Continuous Delivery

Übernehmen einer Softwareentwicklungsdisziplin

Björn Beha und Suhay Sevinc

Zusammenfassung—Der Artikel beschäftigt sich mit der Etablierung der Softwareentwicklungsdisziplin Continuous Delivery. Die Arbeit sich mit der Frage auseinander, ob das vollständige Umsetzen von Continuous Delivery auch ohne die Transformation hin zur DevOps-Kultur möglich ist. Somit fokussiert sich die Abhandlung auf die Umsetzung einer Continuous-Delivery-Pipeline unter Berücksichtigung von DevOps und beschreibt den Zusammenhang dieser Begriffe. Damit verbunden werden kulturelle, architektonische und technologische Aspekte untersucht und Hürden zur Umsetzung dieser Disziplin erläutert. Die Untersuchungen zeigen, dass sich nur durch eine Transformation hin zur DevOps-Kultur eine vollständige Pipeline aufbauen lässt. Ohne diese Umwandlung der Organisation lassen sich nicht alle Vorteile dieser Disziplin nutzen. Außerdem geht hervor, dass sich aus der Einführung von Continuous Delivery eine Einführung von Microservices ergibt. Bei diesem Softwarearchitektur-Ansatz stehen Anforderungen wie Unabhängigkeit und Technologiefreiheit ebenfalls im Vordergrund.

Index Terms—Contniuous Delivery, DevOps, Strategien, Hürden, Fehler, Umsetzung

1 Einführung

CONTINUOUS Integration wird bereits seit langem von vielen Unternehmen genutzt [1] Dadurch haben sie bereits einen stabilen Prozess, um nachhaltig qualitativ hochwertige Software in Produktion zu bringen. An dieser Stelle setzt nun Continuous Delivery an. Die Prinzipien, welche bei Continuous Delivery verfolgt werden klingen im ersten Moment vielversprechend, denn es ermöglicht das Ausrollen von Software mit einer wesentlich höheren Geschwindigkeit sowie Zuverlässigkeit. Auch wiederkehrende Herausforderungen zur Vorbeugung von Seiteneffekten und Regressionen lassen sich mit diesem Ansatz behandeln. Obwohl Continuous Delivery lediglich eine Sammlung von Techniken, Prozessen und Werkzeugen ist, stellt das Aufsetzen dieser Disziplin vor allem in großen Unternehmen mit unflexiblen Strukturen eine Herausforderung dar [2].

> *„Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations. “* [**?**, S.5]

Laut dem Gesetz von Conway bedeutet Continuous Delivery einen Bruch mit der bisherigen Organisation im Unternehmen. Da sich Continuous Delivery um das Ausliefern von Software dreht, müssen Entwicklung und Betrieb eng miteinander kooperieren. Es bedarf somit einer Ergänzung um ein entsprechendes organisatorisches Modell - DevOps. In dieser Arbeit wird ein Pfad beleuchtet, welchen ein Unternehmen gehen muss, um eine vollständige Continuous-Delivery-Pipeline aufzusetzen. Dabei gibt es unterschiedliche Voraussetzungen für Unternehmen. Beispielsweise hängt es vom aktuellen Zustand der Organisation bzw. zu welchem Zeitpunkt sie sich für diesen Schritt entscheidet ab. Ist das Projekt bereits fortgeschritten und muss die Bestehende Pipeline entsprechend erweitert werden oder kann das Projekt direkt von Anfang an darauf ausgelegt werden? Außerdem wird behandelt, was ein Unternehmen aufwenden muss, um diesen Schritt zu gehen und welche Auswirkungen diese Entscheidung auf das Unternehmen hat. Dabei wird gezeigt, welche Rolle DevOps bei dieser Entscheidung spielt, welche Komponenten dabei beeinflusst werden und welche Barrieren es gibt.

Zu Beginn der Arbeit erfolgt somit eine kurze Erläuterung von DevOps, die Prämisse für die unbeschränkte Umsetzung von Continuous Delivery. Darauf aufbauend wird beschrieben, welche Teile einer Organisation bei der Entwicklung zur DevOps-Kultur betroffen sind und welche Barrieren es hier zu bewältigen gibt. Abschließend wird in der Diskussion erläutert, welche Änderungen der Unternehmenskultur zu erwarten sind, wenn alle Vorteile von Continuous Delivery nutzen werden sollen.

2 Verwandte Arbeiten

Einen Überblick zum Themengebiet geben die beiden Bücher ”Continuous Delivery” [2] und ”Product-Focused Software Process” [3]. Diese werden in dem Artikel als Standard-Referenz herangezogen. Die angewandte Vorgehensweise steht in Beziehung zu diesen Werken. Für die Analyse der Barrieren greift der Artikel auf die Arbeiten von Laukkanen et al [4] und Shahin et al [5] zurück. Die beiden Publikationen betrachten eine Menge von Case-Studies und führen Literaturreviews aus, die hier aufgerührt werden.

3 DevOps

DevOps ist eine Organisationsform, mit der Continuous Delivery optimal funktioniert. Im Zentrum steht das Zusammenspiel von Entwicklung (Developement) und Betrieb (Operations), wodurch bereits das Wesentliche des Gedankens hervorgeht: Ein Zusammenwachsen der beiden Abteilungen zu einem Verband. Dabei fokussiert sich dieser Ansatz auf die Liefergeschwindigkeit,

das kontinuierliche Testen in produktionsähnlichen Umgebungen, kontinuierliche Rückmeldung und schnelle Reaktionsfähigkeit. Jedes Team ist somit für eine Domäne zuständig und kann diese unabhängig von anderen Teams als Service implementieren. Der Dienst kann schließlich selbständig optimiert werden, um einen guten Betrieb zu ermöglichen. Des Weiteren kann ohne weitere Abstimmung eine Weiterentwicklung am Dienst vorgenommen werden. Die Teams übernehmen die vollständige Verantwortung für ihre Komponenten und den eingesetzten Technologie-Stack. Daraus folgt eine geringere Koordination untereinander, wodurch wiederum Software schneller entwickelt und in Produktion gebracht werden kann. Auch Kunden wissen direkt, welcher Ansprechpartner für welchen Service zuständig ist (siehe Abbildung 1) [2].

In klassischen Organisationsformen sind Entwicklung und Betrieb in mindestens zwei unterschiedliche Abteilungen getrennt und verfolgen verschiedene Ziele. Während bei dem Betrieb Kosteneffizienz in den Vordergrund rückt, stellt die Entwicklung neue Features bereit und wird anhand der Geschwindigkeit und Effizienz ihres Vorgehens gemessen. Wird Continuous Delivery eingeführt, sollten Betrieb und Entwicklung Hand in Hand arbeiten, um das Notwendige Wissen und entsprechende Werkzeuge zu teilen. Jede Abteilung beherrscht durch die unterschiedliche Perspektive auf die Anwendungen einen Teil von Continuous Delivery. Ohne Monitoring, Security oder Netzinfrastrukturen kann eine Anwendung nicht sinnvoll realisiert oder betrieben werden. Der Betrieb ist für diese Aspekte zuständig, wohingegen die Entwicklung den Code, die Entwicklungsinfrastruktur und Middleware wie Application Server etc. kennt. Gemeinsam lässt sich somit eine Continuous-Delivery-Pipeline aufbauen. Mit der Einführung von Continuous Delivery sollte allerdings auch die Einführung von DevOps in Betracht gezogen werden [2].

Continuous Delivery wird häufig auch mit DevOps gleichgesetzt. Das trifft jedoch nicht zu. Durch DevOps lässt sich Continuous Delivery zwar drastisch vereinfachen, allerdings gibt es neben dieser (wesentlichen) Praktik im DevOps-Umfeld noch weitere Bereiche (Monitoring, Troubleshooting etc.), bei denen diese Organisationsform und eine damit einhergehende Zusammenarbeit sinnvoll sein kann [2].

4 Entwicklung zur DevOps-Kultur

Es ist denkbar, einen gewissen Teil von Continuous Delivery auch ohne DevOps einzuführen. Letztendlich bringen diese Ansätze weniger Vorteile mit sich. Hierfür muss es nicht zwingend fachlich organisierte Teams geben. Im Grunde reicht eine Kooperation der Abteilungen, um gemeinsam an der Continuous-Delivery-Pipeline zu arbeiten. Continuous Delivery verfolgt eine Vielzahl von Zielen. Einige können trotz reduzierter Pipeline erreicht werden. Hierfür ist keine umfangreiche Umorganisation notwendig.
Wenn Continuous Delivery vollständig umgesetzt werden soll, gelingt dies nur mit der Einführung von DevOps. Dies bedeutet eine fundamentale Änderung der Organisation. Eine solche Transformation ist mit einem erheblichen Aufwand verbunden und kann vor allem in großen Konzernen schwer umsetzbar sein. In diesem Kapitel wird darauf eingegangen, welche Aspekte zu berücksichtigen sind und welche Folgen diese Neuorientierung für Unternehmen hat [2].

4.1 Umsetzung von Continuous Delivery

Continuous Delivery ist lediglich eine technische Praktik, trotzdem hat diese Disziplin verschiedene Auswirkungen auf die Organisation. Continuous Delivery steht und fällt mit der Pipeline. Das Umsetzen einer einfachen Delivery-Pipeline kann mit simplen mitteln erfolgen. Vor allem zu Beginn eines Projekts kann der Aufbau einer solchen Pipeline verfolgt werden. Beispielsweise kann ein Unternehmen hier vorab mit den Phasen Commit und Deploy beginnen, da diese schließlich existieren müssen. Es ist dann möglich, die Pipeline im weiteren Verlauf schrittweise weiterzuentwickeln, indem entsprechende Phasen (z. B. automatisierte Tests) hinzugefügt werden. Wird Continuous Delivery zum Start eines Projekts eingeführt, kann dies auch bei der Auswahl der Technologien und Architekturen berücksichtigt werden [2].

Oft setzt Continuous Delivery an eine bestehende Codebasis an. Die bisherige Architektur ist gegebenenfalls nicht für den Einsatz von Continuous Delivery konzipiert worden. Wie eben erwähnt, müssen Prozesse zur Auslieferung der Software bereits bestehen, wodurch eine Delivery-Pipeline in gewisser Weise vorausgesetzt werden kann. Nur können diese Prozesse komplex sein. Das Ziel von Continuous Delivery ist allerdings, einen konstanten Fluss von Features und Codeänderungen durch die Pipeline zu erreichen. Hierbei muss der Value Stream betrachtet werden. Dies bezeichnet den derzeitigen Ablauf bis zum Release der Software. Werden die einzelnen Stationen betrachtet, lassen sich Optimierungsmaßnahmen ableiten. Es können beispielsweise Engpässe ermittelt werden, um den Fluss zu optimieren, auch wenn die Pipeline noch nicht gänzlich automatisiert ist. Dieses Verfahren wird als Value Stream Mapping bezeichnet und stellt eine wertvolle Analysetechnik dar, um bestehende Pipelines schrittweise zu verbessern. Obwohl Continuous Delivery eine Sammlung verschiedener Werkzeuge ist, reicht es bei der Umsetzung nicht, schlicht entsprechende Techniken und Tools zu verwenden. Solange die Prozesse innerhalb der Organisation nicht einwandfrei ablaufen, kann hier keine signifikante Steigerung der Effizienz erreicht werden [6]. Der Schlüssel hierfür ist DevOps, da das Know-How aus ausschlaggebenden Abteilungen zusammenkommt [2].

4.2 Umsetzung von DevOps

Viele Unternehmen sind bereits dabei, DevOps erfolgreich umzusetzen. Allerdings bleiben nach wie vor kritische Barrieren bei der Entwicklung, wie sie im folgenden Kapitel näher beschrieben werden. Dabei ist die größte Hürde die Angst vor Veränderungen (siehe vorherige Abneigung gegenüber Clouds) [7]. Um diese Philosophie anzuwenden gibt es verschiedene Möglichkeiten, allerdings muss klar

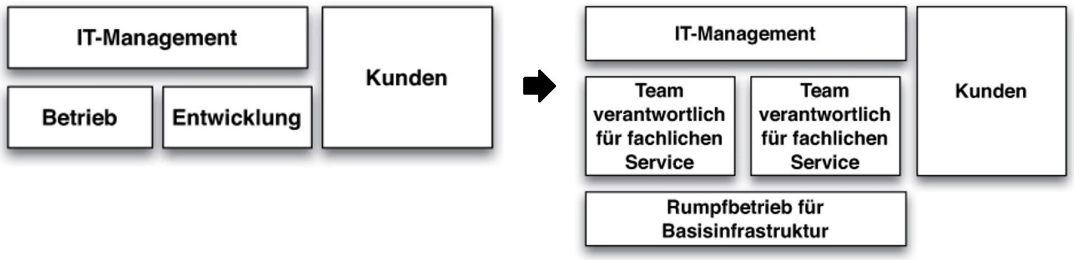

Abbildung 1. Wandel von klassischer Organisation zu DevOps-Teams

sein, was DevOps für das Unternehmen bedeutet. Zwei der größten Hindernisse, diese Disziplin erfolgreich umzusetzen, sind die Unternehmensarchitektur sowie Unternehmenskultur, die davon beeinflusst werden.

4.2.1 Kultur

Die Unternehmenskultur ist von der Transformation betroffen. Es ist wichtig, eine Kultur zu schaffen, in welcher alle Menschen der Organisation bei der Verfolgung gemeinsamer Ziele zusammenarbeiten. Bei DevOps wird stets die primäre Bedeutung der Kultur hervorgehoben. Der Fokus liegt auf einer effektiven Zusammenarbeit der Entwickler und IT-Betriebsteams. Durch diese Kollaboration, welche die Unabhängigkeit und Selbständigkeit der Teams fördert, bedarf es einer zentralen Kontrolle nur noch um bestimmte Rahmenbedingungen zu etablieren und zu kontrollieren.

Ein Extrembeispiel bezüglich der im Unternehmen verfolgter Kultur ist wohl Netflix mit seiner Simian Army. Dabei werden beispielsweise regelmäßig zufällige Instanzen ihrer Architektur zerstört, um (kontrollierte) Unfälle zu simulieren. Somit wird das unabhängige Funktionieren der Komponenten sichergestellt. Leistungsstarke Unternehmen warten also nicht darauf, dass Ausfälle passieren, um daraus zu lernen und sich zu verbessern. Durch dieses radikale Vorgehen bzw. diese Architektur des Systems lassen sich Optimierungen ableiten [7].

4.2.2 Architektur

Continuous Delivery sowie DevOps und die somit weitgehend autonome Teams bedeuten auch für die Architektur der Lösungen eine Herausforderung. Zum einen erfordert es ein gewisses Maß an Abstimmung zwischen den Teams, um gemeinsam die Architektur des Systems zu entwickeln. Das Festlegen der Architektonik durch eine zentrale Instanz stellt einen Widerspruch zu den eigenverantwortlichen Teams dar. Die Abstimmung erfolgt über entsprechend definierte und moderierte Meetings, an denen auch zentrale Architekten teilnehmen können. Zum anderen muss die Delivery-Pipeline des Systems entsprechend aufgebaut bzw. angepasst werden.

Wird an einer Stelle des Systems eine Änderung durchgeführt, muss nicht die ganze Pipeline durchlaufen werden. Das würde zum Testen des gesamten Systems führen. Zwar kann dies durchaus sinnvoll sein, da Fehler in einer Komponente erst durch ein Fehlverhalten einer anderen Komponente auftreten können (Seiteneffekte), trotzdem wäre die Folge ein unnötig großer Aufwand und ein verzögertes Feedback. Durch Continuous Delivery ergibt sich demnach eine andere technische Umsetzung der Komponenten, denn jede Komponente sollte als eigene Einheit betrachtet und implementiert werden. Somit lassen diese sich unabhängig voneinander als Teil des Gesamtsystems deployen und durchlaufen jeweils eine eigene Pipeline. Das schnelle Feedback und die angestrebte Risikominimierung wird hierdurch erreicht. Schleicht sich ein Fehler ein, kann er leichter identifiziert werden, da die Deployment-Einheiten wesentlich kleiner sind. Auch die Kommunikation zwischen den Komponenten ist betroffen. Wird jede Komponente als eigenständiger Service als Teil eines verteilten Systems implementiert, muss die Kommunikation angepasst werden. Potenzielle Lösungen bieten REST oder entsprechende Message Oriented Middleware. Möglichst kleine, lose gekoppelte und gekapselte Module sind somit für Continuous Delivery sinnvoll (im Rahmen der objektorientierten Programmierung werden hier die Prinzipien des SOLID-Designs verfolgt). Die daraus resultierenden verteilten Systeme machen das Design entsprechender Schnittstellen zu einem wichtigen Faktor [2].

Durch verschiedene Anpassungen der Architektur und den bisher erwähnten Anforderungen an die Organisation ergibt sich ein umfangreiches Zusammenwirken von Continuous Delivery und dem Softwarearchitekturansatz von Microservices. Microservices genügen den Anforderungen an einer Architektur für ein Continuous Delivery-System und setzen auch andere Aspekte (Technologiefreiheit, unabhängige Einheiten etc.) um [2]. Diese architektonischen Eigenschaften wurden hier sogar explizit priorisiert [7]. Somit ist die Einführung von Continuous Delivery im Grunde eine Einführung von Microservices. Microservice-basierte Anwendungen sind Applikationen, die sich aus einer Reihe von kleinen Komponenten zusammensetzen. Diese durchlaufen dabei alle ihre eigenen Prozesse. Jede Einheit muss unabhängig deployt werden können. Hierfür wird die Continuous-Delivery-Pipelne genutzt, die durch eine weitgehende Automatisierung dafür sorgt, dass Software bereitgestellt werden kann [Microservice Buch vom Eberhard]. Bei der Umsetzung von Continuous Delivery muss somit die Architektur der Software angepasst werden. Lediglich die Prozesse anzupassen bzw. zu ändern, ist nicht ausreichend [2].

4.2.3 Technolgie

Wie erwähnt, ist die Technologiefreiheit ein Aspekt dieses Ansatzes. Da die Teams weitgehend autonom handeln können, übernehmen sie die vollständige Zuständigkeit bei der Auswahl ihrer Implementierungstechnologien. Frameworks, Programmiersprachen oder Application Server werden eigenverantwortlich ausgewählt. Diese Entscheidungen haben an einigen Stellen Einfluss auf die Pipeline. Programmiersprachen beeinflussen beispielsweise die Geschwindigkeit der Compiler, wodurch die benötigte Zeit beim Durchlaufen der Pipeline beeinflusst wird. Technologieentscheidungen haben demnach einen direkten Einfluss auf die Continuous-Delivery-Pipeline [2].

Treten mit den ausgewählten Technologien Probleme auf, muss sich auch nur dieses Team mit der Behebung auseinandersetzen. So lässt sich eine Vielzahl von Technologien im Unternehmen nutzen, während eine klassische Organisation diese möglichst einschränken und kontrollieren möchte. So sollen beispielsweise potentielle Risiken beim Interagieren der Komponenten minimiert werden. Teilen sich Komponente und Projekte dieselbe Programmiersprache und Infrastruktur, kann jeder Entwickler mit der entsprechenden Technologie umgehen. Es kann jedes Projekt in gewisser Weise unterstützt werden. Auch der Betrieb kann sich so auf eine Technologie fokussieren und ist mit möglichen Problemen dieses Technologie-Stacks vertraut. Bei DevOps hingegen steht die Freiheit, eigene Entscheidungen zu treffen im Vordergrund. Ein Standard-Stack kann etabliert werden, indem auf Erfahrungen anderer Teams zurückgegriffen wird. Somit könnte es bereits eine Ansammlung von Tools geben, welche für andere Teams nutzbar ist [2].

5 TRANSFORMATIONSFEHLER

In Kapitel 4 wurde gezeigt wie komplex die Etablierung von Continuous Delivery für Unternehmen ist. Bei diesem Prozess können Probleme auftreten, die in dem Abschnitt diskutiert werden. Es tritt häufig der Fehler auf, dass Unternehmen dazu neigen, Continuous Delivery als einen Endzustand anzusehen und die Mentalität innerhalb der Organisation nicht anpassen. Allgemein lassen sich die Menge der Probleme in die sechs Kategorien Builddesign, Systemmodularisierung, Integration, Testphase, Release, Mensch und Organisationsstruktur klassifizieren. In der Arbeit von Laukkanen et al treten hier 40 Probleme auf (siehe Abbildung 2) [4].

5.1 Builddesign

Bereits die Entscheidung, wie das Builddesign aufgebaut und konfiguriert wird, kann eine zukünftige Problemursache darstellen. Ein Grund hierfür kann die Erstellung von komplexen und unflexiblen Buildskripten sein. Das Builddesign ist dadurch stark an das Zielsystem gekoppelt, sodass minimale Skritpänderungen zu Buildfehlern führen. Dies erfordert intensive Pflege- und Wartungsarbeiten, welche die Einführung einer DevOps-Kultur drosseln. Aber auch stellt die Modellierung des Systems eine Abhängigkeit für das Builddesign dar, da die Auflösung von Abhängigkeiten kritisch sein könnte (zum Beispiel zyklische Abhängigkeiten).

5.2 Systemmodularisierung

Wie in Abschnitt 5.1 aufgezeigt wurde stellt die Systemmodularisierung eine weitere Problemursache dar. Eine geeignete Systemmodellierung führt zu einer autonomen und unabhängigen Entwicklung. Von einer ungeeigneten Architektur ist die Rede, wenn sie monolithisch gekoppelt ist. Eine ungeeignete Systemmodularisierung führt zu einem überhöhten Entwicklungsaufwand, Testbarkeit und Wartbarkeit. Dadurch erhöht sich sowohl die Fehleranfälligkeit des Systems als auch die Code-Inkonsistenz. Daraus folgt, dass sich die Software in einem nicht auslieferbaren Zustand befindet. Jedoch kann auch bei einer geeigneten Architektur Probleme auftreten, indem zu starke Systemmodellierung betrieben wird. Das führt zu einer Erhöhung der Komplexität, sodass neue Teammitglieder die Software schwer weiterentwickeln und warten können [4].

5.3 Integration

Die Integration umfasst Probleme, die bei der Zusammenführung von Softwarekomponenten entstehen [8]. Sobald diese nicht wie vorgesehen zusammengeführt werden, treten hier Probleme auf. Nicht selten kommt es vor, dass Entwickler einmal täglich Quellcodeänderungen hochladen. Das hat die Folge, dass eine hohe Anzahl von Änderungen mit sich bringt und diese im Konflikt mit anderen Änderungen stehen. Im Fall eines Fehlers im Buildprozess kann der Fehler nicht sofort identifiziert werden und es entsteht eine Arbeitsblockade. Des Weiteren hat diese Vorgehensweise den Nachteil, dass überhöhter Netzwerklatenz erzeugt wird. Eine oft genutzte Methode ist hier das Erstellen von mehreren Abzweigungen des Hauptentwicklungsstrangs. Diese Lösung ist allerdings problembehaftet. Während die Entwicklung am Hauptentwicklungsstrang voranschreitet, divergieren Verzweigungen, sodass die Vereinigung der Stränge zum Problem wird. Die Lösung dieser Viereinigungsprobleme resultiert in mehr Aufwand, was zur einer reduzierten Produktivität führt [4].

5.4 Testphase

Tests sollten eindeutige Validierung vornehmen. Bei bestimmten Tests, wie beispielsweise ”flaky tests”, entstehen hier langwierige Probleme. Bei diesen Tests werden zufällige Resultate erzeugt, die nicht determinierbar sind. Das erschwert die Testphase. Das geht aus dem folgenden Zitat hervor:

> „...several of the automated activities do not yield a clear “pass or fail” result... “ [4, S.65]

5.5 Mensch und Organisationsstruktur

Eine letzte Problemursache ist der Mensch, welcher aktuell die größte Herausforderung darstellt. Die Adaption von Continuous Delivery erwartet wie aufgeführt Agilität und Flexibilität. Das setzt voraus, dass eine Akzeptanz für diese Vorgehensweisen vorhanden ist. Dies stellt die erste

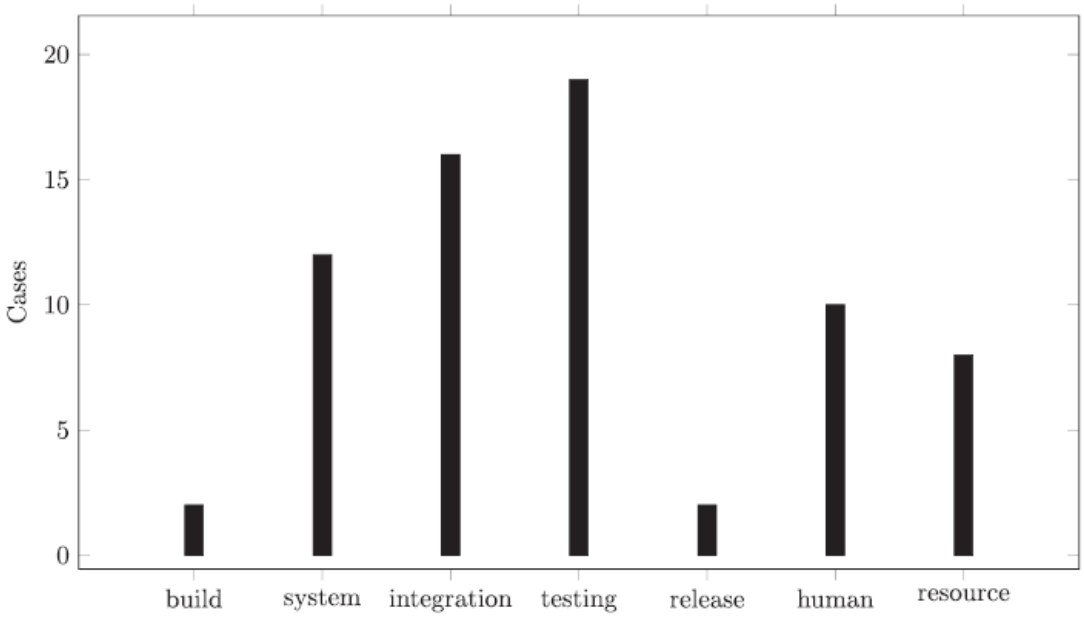

Abbildung 2. Visualisierung der Probleme der einzelnen Bereiche [4].

Hürde dar, wie im folgenden Zitat zu sehen ist. Sowohl auf Management- als auch Mitarbeiterebene wird Motivation und Disziplin verlangt. Des Weiteren muss erwähnt werden, dass in Softwareteams mehr Druck herrscht, da sich die Software in einem ständig auslieferbaren Zustand befinden muss. Das kann negative Auswirkungen auf die Motivation der Teammitglieder haben und führt gegebenenfalls zu einer angespannten Arbeitsatmosphäre. Aus technischer Sicht verlangt Continuous Delivery sowohl umfangreiches Wissen im Bereich Buildmanagement als auch bei der Programmierung von unterschiedlichen Skripten [4]. Das erfordert eine Einarbeitung in die Thematik.

> „...But it was hard to convince them that we needed to go through our implementation "hump of pain"to get the pieces in place that would allow us to have continuous integration. I worked on a small team and we didn't seem to have any "extra"time for me to work on the infrastructure we needed. " [9, S.373]

6 Diskussion

Continuous Delivery ist ohne Frage ein lohnenswerter Schritt, den ein Unternehmen gehen kann, um seine Softwarequalität weiter zu verbessern und das Risiko beim Release drastisch zu minimieren. Wie bereits erwähnt, kann Continuous Delivery bzw. ein Teil davon auch ohne DevOps umgesetzt werden. Da es teilweise schon ausreicht, eine gewisse Zusammenarbeit zu ermöglichen, müssen die Teams nicht vermischt bzw. die Organisation umstrukturiert werden. Diese Zusammenarbeit erreicht man bereits, wenn die Arbeitsplätze der beiden Abteilungen zusammengelegt werden. Diese erleichtert die Kommunikation der Mitarbeiter wesentlich. Dies ist bereits ein sinnvoller Schritt für Unternehmen, es kann allerdings nicht der ganze Nutzen aus Continuous Delivery gezogen werden. Zu Beginn des Artikels wurde davon ausgegangen, dass Continuous Delivery vollständig eingeführt werden soll um alle Vorteile dieser Disziplin zu erhalten. Dabei reicht es nicht, entsprechende Werkzeuge anzuwenden und die Kommunikation der Teams anzupassen. Es gilt diverse Hürden zu überwinden, wie aus den obigen Kapiteln bereits hervorgegangen ist.

Um eine geeignete DevOps-Kultur in das Unternehmen einzuführen bedeutet zunächst auch einen enormen initialen Aufwand für das Unternehmen. Auch im laufenden System fallen nun regelmäßig Weiterentwicklungen und womöglich Wartungen an. Die Erstellung von Tests muss zudem ein integraler Bestandteil der Entwicklung werden. Dabei muss entsprechendes Know-how zum Schreiben der Tests in den Teams verteilt und Testframeworks evaluiert werden. Die Tests müssen dabei unabhängig voneinander sein. Anschließend dreht sich sehr viel um die Automatisierung. Diese umfasst die Ausführung genannter Tests und die Bereitstellung resultierender Daten. Damit diese Tests zuverlässige Ergebnisse liefern, muss die Entwicklung und der Betrieb das Deployment und die Konfiguration der Anwendung gemeinsam planen, anpassen und warten [10]

Dabei sollte dieser Ansatz möglichst zu Beginn eines Projekts verfolgt werden, um die Pipeline schrittweise aufzubauen und Entscheidungen bezüglich Technologie und Architektur darauf auszurichten. An eine bestehende Codebasis anzusetzen bedeutet oftmals eine fundamentale Änderung der bisherigen Techniken und Abläufe, denn wie bereits aus den obigen Kapiteln hervorgegangen ist, bedeutet das Einrichten einer DevOps-Kultur unter anderem eine Änderung an der gesamten Architektur des Systems. Dies bedeutet oftmals einen gewissen Leistungsdruck. Die Implementierung der Automatisierung benötigt ebenfalls Zeit. Der hier erbrachte Aufwand ist in dieser Zeit nicht für eine Weiterentwicklung des Systems

bzw. der Features verfügbar.

Continuous Delivery beeinfluss somit weitgehend Elemente eines Unternehmens und ohne die Ergänzung des organisatorischen Modells DevOps, kann dieser Ansatz nicht vollständig umgesetzt werden. Die Umsetzung bedeutet hierbei eine Einführung von Microservices, welche entsprechende Anforderungen sogar priorisieren. Dadurch, dass es bei Microservice-basierten Systemen zu tausenden Deployments pro Jahr kommen kann, bietet die Delivery-Pipeline mit ihrer Automatisierung eine notwendige Technik, um anfallende Aufwände im Zaum zu halten.

LITERATUR

[1] automic, "Was ist continuous integration? und was ist es nicht?" 2017. [Online]. Available: https://automic.com/de/blog/was-ist-continuous-integration-und-was-ist-es-nicht

[2] E. Wolff, *Continuous Delivery: Der pragmatische Einstieg*, 2nd ed., 2016.

[3] A. Jedlitschka, P. Kuvaja, M. Kuhrmann, T. Männistö, J. Münch, and M. Raatikainen, *Product-Focused Software Process Improvement*. Cham: Springer International Publishing, 2014, vol. 8892.

[4] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—a systematic literature review," *Information and Software Technology*, vol. 82, pp. 55–79, 2017.

[5] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[6] M. Virmani, "Understanding devops & bridging the gap from continuous integration to continuous delivery," in *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, Manish Virmani, Ed. IEEE, 2015, pp. 78–82.

[7] continuousdelivery, "Implementing continuous delivery - continuous delivery," 2017. [Online]. Available: https://continuousdelivery.com/implementing/

[8] L. Chen and P. Power, "Continuous delivery: Huge benefits, but challenges too," *IEEE Access*, pp. 50–54, 2015.

[9] S. Stolberg, "Enabling agile testing through continuous integration," in *Agile '09, Agile Conference, 2009*, Y. Dubinsky, Ed. Piscataway, NJ: IEEE, 2009, pp. 369–374.

[10] A. Birk and C. Lukas, "Eine einführung in continuous delivery, teil 1: Grundlagen," 2014. [Online]. Available: https://www.heise.de/developer/artikel/Eine-Einfuehrung-in-Continuous-Delivery-Teil-1-Grundlagen-2176380.html