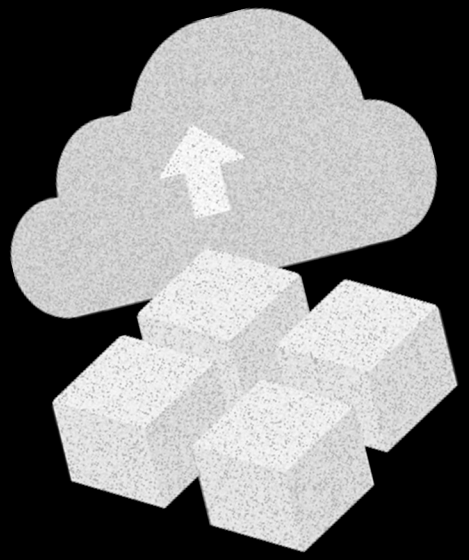


CLOUD-NATIVE COMPUTING

Unit 08:

Serverless Architekturen

Stand: 27.03.23

CLOUD NATIVE COMPUTING

PROF. DR.
NANE KRATZKE

KAPITEL 12

Serverless Architekturen

Nane Kratzke

Cloud-native Computing

Software Engineering von Diensten und Applikationen für die Cloud

284 Seiten. E-Book inside

€ 59,99. ISBN 978-3-446-46228-1

Weitere Informationen unter: **www.hanser-fachbuch.de** HANSER

CLOUD NATIVE COMPUTING

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

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

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

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

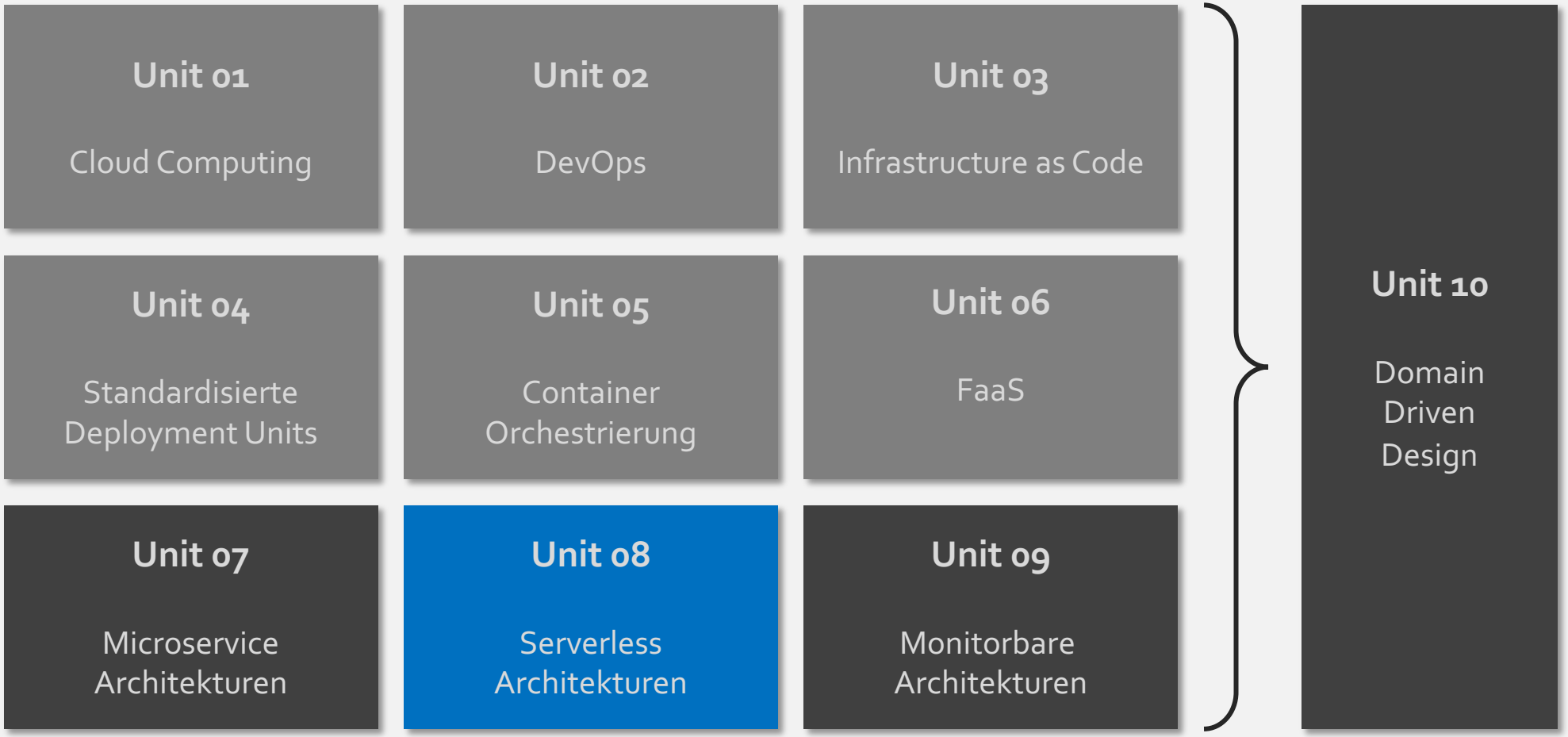
Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

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

Die Quellenangaben finden sich auf den entsprechenden Folien.

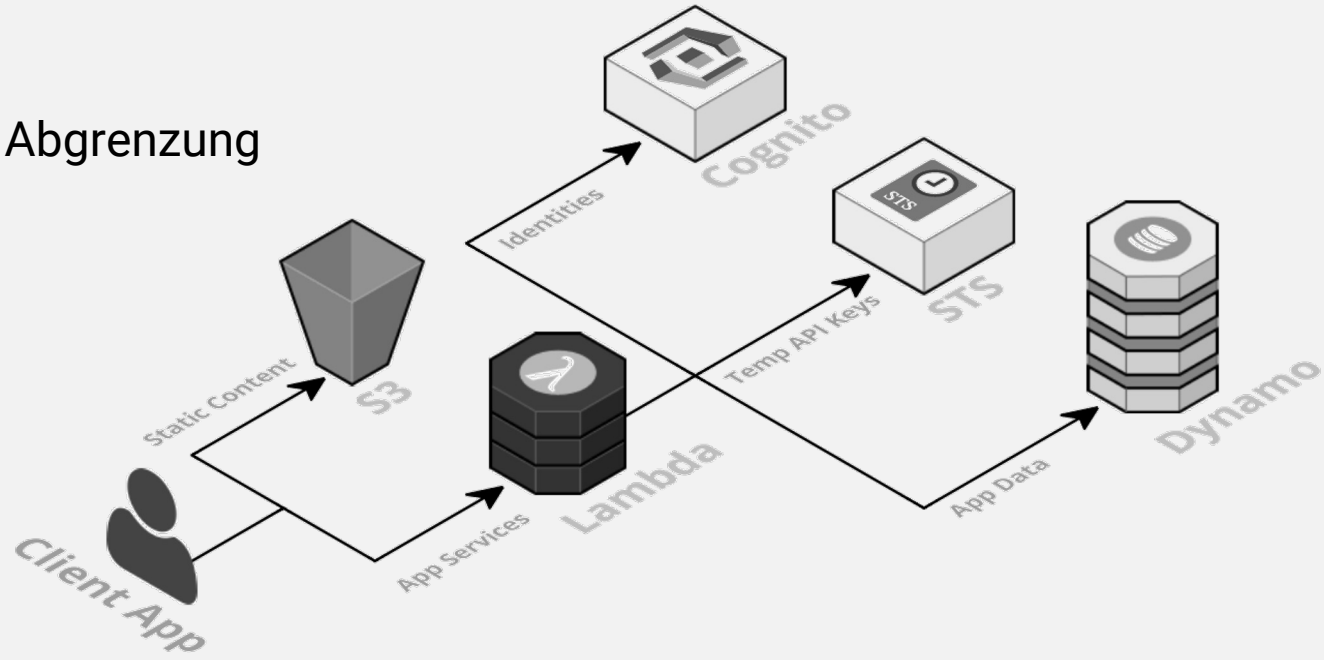
INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



INHALTE

- Was ist Serverless?
- Limitierungen von Serverless Ansätzen
- Wie werden Serverless Architekturen eingesetzt?
- API-Gateways
- Zusammenfassung und Abgrenzung zu Microservices

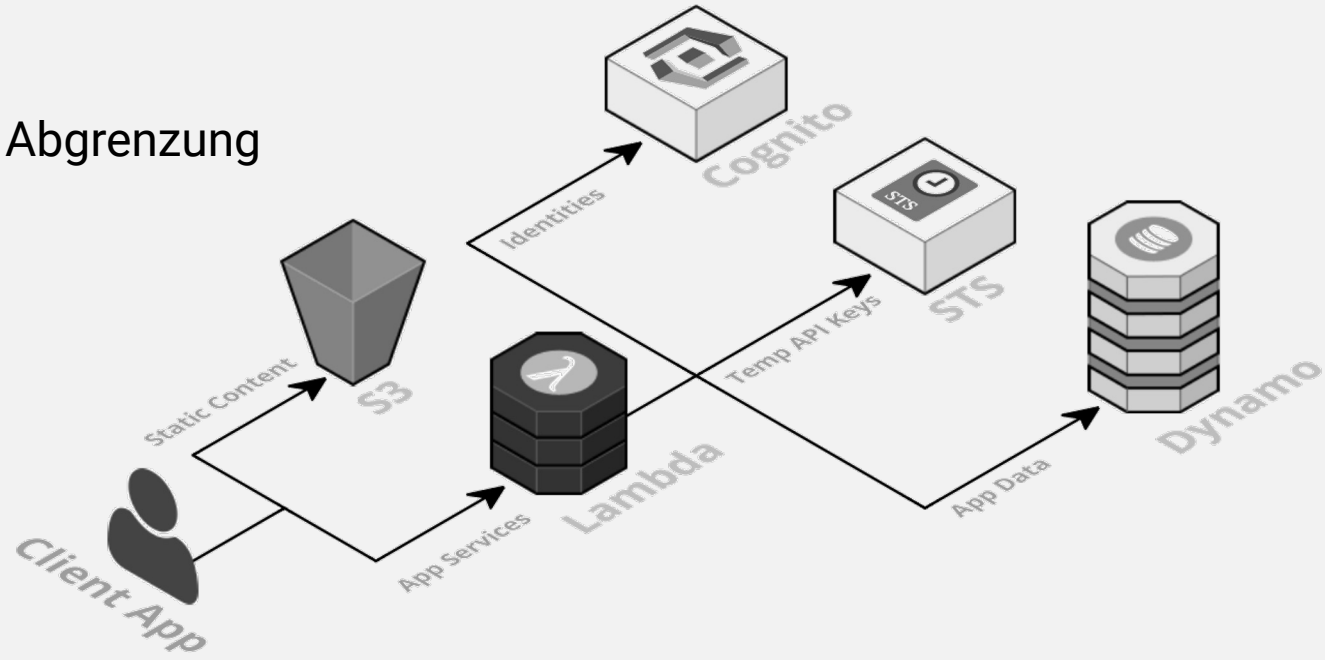


Es ist zu beachten, dass viel was für Microservice Architekturen gültig ist, auch für Serverless Architekturen gilt.

Diese Unit geht daher primär auf die Serverless-Besonderheiten ein und ist daher deutlich kürzer als die Unit 7 (Microservices).

INHALTE

- **Was ist Serverless?**
- Limitierungen von Serverless Ansätzen
- Wie werden Serverless Architekturen eingesetzt?
- API-Gateways
- Zusammenfassung und Abgrenzung zu Microservices



In dieser Unit betrachten wir vor allem die architekturellen Implikationen, die sich aus dem FaaS Ansatz der Unit 06 ergeben.

WAS IST SERVERLESS?

CNCF Definition

*„Serverless computing refers to a new model of cloud native computing, enabled by architectures that do **not require server management** to build and run applications. This landscape illustrates a **finer-grained deployment model** where applications, bundled as one or more **functions**, are uploaded to a platform and then executed, scaled, and billed in response to the **exact demand** needed at the moment.“*

Cloud Native Computing Foundation, CNCF

SERVERLESS COMPUTING

Scale-to-Zero vermeidet das teuerste Cloud-Nutzungsmuster

Serverless Computing ist ein Service-Modell, bei dem die Zuweisung von Ressourcen dynamisch auf Basis eingehender Requests (Events) verwaltet wird und absichtlich außerhalb der Kontrolle des Servicekunden liegt.

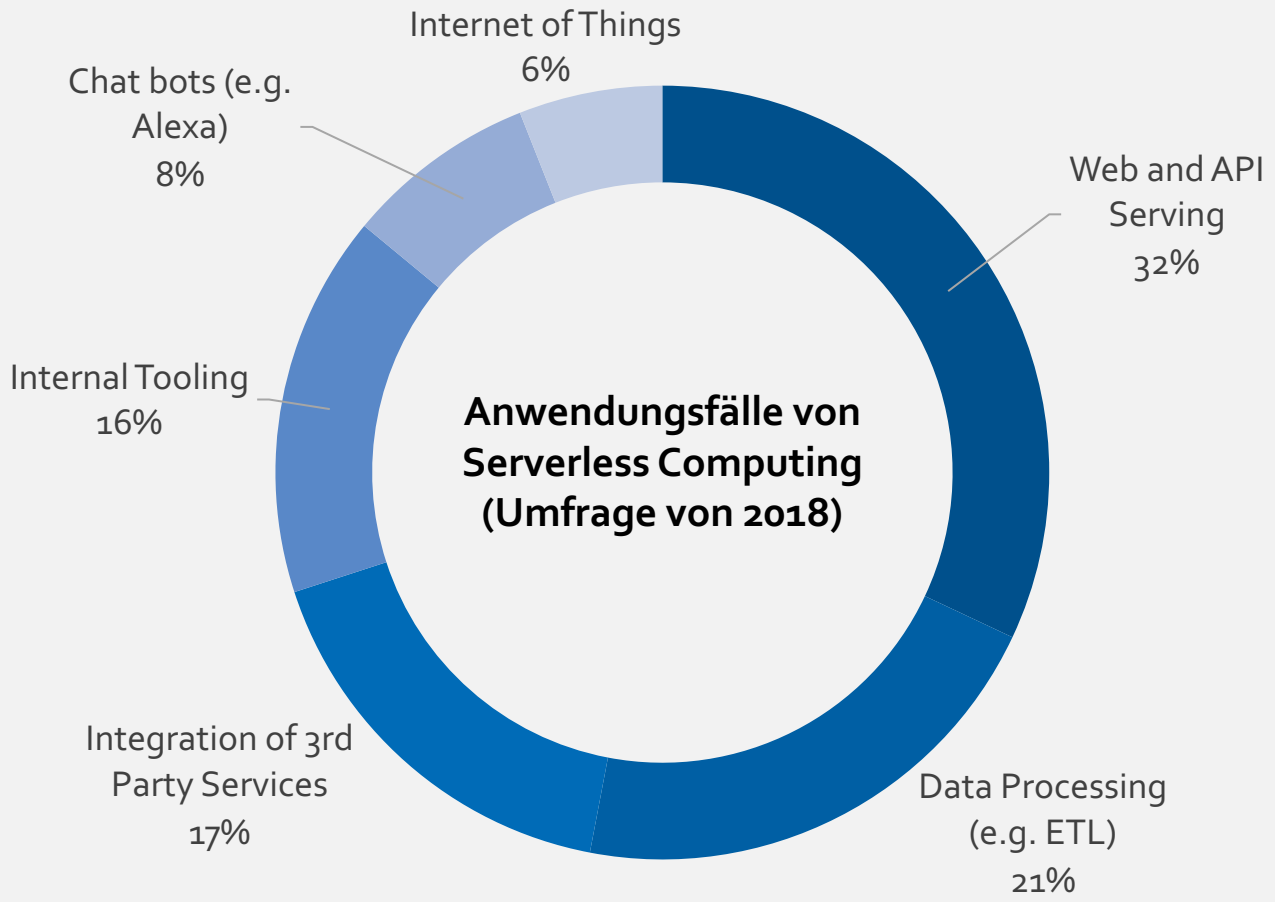
Die Möglichkeit, auf Computing-Ressourcen null Instanzen zu skalieren (Scale-to-Zero), ist eines der entscheidenden Unterscheidungsmerkmale von Serverless Plattformen im Vergleich zu Container-Plattformen (PaaS) oder IaaS mit einem Fokus auf virtuellen Maschinen.

Scale-to-Zero ermöglicht die Vermeidung von Always-On-Komponenten und schließt daher das teuerste Cloud-Nutzungsmuster aus (vgl. Unit 01).

Scale-to-Zero hat dummerweise ein paar architekturelle Auswirkungen.

WOFÜR WIRD SERVERLESS MEIST EINGESETZT

A Berkley View on Cloud Computing



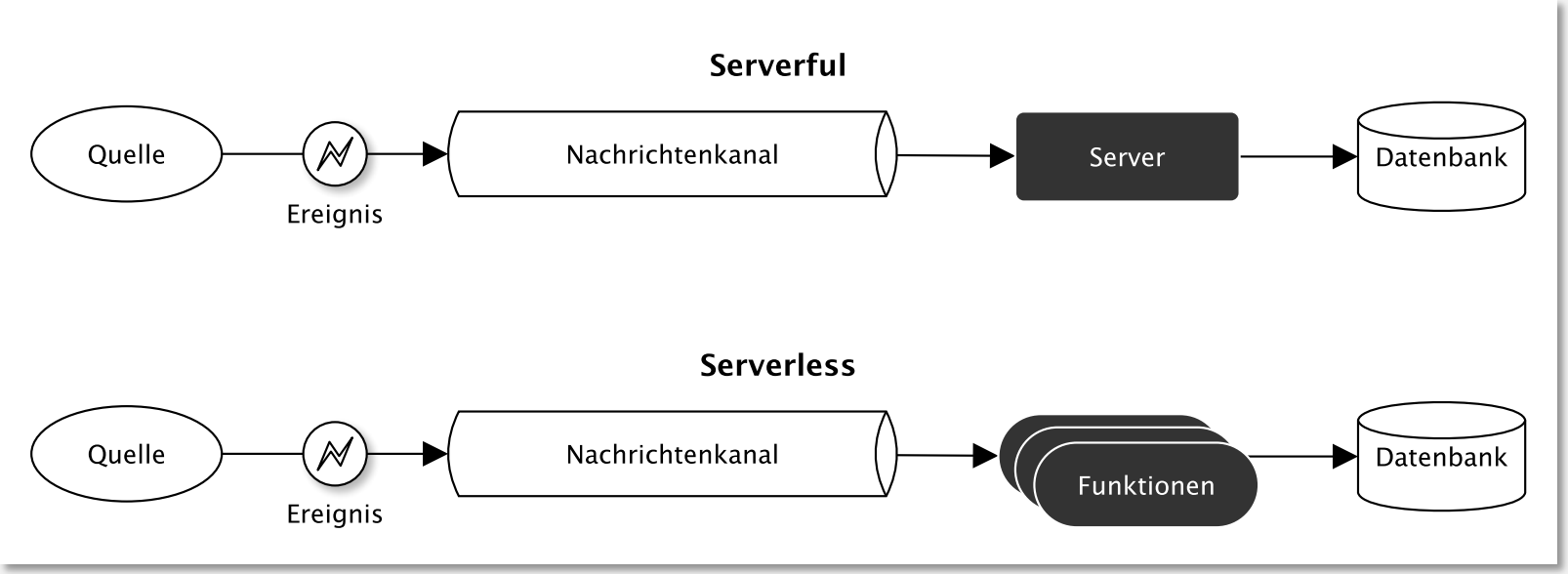
FaaS wird überwiegend für Web und API-Serving, ETL-Data Processing und Integration (Glueing) von Drittanbieter-Diensten genutzt.

Alles Ereignis-gesteuerte Anwendungs-fälle.

Quelle: Jonas et al., A Berkley View on Serverless Computing, 2019, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf

WAS IST SERVERLESS?

A ThoughtWorks Point of View



„Serverless [...] mean applications where server-side logic is still written by the application developer, but, unlike traditional architectures, it's run in stateless compute containers that are event-triggered, ephemeral (may only last for one invocation), and fully managed by a third party."

Mike Roberts

Quelle: https://martinfowler.com/articles/serverless.html

WAS IST SERVERLESS?

Serverless Computing ist ein Cloud-Computing-Modell, bei dem Cloud-Provider Ressourcen Request-basiert zuweisen.

Entwickler von Serverless Anwendungen müssen sich daher nicht mit Kapazitätsplanung, Konfiguration, Verwaltung, Wartung, Betrieb oder Skalierung von Containern, VMs oder physischen Servern befassen (daher Serverless!).

Serverless Computing ist stateless, d.h. es werden keine Zustände in Serverless Compute Units (Functions) zwischen Aufrufen gespeichert. Wenn Functions keine Requests erhalten, werden Functions auch keine Computing-Ressourcen zugewiesen.

Die Preisgestaltung basiert auf dem Pay-as-you-go Prinzip basierend auf der Menge verbrauchter Computing Ressourcen zur Verarbeitung eingehender Requests.

WAS IST SERVERLESS?

CLOUD NATIVE COMPUTING

Serverless Computing kann den Prozess der Entwicklung und Bereitstellung von Code in die Produktion vereinfachen und die Time-to-Market damit reduzieren.

Serverloser Code kann in Verbindung mit anderen Ansätzen wie Microservices oder SOA (Monolithen) verwendet werden.

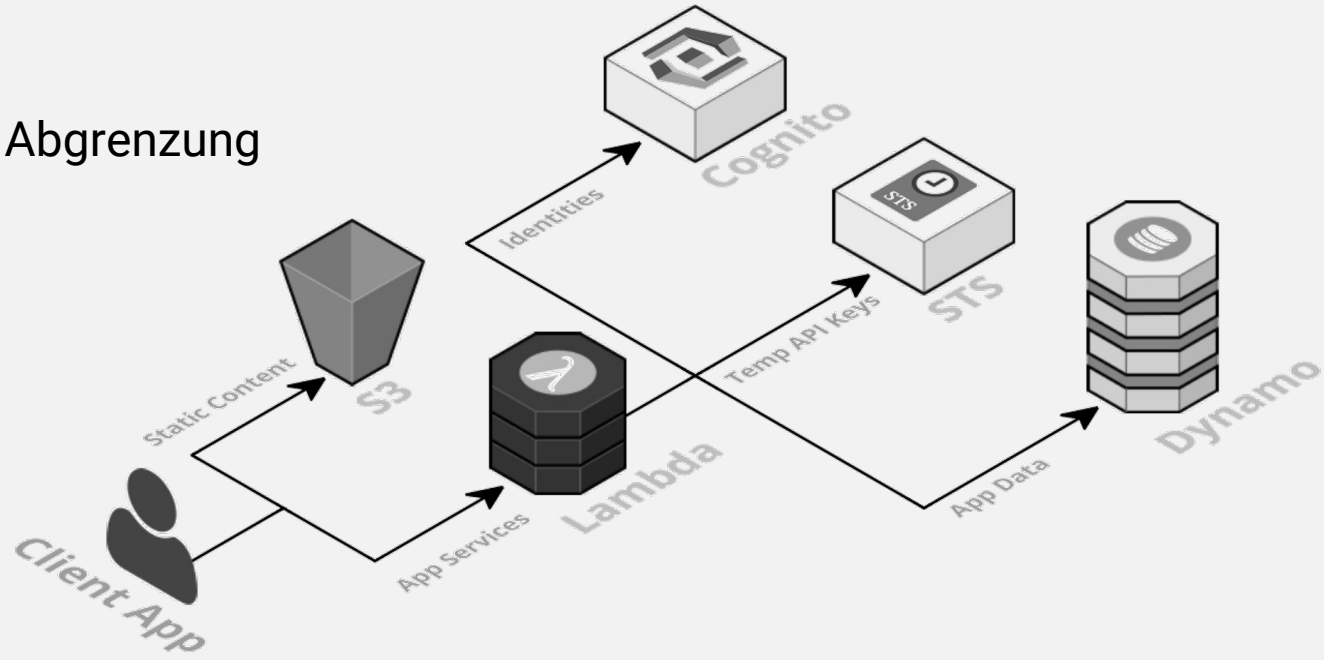
Cloud-native Services können aber auch so geschrieben werden, dass sie rein serverlos sind und sich weder auf IaaS-Infrastrukturen oder PaaS-Container-Plattformen abstützen.

Serverless Computing sollte nicht mit dezentralen Processing-Modellen wie Peer-to-Peer (P2P) oder Volunteer Computing (VC) verwechselt werden. Mittels Serverless Computing bereitgestellte Services sind (aus Consuming Service-Sicht) weiterhin konzeptionell-logische Zentralservices, die dem Client-Server-Modell folgen (auch wenn diese technisch dezentral und verteilt realisiert sein mögen).

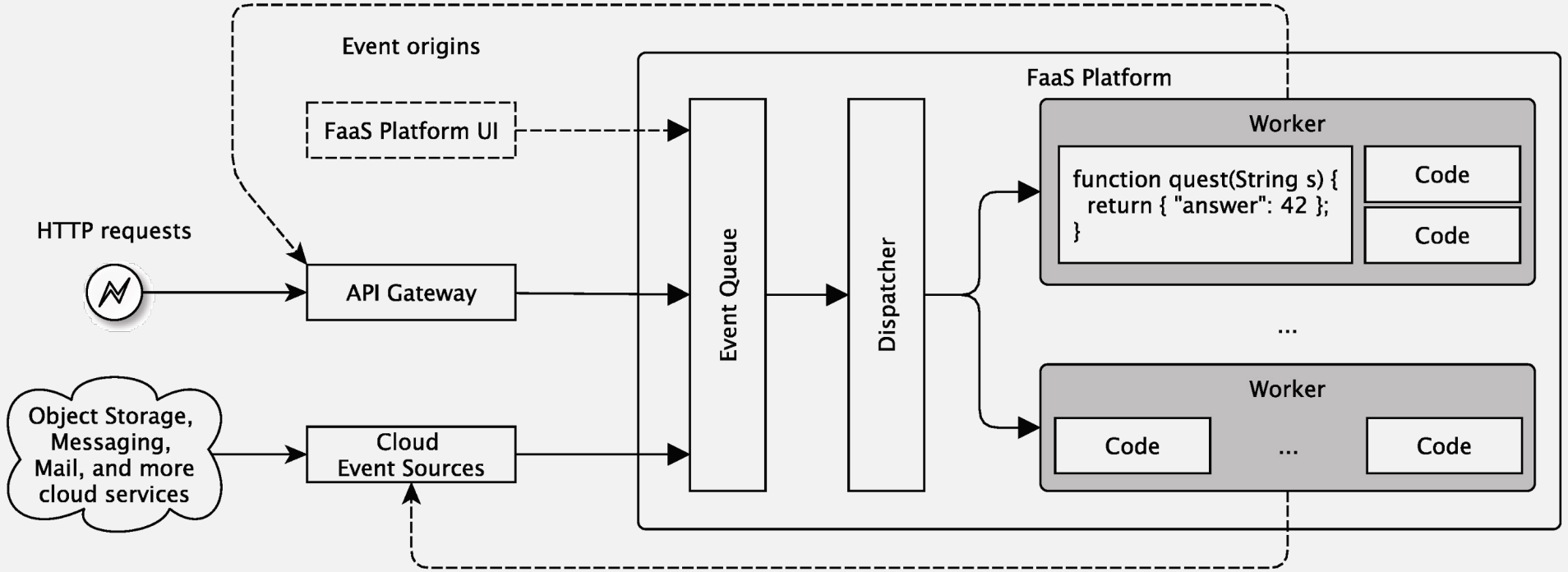
Serverless Provider bieten Laufzeitumgebungen an, die auch als FaaS-Plattformen (Function as a Service) bezeichnet werden und die Anwendungslogik ausführen, jedoch keine Daten speichern.

INHALTE

- Was ist Serverless?
- **Limitierungen von Serverless Ansätzen**
- Wie werden Serverless Architekturen eingesetzt?
- API-Gateways
- Zusammenfassung und Abgrenzung zu Microservices



SERVERLESS PLATFORM ARCHITECTURE



Eine serverlose Plattform ist lediglich ein Ereignisverarbeitungssystem. Ereignisse können z.B. über HTTP gesendet oder von weiteren Ereignisquellen (aus Cloud-Infrastrukturen) empfangen werden. Die Plattform bestimmt dann, welche Funktionen für ein Ereignis registriert sind, sendet das Ereignis an die Funktionsinstanz und wartet auf eine Antwort um diese Antwort an den Requestor weiterzuleiten (im Falle von Request-Response-basiertem Triggering).

Vgl. Unit 06! FaaS

Quelle: Nane Kratzke: A Brief History of Cloud Application Architectures. In *Applied Sciences*; MDPI, 2018, https://doi.org/10.3390/app8081368

LIMITIERUNGEN VON SERVERLESS PLATTFORMEN

CLOUD NATIVE COMPUTING

State, Execution Duration, Startup Latency and Cold Starts

Zustand

- Jeder persistente Zustand einer FaaS-Funktion, muss außerhalb der FaaS-Funktionsinstanz ausgelagert werden.
- Für pure-functional FaaS-Funktionen, ist dies nicht von Belang. Auch „Zwölf-Faktoren-Apps" unterliegen dieser Einschränkung.
- Stateful Funktionen verwenden normalerweise Datenbanken, Cache-Systeme (wie Redis) oder einen Netzwerkdatei- / Objektspeicher (wie S3), um den Status über Anforderungen hinweg zu speichern

Execution Timeouts

- Die Ausführungsdauern von FaaS-Funktionen sind normalerweise begrenzt.
- Typische Grenzwerte sind mehrere Minuten (z. B. 5 bis 10 Minuten).
- Daher sind insb. langlebige Aufgaben nicht für FaaS-Plattformen geeignet.
- In diesen Fällen muss man langlaufende Aufgaben in kürzere Aufgaben dekomponieren und deren Ausführung (z.B. mittels Message-Queues) koordinieren.

Latenz und Cold Starts

- FaaS-Plattform müssen vor einem Event (Request) eine Funktionsinstanz erst einmal initialisieren.
- Diese Startlatenz kann selbst für eine bestimmte Funktion in Abhängigkeit von einer Vielzahl von Faktoren erheblich variieren und zwischen einigen Millisekunden und mehreren Sekunden liegen.
- Die Initialisierung einer Funktion ist entweder ein „Warmstart", bei dem eine Instanz und ihr Hostcontainer aus einem früheren Event wiederverwendet werden, oder ein „Kaltstart", bei dem eine neue Containerinstanz erstellt und der Funktionshostprozess gestartet wird.

Quelle: https://martinfowler.com/articles/serverless.html

LIMITIERUNGEN VON SERVERLESS PLATTFORMEN

CLOUD NATIVE COMPUTING

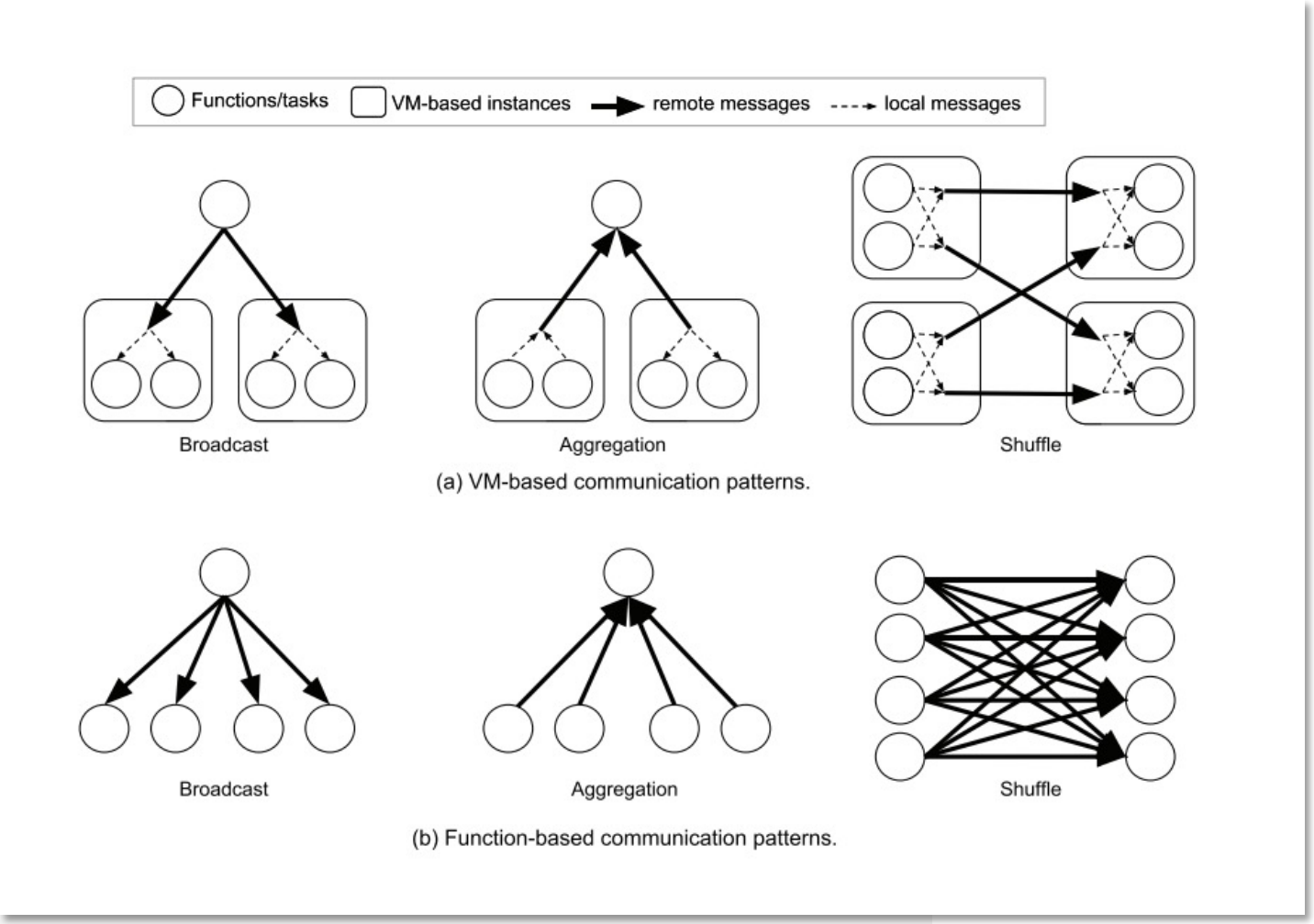
Kommunikations-Muster

Drei gängige Kommunikationsmuster für verteilte Anwendungen: Broadcast, Aggregation und Shuffle.

a) Zeigt diese Kommunikationsmuster für VM-Instanzen an, bei denen jede Instanz zwei Funktionen / Aufgaben ausführt.
b) Zeigt dieselben Kommunikationsmuster für Cloud-Funktionsinstanzen.

Beachten Sie die deutlich geringere Anzahl von Remote-Nachrichten für die VM-basierten Lösungen (a).

VM-Instanzen bieten zahlreiche Möglichkeiten (z.B. Proxies, Caches) um Daten vor dem Senden oder nach dem Empfang lokal über Funktionen / Aufgaben hinweg gemeinsam zu nutzen, zu aggregieren oder zu kombinieren.



Quelle: Jonas et al., A Berkley View on Serverless Computing, 2019, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf

LIMITIERUNGEN VON SERVERLESS PLATTFORMEN

CLOUD NATIVE COMPUTING

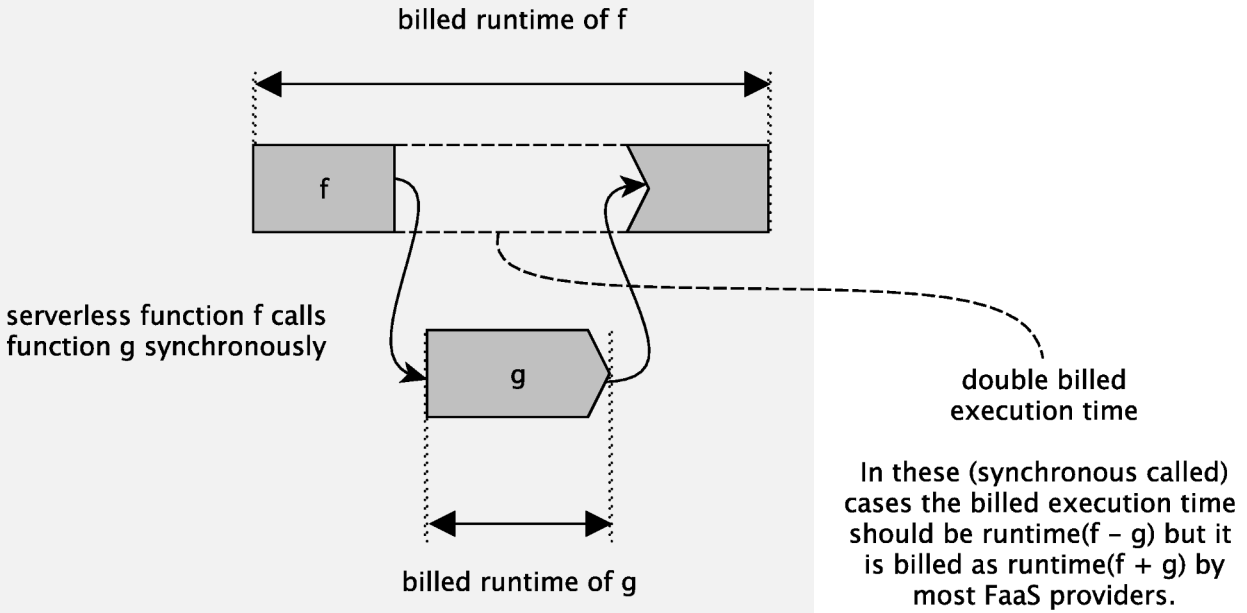
Double-Spending-Problem

Dieses Problem tritt auf, wenn eine Funktion *f* synchron eine andere Funktion *g* aufruft.

In diesem Fall wird dem Verbraucher die Ausführung von *f* und *g* in Rechnung gestellt - obwohl nur *g* Ressourcen verbraucht, weil *f* auf das Ergebnis von *g* wartet.

Um dieses Problem der doppelten Ausgaben zu vermeiden, delegieren viele Serverless-Anwendungen die Komposition von Funktionen (also die Steuerung des Workflows) an Clientanwendungen und Edge-Geräte außerhalb des Bereichs von FaaS-Plattformen.

Dieses Kompositionsproblem führt somit zu neuen, verteilteren und dezentralisierteren Formen von Cloud-nativen Architekturen.



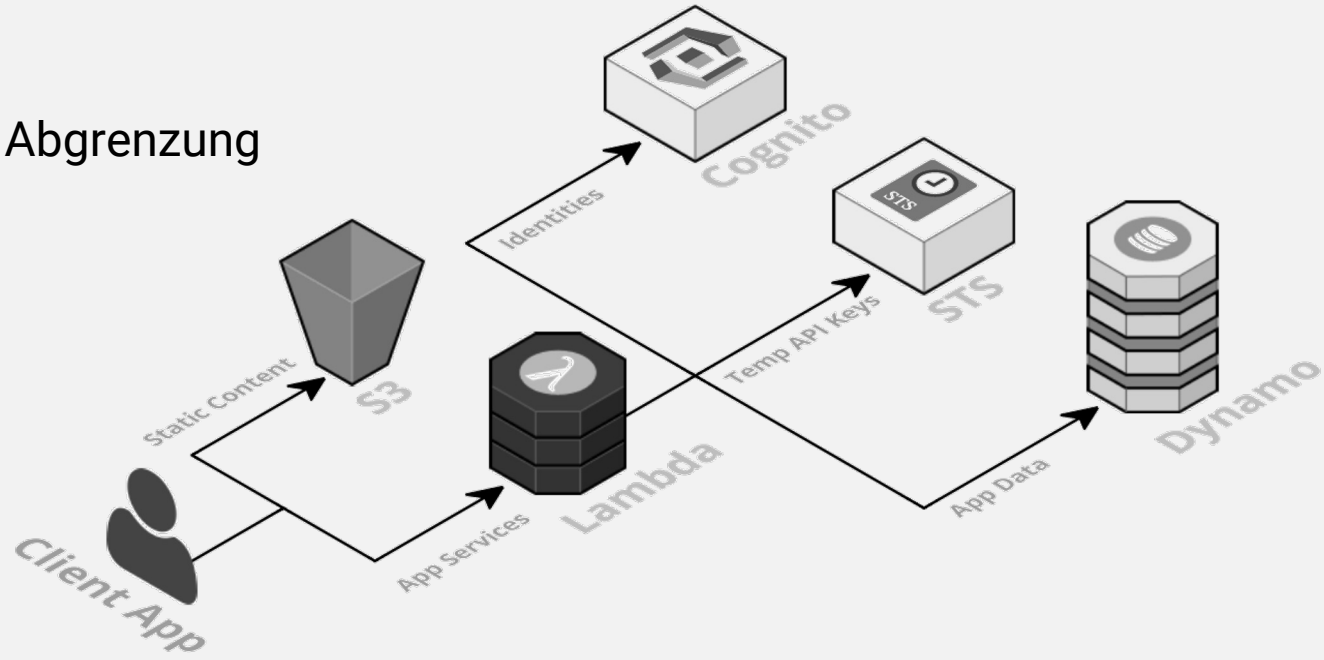
*Was man also **vermeiden** sollte, ist dass sich Funktionen **synchron untereinander aufrufen**!*

OpenWhisk addressiert dieses Problem bspw. mittels Sequences.

Quelle: Baldini et al. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*; Springer, 2017
https://arxiv.org/pdf/1706.03178.pdf

INHALTE

- Was ist Serverless?
- Limitierungen von Serverless Ansätzen
- **Wie werden Serverless Architekturen eingesetzt?**
- API-Gateways
- Zusammenfassung und Abgrenzung zu Microservices

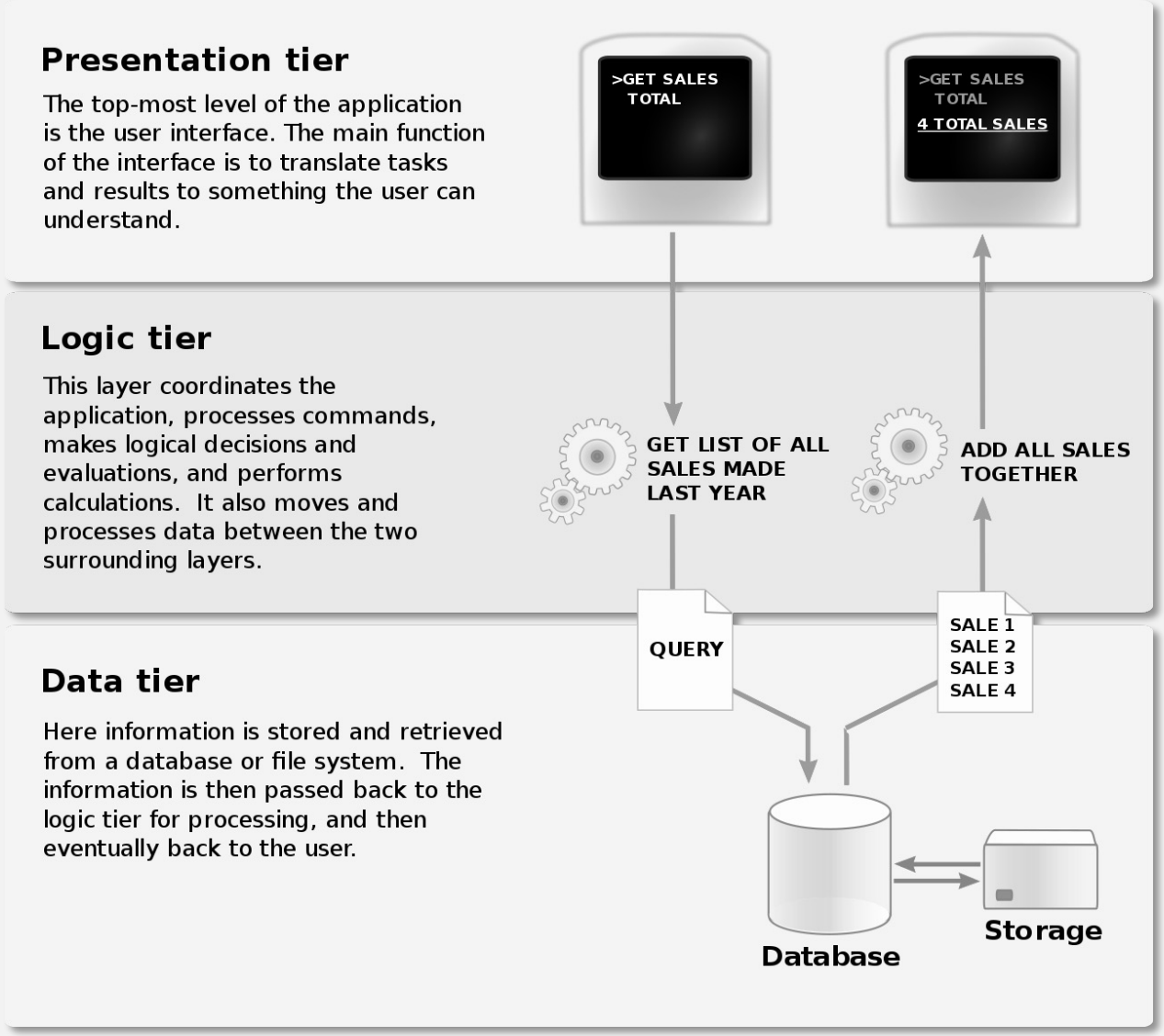


WIE WIRD SERVERLESS EINGESETZT?

Die Drei-Schichten-Architektur ist ein beliebtes Muster für benutzerbezogene Anwendungen.

Diese Architektur besteht aus der **Darstellungsschicht**, der **Logikschicht** und der **Datenschicht**. Die Darstellungsschicht ist die Komponente, mit der die Benutzer direkt interagieren, beispielsweise eine Webseite, die Benutzeroberfläche einer mobile App o. ähnl.

Die Logikschicht enthält den Code, der erforderlich ist, um Benutzeraktionen auf der Darstellungsschicht in die Funktionalität zu übersetzen, die das Verhalten der Anwendung steuert.Die Datenschicht umfasst die Speichermedien (Datenbanken, Objektspeicher, Cache-Speicher, Dateisysteme usw.), auf denen die für die Anwendung relevanten Daten enthalten sind. Abbildung 1 zeigt ein Beispiel einer einfachen Drei-Schichten-Anwendung.

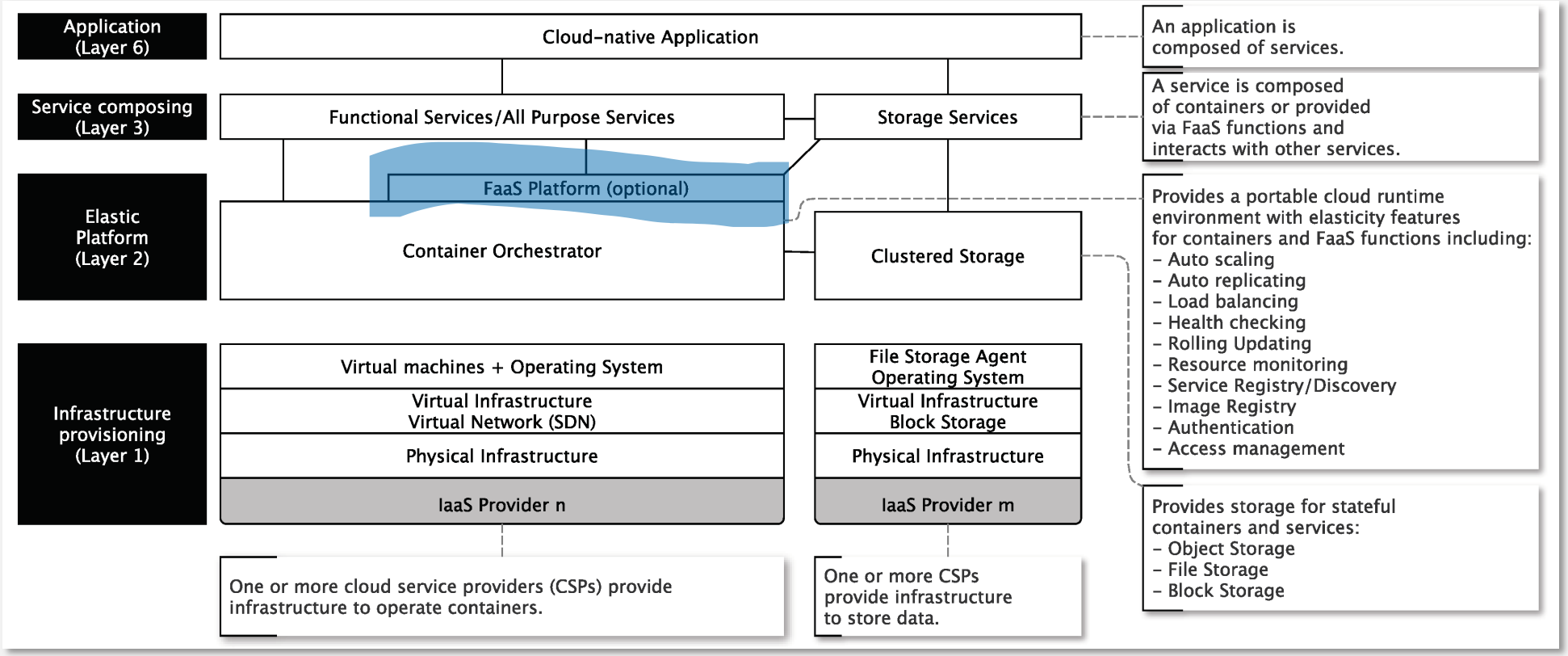


Serverless wird meist auf Ebene des Logic-Tier eingesetzt.

Dies kann in Reinform oder auch in Kombination mit anderen Ansätzen wie Microservices oder SOA-Monolithen erfolgen.

CLOUD-NATIVE REFERENZARCHITEKTUR

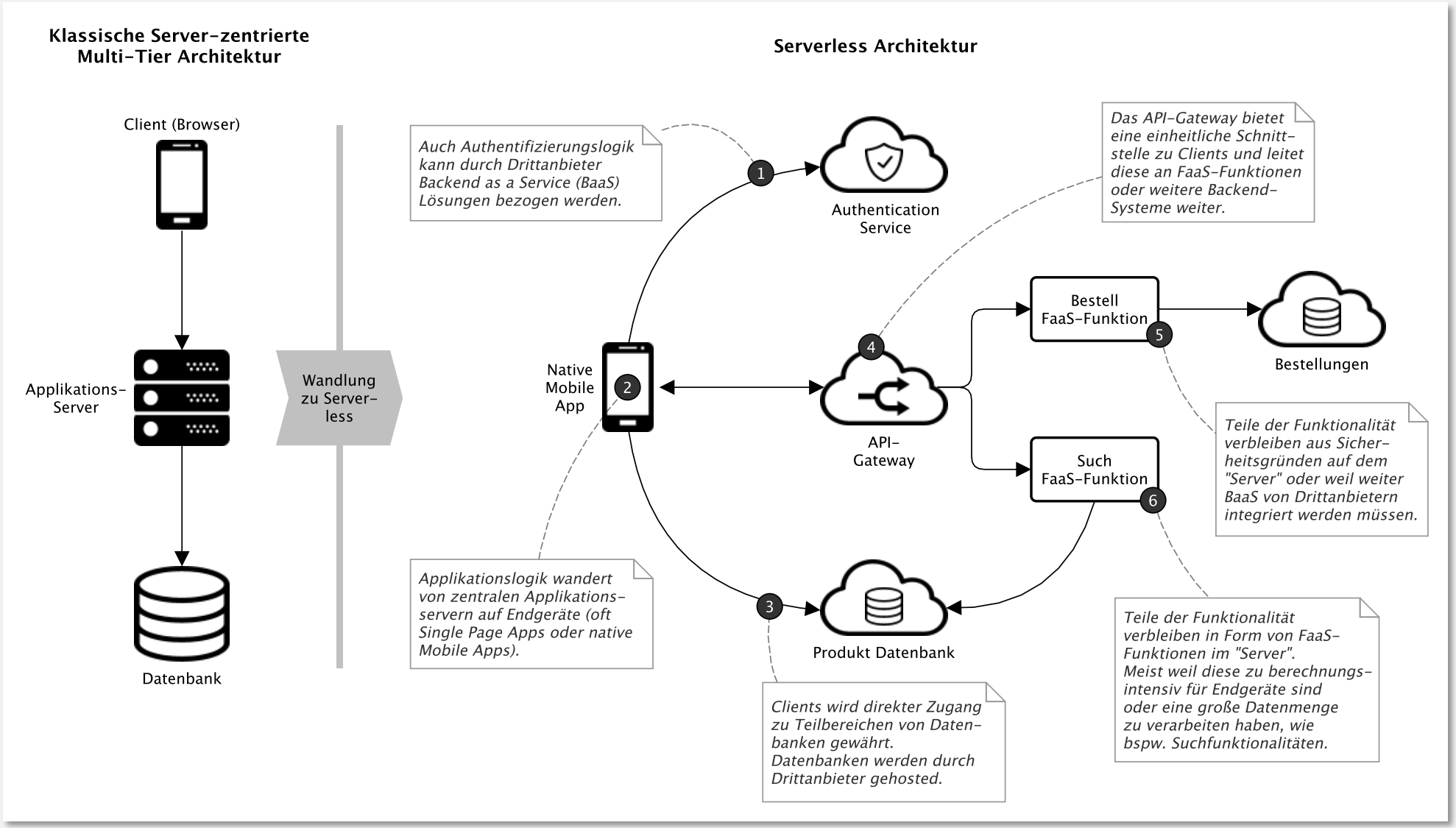
Wo sitzen FaaS-Plattformen?



Diese Referenz-architektur hatten wir schon einmal in Unit 01!

Quelle: Nane Kratzke: A Brief History of Cloud Application Architectures. In *Applied Sciences*; MDPI, 2018, https://doi.org/10.3390/app8081368

DER EFFEKT DES DOUBLE-SPENDING-PROBLEMS



Serverless Architekturen führen zu weniger zentralisierten Kompositionen von Anwendungs-komponenten und Backend-Diensten im Vergleich zu klassischen (schichten-basierten) Anwendungs-architekturen.

Quelle: Nane Kratzke: Cloud-native Computing. Hanser Verlag, 2021

SERVERLESS ARCHITEKTUREN

Konsequenzen für Cloud-native Anwendungsarchitekturen

CLOUD NATIVE COMPUTING

Insbesondere Serverless Architekturen erfordern im Vergleich zu herkömmlichen E-Commerce-Anwendungen ein Redesign der Cloud-Anwendungsarchitektur, die oft nachhaltig bis zum Kontrollfluss durchschlägt.

Serverless Architekturen gehen häufiger weiter als Microservice-Architekturen und integrieren konsequenter Backend-Services von Drittanbietern wie Authentifizierung oder Datenbankdienste.

Die Entwicklung von Funktionen für FaaS-Plattformen werden häufig auf sehr Anwendungs-spezifische, sicherheitsrelevante oder rechenintensive Funktionen beschränkt.

Funktionen, die klassisch auf einem zentralen Anwendungsserver bereitgestellt worden wären, werden jetzt als viele isolierte Micro- oder sogar Nanoservices (Funktionen) bereitgestellt.

Die Integration all dieser isolierten und autarken Services als sinnvolle Endbenutzerfunktionalität wird an Endgeräte delegiert (sehr häufig in Form von nativen mobilen Anwendungen oder progressiven Webanwendungen).

TECHNISCHE ENTSCHEIDUNGEN

in Serverless Architekturen

CLOUD NATIVE COMPUTING

- Querschnittslogik wie bspw. Authentifizierung oder Speicherung wird an externe Dienste von Drittanbietern delegiert.
- Die Komposition und Workflow-Steuerung von Services wird auf Endbenutzer-Clients oder Edge-Geräte verlagert. Dieses Erfordernis kann durch API-Gateways teilweise etwas reduziert werden.
- Dies bedeutet, dass selbst die Downstream-Service Orchestrierung nicht mehr vom Service-Provider selbst, sondern vom Service-Consumer über bereitgestellte Anwendungen durchgeführt wird. Solche Client- oder Edge-Geräte verbinden Services von Drittanbietern häufig direkt.
- Diese Endbenutzer-Orchestrierung hat zwei interessante Auswirkungen:

1. Der Service-Consumer stellt jetzt die für die Service-Orchestrierung erforderlichen Ressourcen bereit.
2. Da die Service-Komposition außerhalb des Bereichs der FaaS-Plattform erfolgt, werden immer Probleme wie das Double-Spending-Problem vermieden.

KONZENTRATION AUF DAS WESENTLICHE

in Serverless Architekturen

- In Serverless Architekturen werden mittels FaaS-Plattformen meist nur dann Anwendungs-spezifische Funktionen bereitgestellt, wenn:

    1. Funktionalität ist sicherheitsrelevant ist und muss vom Dienstanbieter in einer kontrollierten Laufzeitumgebung ausgeführt werden.
    2. Die Funktionalität ist für Consumer-Clients oder Edge-Geräte zu Processing- oder Datenintensiv.
    3. Die Funktionalität ist so domänen-, Problem- oder Anwendungs-spezifisch, dass einfach kein externer Drittanbieter-Dienst vorhanden ist.

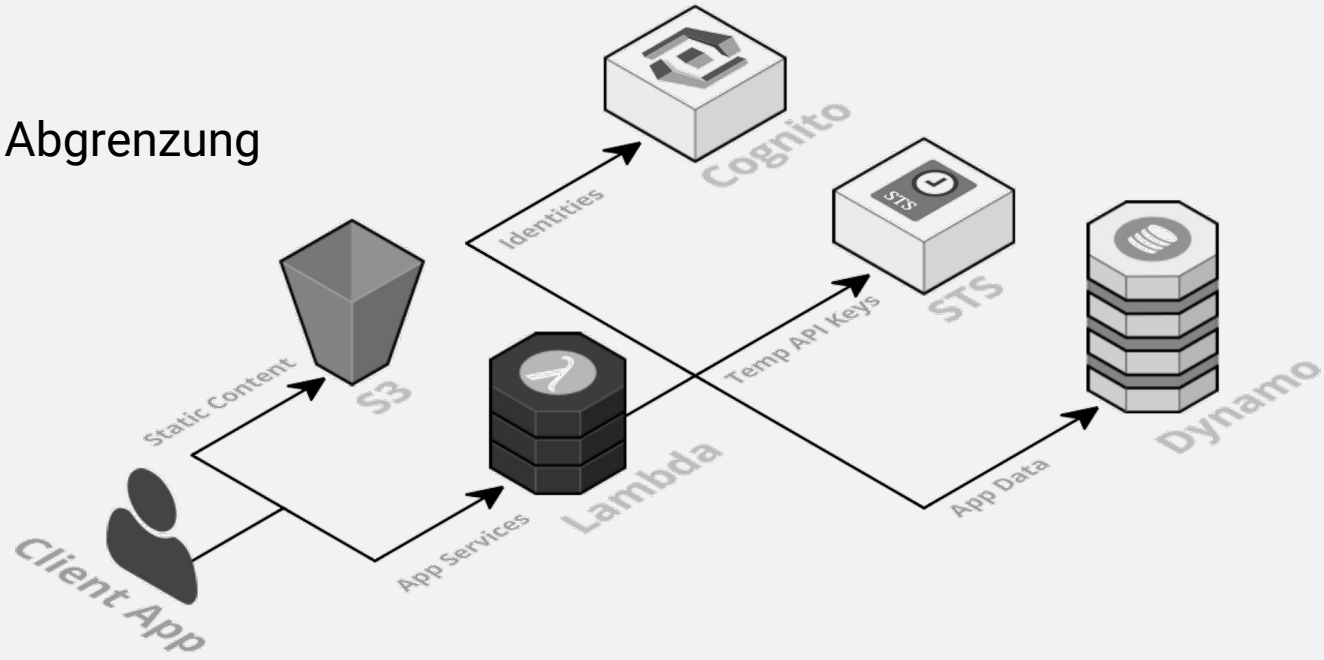
- Endpunkte für Anwendungs-spezifischen Funktionen werden über API-Gateways bereitgestellt. Dabei werden meist HTTP- und REST-basierte / REST-ähnliche Kommunikationsprotokolle bevorzugt.

CLOUD NATIVE COMPUTING

Es lässt sich also ein Trend bei serverlosen Architekturen beobachten, dass diese Art von Architektur dezentraler und verteilter ist, von Drittanbietern unabhängig bereitgestellte Dienste gezielter nutzt und daher im Vergleich zu Microservice-Architekturen wesentlich dezentraleren und „Mesh-artigen" Charakter hat.

INHALTE

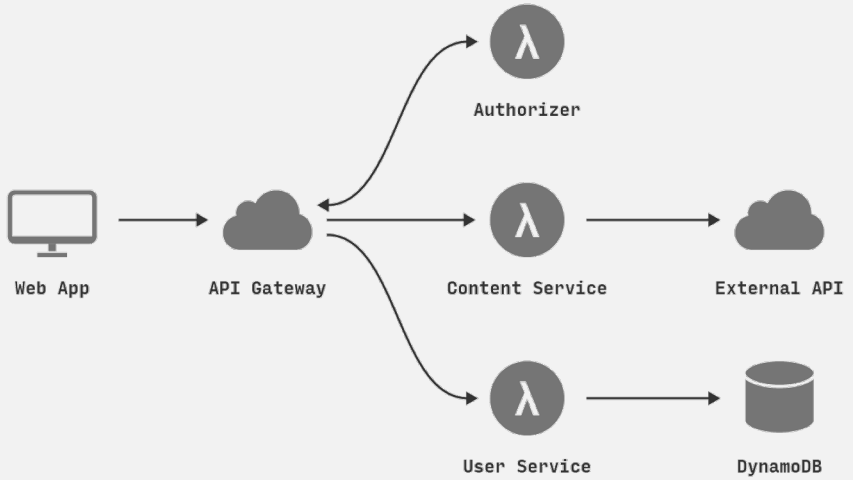
- Was ist Serverless?
- Limitierungen von Serverless Ansätzen
- Wie werden Serverless Architekturen eingesetzt?
- **API-Gateways**
- Zusammenfassung und Abgrenzung zu Microservices



API-GATEWAYS

API Gateways werden mehr und mehr zu einem fundamentalem Bestandteil einer Cloud-nativen Architektur. Sie bilden einen zentralen Request-Response-basierten Zugangspunkt zu den Backend-Services.

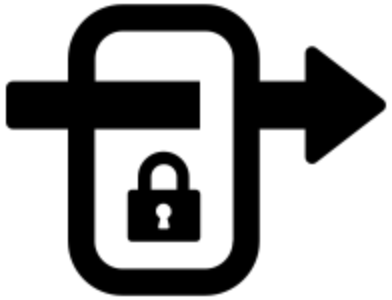
Durch die Einführung eines API Gateways, das eine Grenze zu den Backend-Systemen bildet (daher auch ab und an "Edge-Service" genannt wird) und als zentraler Zugangspunkt dient, können diverse Orchestrations-Nachteile in Serverless-Architekturen (aber auch Microservice Architekturen) kompensiert werden.



WARUM WERDEN API-GATEWAYS VERWENDET?

CLOUD NATIVE COMPUTING

Clients von der Backend-Implementierung entkoppeln



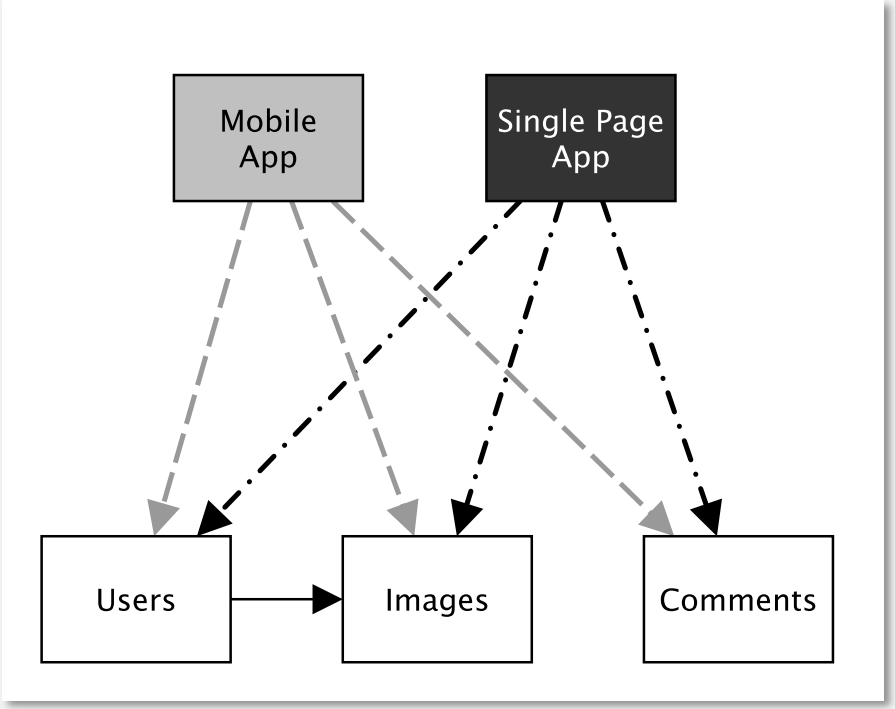
Die grundlegende Funktion eines API-Service ist es, Remote-Anfragen entgegenzunehmen und zu beantworten. Die Komplexität nimmt dabei mit der Anzahl bereitgestellter APIs zu. Hieraus ergeben sich weitere Anforderungen:

- APIs müssen vor Überlastung und Missbrauch geschützt werden und verwenden deshalb einen Authentifizierungsservice und Rate Limiting.
- Man möchte die Nutzung von APIs mittels Analyse- und Überwachungstools überwachen, um zu bestimmen für welche Zwecke bereitgestellte APIs verwendet werden.
- Wenn APIs monetarisiert werden sollen, muss eine Verbindung zu einem Fakturierungssystem hergestellt werden.
- In Service-of-Service-Architekturen können zur Beantwortung einzelner Requests Dutzende von Downstream Services abgefragt werden müssen.
- Im Laufe der Zeit werden neue API-Services hinzukommen oder entfernt werden. Dennoch sollten all diese Änderungen nicht bis zum Client durchschlagen, der alle Services stets am gewohnten Platz finden sollte.

API-GATEWAYS

Nachteile von Service-Architekturen

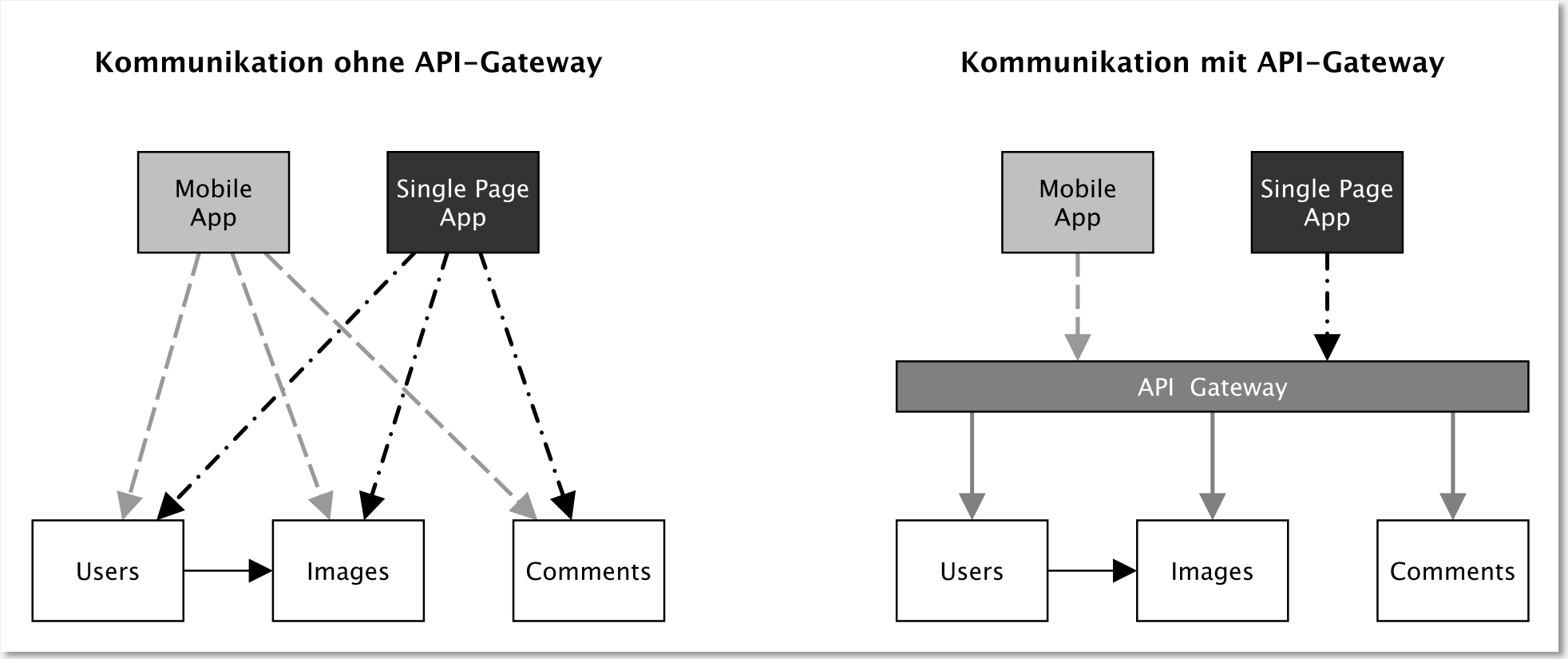
- Viele Kommunikationsverbindungen für Clients., u.a. muss jeder Client jeden Backend-Service kennen.
- Das wird dann problematisch, wenn sich die Backends verändern.
- Same-Origin-Policy: Damit die Clients mit den verschiedenen Backend Services kommunizieren dürfen, muss bei jedem Backend eine Cross-Origin-Resource-Sharing-(CORS)-Ausnahme definiert sein.
- Zusätzlicher Schutz von internen Endpunkten (bspw. zwischen Users <-> Service). Da User und Service öffentliche Endpunkte haben, ist die interne Schnittelle auch von den Clients erreichbar, oft nicht gewollt.
- Cross-cutting concerns, die in jedem Backend-Service implementiert werden müssen. Dazu zählt beispielsweise eine erste Authentifizierung, SSL-Terminierung oder das Handling von (Security-)Headern wie etwa CSRF oder HSTS. Aber auch Schutzmechanismen wie *Rate Limiting* und *Throttling* sind hier zu nennen.



Service-of-Service Ansätze kommen mit einigen Nachteilen für Clients einher.

API-GATEWAYS

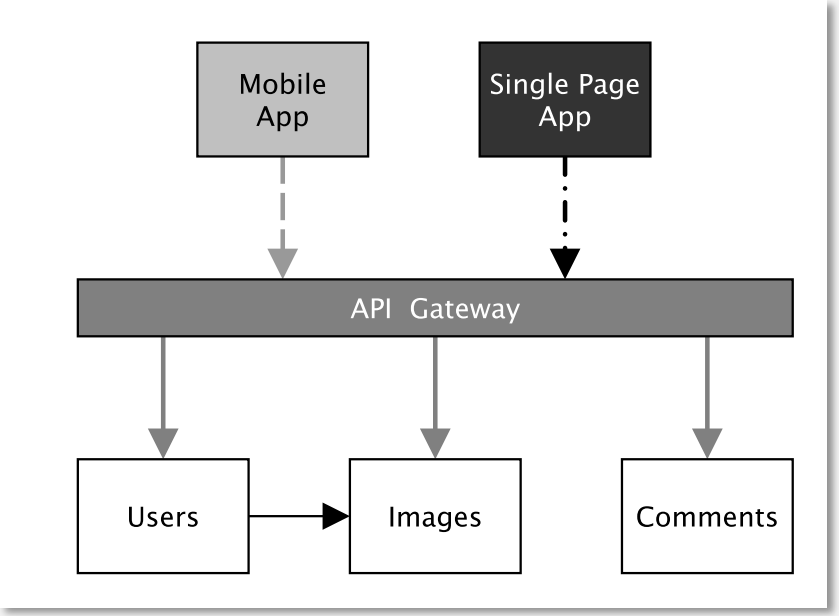
Reduzierung von Kommunikationskomplexität für Clients



API-GATEWAYS

Sinn und Zweck

- Die Kommunikation der Clients zu den Downstream Services verläuft immer über das API-Gateway.
- Die Clients müssen folglich auch nur noch dessen Adresse kennen – das Gateway leitet die Anfragen dann an den spezifischen Service weiter.
- Somit ist eine Hauptaufgabe eines API-Gateways das Reverse Proxying.
- In dieser zentralen Stelle lassen sich querschnittliche Belange umsetzen, sodass diese nicht von jedem Service separat bereitgestellt werden müssen, wie bspw. einheitliche CORS-Regeln (*Same-Origin-Policy) oder Authentication, etc.*



Insb. das Reverse Proxying eines API-Gateways ist von zentraler Bedeutung, da verschiedene Endpunkte unter einem Namen zusammengefasst werden können. Ohne Reverse Proxying müssten die drei beispielhaften Services Users, Comments und Images unter drei verschiedenen Adressen erreicht werden. Gibt es Clients, die man nicht kontrollieren kann (z. B. eine ausgerollte Mobile App ohne Update-Zwang), ist man in den möglichen Änderungen sehr beschränkt, die man an den Endpunkten vornehmen kann.

CLOUD NATIVE COMPUTING

Merke:
Grundfunktionalität eines API-Gateways ist das Reverse Proxying

API-GATEWAY

Ja oder nein?

Die zwei Kernargumente für ein Gateway sind die Einfachheit für Clients und die Kapselung der Service-Architektur. Hat man mehr als einen Endpunkt, ist es oft hilfreich diese zu bündeln. Die Verwaltung von "Endpunktlisten" in Clients sollte vermieden werden, da sie schnell unübersichtlich werden können.

Da Architekturen sich gem. der DevOps Philosophie evolutionär entwickeln, werden sich im Laufe der Zeit auch die Endpunkte immer wieder ändern. API-Gateways vermeiden, dass Clients Änderungen direkt ausgesetzt werden und ermöglichen es, APIs kompatibel zu halten, selbst wenn sich die Service-Landschaft sich immer wieder verändert.

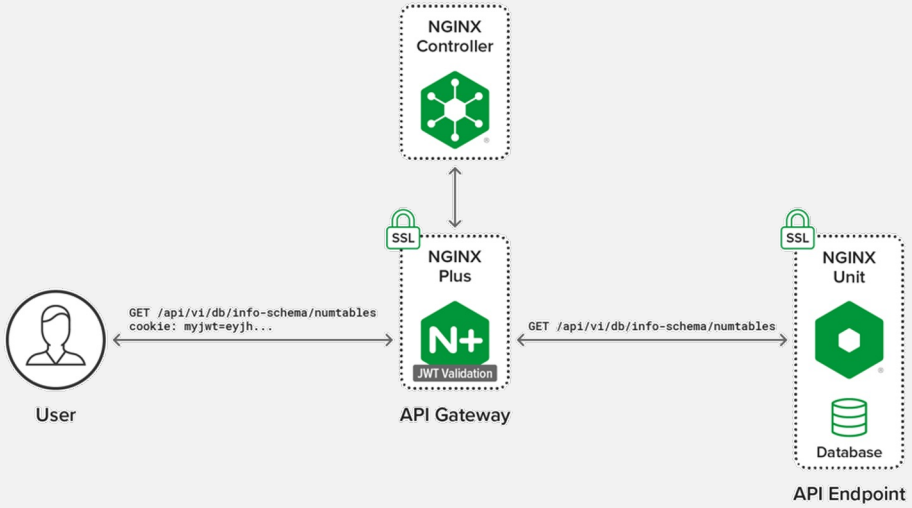
Zu berücksichtigende Randbedingungen:

- Ein API-Gateway ist ein konzeptioneller Single Point of Failure, der in die Architektur eingebracht wird.
- Damit ergibt sich eine Reihe von Anforderungen an die Verfügbarkeit, Belastbarkeit und an die Fähigkeiten des Gateways.
- Ein Gateway muss mindestens so verfügbar sein, wie die Höchstanforderung an einen beliebigen Service dahinter.
- Es muss auch alle Kommunikationstechnologien unterstützen, welche die aufgerufenen Services benötigen (beispielsweise HTTP/2 oder Websockets).

API-GATEWAY

Empfehlungen zum „richtigen" Einsatz

- Einige API-Gateway Produkte versprechen unzählige Features mit entsprechender Auswirkung auf die Komplexität.
- Ein Austausch eines API-Gateways ist meist problemlos in Produktion möglich und für die Clients völlig transparent.
- Deshalb lohnt es sich, die exakten Anforderungen abzuwarten und komplexere Lösungen erst bei Bedarf später einzusetzen.
- Auch sollte man es vermeiden, Geschäftslogik oder "Features" aus den Services in das Gateway zu verschieben.
- Das führt oft zu unübersichtlichen und schwer zu wartenden Systemen mit vielen Abhängigkeiten.



Einfache API-Gateways lassen sich auch mit Reverse Proxies wie NGINX realisieren. NGINX ist der Standard Ingress Controller in Kubernetes. Anders ausgedrückt. Ein Kubernetes Ingress kann oft als API-Gateway schon ausreichen.

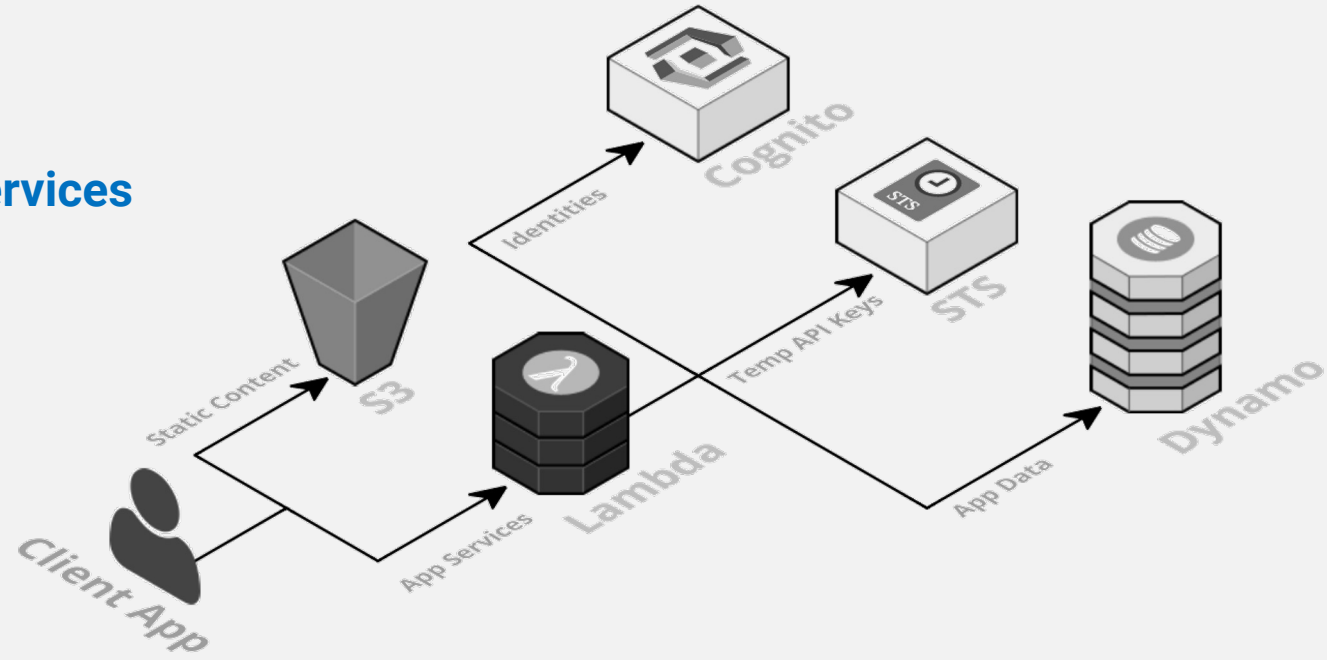
CLOUD NATIVE COMPUTING

KISS keep it simple stupid

Avoid overambitious API Gateways

INHALTE

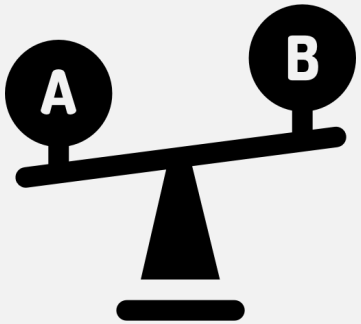
- Was ist Serverless?
- Limitierungen von Serverless Ansätzen
- Wie werden Serverless Architekturen eingesetzt?
- API-Gateways
- **Zusammenfassung und Abgrenzung zu Microservices**



VOR- UND NACHTEILE

Von Serverless Architekturen

Zusammenfassend sollte man die folgenden Vor- und Nachteile von Serverless Architekturen bei der Wahl geeigneter Anwendungsfälle für Serverless Services berücksichtigen.



Vorteile	Nachteile
RESSOURCE EFFIENZIENZ	
Auto-Scaling (durch event stimulus)	Ausführungsdauer von Functions begrenzt
Meist Reduzierte Betriebskosten	Startup-Latenzen müssen berücksichtigt werden
Scale to Zero Fähigkeit (keine Always-on Komp.)	Funktionsausführung unterliegt gewissen Varianzen (mal schneller, mal langsamer)
	Funktionen können keinen Zustand über Funktionsaufrufe hinweg speichern (stateless)
	Mittels Externalisierung von Zuständen (cache, key/value stores, etc.) kann dies kompensiert werden, dennoch diese um Größenordnungen langsamer Netzwerkzugriffe.
	Double spending problem (FaaS Funktionen rufen andere FaaS Funktionen auf)
BETRIEB	
Einfacheres Deployment	Erhöhte Angriffsoberfläche (Attack Surface)
Einfacherer Betrieb (u.a. wegen Auto-Scaling)	Jeder Endpoint kann Vulnerabilities beinhalten
	Fehlende “Firewall“ monolitischer Web-Applikationen
	Teile der Applikationslogik wandern in Clients (die meist nicht unter Kontrolle der Service Provider sind)
	Erhöhter Vendor Lock-in (kein nennenswerter FaaS Standard für API Gateways oder FaaS Laufzeitumgebungen)
ENTWICKLUNGSGESCHWINDIGKEIT	
Entwicklungsgeschwindigkeit	Erhöhte Komplexität im Client
Einfacheres Unit Testing von Funktionen	Applikationslogik wandert auf die Client-Seite
Schnelleres Time-to-market	Code Wiederholung auf Client-Seite für unterschiedliche Client-Plattformen
	Kontrollfuss wird durch Client-Seite bestimmt, um Double-sending-problem zu umgehen
	Erhöhte Komplexität beim Integration Testing
	Noch unzureichende Integrations-Test-Tools

ZUSAMMENFASSUNG

Und Abgrenzung zu Microservices

CLOUD NATIVE COMPUTING

- Serverlose Architekturen sind Anwendungsdesigns, die konsequent BaaS-Dienste (Backend as a Service) von Drittanbietern einbinden.
- Benutzerdefinierter Code wird auf ein Minimum reduziert, meist auf den Logic-Tier beschränkt und in verwalteten, kurzlebigen Containern auf einer FaaS-Plattform (Functions as a Service) ausgeführt.
- Dadurch entfällt bei solchen Serverless-Architekturen ein Großteil der Notwendigkeit für Always-on Komponenten.
- Serverless Architekturen können so von erheblich reduzierten Betriebskosten, Komplexität und schnelleren Entwicklungszyklen (Time-to-Market) profitieren.
- Dies wird wird allerdings durch eine erhöhte Abhängigkeit von Cloud-Providern und noch vergleichsweise unreifen unterstützenden BaaS-Diensten erkauft.

Abgrenzung zu Microservices

- Serverless und Microservice Architekturen widersprechen sich nicht.
- Sie können gemeinsam auftreten und sind kompatibel, da sie viele Gemeinsamkeiten teilen (bspw. die Präferenz für Statelessness aufgrund des Fokus auf horizontaler Skalierbarkeit).
- Microservices eignen sich für lang laufende, komplexere Dienste mit hohem Ressourcen- und Managementbedarf.
- Demgegenüber führen Serverless-Architekturen Funktionen nur bei Bedarf aus und eignen sich damit insbesondere für die ereignisgesteuerte Abläufe.
- Serverless Services können in Microservice-Architekturen als Dienste auftreten und andersherum.

KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de