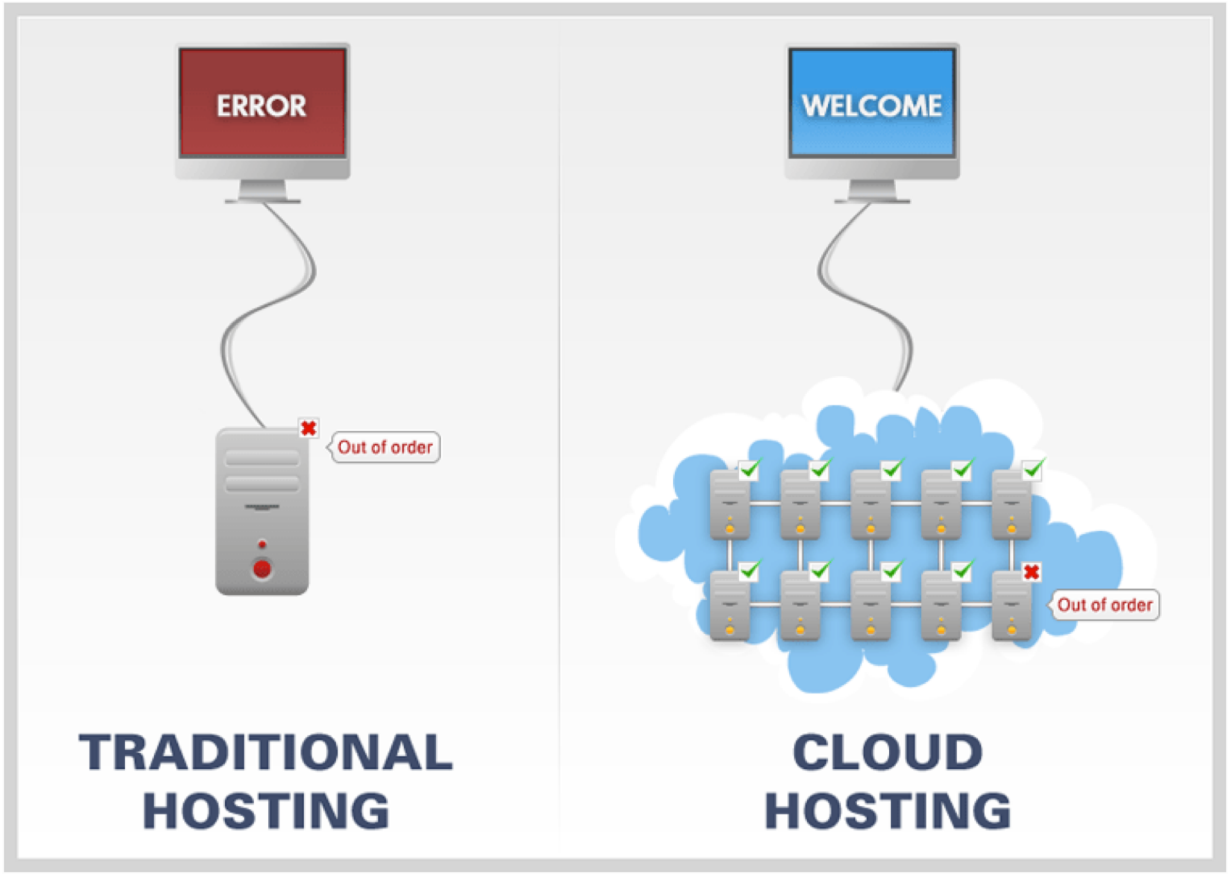



Abb. 1: Wichtige Treiber und Prinzipien von Cloud-Anwendungen im Überblick

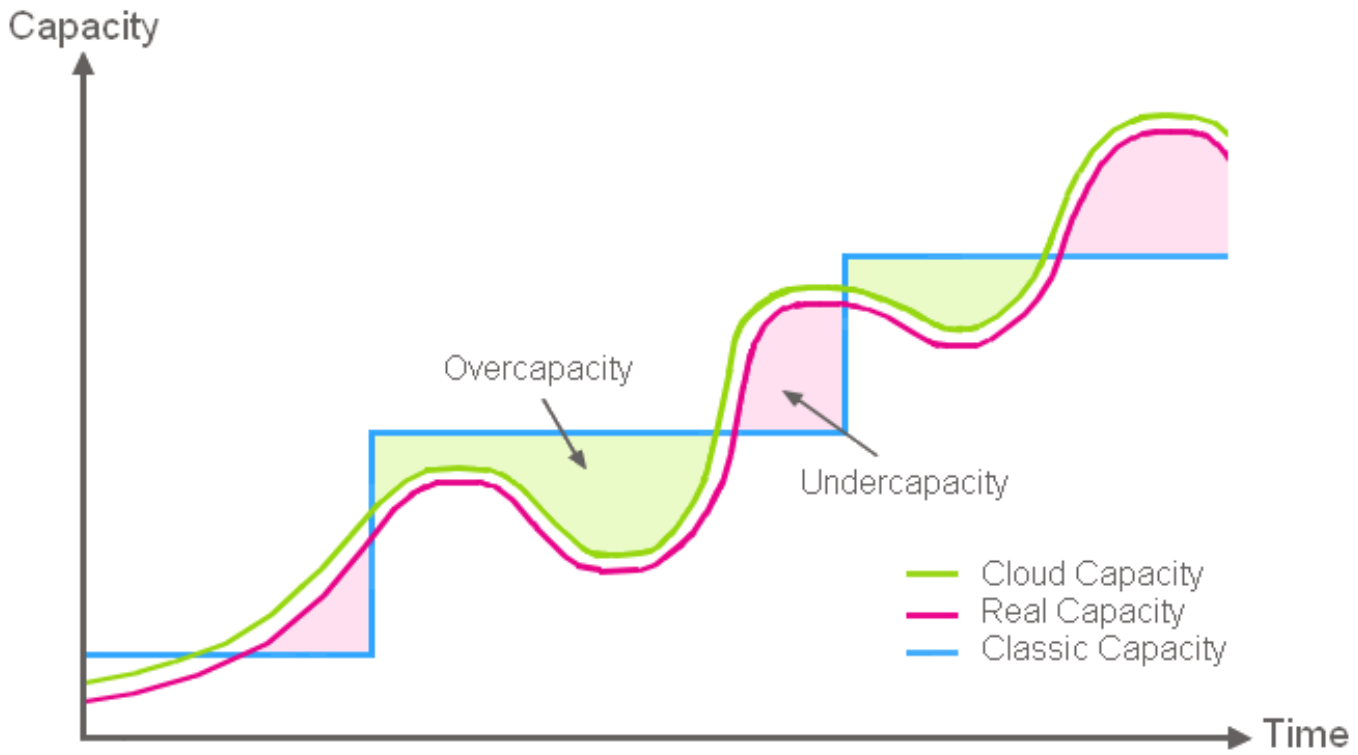
Quelle: www.codecentric.de

# 1. MOTIVATION FÜR CLOUD-COMPUTING



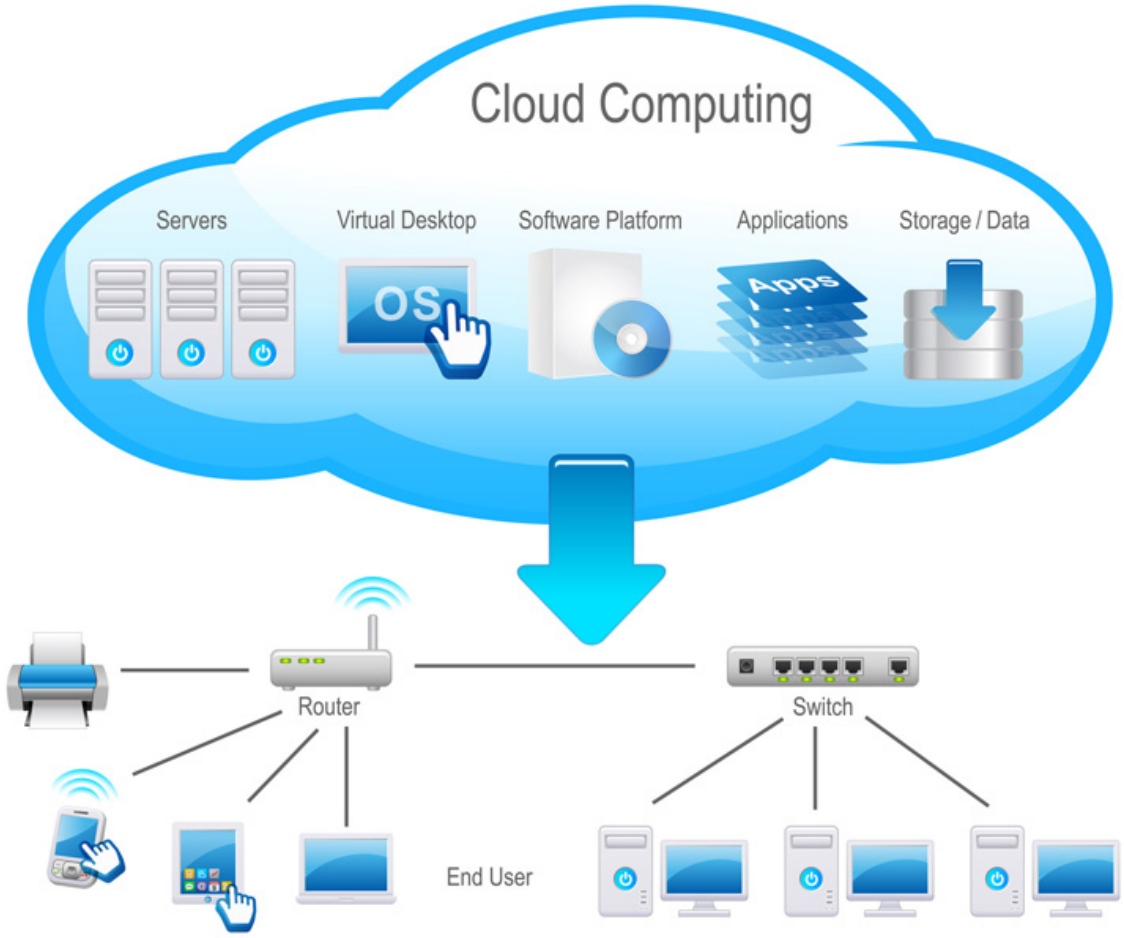

Quelle: www.namehero.com

# 1. MOTIVATION FÜR CLOUD-COMPUTING




Quelle: elasticsecurity.files.wordpress.com

# 2. WAS IST CLOUD-COMPUTING?




Quelle: www.nodetx.com

# 2.1 TYPEN DES CLOUD-COMPUTING

**Private Cloud**

- Stellt ein Unternehmen auf Basis ihrer eigenen Hardware bereit
- Nur eigene Mitarbeiter haben Zugriff drauf
- Bei B2B-Services auch die dedizierten Partner

**Hybrid Cloud**

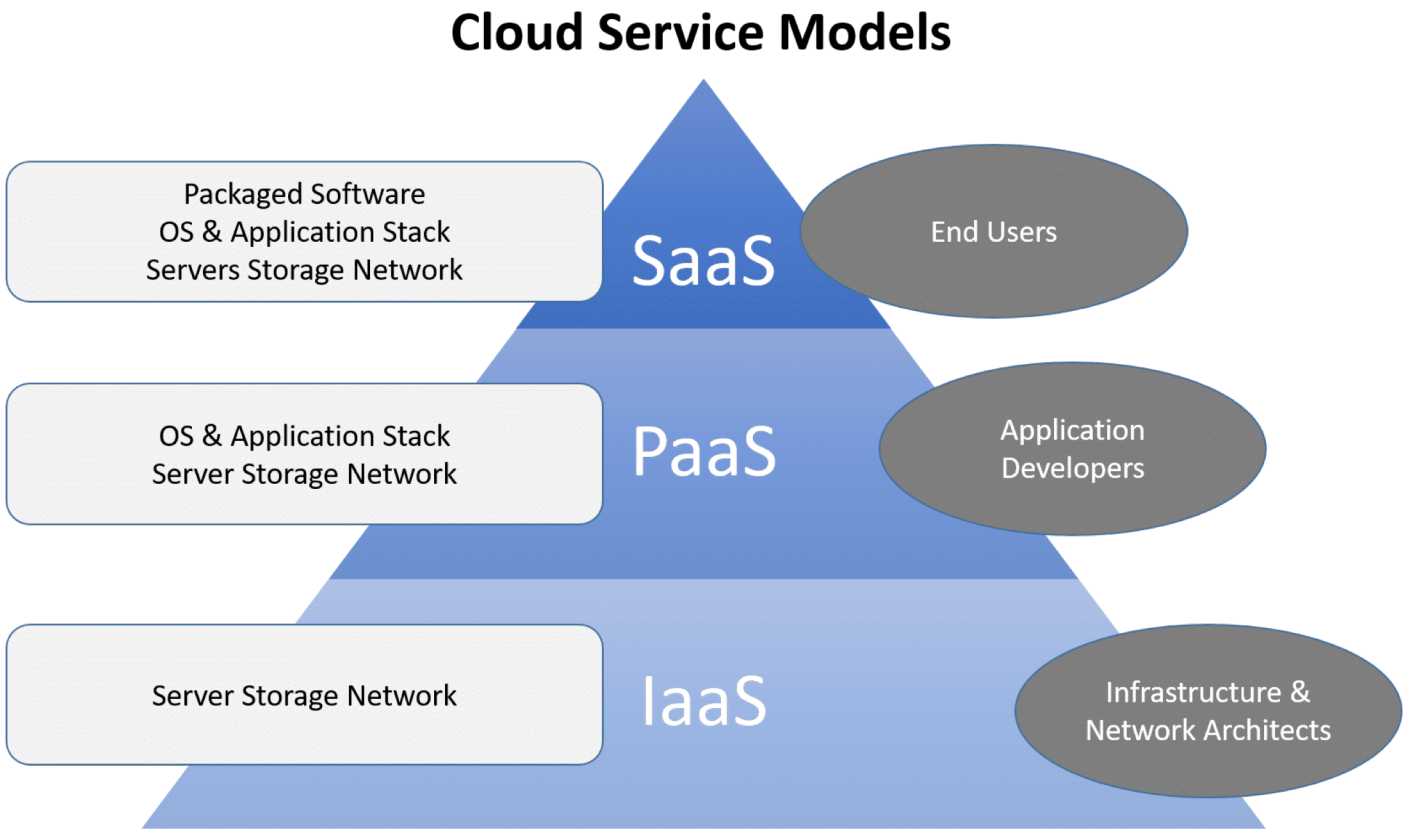
- Mischform aus Private und Public Cloud
- Services werden zwischen Public und Private Cloud verteilt
- Vertrauliche Daten werden intern und allgemeine Daten extern auf Public-Cloud-Servern gespeichert

**Public Cloud**

- Basieren auf globalen Netzwerken aus Rechenzentren und stellen nutzungsbasierte Services für die Öffentlichkeit oder große Unternehmen bereit

Quelle: www.impressico.com

# 2.2 VARIANTEN DES CLOUD-COMPUTING




Quelle: www.uniprint.net

# 3. ÄNDERUNGEN AN DER KLASSISCHEN SOFTWAREENTWICKLUNG

- Verfügbarkeit
  - In herkömmlichen Anwendungen: Einsatz von Hochverfügbarkeitsangeboten von Hardware- und Softwareanbietern → lassen sich nicht beliebig skalieren und die Kosten steigen schnell an
  - In Cloud Umgebungen: Einsatz von Standardhardware und Redundanz zu Sicherstellung der Verfügbarkeit
- Datenkonsistenz
  - Datenkonsistenz kann in Cloud-Umgebungen nicht uneingeschränkt aufrechterhalten werden (CAP-Theorem)
  - Es darf keine Mechanismen geben, die auf die Konsistenz der verarbeiteten Daten absolut vertrauen

# 3. ÄNDERUNGEN AN DER KLASSISCHEN SOFTWAREENTWICKLUNG

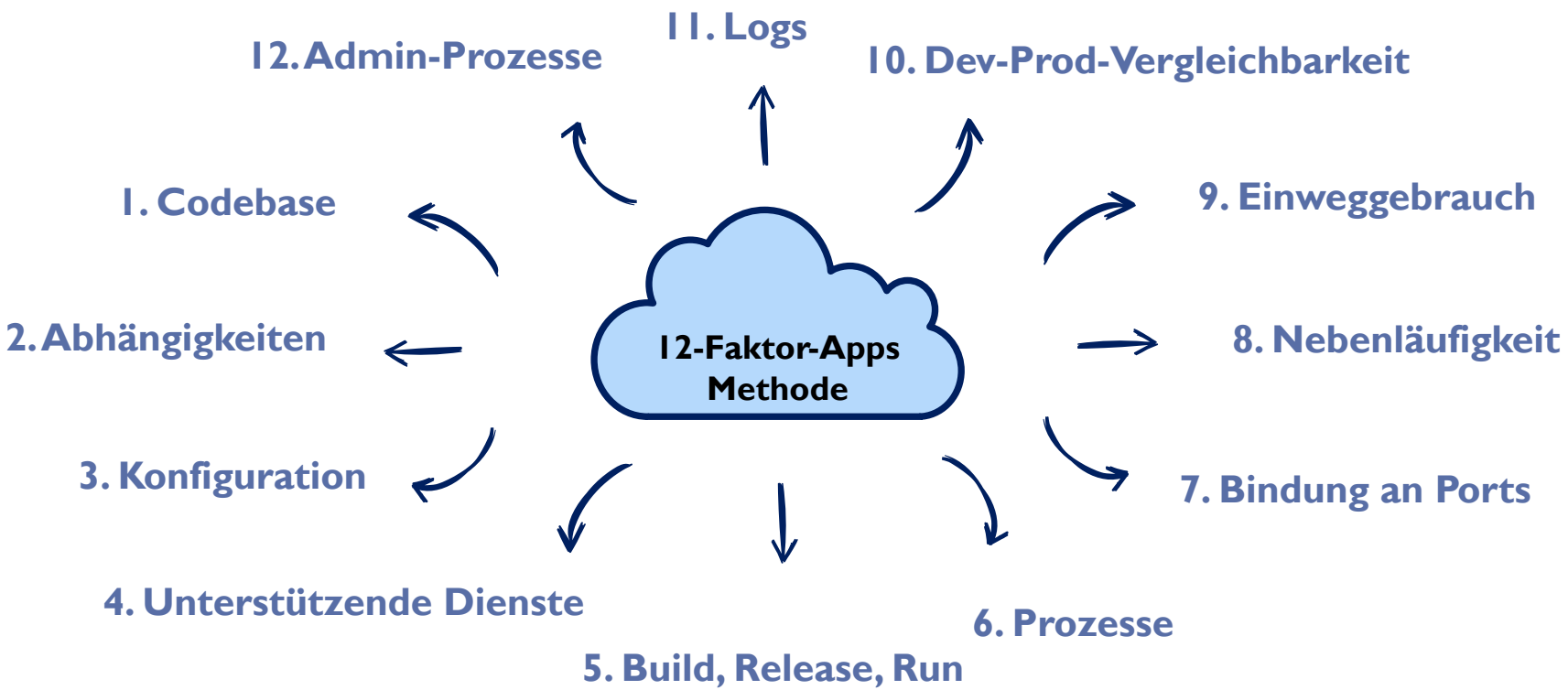
- Shared-Nothing
  - Klassische Architekturen lassen sich nicht beliebig skalieren → Komponenten teilen Zustand und bei steigender Skalierung wird Kommunikationsaufwand bei Replikation zu groß
  - Shared-Nothing-Gedanke: Komponenten teilen keinen Zustand, sie kommunizieren nur noch in Form von Nachrichten miteinander
- Lose Kopplung
  - Komponenten auf fachlicher und technischer Ebene sind möglichst unabhängig voneinander
  - Ein Block der Anwendungslogik lässt sich leichter skalieren, wenn er sich vom Rest der Anwendungslogik abkoppeln lässt
  - Auf technischer Ebene: iVm Shared-Nothing-Gedanken → Blöcke kommunizieren nur mittels Nachrichten

# 3. ÄNDERUNGEN AN DER KLASSISCHEN SOFTWAREENTWICKLUNG

- Parallelisierbarkeit
  - Möglichst wenig nicht-parallelisierbare Anteile
  - Je mehr nicht parallelisierbare Anteile ein Programm enthält, desto geringer ist der Geschwindigkeitsgewinn bei einer Parallelisierung der Ausführung
- Automatisierung
  - Automatisertes Deployment und automatisierter Start der verschiedenen Anwendungsteile
  - Automatisierte Bereitstellung der benötigten Infrastruktur
  - Monitorprogramm muss bei Erkennen einer hohen Last  selbstständig eine weitere Instanz des Anwendungsteils aufsetzen und starten

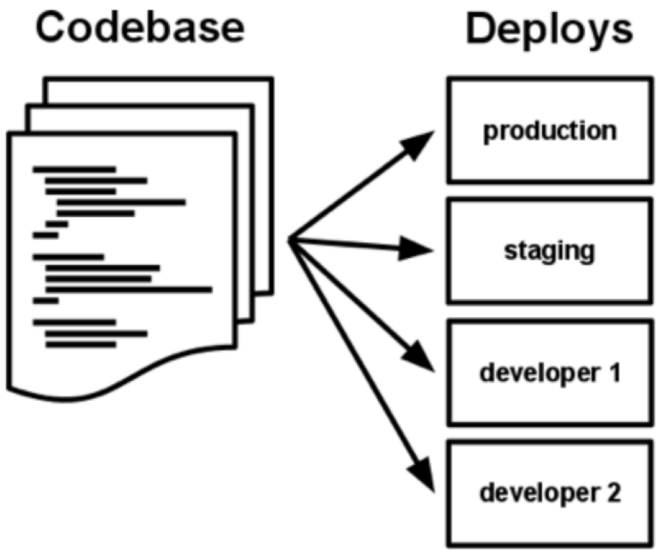
→ durch diese Prinzipien kann eine Cloud-Anwendung Anforderungen wie Skalierbarkeit, Elastizität und Robustheit gerecht werden

# 4. DIE 12-FAKTOR-APPS METHODE



# 4.1 CODEBASE

Eine im Versionsmanagementsystem verwaltete Codebase, viele Deployments




Quelle: 12factor.net

# 4.2 ABHÄNGIGKEITEN

Abhängigkeiten explizit deklarieren und isolieren

- Klassische Umgebungen: Server stellt alles bereit, was die Anwendung benötigt („Mommy Server“)
- 12-Faktor-App: kann sich nicht auf die Auflösung von Abhängigkeiten verlassen
- z.B. in Java: Anwendung kann nicht davon ausgehen, dass ein Container den Klassenpfad auf dem Server verwaltet
- Sie werden hier über Abhängigkeitsdeklaration deklariert
- Zur Laufzeit: Werkzeug zur Isolation von Abhängigkeiten

# 4.3 KONFIGURATION

Die Konfiguration in Umgebungsvariablen ablegen

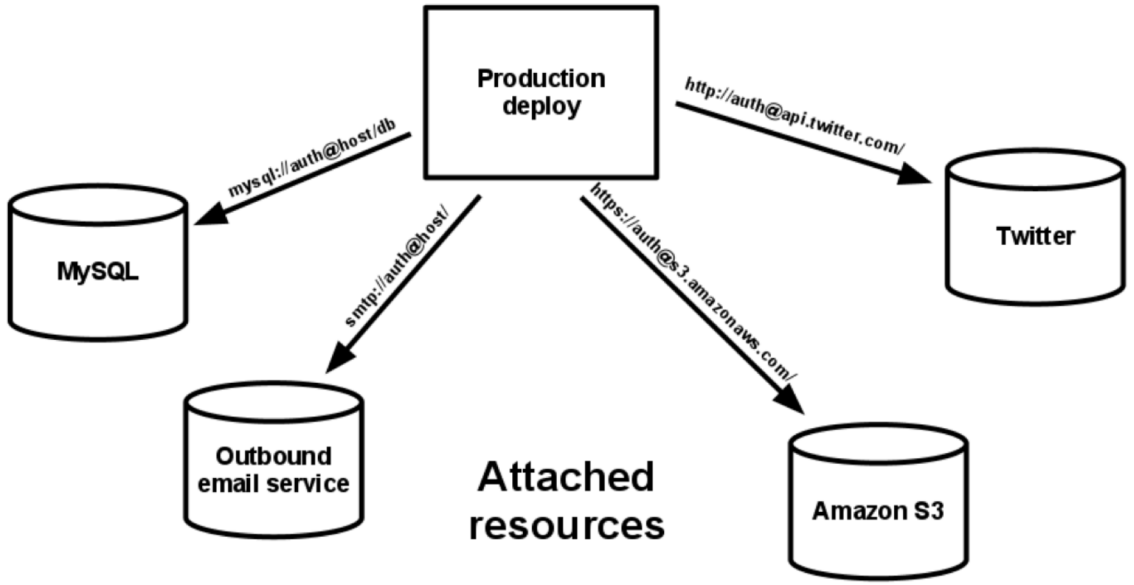
- Strikte Trennung der Konfiguration vom Code

  → Konfiguration darf auf keinen Fall als Konstante im Code gespeichert werden, da sie sich im Gegensatz zum Code in den Deploys unterscheiden

- Beispiele für Konfiguration: URLs zu Web-Services und SMTP-Servern, Informationen zu einer Datenbankverbindung
- Konfiguration wird in Umgebungsvariablen (env) gespeichert
- Vorteil: können im Gegensatz zu Konfigurationsdateien nicht versehentlich im Repository landen

# 4.4 UNTERSTÜTZENDE DIENSTE

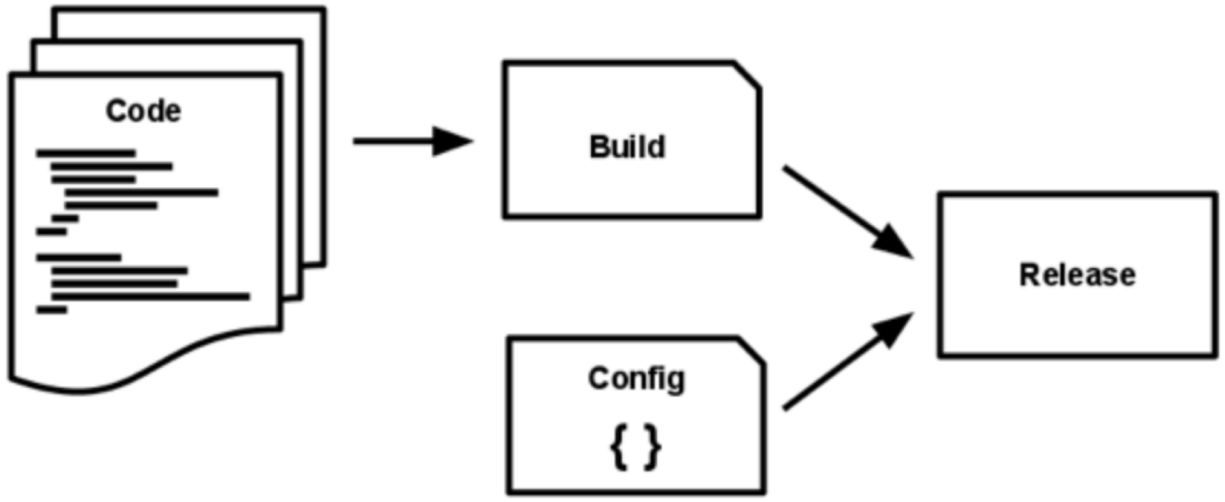
Unterstützende Dienste als angehängte Ressourcen behandeln




Quelle: 12factor.net

# 4.5 BUILD – RELEASE - RUN

Build- und Run-Phase strikt trennen




Quelle: 12factor.net

# 4.6 PROZESSE

Die App als einen oder mehrere Prozesse ausführen

- Anwendungen sollen als einzelner, zustandsloser Prozess ausgeführt werden
- Anders gesagt: Zustände sind zwar erlaubt, müssen sich jedoch außerhalb der Anwendung befinden
- Wenn sich Prozesse Daten teilen müssen, z.B. einen Sitzungsstatus, sollte dieser externalisiert und über einen unterstützenden Dienst bereitgestellt werden
- Drittanbieter für Caching-Produkte:

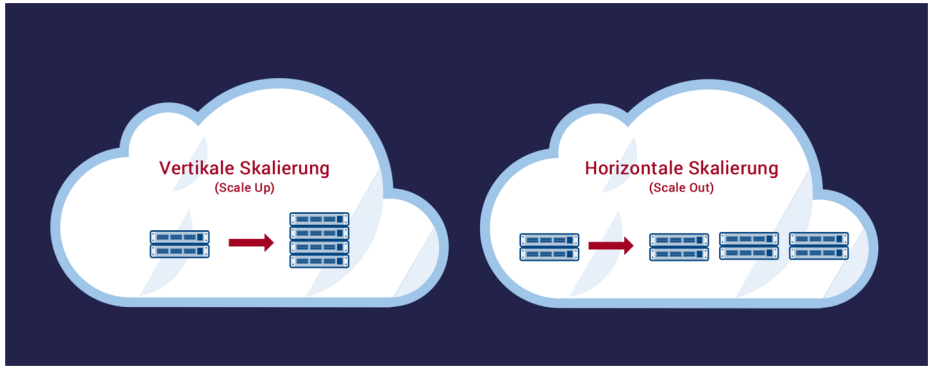

# 4.7 BINDUNG AN PORTS

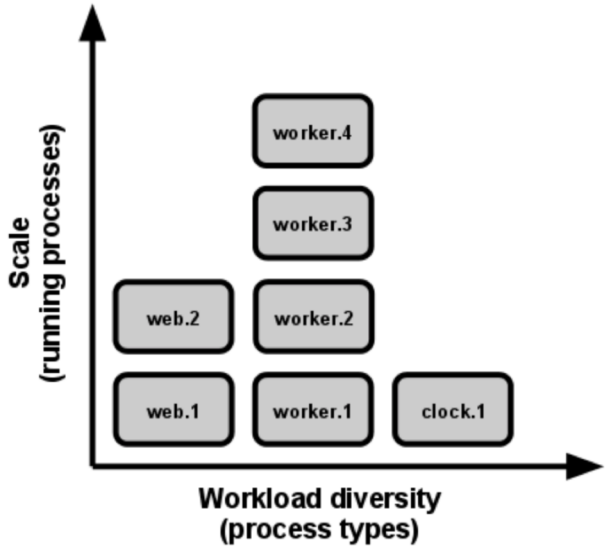
Dienste durch das Binden von Ports exportieren

- Web-Anwendungen werden oft in Containern ausgeführt (z.B. Java in Tomcat)
- Nicht-Cloud-Umgebung: Container ist für Zuweisung von Ports verantwortlich
- Cloud-Umgebung: Cloud-Anbieter sollte Portzuordnung übernehmen
- Dienste sollten jedoch nicht über IP + Portnummer aufgerufen werden, werden oder URLs

# 4.8 NEBENLÄUFIGKEIT

Mit dem Prozess-Modell skalieren




Quelle: 12factor.net,
www.scaleuptech.com

# 4.9 EINWEGGEBRAUCH

Robuster mit schnellem Start und problemlosen Stopp

- Prozesse in Cloud-Anwendung sind „wegwerfbar“

  → müssen schnell gestartet und gestoppt werden können

- Langsame Startzeit sollte Warnungen auslösen, extrem lange Startzeiten können sogar verhindern, dass die App überhaupt in der Cloud gestartet wird

# 4.10 DEV-PROD VERGLEICHBARKEIT

Entwicklung, Staging und Produktion so ähnlich wie möglich halten

- Schließung von
  - Zeitlücke
  - Personenlücke
  - Werkzeuglücke

| | Traditionelle App | Zwölf-Faktor-App |
|---|---|---|
| **Zeit zwischen Deployments** | Wochen | Stunden |
| **Code-Autoren und Code-Deployer** | Andere Menschen | Dieselben Menschen |
| **Entwicklungs- und Produktions-Umgebung** | Unterschiedlich | So ähnlich wie möglich |

Quelle: 12factor.net

# 4.11 LOGS

Logs als Strom von Ereignissen behandeln

- Logs = Folge von zeitlich sortierten Ereignissen aller laufenden Prozesse und unterstützenden Diensten
- Herkömmliche Entwicklung: Entwickler konnten Ziel der Logs streng kontrollieren
- 12-Faktor-App befasst sich nicht mit dem Routing oder der Speicherung ihrer Output-Streams
- Laufende Prozesse einer Cloud-Anwendung schreibt Logeinträge in Standardausgabe

# 4.12 ADMIN-PROZESSE

Admin/Management-Aufgaben als einmalige Vorgänge behandeln

- Entwickler möchten oft, während die üblichen Prozesse der App laufen, nebenbei einmalige administrative Aufgaben an der App erledigen, z.B.:
  - Starten einer Datenbankmigration
  - Einmaliges Ausführen von Skripten aus dem Repository
  - Starten von Code aus der Konsole heraus
- Diese Prozesse sollten in einer Umgebung laufen, die der Umgebung der üblichen Prozesse gleich ist
- Sie benutzen die gleiche Codebase und Konfiguration wie jeder Prozess, der gegen ein Release läuft

# 5. BEISPIEL FÜR DIE WICHTIGSTEN PUNKTE DER 12-FAKTOR-APPS METHODE

Beispielanwendung: CRM-System von Salesforce

# WOFÜR WIRD CRM EINGESETZT?

- Beispiel Autohaus:

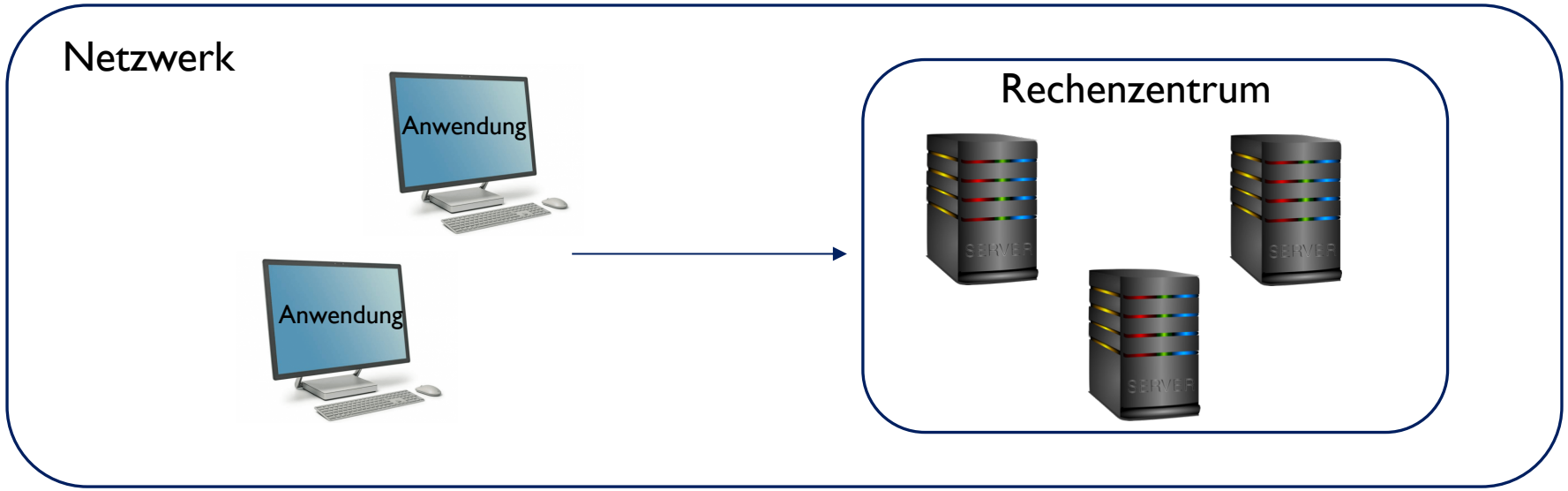
- Überblick über Kundenbeziehungen
- Leasing-Vertragslaufzeiten überwachen und kontrollieren

→ Diese Daten sollen nicht in der Schublade der Verkäufer liegen, sondern zentral in CRM-System verwaltet werden

# WELCHE MÖGLICHKEITEN HAT DAS AUTOHAUS?

Selbst entwickeln bzw. fertige Lösung z.B. von SAP installieren und betreiben

→Was braucht das Autohaus, um ein fertiges oder selbst entwickeltes CRM-System betreiben zu können?



# MÖGLICHKEIT 1: WEB-ANWENDUNG

- Problem der Administration kann entschärft werden
- Die Web-Anwendung kann über einen Browser aufgerufen werden → wesentliche Anwendungslogik läuft nur noch auf dem Server


Quelle: cdn3.iconfinder.com

# MÖGLICHKEIT 2: EINSATZ VON IAAS

- Problem der teuren Infrastruktur wird umgangen
- Vorteile: lassen sich dynamisch nach Nutzeranzahl skalieren

  → hohe Kosten mit steigender Nutzeranzahl können vermieden werden
- Unternehmen kann Anwendung also nun skalieren, entwickelt und verwaltet das CRM-System jedoch immer noch selbst


Quelle: www.firstlight.net

# MÖGLICHKEIT 3: EINSATZ VON PAAS

- CRM-Systeme bieten alle eine ähnliche Funktionalität und verwenden ähnliche Technologien → Cloud-Anbieter stellen Plattform bereit, auf dessen Basis Unternehmen eigene Anwendungen entwickeln können
- Unternehmen entwickelt Anwendung immer noch selbst, lädt diese aber in eine bestehende Plattform hoch


Quelle: microchannel-dev.com

# MÖGLICHKEIT 4: EINSATZ VON SAAS

- Basis der 12-Faktor-Apps-Methode
- Nicht nur Plattform oder Infrastruktur wird bereitgestellt, sondern gesamte Anwendung
- Wesentlicher Vorteil: wird nach Pay-per-Use-Prinzip abgerechnet
- Unternehmen muss sich nicht um Betrieb und Support kümmern


Quelle: www.securens.in

# ENTWICKLUNG EINER SAAS-ANWENDUNG

- Autohaus vs. Salesforce → lokale Anwendung vs. Cloud-Anwendung

→ was muss beachtet werden?


Quelle: www.ipfolio.com

# FAKTOR 9: EINWEGGEBRAUCH

- Funktionalität muss unterschiedlichen Benutzern hochverfügbar bereit stehen → Als Softwareanbieter liefert man nicht mehr einzelne Softwarepakete, die der Autohändler dann deployed
- Dafür spielt Skalierbarkeit eine große Rolle: System lässt sich beliebig skalieren, wenn neue Instanzen der Anwendung schnell hochgefahren werden können

Quelle: www.tanaza.com

# FAKTOR 1: EINE CODEBASE, VIELE DEPLOYS

- Annahme: Salesforce möchte seinen Dienst in Europa und in den USA anbieten
- Dafür müssen von der Anwendung mehrere Instanzen erstellt werden können
- Anforderung: Daten der amerikanischen Benutzer in amerikanischen Rechenzentren, Daten der europäischen Benutzer in europäischen Rechenzentren verwalten
- Bedeutung für Entwickler:
  - Früher: eine Codebase wird für ein bestimmtes System für ein bestimmtes Rechenzentrum deployed
  - Jetzt: eine Codebase wird in unterschiedlichen Konfigurationen deployed: in der Cloud in Europa und in der Cloud in den USA

Quelle: www.senawave.com

# FAKTOR 3: KONFIGURATION

- Konfiguration ändert sich in den verschiedenen Deploys
- Jeweils eine Konfiguration für USA und Europa
- Beispielsweise unterschiedliche Verschlüsselungsverfahren für die Datenhaltung
- Unterschiede der Konfigurationen sollen nicht im Code stehen → Der Code muss unabhängig davon sein, in welchem Markt und mit welcher Konfiguration er deployed wird

Quelle: www.businesstech.fr

# FAKTOR 3: KONFIGURATION

- Wenn Code auf bestimmte Ressourcen zugreift, darf die Adressierung nicht direkt im Code stehen
- Muss in der Konfiguration enthalten sein
- 12-Faktor-App holt sich diese Konfigurationsinformation aus einer Umgebungsvariable
- Information wird beim Bauen eines Releases für ein spezielles Deployment (z.B. USA) aus der Konfiguration entnommen

  → Hier wird der Code mit der Konfiguration zusammengeführt

# FAKTOR 5: BUILD – RELEASE – RUN

– Phasen Build, Release und Run müssen daher strikt getrennt werden

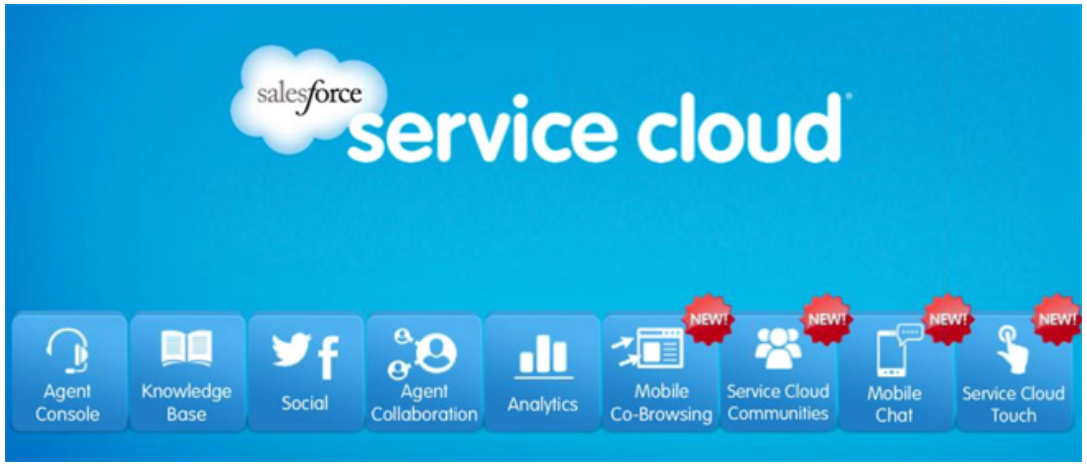
1. Build: Codebase wird unabhängig vom Markt in ein kompiliertes Artefakt umgewandelt
2. Release: Artefakt wird mit der jeweiligen Konfiguration kombiniert
3. Run: das Release wird in der jeweiligen Cloud in Europa und USA ausgeführt

Vorteile dieses Verfahrens:

– Anwendung kann kontinuierlich deployed werden
– Entwickler müssen keine „Angst“ mehr vor Deploy haben
– Salesforce kann in beliebigen Märkten deployen, ohne etwas am Code zu ändern

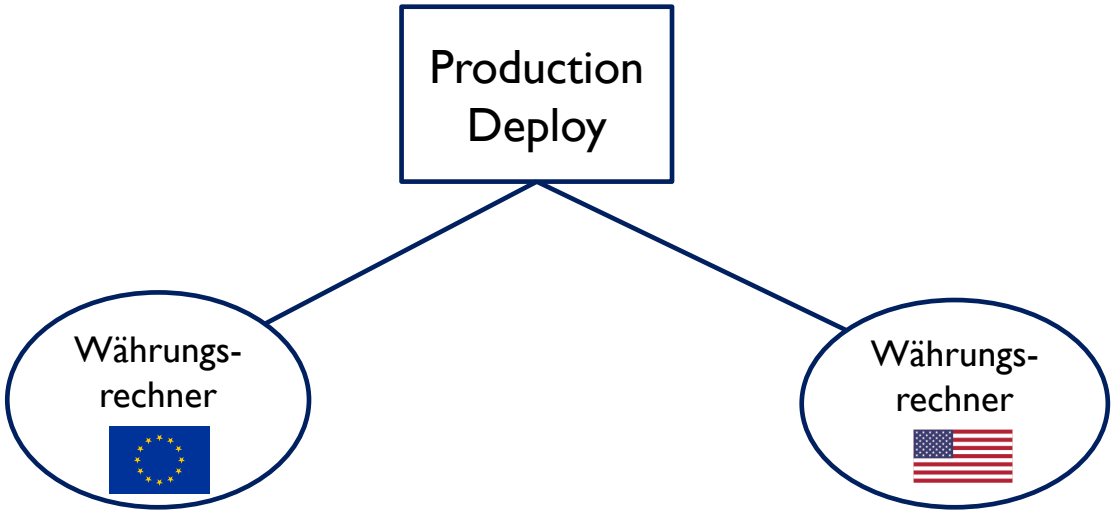
# NEUE FEATURES

- Salesforce möchte ein neues Feature bereitstellen
- Muss schnell eingebracht und deployed werden können
- Anwendung darf daher nicht aus einem großen Monolithen bestehen, sondern muss auf mehrere Komponenten verteilt werden
- Mehrere Teams können an verschiedenen Features arbeiten
- z.B. Sales Analytics oder Adressverwaltung der Kunden
- Entwickler müssen nicht ganzen Code auschecken, sondern nur die einzelne Komponente




Quellewww.signitysolutions.com

# FAKTOR 4: UNTERSTÜTZENDE DIENSTE




- Währungsdienst muss beliebig austauschbar sein
- Man baut keine libary ein, sondern verwendet einen Dienst, der den entsprechenden Währungsrechner anbindet
- So kann dieser in den verschiedenen Märkten, aber auch innerhalb eines Marktes ausgetauscht werden

# FAKTOR 6: ZUSTANDSLOSE PROZESSE

- Bezogen auf den Punkt des schnellen Aufbauens von Instanzen
- Wenn die Anwendung für das Autohaus entwickelt wird kann ein Zustand gehalten werden
- Bei Salesforce: Anwendung wird vielen Unternehmen angeboten → Zustand in einem Prozess soll nicht für die verschiedenen Mandanten gehalten werden
- Bei steigender Last sollen schnell weitere Prozesse instanziiert werden und nicht darauf gewartet werden, bis ein Zustand geladen wurde
- Baut auf auf Faktor 9 der Nebenläufigkeit auf:

  → im Rahmen der horizontalen Skalierung können schnell neue Prozesse erzeugt und aufgrund der Zustandslosigkeit mehrere Instanzen der Anwendung nebenläufig ausgeführt werden

# FAKTOR 6: ZUSTANDSLOSE PROZESSE

- Beispiel: Reportfunktion (gibt Auskunft über den Auftragseingang)
- Während der Prozess des Reports läuft soll der Benutzer davon nichts mitbekommen

  → Prozess soll nebenläufig zu allen anderen Prozessen laufen
- Dies bedeutet bezogen auf Datenkonsistenz (CAP-Theorem):
  - Daten sind nur „eventuell konsistent“
  - Klassische Entwicklung: In der Zwischenzeit eingehender Auftrag würde nicht angelegt werden, da durch Report gesperrt oder Report würde abgebrochen werden
  - Cloud: Report berücksichtigt nur die Aufträge bis zum Start des Reports (dies wird dem Anwender mitgeteilt)

# FAKTOR 3: ABHÄNGIGKEITEN

- Klassische Entwicklung: Klassenpfad oder Datenbanktreiber werden in einem Rechenzentrum verwaltet bzw. aufgelöst
- Man ist der Annahme, dass wenn die Anwendung deployed wird, alle Abhängigkeiten aufgelöst sind
- Salesforce kann so nicht vorgehen
- Cloud-Anwendung muss in einer Abhängigkeitsdeklaration angeben, welche Abhängigkeiten sie benötigt → können ggf. automatisiert bereitgestellt werden
- Idealerweise werden sie bei einem automatisierten Aufsetzen per „Infrastructure as Code“ aufgelöst

  → für jede Konfiguration existiert Code, der die Infrastruktur per API der Cloud-Plattform konfigurationsspezifisch erzeugt

# FAZIT

- Einige Vorteile: kostengünstig skalieren, Hochverfügbarkeit, Elastizität
- Aber: bei der Entwicklung einer Cloud App muss einiges beachtet werden → weg vom lokalen Rechenzentrum hin zum umgebungsunabhängigen Cloud-Computing

→ Kann auch in der klassischen Entwicklung eingesetzt werden, aber in der Cloud ist es Pflicht, um erfolgreich eine echte Cloud App entwickelt und betreiben zu können


©Alex Hallatt