Fachartikel

Was autonome Systeme und Produktorientierung mit dem Einsatz von Cloud-Native in Unternehmen zu tun haben

Cloud-Native Enterprises

von Christian Kamm, Johannes Weigend und Josef Adersberger

Cloud-native ist in den Unternehmen angekommen und schlägt dort mit voller Wucht ein: Technologisch, organisatorisch und kulturell bleibt kaum ein Stein auf dem anderen. An jeder Ecke verspricht die Disruption Hoffnungsvolles. Grund genug, innezuhalten und genau hinzuschauen: Was ist Cloud-native im Kern? Warum setzen Unternehmen so viel Hoffnung darauf und krempeln ihre ganze IT in diese Richtung um? Was von dem, was uns im Software Engineering bisher erfolgreich gemacht hat, sollten wir beibehalten, was aus gutem Grund über Bord werfen und mit Mut neu denken, wenn wir Cloud-native Anwendungen bauen?

Cloud-native Technologien sind der Katalysator für drei bedeutende, noch recht junge Strömungen im Software Engineering:

1. **Autonome Systeme:** Ein System wird entsprechend der Teamorganisation und Fachlichkeit in Teilsysteme aufgeteilt, die möglichst autonom entwickelt und betrieben werden können (*Domain Driven Design, Microservices, Self Contained Systems*). Teams skalieren damit horizontal und ihre Größe bleibt beherrschbar klein (*Agile*). Das ermöglicht einen kontinuierlichen und breiten Fluss an Features bis in den Produktivbetrieb (*Continuous Delivery*).
2. **Produktorientierung:** Teams arbeiten interdisziplinär (*Design Thinking*), unternehmensübergreifend (*Co-Creation*) und End-to-End-verantwortlich (*BizDevOps*) an Softwareprodukten und ihren Qualitätsaspekten wie Bedienbarkeit (*UI/UX*), Sicherheit (*DevSecOps*), Betreibbarkeit, Resilienz und Elastizität (*Site Reliability Engineering*).
3. **Reduzierte Verbauungstiefe:** Die vertikale Wertschöpfung wird möglichst gering gehalten; die Technik (*IaaS, PaaS, Serverless/FaaS*) und die Fachlichkeit (*SaaS, API Economy*) kommen möglichst aus der Steckdose, um den Geschäfts-Motor anzutreiben.

Zusammen genommen rütteln alleine diese drei Strömungen an fast allem, was in den letzten zwanzig Jahren in IT-Organisationen und im Software Engineering als gesetzt galt.

Warum Cloud-native?

Das Cloud-native Ökosystem ist jung, volatil und recht schwer zu überschauen (siehe Cloud-native Landkarten der CNCF [1]). Dennoch nutzen schon rund 25% der Entwickler Container-Technologien in Produktion [2]. Warum setzen so viele IT-Kapitäne auf diese Technologie, und wieso folgen die Unternehmenslenker ihrem Kurs? Es gibt mindestens zwei gute Gründe dafür:

1. **Es geht ums Überleben:** Disrupt or be disrupted. Diesen Spruch hört man seit geraumer Zeit auf C-level-Konferenzen. Im Kern steckt dahinter die Angst, die Digitalisierung zu verschlafen. Nach mehr als einem Jahrzehnt, in dem die IT-Chefs der letzten Dekade Nicholas Carr gefolgt sind („IT doesn't matter" [3]) und sie IT maßgeblich als Kostenfaktor eingestuft haben, folgt nun die Einsicht der heutigen CxOs, dass der künftige Lebensnerv aller Unternehmen digital ist und man

möglichst rasch so werden sollte, wie die FANG-Companies von Beginn an waren (Facebook, Amazon, Netflix, Google). Und genau die setzen auf Cloud-native, sie haben es erfunden und damit die digitale Revolution gestaltet. Was also liegt näher, als sie rasch nachzuahmen und damit die Hoffnung zu verbinden, eine digitale und somit überlebensfähige DNA in das eigene Unternehmen zu integrieren?
2. **Es ist cool:** Auf den ersten Blick ist das keine befriedigende Antwort auf das Warum. Wenn man sich allerdings die These vor Augen hält, dass die Welt in den nächsten Jahrzehnten mehr IT benötigt, als sie produzieren kann, wird klar, was hinter dem Cool-Sein steckt: die berechtigte Hoffnung, damit die begehrten IT-Talente zu fischen, welche die dringend benötigten IT-Anwendungen bauen können. Die Informatiker der Generation Z haben allerdings andere Lebensentwürfe als die aktuelle C-level-Generation. Geld und Karriere sind sekundär, was zählt sind Autonomie, Spaß und coole Aufgaben. Und Cloud-native bedient dieses Bedürfnis.

Es braucht Erfahrung und Durchblick, um bei all den Neuerungen zu erkennen, welche alten Weisheiten auch in Zukunft nützlich sind, um nicht im Strudel der technologischen Disruption unterzugehen.

Im Folgenden geben wir zunächst eine kompakte Einordnung des Begriffs Cloud-native und beleuchten dann einige aus unserer Sicht interessante Aspekte davon: autonome Systeme, Agilität, DevOps und Architektur von Anwendungslandschaften und einzelnen Anwendungen.

Was ist Cloud-native?

Cloud-native ist ein Sammelbegriff für Technologien und Methoden, die aus den technischen Möglichkeiten des Cloud Computings und den fachlichen Anforderungen im Internet-Zeitalter heraus entstanden sind. Die Definition der Cloud-native Computing Foundation (CNCF) von Cloud-native bringt das Nutzenversprechen dahinter auf den Punkt: „[...] allow engineers to make high-impact changes frequently and predictably with minimal toil.“[4].

Der Begriff *Cloud-native* ist mit der CNCF und ihrer Gründung im Jahr 2015 unter dem Schirm der Linux Foundation verbunden. Die CNCF fördert ein Ökosystem, das sich langsam zu einem Betriebssystem für die Cloud formiert. Im Zentrum stehen Container-Technologien wie Docker und Container-Manager wie dem de-facto Standard Kubernetes. Sie sind das Pendant zu Prozessen und System-Managern aus der Linux-Welt.

Cloud-native Technologien sind eine Dekade lang bei Internetgiganten wie Google intern gereift, bevor sie aus einer Mischung an Altruismus und Ringen um die Vorherrschaft im Cloud-Geschäft Open Source veröffentlicht und an die CNCF gespendet wurden.

Cloud-native hat sich seit 2015 von einer rein technologischen Bewegung zu einer Kultur entwickelt, die sich rund um Glaubenssätze rankt wie „Everything fails all the time“ (Werner Vogels, Amazon [5]), „The Datacenter as a Computer“ (Urs Hölzle, Google [6]), „Infrastructure as cattle not as pets“ (Bill Baker, Microsoft [7]), „Experiment & Learn Rapidly“, „Deliver Value Continuously“ [8]. Daraus werden eifrig neue Technologien und Methoden abgeleitet und engmaschig verwebt mit ihren Geschwistern Agile und DevOps.

Wie geht Cloud-native?

Autonome Systeme und Agilität

Ab einer gewissen Teamgröße funktioniert agile Software-Entwicklung nicht mehr gut. Man kann zwar auch in großen Projekten agil zum Ziel kommen, aber das ist alles andere als trivial [9].

Agilität verspricht Lösungen bei unklaren und volatilen Zielen, kurze Time to Market und höhere Nutzerorientierung. Darum ist es vielversprechend für digitale Unternehmen, möglichst agil zu sein. Also sollte man versuchen, die Anwendungslandschaft in Domänen, Produkte und Subprodukte so zu schneiden, dass Teams mit überschaubarer Größe (ideal sind sieben bis neun Personen, nie mehr als 20) möglichst autonom arbeiten können und dabei End-to-End Verantwortung für ein Produkt übernehmen. Das ist im Kern die Bedeutung von BizDevOps.

Die Dekomposition der Anwendungslandschaft eines typischen DAX-Konzerns mit üblicherweise mehr als 2000 Anwendungen in

digitale Produkte, die dann von überschaubar großen Teams weitgehend autonom entwickelt und betrieben werden können, ist hochkomplex. Mit SOA hat man es in den 2000er Jahren erstmals versucht, und ist damit mehr oder weniger gut vorangekommen. Jetzt erfolgt in vielen Unternehmen ein zweiter Anlauf, denn das Zielbild ist vielversprechend und die FANG-Unternehmen machen vor, wie es geht:

> Die Produktivität ist bei kleinen Teams am größten. Achtet man darauf, setzt man die Arbeitskraft von wertvollen Fachkräften optimal ein.
> Die Identifikation der Teammitglieder mit „ihrem" Produkt ist hoch. Die Menschen erleben Wirksamkeit bei Ihrer Arbeit, sie ziehen Sinn aus ihrem Tun.
> Neue Features bekommt man sehr rasch produktiv, ebenso rasch können etwaige Fehler behoben und Rückmeldungen von Endanwendern umgesetzt werden. Man ist wirklich agil, nicht nur in der Entwicklung, sondern End-to-End.

Das alles klingt wunderbar. Doch dafür braucht es einige Voraussetzungen (siehe Kasten 1).

Kasten 1: Voraussetzungen für autonome, agile Teams

- Eine unabhängige Software-Entwicklungsumgebung pro Team (inklusive Source Code Repository, CI-Pipeline, Testumgebung, Issue Tracker, Wiki, Chat)
- Einen fachlich klar abgegrenzten Bereich mit möglichst wenig externen Abhängigkeiten, der autonom zu entwickeln und gut testbar ist
- Für Integrationsbeziehungen: Ansätze, diese so zu entkoppeln, dass sich Probleme nicht fortpflanzen – sowohl in der Entwicklung (z.B. über Consumer-Driven Contract Testing) als auch im Betrieb (z.B. über Resilience Patterns wie Circuit Breaker).
- Prozesse und Werkzeuge, um das Produkt automatisiert auszuliefern
- Ein interdisziplinäres Team mit allen benötigten Talenten, das mit klar formulierten Zielen möglichst konvergent am Produkt arbeiten kann
- Das End-to-End-Mandat für dieses Team für ihr Produkt; das klingt harmlos, aber ist in Konzernen eine kulturelle Revolution

Cloud-native und DevOps

Agile Software Entwicklung gibt es schon seit den frühen 2000er-Jahren. Der Unterschied von damals zur heutigen Cloud-native Ära ist folgender: Nun stehen Werkzeuge zur Verfügung, mit denen Entwickler ein Produkt vollautomatisiert und standardisiert bis in Produktion bringen können. Denn Agilität bedeutet im Kern, dass man alle wichtigen Dinge kontinuierlich macht – und dazu gehört in der Softwareentwicklung, Änderungen in Produktion zu nehmen und dafür Feedback zu bekommen. Dieses ermöglichen insbesondere Cloud-native Plattformen mit ihren Build-Pipelines und Container-Technologien. Damit wird der Anwendungsbetrieb im Self-Service für Entwickler möglich, was eine Revolution im Vergleich zu bisherigen Betriebsstrukturen ist.

Derzeit ist spätestens an der Ops-Schnittstelle Schluss mit der Zuständigkeit im Team. Den Betrieb übernimmt meist ein projektferner Betriebspartner zum Preis von umfangreichen Freigabe- und Inbetriebnahme-Prozessen und Round-Trip-Zeiten von Stunden bis Tagen, manchmal sogar Wochen. In einem solchen Szenario darf nichts fehlschlagen.

In der Cloud-native Welt ist das anders: Schlägt ein Produkt-Deployment fehl, dann merkt man das in kurzer Zeit durch geeignetes Monitoring und Alerting und kann in wenigen Sekunden auf die Vorgängerversion zurückschalten oder ein korrigiertes Deployment nachschieben. Ein Deployment besteht dabei aus der Applikation nebst notwendiger Infrastruktur – beides liegt als Code vor und durchläuft dieselben Build-Pipelines. Wichtige Betriebsprozeduren wie Deployment, Rollback und Konfigurationsänderung erfolgen versionskontrolliert und voll automatisiert. Getreu der DevOps Kultur löst das Produktteam diese Prozeduren durch Code-Änderungen selbst aus, denn es weiß am besten, was zu tun ist. Das alles passiert in wenigen Sekunden. Brave new world! Man muss seine Anwendung aber entsprechend bauen: Sie muss gut betreibbar und überwachbar sein, umfassende Diagnoseschnittstellen bieten, die benötigte Infrastruktur in Form von Code definieren und robust oder gar selbstheilend sein, so dass kein Entwickler nachts raus muss, wenn es in der Produktion holpert.

Architektur von Cloud-native Anwendungslandschaften

Es bleibt die Frage: Wie schafft man es, eine große Anwendungslandschaft so in ein Produktportfolio zu schneiden, dass

überschaubar große Teams parallel und End-to-End an weitgehend autonomen Produkten arbeiten können?

Hier liefert die Kombination von etablierten und neuartigen Konzepten zumindest eine erste Fährte, wenn auch nicht die komplette Lösung: Domain Driven Design (*DDD*), Komponentenorientierung und Self Contained Systems.

Das Vorgehen ist zweistufig:

1. **Erste Stufe: Partitionierung der Anwendungslandschaft in Domänen für Produkte**
   Man zerlegt die bestehende Anwendungslandschaft in disjunkte Domänen; diese dienen als Heimat für die digitalen Produkte. Eine Daumenregel ist: pro Unternehmen ca. 10 bis 20 Domänen und je Domäne ca. 10 bis 20 Produkte. Grobe Orientierung für den Schnitt der Domänen und Produkte geben die Geschäftsprozesse auf oberster Ebene.

2. **Zweite Stufe: Produktbau aus autonomen Sub-Produkten**
   Produkte zerlegt man idealerweise in möglichst autonome Sub-Produkte, die von überschaubar großen Teams End-to-End verantwortet werden.

In der ersten Stufe reicht häufig gesunder Menschenverstand (und ein paar SOA-Prinzipien). Wenn es kompliziert wird, helfen Rahmenwerke zur Unternehmensarchitektur wie TOGAF [10] und Quasar Enterprise [11].

In der zweiten Stufe leisten Domain Driven Design, Microservice-Prinzipien und die Komponentenorientierung gute Dienste.

Komponentenorientierte Architekturen gibt es seit den späten 90ern. Typischerweise sind Architekten aus dieser Zeit darauf gepolt, azyklische und redundanzfreie Software-Architekturen zu bauen, die Zusammengehöriges in Komponenten verstecken, diese per Schnittstellen möglichst lose untereinander koppeln und mit einer exklusiven Datenhoheit ausstatten. Die Komplexität ist hinter einer Schnittstelle verborgen, die Implementierung kann bei Bedarf getauscht werden und die Datenkonsistenz ist sichergestellt. Die Anwendung wird besser wartbar und entwickelbar.

Komponenten führen aber nicht automatisch zu stark entkoppelten autonomen Produkten. Komponenten, die von vielen anderen referenziert werden, sind in einer komponentenorientierten Architektur notwendig und sinnvoll: Sie vermeiden Redundanz, bieten Wiederverwendung und sichern die Datenkonsistenz. Sie werden aber zum Problem, wenn man sämtliche Komponenten von möglichst unabhängigen Teams entwickeln und betreiben lassen will: Es kann schnell zu zeitlichen Abhängigkeiten bei der Entwicklung der nutzenden Komponente und zu einem Engpass bei der bereitstellenden Komponente kommen. Die Rechnung geht mit der Komponentenorientierung alleine nicht auf.

Hier liefert Domain Driven Design (*DDD*) eine ergänzende Lösung: Anwendungen werden in fachliche Bereiche („Bounded Context“) so zerlegt, dass eine möglichst geringe fachliche Kopplung entsteht. Redundanz von Logik und Daten ist dabei erlaubt. Die Kommunikation erfolgt lose über Events. Das führt dazu, dass es möglichst wenig zentrale Komponenten gibt, hat aber Auswirkungen auf die Transaktionalität: Atomare Transaktionen (*ACID*) werden durch Eventual Consistency (*BASE*) ersetzt.

Ein Architekturmuster zur Umsetzung von DDD ist Command Query Responsibility Segregation (*CQRS*). Dabei verfügen Dienste, die die gleichen Daten lesen und schreiben, über getrennte Datenhaltungen, die aber miteinander synchronisiert werden. Oft nutzt man dabei ein RDBMS für die schreibenden Dienste und eine NoSQL-Datenbank für lesende Dienste. Der schreibende Dienst führt bei sich transaktional Änderungen durch. Der lesende Dienst wird asynchron über ein Event über diese Änderung informiert. Die Dienste für Query und Transaktion können getrennt weiterentwickelt, ihre spezifischen Datenmodelle erweitert oder an spezielle Performanz-Anforderungen angepasst werden.
Der Preis für die autonomen Systeme ist somit die zeitweise Inkonsistenz von Daten sowie eine Redundanz von Daten und Code. There is no lunch for free! Man muss sich also die Trade-offs gut überlegen.

Kasten 2: Kriterien für den Schnitt von Microservices

- Self Contained: Sie sind lose gekoppelt und können eigenständig laufen.
- Single Responsibility: Sie erfüllen eine spezifische, fachlich sinnvolle Aufgabe.
- Service-orientierte Schnittstelle: Sie sind remote-fähig und verteilbar.
- Deployment-, Skalierungs- und Release-Einheit: Sie sind eine unabhängige Deployment Einheit und haben einen eigenständigen Release Zyklus.
- Datenhoheit: Die Daten, die ein Microservice schreibt, können von keinem anderen Microservice geschrieben werden.

Ein spezieller Ansatz zum Aufbau von Microservices, der gleichzeitig zu einer hohen Autonomie führt, sind *Self Contained Systems*. Dabei ist die UI, die Geschäftslogik und die Datenhaltung einer fachlichen Einheit in einem Microservice gebündelt und damit in einer Deployment-Einheit. Gemäß dem Prinzip „Verteile nicht, wenn Du nicht musst“ wird somit der Verteilungsgrad innerhalb einer fachlichen Einheit auf null reduziert. Damit kann diese Einheit einfacher entwickelt, deployed und betrieben werden.

Wie baut man Microservices?

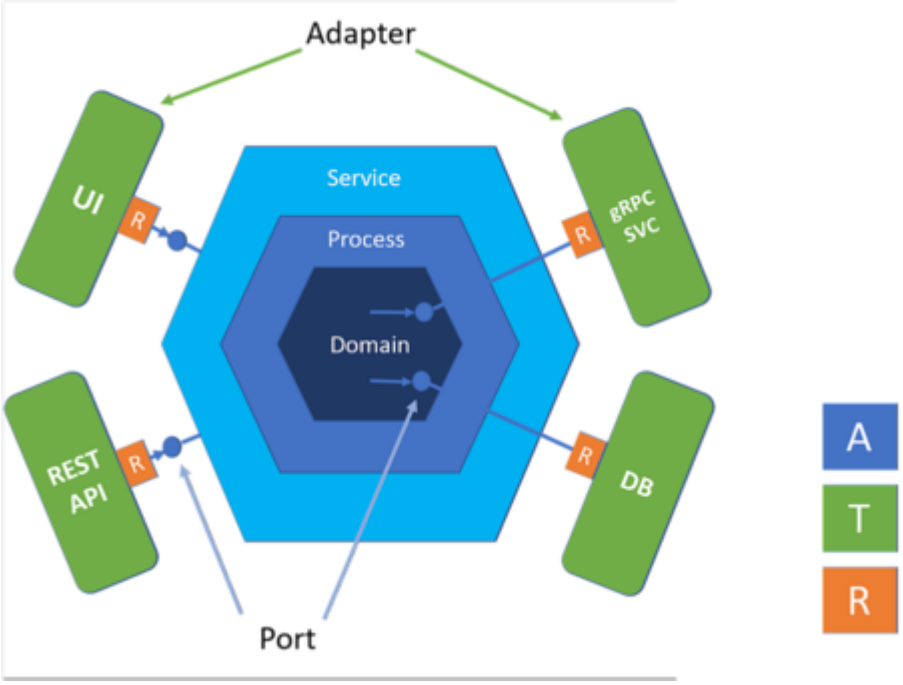

Abbildung 1: Hexagonal Architecture

Die Architektur von Microservices kann als Hexagon visualisiert werden [12]. Es gibt einen zentralen Anwendungskern mit der Geschäftslogik (Softwarekategorie A für Anwendungslogik [13], blau dargestellt): Im Zentrum sind die fachlichen Entitäten (Domain), auf die Geschäftsabläufe (Process) wirken und die über fachliche Dienste (Service) gekapselt sind. Die Kommunikation mit der Außenwelt erfolgt über Ports (Softwarekategorie R für Repräsentation, orange dargestellt). Es gibt dabei Inbound Ports, die auf Ebene der Services andocken und Outbound Ports, die auf Ebene der Domain andocken. Ports sind Schnittstellen und Teil des Anwendungskerns und somit so nicht-technisch wie möglich. Eisernes Gesetz der Softwarearchitektur ist und bleibt, die Anwendungslogik von der Technik zu trennen. Die Implementierung der Port-Schnittstellen erfolgt über Adapter. In diesen ist das technologiespezifische Wissen zur Integration der Außenwelt gekapselt (Softwarekategorie T, grün). Das können Datenbankzugriffe oder REST-Aufrufe an eine Nachbarsystem-Schnittstelle sein. Ebenso ist der Code, der den REST-Controller implementiert, ein Adapter.

Atomic Architecture: Cloud und Anwendung entkoppeln

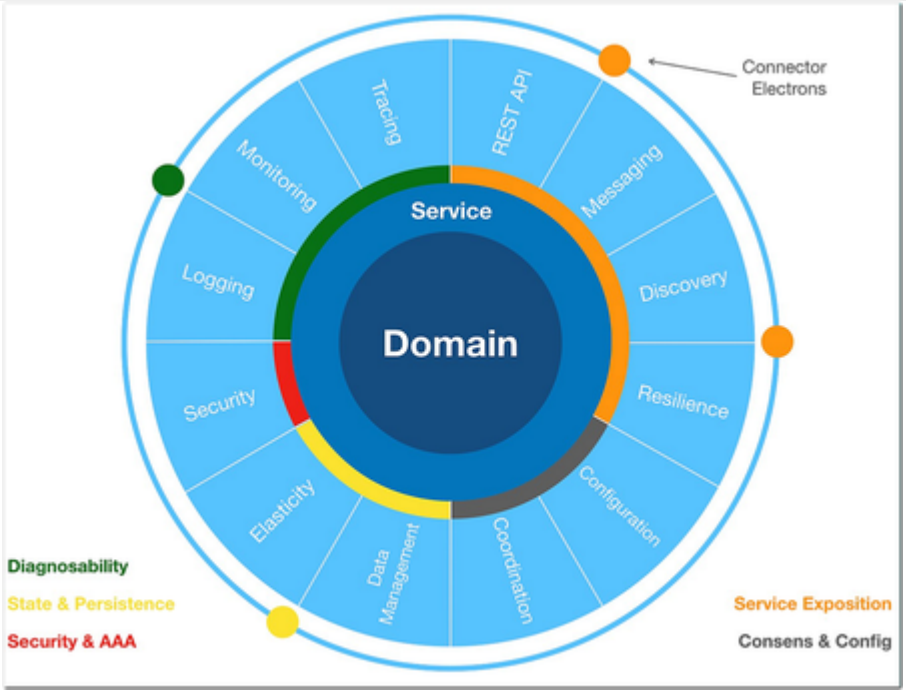

Abbildung 2: Atomic Architecture

Wenn man die beiden Konzepte von Hexagon-Architektur und Trennung von Anwendungslogik und Technik konsequent zu Ende denkt, dann entstehen Systeme, deren Anwendungslogik komplett ausführbar und testbar ist, ohne dass man sich um Verteilung, Datenbanken, Middleware und Cloud-Technologie kümmern muss. Der Trick dabei ist, sämtliche technische Querschnittsfunktionen vom Anwendungskern gezielt zu entkoppeln. Wir nennen dies Atomic Architecture [14]. Die folgende Abbildung zeigt die typischen Querschnittsfunktionen cloud-nativer Anwendungen rund um den Anwendungskern (Domain bis Service).

Klassischerweise erfolgt diese Entkopplung über mehr oder weniger standardisierte Schnittstellen (z.B. slf4j beim Logging oder JDBC bei Datenbanken) und entsprechende Bibliotheken, die diese Schnittstellen implementieren, aber austauschbar sind. In der Welt der Cloud kommt noch eine sehr vielversprechende Option mit hinzu: Die Querschnittsfunktionen werden direkt von der Infrastruktur übernommen und sind somit nicht mehr Teil der Anwendung. So können z.B. die Resilienz, bestimmte Sicherheitsfunktionen und auch Monitoring und Tracing komplett transparent von der umgebenden Infrastruktur (z.B. über ein *Service Mesh* wie Istio) übernommen werden. Noch ein wenig weiter gedacht landet man bei Serverless-Ansätzen, bei denen lediglich der Anwendungskern gegen Standard-Schnittstellen entwickelt wird. Wie diese dann implementiert werden und wie die Anwendung zum Laufen gebracht wird, ist Geheimnis der Cloud-native Plattform.

Man entkoppelt sich im Bau des Produkts von der hoch volatilen Technikwelt von Cloud-native und schafft nachhaltigen Geschäftswert: Der steckt in der Anwendungslogik, und die ist weitgehend separierbar von der vielfältigen Technologie drumherum.

Fazit

Die Cloud-native Computing Foundation (CNCF) hat den Begriff *Cloud-native* geprägt und fördert seit 2015 ein technologisches und kulturelles Ökosystem rund um Container-Technologien wie Docker und Kubernetes. Cloud-native Technologien sind in den 2000er-Jahren bei den Internet Giganten intern gereift und stehen nun als Open Source für die IT-Community zur Verfügung. Derzeit nutzen rund 25% der Entwickler weltweit Cloud-native Technologien in Produktion [2].

Unternehmen setzen *Cloud-native* maßgeblich aus zwei Gründen ein: Sie versprechen sich davon das Geschäft der Zukunft in Zeiten des digitalen Wandels; und sie erhoffen sich, damit die begehrten IT-Talente anzuziehen. Damit die neue Technologie ihr Potenzial voll entfalten kann, sind technische, organisatorische und kulturelle Veränderungen in den bestehenden Konzernstrukturen nötig.

Build-Pipelines und Container-Technologien ermöglichen Agilität End-to-End: Der Anwendungsbetrieb wird im Selfservice für Entwickler möglich und Deployments funktionieren in kurzer Zeit weitgehend automatisiert. Continuous Delivery wird damit möglich.

Komplexe Anwendungslandschaften sollten in möglichst unabhängige, autonome Produkte dekomponiert werden, so dass überschaubar große Teams diese agil entwickeln und betreiben können (BizDevOps). Domain Driven Design, Self Contained Systems, Hexagonalarchitekturen für Microservices, sowie Entwurfsmuster wie CQRS helfen bei der Architektur von autonomen Cloud-native Anwendungen. Der Nutzen von autonomen Systemen ist hohe Produktivität, der Preis dafür ist die zeitweise Inkonsistenz von Daten sowie die Redundanz von Daten und Code. Diesen Trade-off sollte man bewusst treffen.

Cloud Technologie verändert sich rasant und disruptiv. Anwendungen werden aber über Jahre entwickelt und erweitert. Es macht also viel Sinn, sich über die Kombination von Atomic Architecture und Serverless Ansätzen von der Cloud-Technologie soweit als möglich zu entkoppeln, ohne deren Vorteile bei der Entwicklung, dem Deployment und dem Betrieb zu verlieren.

Literatur & Links

[1] https://landscape.cncf.io

[2] https://insights.stackoverflow.com/survey/2019#technology-_-using-containers

[3] https://hbr.org/2003/05/it-doesnt-matter

[4] https://github.com/cncf/toc/blob/master/DEFINITION.md

[5] https://vimeo.com/1386054

[6] https://ai.google/research/pubs/pub41606

[7] http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle

[8] http://modernagile.org

[9] https://www.sigs-datacom.de/uploads/tx_dmjournals/kamm_adersberger_OS_04_16_ryyx_01.pdf

[10] https://www.opengroup.org/togaf

[11] https://www.researchgate.net/publication/221231925_Quasar_Enterprise_-_Anwendungslandschaften_serviceorientiert_gestalten

[12] http://wiki.c2.com/?HexagonalArchitecture

[13] Johannes Siedersleben, Moderne Software-Architektur Umsichtig planen, robust bauen mit Quasar, 2004, dpunkt.verlag

[14] https://de.slideshare.net/QAware/istio-playground

Christian Kamm

ist Geschäftsführer bei QAware. Er hat Mathematik und Informatik an der Universität Augsburg studiert und bei QAware das agile Projektvorgehen in Großprojekten maßgeblich geprägt. Der Einfluss von Cloud-native auf alle Aspekte des Projektmanagements interessiert ihn besonders.

E-Mail: christian.kamm(at)qaware.de

Johannes Weigend

ist Chefarchitekt, Geschäftsführer und Mitgründer der QAware. Er studierte Informatik mit Schwerpunkt „Verteilte Systeme“ an der Hochschule Rosenheim und hält dort seit 2001 Vorlesungen. Seine Themenschwerpunkte sind moderne Software Architektur, BigData und Search.

E-Mail: johannes.weigend(at)qaware.de

Josef Adersberger

ist technischer Geschäftsführer und Mitgründer der QAware. Er ist dort für das Software Engineering, die Ausbildung und als technischer Lead für das Allianz Joint Venture Syncier Cloud zuständig, das eine offene Cloud-native Plattform für regulierte Enterprises entwickelt.

E-Mail: josef.adersberger(at)qaware.de

Bildnachweise:

QAware