Java EE meets Microservices ▸ 32

JAVA Mag

4.2016

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

BPM

Kann mehr, als

Der Cloud-Native-Stack: Mesos, Kubernetes und Spring Cloud

Über den Wolken

Cloud-Größen wie Google, Twitter und Netflix haben die Kernbausteine ihrer Infrastruktur quelloffen verfügbar gemacht. Das Resultat aus vielen Jahren Cloud-Erfahrung ist heute frei zugänglich, jeder kann seine eigenen Cloud-nativen Anwendungen entwickeln – Anwendungen, die in der Cloud zuverlässig laufen und fast beliebig skalieren. Die einzelnen Open-Source-Bausteine wachsen zu einem großen Ganzen zusammen, dem Cloud-Native-Stack.

von Dr. Josef Adersberger, Mario-Leander Reimer und Andreas Zitzelsberger

Mit dem Cloud-Native-Stack kann eine Cloud-native Anwendung als Microservices-System entwickelt und betrieben werden. Für hochskalierbare Cloud-Anwendungen ist dieser Stack heute Pflicht, aber letztlich ist jede verteilte Anwendung ein Kandidat für die neue Technologie. In diesem Artikel beschreiben wir die Anatomie des Cloud-Native-Stacks, nennen Vor- und Nachteile und skizzieren die wichtigsten Konzepte. Am Ende werden Sie entscheiden, ob auch Ihre Anwendung Cloud-native werden soll.

2015 hat sich mit der Cloud Native Computing Foundation [1] ein Gremium rund um Google auf den Schultern der Linux Foundation gebildet, das einen Stack für Cloud-native Anwendungen entwickeln und populär machen will. Sie definiert drei wesentliche Eigenschaften von Cloud-nativen Anwendungen: Sie sind aus Microservices aufgebaut und damit ein verteiltes System aus Microservices. Die Microservices werden in Containern paketiert sowie verteilt, und die Container werden auf

Artikelserie

Teil 1: Der Cloud-Native-Stack
Teil 2: Cloud-native Anwendungen mit Spring Cloud bauen
Teil 3: Clusterorchestrierung mit Kubernetes
Teil 4: Mesos: Das Betriebssystem der Cloud

den Knoten der Cloud dynamisch zur Ausführung gebracht.

Dies ist nichts anderes als die konsequente Anwendung des Prinzips der Komponentenorientierung bis in den Betrieb: Microservices sind Komponenten mit HTTP-Schnittstellen. Packt man sie in einen Container, so wird die Komponente auch eine eigenständige Einheit im Betrieb:

- eine eigenständige Releaseeinheit
- eine eigenständige Deployment-Einheit
- eine isolierte Laufzeiteinheit
- eine eigenständige Skalierungseinheit

DevOps wird erst durch diese bis in den Betrieb getragene Komponentenorientierung möglich. Die Kluft zwischen Softwarearchitektur und Betriebsarchitektur verschwindet; beide Welten verwenden denselben Komponentenbegriff. Dies ist ein wesentlicher Grund für die Popularität Cloud-nativer Anwendungen. Doch wie baut man so etwas? Mit einem Cloud-Native-Stack!

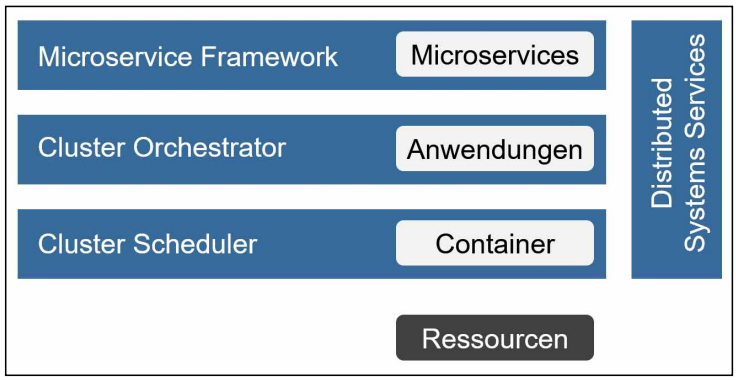

Abb. 1: Die Anatomie eines Cloud-Native-Stacks

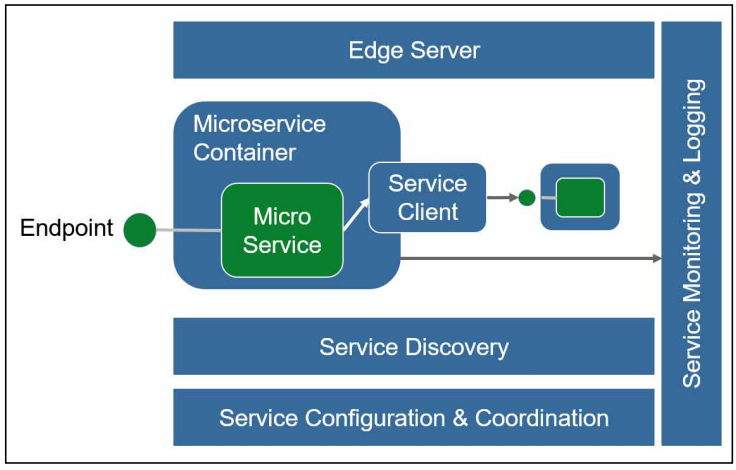

Abb. 2: Technische Infrastruktur einer Cloud-nativen Anwendung

Die Anatomie eines Cloud-Native-Stacks

Ein Cloud-Native-Stack ist die Basis von Cloud-nativen Anwendungen. Auf diesem Stack werden Cloud-native Anwendungen entwickelt und betrieben. Es sind mehrere konkrete Cloud-Native-Stacks verfügbar, jedoch lässt sich bei allen eine gemeinsame Anatomie erkennen: Sie bestehen aus drei Schichten sowie Diensten für verteilte Anwendungen, die über alle Schichten hinweg genutzt werden (**Abb. 1**).

Der Cluster Scheduler hat als erste Schicht die Aufgabe, die einzelnen Container auf Cloud-Knoten zuverlässig und ressourcenschonend auszuführen. Er allokiert Ressourcen und steuert die Ausführung von Containern auf diesen Ressourcen. Ressourcen können sein: virtuelle Maschinen auf einem Rechner, öffentliche oder private Infrastructure-as-a-Service-Clouds oder klassische Serverracks. Der Cluster Scheduler ermittelt die notwendigen Ressourcen über einen geeigneten Scheduling-Algorithmus, allokiert sie für einen bestimmten Zeitraum und führt den Container aus. Re-Scheduling erfolgt bei Bedarf, z. B. beim Ausfall einer Ressource oder beim Freiwerden von alternativen, passenderen Ressourcen. Dann wird die Ausführung eines Containers auf eine andere Ressource verlagert.

Der Cluster Orchestrator als zweite Schicht steuert und überwacht die Ausführung aller Container einer Anwendung. Er verwaltet ganze Anwendungen, während der Cluster Scheduler nur einzelne Container kennt. Die Orchestrierung von Cloud-nativen Anwendungen hat den Anspruch, alle Betriebsprozeduren einer Anwendung zu automatisieren: von Roll-out, Upgrade und der Skalierung bis hin zur Kompensation von Fehlersituationen – getreu dem Motto: „How would you design your infrastructure if you couldn't login? Ever." (Kelsey Hightower, Developer Advocate Google Coud Platform).

Der Cluster Orchestrator beauftragt den Cluster Scheduler mit der Ausführung der einzelnen Container einer Anwendung. Dies ist eine kontinuierliche Aktivität, denn der Orchestrierer muss ständig reagieren, z. B. auf Fehler, Skalierungsbedarf oder Roll-outs.

Die dritte Schicht ist das Microservices-Framework. Es stellt die technische Infrastruktur bereit, auf der Microservices implementiert und ausgeführt werden. Eine Cloud-native Anwendung ist ein verteiltes System aus Microservices. Die Forderungen lauten: Das System muss erstens elastisch und horizontal skalieren, also je nach Bedarf in beliebiger Parallelität laufen, und zweitens dabei maximal fehlertolerant sein, denn in der Cloud gilt: „Everything fails all the time" (Werner Vogels, CTO Amazon.com).

Ein Microservices-Framework bietet eine technische Infrastruktur, die auf Skalierbarkeit und Fehlertoleranz ausgelegt ist (**Abb. 2**). Sie besteht aus folgenden Komponenten, die allesamt in der Regel als Microservices implementiert sind:

- Der Microservices-Container ist ein eingebetteter Webserver, auf dessen Basis ein Microservice implementiert wird, zusammen mit den notwendigen

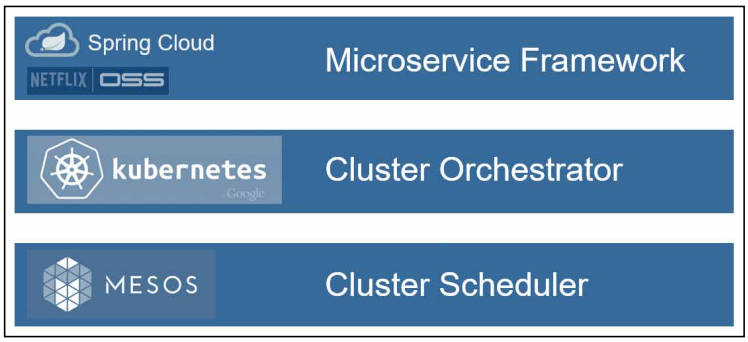

Abb. 3: Der Cloud-Native-Stack: Mesos, Kubernetes, Spring Cloud

	JEE	Cloud Native
Applikation	WAR	Microservices
App-Framework	JEE, Spring	Spring Cloud
App-Server	WildFly, Tomcat, …	Kubernetes
Betriebssystem	Linux	Mesos

Abb. 4: JEE vs. Cloud-native

Programmierschnittstellen wie einem Serviceclient zum Aufruf anderer Microservices mit automatischen Look-ups, systematischer Fehlerbehandlung (*Circuit Breaker*) und Load Balancing. Der Container exportiert automatisch Container- und Microservices-spezifische Metriken und Logeinträge.

- Der Edge-Server ist das Eintrittstor aus dem Internet in eine Cloud-native Anwendung. Neben den klassischen Sicherheitsprüfungen (Authentifizierung, Autorisierung, Request-Validierung) übernimmt er das Routing von Anfragen zum passenden Microservice, eine eventuell notwendige Limitierung der Anfragemenge (*Rate Limiting*) und protokolliert Anzahl, Performance und Zuverlässigkeit von Anfragen für das zentrale Monitoring.
- Die Service Discovery ist ein Dienst, an dem Microservices ihre Endpunkte unter einem symbolischen Namen registrieren. Somit können Edge-Server oder Serviceclients den passenden Endpunkt per Service Discovery über einen symbolischen Namen ermitteln. Horizontale Skalierung wird möglich, indem ein symbolischer Name auf mehrere, möglicherweise sehr viele Endpunkte zeigt.
- „Service Configuration & Coordination“ ist das Herzstück vieler verteilter Anwendungen. Dieser Dienst verwaltet Microservices-übergreifende Zustände wie Konfigurationsparameter der gesamten Anwendung. Er übernimmt zentrale Koordinationsaufgaben wie das Sperren und Abstimmungen. Er basiert auf einem Protokoll, mit dessen Hilfe ein verteilter Konsens über Zustand und Ergebnis von Koordinationsaufgaben hergestellt werden kann, auch wenn einzelne Microservices nicht erreichbar sind.
- „Service Monitoring & Logging“ ist die zentrale Sammelstelle für Logs und Metriken aller Microservices.

Nicht immer bringen die Komponenten des Microservices-Frameworks ihre Funktionalität selbst mit. Manchmal werden lediglich zentrale Dienste im Cloud-Native-Stack angebunden (Distributed Systems Services). Denn nicht nur das Microservices-System, auch der Cluster Scheduler und Cluster Orchestrator sind verteilte, Cloud-native Anwendungen und benötigen entsprechende Infrastruktur – insbesondere Service Configuration & Coordination.

Damit ist die Anatomie eines Cloud-Native-Stacks beschrieben. Doch welche konkreten Stacks sind aktuell einen Blick wert?

Verfügbare Cloud-Native-Stacks

Die momentanen Platzhirsche im Bereich Cloud-Native-Stacks sind der MEKUNS-Stack und der Docker-Stack. Eine Übersicht weiterer Alternativen finden Sie unter [2]. Der MEKUNS-Stack besteht aus den Open-Source-Projekten Mesos, Kubernetes und Netflix OSS mit Spring Cloud (**Abb. 3**):

- Mesos [3] als Cluster Scheduler, der ein dynamisches Scheduling von Docker-Containern erlaubt.
- Kubernetes [4] als Cluster Orchestrator, der Microservices-Anwendungen auf ein Mesos-Cluster bringt und alle dafür notwendigen Betriebsautomatismen bietet. Zwar enthält Kubernetes dafür einen eigenen Cluster Scheduler, aber Mesos ist leistungsfähiger, reifer und besser zu skalieren.
- Spring Cloud [5] mit Netflix OSS [6] als Microservices-Framework mit den dazugehörigen Querschnittsservices für die Programmierung von Cloud-nativen Anwendungen.

Der Docker-Stack besteht aus Docker Swarm als Cluster Scheduler und Docker Compose als Cluster Orchestrator. Beim Microservices-Framework führt auch hier kein Weg an Spring Cloud vorbei. Der Docker-Stack ist einfacher einzusetzen und aufzubauen als der MEKUNS-Stack, da er aus einer Hand stammt. Er wird aus diesem Grund gerne zur Entwicklung von Cloud-nativen Anwendungen genutzt.

In Produktion existiert aktuell unserer Meinung nach aber keine Alternative zum MEKUNS-Stack. Sowohl Mesos als auch Kubernetes haben sich bereits mehrfach und über längere Zeit in Produktion auf Clustern von tausenden Rechnern bewährt, was bei Docker Swarm und Compose nicht der Fall ist. So werden Apple Siri und Twitter vollständig auf Mesos betrieben, und bei Google basiert die Container-Engine der Google Cloud auf Kubernetes.

Baut man nun alle Anwendungen mit einem Cloud-Native-Stack, oder macht auch das aus der Prä-Cloud-Ära stammende JEE-Stack noch Sinn?

Cloud-native vs. JEE

Der wesentliche Unterschied zwischen den beiden Stacks ist die Art und Weise, wie horizontale Skalierung adressiert wird: JEE-Anwendungen sind Ausführungsmonolithen. Das Gesamtsystem läuft in einem Prozess. Bei horizontaler Skalierung wird der App-Server geklont, die verschiedenen Instanzen müssen ihren Zustand im Hauptspeicher oder über die Datenbank synchronisieren. Cloud-native Anwendungen sind in Betriebskomponenten geschnitten. Jede erfüllt ihre spezielle Aufgabe, hält einen Teil des Zustands und besitzt ein eigenes Rezept, wie sie skalieren kann. **Abbildung 4** zeigt einen typischen JEE-Stack im Vergleich zum MEKUNS-Cloud-Native-Stack.

Der JEE-Stack kennt nur einzelne Knoten: Betriebssystem, App-Server, App-Framework und die Anwendung selbst als WAR-Paket zum Deployment auf dem App-Server. Solche Knoten können durchaus auch horizontal skalieren, also als viele Klone parallel zueinander laufen. Die dafür notwendige Infrastruktur wie Load Balancing oder Configuration und Coordination sind jedoch keine First Class Citizens im Stack. Oder was denken Sie über JNDI und Session-Synchronisation? Es gibt aber auch Open-Source-Bausteine, mit denen man dieses Manko von Plain JEE etwas kompensieren kann.

Der Cloud-Native-Stack ist von vornherein auf viele Knoten ausgelegt: Der Cluster Scheduler übernimmt die Rolle eines Betriebssystems im Cluster, der Cluster Orchestrator spielt die Rolle des App-Servers im Cluster, das App-Framework bietet Querschnittsdienste für verteilte Anwendungen und die Anwendung selbst, bestehend aus autarken Microservices. Natürlich ist auch eine Kombination von JEE und Cloud-native denkbar: Ein JEE-Server ist ein möglicher Container für einen Microservice. Das JEE-API bietet aber nicht die vollständig notwendige Infrastruktur drumherum. In diesem Fall können dann z. B. die Netflix-OSS-Bausteine helfen, die auch ohne den Komfort von Spring Cloud genutzt wer-

Vorteile	Nachteile
• Praktisch unbegrenzte horizontale Skalierbarkeit für große Anfrage- und Datenmengen. • Hohe Fehlertoleranz, da alle Bausteine Diagnoseschnittstellen besitzen und viele Mechanismen zur Fehlerkompensierung mitbringen. Twitter konnte damit z. B. ihre berühmten Fail-Whale-Ausfälle bei Hochlast eliminieren. • Microservices sind First Class Citizens, der Cloud-Native-Stack ist primär auf Microservices ausgelegt. Dies ist ein Vorteil, wenn schon entschieden ist, eine Anwendung in Form von Microservices zu bauen. • Gute Betreibbarkeit, da viele Standard-betriebsprozeduren automatisiert sind und per API angestoßen werden können. Der Cloud-Native-Stack ist die ideale Basis für DevOps. • Geringe Betriebskosten, weil das dynamische Scheduling den Ressourcenverbrauch minimiert. Die Einsparungen betragen 10 bis 40 Prozent. Der hohe Automatisierungsgrad reduziert den Arbeitsaufwand im Betrieb. • Gute Isolation von Fehlern, weil jeder Microservice eine eigenständige Laufzeiteinheit ist und damit auch abstürzen kann, ohne andere Services mitzureißen. • Gute Entwickelbarkeit dank durchgängiger Komponentenorientierung bis in den Betrieb. • Kurze Change-to-Production-Zeiten, da Updates einzelner Microservices isoliert ausgerollt werden können. Ferner sind alle dazugehörigen Betriebsprozeduren wie Canary-Updates und Rollbacks vollständig automatisiert verfügbar, und der eigentliche Deployment Roundtrip ist durch die handlichen und schnell startenden Docker-Container kurz. Das ist Continuous Delivery im engen Sinn des Wortes.	Cloud-native Anwendungen sind komplex in Entwicklung und Betrieb. Das liegt vor allem an der hochgradigen Verteilung. Es gibt aber Ansätze, um gefürchtete Komplexitätsnester zu reduzieren: • Ein zentrales Paket- und Abhängigkeitsmanagement kann die hohe Betriebskomplexität durch einen hohen Automatisierungsgrad kompensieren (z. B. per Docker Registry und Helm [7]). • Verteilte Trace-Analysen (z. B. mit Zipkin [8]) und automatische Anomaliedetektion (z. B. mit EGADS [9]) helfen bei der schwierigen Diagnose von Microservices-übergreifenden Fehlern. • API-Spezifikationen (z. B. Swagger oder RAML) und Teststellung zur automatischen Prüfung von API-Kontrakten (z. B. Pact [10]) unterstützen bei der komplexen Integration von Microservices, vor allem bei unterschiedlichen Versionsständen. • Eigenständiges Auto-Scaling pro Microservice unterstützt bei der Skalierung. Da jeder einzelne Microservice eine eigene Skalierungseinheit ist, kann es z. B. bei schlechten Antwortzeiten schwer sein, den schuldigen Microservice zu ermitteln. Bei JEE wurde einfach der gesamte App-Server geklont. Hier helfen Regeln zur automatischen Skalierung pro Microservice. Neben der erhöhten Komplexität hat der Cloud-Native-Stack noch weitere Nachteile: • Es entsteht ein höherer Overhead durch die Remote-Kommunikation zwischen den Microservices (Verteilungsschuld) und durch den Speicherverbrauch jeder Ausführungsumgebung (z. B. JVM) pro Microservice. Schneidet man seine Microservices klug und auf der richtigen Granularitätsebene, so spielt der Overhead jedoch nur eine geringe Rolle. • Der Cloud-Native-Stack ist noch unreif. Einige der Komponenten haben noch nicht den Versionsstand 1.0 erreicht. Außerhalb der Cloud-Größen (wie Google oder Netflix) liegen noch wenige Erfahrungen vor. Man sollte wissen, was man tut; die neue Technologie ist nichts für Anfänger. Darum sollte man schon früh, also jetzt, Erfahrungen mit dem Cloud-Native-Stack sammeln. • Der Umgang mit umfangreichen Zuständen, die typischerweise in der Datenbank liegen, ist nicht immer klar. Das liegt sicher daran, dass Datenbanken meist außerhalb eines Cloud-Native-Stacks betrieben werden. Für Datenpersistenz beim Re-Scheduling oder zur Datenmigration bei Updates gibt es erste Tools, aber noch keine überzeugenden Konzepte. Hier ist jedoch mit einem deutlichen technologischen Fortschritt in 2016 zu rechnen.

Tabelle 1: Vor- und Nachteile eines Cloud-Native-Stacks

Bei großen skalierbaren, fehlertoleranten Systemen ist der Cloud-Native-Stack schon heute unvermeidbar. Aber eigentlich ist jede verteilte Anwendung ein Kandidat für den Cloud-Native-Stack.

den können. Der Cloud-Native-Stack macht JEE also nicht obsolet, reduziert aber den JEE-App-Server auf einen Microservices-Container.

Bei Cloud-nativen Anwendungen verschwimmen Systemgrenzen. Eine Menge von Microservices kann alles sein, von kleinen Anwendungen bis zu riesigen Anwendungslandschaften, wie bei Netflix oder Walmart. Insofern ist die Cloud-native-Architektur eine Weiterentwicklung der in die Jahre gekommenen und teilweise auch in Ungnade gefallenen serviceorientierten Architekturen.

Die Entscheidung, ob man eine Anwendung eher auf einem klassischen Stack umsetzt oder doch auf einem Cloud-Native-Stack ist ein Trade-off aus den Vor- und Nachteilen des Cloud-Native-Stacks (Tabelle 1).

Fazit

Mit dem Cloud-Native-Stack steht ein neuartiger Stack für Entwicklung und Betrieb von hoch skalierbaren und fehlertoleranten Anwendungen zur Verfügung. Wir sind sicher, dass sich diese Technologie durchsetzen wird. Bei großen skalierbaren, fehlertoleranten Systemen ist der Cloud-Native-Stack schon heute unvermeidbar. Aber eigentlich ist jede verteilte Anwendung ein Kandidat für den Cloud-Native-Stack. Ob man schon jetzt eine solche Anwendung auf diesem Stack entwickelt oder eine bestehende Anwendung darauf portiert, hängt vom Mut ab und davon, wie man das hier dargestellte Trade-off einschätzt. Auf jeden Fall ist jetzt die Zeit, um zumindest unter Laborbedingungen schon Erfahrungen zu sammeln. Eine sehr reife Variante ist der MEKUNS-Stack aus Mesos, Kubernetes, Netflix OSS und Spring Cloud.

Der Cloud-Native-Stack wird weiter reifen und sich weiter entwickeln. Insbesondere wird er sich in Richtung Platform as a Service weiterentwickeln, das die Bedürfnisse der Entwickler weiterführend adressiert:

- Einfaches Deployment über Git
- Wiederverwendung von Infrastruktur über ein integriertes Package-Management
- Komfortfunktionen wie die automatische Erkennung und Bereitstellung der notwendigen Infrastruktur

Erste Projekte existieren bereits, die den Cloud-Native-Stack in diese Richtung erweitern, wie Deis [11], PaaSTA [12] oder Mantl [13].

Wir beleuchten die einzelnen Bausteine des MEKUNS-Stacks in den weiteren Artikeln dieser Serie. Wer schon jetzt tiefer eintauchen will, für den stehen Vorlesungsunterlagen der Autoren zur Verfügung [14].

Dr. Josef Adersberger ist technischer Geschäftsführer der QAware GmbH, einem IT-Projekthaus mit Schwerpunkt auf Cloud-native Anwendungen und Softwaresanierung. Er hält seit mehr als zehn Jahren Vorlesungen und publiziert zu Themen des Software-Engineerings, aktuell insbesondere zu Cloud Computing.

Andreas Zitzelsberger ist Cheftechnologe bei der QAware. Zu seinen Schwerpunkten gehören leichtgewichtige Enterprise-Integration, moderne Big-Data-Architekturen sowie die Diagnose und Stabilisierung komplexer Systeme.

Mario-Leander Reimer ist Cheftechnologe bei der QAware. Er ist Spezialist für den Entwurf und die Umsetzung von komplexen System- und Softwarearchitekturen auf Basis von Open-Source-Technologien. Als Mitglied im Java Community Process (JCP) ist sein Ziel, die Java-Plattform weiter zu verbessern und praxistaugliche Spezifikationen zu entwickeln.

Links & Literatur

[1] Cloud Native Computing Foundation: http://bit.ly/1TScflZ

[2] Cloud-Native-Stack-Technologieradar: http://bit.ly/1WRDWUI

[3] Apache Mesos: http://bit.ly/1LgfcDC

[4] Kubernetes: http://bit.ly/1lWDho6

[5] Spring Cloud: http://bit.ly/1QKy271

[6] Netflix OSS: http://bit.ly/1TafDiE

[7] Helm by DEIS: http://bit.ly/1nCFwhi

[8] Zipkin: http://bit.ly/1nCFyFQ

[9] EGADS: http://bit.ly/1PEEzuI

[10] Pact: http://bit.ly/1WREk5h

[11] Deis: http://bit.ly/1Uv2jpy

[12] PaaSTA: http://bit.ly/1Kf2g0A

[13] Mantl: http://bit.ly/1VvcmLg

[14] Vorlesung Cloud Computing: http://bit.ly/1TrnwIP