Sonderdruck aus OBJEKTspektrum

Cloud-Development

Cloud-native – Das Cloud-Potenzial voll ausschöpfen

Der Artikel zeigt auf, welche besonderen Herausforderungen bei der Umsetzung einer Cloud-native Anwendung für die AWS (Amazon Web Services)-Cloud zu bewältigen sind. Anhand einer fiktiven Anwendung werden wichtige Aspekte der Entwicklung einer Cloud-native Anwendung von den Anforderungen bis zum Deployment beschrieben. Architekturentscheidungen werden verifiziert und Lösungen für mögliche Fallstricke behandelt. Im Abschluss wird betrachtet, welche besondere Rolle DevOps-Methodiken bei der Entwicklung einer Cloud-native Anwendung spielen.

Cloud-native beschreibt einen Ansatz, bei dem das Design und die Implementierung einer Anwendung konform mit der Cloud-Architektur und für diese optimiert erfolgt. Das Ziel dabei ist, dass solche Anwendungen das Potenzial von Cloud-Architekturen nutzen und deren Vorteile voll ausschöpfen, wie zum Beispiel:

- hohe Verfügbarkeit,
- flexible und nahezu unbegrenzte Skalierbarkeit,
- nutzungsbezogene Abrechnungsmodelle,
- Vereinfachung des Betriebs.

Ein wesentliches Merkmal einer *CNA* (Cloud-native Anwendung) ist, dass diese Anwendung mit der Infrastruktur sozusagen verschmilzt, indem Cloud-Services als Implementierungsbausteine genutzt werden. Mittlerweile steht eine fast schon unüberschaubare Anzahl von Cloud-Services unterschiedlichster Art zur Verfügung, die für diesen Zweck genutzt werden können. So bietet Amazon Web Services [AWS] als führender Cloud-Provider über 100 verschiedene Dienste aus unterschiedlichen Bereichen an. Diese reichen von Diensten für die einfache dateibasierte Speicherung über Datenbanken und Analysen bis hin zu Diensten für die Entwicklung von Blockchain- und IoT(Internet of Things)-Anwendungen und sogar zur Steuerung von Satelliten.

Eine fiktive Beispielanwendung

Stellen wir uns einen sehr vereinfachten Wetterdienst vor, der die aktuellen Wetterdaten über eine Webanwendung und eine mobile Anwendung regional bezogen anbietet. Darüber hinaus soll der Benutzer des mobilen Gerätes über spezielle Unwetterwarnungen, für die er sich registrieren kann, informiert werden. Im Wesentlichen gliedert sich die Anwendung in folgende vereinfachte Prozessschritte:

1. Ermittle die Wetterdaten aus einer externen Quelle zeitgesteuert all n Minuten, partitioniere diese Daten nach Regionen und ermittle Unwetterwarnungen für die Regionen.
2. Aktualisiere die partitionierten Daten in einer Datenbank zur Bereitstellung für die Web- und die mobile Anwendung.
3. Versende die Unwetterwarnungen an interessierte Benutzer.
4. Ermögliche die Anzeige der Daten in einer Web- und einer mobilen Anwendung.

Leser, die möglicherweise bereits Erfahrungen mit der Entwicklung einer echten Wetteranwendung sammeln konnten, bitten wir um Nachsicht, wenn wir den Prozess zu sehr vereinfacht haben sollten. Uns erscheint dieser Prozess plausibel, und er eignet sich vorzüglich für unsere Zwecke.

Die Architektur

Zur Umsetzung der Prozessschritte wollen wir Dienste der AWS-Cloud einsetzen, sodass die Entwickler sich auf die Umsetzung der Funktionalität fokussieren können und sich nicht um alle infrastrukturellen Belange kümmern müssen. Ein Überblick über die verwendeten Dienste der AWS-Cloud ist in **Tabelle 1** dargestellt.
In **Abbildung 1** wird ein Überblick über die Laufzeitkomponenten der Architektur und deren Zusammenspiel, bezogen auf die oben skizzierten Prozessschritte, gegeben.

Die Implementierung

Die Implementierung erfolgt in mehreren Prozessschritten.

Cloudwatch-Trigger	Dieser ermöglicht, andere Dienste zeitgesteuert zu starten.
Lambda	Eine Lambda ist ein (serverloser) Funktionsbaustein, der über definierte Ein- und Ausgabeschnittstellen verfügt. Er kann in verschiedenen Sprachen implementiert sein. Instanzen von Lambdas werden von der Cloud verwaltet und dynamisch (lastbezogen) erzeugt. Lambdas werden technisch in einem Container ausgeführt und benötigen daher einige Zeit beim initialen Start. Dieses nennt man Cold-Start-up-Time.
SQS	SQS (Simple Queue Service) ist eine hochverfügbare Nachrichtenwarteschlange, die die Zustellung einer Nachricht mindestens einmal garantiert.
SNS	SNS (Simple Notification Service) ist ein Publish/Subscribe-Nachrichtendienst zur Weiterleitung von Push-Nachrichten. Er ermöglicht die Anbindung unterschiedlicher Subscriber, so z. B. auch mobile Push-Benachrichtigungsservices.
ECS	ECS (Elastic Container Service) ist ein Container-Service. Wir nutzen diesen Dienst, um die Server-Implementierung der REST-Schnittstellen in Form eines Docker-Containers für die Web- und mobile Anwendung bereitzustellen.
API-Gateway	API-Gateway ist ein Dienst, der das Erstellen, Veröffentlichen, Warten, Überwachen und Sichern von APIs ermöglicht. Es unterstützt REST- und Websocket-APIs und dient gleichzeitig zur Absicherung des REST-Servers gegen unautorisierten Zugriff.
S3	S3 ist ein Service zur Speicherung von Daten. Er wird zur Bereitstellung der statischen Webdateien (JavaScript, HTML, CSS, Grafiken usw.) genutzt.

Tabelle 1: Übersicht über die verwendetet Dienste der AWS

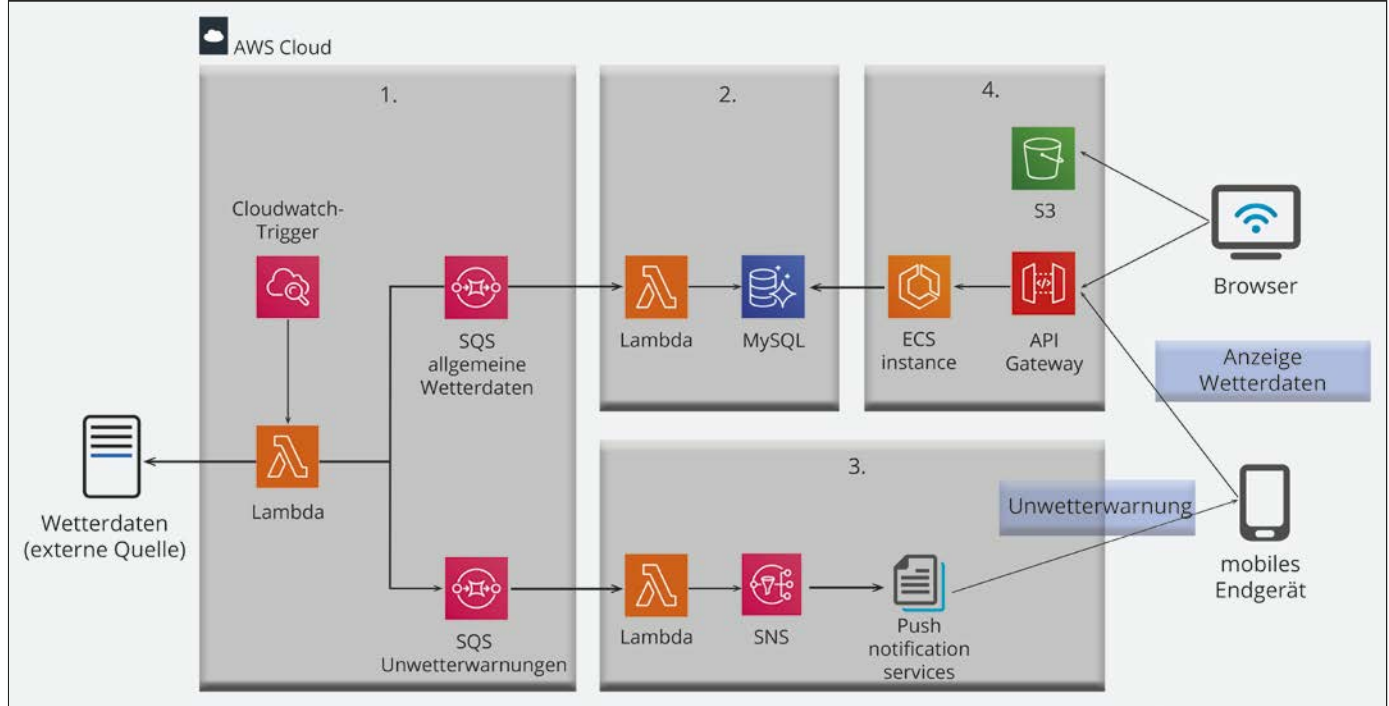

Abb. 1: Überblick über die Laufzeitkomponenten der Architektur

Schritt 1
Alle 30 Minuten startet ein Cloudwatch-Trigger einen Funktionsbaustein als Lambda, der alle Wetterdaten abholt, nach Regionen partitioniert, die Unwetterwarnungen herausfiltert und die Pakete in eine Queue für die allgemeinen Wetterdaten und in eine Queue für Unwetterwarnungen legt.

Schritt 2
Die Lambda zur Verarbeitung der allgemeinen Wetterdaten ist mit der entsprechenden Queue via Konfiguration verbunden und wird automatisch bei eintreffenden Nachrichten in der Queue aufgerufen. Diese Lambda transformiert die Daten und speichert sie in der MySQL-Datenbank.
An dieser Stelle zeigt sich die Flexibilität der Verbindung von SQS und Lambda: Je nach Konfiguration und Bedarf werden dynamisch durch die Cloud *n* Lambda-Instanzen erzeugt, die dann zur Verarbeitung von Nachrichten aus der Queue bereitstehen. Einmal instanziierte Lambdas werden gepoolt und für neue Nachrichten wiederverwendet. Dadurch kann je nach Bedarf ein optimaler Durchsatz (Skalierbarkeit und Performanz) erreicht werden. Da die Kosten-Berechnung abhängig von der Laufzeit einer Lambda (inkl. Instanziierung) ist, werden nur Kosten erzeugt, wenn Nachrichten in der Queue sind und Lambdas aufgerufen werden.

Schritt 3
Die Lambda zur Verarbeitung der Unwetterwarnungen ist ebenfalls mit der entsprechenden Queue verbunden. Diese Lambda bereitet die Daten auf, ermittelt die Anmeldeinformation für den entsprechenden Push-Benachrichtigungsservice und übergibt die Nachricht an den SNS-Dienst.
Die Einrichtung der Push-Benachrichtigungsservices wie auch die Ermittlung der Anmeldeinformationen sind komplexe Prozesse, die an dieser Stelle daher nicht im Detail betrachtet werden sollen. Mehr Informationen dazu finden Sie in der umfangreichen Dokumentation zu den AWS-Diensten [AWS].

Schritt 4
Die statischen Inhalte für die Webanwendung (JavaScript, HTML, CSS, Grafiken usw.) können einfach von dem Datenspeicher S3, organisiert in sogenannten Buckets, bereitgestellt werden. Buckets sind standardmäßig unter einer AWS-URL erreichbar, eigene Domain-Namen können ebenso verwendet werden. In S3 werden die statischen Dateien unserer Webanwendung abgelegt.
Sowohl die Webanwendung als auch die mobile Anwendung benötigen für die dynamischen Inhalte den Zugriff auf REST-Schnittstellen, die von der Backend-Implementierung, die mit Java/Spring Boot entwickelt werden soll, bereitgestellt werden. Der Java-Prozess läuft in einem Docker-Container auf einem ECS-Host. Die REST-Schnittstellen werden über das API-Gateway im Internet zur Verfügung gestellt. Das API-Gateway übernimmt die Autorisierung und führt Zugriffskontrollen für die REST-Services aus.

Grundlegende Design-Entscheidungen
Im Falle unserer Beispielanwendung haben wir folgende Cloud-spezifische Design-Entscheidungen getroffen.

Verwendung von Lambda/SQS
Wir haben uns bei der Datenübernahme (Schritte 1 und 2) für die Verwendung von Lambdas im Zusammenspiel mit SQS entschieden. In der Startphase der Anwendung wollen wir nur große Regionen berücksichtigen und eine Datenaktualität von 30 Minuten erreichen.
In der Zukunft sollen die Regionen feingranularer abgebildet und in einem kleineren Zeitintervall Daten bereitgestellt werden, um so für kleinere Regionen aktuelle Daten vorhalten zu können. Dadurch sind wir in diesem Bereich hinsichtlich Skalierbarkeit und Performanz für die Zukunft gewappnet. Darüber hinaus laufen die Lambda-Funktionen nicht durchgängig und erzeugen geringere Kosten, als wenn wir einen Container permanent durchlaufen lassen würden, der in der Startphase nur alle 30 Minuten etwas zu tun hätte.

Verwendung eines Spring Boot-Microservice in einem Docker-Container
Für den Einsatz eines Spring Boot-Microservice in Java haben wir uns aus folgenden Gründen entschieden:
Wir erwarten über weite Teile des Tages eine relativ gleichmäßige Lastverteilung. Die Anpassungsfähigkeit der Lambdas bei schwankenden Lastprofilen zahlt sich daher in unserem Szenario nicht aus.

Natürlich sind wir alle polyglott. Aber haben wir vielleicht ein gut eingespieltes Team von Java-Entwicklern, die auf Basis des Spring Boot-Stacks die Implementierung der REST-Schnittstellen sehr schnell umsetzen könnten? Allerdings wäre der Einsatz von mächtigen Frameworks wie Spring Boot zur Implementierung als Lambda-Funktionen eher kritisch, da die Cold-Start-up-Time solcher Lambdas sehr hoch sein kann (in Größenordnungen von 10 bis 30 Sekunden), je nachdem, wie groß das Java-Deployment-Artefakt ist. Solche (Warte-) Zeiten bei der Interaktion mit Benutzeranfragen sind für eine optimale Usability nicht akzeptabel. Zwar gibt es technische Möglichkeiten, eine Lambda durch regelmäßige Trigger „heiß" zu halten. Im Falle von Lambdas, die dazu dienen, Benutzeranfragen zu bedienen, helfen derartige Strategien aber leider auch nur begrenzt weiter: Zwar lässt sich damit die Wahrscheinlichkeit, dass ein Benutzer-Request an eine „kalte" Lambda delegiert wird, minimieren, aber nicht gänzlich verhindern, da weder die Benutzer-Interaktion noch der Wiederanlauf von Lambdas planbar sind.
Abgesehen von dem Cold-Start-up-Problem hätten wir alternativ wahrscheinlich eine Reihe von Lambdas für die Implementierung der REST-Schnittstellen umsetzen müssen. Dieses würde zusätzlichen Aufwand für die Konfiguration und das Deployment nach sich ziehen. Tatsächlich muss aus unserer Erfahrung die Implementierung von Funktionalität in Lambdas sehr genau abgewogen werden. Lambdas spielen ihre Stärken in der extremen Skalierbarkeit und damit der gleichmäßigen Performanz und in ihrem lastbezogenem Abrechnungsmodell aus. Nachteilig wirken sich der erhöhte Aufwand für das automatisierte Deployment (mehr dazu im nächsten Abschnitt), ein relativ hoher Testaufwand und die manchmal schwierigere Fehlersuche bei komplexen Prozessketten aus.
Alternativ hätte man die Lambdas nicht in Java, sondern in einer anderen leichtgewichtigeren von Lambda unterstützten Laufzeitumgebung, wie in JavaScript (Node.js) oder Python, implementieren können. Dieses hätte Wartezeiten auf Basis von Cold-Start-ups zwar minimieren, aber nicht gänzlich verhindern können.

Deployment/DevOps-Pipeline
Ein wichtiger Aspekt bei der Entwicklung einer CNA ist die Konfiguration, das Provisionieren und das Deployment von Cloud-Ressourcen. AWS bietet zwar eine komfortable Webanwendung (AWS Management Console) zur manuellen Konfiguration der Dienste. Diese Webanwendung ist aber nur hilfreich beim Ausprobieren und zum stichprobenartigen Monitoring von Diensten. Bereits eine kleine Anwendung, wie unsere Beispielanwendung, nutzt eine beträchtliche Anzahl von Cloud-Ressourcen, die mit einem „manuellen" Deployment nicht wirtschaftlich betrieben werden können.
Verschärfend kommt in der Praxis hinzu, dass ein System in der Regel in verschiedenen Stages betrieben wird (z. B. Test, Abnahme und Betrieb), die separat deployt werden müssen. Hieraus resultiert die Anforderung, dass ein automatisiertes Provisionieren von Cloud-Ressourcen und das automatisierte Deployment der Softwareartefakte für eine CNA ein Muss ist. Basis für den Entwicklungsprozess ist daher eine klassische DevOps-Pipeline (**siehe Abbildung 2**), die in dem Bereich Deployment Cloud spezifisch anzupassen ist [Dav19].
Die Cloud spezifische Anpassung des Deployment-Prozesses ist prinzipiell über einen der folgenden Mechanismen möglich:

- Nutzung von Cloud spezifischen Werkzeugen (z. B. AWS Cloudformation),
- Nutzung eines generischen „Infrastructure as code"-Werkzeugs.

Zum Deployment in die AWS-Cloud bietet sich beispielsweise Cloudformation als Beschreibungssprache und Ausführungsumgebung für die Infrastruktur an. Cloudformation wurde von Amazon speziell für die AWS-Cloud entwickelt, alternative Tools wie zum Beispiel Terraform [TER] sind dagegen unabhängig von einem spezifischen Cloud-Provider. Aus unserer Erfahrung ist dies allerdings eher ein theoretischer Vorteil, da man sich hinsichtlich der Parametrisierung der Laufzeitkomponenten auch in Terraform an spezifische AWS-Module und deren Eigenschaften bindet. Aus unserer Erfahrung dienen die folgenden Kriterien als Entscheidungshilfe für einen der beiden Deployment-Ansätze:

- Falls nur eine Cloud als Zielsystem unterstützt werden soll und es sich um eine einfache Anwendung mit wenig Infrastruktur-Komponenten handelt, ist der Cloud spezifische Ansatz der einfachste.
- Für Anwendungen/Entwicklungen, die in verschiedenen Cloud-Zielsystemen verfügbar sein sollen und die eine Vielzahl von Infrastruktur-Komponenten benötigen, ist der Einsatz von generischen Werkzeugen der optimalere Ansatz – insbesondere da diese Werkzeuge die Modularisierung der Deployment-Skripte unterstützen und so eine vereinfachte Entwicklung/Wartung ermöglichen.

Für unsere relativ übersichtliche Beispielanwendung müssten vier Artefakte gebaut, konfiguriert und deployt, 12 Laufzeitkomponenten und fast ebenso viele Beziehungen dieser Komponenten untereinander konfiguriert werden. Für diesen Anwendungsfall ist AWS Cloudformation für das Deployment völlig ausreichend und bietet sich daher als geeignetes Tool für unsere Beispielanwendung an.
Um die Anwendung auch bereits während des Entwicklungsprozesses testen zu können, ist es notwendig, bereits zu einem frühen Entwicklungszeitpunkt das Deployment der Anwendung zu automatisieren. Es ist daher auch die Aufgabe der Entwickler, die benötigten Deployment-

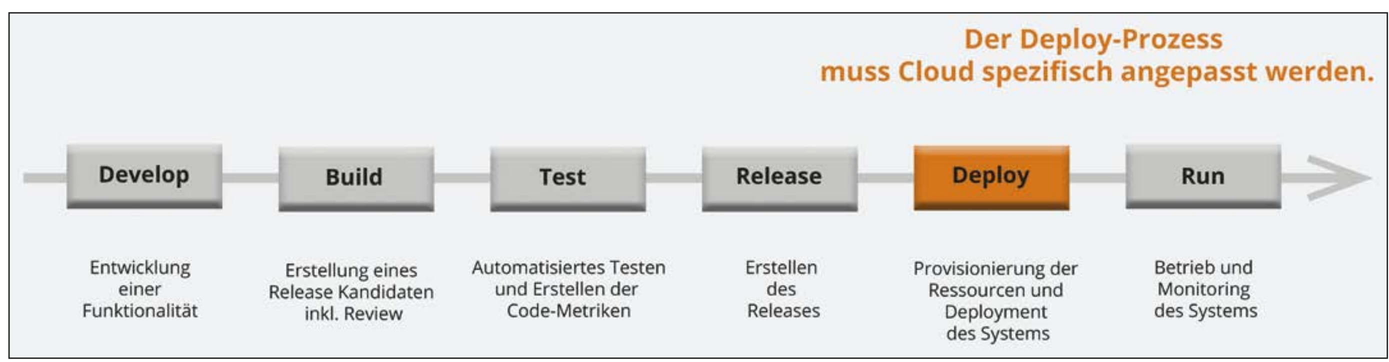

Abb. 2: DevOps-Pipeline

und Konfigurations-Skripte im Sinne von *Infrastructure as Code* zu entwickeln. Dieser Prozess wird dabei idealerweise von Cloud-Infrastruktur-Experten unterstützt.
Indem die Entwickler somit die Verantwortung für das Deployment übernehmen, verwischen die Grenzen zwischen Entwicklung und Betrieb. Dieses führt zu einer Reduktion der notwendigen Dokumentation, die ja bereits in den Skripten abgebildet ist, sowie zur Vermeidung von Missverständnissen und Übergabefehlern zwischen diesen Organisationseinheiten – ganz klassisch im Sinne von DevOps.

Betrieb und Monitoring
Der Betrieb, die Überwachung und die Fehlersuche einer CNA unterscheiden sich erheblich von einer monolithischen Applikation. Insbesondere der Einsatz von Lambdas als serverlose Funktionen resultiert in einer Microservice-Architektur mit einer Vielzahl feingranularer Services, die ohne eine entsprechende Logging-Infrastruktur nicht überwacht werden können. Für die Beispielapplikation könnte dieses Problem mit den folgenden etablierten Mechanismen einer Microservice-Architektur gelöst werden [Ric18]:

- *Distributed Tracing* bietet die Möglichkeit, für Systemaktionen, an denen mehrere Microservices/Funktionen beteiligt sind, eine eindeutige Korrelations-Id („Trace-Id") zu erzeugen, die über die einzelnen Services propagiert und gelogged wird. Hierdurch ist eine Nachvollziehbarkeit/Fehlersuche für Systemaktionen möglich, die nicht innerhalb eines Prozesses abgearbeitet werden können.
- Durch *Logfile Aggregation* werden die Logfiles aus den verschiedenen Microservices zusammengeführt und können anschließend mit Werkzeugen zur Logauswertung [Sri18] (z. B. ELK Stack [ELK], Splunk [SPL]) analysiert werden (**siehe Abbildung 3**).

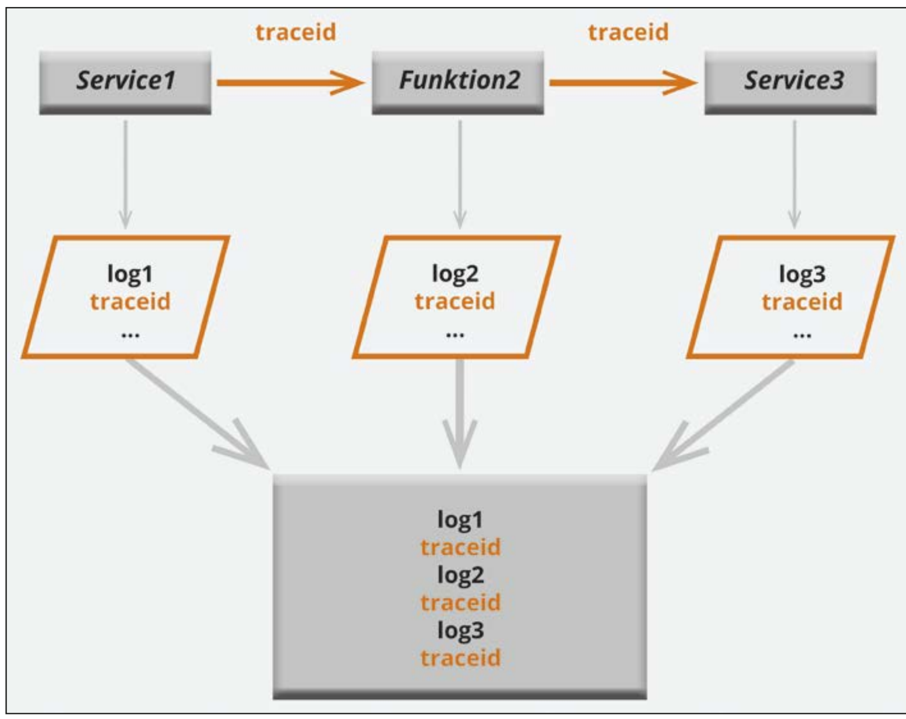

Abb. 3: Distributed Tracing und Log-Aggregation

Fazit

Die Entwicklung von CNAs unterscheidet sich von „klassischen" Entwicklungsprojekten im Wesentlichen hinsichtlich der infrastrukturellen Komplexität und der Komplexität, die aus der Verteilung des Systems in Form von Microservices resultiert.
Als grundlegende Architekturentscheidung einer CNA muss festgelegt werden, für welche Funktionalitäten der Einsatz von serverlosen Funktionen sinnvoll ist oder besser serverbasierte Komponenten verwendet werden:

- Der Einsatz von serverlosen Funktionen [Sba17] bietet sich für Prozesse an, die nicht permanent benötigt werden, und für Lastszenarien, die eine sehr hohe Skalierbarkeit erfordern.
- Der Einsatz einer serverbasierten Lösung ist von Vorteil, wenn Applikationen eine niedrige Latenzzeit erfordern – für diese Anwendungsfälle stellt die Cold-Start-up-Time von serverlosen Funktionen in der Regel ein Problem dar.

Da die Komplexität eines solchen Systems aus dem Zusammenspiel vieler Funktionsbausteine resultiert, sind auch das Deployment und die Fehlersuche komplexer. So ist es für den Erfolg einer CNA ebenso wichtig, bereits vor Entwicklungsstart die passende DevOps-Pipeline sowie die Logging- und Monitoring-Infrastruktur festzulegen. ||

Literatur & Links

[AWS] AWS Documentation, siehe: **https://docs.aws.amazon.com/**

[Dav19] C. Davis, Cloud Native Patterns, Manning Publications, 2019

[ELK] Elasticsearch, siehe: **https://www.elastic.co/**

[Ric18] Ch. Richardson, Microservices Patterns, Manning Publications, 2018

[Sba17] P. Sbarski, Serverless Architectures on AWS: With examples using AWS Lambda, Manning Publications, 2017

[Sri18] C. Sridharan, Distributed Systems Observability, O'Reilly Media, 2018

[SPL] Splunk, siehe: **https://www.splunk.com/**

[TER] Terraform, siehe: **https://www.terraform.io/**

Der Autor

Dipl.-Wirt.-Inform. Jörg Groß
(gross@consist.de)
beschäftigt sich seit fast 25 Jahren mit der Entwicklung und Architektur von Java-Enterprise-Anwendungen. Seit 5 Jahren arbeitet er bei Consist Software Solutions GmbH in Kiel. Sein aktueller Schwerpunkt liegt in der Entwicklung von Cloud-basierten Anwendungen.