Advertorial

Continuous Delivery als Innovationsmotor

Ein Erfahrungsbericht aus dem Maschinenraum

von Daniel Takai

Ich schreibe diesen Artikel aus der Perspektive vom Februar 2019 und schaue auf erst 13 Jahre Cloudgeschichte seit dem Launch von Amazon EC2 zurück. Docker wird dieses Jahr sechs Jahre alt, und es ist beeindruckend, wie schnell sich diese neuen Technologien etabliert haben. Bei geschäftskritischen Systemen trifft man heute aber mehrheitlich auf solche, die älter als sechs Jahre sind, nicht im Container geliefert werden und auch nicht cloud-native entwickelt wurden. Ältere Softwareprodukte sind meistens Monolithen, die aus einer Zeit stammen, als Hardware noch ein Investitionsgut war und möglichst viel Logik auf ein und derselben physischen Maschine laufen musste. Diesen Systemen ist vor allem zu eigen, dass sie alt sind, d. h. Release- und Deploymentprozesse gibt es hier schon seit vielen Jahren. Aber das heißt nicht, dass das Ausspielen neuer Versionen technisch und organisatorisch gut gelöst ist. Ein kritischer Blick kann sich also lohnen. Continuous Delivery (CD) ist nun für Monolithen ein überraschend schwieriges Problem, und es stellt sich die Frage, welche Aspekte von CD bei Monolithen Sinn machen und welche Fallstricke zu erwarten sind.

Bevor ich mich den Fallstricken zuwende, möchte ich das Wertversprechen von Continuous Delivery wiederholen. Die Idee ist es, Änderungen an Services effizient und mit geringem Risiko durchführen zu können. Es besteht ein hohes Interesse daran, geschäftskritische Services sicher und häufig verändern zu können. Die Notwendigkeit häufiger Änderungen von Geschäftsdiensten liegt im steten Wandel der Umwelt im Kontext der Organisation begründet. Es finden täglich auf allen Ebenen Veränderungen statt, und diese muss der Dienst begleiten. Dabei ist die Effizienz in der Ausführung für viele Organisationen weniger wichtig als die Vermeidung eines Outage nach einem Deployment.

Qualitäten sichtbar machen

Als Systemarchitekt sind mir die Qualitäten der Systeme wichtig, weil diese eine Vergleichbarkeit jenseits von Funktionen erlauben. Funktionen sind schnell programmiert, beeinflussen aber die Qualität des Systems nur zu einem geringen Teil. Die eigentliche Qualität stammt aus dem Verbund der verschiedenen Dienste im Tandem mit der einsetzenden Organisation. Eine interessante Frage lautet nun: Welche Qualitäten eines Systems werden von Continuous Delivery beeinflusst?

Allen voran ist es die **Änderbarkeit** des Systems, die durch CD positiv beeinflusst wird, denn Änderungen können nun effizienter ausgespielt werden. Zudem entstehen durch die kontinuierlich durchgeführten Regressionstests eine höhere **Verfügbarkeit** und ein gutes Gefühl für die **Performance** des Systems. Niemand hat einen langsamen Build durch ewige Tests gerne, weswegen die Entwickler bereits früh damit beginnen, genau auf die Performance zu schauen und diese zu verbessern. Im Zusammenhang mit

der Performance steht auch die **Skalierbarkeit**. Es ist nicht die Skalierung der Laufzeiteinheiten, die Probleme bereitet, sondern die Skalierung des Storage, also die Datenarchitektur. Auch hier kann eine detaillierte Auseinandersetzung mit dem Thema sehr positive Auswirkungen auf die Skalierbarkeit haben. Architekturstile wie Message-Driven oder Command Query Responsibility Segregation (CQRS) können Wunder bewirken, um Bottlenecks beim Storage zu verhindern. Außerdem ist selbstverständlich die Wahl der Technologie des Storage entscheidend. Durch die besondere Berücksichtigung von Zugriffskontrolle und Identitätsmanagement steigt zudem die **Sicherheit** des Systems. Wer ein föderiertes Identity Management für die Kontrolle seiner Build Pipelines einsetzt, dem fällt es leicht, dies auch in der Applikation selbst zu integrieren.

Capabilities für Continuous Delivery

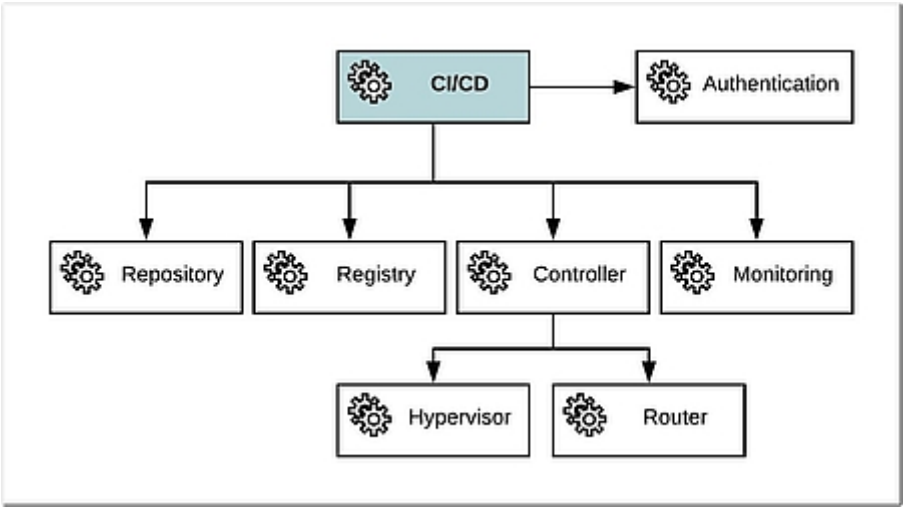

Abb. 1: Capabilities für Continuous Delivery

In den Softwareorganisationen dieser Welt ist die Ausgangslage verschieden, und so kommen Sie um eine detaillierte Analyse des Vorhandenen ebenso wenig herum wie um eine detaillierte Planung der gewünschten Zielarchitektur. Um diese planen zu können, möchte ich nun über die benötigten Capabilities sprechen, die notwendig sind, um CD möglich zu machen. **Abbildung 1** stellt diese in der Übersicht dar.

Aus der Perspektive der **Systemsicherheit** ist die Auseinandersetzung mit Authentifizierung und Zugriffsberechtigungen notwendig. Für CD ist die Fähigkeit, Personen und Maschinen zuverlässig und sicher zu authentifizieren, von hoher Wichtigkeit, genauso wie die Fähigkeit, diesen Personen und Maschinen die benötigten Ressourcen geplant verfügbar zu machen, und die Auditierung der Nutzung. Ist ein Geschäftssystem strategischer Natur, so möchte man schließlich wissen, wer wann was mit dem System erledigt hat, und zwar aus regulatorischen Gründen, aber auch weil sich daraus interessante Daten über die Nutzung des Systems auslesen lassen. Das bedeutet in der Konzeption die Auseinandersetzung mit Berechtigungen und die Etablierung föderierten Identity Managements. Eine durchgängige **Auditierbarkeit** des CD-Systems stellt ein gutes Governance Log der Aktivitäten im Betrieb dar.

Die Organisation sollte in der Lage sein, Softwareprodukte mittels Continuous Integration aus Quelltexten in Kombination mit einem Software **Repository** zu erzeugen. Hierfür muss die Organisation ein solches Repository kontrollieren und bewirtschaften, um die Abhängigkeiten zu externen Bibliotheken zu verwalten. Eine weitere Capability ist der Betrieb einer **Registry** für Machine Images. In dieser Registry sollten die benötigten Abziehbilder der Laufzeitumgebungen verwaltet werden, die dann von der CD instanziert werden können.

Es wird außerdem ein **Controller** benötigt. Dieser Dienst ist in der Lage, aus Machine Images oder Containern korrekt konfigurierte Laufzeiteinheiten zu instanzieren. Hierfür muss der Controller neben dem Lifecycle über den Hypervisor vor allem das Routing, auch Wiring genannt, der erstellten Maschinen kontrollieren.

Damit die Überwachung des Lifecycles erfolgen kann und um reiche Informationen über das Verhalten der instanzierten Maschinen zu bekommen, benötigt eine Organisation ein funktionierendes **Monitoring**. Der Controller muss sich auf das Monitoring verlassen können, damit er defekte Laufzeiteinheiten selbstständig austauschen kann.

Zusammenfassend lässt sich zu den Capabilities sagen, dass die verfügbaren *Container as a Service (Caas)*- und *Platform as a Service (PaaS)*-Umgebungen viele der genannten Capabilities bereits mitbringen. Die Einführung von CD setzt oft die Einführung von PaaS oder CaaS voraus.

Konfiguration klären

Für die Delivery benötigen wir korrekt konfigurierte Laufzeiteinheiten, üblicherweise einen Container. Damit dieser Container funktionieren kann, muss geklärt sein, wie seine Konfiguration zustande kommt. Hier gibt es die Möglichkeit, entweder bereits konfigurierte Container zu provisionieren oder aber die Konfiguration durch die Software selbst durchführen zu lassen. Zudem sind Mischformen möglich. Ein verbreitetes Pattern in der Konfiguration ist die Etablierung einer Service Registry für dynamische Service Discovery. Ebenfalls haben sich Pattern zur dynamischen Konfiguration über spezialisierte Schnittstellen einer Anwendung etabliert, bei denen die Konfiguration über einen Configuration Management Service verändert werden kann. Verbreitet ist auch die Konfiguration über Umgebungsvariablen, wie es beispielsweise die 12-Factor-App [12F] tut.

Datenarchitektur

Die größten Probleme von CD finden wir in der Datenarchitektur. Das Management der Storage States unserer Laufzeiteinheiten ist eines der dringendsten und in vielen Fällen am wenigsten verstandenen Elemente von CD. Dies liegt vor allem daran, dass dieses Management von der gewählten Persistenzstrategie des Service abhängt und deswegen einen sehr starken Bezug zur individuellen Applikation aufweist.

In der Produktionskette eines Softwaresystems ist der Einsatz von diskreten Test- und Produktionsumgebungen Standard. Um testen zu können, benötigt die Testumgebung Daten aus der Produktionsumgebung. Wenn diese Daten aber regulatorischen Auflagen unterliegen, etwa weil sie personenbezogen oder medizinischer Natur sind, beispielsweise ein Minimal Clinical Dataset, dann müssen diese Daten vor dem Transfer anonymisiert werden. Auch hier wird ein tiefes Wissen um die Softwarearchitektur des Service benötigt, um die Anonymisierung der Daten programmieren zu können. Gut also, wenn hier Entwicklung und Betrieb zusammenarbeiten (DevOps).

Ein weiterer Faktor des Managements von Storage States ist die Abhängigkeit von einer bestimmten **Softwareversion**. Der Service kann seinen Storage State im Schema verändern und garantiert so gut wie nie eine Rückwärts- oder Vorwärtskompatibilität. Aus den Strategien der **Dependency-Management**-Systeme wie Maven oder npm lässt sich hier viel lernen. Es ist somit angeraten, bei der Kopie eines Storage States die jeweilige Softwareversion zu erheben. Auf Basis dieser Daten kann man dann beispielsweise einfache Regeln schreiben, die eine Fehlkonfiguration eines Service verhindern (Version der Software muss größer oder gleich sein als zum Zeitpunkt der Kopie, oder Major Version muss gleich sein oder oder oder). Auch hier ist umfassendes Wissen über den Service notwendig und wird durch DevOps begünstigt.

Um den Storage State eines Service nach Belieben verwalten zu können, sollte der Service **crash-konsistent** sein, d. h. auch bei Inkonsistenzen in der Datenbank starten können. Crash-Konsistenz ist umso schwieriger, je mehr verschiedene Storagesysteme zum Einsatz kommen. Werden Binaries beispielsweise ins Dateisystem geschrieben, aber die Metadaten in eine Datenbank, so ist es ungleich schwerer, einen konsistenten Datenbestand abzuziehen, als wenn nur eine Datenbank oder nur das Filesystem zum Einsatz kämen. Für CD, und auch für automatische Tests, sollte es also jederzeit möglich sein, auch im laufenden Betrieb einen Volldump zur Weiterverarbeitung zu erstellen.

Die Größe der Daten, die zwischen den Umgebungen verschoben werden, ist ein wesentlicher Faktor bei der Realisierung von Continuous Delivery. Wir möchten möglichst schnell Ergebnisse von Tests und Deployment-Operationen beobachten können, und Datenbestände von mehreren Terabytes lassen sich auch mit aktuellen Storage Area Networks nicht in nützlicher Zeit transferieren. Nun ist dies auch nicht in jedem Fall nötig; wenn beispielsweise Penetration Tests oder ein Stresstest anstehen, können die dafür benötigten Umgebungen rechtzeitig einige Tage vorher vorbereitet werden. Im Daily Business, wenn es darum geht, täglich mehr als zehn Deployments zu fahren, sind lange Wartezeiten aber nicht akzeptabel. Aus diesem Grund ist es angeraten, einen kanonischen State als Fixture der automatischen Regressionstests zu nutzen, der genau so viele Daten umfasst, wie für die Tests benötigt werden.

Was die Speicherung der Storage States angeht, gibt es viele Möglichkeiten, die Daten in praktikablen Formaten in leicht manipulierbarer Form abzulegen. Eine relationale Datenbank lässt sich als SQL Dump exportieren und mit Git versionieren. Die Skripte für den Import und Export der Daten, die Entwicklern und Testern das Leben leichter machen, können zusammen mit den Storage States versioniert werden. In einer nachrichtenorientierten Architektur speichern sie am besten nur die Nachrichten, und zwar so, dass sie von der Entwicklung gut editiert werden können.

Monolithen umsichtig angehen

Wie eingangs erwähnt, sind die dicken Dinosaurier der Prä-Cloud-Ära für diejenigen, die flotte und leichtgewichtige Prozesse suchen, eine harte Nuss. Monolithen zeichnen sich in der Regel durch ihre wuchernden, allumfassenden, nicht kanonisierbaren Datenbestände aus. Mal eben die Daten für eine Testumgebung bereitzustellen gerät zu einem längeren Projekt mit vielen involvierten Personen, die alle etwas dazu sagen möchten. Transfer per Knopfdruck ist hier häufig ausgeschlossen. Aus der Not heraus finden die Teams häufig faule Kompromisse, wie etwa die Replikation der Produktionsdaten jeweils einmal zu Beginn des Jahres.

Ein weiterer Punkt, der im Umgang mit Monolithen schwierig ist, ist die in vielen Fällen äußerst magere Abdeckung mit automatischen Regressionstests. Diese benötigen wir im Rahmen von CD aber dringend, um festzustellen, dass sich der Service auch nach den Änderungen noch so verhält wie vorher. Bei Monolithen werden Fehler häufig erst spät, manchmal erst nach dem Ausspielen auf Produktion erkannt, und die geringe Testabdeckung ist hierfür der Hauptgrund. Dies wiederum stellt den Einsatz von CD per se in Frage: Wenn ich schon nicht beobachten kann, dass der Service funktioniert, wie möchte ich ihn dann automatisch in Produktion nehmen?

Der dritte Punkt, der in Bezug auf Monolithen schwierig ist, ist die Bereitstellung einer Laufzeitumgebung. Durch das jahrelange Reifen eines anachronistischen Technologiestacks sind die Laufzeitumgebungen in der Regel Snowflakeserver, die sich nur unter größten Anstrengungen automatisch herstellen lassen.

Hat man nun einen Monolithen mit an Bord, so muss man sich heute die Frage stellen, wie man diesen wieder los wird. Diese Frage lässt sich leider nicht allgemeingültig beantworten, aber eine Strategie, die sich in den letzten Jahren vermehrt wirksam gezeigt hat, ist ihn zu *erwürgen*. Dabei werden sukzessive Geschäftsdienste aus dem Monolithen entfernt und in betreibbare Softwaredienste ausgelagert. Nach und nach sinkt so der Geschäftswert des Monolithen, bis er irgendwann so wenige Funktionen hat, dass er leicht ausgetauscht werden kann. Diese Strategie benötigt einen langen Atem, kann aber für das Geschäft einer Organisation überlebenswichtig sein, um ihre Geschäftsdienste in nützlicher Zeit an die Welt anzupassen. Die größte Hürde bei einem solchen Unterfangen ist es, herauszufinden, welche Geschäftsdienste überhaupt vom Monolithen angeboten werden, und diese dann sukzessive herauszulösen. Dabei beginnt man am besten mit den Services, die am meisten von der besseren Änderbarkeit eines Microservice profitieren.

Microservices liefern

Aufgrund ihrer Entwurfsphilosophie sind Microservices klein im Funktionsumfang und damit in vielen Fällen nicht speicherhungrig. Sie eignen sich also gut für CD, weil sich ihre Storage States einfach verwalten lassen. Dafür gibt es dann eine größere Anzahl von ihnen, so dass eine übersichtliche Verwaltung, zum Beispiel in Form eines Servicekatalogs, helfen kann. Analog dazu lohnt es sich auch, ein Dateninventar der Services zu dokumentieren und die jeweilige Persistenzstrategie zu erklären. So sind alle Informationen vorhanden, um orchestrierte Continuous Delivery zu realisieren, und zusätzlich lässt sich über die Kataloge ein formales Lifecycle Management organisieren.

Erfolgreiche CD lebt von Entwurfsstandards für die Services. Verwenden alle Services dieselbe Persistenzstrategie, weisen eine API zur Konfiguration auf und lassen sich einheitlich beobachten, so können auch die Werkzeuge der CD für alle eingesetzt werden. Diese Dinge können der Entwicklung als Entwurfsstandard vorgegeben werden. Ein typischer Entwurfsstandard umfasst die folgenden Punkte; mehr Details dazu finden Sie in meinem Buch [tak]:

Der zu verwendende Message Exchange Standard, z. B. Synchron, idempotent
Das Protokoll, z. B. REST mit JSON
Das Format der API-Dokumentation

Angabe eines Domänenmodells und einer Kontextkarte mit Glossar
Vorlage für Nutzungsbedingungen
Persistenzstrategie und verfügbare Persistenzdienste
Point of Observation
Point of Control
Identity Management
Access Management

Organisation umstellen

Ich habe als Thema für diesen Artikel „Continuous Delivery als Innovationsmotor“ gewählt, weil ich gerne darüber schreiben wollte, welche Auswirkungen die Einführung von CD auf eine Softwareorganisation haben kann. CD wird häufig als Thema der Entwicklung gehandelt, obwohl wir in diesem Artikel gesehen haben, dass Architektur und Betrieb eine wichtige Rolle spielen. Natürlich profitieren auch die Fachabteilungen, denn diese bekommen ihre Changes künftig schneller geliefert. Welches sind also die zentralen Punkte, die bei der Transformation einer Organisation eine Rolle spielen?

Niemand ändert gerne seinen liebgewonnenen Rhythmus, und so gibt es von Grund auf eine abwehrende Haltung, besonders in verwurzelten Betriebsorganisationen, die seit Jahren denselben Monolithen hegen und pflegen. Das Argument der gewonnenen Geschwindigkeit zählt hier gar nicht, denn der Disconnect zwischen Betrieb und Geschäftswerten ist so groß, dass sein Zweck die beteiligten Personen vermeintlich gar nicht betrifft. Sie sind auch nicht in die Kommunikation im Projekt integriert, sehen die Probleme nicht, die die unflexible Software schafft, und können deswegen auch keine **Motivation** entwickeln, um Veränderung zu unterstützen. Mit einfachen Praktiken der Facilitation versuche ich hier, die Beteiligten zur Lösung an einen Tisch zu bringen, um Verständnis zu schaffen und diese Hürden zu überwinden.

Ein weiterer wichtiger Punkt in der Verhinderung von Veränderung sind individuelle Befürchtungen. Weil die betroffenen Personen das Ergebnis der Veränderung noch nicht kennen, weil sie es selbst noch nicht erlebt haben, bilden sich mehr oder weniger bewusste **Befürchtungen**, und es wird dann erstaunlich viel Energie in die Verhinderung investiert (Uncanny Valley). Sehr hilfreich ist eine offene Diskussion der zu erwartenden Ergebnisse unter Einbezug der individuellen Situation der Mitarbeitenden. Anstatt wie bis anhin einzelne Snowflake Server zu pflegen, entwickelt das Betriebsteam nun Skripte für die Automatisierung und Beobachtung der Dienste. Eine anspruchsvollere, aber sinnstiftendere Tätigkeit. Der Aufwand für solche Transformationen ist nicht zu unterschätzen, denn die liebgewonnenen Tätigkeiten, die zur Diskussion stehen, sind ein Teil der Arbeitskultur, und Kultur ist nun mal gegenüber Veränderungen resistent.

Zusammenfassend sind es also Maßnahmen im Bereich der Entwicklung einzelner Personen, Diskussionen in der Gruppe zur Entwicklung gemeinsamer Ziele, die Zusammenlegung von Betrieb und Entwicklung im Rahmen von DevOps sowie das Wecken von Motivation bei den Beteiligten. Ich würde sogar so weit gehen, dass eine selbstständige und eigenverantwortliche Systemorganisation eine notwendige Capability für erfolgreiche CD ist.

Fazit

Ich habe gezeigt, dass der richtige und reflektierte Einsatz von CD, auch bei Monolithen, die damit umgehen können, sowohl aufgrund der Kosteneffektivität im Deployment, der Veränderung von Aufgaben und Verantwortlichkeiten im Betrieb, als auch in Bezug auf die Qualität des produzierten Systems vorteilhaft und nützlich ist. Dabei ist Continuous Delivery umso einfacher, je mehr die zentralen Qualitäten von Cloud-Native-Systemen erreicht werden, allen voran die Skalierbarkeit, Performance und Änderbarkeit. Herstellern älterer Systeme sei angeraten, für ihre Monolithen Werkzeuge für CD anzubieten, um den Kunden den Betrieb zu vereinfachen, beispielsweise die vorgeschlagenen Tools für den sauberen Export und Import von Daten, oder Werkzeuge der Kanonisierung und Anonymisierung.

Beste Bedingungen für CD hat, wer auf Wandel eingestellt ist, einen ausreichend budgetierten Betrieb vorweisen kann, ein Team hat, das miteinander spricht und an der Zusammenarbeit Freude findet, sowie skalierbare, sichere und änderbare Services in Produktion nehmen möchte.

Links & Literatur

[12F] 12 Factor App: https://12factor.net

[tak] Daniel Takai, Architektur für Websysteme, Hanser Verlag München, 2017

Daniel Takai

ist Autor, Facilitator, Unternehmens- und Lösungsarchitekt der Silberrücken AG im Kanton Bern. Er steht Organisationen bei Software- und Transformationsprojekten mit seiner Erfahrung und Expertise zur Seite.

E-Mail: takai(at)silberruecken.ch

Bildnachweise:

Daniel Takai