Fachartikel

Cloud-native, Microservices, Domain-Driven Design, Kubernetes

Alles nur ein Hype?

von Eberhard Wolff

Software-Entwicklung unterlag schon immer Hypes. Im Moment reden alle über Microservices und Cloud-native. Aber helfen diese Ansätze wirklich weiter?

Cloud-Umgebungen sind nicht mehr neu. Amazon ist bereits 2006 den Schritt in die Cloud gegangen. Ein häufiges Missverständnis bezüglich der Cloud ist, dass es um eine Kostenreduktion im Betrieb geht. Zumindest für die damalige Situation bei Amazon kann das kaum stimmen. Es gab damals keine schlüsselfertigen Cloud-Lösungen. Amazon musste also eine erhebliche Investition tätigen, um die notwendige Infrastruktur zu schaffen.

Cloud-native

Der wichtigste Vorteil der Cloud ist, dass sie eine Umgebung bietet, die Probleme wie Skalierung löst. Software, die in der Cloud deployt ist, kann praktisch beliebig skalieren, wodurch dieses Problem gelöst ist. Entwickler können sich um andere Aufgaben kümmern, die das Geschäft direkt unterstützen.

Außerdem erlaubt die Cloud den Teams, mehr Verantwortung zu übernehmen. Sie können ihre Umgebungen im Rahmen der Angebote der Cloud beliebig konfigurieren und mehr Einfluss auf den Betrieb nehmen. So können Aufgaben, die klassisch beim Anwendungsbetrieb liegen, durch die Teams übernommen werden. Diese Teams kennen die Anwendungen oft besser. Außerdem können sie die Anwendungen so modifizieren, dass der Anwendungsbetrieb einfacher wird. So wachsen Entwicklung (Dev) und Betrieb (Ops) zu DevOps zusammen.

Warum eigentlich Continuous Delivery?

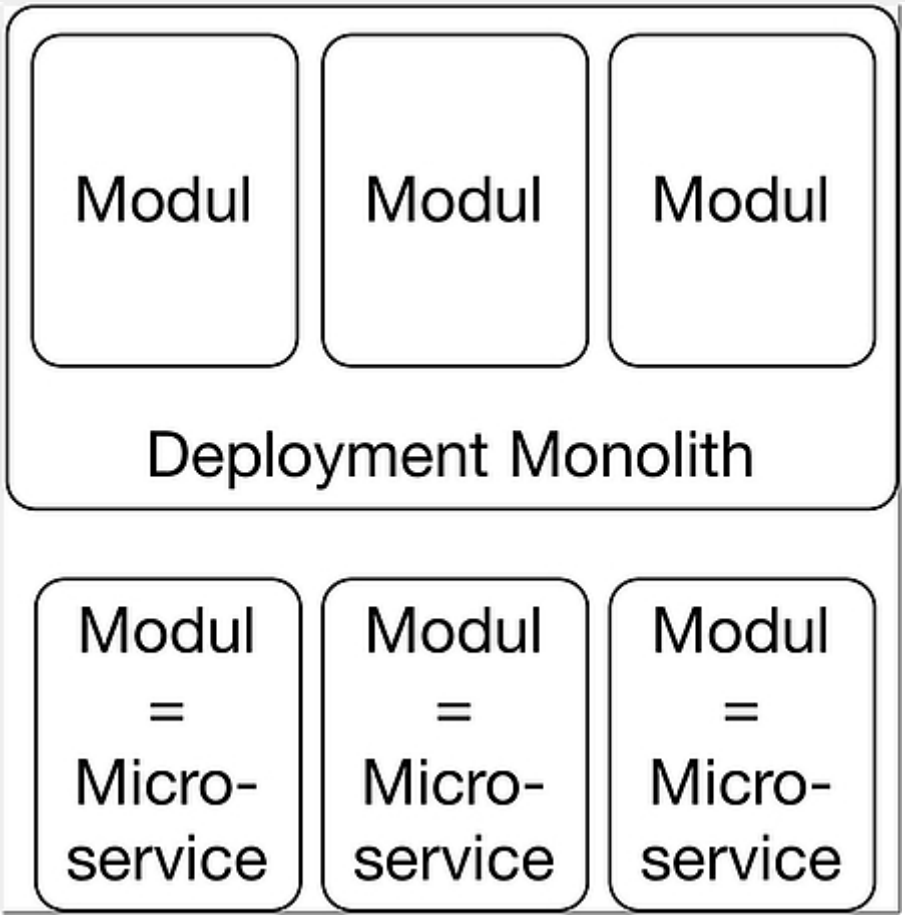

Abb. 1: Microservices vs. Deployment-Monolithen

Und schließlich erleichtert die Cloud Infrastruktur-Automatisierung. Das vereinfacht Continuous Delivery [1], also das regelmäßige Ausliefern der Software. Continuous Delivery hat offensichtliche Vorteile: Änderungen gehen schneller in Produktion. Daher kann man Features schneller ausrollen. Besseres Time-to-Market kann einen Wettbewerbsvorteil darstellen. Eigentlich ist auch offensichtlich, dass regelmäßige Deployments weniger häufig fehlschlagen und damit das Risiko geringer ist. Schließlich werden weniger Änderungen pro Deployment ausgeliefert, und wenn man etwas häufiger macht, ist man geübter darin.

Mittlerweile gibt es eine Studie [2], die belegt, dass auch die Zeit zum Beheben eines Ausfalls bei Systemen mit Continuous Delivery kürzer ist. Da Deployments weniger häufig fehlschlagen und Ausfälle schneller behoben werden können, müsste man Systeme, die sehr verfügbar sein sollen, eigentlich regelmäßig ausliefern.

Außerdem zeigt die Studie, dass Projekte, die oft deployen, 50 % der Entwicklerzeit auf neue Features verwenden, während andere Projekte nur 30 % aufwenden können. Gemessen an neuen Features ist die Produktivität der Teams also verbessert. Gleichzeitig geht der Anteil der Arbeit, der sich mit Sicherheitsproblemen, von Endbenutzern gemeldeten Fehlern und Kundensupport beschäftigt, zurück. Dies weist darauf hin, dass die Qualität der Software aus Benutzerperspektive ebenfalls besser wird. Continuous Delivery hat also so viele Vorteile in verschiedenen Bereichen, dass dieses Vorgehen in jedem Fall sinnvoll erscheint.

Die Studie zeigt auch, ab welcher Deployment-Frequenz man mit diesen positiven Effekten rechnen kann. Sie treten ein, wenn man mehrmals pro Tag deployt. Eine solche Deployment-Frequenz ist ohne eine dynamische Infrastruktur, wie sie die Cloud bietet, kaum denkbar.

Viele Anwendungen werden aber nur einmal im Quartal ausgeliefert und müssen wochenlang getestet werden. Solche Anwendungen mehrmals pro Tag zu deployen würde bedeuten, dass man die Geschwindigkeit irgendwo um mehrere Größenordnungen erhöhen muss. Man kann nur so häufig deployen, wenn man die Software in kleine, unabhängig deploybare Artefakte aufgeteilt hat. Genau das sind Microservices [3][7]: kleine, unabhängig deploybare Module einer großen Anwendung.

Microservices sind cloud-native

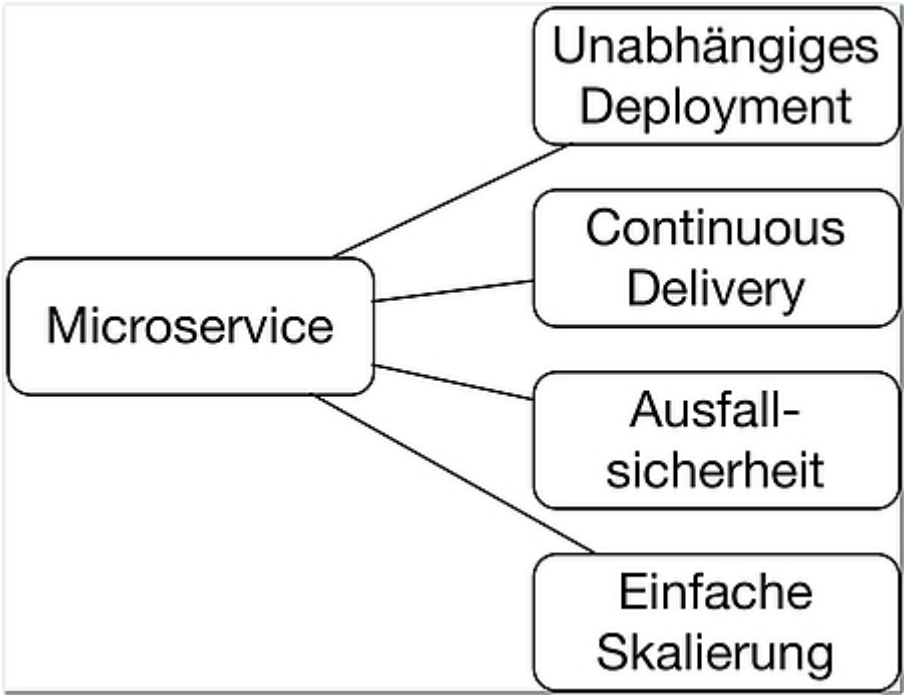

Abb. 2: Gerade für die Cloud bieten Microservices Vorteile.

Microservices helfen in der Cloud aber auch aus anderen Gründen. Cloud-Infrastrukturen sind nicht besonders zuverlässig. Maschinen können ausfallen oder neu starten. Sogar ganze Rechenzentren können ausfallen, ohne dass die Service Level Agreements der Cloud verletzt werden. Ebenso kann die Kommunikation über das Netzwerk ausfallen. Es scheint kaum machbar, auf solchen Umgebungen ohne Ausfälle Software zu betreiben. Aber einige der Systeme mit der höchsten Verfügbarkeit basieren genau auf solchen Infrastrukturen.

Auf der anderen Seite bieten Clouds nämlich an, dass man Anwendungen verteilt in mehreren Rechenzentren laufen lassen kann. Und wegen der Infrastruktur-Automatisierung können Anwendungen auch in anderen Rechenzentren automatisch gestartet werden, oder es können zusätzliche Instanzen gestartet werden, wenn die Last dies notwendig macht. Die Architektur muss aber zeitweise Ausfälle kompensieren können und Teile der Anwendung in neuen Rechenzentren neu starten können. Microservices teilen Anwendungen in getrennte Prozesse auf, die unabhängig voneinander laufen. Der Ausfall eines Servers wird daher wahrscheinlich nur einige Microservices ausfallen lassen. Außerdem können Microservices aufgrund ihrer geringen Größe schneller neu gestartet werden als große, monolithische Anwendungen.

Zusätzlich können Microservices feingranularer skalieren: Jeder Microservice kann in mehreren Instanzen auf verschiedenen Rechnern gestartet werden und so mit einer höheren Last umgehen. Das kann ein so wichtiger Vorteil sein, dass er allein schon den Einsatz von Microservices rechtfertigt.

Also sind Microservices cloud-native: Sie verstärken die Vorteile einer Cloud-Umgebung, etwa das regelmäßige Deployment oder die bessere Skalierung. Auf der anderen Seite können sie Schwächen wie die Unzuverlässigkeit der Infrastruktur kompensieren. Gerade das regelmäßige Deployment verspricht viele Vorteile, und zwar nicht nur beim Time-to-Market, sondern auch bei der Produktivität und der Zuverlässigkeit.

Neue Module braucht das Land

Microservices sind aber nicht nur wegen der Cloud interessant. Microservices stellen eine neue Art von Modulen dar. Damit stehen Microservices in Konkurrenz zu anderen Modularisierungsansätzen – sei es mit Programmiersprachenfeatures oder beispielsweise Bibliotheken. Eine Entscheidung für Microservices ist eine Entscheidung für eine Art der Modularisierung innerhalb eines Projekts und kann für jedes Projekt anders getroffen werden.

Modularisierung ist der Schlüssel zur Entwicklung komplexer Systeme. Nur mit Modulen ist es möglich, große Systeme so

aufzuteilen, dass ein Mensch zumindest noch einen Teil des Systems verstehen kann. Der entscheidende Schlüssel ist die Entkopplung: Änderungen sollen nur ein Modul betreffen, sonst muss man am Ende doch wieder das gesamte System verstehen.

Klassisch wird Entkopplung bezogen auf die Änderung der Software betrachtet. Da Microservices klare Schnittstellen haben, unterteilen sie das System besser. Es kann ohne weiteres passieren, dass man bei klassischer Modularisierung aus Versehen eine Klasse in einem anderen Modul nutzt und so eine Abhängigkeit erzeugt, die eigentlich nicht erwünscht ist. Microservices haben klare Schnittstellen (z. B. als REST-Schnittstellen) und können so den Zugriff auf die eigene Logik kontrollieren. Auf Klassen aus anderen Microservices kann man gar nicht direkt zugreifen.

Microservices bieten aber auch in anderen Bereichen eine Entkopplung: Sie können wie schon erwähnt getrennt skaliert und deployt werden. Dem Deployment kommt dabei eine besonders wichtige Rolle zu: Durch das getrennte Deployment können Änderungen nicht nur begrenzt in einem Modul programmiert, sondern auch in Produktion gebracht und vom Benutzer verwendet werden. Erst eine ausgelieferte Änderung ist wertvoll. Eigentlich ist also erst eine Entkopplung, die bis zum Deployment reicht, eine wirkliche Entkopplung.

Zudem können pro Microservice andere Technologien genutzt werden. In der Praxis wird man zwar vermutlich bei einem im Wesentlichen einheitlichen Technologie-Stack landen; aber das Update auf eine neue Bibliothek oder eine neue Version einer Programmiersprache kann jeder Microservice getrennt vornehmen. Dies reduziert das Risiko und erlaubt es auch, die Technologie-Updates auf die Microservices zu konzentrieren, die einen besonders großen Vorteil aus den neuen Technologien ziehen können.

Weil jeder Microservice nicht nur unabhängig entwickelt, sondern auch unabhängig in Produktion gebracht werden kann und unabhängige technische Entscheidungen getroffen werden können, sind Microservices gerade in der wichtigsten Disziplin der Module – nämlich Unabhängigkeit – anderen Konzepten deutlich überlegen.

Dies erlaubt es, Entwicklung in großen Organisationen anders zu skalieren: Da Microservices so unabhängig sind, können Teams viel eigenständiger entwickeln. So wird der Kommunikationsbedarf zwischen den Teams reduziert, was eine Skalierung der Entwicklung nicht auf Basis von organisatorischen Maßnahmen, sondern auf der Basis von architektonischen Maßnahmen erlaubt, die mehr Kommunikation und eine engere Abstimmung überflüssig machen. Wenn Teams unabhängig Fachlichkeit entwickeln und deployen können und technisch ebenfalls weitgehend unabhängig sind, dann bleibt wenig Koordination übrig.

Die Skalierung der Organisation ist aber nicht immer der Grund für den Einsatz von Microservices. Einfachere Continuous Delivery, bessere Aufteilung in Module, bessere Skalierung und mehr Zuverlässigkeit können unabhängig von der Teamgröße ausreichende Gründe sein.

Fachlichkeit mit Domain-Driven Design

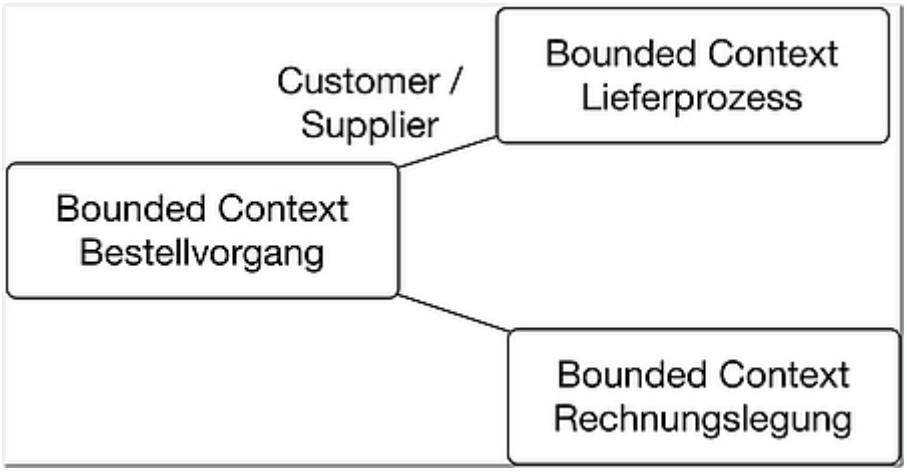

Abb. 3: Bestellvorgang, Lieferprozess und Rechnungslegung

Die stärkere Entkopplung nützt aber wenig, wenn eine fachliche Anforderung dann doch Änderungen an mehreren Microservices verursacht. Idealerweise sollte jede fachliche Änderung zu Änderungen an nur einem Microservice führen. Um zu einer solchen Aufteilung zu gelangen, bietet sich Domain-Driven Design (DDD) an [4][5]. DDD bietet eine umfassende Methodik, um Systeme

fachlich zu entwerfen. Der Teil, der bei Microservices eine besondere Rolle spielt, ist das Strategic Design. Eine wesentliche Idee ist es, ein System in Bounded Contexts aufzuteilen. Das sind Teile des Systems, die jeweils ein eigenständiges Domänenmodell haben.

Beispielsweise könnte es die Bounded Contexts Bestellvorgang, Lieferprozess und Rechnungslegung geben. Jeder dieser Bounded Contexts hat ein Domänenmodell: Der Lieferprozess kann entscheiden, wie und zu welchen Kosten die Waren zum Kunden transportiert werden. Die Rechnungslegung kennt Steuern und Preise. Jeder Bounded Context kennt bestimmte Eigenschaften grundlegender Domänenobjekte: So kennt der Lieferprozess die Lieferadressen und die bevorzugten Lieferservices eines Kunden. Die Rechnungslegung hingegen kennt die Zahlungsmöglichkeiten, die ein Kunde nutzen möchte. So kennt zwar jeder Bounded Context dieselben Domänenobjekte wie Kunde oder Ware, aber in Wirklichkeit sind es unterschiedliche Aspekte dieser Domänenobjekte, so dass nur wenige Redundanzen entstehen. Wenn sich die Regeln für die Rechnungslegung ändern oder ein Lieferdienst seine Schnittstelle ändert, muss nur jeweils ein Bounded Context angepasst werden. Das vereinfacht die Änderung und spricht für eine gute Modularisierung. Wenn der Bounded Context als Microservice implementiert ist, kann er unabhängig deployt werden und technologische Entscheidungen können isoliert getroffen werden. Das Ergebnis ist eine weitgehende Unabhängigkeit in den Dimensionen Deployment, Technologie und Fachlichkeit.

Bounded Contexts sind allerdings Module eines Gesamtsystems. Es ist daher eher selten, dass Anforderungen immer nur einen Bounded Context betreffen. Anforderungen können mehrere Bounded Contexts umfassen. Nehmen wir beispielsweise an, dass es einen neuen Lieferdienst gibt, der im Bestellprozess ausgewählt werden soll und dann im Lieferprozess beachtet werden muss. In diesem Fall müssen also zwei Bounded Contexts geändert werden. Dazu liefert Strategic Design auch Patterns. Beim Cluster/Supplier Pattern kann der Customer (Kunde) den Supplier (Lieferant) bitten, Features zu implementieren. Es wäre logisch, wenn das Feature für einen neuen Lieferdienst im Lieferprozess umgesetzt wird. Wenn das Lieferprozess-Team eine Customer/Supplier-Beziehung zum Bestellprozess-Team hat, dann kann das Lieferprozess-Team vom Bestellprozess Änderungen verlangen, um den neuen Lieferdienst auswählen zu können. So liegt die fachliche Gesamtzuständigkeit beim Lieferprozess, wenngleich das Bestellprozess-Team helfen muss. Es ist keine zentrale Koordination notwendig: Wenn die Customer/Supplier-Beziehung etabliert ist, können die Teams auf Basis dieser Beziehung miteinander interagieren. Eine zentrale Kontrolle ist nicht notwendig.

DDD unterstützt also nicht nur die fachliche Unabhängigkeit, sondern auch die Umsetzung der organisatorischen Unabhängigkeit.

Infrastrukturen: Kubernetes

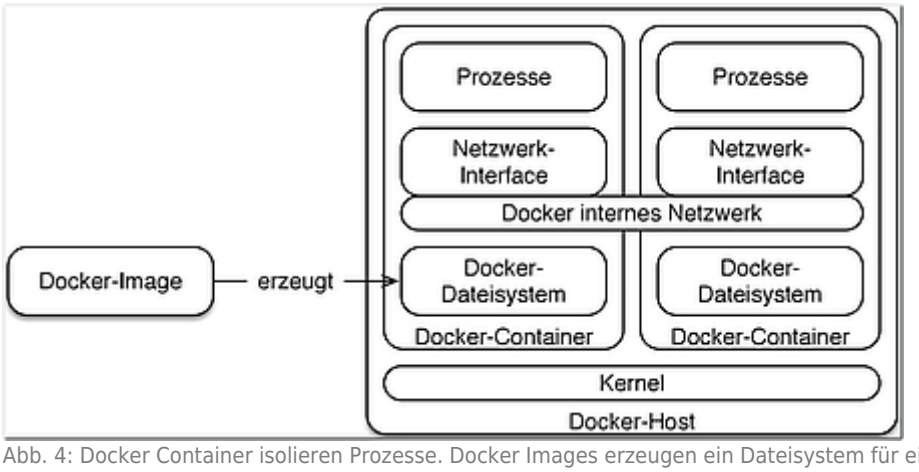

Abb. 4: Docker Container isolieren Prozesse. Docker Images erzeugen ein Dateisystem für einen Docker Container mit einem eigenen Netzwerk-Interface.

Bei all diesen Vorteilen bieten Microservices natürlich auch Herausforderungen. Wenn ein System in eine Vielzahl von Microservices aufgeteilt wird, gibt es viel mehr Systeme, die betrieben werden müssen. Das stellt den Betrieb vor neue Herausforderungen, da der Aufwand für den Betrieb meistens von der Anzahl der Systeme abhängt. Für die Implementierung und den Betrieb von Microservices gibt es zahlreiche Technologien. [4][6]

Virtualisierung hat sich mittlerweile durchgesetzt: Für einen neuen Server erstellt der Betrieb eine neue virtuelle Maschine, statt tatsächlich Hardware zu kaufen. Man könnte jeden Microservice in einer virtuellen Maschine betreiben. Ein wesentlicher Vorteil:

Jeder Microservice hat sein eigenes Betriebssystem. So kann er alle notwendigen Bibliotheken und anderen Infrastruktur-Bestandteile im Dateisystem mitbringen. Außerdem kann der Microservice einen beliebigen Netzwerk-Port nutzen. Wären alle Microservices auf derselben virtuellen Maschine beheimatet, müsste diese virtuelle Maschine so konfiguriert sein, dass sie alle Infrastruktur-Komponenten für alle Microservices zur Verfügung stellen kann. Das kann schwierig zu erreichen und unübersichtlich sein. Außerdem müsste die Portvergabe der Microservices koordiniert werden, so dass kein Port an zwei Microservices vergeben wird.

Virtuelle Maschinen sind jedoch schwergewichtig. Sie benötigen viel RAM und viel Speicher für virtuelle Festplatten. Auch die CPU ist mit der Emulation der virtuellen Maschine beschäftigt.

Docker [8] bietet auch getrennte Dateisysteme und getrennte Netzwerk-Schnittstellen. Aber es ähnelt vielmehr einem Prozess als einer virtuellen Maschine: Jeder Prozess bekommt nur eine eigene virtuelle Netzwerkschnittstelle und ein eigenes Dateisystem. Dabei ist das Dateisystem optimiert, so dass mehrere Docker Container sich gemeinsame Teile des Dateisystems teilen können.

Docker Container sind praktisch genauso effizient wie Prozesse, bieten aber ausreichende Isolation für Microservices. Auf einem Laptop können ohne Probleme Hunderte Docker Container laufen – schließlich können auf einem Laptop auch Hunderte von Prozessen laufen.

Aber für den Einsatz in einem anspruchsvollen Kontext ist es notwendig, dass die Docker Container in einem Cluster laufen, um so Ausfallsicherheit und die Lastverteilung auf mehrere Server zu gewährleisten. Genau das löst Kubernetes [9].

Aber Kubernetes kann mehr:

- Microservices können andere Microservices suchen (Service Discovery). Kubernetes löst dies mit DNS (Domain Name System), also dem System, mit dem auch im Internet Namen wie ewolff.com zu IP-Adressen aufgelöst werden. Service Discovery ermöglicht eine weitere Entkopplung, da Microservices nur die Namen anderer Microservices kennen.
- Kubernetes kann die Last zwischen mehreren Microservice-Instanzen verteilen. Dazu nutzt Kubernetes eine IP-Adresse, hinter der sich mehrere Microservice-Instanzen verbergen. Bei einem Request wird dann eine dieser Instanzen angesprochen. Nur mit Lastverteilung ist die unabhängige Skalierung der Microservices möglich.
- Schließlich kann Kubernetes Requests, die von außen kommen, an einen der Microservices weiterleiten. Dazu dienen Ingress, die umfangreich konfiguriert werden können. So erscheint das System, das intern aus Microservices besteht, nach außen wie ein einziges System.

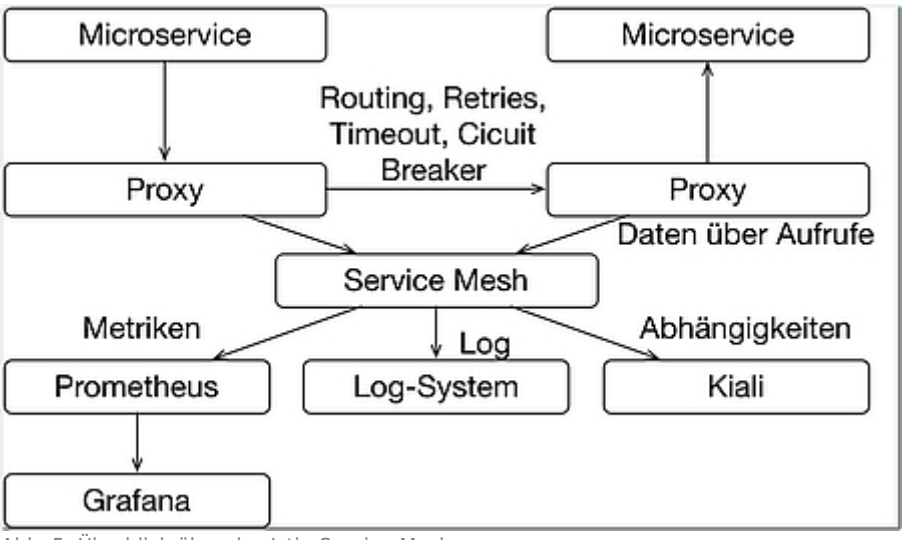

Abb. 5: Überblick über das Istio-Service-Mesh

Auf der Basis von Kubernetes bietet ein Service Mesh wie Istio [10] weitere Features. Es leitet den Netzwerkverkehr zwischen den Microservices durch Proxys. Dadurch kann es einige Features implementieren:

- Grundlegende Metriken wie die Anzahl oder Dauer von Aufrufen, aber auch die Anzahl der Fehler bei den Aufrufen, können gemessen werden. Istio bringt mit Grafana und Prometheus zwei Werkzeuge mit, die solche Metriken auswerten können.
- Wenn Microservices sich gegenseitig aufrufen, können dabei komplexe Aufruf-Graphen entstehen. Istio kann diese Graphen aufzeichnen und mit Jaeger auch visualisieren. Außerdem bietet Istio mit Kiali ein Werkzeug, um die Abhängigkeiten zwischen Microservices zu visualisieren.
- Aufrufe können als Log-Einträge gespeichert werden.
- Schließlich können komplexere Algorithmen für das Routing von Requests zwischen den Microservices umgesetzt werden. Dazu

zählen beispielsweise A/B-Tests, bei denen unterschiedlichen Kundengruppen unterschiedliche Versionen der Software angeboten werden, um zu entscheiden, welche attraktiver ist.
Die Kommunikation zwischen den Microservices kann verschlüsselt werden. Außerdem können die Microservices sich gegenseitig authentifizieren. Das verbessert die Sicherheit.
Durch Retrys, Timeouts und Circuit Breaker kann die Zuverlässigkeit des Microservice-Systems verbessert werden.

Istio und Kubernetes lösen also zahlreiche Probleme, die Microservices mit sich bringen. Dabei ist die Nutzung dieser Infrastruktur für die Microservices transparent. Das macht nicht nur die Nutzung einfacher, sondern es bedeutet auch, dass die Microservices in beliebigen Sprachen mit beliebigen Frameworks implementiert sein können. Also wird die Technologiefreiheit nicht eingeschränkt.

Kubernetes und Istio sind interessant, weil sie zeigen, dass der Aufwand für die Implementierung und den Betrieb von Microservices durch moderne Lösungen wesentlich geringer ist, als dies noch vor nicht allzu langer Zeit der Fall war. Außerdem ist Kubernetes aus einem anderen Grund interessant: Alle großen Cloud-Anbieter wie Microsoft, Amazon und Google bieten schlüsselfertige Kubernetes-Cluster an. Damit gibt es eine gemeinsame Basis-Technologie, auf die sich alle Cloud-Anbieter geeinigt haben. Allerdings ist der Hauptvorteil der Cloud-Lösungen gar nicht unbedingt die Infrastruktur. Clouds bieten zum Beispiel Datenbanken an, die mit einem Mausklick zur Verfügung stehen. Auch Features wie Backup oder Skalierung sind nur wenige Mausklicks entfernt. Es ist nicht ungewöhnlich, dass die Bereitstellung einer Datenbank in einer klassischen IT-Organisation Wochen oder Monate dauert; Backups auf Knopfdruck sind dann oft undenkbar.

Mit Kubernetes gibt es eine Infrastruktur, auf der solche Angebote aufgebaut werden können. Schließlich geht es am Ende „nur“ um die Provisionierung von Software und Storage. Genau das bietet Kubernetes an. Der Vorteil: Wenn die Lösung auf Kubernetes aufsetzt, kann sie im eigenen Rechenzentrum, aber auch in einer beliebigen Cloud genutzt werden. Solche Lösungen entstehen gerade, so dass Kubernetes sich anschickt, nicht nur die Microservices-Welt, sondern auch die Cloud-Welt zu revolutionieren.

Fazit

Microservices erlauben es, das Continuous-Delivery-Versprechen der Cloud tatsächlich umzusetzen. Dies führt nicht nur zu besserem Time-to-Market, sondern auch zu besserer Zuverlässigkeit der Software und Produktivität der Entwickler. Dazu kommen weitere Vorteile bezüglich Skalierbarkeit und Sicherheit. So erlauben Microservices eine sehr starke Entkopplung und eigenständigere Teams. Zur auch fachlichen Umsetzung dieser technischen Unabhängigkeit dient Domain-Driven Design. Bei den Basistechnologien vereinfachen Istio und Kubernetes nicht nur Microservices, sondern bieten darüber hinaus eine leistungsfähige Abstraktion über die verschiedenen Cloud-Angebote.

Microservices und die damit in Verbindung stehenden Technologien sind also nicht nur ein Hype. Sie haben sehr viele Vorteile und helfen, grundlegende Probleme wie Produktivität oder Modularisierung anders anzugehen. Ein genauer Blick lohnt sich also auf jeden Fall.

Quellen

[1] Eberhard Wolff: Continuous Delivery: Der pragmatische Einstieg, 2. Auflage, dpunkt, 2016, ISBN 978-3864903717

[2] DORA DevOps-Studie https://cloudplatformonline.com/2018-state-of-devops.html

[3] Eberhard Wolff: Microservices: Grundlagen flexibler Softwarearchitekturen, 2. Auflage, dpunkt, 2018, ISBN 978-3864905551

[4] Eberhard Wolff: Das Microservices-Praxisbuch: Grundlagen, Konzepte und Rezepte, dpunkt, 2018, ISBN 978-3864905261

[4] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2004, ISBN 978-0321125217

[5] Eric Evans: Domain-Driven Design Referenz, kostenlos unter http://ddd-referenz.de

[6] Eberhard Wolff: Microservices Rezepte - Technologien im Überblick, kostenlos unter https://microservices-praxisbuch.de/rezepte.html

[7] Eberhard Wolff: Microservices - Ein Überblick, kostenlos unter https://microservices-buch.de/ueberblick.html

[8]https://www.docker.com/

[9]kubernetes.io

[10]https://istio.io/

Eberhard Wolff

arbeitet als Fellow bei INNOQ und berät in dieser Funktion Kunden in Bezug auf Architekturen und Technologien. Sein technologischer Schwerpunkt liegt auf modernen Architektur-Ansätzen – Cloud, Continuous Delivery, DevOps oder Microservices spielen oft eine Rolle. Er ist Autor von über hundert Artikeln und Büchern u. a. zu Microservices, Microservice-Technologien und Continuous Delivery.

E-Mail: eberhard.wolff(at)innoq.com

Bildnachweise:

Eberhard Wolff