OBJEKTspektrum ONLINE Themen special

Advertorial

Erfolgsfaktoren und Risiken bei der Architektur für Microservices und Self-contained Systems

Wenn Cloud-native den Monolithen ablöst:

von Stefan Kühnlein

Immer mehr Unternehmen erkennen die Vorteile von Cloud-basierten Services und sind bereit, ihre bestehenden Anwendungen in die Cloud zu migrieren. Mithilfe eines Lift-and-Shift-Ansatzes können monolithische Anwendungen ja auch relativ einfach in der Cloud betrieben werden. Allerdings nutzen diese Anwendungen nur einen Teil der Cloud-Vorteile, bei Flexibilität, Skalierbarkeit und Elastizität gibt es zum Beispiel erhebliche Bremsen. Cloud-native-Anwendungen können hingegen langfristig erfolgreich in der Cloud betrieben werden und alle Vorteile ausnutzen.

Cloud-native-Anwendungen werden speziell für eine Cloud-Architektur entwickelt und nutzen deren Vorteile durch den Einsatz von entsprechenden Frameworks deshalb vollständig aus. Somit unterscheiden sich Cloud-native-Anwendungen sehr stark von traditionellen Anwendungen, die in einer Schichtenarchitektur aufgebaut sind.

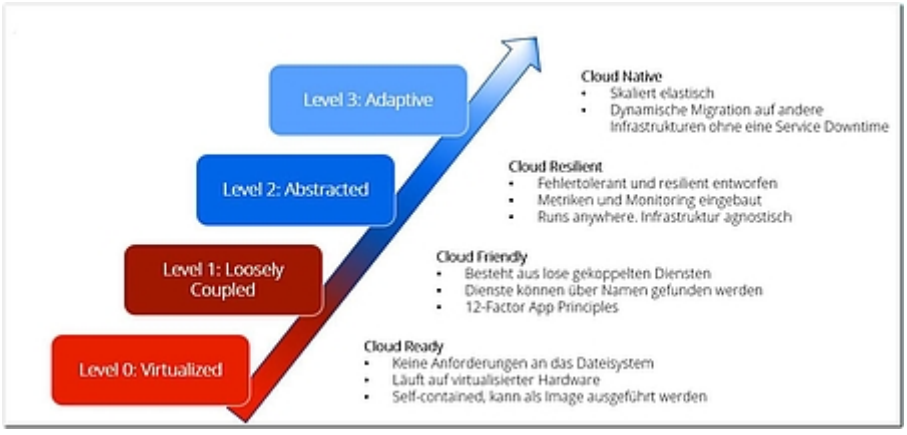

Abb. 1: Reifegrade von Cloud-Lösungen (Quelle [1])

Die Open Data Center Alliance hat für die Bewertung der Reife einer Cloud-basierten Anwendung die folgenden Reifegrade definiert (**siehe Abbildung 1**). Die Alliance ist davon überzeugt, dass Level 3 nur mithilfe von Microservices-Architekturen oder containerisierten Anwendungen zu erreichen ist.

Das sieht die Organisation Cloud Native Computing Foundation (CNCF) ähnlich und empfiehlt Unternehmen Open-Source-Software wie Docker und Kubernetes, wenn es darum geht, Applikationen in die Cloud zu bringen. In der CNCF engagieren sich unter anderem so einflussreiche Unternehmen wie Google, Dell EMC oder Fujitsu Siemens. Sie möchten quelloffene Tools in diesem Bereich voranbringen. Mit Microservices-Architekturen und Self-Contained Systems sollen sich das Development und der Roll-out von Cloud-Applikationen sehr viel schneller und einfacher gestalten.

Self-Contained Systems vs. Microservices-Architektur

Die wesentlichen Architekturpatterns von Self-Contained Systems (SCS) leiten sich von den Architekturpatterns der Microservices ab. Insbesondere gehört dazu das Prinzip, unabhängig einsetzbare Einheiten zu isolieren und an organisatorischen und architektonischen Grenzen auszurichten, dem sogenannten Bounded Context. Auch in Bezug auf die Auswahl der Technologie, der verwendbaren Frameworks sowie der fehlenden Infrastruktur unterscheidet sich ein SCS nicht von einer Microservices-Architektur. Es kann also durchaus als Spezialisierung der Microservices-Architektur betrachtet werden.

Unterschiede finden sich bei Größe und Bedienbarkeit:

Eigenständige Webanwendung: Microservices-Architekturen sind in der Regel kleiner als SCS. Bei SCS handelt es sich um eigenständige Web-Anwendungen mit UI, Geschäftslogik und Daten. Sie sollten daher einen messbaren Mehrwert liefern und zu den strategischen Zielen des Unternehmens passen.
UI-Layer-Integration: Während bei Microservices-Systemen die UI nicht notwendig ist, sollte der UI-Layer in ein SCS integriert sein.

Die beiden Architekturstile haben aber vor allem vieles gemeinsam:

Wie eine Microservices-Architektur stellt auch ein SCS eine API zur Verfügung, um Funktionalität von anderen Services zugänglich zu machen. Eine Integration von unterschiedlichen Systemen erfolgt auf Ebene der Web-Anwendung, im einfachsten Fall über Hyperlinks.
Ebenso wie bei Microservices führt die Integration von unterschiedlichen SCS zu einer sehr losen Kopplung, und die einzelnen Systeme können zu einem späteren Zeitpunkt unabhängig voneinander weiterentwickelt und gegebenenfalls ausgetauscht werden.
Sowohl Microservices als auch SCSs eignen sich als Architekturpattern, um einen Monolithen in eine Cloud-native-Anwendung zu überführen.

Bei der Zerlegung einer Anwendung in ihre fachlichen Bestandteile kann der Ansatz des Domain-driven-Design (DDD) von Evans [2] unterstützen, der sowohl Entwickler als auch Fachexperten einbezieht.

Erfolgsfaktoren für Cloud-native-Anwendungen

Serviceschnitt

Einer der wichtigsten Erfolgsfaktoren für eine gelungene Cloud-native-Anwendung ist nicht zwangsläufig deren Größe (Anzahl Codezeilen), sondern der richtige Serviceschnitt. Die zentrale Frage, die sich hierbei stellt, ist: Wie zerlege ich den Monolithen so, dass die einzelnen Bausteine als unabhängige Einheiten deployt und betrieben werden können? Letztendlich sollen die Anforderungen so auf die jeweiligen Fachdomänen aufgeteilt werden, dass ein eintretendes Ereignis eine Kette von wertschöpfenden Ereignissen auslöst.

Resilienz

Eine resiliente Cloud-native-Anwendung kann Fehler mit minimalen Funktionseinbußen abfedern und verfügt über Mechanismen, um sich selbst zu „heilen“. Dies ist ein entscheidender Erfolgsfaktor für eine Cloud-Anwendung, die aus vielen separat entwickelten, deployten und skalierten Services besteht. Cloud-Provider gewährleisten zwar die Verfügbarkeit der Cloud-Infrastruktur, Ausfälle der Services einer Cloud-native-Anwendung sind da jedoch nicht inbegriffen. „Selbstheilende“ Systeme sind in der Lage, den Ausfall einzelner Microservices oder Container zu erkennen und selbständig in einen Normalbetrieb zurückzukehren.

Bei einer für Cloud-native-Anwendungen typischen großen Zahl an Services ist es nahezu unmöglich, die Verfügbarkeit der Services durch manuelles Monitoring und manuelle Eingriffe zu gewährleisten. Daher ist Self-Healing so wichtig für die Resilienz

einer Cloud-native-Anwendung. Doch wie kann ein automatisierter „Selbstheilungseffekt“ erreicht werden. Hier schauen wir etwas genauer hin:

Der Weg zum „selbstheilenden“ System

Vorbeugung ist bekanntlich die beste Therapie. Auch bei Cloud-native-Anwendungen lohnt es sich, zunächst optimale Bedingungen für fehlerfreie Funktionalitäten zu schaffen. Eine Grundvoraussetzung dafür sind Isolation und Limitierung.

Isolation und Limitierung

Wenn Fehler in einem verteilten System zu kaskadierenden Effekten führen, liegt das häufig an der nicht ausreichenden Isolierung. Bei der Isolierung geht es darum, dass ein Service andere nicht aufgrund seiner Ressourcennutzung (CPU, Hauptspeicher, Storage, Netzwerk etc.) nachhaltig beeinträchtigt.

Anwendungsressourcen werden dafür explizit limitiert. Unlimitiert neigen sie dazu, in Ausnahmesituationen über die üblichen Normwerte hinweg verbraucht zu werden. Dies wirkt sich schnell auf andere Services aus. So stauen sich beispielsweise die Requests in Thread-Pools, wenn ein Service die Anfragemenge nicht bedienen kann. Diese stehen bei fehlender Aufteilung des Pools auch für andere Anfragen nicht mehr zur Verfügung. Gleiches passiert, wenn Aufrufe dadurch immer länger brauchen und keine Request-Timeouts definiert wurden.

Eine weitere Strategie, mit Ausnahmesituationen umzugehen, besteht darin, einzelne Funktionen abzuschalten. Grundsätzlich gilt: Ein reduzierter Funktionsumfang ist besser als keine Funktion. Die Auswirkung eines Serviceausfalls ist daher auf ein Mindestmaß zu begrenzen:

> Statt einen Fehler bis zum Benutzer durchzuleiten, sollte geprüft werden, ob an geeigneter Stelle ein Fallback-Verhalten oder eine alternative Funktionalität genutzt werden kann.
> Ist eine Webkomponente nicht funktionsfähig, kann diese auf der Webseite ausgeblendet werden. Ist eine Datenabfrage nicht möglich, können die Daten aus einer lokalen Kopie geliefert werden. Das Circuit-Breaker-Pattern hilft, nach einem solchen Ausfall rechtzeitig wieder in den Normalbetrieb überzugehen. Die Funktionsweise: Wenn ein aufgerufener Service mehrfach Fehler liefert, wird temporär auf eine Fallback-Funktion umgeleitet, um den Service nicht weiter zu belasten. Durch regelmäßige Aufrufversuche wird sofort erkannt, wenn der Service wieder erreichbar ist.

Self-Healing-Mechanismen

Wie bereits erwähnt, ist es bei einer für Cloud-native-Anwendungen typischen großen Zahl an Services nahezu unmöglich, die Verfügbarkeit der Services durch manuelles Monitoring und manuelle Eingriffe zu gewährleisten. Daher ist Self-Healing so wichtig für die Resilienz einer Cloud-native-Anwendung. Doch was ist dafür nötig?

Im ersten Schritt gilt es zu erkennen, dass Handlungsbedarf vorliegt. Dazu müssen für den Service Metriken zu allen relevanten Ressourcenverbräuchen und Kennzahlen erfasst werden. Welche Ressourcen typischerweise zu Engpässen werden, hängt sehr von der Anwendung ab. Dies muss zunächst durch Lasttests und im überwachten Regelbetrieb ermittelt werden. Stehen die Normwerte fest, müssen Schwellwerte definiert und bei deren Unter- oder Überschreitung Alarme generiert werden.

Diese können dann zu automatischen Operationen wie Einschalten eines Fallbacks oder Skalierung eines Service führen. Ein zustandsloses Servicedesign vereinfacht das Auto Scaling, da ausgefallene Instanzen oder zusätzliche Last leicht durch weitere Serviceinstanzen abgefedert werden können.

Um Self-Healing zu ermöglichen, ist es außerdem notwendig, die Auswirkung von Fehlern zu begrenzen. So erhalten fehlerhafte Komponenten die Chance, in einen Normalzustand zurückzukehren. Das sogenannte Bulkhead-Muster kann bei der Fehlerbegrenzung helfen. Dabei werden Ressourcen wie Thread- oder Connection-Pools nach aufgerufenem Service partitioniert, so dass andere Services bei Ausfall nicht beeinträchtigt werden. Außerdem kann eine Aufrufdrosselung dafür sorgen, dass nur so viele Anfragen an eine Schnittstelle gehen, wie diese in ihrem aktuellen Zustand verkraften kann.

Zentrales Log-Management

Bei Cloud-native-Anwendungen erfolgt die Ausführung nicht mehr auf realen Servern, sondern in virtuellen Containern. Somit verändert sich die Lebensdauer der Umgebungen. Hiervon sind auch die Logs der Anwendung betroffen. Durch den Wegfall der

Umgebung besteht die Gefahr, dass diese komplett verlorengehen. Da die Logs für die Analyse von Fehlern und das anschließende Self-Healing dringend gebraucht werden, ist ein zentrales Log-Management für Cloud-native-Anwendungen eine sinnvolle Lösung.

Neben der Sammlung von Logs bietet ein zentrales Log-Management weitere Vorteile. Durch den Einsatz einer zentralen Event-Processing-Engine ergeben sich in der Auswertung der Logs neue Chancen. So kann die Korrelation von Firewall und Logs beispielweise Indizien ergeben, die auf einen Angriff schließen lassen.

Beim Aufbau eines zentralen Log-Management müssen einige Herausforderungen gemeistert werden:

Die Formate sind unterschiedlich. Log-Daten sind semi-strukturiert. Das Format ist von Applikation zu Applikation unterschiedlich.
Log-Daten sind verteilt. Log-Daten sind über unterschiedliche Knoten sowie Schichten (Frontend, Backend und Middleware) verteilt und aufgrund von Lastverteilung und Ausfallsicherheit mehrfach vorhanden.
Log-Datenquellen ändern sich kontinuierlich. Gerade im Bereich Cloud-Infrastrukturen variiert die Anzahl der Knoten.
Logs enthalten sensitive Informationen. Log-Daten können ggf. sensitive Daten enthalten. Insbesondere wenn ein größerer Kreis von Benutzern Zugriff auf Log-Daten bekommt, können sensitive Informationen gefiltert werden.

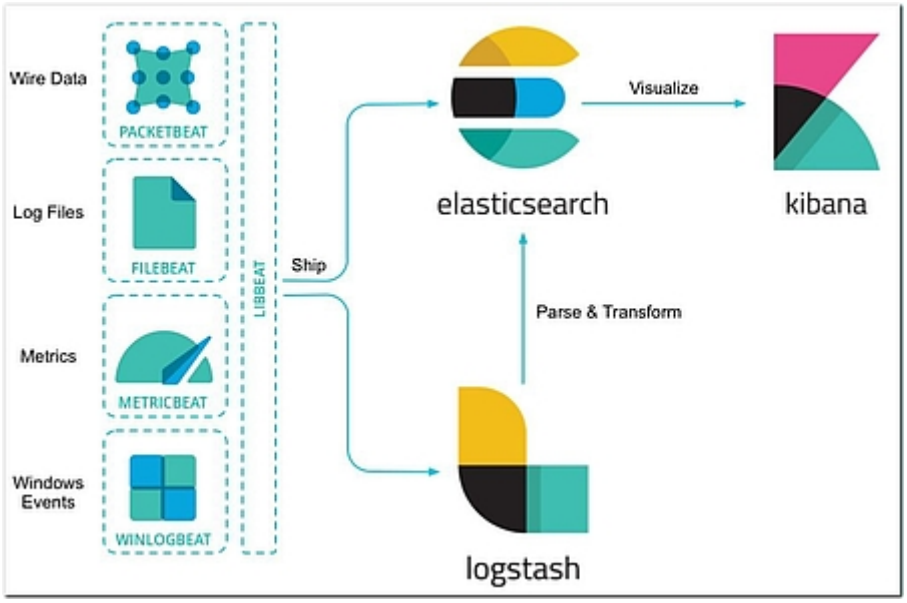

Abb. 2: Zentrales Log-Management mit dem ELK Stack

Log-Management-Lösungen wie der Marktführer ELK Stack von elastic helfen bei der Bewältigung dieser Herausforderungen. Die Architektur eines zentralen Log-Managements mit dem ELK Stack wird in **Abbildung 2** veranschaulicht. Auch Splunk [4] bietet eine kommerzielle Lösung mit vielen Features, die über das reine, zentrale Log-Management hinausgehen und spezifisch auf unterschiedliche Bedürfnisse und Stakeholder zugeschnitten sind. Etwas schlanker kommt die Open-Source-Software Graylog [5] daher, die in Kombination mit Grafana [6] für die Visualisierung eine leichtgewichtige Alternative zum komplexen ELK Stack sein kann.

Fazit

Bei der Entwicklung von Cloud-native-Anwendungen ist neben der eigentlichen Geschäftslogik noch eine Reihe von weiteren architekturellen Aspekten zu berücksichtigen. Werden alle Aspekte berücksichtigt, so können mit kurzen Innovationszyklen innovative Anwendungen entwickelt werden, die das Potenzial besitzen, die Wettbewerbsfähigkeit eines Unternehmens signifikant zu erhöhen. Neben den in diesem Artikel angesprochenen Architekturfragmenten müssen weitere Aspekte wie automatisiertes Testen sowie Build und Deployment mitberücksichtigt werden, um den Paradigmenwechsel entsprechend zu unterstützen.

Verweise

[1] dzone.com/articles/cloud-native-application

[2] Eric J. Evans: Domain Driven Design: Tackling Complexity in the Heart of Software. Boston, 2003.

[3]https://www.elastic.co/guide/en/beats/libbeat/6.2/beats-reference.html

[4]https://www.splunk.com/en_us/solutions/solution-areas/log-management.html

[5] https://www.graylog.org/overview

[6]https://grafana.com/

Stefan Kühnlein

ist als Solution Architect in den verschiedensten Projekten tätig. Sein Fokus liegt im Aufbau von robusten und zukunftsorientierten Microservices- bzw. Cloud-native-Architekturen. Aktuell unterstützt er namhafte Unternehmen bei der digitalen Transformation durch die Etablierung von entsprechenden Architekturen. Zudem betreut er das Business Development zu den Themen Cloud-Architekturen und DevOps und engagiert sich als Speaker und Autor in der Fach-Community.

E-Mail: stefan.kuehnlein(at)opitz-consulting.com

Bildnachweise:

Opitz Consulting Deutschland GmbH, www.elastic.co