JAVA Mag
Java magazin™
Java | Architektur | Software-Innovation

Agile und DevOps

Einzeln gut, zusammen besser

F Grao/Shutterstock.com

www.javamagazin.de
Ausgabe 8.2017

Agil trifft DevOps

Best Practices für die Cloud-Entwicklung

Was ändert sich, wenn ein bereits agiles und nach dem DevOps-Ansatz arbeitendes Team damit beginnt, für die Cloud serverless zu entwickeln? Wir beleuchten diese Frage anhand der Erfahrungen aus einem realen Projekt, in dem das Team einen Serverless-Cloud-Technologiestack entwickelt hat. Fazit: Es bleibt (fast) alles beim Alten. Nur anders.

von Andreas Arnold und Stephan Dahm

Warum spielt DevOps oder genauer gesagt, die Kombination von DevOps und Agile, gerade für die Cloud-Entwicklung eine Rolle? Ganz lapidar gesagt, weil es bei DevOps darum geht, kontinuierlich auszuliefern, seien es Produkte, Releases oder Funktionalitäten. DevOps konzentriert sich auf den direkten Zusammenhang zwischen Kundenwert und kontinuierlicher Auslieferung. Auch die agile Softwareentwicklung zielt letztendlich darauf ab, in kurzen Intervallen lieferfähig zu sein. Genau das erwarten viele Unternehmen und deren Kunden von einer Cloud-Lösung: Schnelle Verfügbarkeit. Schwergewichtige und lange Entwicklungszyklen wollen oder können sich viele Unternehmen nicht mehr leisten.

Wenn Wert in dem Moment entsteht, in dem etwas ausgeliefert wird, dann reicht es nicht, sich auf einzelne Stationen der Wertschöpfungskette zu konzentrieren. Deshalb ist es nicht überraschend, dass agile Softwareentwicklung und DevOps eine gemeinsame Basis haben: Kern ist bei beiden die Betrachtung der gesamten Wertschöpfungskette von der Idee oder Anforderung bis zum Feedback des Markts. Unsere Definition von DevOps geht daher nicht in die Richtung „DevOps beleuchtet den Prozess ab erfolgreichen Tests bis Kundenfeedback“. Wenn es um den Kundennutzen geht, dann ist unserer Meinung nach nur die ganzheitliche Betrachtung sinnvoll. Dabei ergänzen sich DevOps und agile Softwareentwicklung sehr gut.

Serverless in die Cloud: wo weniger mehr ist

Bleibt die Frage, warum ein Produkt in der Cloud laufen oder serverless entwickelt werden soll, wenn das primär noch kürzere Releasezyklen und damit zusätzliche Komplexität in der Entwicklung bedeutet. Hier seien nur zwei der Vorteile genannt: Bei kommerziellen Cloud-Lösungen besteht ein Vorteil in den – je nach Anwendungsfall – oft niedrigeren Kosten im Vergleich zu einer Vollversion. Serverless-Modelle sehen in der Regel vor, nach der tatsächlichen Laufzeit der Funktionen zu bezahlen. Außerdem nimmt der Cloud-Dienstleister den Kunden viele zeitraubende Dinge ab. Das beschleunigt häufig die Umsetzung und damit die Time-to-Market. Man könnte also sagen, dass die Serverless-Entwicklung für die Cloud das, was sie fordert – kurze Lieferfristen – auch selbst ermöglicht.

Um ein potenzielles Missverständnis auszuräumen: Gerade im Sinne kontinuierlicher Auslieferungen ist DevOps eine wichtige Ergänzung zur agilen Softwareentwicklung, kein Ersatz dafür. DevOps konzentriert sich auf die kontinuierliche Auslieferung immer neuer, für den Kunden wertvoller Funktionen. Agile Softwareentwicklung schafft die handwerklichen und prozessorganisatorischen Voraussetzungen, die notwendig sind, damit überhaupt auslieferfähige Inkremente erzeugt werden können.

Nach unserer Beobachtung haben einige Firmen und Teams leider die Herausforderung agiler Softwareentwicklung (noch) nicht gemeistert. Tatsächlich arbeiten sie nur in Teilen wirklich agil. Auch die agile Denkweise hat sich nicht überall durchgesetzt, wo formal nach agilen Prinzipien gearbeitet wird. Typisch für diese Praxis ist beispielsweise eine kleinteilige Aufteilung der Arbeit auf einzelne Köpfe und die Kommunikation mithilfe von Dokumenten. Damit sind wichtige qualitätsbezogene Voraussetzungen nicht erfüllt. Das kann es schwer bis unmöglich machen, am Ende jedes Sprints ein hochwer-

tiges Inkrement zu liefern. In diesen Fällen ist es eher unwahrscheinlich, dass es den Betroffenen gelingen wird, DevOps gewinnbringend für den Kunden umzusetzen. „Shipping shit faster" ist definitiv nicht das Ziel von DevOps. Solange die Methoden der agilen Softwareentwicklung nicht etabliert sind, empfehlen wir, sich zuerst darauf zu konzentrieren, bevor DevOps zum Thema wird. Ein guter Indikator ist hier die Zuverlässigkeit des vom Team am Sprint-Ende gelieferten Umfangs.

Blick in die Praxis: Serverless und die agile Entwicklung

Gehen wir davon aus, dass die beiden Voraussetzungen gegeben und sowohl fundierte Agil-Kenntnisse als auch Erfahrung mit DevOps-Techniken im Team vorhanden sind. Dann sind immer noch diese Fragen offen: Vor welchen Besonderheiten sieht sich nun ein Team, das einen Serverless-Cloud-Technologiestack entwickelt? Welche Best Practices sind gerade in einem solchen technologischen Umfeld hilfreich? Grundvoraussetzung sind zwei alte Bekannte: Crossfunktionalität und Verantwortung:

- Das Team ist mit den nötigen Fähigkeiten ausgestattet, um Betriebsthemen zu bearbeiten.
- Das Team hat die Verantwortung, auch den Betrieb auf dem Zielsystem zumindest für eine Testinstallation, am besten für die produktive Umgebung, sicherzustellen.

Wenn wir uns die agilen Werte, Scrum und die Verantwortung des Teams anschauen, beginnt alles mit der Definition of Done – jenem Dokument, das die Verantwortung des Teams und das zu liefernde Inkrement genau beschreibt. Welche Anpassungen oder Ergänzungen haben wir hier Cloud-gerecht vorgenommen? Unverändert wichtig sind Qualitätsaspekte wie Unit-Tests und der Softwarequalitätsindex. Dazu haben wir in unserem Projekt in der Definition of Done Punkte notiert wie „Alle Dienste laufen in der Cloud" und „Vollständig in die Cloud-Zielumgebung ausgeliefert". Das hat zur Folge, dass sich das Entwicklungsteam ebenfalls um die Inbetriebsetzung der Lösung kümmern muss und dabei zusätzlich motiviert ist, für die entsprechende Qualität und einen hohen Automatisierungsgrad zu sorgen. Weitere Punkte sind je nach Anforderung an die Teams individueller Art – insbesondere die Übernahme von First-Level-Support in den Kernzeiten oder zumindest Unterstützungsangebote sind denkbar. Diese Anforderung hatten wir allerdings nicht.

In dem Projekt haben wir uns an den Best Practices bestehender klassischer Projekte orientiert und eine dreiteilige Zielumgebung geschaffen: Dev, Test und Prod. Die Dev-Umgebung wird vollautomatisch bei jedem Einchecken ausgeliefert, auf Test und Prod werden Tags automatisiert ausgeliefert, wenn gewünscht. Die Idee dahinter ist folgende: Die vollständig automatisiert ausgelieferte Dev-Umgebung ist vergleichsweise instabil. Hat z. B. ein Entwickler beim Committen vergessen, die Skripte entsprechend seiner geänderten Funktionalität anzupassen, kann der Cloud-Dienstleister nicht wissen, welchen neuen Dienst er bereitstellen muss, damit die Anwendung funktioniert. Deshalb wird auf Test nur dann deployt, wenn das System stabil läuft. Dort kann dann der Product Owner die neuen Features abnehmen. Wenn alles läuft, steht der Produktivnahme des gerade abgenommenen Stands nichts mehr entgegen.

Während in vielen produktiven Systemen eher auf Kanban gesetzt wird, um neue Features oder Bugfixes schnellstmöglich produktiv zu nehmen, hatten wir uns in der Entwicklungsphase für Scrum entschieden. Das ermöglicht eine bessere Releaseplanung. Wie auch in anderen agilen Projekten, muss man hier abwägen, wo man die Priorität setzen will. Ist die reine Time-to-Market entscheidend, weil es um Tage geht und nicht um Wochen, ist Kanban vermutlich die bessere Lösung. Steht die Planbarkeit im Vordergrund, etwa bei einer Neuentwicklung, halten wir Scrum für geeigneter.

In der Cloud sind neue Ressourcen wie Datenbanken oder Frameworks des Cloud-Anbieters schnell zu bekommen. Das bedeutet auch, dass sie sich oft zwischen Releases ändern. Damit man den Überblick nicht verliert, welche Ressourcen man für welche Version benötigt, hat es sich etabliert, die Ressourcen als einen Teil des Codes zu betrachten. Sie werden mithistorisiert und automatisiert erstellt.

Knackpunkt Infrastructure as Code

Hier spielt auch DevOps eine Rolle. Die Ressourcen werden vom Team angelegt. Daher ist es ebenfalls Aufgabe des Teams, sie richtig zu konfigurieren und das System zum Laufen zu bringen. Der klassische Betrieb liegt nun beim Cloud-Anbieter, der sicherstellen muss, dass der Service auch zur Verfügung gestellt wird.

Es gibt eine recht große Zahl an Tools, mit denen man die Ressourcen erstellen kann. Wir haben drei Tools verglichen: CloudFormation, Terraform und Serverless. CloudFormation hält viele AWS-Ressourcen bereit, ist unserer Meinung nach aber – bezogen auf die drei

Serverless: eine Definition

Serverless bezeichnet hier die vollständige prozess- und maschinenlose Entwicklung eines Softwareprodukts. Anders gesagt sind die Entwickler davon befreit, spezielle Rahmenbedingungen wie das installierte Betriebssystem oder Skalierungsvorgaben einzuhalten, da diese Faktoren in der Verantwortung der Cloud-Betreiber liegen. In der aktuellen Praxis funktioniert Serverless über die Verwendung diverser Plattformen (PaaS) oder Dienste (SaaS) gängiger Cloud-Betreiber. Letztere bieten mittlerweile ein reiches Spektrum an Frameworks an, z. B. Cognito von AWS zur Synchronisierung der Daten zwischen Cloud und mobilen Endgeräten. Nachteilig finden wir die explizite Bindung an den jeweiligen Cloud-Dienstleister. Die Open-Source-Lösungen weisen nach unserer Erfahrung aktuell geringere Funktionsumfänge auf.

Gerade im Sinne kontinuierlicher Auslieferungen ist DevOps eine wichtige Ergänzung zur agilen Softwareentwicklung, kein Ersatz.

Tools – das umständlichste Tool. Serverless kommt mit einer kleinen Menge an Konfigurationscode aus, bietet aber im Gegenzug nicht für alles eine Lösung. Frei nach dem Motto „Code wird viel häufiger gelesen als geschrieben" fanden wir Serverless trotzdem am praktischsten. Wenn ein Feature fehlt, das man dringend braucht, ist es möglich, eigene Plug-ins zu schreiben und einzubinden.

Unsere Erfahrung lehrt, dass es wirklich wichtig ist, die Infrastruktur so weit wie möglich automatisiert zu erstellen. Gerade beim Übergang von Dev nach Test nach Prod zeigt sich schnell, dass beim Ausführen der Skripte immer wieder Kleinigkeiten nicht automatisiert wurden. Das kann einen Rollback nötig machen und damit die Fehlerwahrscheinlichkeit deutlich erhöhen. Wenn alles mit dem Code eingecheckt ist, wird die Infrastruktur direkt beim Ausliefern des richtigen Tags mit aktualisiert. Man sollte auch das Dev-System nur vollständig automatisiert erstellen lassen, am besten im Build-Server direkt beim Einchecken. Nur dann kann man sich sicher sein, dass die Infrastrukturskripte wirklich alles enthalten, was für das lauffähige System benötigt wird, und man erhält ein unmittelbares Feedback.

Was, nicht Wie

Generell konnten wir festhalten, dass der Fokus bei der Serverless-Entwicklung während der gesamten Implementierung auf dem Was liegt – das heißt, auf der Frage, was der Code leisten soll und welche Logik benötigt wird. Hingegen spielen beim Schreiben des Codes die Wie-Fragen keine Rolle mehr, etwa wie der Code aufgerufen wird oder wie der Programmablauf aussieht. Ausnahmen sind allenfalls die Konfiguration und die Infrastrukturskripte. Das Entwicklungsteam muss nicht mehr viel über Skalierung und sonstige Programmablauf- oder Ressourcenthemen nachdenken, was seine Arbeit erleichtern dürfte. Diesen Part übernehmen die Serverless-Frameworks der Cloud-Anbieter. Stattdessen kann sich das Team (fast) vollständig auf die fachlichen Anforderungen konzentrieren, was besonders funktionalen Sprachen oder einer funktionalen Schreibweise in verbreiteten Sprachen entgegenkommt.

Frameworks: Auf die Größe kommt es an

In einem Punkt sollte man wieder etwas weiter zurückblicken, bis auf die Zeiten, in denen Entwickler beim Programmieren noch stärker auf die Größe des Deliverables achten mussten. Gerade wenn man sich für den Serverless-Ansatz entscheidet, ist es wichtig, die Größe einer Serverless-Funktion im Blick zu haben. Diese Erfahrung haben wir relativ früh im Projekt gemacht. Die Serverless-Funktionen haben eine hohe initiale Laufzeit und werden erst mit der häufigen Verwendung wieder schneller. Daher ist es meist wichtig, dass die Funktionen nicht erst einen ganzen Sprint-Boot-Container hochfahren und so Zeit vergeuden, um ein Framework zu initialisieren, von dessen Funktionalitäten man nur einen Bruchteil benötigt. Deshalb kommt früher oder später jedes Framework auf den Prüfstand. Empfehlenswert sind hier besonders kleinteilige Frameworks, d. h. kleine Pakete, die möglichst genau den gewünschten Funktionsumfang abbilden und nicht eine Vielzahl von Funktionen aufweisen, die nur die initiale Ladezeit erhöhen. Letztere fällt zwar nicht bei jedem Aufruf an, bei niedriger Last jedoch immer mal wieder bei einem Anwender. Steigt die Last, sieht sich bei jedem Skalierungsschritt ein Anwender von hohen initialen Ladezeiten betroffen.

Nach CI wird CD zum wichtigsten Werkzeug

Wir sind zu der Überzeugung gelangt, dass sich die Technik der Continuous Integration (CI) zu einem Continuous Deployment (CD) weiterentwickelt und auch weiterentwickeln muss, um den Anforderungen an die Cloud-Architektur und einer kurzen Time-to-Market gerecht zu werden. Wenn das Entwicklungsteam auch für das Thema Betrieb verantwortlich ist, benötigt es eine Build-Pipeline, die auch die Infrastruktur abdeckt, denn nur dann erhält es das erforderliche Feedback früh genug.

Die AWS CodePipeline bietet eine Möglichkeit, die Pipeline von Code zu der laufenden Anwendung hin zu definieren. Ein Beispiel dafür, wie diese Definition aussehen kann: Wenn eine Codeänderung eingecheckt wird, triggert sie einen automatischen Build. Nach dem Build werden die Unit-Tests durchgeführt. Danach wird die Anwendung auf der Dev Stage ausgeliefert. Man würde also alle Microservices und auch die Webseiten automatisch in die entsprechenden S3 Buckets hochladen und die Lambdafunktionen aktualisieren. Nun lässt man automatisierte Integrations-, Akzeptanz- und UI-Tests auf der neuen Version laufen. Nachdem diese Tests erfolgreich waren, kann man die neue Version automatisiert auf dem Testsystem für manuelle Abnahmetests bereitstellen. Die Codepipeline führt den nächsten Schritt nur dann aus, wenn der vorhergehende erfolgreich war. So kann man sicherstellen, dass auf dem Testsystem immer nur dann eine neue Version ausgeliefert wird, wenn sie sicher funktioniert. Sobald die manuellen Abnahmetests erfolgreich waren, sollte man die Anwendung auch automatisiert auf die Produktivebene ausliefern.

Bei einer wachsenden Anwendung wird man früher oder später die Codepipeline für die verschiedenen Microservices splitten. Die Integrations-, Akzeptanz- und

UI-Tests bleiben jedoch auf der gesamten Anwendung, um sicherzustellen, dass sich die aktualisierten Services auch verstehen. So kann jeder einzelne Microservice für sich ausgeliefert werden. Die Regeln von Clean Code und agiler Softwareentwicklung bleiben unverändert bestehen. Gerade bei sehr kurzen Deploy-Zeiten sind eine hohe Codequalität und gute Wartbarkeit sehr wichtig, und auch die Testautomatisierung ist unabdingbar.

Microservices in ihrem Element

In der Cloud haben sich Microservices etabliert. Die Idee dahinter ist, dass eine Anwendung nicht mehr aus einem Stück besteht, sondern aus vielen kleinen Services, die miteinander kommunizieren. Dabei legt der Begriff Micro bereits nahe, dass es sich um einen kleinen Service handelt, der sich um genau eine Aufgabe kümmert. Dieses Prinzip ist schon aus der Objektorientierung bekannt, nur mit dem Unterschied, dass ein Microservice seinen Status in einer Datenbank speichern muss, während ein Objekt den Status selbst enthält. Gerade in einer Serverless-Architektur muss ein Microservice zustandslos sein, da man gar nicht weiß, ob der nächste Aufruf auf dem gleichen Server erfolgt oder nicht. Die Microservices machen es daher recht einfach, einen Unit-Test zu schreiben. Wichtig ist nur, dass man sie so schreibt, dass man alle externen Datenquellen, also andere Microservices oder Datenbanken, per Dependency Injection austauschbar macht, sodass man im Unit-Test einen definierten Zustand übergeben kann.

REST-Services und Queues nutzen

Um die Microservices voneinander zu entkoppeln, nutzt man entweder synchrone HTTP-Aufrufe oder für asynchrone Operationen eine Queue. Gerade eine Queue bietet den großen Vorteil, dass der aufrufende Service auch dann noch funktioniert, wenn der asynchrone Service gerade nicht verfügbar ist. Die benötigte Operation wartet dann in der Queue, bis der Service wieder läuft. Bei synchronen Aufrufen ist das natürlich nicht möglich. Hier sollte man den Anwender über die Fehlercodes und Fehlermeldungen informieren, ob die Operation gar nicht möglich (Daten falsch oder nicht vorhanden) oder ob sie nur temporär nicht verfügbar ist. Da die einzelnen Services jederzeit neu ausgeliefert werden können, die Deploy-Zeiten aber sehr kurz sind, kann ein Anwender die Operation eine kurze Zeit später erneut probieren. Da diese Fehlerbehandlung noch deutlich wichtiger ist als bei klassischen Anwendungen, sollte man auch Unit-Tests schreiben, die die Fehlercodes prüfen.

Die Testpyramide bleibt

An dem Prinzip der Testpyramide ändert sich nichts. Am wichtigsten sind weiterhin die Unit-Tests, denn sie stellen sicher, dass die neu ausgelieferten Services auch ihre Spezifikation erfüllen, also mit den anderen Services kommunizieren können. Deshalb prüft man vor dem Ausliefern der neuen Version in der Cloud, dass die Unit-Tests alle laufen. Sind sie erfolgreich, kann das CD gestartet werden. Wenn die neue Version ausgeliefert wurde, kann man die Integrationstests in der Cloud laufen lassen. Sie sollen nun nicht mehr die Funktionalitäten der Services testen, sondern nur noch deren Zusammenspiel.

Der nächste Schritt sind die Tests des User Interface, die nun nur noch prüfen sollen, ob die Daten richtig angezeigt werden. Waren die UI-Tests erfolgreich, ist es ratsam, noch eine manuelle Abnahme folgen zu lassen, bevor das System produktiv genommen wird. Um schnelle Deploy-Zeiten zu gewährleisten, ist es hier wichtig, möglichst viel zu automatisieren und wenig manuell zu testen.

Der Fokus der Integrationstests

Microservices haben den großen Vorteil, dass man eine Komponente schnell erweitern oder korrigieren kann, ohne das ganze System austauschen zu müssen. Dadurch kann eine Webseite weiter funktionieren, wenn nur ein kleiner Teil ausgewechselt werden muss. Einen Nachteil bringt dieser Umstand allerdings mit sich: Vor der Produktivnahme muss sichergestellt sein, dass der neue Service auch in dem System funktioniert. Dafür benötigt man Integrationstests. Bei klassischen monolithischen Anwendungen liegt der Fokus der Integrationstests darauf, die Integration anderer Systeme sicherzustellen. Jetzt verlagert sich der Fokus darauf, das Zusammenspiel der einzelnen Module zu prüfen, sprich der Microservices. Dementsprechend werden Integrationstests jetzt noch weitaus wichtiger, da es wesentlich mehr Microservices gibt als fremde Komponenten. An der Testpyramide und ihrer Sinnhaftigkeit ändert das jedoch nichts: Der Unit-Test prüft die Funktion des Microservice, der Integrationstest kontrolliert nur noch das Zusammenspiel.

Automatisierte Anwendungstests werden auch wichtiger, um ein schnelles Deployment mit hoher Qualität zu ermöglichen. Manuelle Abnahmetests sollten sich nach Möglichkeit nur damit beschäftigen müssen, neue Features auf Anwendbarkeit zu prüfen. Je mehr Tests man automatisieren kann, desto geringer ist das Risiko bei einer neuen Version. Und damit kann man auch bei mehreren Deployments innerhalb einer Woche eine hohe Qualität sicherstellen.

Andreas Arnold ist seit 10 Jahren Entwickler bei andrena objects. In den letzten 2 Jahren hat er sich intensiv damit beschäftigt, das Thema Agilität und Cloud zusammenzubringen.

Stephan Dahm hat eine starke Historie als Software Engineer und sich zu einem Scrum Master und Agilem Coach entwickelt. Seine Leidenschaft ist die Formung und das Coaching von cross-funktionalen Teams vor dem Hintergrund aktueller Technologien und Herausforderungen. Dabei liegt für ihn der Fokus auf der Software Qualität.