Gemeinsam: schneller, besser, sicherer!

Kernbestandteile einer kontinuierlichen Delivery Pipeline

von Erik Benedetto

Nachdem in der NEWS 01/2017 der Konflikt zwischen Softwareentwicklung und Betrieb sowie Lösungsmöglichkeiten dieses relevanten Themas untersucht wurden, legt dieser Artikel den Fokus auf die Begriffe DevOps, Continuous Integration, Continuous Delivery und Continuous Deployment und die Bausteine einer kontinuierlichen Delivery Pipeline.

DevOps, Continuous Integration, Continuous Delivery und Continuous Deployment

Entwicklung und Betrieb werden in der Regel als eigenständige Bereiche getrennt betrachtet; als Gründe werden Arbeitsteilung, Spezialisierung sowie höhere Effizienz durch Standardisierung und Industrialisierung genannt. Das gemeinsame Ziel, einen möglichst guten Service für interne oder externe Kunden anzubieten, gerät dadurch leicht in Vergessenheit. Dennoch ist diese Trennung in einzelne Organisationseinheiten faktisch eine große Gemeinsamkeit heutiger IT-Organisationen (siehe Abbildung 1), die in der Regel mit einer unterschiedlichen Perspektive auf die Anwendungen begründet wird. Die Kombination von Wissen aus Entwicklung und Betrieb ist aber für einen sinnvollen Anwendungsbetrieb notwendig und aus Kundensicht sogar unerlässlich. Denn für die Lösung eines Problems kann, je nach Ursache, das Wissen der einen oder anderen Einheit beitragen.

Dass die Kollaboration von Entwicklung (Dev) und Betrieb (Ops) möglich ist, zeigt der DevOps-Ansatz. Hier arbeiten Entwicklung und Betrieb gemeinsam in einem Team und sind jeweils für einen bestimmten fachlichen Service zuständig (siehe Abbildung 2). Der für Entwicklung und Betrieb benötigte Technologiemix, also der „Technologie-Stack“, wird eigenverantwortlich durch das gesamte Team betrieben und gewartet. Jedes Team kann so den Service weiterentwickeln, optimieren und betreiben, wie es ihn benötigt. Da sowohl der Betrieb (Entwicklung/Produktion) als auch die Lösung von Problemen in der Verantwortung desselben Teams liegt, können Abstimmungen zu Erweiterungen des Technologie-Stacks, zum Beispiel im Monitoring, schnell realisiert und die Entscheidungen eigenverantwortlich getroffen werden. Der Kunde profitiert von einem dedizierten Ansprechpartner für einen fachlichen Service, der sowohl die Betriebs- als auch die Anwendungssicht kennt. Unternehmensweite Standards beziehen sich nur noch auf die Hardware und eine Cloud-Infrastruktur. Übergreifend über die verschiedenen

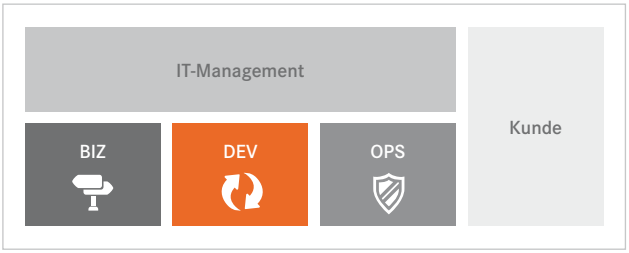

Abbildung 1: Klassische IT-Organisation

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

Umbau der Organisation nicht zwingend erforderlich

Auch wenn es auf den ersten Blick so erscheint, ist es kein Muss, die Organisation für die Einführung von DevOps umfassend zu verändern. Zum Einstieg ist es sinnvoll, zunächst einen fachlichen Service nach dem DevOps-Gedanken durch ein gemeinsames Team aus Fachbereich, Betrieb und Entwicklung zu etablieren und weiterzuentwickeln. Noch einfacher wird der Einstieg, wenn im ersten Schritt die Kollaboration der drei Organisationseinheiten Fachbereich, Betrieb und Entwicklung forciert wird, zum Beispiel durch Zusammenlegung der Räumlichkeiten, Durchführung von Workshops zum Wissenstransfer etc. Nicht sinnvoll ist es dagegen, ein DevOps-Team neben Entwicklung und Betrieb zu installieren. Dies wäre ein weiteres Silo, was dem DevOps-Gedanken widerspricht.

Continuous Integration (und Test)

Continuous Integration ist eine Methode in der Softwareentwicklung, die auf eine kontinuierliche Integration von Codeänderungen in die Codebasis einer Softwareanwendung, die Validierung des Codes durch Unit-Tests und eventuell Integrationstests fokussiert. Kontinuierliche Integration bedeutet, dass isolierte Änderungen an der Codebasis sofort geprüft und anschließend zur Gesamtcodebasis einer Software hinzugefügt werden. Das heißt, Entwickler integrieren ihren Code mehrmals am Tag in ein gemeinsames Repository und jeder Commit in der Versionsverwaltung führt zu einem automatisierten Build-Prozess. Über einen automatischen Fehlerreport oder Alarm erhalten sie ein unmittelbares Feedback, sodass ein versehentlich integrierter Fehler schnellstmöglich identifiziert und korrigiert werden kann. Mit Tools, die eine kontinuierliche Integration ermöglichen, können meist auch Tests automatisiert und eine fortlaufende Dokumentation erstellt werden. Die Tests können sowohl Unit- und/oder Integrationstests, funktionale und/oder nicht funktionale Tests als auch Performance- und/oder 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 Applikation ist danach in einer Testumgebung verfügbar. 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 Testset mit jedem Build geprüft wird. Kontinuierliche Tests können somit als Erweiterung der testgetriebenen Entwicklungspraktik (TDD) genutzt werden. Wesentliche Bestandteile von Continuous Integration sind in der Regel folgende:

Gemeinsame Codebasis: In eine gemeinsame Codebasis (Repository) mit einer Versionsverwaltung können alle Entwickler einer Arbeitsgruppe ihre Änderungen kontinuierlich integrieren.

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 darauf auch gezielt Verfahren implementiert werden können, um die Testlaufzeit zu minimieren.

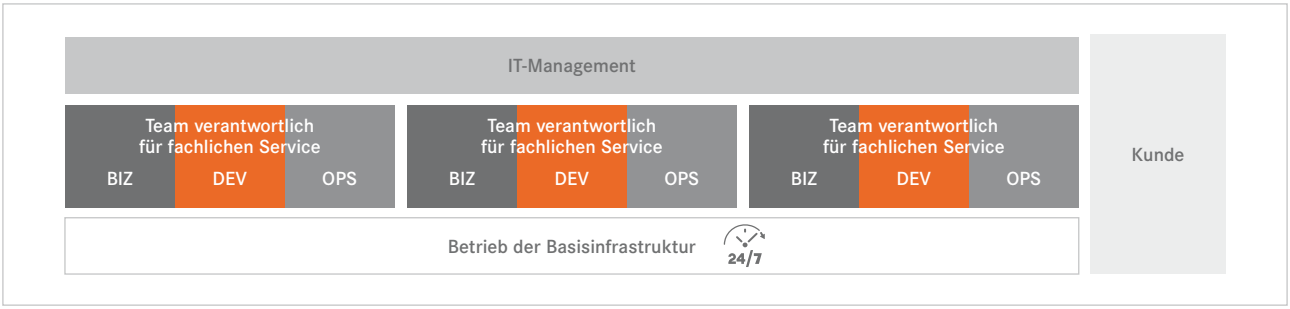

Abbildung 2: DevOps-Organisation

Kontinuierliche Testentwicklung: Jede Codeänderung sollte möglichst zeitgleich mit einem dazugehörigen Test entwickelt werden (zum Beispiel mittels testgetriebener Entwicklung TDD).

Häufige Integration: Entwickler sollten ihre Änderungen so oft wie möglich, aber 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 Hauptbranch: Entwickler sollten ihre Änderungen so oft wie möglich in den Hauptbranch des Produktes integrieren. Die Entwicklung in multiplen Zweigen der Hauptversion sollte minimiert werden, um die Komplexität gering und die Abhängigkeiten überschaubar zu halten.

Kurze Testzyklen: Ein kurz gehaltener Testzyklus vor der Integration fördert häufige Integrationen. 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, weniger wichtige Tests nach der Integration durchgeführt werden.

Gespiegelte Produktionsumgebung: Die Änderungen sollten in einem Abbild der realen Produktionsumgebung getestet werden. Wenn Testdaten regelmäßig aus der Produktionsumgebung in die Testumgebung eingespielt werden, können produktionsnahe Testszenarien simuliert werden (aus Datenschutzgründen eventuell anonymisiert).

Einfacher Zugriff: Auch Nichtentwickler brauchen einen einfachen Zugriff auf die „Ergebnisse" der Softwareentwicklung, nicht notwendigerweise auf die Quellen, aber beispielsweise auf das in das Testsystem gespielte Produkt für Tester, die Qualitätszahlen für Qualitätsverantwortliche, die Dokumentation oder ein fertig paketiertes Abbild für Release Manager.

Automatisiertes Reporting: Die (Test-)Ergebnisse der Integrationen müssen sowohl für Entwickler als auch andere Beteiligte einfach abrufbar sein: wann die letzte erfolgreiche Integration ausgeführt, welche Änderungen seit der letzten Lieferung eingebracht wurden und welche Qualität die Version hat.

Automatisiertes Deployment: Durch eine grundsätzlich automatisierte Softwareverteilung kann jeder Build leicht in eine Produktionsumgebung (oder ein Abbild derselben) deployed werden. Für ein voll integriertes Anwendungsszenario im Sinne eines Continuous Deployment wäre auch ein paralleles Deployment in mehrere Umgebungen möglich (Test- und Produktionsumgebung) – unter Einbindung weiterer automatisierter Testschritte, die die Testabdeckung erhöhen oder die Qualität für einen Produktionseinsatz tiefergehend überprüfen.

Continuous Delivery

Während der Continuous-Integration-Prozess nach den Änderungen am Source Code und der Ausführung der Tests abgeschlossen ist, setzt Continuous Delivery an genau dieser Stelle an und erweitert den Feedbackzyklus bis in die Produktion. Erst wenn die Anwendung in die Produktion deployed wurde und dem Kunden zur Verfügung steht, ist die „Definition of Done" erfüllt. Vor diesem Hintergrund wird Continuous Delivery auch als finale Stufe oder „letzte Meile" von Continuous Integration bezeichnet.

Das originäre Ziel und die höchste Priorität von Continuous Delivery ist – nach dem ersten Prinzip des agilen Manifests –, „den Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufriedenzustellen". Continuous Delivery adressiert dieses Ziel, indem es agile Entwicklungspraktiken bis in die Produktion fortführt. Dabei ist es im Grunde nicht mehr als eine Sammlung von Techniken, Prozessen und Werkzeugen, die den Prozess der Softwareauslieferung verbessern. Der Schwerpunkt liegt auf den Werkzeugen und dem Auslieferungsprozess, der sich wiederum unter zeitlichen (Time-to Market) und qualitativen Aspekten (Automatisierung, wiederholbare und zuverlässige Prozesse) differenzieren lässt. Continuous Delivery beruht auf acht Prinzipien, die als konkrete Leitsätze oder Empfehlungen formuliert sind:

1. Der Prozess der Softwarebereitstellung/Release muss wiederholbar und zuverlässig sein. Die Automatisierung manueller Schritte hilft, wiederholbare, zuverlässige Prozesse zu etablieren.

2. Automatisieren Sie alles! Denn 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. Denn je öfter Sie Hürden nehmen müssen, umso wahrscheinlicher ist es, dass Sie beginnen, den Prozess zu vereinfachen und zu automatisieren, so dass er zukünftig einfacher und wiederholbarer wird.

4. Pflegen und managen Sie alles in der Quellcodeverwaltung. Code, Konfigurationen, Skripte, Datenbanken, Dokumentationen, alles: Mit einer vertrauenswürdigen und zuverlässigen Quelle als Basis für Ihre Informationen haben Sie eine stabile Grundlage, um Ihre Prozesse aufzubauen.

5. Fertig bedeutet „released". Das heißt, dass die Verantwortung grundsätzlich weit über den eigenen Bereich und die Aufgabe hinausgeht. 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 dafür ausreichend Zeit. Qualität in allen Phasen messbar machen führt zu einem steuerbaren Prozess, mit dem Qualität verbessert werden kann. Eine verbesserte Qualität wiederum führt zu einer einfacheren Wartung und langfristigen Kostenreduktion.

7. Jeder hat die Verantwortung für den Release-Prozess. Unternehmen verdienen nur dann Geld, wenn die entwickelten Produkte auch für den Endkunden verfügbar, im Falle von Software, also „released" sind. Daher sollten alle gemeinsam Verantwortung tragen. Jede Aufgabe sollte den Release-Prozess als ein 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 Prozesse und Verfahren an sich verändernde Rahmenbedingungen. Jede Verbesserung führt wiederum zu mehr Effektivität und Effizienz und ermöglicht es, schneller auf Veränderungen zu reagieren.

Neben diesen Grundprinzipien existieren eine Fülle weiterer Handlungsempfehlungen, die sich ebenfalls bewährt haben. Sie vollständig aufzuführen würde allerdings den Rahmen dieses Artikels weit übersteigen.

Continuous Deployment

Continuous Deployment ist der letzte logische Schritt einer kontinuierlichen Delivery Pipeline und eine Fortführung des Continuous-Delivery-Gedankens. Doch während bei Continuous Delivery bewusst entschieden und festlegt wird, wann man mit einer Softwareversion in Produktion geht, resultiert bei Continuous Deployment jeder erfolgreiche Build in einem automatisierten Deployment in Produktion. Eine bewusste Entscheidung wird nicht (mehr) getroffen.

Spätestens jetzt ist eindeutig, wie tief greifend 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. Klar ist auch, dass man eine kontinuierliche Delivery Pipeline unter anderem deshalb implementiert, um eine schnellere Time-to-Market zu realisieren, die der Endkunde auch sichtbar wahrnehmen kann. Gerade in dieser letzten Phase steckt also das größte Risiko, da sie eine Kundenauswirkung hat. Gleichzeitig wird erst hier der Mehrwert einer kontinuierlichen Delivery Pipeline realisiert. Die Frage ist, wie sich Continuous Deployment umsetzen lässt, ohne die Stabilität durch häufige Releases zu beeinflussen? Eine DevOps-Forderung lautet „fail fast, fail often“. Das heißt, viele Fehler zu machen, ist ein explizites Ziel von DevOps, denn 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. Eine weitere DevOps-Forderung lautet „learn fast, learn often“. Sie ermöglicht, schnell, wiederholbar, automatisiert und auch qualitativ hochwertig Softwarereleases zu deployen. Denn Softwareversionen, die fehlerbehaftet sind, dürften gar nicht durch den Automatismus in Produktion kommen, sondern müssten vorher zwingend in der Pipeline scheitern.

Der amerikanische Wissenschaftler und Autodidakt Thomas Alva Edison ging als Erfinder der Glühlampe in die Geschichte ein. Allerdings brauchte er rund 2.000 Anläufe, bis er den ersten Kohlefaden in einer Lampe zum Leuchten bringen konnte. Trocken kommentierte Thomas Edision seine Fehlversuche so: „Ein Misserfolg war es nicht. Denn wenigstens kennt man jetzt 2.000 Arten, wie ein Kohlefaden nicht zum Leuchten gebracht werden kann."

Die Automatisierung der Prozesse ermöglicht es also, Softwarepakete schnell in Produktion zu bringen, während die kontinuierliche Kollaboration sicherstellt, dass nur solche in Produktion gelangen, die qualitativ hochwertig und fehlerfrei sind. Bei Continuous Deployment geht es also vor allem um Risikomanagement.

Produktionsausfälle durch ein neues Release lassen sich allerdings nicht zu 100 Prozent ausschließen, da Test- und Produktionsumgebung meist unterschiedlich sind. Kommt es zu einem Produktionsausfall, ist es wichtig, schnell auf die alte Version zurückfallen (Roll-Back), also einen langen Ausfall der Produktion verhindern zu können. Das bedeutet aber, dass die Anwendung in Produktion nicht zur Verfügung steht. Ein funktionierender Roll-Back-Prozess ist daher absolut notwendig. Um für den Notfall gewappnet zu sein, muss er im Vorfeld getestet werden und eine frühere Version immer zur Verfügung stehen. Continuous Deployment kennt noch weitere Mechanismen zur Risikoreduktion beim Deployment in Produktion:

Roll-Forward- oder Patch-Forward-Prozess bedeutet, dass bei einem Fehler eine neue Version der Software deployed wird, die den Fehler korrigiert. Diese Version muss ebenfalls getestet werden, setzt aber das Vertrauen in die Delivery Pipeline und

ihre Schnelligkeit voraus. Der Aufwand eines Roll-Forwards entspricht dem eines Roll-Backs, ist aber meist weniger komplex, vor allem deshalb, weil Änderungen an Datenbanken bei einem Roll-Back schwierig rückgängig zu machen sind, während beim Roll-Forward die Datenbankänderungen oft erhalten bleiben können.

Feature Toggle ist eine Funktionalität, bei der bestimmte Features mittels eines Schalters aktiviert oder deaktiviert werden können. Features lassen sich so implementieren und der Code mit dem Feature in Produktion bringen, ohne dass diese aktiviert sind. Voraussetzung ist, dass die Features jeweils separat entwickelt werden. Durch Feature Toggles wird die Implementierung vom Deployment entkoppelt. Features lassen sich bereits in Produktion testen, wenn sie zum Beispiel für bestimmte Nutzer oder Kundengruppen aktiviert werden, um deren Feedback zu erhalten. Auswirkungen und Metriken, die durch den Feature-Einsatz erzielt werden sollen, lassen sich so plausibilisieren (A/B-Testing). Es gibt verschiedene Arten von Feature Toggles, die hier exemplarisch aufgelistet sind:

- Release Toggles dienen dazu, die Aktivierung eines Features von dem Release-Termin der Codeänderungen für dieses Feature zu entkoppeln. Zunächst wird der Code deployed und das Feature deaktiviert. Wenn das Feature tatsächlich fertig und getestet ist, wird der Toggle aktiviert.
- Geschäftliche Toggles dienen dazu, Features für Kunden auszuprobieren oder gezielt nur bestimmten Kundengruppen anzubieten.
- Betriebliche Toggles dienen dazu, Features zu deaktivieren, um den Ausfall der gesamten Anwendung zu vermeiden.

Blue/Green-Deployment ist dadurch charakterisiert, dass es zwei parallele Produktionsumgebungen gibt. Ein Router steuert die entsprechende Umgebung für den Endnutzer 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 letztendlich verfügbar gemacht werden. Ein Vorteil davon ist, dass das neue Release neben dem alten Release betrieben und später gegebenenfalls umgeschaltet werden kann. Zusätzlich erfolgt durch den Produktionseinsatz keine Downtime, da die Anwendung durchgehend verfügbar ist. Ein weiterer Vorteil: Die neue Softwareversion kann, auch in Bezug auf die Performance, in einer Produktionsumgebung ausgiebig getestet werden, bevor sie livegeschaltet wird.

Canary Release baut auf dem Blue/Green-Deployments-Ansatz auf und ergänzt ihn um eine stufenweise Lasterhöhung. Ein neues Softwarerelease wird zunächst nur auf einigen Server 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 Benutzergruppe verfügbar zu machen. Der Rollout an die Endnutzer erfolgt dann jedoch, im Gegensatz zu den Blue/Green-Deployments, nach erfolgreichem Test nicht auf Knopfdruck (von 0 auf 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 (von 10 auf 20 Prozent etc.) zur Verfügung gestellt. Sollte es zu Problemen kommen, ist nach wie vor ein schneller Wechsel auf die alte Release-Version möglich.

Daraus ergeben sich folgende Vorteile von Continuous Deployment:

- Jede Änderung geht direkt in Produktion.
- Das Feedback erfolgt noch schneller als durch Continuous Delivery.
- Durch den Einsatz von Feature Toggles ist es möglich, nach Produktion zu deployen, ohne die neuen Funktionalitäten zu nutzen. Es sind also Teile, die noch unfertig sind, bereits in Produktion deployed. So kann bereits Feedback über das Deployment der neuen Teile gesammelt werden.
- Ebenfalls ist es möglich, eine neue Funktionalität bereits von einer kleinen Nutzergruppe testen zu lassen, um so ein frühes Feedback zu bekommen.

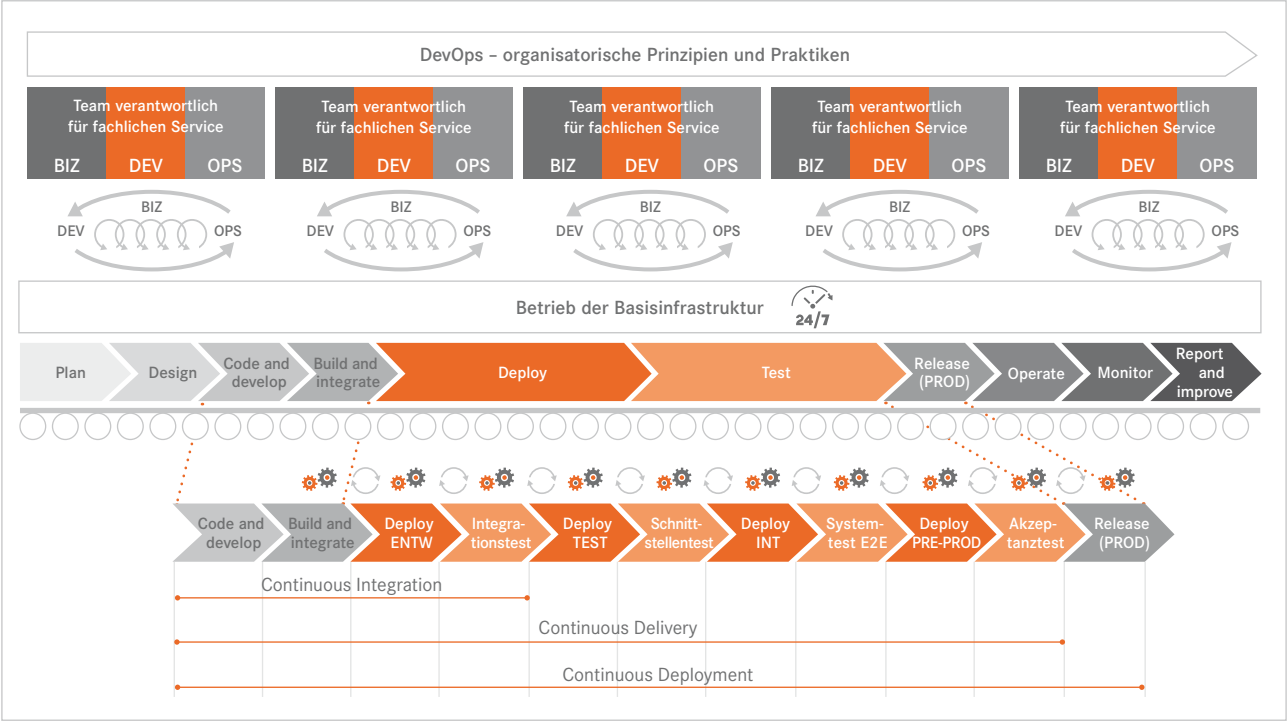

Abbildung 3: Continuous Delivery Pipeline

Bausteine einer Continuous Delivery Pipeline

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

- **Organisation/Personen:** Definition und Implementierung eines Organisationsmodells, welches 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, Umgebungsbereitstellung, 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.

Abbildung 3 gibt einen zusammenfassenden Überblick über alle Bausteine einer Continuous Delivery Pipeline inklusive einer zeitlichen Einordnung.

Ausblick

Im nächsten Teil der Artikelreihe lesen Sie, wie man den Continuous-Delivery-Reifegrad ermitteln kann, welche KPI genutzt werden können, um Erfolge messbar zu machen, und wie sich der Return on Investment eines DevOps-/Continuous-Delivery-Projektes ermitteln lässt.

Ansprechpartner

Erik Benedetto
Senior IT Consultant,
Center of Competence Application Management
erik.benedetto@msg-gillardon.de