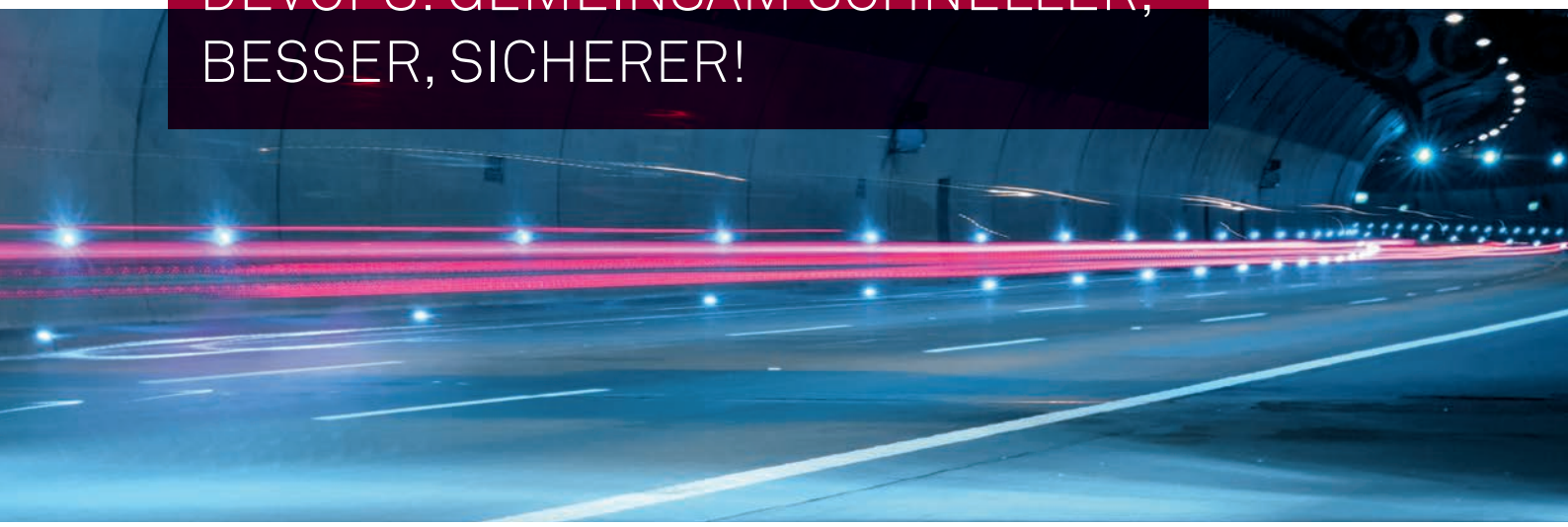
DEVOPS: GEMEINSAM SCHNELLER, BESSER, SICHERER!



Teil II der DevOps-Serie: Die Kernbestandteile einer kontinuierlichen Delivery Pipeline

| von ERIK BENEDETTO und DR. ANDREAS ZAMPERONI[1]

Im ersten Teil unserer Artikelreihe wurde der zugrunde liegende Konflikt zwischen Softwareentwicklung und Betrieb untersucht.[2] Wir zeigten auf, warum gerade in der öffentlichen Verwaltung durch die digitale Transformation der Handlungsdruck, schneller und flexibler zu werden, groß ist und warum es deshalb umso wichtiger ist, im Zusammenspiel zwischen Entwicklung und Betrieb eine schnelle Time-to-Market zu realisieren. Hierfür wurde DevOps als Mechanismus, der den zugrunde liegenden Konflikt auflösen kann, skizziert und als ein mögliches Lösungsmodell vorgestellt.

In diesem Teil werden wir zuerst die Begriffe DevOps, Continuous Integration, Continuous Delivery und Continuous Deployment schärfer abgrenzen, um dann im Verlauf die Bausteine einer kontinuierlichen Delivery Pipeline näher zu betrachten.

ZUSAMMENFASSUNG TEIL I

Entgegengesetzte Ziele und Wege der Entwicklung und des Betriebs verlangsamen die Auslieferung von neuen Verfahren und Funktionalitäten an die Kunden. DevOps bedeutet den Schulterschluss zwischen Softwareentwicklung und IT-Betrieb. Durch die Auflösung des grundlegenden Konflikts der beiden Einheiten sollen kürzere Releasezyklen realisiert werden.

Eine schnellere Auslieferung wiederum führt zu glücklichen Kunden. Um die Softwareauslieferung auf die gewünschte Geschwindigkeit des Business zu beschleunigen, müssen sich also die Ziele und Wege von Entwicklung und Betrieb annähern.

DevOps stellt die Kollaboration von Entwicklung (Dev) und Betrieb (Ops) in den Mittelpunkt. Diese werden in der Regel getrennt in Silos betrachtet und sind aus Gründen der Arbeitsteilung, Spezialisierung und höheren Effizienz durch Standardisierung und Industrialisierung eigenständig. Das gemeinsame Ziel, einen möglichst guten Service für interne oder auch externe Kunden anzubieten, gerät dabei leicht in Vergessenheit. Eine Trennung der Organisationseinheiten ist in einer unterschiedlichen Perspektive auf die Anwendungen begründet.

GEMEINSAM: CONTINUOUS INTEGRATION, CONTINUOUS DELIVERY UND CONTINUOUS DEPLOYMENT

Der Betrieb schaut auf Anwendungen als Betriebssystemprozesse und überwacht diese mittels entsprechender Tools (CPU- oder I/O-Last), um so das Verhalten der Anwendung und auch Incidents zu analysieren. Was tatsächlich im Betrieb innerhalb einer Anwendung abläuft, ist für die Mitarbeiter im Betrieb eine Black Box, da

1 Dieser Artikel erschien ursprünglich in der msgGillardon NEWS und wurde für die .public von Dr. Andreas Zamperoni angepasst.
2 Siehe .public 02-2017

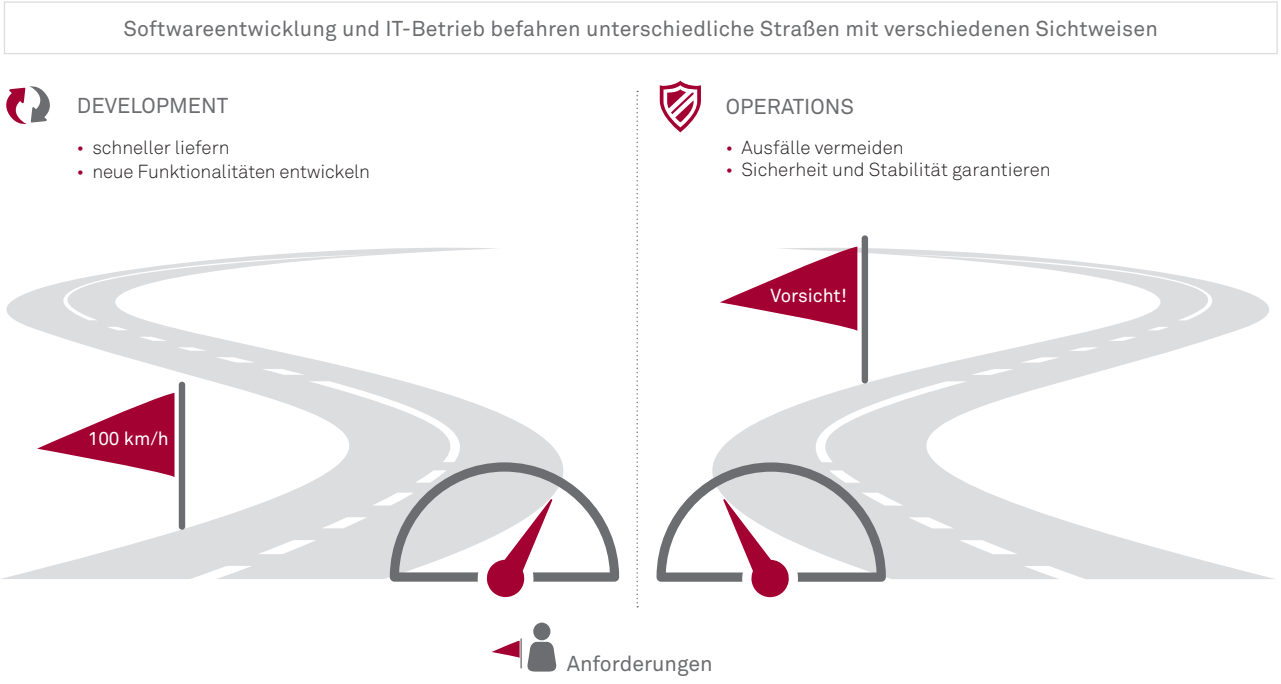


Abbildung 1: Der Interessenkonflikt zwischen Dev und Ops

ihnen die internen fachlichen Abläufe und Prozesse unbekannt sind. Gerade dieses Wissen ist aber ein Kernbestandteil in der Entwicklung, die mittels Exception und Log-Dateien arbeitet.

Der Betrieb muss demnach auch dieses Wissen erwerben, kennt aber zumeist nicht einmal die zugrunde liegenden Infrastrukturen (zum Beispiel die Java Virtual Machine), obwohl diese für die Analyse von Problemen sehr relevant sind. Im Gegenzug kennen

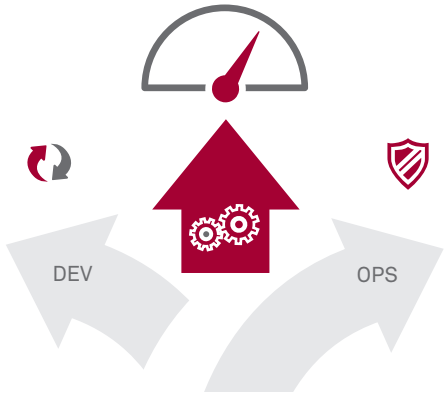


Abbildung 2: Um die Softwareauslieferung auf die Geschwindigkeit des Business zu beschleunigen, müssen die Ziele & Wege sich annähern

Entwickler die Werkzeuge des Betriebs nicht, da sie meist ihre eigenen Rechner/Server für die Entwicklung und den Testbetrieb verwalten und ihnen diese nicht zur Verfügung stehen.

Die Kombination von Wissen aus beiden Bereichen ist demnach für einen sinnvollen Betrieb von Anwendungen eigentlich notwendig. Aus Kundensicht ist dies sogar unerlässlich, denn für die Lösung eines Problems kann je nach Ursache das Wissen der einen oder der anderen Einheit beitragen.

Wie dies geht, zeigt der DevOps-Ansatz. Hier arbeiten Entwicklung und Betrieb in einem Team zusammen und sind jeweils für einen bestimmten fachlichen Service zuständig.

Der für Entwicklung und Betrieb benötigte Technologie-Stack wird eigenverantwortlich durch das Team gemeinsam betrieben und gewartet. Jedes Team kann somit diese Infrastruktur so weiterentwickeln, optimieren und betreiben, wie es sie benötigt. Abstimmungen bezüglich Erweiterungen des Technologie-Stacks, zum Beispiel im Monitoring, sind schnell realisierbar. Die Entscheidungen können eigenverantwortlich getroffen werden, denn sowohl der Betrieb als auch die Lösung von Problemen liegen in der Verantwortung desselben Teams.

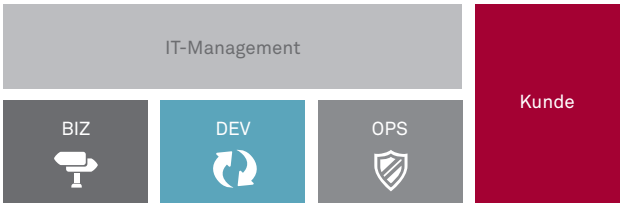


Abbildung 3: Klassische IT-Organisation

Der Kunde profitiert von einem dedizierten Ansprechpartner für einen fachlichen Service, der sowohl die Betriebssicht als auch die Anwendungssicht kennt. Unternehmensweite Standards beziehen sich nur noch auf die Hardware, die gegebenenfalls eine Cloud-Infrastruktur ist.

Übergreifend über die verschiedenen Teams arbeitet somit lediglich ein Basisbetrieb, der die Hardware und die Cloud-Infrastruktur wartet und bereitstellt. Die Installation/Konfiguration von Verfahren auf den virtuellen Maschinen erfolgt jeweils selbstständig durch die Teams.

Ein einfacher Anfang kann auch schon sein, die Zusammenarbeit der drei Organisationseinheiten, beispielsweise durch das Zusammenlegen der Räumlichkeiten oder Etablieren von Workshops zum Wissenstransfer zu forcieren.

CONTINUOUS INTEGRATION (UND TEST)

Continuous Integration ist eine Methode in der Softwareentwicklung. Sie fokussiert auf das kontinuierliche Integrieren von Codeänderungen in die Codebasis einer Softwareanwendung sowie auf die Validierung des Codes durch Unit-Tests und idealerweise auch Integration-Test-Level.

Eine kontinuierliche Integration bedeutet, dass isolierte Änderungen an der Codebasis sofort geprüft und anschließend zur Gesamtcodebasis einer Software hinzugefügt werden. Dies geschieht automatisch, wenn Entwickler ihren Code wie gewöhnlich mehrmals am Tag in einem gemeinsamen Repository ablegen.

Jeder Commit in der Versionsverwaltung führt dann zu einem automatisierten Build-Prozess. Entwickler erhalten über einen automatischen Fehlerreport oder Alarm unmittelbar Feedback über die Korrektheit und Konformität ihres Codes, sodass ein versehentlich integrierter Fehler schnellstmöglich identifiziert und korrigiert werden kann. Tools, die eine kontinuierliche Integration ermöglichen, bieten zumeist auch die Möglichkeit, Tests zu automatisieren und eine fortlaufende Dokumentation darüber zu erstellen. Dabei können die Tests Unit-Tests und Integrationstests, funktionale und nichtfunktionale Tests sowie Performance- und Security-Tests umfassen.

Eine einfache Variante der kontinuierlichen Integration ist zum Beispiel der Nightly Build, bei dem jeweils über Nacht alle Codeänderungen in einem Build integriert und automatisch einem Integrationstest unterzogen werden. Die Anwendung wird danach automatisch in einer Testumgebung verfügbar gemacht.

Ein Vorteil von automatisierten Tests ist, dass spezifische Prüfungen und Prüfbedingungen priorisiert werden können. So kann sichergestellt werden, dass entweder nur ausgewählte Testfälle oder aber das gesamte Test-Set mit jedem Build geprüft wird. Kontinuierliche Tests können somit als Erweiterung eines testgetriebenen Entwicklungsvorgehens (Test-driven Development, TDD) genutzt werden. Wesentliche Bestandteile von Continuous Integration sind in der Regel folgende:

Gemeinsame Codebasis

Es existiert eine gemeinsame Codebasis (Repository) mit einer Versionsverwaltung, in die alle Entwickler einer Arbeitsgruppe ihre Änderungen kontinuierlich integrieren können.

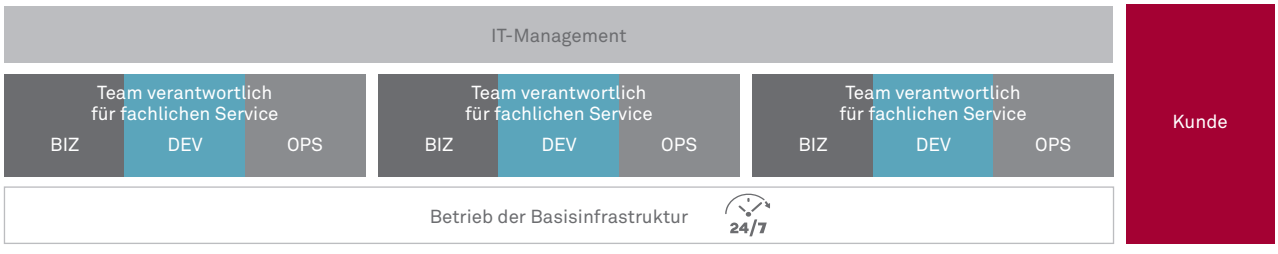


Abbildung 4: DevOps-Organisation

CONTINUOUS INTEGRATION

Ist eine technische Methode aus der Softwareentwicklung.

Ist auch ohne DevOps anwendbar.

Fokus: Automatische kontinuierliche Integration von Code in die Codebasis. Automatische Validierung der Codebasis mit Unit-/Integration-Tests.

Ergebnis: Applikation ist in einer Testumgebung verfügbar.

Automatisierte Tests

Jede Integration muss einheitlich definierte Tests und statische Codeüberprüfungen durchlaufen, bevor die Änderungen integriert werden. Hierfür ist ein automatisierter Build-Prozess notwendig. Idealerweise werden separate Testumgebungen genutzt, damit auf diesen Umgebungen auch gezielt Verfahren implementiert werden können, um die Testlaufzeit zu minimieren.

Kontinuierliche Testentwicklung

Jede Codeänderung sollte möglichst zeitgleich mit einem dazugehörigen Test entwickelt werden (zum Beispiel mittels TDD).

Häufige Integration

Entwickler sollten ihre Änderungen so oft wie möglich, mindestens einmal täglich, in die gemeinsame Codebasis integrieren. Kurze Integrationsintervalle reduzieren das Risiko fehlschlagender Integrationen und sichern gleichzeitig den Arbeitsfortschritt der Entwickler in der gemeinsamen Codebasis.

Integration in den Hauptzweig

Entwickler sollten ihre Änderungen so oft wie möglich in den Hauptzweig der Konfiguration des Produkts integrieren. Die Entwicklung in multiplen Zweigen der Hauptversion sollte möglichst minimiert werden, um die Komplexität minimal und die Abhängigkeiten überschaubar zu halten.

Kurze Testzyklen

Der Testzyklus vor der Integration sollte kurz gehalten sein, um häufige Integrationen zu fördern. Mit steigenden Qualitätsanforderungen für die einzelnen Integrationen steigt auch die Laufzeit zur Ausführung der Testzyklen. Die Menge der vor der Integration durchgeführten Tests muss sorgfältig abgewogen werden. Weniger wichtige Tests werden nach der Integration durchgeführt.

Gespiegelte Produktionsumgebung

Die Änderungen sollten in einem Abbild der realen Produktionsumgebung getestet werden.

Testdaten sollten regelmäßig aus der Produktionsumgebung in die Testumgebung eingespielt werden, um produktionsnahe Testszenarien simulieren zu können. Gegebenenfalls sind diese aus Datenschutzgründen zu anonymisieren.

Einfacher Zugriff

Auch Nicht-Entwickler brauchen einen einfachen Zugriff auf die Ergebnisse der Softwareentwicklung. Dies sind in der Regel nicht die Quellen, sondern können beispielsweise für Tester das in das Testsystem gespielte Produkt, für Qualitätsverantwortliche die Qualitätskennzahlen oder für den Release-Manager die Dokumentation oder eine fertig paketierte Auslieferung (Image) sein.

Automatisiertes Reporting

Die (Test-)Ergebnisse der Integrationen müssen leicht zugreifbar sein. Sowohl Entwickler als auch andere Beteiligte müssen einfach Informationen darüber bekommen können, wann die letzte erfolgreiche Integration ausgeführt wurde, welche Änderungen seit der letzten Lieferung eingebracht wurden und welche Qualität die Version hat.

Automatisiertes Deployment

Jeder Build sollte leicht in eine Produktionsumgebung (oder ein Abbild der selbigen) ausgeliefert (deployed) werden können. Hierfür sollte die Softwareverteilung grundsätzlich automatisiert erfolgen. Durch die Ausführung eines Skripts nach jedem erfolgreichen Build kann zum Beispiel die aktuelle Softwareversion automatisch auf einen Testserver deployed werden, sodass jeder die Integration überprüfen kann. Für ein voll integriertes Anwendungsszenario im Sinne eines Continuous Deployment wäre somit auch ein paralleles Deployment in mehrere Umgebungen (Test- und Produktionsumgebung) unter Einbindung weiterer automatisierter Testschritte möglich, die die Testabdeckung erhöhen oder die Qualität für einen Produktionseinsatz tiefergehend überprüfen.

Der Continuous-Integration-Prozess ist nach den Änderungen am Source Code und der Ausführung der Tests abgeschlossen und beginnt daraufhin wieder von vorn.

CONTINUOUS DELIVERY

Continuous Delivery schließt sich an die Continuous Integration an und erweitert den Integrationszyklus bis in die Produktion. Erst wenn die Anwendung in die Produktion ausgeliefert und installiert wurde und dem Kunden zur Verfügung steht, ist die Lieferkette von der Entwicklung zum Kunden abgeschlossen. Continuous Delivery wird daher auch häufig als „finale Stufe“ oder „letzte Meile“ von Continuous Integration bezeichnet.

Continuous Delivery geht schon auf das agile Manifest zurück. Dort heißt es im ersten Prinzip: „Unsere höchste Priorität ist es, den Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufrieden zu stellen.“[3] Continuous Delivery verfolgt dieses Ziel durch Fortführen agiler Entwicklungspraktiken bis in die Produktion. Continuous Delivery ist eine Sammlung von Techniken, Prozessen und Werkzeugen, die den Auslieferungsprozess der Software automatisieren und verbessern. Dieser Auslieferungsprozess lässt sich sowohl unter zeitlichen (time-to-market) als auch unter qualitativen Aspekten (Automatisierung, wiederholbare und zuverlässige Prozesse) verbessern.

CONTINUOUS DELIVERY

Ist auch ohne DevOps anwendbar. Erweitert den Feedbackzyklus von CI bis in die Produktion. Ist im agilen Manifest begründet. Sammlung von Techniken, Prozessen und Werkzeugen, die den Prozess der Softwareauslieferung verbessern.
Applikation ist in der Pre-Prod-Umgebung verfügbar. CDel = fordert ein kontinuierliches Bereitstellen von SW-Paketen in der Pre-Prod, das Deployment ist manuell; CDepl = fordert ein kontinuierliches Deployment in die Prod.

Continuous Delivery beruht auf acht, durch J. Humble und D. Farley 2010 ausgearbeitete Prinzipien[4], die als konkrete Leitsätze oder Empfehlungen formuliert sind:[5]

1. Der Prozess der Software Bereitstellung/-Release muss wiederholbar und zuverlässig sein.

2. Automatisieren Sie alles! Ein manuelles Deployment kann niemals als wiederholbar und zuverlässig beschrieben werden. Ein ernsthaftes Investment in die Automatisierung aller Aufgaben, die Sie wiederholt durchführen, führt zwingend zu einer erhöhten Zuverlässigkeit.

3. Wenn es schwierig oder schmerzhaft ist, tun Sie es öfter. Dies scheint zunächst widersprüchlich, führt allerdings zu einem automatischen, kontinuierlichen Verbesserungsprozess. Je öfter Sie Hürden nehmen müssen, umso wahrscheinlicher ist es, dass Sie beginnen werden, den Prozess zu vereinfachen und zu automatisieren, damit er zukünftig einfacher und wiederholbarer wird.

4. Pflegen und managen Sie alles in der Quellcodeverwaltung.

5. Fertig bedeutet „released“. Dieses Prinzip trägt die Verantwortung weit über den eigenen Bereich und die Aufgabe hinaus. Die Verantwortung eines Entwicklers endet nicht mit dem Einchecken des Codes in das Repository, sondern erst, wenn sichergestellt ist, dass der Code in der Produktion fehlerfrei läuft und das Release-Monitoring dies bestätigt.

6. Bauen Sie Qualität ein! Berücksichtigen Sie den Qualitätsaspekt umfassend in den Metriken und investieren Sie ausreichend Zeit hierfür. Erst das Messen der Qualität in allen Phasen ermöglicht es, einen kontrollierten Prozess zu etablieren, bei dem die Qualität an verschiedenen Stellen im Entwicklungsprozess gesteuert verbessert werden kann. Eine verbesserte Qualität wiederum führt zu einfacherer Wartung und zu einer langfristigen Kostenreduktion.

7. Jeder hat die Verantwortung für den Release-Prozess. Nur die für den Bürger (den Endkunden) verfügbaren Dienstleistungen (und Produkte) bestimmen die Wahrnehmung und Bewertung der dienstleistenden Behörde. Im Falle von Software ist das also die Software, die produktiv (released) ist. Aus diesem Grund sollten auch alle gemeinsam für den Release-Prozess die Verantwortung tragen. Jede Aufgabe sollte deshalb immer die Effizienz und die Qualität des Release-Prozesses als Ziel berücksichtigen, damit neben der originären Aufgabe frühzeitig auch die Bereitstellung für den Kunden berücksichtigt wird.

8. Verbessern Sie kontinuierlich. Eine kontinuierliche Verbesserung ist eine ständige Anpassung der Prozesses und Verfahren an sich verändernde Rahmenbedingungen. Jede Verbesserung führt wiederum zu mehr Effektivität und Effizienz und ermöglicht es wiederum, im Falle von neuen Rahmenbedingungen schneller darauf zu reagieren.

Neben diesen Grundprinzipien existiert eine Fülle von weiteren Handlungsempfehlungen, die sich bewährt haben. Im Rahmen dieses Fachartikels ist es allerdings nicht möglich, diese vollständig aufzuzählen, weshalb nur ein kleiner Ausschnitt präsentiert wird.

CONTINUOUS DEPLOYMENT

Beschreibt das automatisierte Deployment in Produktion. Ist die logische Konsequenz aus Continuous Delivery. Ist auch ohne DevOps anwendbar.
Nutzt Methoden wie

- **Feature Toggles,**
- **Canary Releases,**
- **Blue-Green Deployments.**

3 http://agilemanifesto.org/iso/de/principles.html
4 J. Humble, D. Farley, 2010: Continuous Delivery: Reliable Software Releases Through Build, Addison-Wesley
5 https://dzone.com/articles/8-principles-continuous

CONTINUOUS DEPLOYMENT

Continuous Deployment ist der letzte logische Schritt einer kontinuierlichen Delivery-Pipeline und eine Fortführung des Continuous-Delivery-Gedankens. Was ist nun der Unterschied zwischen den beiden?

Während man bei Continuous Delivery festlegt, wann man mit einer Softwareversion in Produktion geht, also eine bewusste Entscheidung fällt und somit die Wahl hat, resultiert bei Continuous Deployment jeder erfolgreiche Build aus einem automatisierten Deployment in Produktion.

Spätestens jetzt ist eindeutig, wie tiefgreifend der Ansatz einer kontinuierlichen Delivery-Pipeline ist, wie wichtig die Bestandteile und deren Umsetzung sind und wie groß die Auswirkungen sowohl im Unternehmen als auch für den Endnutzer sein können. Deutlich ist dann auch die Motivation, eine kontinuierliche Delivery-Pipeline zu implementieren: Dies geschieht in der Regel, um eine schnellere Time-to-market zu realisieren. Diese sollte dementsprechend auch fachlich oder technisch sinnvoll sein, und der Endkunde sollte sie auch sichtbar wahrnehmen können. Gerade in dieser letzten Phase steckt also das größte Risiko, da ein automatisches Deployment eine unmittelbare Kundenauswirkung hat. Gleichzeitig wird erst hier einer der großen Mehrwerte einer kontinuierlichen Delivery-Pipeline realisiert.

Wie lässt sich ein Continuous Deployment also umsetzen, ohne dass die Stabilität durch (zu) häufige automatische Releases negativ beeinflusst wird? Hier greift die Continuous-Delivery-Forderung: „Fail fast, fail often."

Provokant gesagt, ist es ein explizites Ziel von DevOps, viele Fehler zu machen, denn Fehler sind erwünscht. Jeder Fehler, der frühzeitig entdeckt und korrigiert wird, kann zur kontinuierlichen Verbesserung der automatisierten Delivery-Pipeline genutzt werden. Dies setzt wiederum einen ständigen Feedbackprozess in der DevOps-Organisation voraus, der durch Kollaboration der Organisationseinheiten getragen wird.

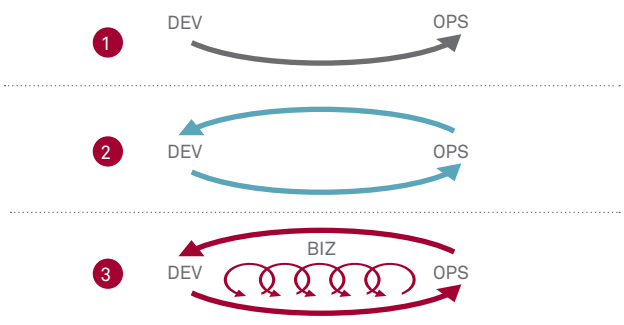


Abbildung 5: DevOps setzt auf kontinuierliche Kollaboration und Feedbacks

„Learn fast, learn often" ist also, konkreter formuliert, die Forderung von Continuous Delivery, die es überhaupt ermöglicht, schnell, wiederholbar, automatisiert und auch qualitativ hochwertig Softwarereleases zu deployen.

Als Erfinder der Glühlampe ging der amerikanische Wissenschaftler und Autodidakt Thomas Alva Edison in die Geschichte ein. Allerdings: Rund 2.000 Anläufe brauchte Edison, bis er den ersten Kohlefaden in einer Lampe zum Leuchten bringen konnte. Ein Ergebnis, das den Amerikaner jedoch wenig schockte, denn trocken kommentierte Thomas Edison seine Fehlversuche mit dem Satz: „Ein Misserfolg war es nicht. Denn wenigstens kennt man jetzt 2.000 Arten, wie ein Kohlefaden nicht zum Leuchten gebracht werden kann."

Softwareversionen, die also fehlerbehaftet sind, dürfen demnach gar nicht erst durch den Automatismus in Produktion kommen, sondern müssen zwingend vorher in der Pipeline scheitern. Die Automatisierung der Prozesse ermöglicht es, schnell Softwarepakete in Produktion zu bringen, während die kontinuierliche Kollaboration sicherstellt, dass nur solche in Produktion gelangen, die qualitativ hochwertig und fehlerfrei sind. Bei Continuous Deployment spielt daher das Risikomanagement eine sehr wichtige Rolle.

Produktionsausfälle durch ein neues Release lassen sich trotzdem nicht 100-prozentig ausschließen, da meist Test- und Produktionsumgebung unterschiedlich sind. Kommt es also zu einem Produktionsausfall, ist es notwendig, schnell auf die alte Version zurückzufallen oder aber einen nennenswerten Ausfall der Produktion zu verhindern.

Traditionell wird der Roll-Back-Prozess (das Zurückspielen der Vorgängerversion) als Maßnahme bei Eintritt des Risikos einer nicht funktionierenden Produktivversion genutzt. Ein Roll-back bedeutet, dass die Anwendung in Produktion nicht zur Verfügung steht, weshalb es umso wichtiger ist, dass der Roll-back-Prozess funktionieren muss. Hierzu muss er also in Vorfeld ausführlich und immer wieder getestet werden und eine alte Version auch immer zur Verfügung stehen, um für den Notfall gewappnet zu sein.

Continuous Deployment kennt allerdings auch weitere Mechanismen zur Risikoreduktion beim Deployment in Produktion:

Roll-forward

Eine Alternative zum Roll-back ist der Roll-forward-/Patch-forward-Prozess. Dabei wird bei einem Fehler eine neue Version der Software deployed, die den Fehler korrigiert. Auch diese Version

muss natürlich getestet werden, was allerdings in einer Continuous-Delivery-Pipeline keinen großen Aufwand darstellt, aber das Vertrauen voraussetzt, dass mit der Pipeline Änderungen schnell genug ausgeliefert werden können, um den Fehler zu beseitigen. Der Aufwand eines Roll-forward ist genauso groß wie bei einem Roll-back, aber zumeist weniger komplex. Dies liegt daran, dass Änderungen an Datenbanken bei einem Rollback oft schwierig wieder rückgängig zu machen sind. Bei einem Roll-forward allerdings können die Datenbankänderungen oft erhalten bleiben, was den Prozess erheblich vereinfacht.

Feature Toggle

Feature Toggles sind eine Funktionalität, bei der bestimmte Features mittels eines Schalters aktiviert oder deaktiviert werden können. Auf diese Weise lassen sich Features so implementieren, dass die Software mit den Features in Produktion gebracht werden kann, ohne dass diese sofort aktiviert sind. Voraussetzung ist, dass diese Features in sich abgeschlossen entwickelt werden können.

Durch Feature Toggles wird die Implementierung vom Deployment entkoppelt.

Features lassen sich bereits in Produktion testen, wenn diese zum Beispiel nur für bestimmte Nutzer aktiviert werden. Oder man kann diese nur für bestimmte Kundengruppen aktivieren, um zunächst das Feedback der Nutzer zu erhalten. Es gibt verschiedene Arten von Feature Toggles, wie zum Beispiel:

Release Toggles

Release Toggles dienen dazu, die Aktivierung eines Features von dem Release-Termin der Codeänderungen für dieses Feature zu entkoppeln. Zunächst wird die Software deployed und das Feature deaktiviert. Wenn das Feature tatsächlich fertig und getestet ist, wird der Toggle aktiviert.

Geschäftliche Toggles

Geschäftliche Toggles dienen dazu, Features gezielt nur durch bestimmten Kundengruppen anzubieten.

Betriebliche Toggles

Betriebliche Toggles dienen dazu, Features zu deaktivieren, um den Ausfall der gesamten Anwendung zu vermeiden.

Blue-/Green-Deployment

Blue-/Green-Deployments sind dadurch charakterisiert, dass es zwei parallele Produktionsumgebungen gibt. Ein Router steuert die aktuelle, für den Endnutzer gültige Umgebung an. Wird eine neue Softwareversion deployed, so wird diese zum Beispiel in die Umgebung Blue deployed, während für die Endnutzer nach wie vor die Green-Umgebung genutzt wird. Durch eine Umkonfiguration des Routers kann die neue Softwareversion für die Endnutzer verfügbar gemacht werden, sobald sie, zum Beispiel nach abschließenden Tests oder der Migration von Daten, freigegeben wurde.

Ein Vorteil von Blue-/Green-Deployments ist, dass das neue Release so neben dem alten Release betrieben und später dann gegebenenfalls umgeschaltet werden kann. Zusätzlich erfolgt durch den Produktionseinsatz keine Downtime, da die Anwendung durchgehend verfügbar ist. Ein weiterer Vorteil ist, dass die neue Softwareversion in einer Produktionsumgebung ausgiebig, auch in Bezug auf die Performance, getestet werden kann, bevor sie „live“ geschaltet wird.

Canary Release

Dieser Ansatz baut auf dem von Blue-/Green-Deployments auf und ergänzt ihn um eine stufenweise Lasterhöhung. Bei Canary Releases wird ein neues Softwarerelease zunächst nur auf einigen Servern im Cluster ausgerollt, bevor die Software auf allen Rechnern deployed wird.

Auch hier ist es möglich, zuerst die Endnutzer von der Nutzung des neuen Releases auszuschließen und das Release initial einer geschlossenen Benutzergruppen wie beispielsweise Mitarbeitern oder Testern verfügbar zu machen. Der Roll-out an die Endnutzer erfolgt dann aber im Gegensatz zu den Blue-/Green-Deployments nach erfolgreichem Test nicht auf Knopfdruck (0-100 Prozent), sondern sukzessiv. Hierzu wird das neue Release auf einer steigenden Anzahl von Servern im Cluster ausgerollt und mit der Lasterhöhung einer stetig steigenden Endnutzerzahl (zum Beispiel 10-20 Prozent) zur Verfügung gestellt. Sollte es dann zu Problemen kommen, ist nach wie vor ein schneller Wechsel auf die alte Releaseversion möglich.

Vorteile von Continuous Deployment sind demnach:

- Jede Änderung geht direkt in Produktion.
- Das Feedback erfolgt noch schneller als nur durch Continuous Delivery.
- Durch den Einsatz von Feature Toggles ist es möglich, in die Produktion zu deployen, ohne die neuen Funktionalitäten sofort zu nutzen.
- Es ist möglich, eine kleine Nutzergruppe neue Funktionalität in Produktion testen zu lassen, um so ein frühes Feedback zu bekommen.

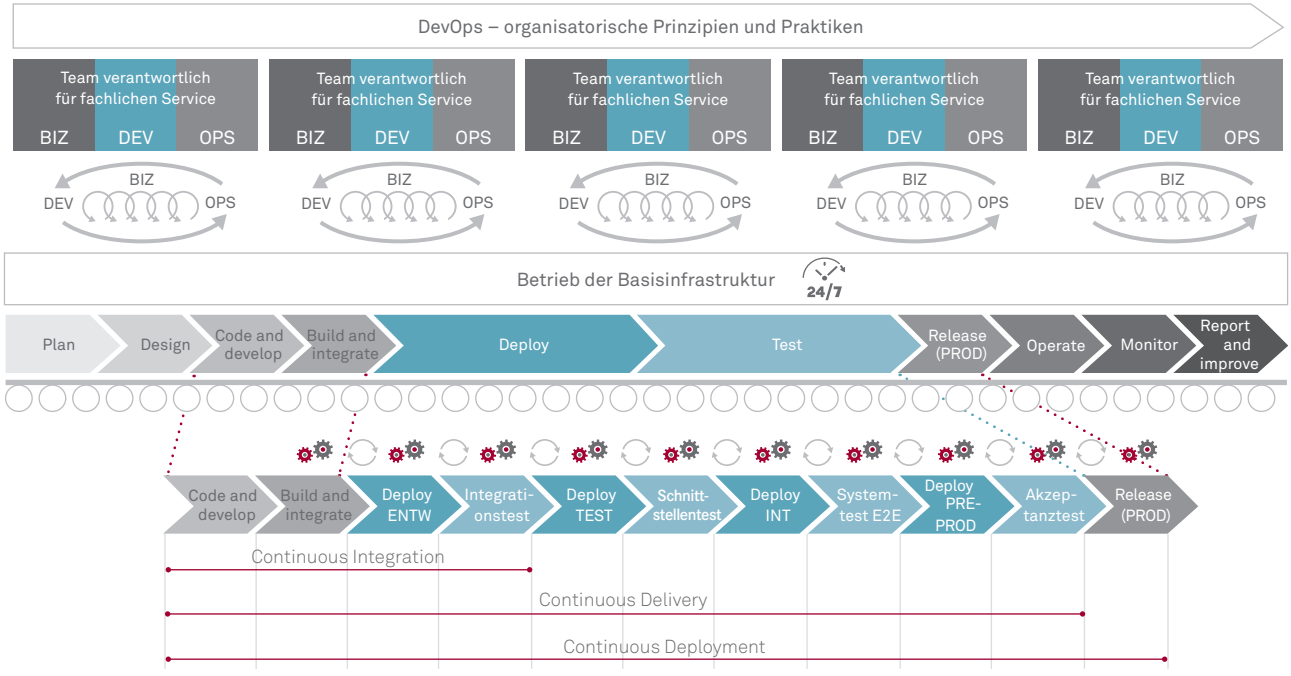


Abbildung 6: Continuous Delivery Pipeline

BAUSTEINE EINER CONTINUOUS DELIVERY PIPELINE

Der Aufbau einer kontinuierlichen Delivery-Pipeline hat viele Aspekte, die es zu beachten gilt. Letztendlich lassen sich die Aktivitäten in drei Bereiche unterteilen:

Organisation/Personen

Definition und Implementierung eines Organisationsmodells, das kurze Entscheidungswege, uneingeschränkte Kollaboration und Eigenverantwortung in Entwicklung, Betrieb und Wartung fordert und fördert sowie Betriebs- und Anwendungswissen in integrierten Teams für einen fachlichen Service bündelt.

Prozesse

Automatisierung von Tests, Deployments, Bereitstellung von Infrastrukturen (zum Beispiel Umgebungen), Prozessen und Schnittstellen über die gesamte Delivery-Pipeline.

Technologie

Auswahl und Konfiguration von Tools und einer Architektur, die die Prozesse und die Organisationsstruktur optimal unterstützen.

Zusammenfassend stellen wir alle Bausteine einer Continuous Delivery Pipeline in einer Übersicht dar und ordnen sie sowohl zeitlich als auch inhaltlich ein.

AUSBLICK

Im nächsten Teil der Artikelreihe lesen Sie:

- Wie kann man den Continuous-Delivery-Reifegrad ermitteln?
- Welche KPIs können genutzt werden, um Erfolge messbar zu machen?
- Wie lässt sich der Return-on-Investment eines DevOps-/Continuous-Delivery-Projekts ermitteln? ●

ANSPRECHPARTNER – DR. ANDREAS ZAMPERONI
Leiter Center of Competence
Projektmanagement
Public Sector Solutions Consulting