Automatisierung: die sichere Basis für eine agile Transformation.

Ein Whitepaper mit einer ganzheitlichen Sicht auf das Thema und praktischen Hinweisen.
Erstellt aus einer Partnerschaft zwischen Cognizant, Red Hat und SIX Group.

Red Hat

Cognizant

Abstract: Agil ja, aber mit Augenmass

Laut einer Gartner-Studie haben mehr als 70 Prozent der global aktiven Grossunternehmen eine schadhafte agile Transformation durchlaufen. Verglichen mit der Automobilindustrie ist es so, wie wenn jedes Unternehmen massive Investitionen getätigt hätte, um die Zulieferung von Ersatzteilen agiler zu gestalten und Arbeitsschritte am Fliessband zu automatisieren, aber dabei teilweise komplett die Qualitätssicherung vergass. Man hatte nur das Ziel vor Augen, das Auto so schnell wie möglich auf den Markt zu bringen und diverse Key-Performance-Indikatoren in der Montage zu optimieren, jedoch kein Konzept, wie man die Qualitätssicherung harmonisch in den gesamtheitlichen Prozess einbindet und entsprechend voll automatisiert.

Die Studie von Gartner belegt diesen Vergleich: 70 Prozent der befragten Unternehmen aus dem Fortune 500 Index sagen aus, dass sie bereits eine DevOps-Strategie implementiert haben und 88 Prozent haben eine agile Transformation abgeschlossen. Aber nur 26 Prozent haben ein umfängliches Konzept zur Testautomatisierung. Ein Grossteil der Unternehmen stellt also die agile Transformation vor die Qualitätssicherung.[1]

Dieses Whitepaper beleuchtet die Grundzüge der agilen Softwareentwicklung und zeigt, welche zwingenden Abhängigkeiten zur DevOps-Thematik sowie zur Testautomatisierung bestehen. Es werden Erfahrungswerte aus beiden Welten aufgeführt und demonstriert, was es zu beachten gilt und wie genau ein Testkonzept auszusehen hat, das die gesamtheitliche Effizienzsteigerung verbessert. Wie die einzelnen Bereiche am besten zusammenarbeiten, wird anhand einer Beispielarchitektur erläutert.

1. Agile Software-Entwicklung

Starten wir zunächst mit der agilen Software-Entwicklung. Die ersten Ansätze zur agilen (inkrementellen) Vorgehensweise in der Software-Entwicklung kann man zurückverfolgen bis ins Jahr 1957, obwohl die Bezeichnung “agil” erst im Februar 2001 bei einem Treffen in Utah definiert wurde. Das ganze Thema ist also nicht neu und viele Unternehmen, aber auch Consulting-Häuser, haben seit dieser Zeit einen massiven Lernprozess durchlaufen.

Zwei oft angewendete Methoden in der agilen Entwicklung möchten wir kurz erklären:

Scrum:
Scrum ist im Grundsatz ein selbst organisiertes, interdisziplinäres Team, das sich täglich abstimmt. Dabei wird ein Produkt auf Basis eines Product Backlog (priorisierte Feature-Liste) in kurzen Inkrementen von typischerweise 2 Wochen Länge, den sogenannten Sprints, kontinuierlich weiterentwickelt. Ein Scrum-Team setzt sich aus Scrum-Master, Product Owner (PO) und dem Projektteam zusammen.

Auf Basis der Produkt-Vision wird das Product Backlog angelegt und gepflegt. Im Sprint Planning wird das Sprint Backlog für das nächste Inkrement erstellt und anschliessend im Tasking auf kleine Arbeitsschritte heruntergebrochen. Während des Sprints wird im Daily Scrum der Arbeitsfortschritt kontinuierlich besprochen, welcher auf einem Scrum-Board festgehalten wird und jederzeit ersichtlich ist. Im Sprint Review werden die Sprint-Ergebnisse geprüft und abgenommen und mit einer Sprint Retrospective das Verbesserungspotential des Teams für den nächsten Sprint diskutiert, welcher im Anschluss dann beginnt.

[1] https://www.plutora.com/blog/agile-devops-failing-fortune-500-companies-wake-call-us

SAFe:
Das Scaled Agile Framework (SAFe) ist das am meisten verbreitete Rahmenwerk, um mit Scrum bzw. agil skalieren zu können. Das Vorgehen ist analog zu Scrum, jedoch bietet SAFe für die Skalierung zusätzliche Gefässe wie Program Increments (ein festgelegter zeitlicher Rahmen, in welchem durch den Release Train ein Inkrement geliefert wird), eine Portfolio-Planung (das Portfolio Backlog ist das hierarchisch höchste Backlog) und Agile Release Trains (das primäre value-delivery Konstrukt).

Im letztjährigen, 14ten Stage of Agile Report[2] war SAFe mit 35 Prozent der Leader unter den agilen Methodologien, gefolgt von Scrum of Scrums mit 16 Prozent. Je nach Grösse und Agilitätsgrad eines Unternehmens gibt es verschiedene Frameworks, die in Frage kommen. Jedoch strebt man typischerweise den dynamischeren Weg in der Softwarelieferung an. Er verspricht weniger Vorbereitungsaufwand, schnelleres Ausliefern von produktiven Ergebnissen und weniger Aufwand bei dynamischen Kurswechseln innerhalb der Releases. Agilität bedeutet aber nicht, dass man Abkürzungen bei der Sicherung der Qualität nehmen kann. Sie fordert sogar eine viel stärkere Bereitschaft zu Disziplin und Qualität über den gesamten Software Delivery Lifecycle hinweg.

Ob SCRUM, SAFe oder sogar KANBAN, alle Methoden haben eines gemeinsam: oftmals rutschen ehemalige Business-Stakeholder näher an die IT-Teams heran. Die Product Owner (PO) bei SCRUM sind oftmals ehemalige Mitarbeiter aus dem Business. Man tut diese Tatsache oft mit Hinweisen ab, wie: “Da ist auch ein kultureller Wechsel notwendig”. Allerdings geht es hier nicht primär um die Kultur, sondern darum, dass die Person, die neu die komplette Lieferverantwortung für ein agiles Team besitzt, gegebenenfalls über keinen IT-Hintergrund verfügt.

Will man IT-seitig eine automatisierte Qualitätssicherung parallel zum Arbeitsalltag hochziehen, sind nicht nur Grundlagen an IT-Wissen wichtig. Das ganze Thema ist zusätzlich alles andere als Basiswissen. Man ist also gut beraten, den POs entweder IT-Coaches zum Thema automatisiertes Testen zur Seite zu stellen oder diese tiefgehend über das Thema zu schulen oder Experten direkt in die jeweiligen agilen Teams zu beordern.

Der organisatorische Aspekt ist allerdings nur eine Seite der Medaille. Eine weitreichende Effizienzsteigerung erreicht man nur, wenn einem agilen Team eine Infrastruktur, ein Toolstack gegenüber steht, das es auch technisch ermöglicht, schneller zu liefern. Wieso das eine rein rechnerisch nicht ohne das andere geht, wird weiter unten erläutert.

[2] https://stateofagile.com/#ufh-i-615706098-14th-annual-state-of-agile-report/7027494

2. DevOps
Zunächst aber ein kurzer Blick auf den Teil der agilen Transformation, welcher alle technischen Aspekte in sich birgt: DevOps. Das Wort DevOps hat mittlerweile sicher schon jede und jeder mindestens einmal gelesen. Der DevOps-Ansatz[3] hat zum Ziel, das Zusammenspiel von Entwicklung und Betrieb von Software zu verbessern, so dass der IT-Betrieb mit der agilen Entwicklung Schritt halten kann. Daher gilt auch, dass DevOps für eine erfolgreiche agile Transformation notwendig ist. Treffend heisst es darüber: “Wenn es um Automatisierung geht, könnte man es auch DevOps nennen”.

[3] Artikel zu DevOps auf Wikipedia

Die durch DevOps entstehende End-2-End-Verantwortung eines oder mehrerer eng zusammenarbeitender Teams verbessert die operationelle Stabilität. Ausgehend vom klassischen Betriebsmodell bestehend aus 1st-, 2nd- und 3rd-Level-Support, steht der Betrieb einer Applikation oft vor folgenden Herausforderungen:

- Probleme, die auf Ebene 3rd-Level-Support gelöst werden müssen, werden für den Kunden oft unbefriedigend adressiert, da es lange dauern kann, bis die richtige Hilfe zur Verfügung steht.
- Ineffizienter Betrieb, da die Umsetzung effizienzsteigernder Funktionen für 1st- und 2nd-Level-Support häufig nicht die notwendige Priorität im Business oder der Entwicklungsabteilung geniesst.

DevOps verspricht im Supportfall nicht nur die Wege zu verkürzen, sondern es sorgt auch dafür, dass ineffiziente Prozesse oder fehlende Toolunterstützung im Betrieb das Team direkt betreffen. Ganz nach dem Motto “eat your own dog food” ist es daher im Interesse des DevOps-Teams, die operativen Prozesse zu optimieren und somit mehr Kapazitäten für die Entwicklung neuer Funktionalitäten zu schaffen.

DevOps verlangt dadurch aber auch einen breiten Blickwinkel und oft auch ein breites technisches Wissen der involvierten Mitarbeiter. Ebenso braucht es den Willen der Mitarbeiter, diese End-2-End-Verantwortung zu übernehmen. Gerade in der Transformation eines Teams hin zum DevOps-Modell kann dies zu Konflikten führen. Die Einführung von DevOps ist damit ein aufwändiger Prozess, dem die notwendige Aufmerksamkeit und Unterstützung des Managements geschenkt werden muss.

3. Automatisierung

Die Vorteile der agilen Software-Entwicklung, wie beispielsweise die stetige Verfügbarkeit einer lauffähigen Version eines sich inkrementell entwickelnden Produkts oder die hohe Transparenz der erzielten Fortschritte, werden von einigen Herausforderungen begleitet.

Das Bauen, Paketieren und Installieren von Software ist in der Regel ein hoch automatisierbarer Prozess, für dessen Unterstützung es heute zahlreiche Tools gibt. In einem agilen Software-Projekt ist es daher stark zu empfehlen, mindestens eine Continuous-Integration-Lösung einzusetzen, welche die Software automatisch baut und paketiert. Wenn sinnvoll, erlaubt Continuous Deployment zusätzlich die Installation auf verschiedenen Betriebsstufen bis hin zur Produktion.

Software-Entwickler checken ihre Arbeit regelmässig in sogenannte Shared Repositories (Source-Code-Verwaltung) ein. Dadurch wird automatisch ein Build und Deploy ausgelöst. Der Continuous Integration (CI)-Server wird nach dem Kompilieren zudem vordefinierte, automatisierte Tests ausführen und Feedback (Compiler Output und Tests) rasch dem agilen Team zurücksenden. Die vordefinierten Tests können neben funktionalen Tests auch Source-Code-Verifikationen nach den Entwicklerrichtlinien, Code-Style, Memory Leak Detection, etc. beinhalten.

Diese regelmässigen Builds stehen nun im Gegensatz zu täglichen oder wöchentlichen Builds. Sie erlauben es, Fehler rascher zu finden und deren Ursachen schneller zu beheben (Fail Fast). Fail Fast aber auch Shift Left, das frühe und umso breit gefächerte Testen wie nur möglich, bringt aber komplett neue Anforderungen an die Testautomatisierung mit sich, die es zu erfüllen gilt.

4. Optimum zwischen Aufwand und Automatisierungsgrad

Durch die kurzen Zyklen in der agilen Entwicklung ist das ständige Testen der Software unerlässlich, um sowohl die Korrektheit des Funktionsumfangs, als auch nicht-funktionale Anforderungen, wie Paketier- und Installierbarkeit, Performance, etc. mit jedem Inkrement sicherstellen zu können. Die Effizienz der agilen Methoden lässt sehr schnell mit der Menge an manuellen Schritten nach, welche in die Sicherstellung der Qualität jedes einzelnen Inkrements fliessen. Daher gilt es für diese Herausforderungen automatisierte Lösungen zu suchen bzw. zu entwickeln.

Beim Testen der Software sowohl auf funktionale Korrektheit als auch auf Aspekte wie die Performance, gilt es sich der optimalen Balance aus manuellem und automatisiertem Testing anzunähern. Was ist mit diesem Optimum gemeint? Stellt man sich den Grad der Testautomatisierung als eine prozentuale Skala von 0 bis 100 vor - wobei 0% für ein komplett manuelles Testing steht und 100% für eine komplett automatische Testabdeckung - dann ist der Zielwert von 100% auf den ersten Blick natürlich erstrebenswert. Demgegenüber steht jedoch der Aufwand, der in die Testautomatisierung investiert werden muss. Wird dieser so hoch, dass die Pflege der Testautomatisierung mehr Aufwand benötigt als das manuelle Testen der betroffenen Testszenarien, dann ist die Automatisierung nicht länger effizient. Sie wird dann selbst zur Bremse.

Abbildung Nr. 1 zeigt dies stark vereinfacht und beispielhaft, wobei die Kurvenverläufe je nach Art des Softwareprojekts sehr unterschiedlich aussehen können. Mit zunehmendem Automatisierungsgrad (x-Achse) sinkt der manuelle Testaufwand, der Aufwand für die Pflege der Automatisierung steigt jedoch. Der ideale Punkt der Automatisierung befindet sich dort, wo die Summe der Aufwände ein Minimum bildet.

Für eine effiziente Testautomatisierung empfiehlt es sich daher, die Software möglichst früh und in mehreren Teststufen zu testen. Je früher und schneller ein Fehler entdeckt wird, desto einfacher ist es, dessen Ursache ausfindig zu machen. Im Idealfall entdeckt der Entwickler den Fehler sogar noch, bevor der Code eingecheckt wird. Dieser Umstand ist der Kern aller Überlegungen zum automatisierten Testen. Denn geht man davon aus, dass ein auf Stufe Entwicklung gefundener Bug einen Aufwand von 1-2h auslöst, dann kann ein Bug auf der Stufe User Acceptance im finalen Performance Test Audit auch gerne mal 6 Wochen Arbeit für einen erfahrenen Entwickler verursachen.

Die folgende Auflistung von Testarten ist beispielhaft und kann je nach zu entwickelnder Software abgewandelt oder ergänzt werden. Jede Testart hat dabei ein gewisses Mass an Komplexität bezüglich der Automatisierung sowie eine spezifische Aussagekraft und Häufigkeit der Ausführung.

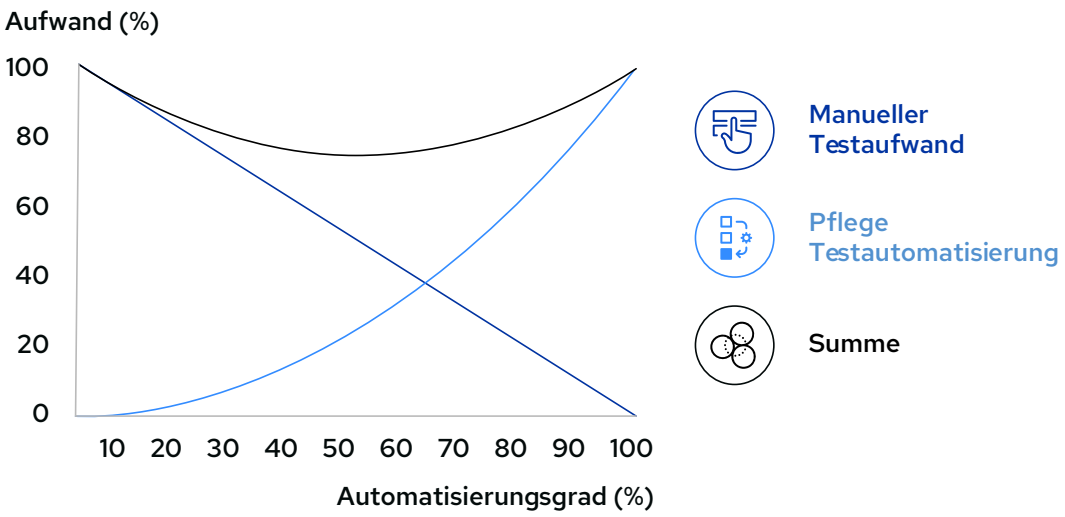

Abb. 1

Testart	Beschreibung
IDE Integrated Testing	Es ist möglich diverse Tools direkt in die Entwicklungsumgebung (IDE) zu integrieren. Dabei verifizieren sie den Code direkt nachdem er geschrieben wurde, vergleichbar mit der Autokorrektur in Word, und geben Feedback über die Code Qualität (z.B. potentielle Gefahren, Endlosschleifen, Exception Handling).
Unit-Test	Die entwickelte Software wird auf der kleinsten Einheit getestet. In der objektorientierten Programmierung sind dies typischerweise Klassen, die eng gekapselt getestet werden. Die Automatisierung ist durch zahlreiche vorhandene Frameworks sehr einfach möglich und durch die gute Integration in moderne IDEs (Integrated Development Environments) sind diese Tests auch einfach lokal ausführbar. Typischerweise ist die Ausführung der Unit-Tests auch ein fester Bestandteil des Build-Prozesses. Sie bilden damit die Basis und dienen als Mittel für schnelles Feedback, wenn sich Fehler in die Software einschleichen. **Komplexität:** Gering, da einfach zu automatisieren und schnell. **Häufigkeit:** Teil des Build-Prozesses sowie jederzeit lokal. **Aussagekraft:** Gering, da das Zusammenspiel mehrerer Units nicht getestet wird.
Integrationstest	In dieser Art von Test werden mehrere Units, die gemeinsam eine komplexere Funktion des Systems erfüllen, gemeinsam getestet. Das Schneiden dieser Tests sowie das automatische Aufsetzen sind hier komplexer. Es muss entschieden werden, welche Units sinnvoll gemeinsam getestet werden und welche umliegende Infrastruktur, wie beispielsweise eine Datenbank, im Test integriert oder simuliert wird. **Komplexität:** Mittel, da entsprechende Integrationseinheiten zusammengesetzt werden müssen. Je nach involvierter Infrastruktur verlängert sich auch die Ausführdauer dieser Tests. **Häufigkeit:** Mehrmals täglich, typischerweise dem Build nachgelagert. **Aussagekraft:** Mittel, da die Funktionalität grösserer Komponenten nachgewiesen werden kann, jedoch der Einfluss der abstrahierten Teile, wie beispielsweise der Infrastruktur, nicht abgedeckt ist.
Systemtest	Das System wird typischerweise von Grund auf installiert, bootstrapped und im Anschluss via seine Schnittstellen wie eine Black Box getestet. **Komplexität:** Hoch, da nicht nur die Installation des Systems sowie dessen Starten und Stoppen sowie das Aufsetzen allfälliger Testdaten automatisiert werden muss, sondern auch die Test-Engine, welche das System von aussen testet. **Häufigkeit:** Mehrmals täglich, idealerweise dem Build nachgelagert. **Aussagekraft:** Hoch, da das System wie aus der Sicht des Endbenutzers getestet wird.

Loadtest	Wenn der Systemtest dies nicht bereits abdeckt, ist zusätzlich ein Loadtest denkbar, der ähnlich wie der Systemtest funktioniert, jedoch das System unter maximaler Last testet. **Komplexität:** Sehr hoch, da hier noch entsprechend produktionsnahe Infrastruktur bereitgestellt werden muss und das Nutzerverhalten im produktiven Bereich simuliert werden muss, um zu einer sinnvollen Aussage zu kommen. **Häufigkeit:** Aufgrund der Laufzeit des Tests sehr unterschiedlich. Regelmässigkeit ist aber dringend empfohlen. **Aussagekraft:** Hoch, da das Lastverhalten des Systems geprüft werden kann.
Kompatibilitätstest	Denkbar beispielsweise bei hochverfügbaren Systemen, die mit einem Blue-Green Deployment installiert werden und jederzeit rückwärtskompatibel sein müssen. **Komplexität:** Sehr hoch, da das System in zwei Versionen gleichzeitig getestet werden muss. **Häufigkeit:** Idealerweise regelmässig aber zumindest vor einem bevorstehenden Deployment **Aussagekraft**: Hoch, in Bezug auf die operationelle Stabilität in der Phase parallel aktiver Versionen.

Die Aufstellung soll zeigen, dass mit zunehmender Aussagekraft der automatisierten Tests auch der Aufwand für die Automatisierung an sich steigt. Daher ist es wie zuvor erwähnt so wichtig, sich an den optimalen Grad der Testautomatisierung

5. Die entscheidende Rolle der Testdaten

Für die Orchestrierung der verschiedenen Teststufen empfiehlt sich der Einsatz einer Pipeline in der Continuous-Integration-Umgebung, die nach und nach die zunehmend komplexeren Tests durchführt und abbricht, sollte ein Fehler auftreten. Das Passieren dieser Pipeline kann damit gleichzeitig als Quality Gate fungieren und weitere Schritte, wie das automatische Deployment auf Teststufen anstossen. Stellt man also die Integrität des Codes sicher - durch ausreichend automatisierte Tests die Funktionalität - dann fehlt trotzdem immer noch mindestens ein Bereich, denn es durch die Pipeline zu automatisieren und zu Standardisieren gilt: die Testdaten.

Die Testdaten spielen für die Testautomatisierung eine wesentliche Rolle.

Die Ausgangslage jedes Testschrittes ist ein vordefinierter Status der Testdaten. Diese Daten entsprechend in den richtigen Status zu versetzen ist also Schritt 1 jedes Testskripts und sollte automatisiert geschehen. Bezgl. Testdaten gibt es grundsätzlich 3 Möglichkeiten für die Bereitstellung. Ein Tooleinsatz ist zwingend notwendig, egal welche Methode man wählt. Hat man den Abzug auf das Testsystem gemacht gilt es noch zwei Dinge zu definieren.

- Wie oft wird ein Neuabzug oder Update der Testdaten gemacht.
- Wie verhindert man die Verschmutzung von anonymisierten oder synthetischen Testumgebungen mit produktiven Daten.

Produktive Testdaten	Im Testumfeld wird ein Auszug aus der Produktion verwendet. Vorteile sind die bestmögliche Aussagekraft der Tests, Nachteile die Datenrisiken und die dadurch notwendige Absicherung der Testumgebung analog der Produktion.
Anonymisierte Testdaten	Können sensible produktive Datenunkenntlich gemacht werden, dann spricht man von anonymisierten Testdaten. Man unterscheidet zwischen weicher und harter Anonymisierung. Bei der weichen Anonymisierung werden Daten, welche die Kunden identifizieren könnten, verfälscht. Durch eine Analyse von Datenmustern (z.B. Transaktionen) könnten aber dennoch Rückschlüsse auf die realen Kunden gezogen werden. Bei der harten Anonymisierung werden zusätzlich auch die Datenmuster verfremdet
Synthetische Testdaten	Synthetische Testdaten lösen das Problem der Anonymisierung. Man erstellt die Testdaten also quasi von Grund auf neu basierend auf produktiven Schlüsselparametern, wie zum Beispiel Last- oder Verhaltensprofilen.

6. Best-Practices

Bei Themen, wie agile und DevOps-Transformation, gibt es viel Interpretationsspielraum. Daher versucht dieses Kapitel das Wesentliche rund um die Testautomatisierung in drei Best-Practices zusammenzufassen:

Keine agile Transformation ohne eine Automatisierung der Code Pipelines

Mit der Methodologie Kanban kann man einen Fix für ein Produktionsproblem jeden Tag verfügbar machen und das Business wird verlangen, diesen Fix so schnell wie möglich auf die Produktion zu liefern. War in der Vergangenheit der gesamte Aufwand für solch ein Deployment von Stufe Entwicklung bis auf die Stufe Produktion angenommen 8 Stunden, hat das sicher nicht jedem gefallen, aber in einem Wasserfallmodell kam das dann vielleicht auch nur ein oder zwei Mal im Monat vor. Benutzt man nun ein agiles Modell wie Kanban und möchte jeden Tag liefern, dann hat man plötzlich nicht 8 oder 16 Stunden Deployment-Aufwand im Monat, nein vielleicht geht das ganze schon in Richtung 160 Stunden und tut schon richtig weh. Daher ist es nach einer agilen Transformation unerlässlich, den Gesamtaufwand der Auslieferung durch jegliche Form der Automatisierung entsprechend effizienter zu gestalten. Man will ja den grösstmöglichen Effizienzgewinn aus einer agilen Transformation ziehen.

Auslieferung von Code-Änderungen müssen in 2 Stunden abgeschlossen sein

Agile Teams verfolgen oft den Ansatz, dass jeder neue Code oder jede neue Funktionalität dem Product Owner auf einer unabhängigen Umgebung präsentiert werden muss. Beim Scrum nennt man diese Stufe “Demo”. Neben den bereits unter
1) gezeigten Herausforderungen ist noch ein weiterer Aspekt interessant: Wie lange darf, angefangen bei einer Code-Änderung eines Entwicklers bis hin zum fertigen Deployment auf solch einer Demo-Umgebung, das Testing, die Paketierung und die Auslieferung solch einer Änderung dauern, bis es dem Entwickler zu lange geht und er die Code-Änderung schnell händisch direkt auf dem Demo System macht?

Entkopplung von umliegenden Systemen mittels intelligentem Mocking-Layer
Punkt 1) in Sachen Automatisierung und Punkt 2) in Sachen Skalierung bringen weitere Anforderungen an meine Testumgebung nahe der Entwicklung: Ein CI-Setup muss vollkommen neue Ansprüche der Skalierung erfüllen. Hinzu kommt die Stabilität (on-the-fly-Updates), aber auch die Entkopplung von externen Systemen (on-the-fly-Mocking). Denn der beste Testautomatisierungsgrad bringt nichts, wenn man nicht sauber mit umliegenden Systemen testen kann, da diese nicht verfügbar genug sind, sondern stattdessen mit nichtssagenden Mockups interagiert. Daher benötige ich eine weitere Schicht zwischen meiner Anwendung und den benötigten Umsystemen, welche es mir ermöglicht auf diese Systeme zuzugreifen, aber im Falle eines Unterbruchs automatisch einen entsprechenden Mockup verfügbar zu haben. Das heisst, selbst wenn ich meine Hausaufgaben in Sachen Testautomatisierung und Schnelligkeit beim Ausliefern erfüllt habe, muss ich noch für die nötige Stabilität des ganzen Setups sorgen.

7. Referenzarchitektur
Vollumfängliche Testkonzepte und Vorschläge oder Definitionen, welche Tools für welche Technologie oder welche Phase im Software Delivery Lifecycle eine gute Idee sind, gibt es mittlerweile zuhauf. Dazu kommt, dass deren Umsetzung ebenfalls schon oft genug durchgeführt wurde und man damit über einen grossen Schatz an Erfahrungen und best-practices verfügt. Wie bei allen anderen Themen zuvor gibt es nicht die eine Lösung. Es ist auch durchaus eine gute Idee das aktuelle Setup vor jedem neuen Programm Increment zu besprechen, ob vielleicht für die nächste Lieferung Anpassungen in der Architektur oder in den Verhaltensregeln notwendig ist.

Nötige Verhaltensregeln, einzelne Tools und Guidelines, sollten in eine Referenzarchitektur gegossen werden und entsprechend allen Beteiligten geläufig sein. Das kann wie angesprochen einen Auffrischungskurs vor jedem neuen Programm-Inkrement bedeuten.

Lifecycle view

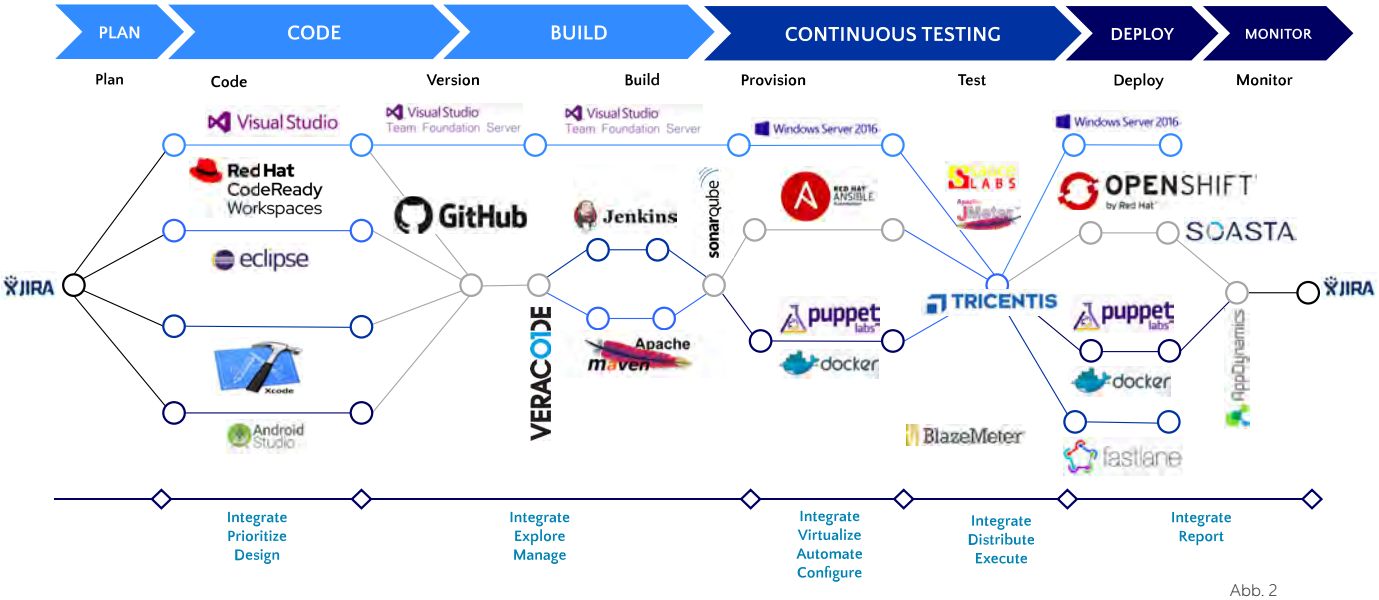

Abb. 2

8. Tools

Verschiedene Tools können das Testing erleichtern. Eine beispielhafte Auswahl ausgebreitet auf den Software Delivery Lifecycle zeigt die Abbildung Nr. 2:

Folgende Kernaspekte gilt es in diesem Graph hervor zu heben:

- Zur effizienten Koordinierung des Defect Management Prozess ist es wichtig eine durchgängige Lösung zu verwenden und dies bereits ab der Stufe CI
- Es muss möglich sein, mit dem Toolstack eine end to end Automatisierung zu erreichen. Sollte in der Kette ein Framework oder Tool zum Einsatz kommen, welches manuelle Schritte in die Steuerung oder den Betrieb benötigt, ist dies sehr kritisch zu hinterfragen
- Je mehr Produkte zum Einsatz kommen, die untereinander sich in ihrer Effizienz ergänzen, desto besser ist nachher meine Gesamtperformance. Es gibt Produkte, die sowohl Automatisierung, Hosting von Hilfskomponenten, aber auch die Plattform direkt stellen und damit eine drastische Reduzierung des gesamtheitlichen Computes erreichen und damit eine Senkung des Total Cost of Ownership (TCO) des gesamten Setups.
- Je mehr Integration mit anderen Teams, umso besser. Zum Beispiel gibt es mittlerweile Workspace Factory Link Konzepte (Stichwort CodeReady Workspaces CRW), die folgenden Use-Case ermöglichen:

  1. Ein automatisierter Test auf Stufe CI schlägt fehl
  2. Es wird automatisch ein Defect erstellt
  3. In der Defect Beschreibung ist nicht nur ein Bilder GUI zum Zeitpunkt des Fehlers, Stacktrace, Testdaten, es ist auch ein Link in der Beschreibung, mit welchem der Entwickler automatisch genau den Workspace mit genau den Code Versionen aufstarten kann, welche er benötigt, um mit der Fehleranalyse zu starten

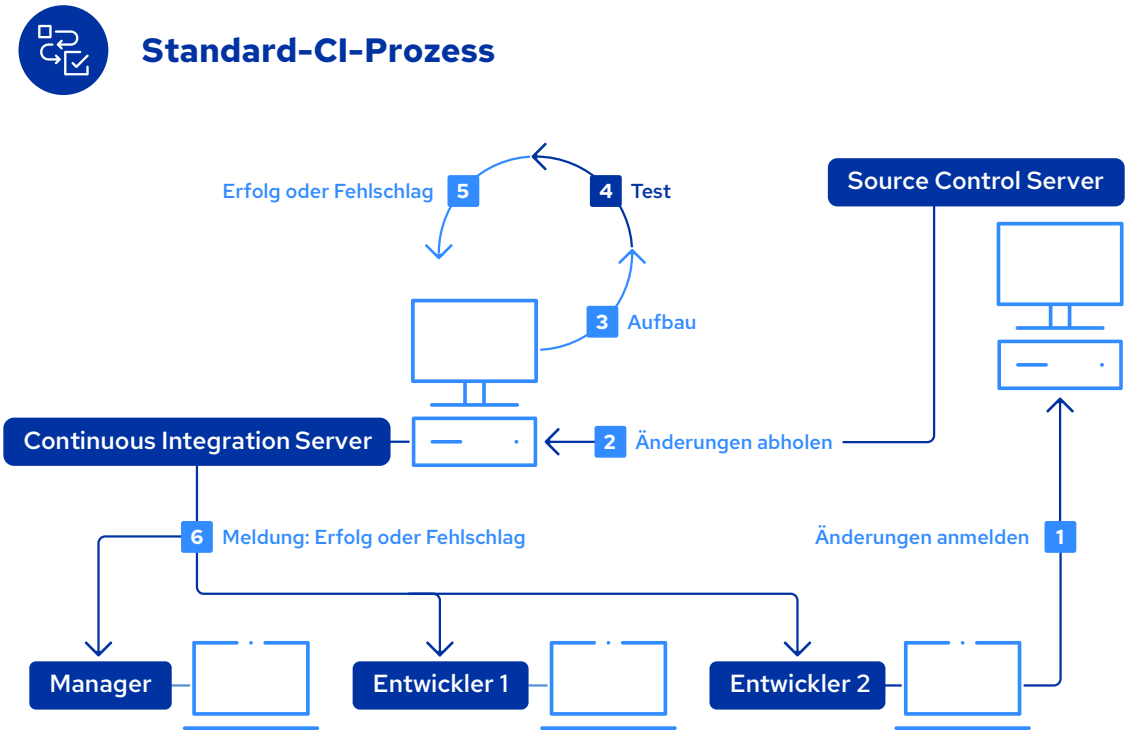

Abb. 3

9. Beispiel: Applikations-Infrastruktur
Zu guter Letzt wollen wir noch einen Blick auf eine Beispielarchitektur werfen. Hier geht es darum, wie man verschiedene Umgebungen am besten miteinander verdrahtet, um das höchste Mass an Effizienz zu gewinnen, sowohl im Hosting aber auch bei der Liefergeschwindigkeit (Time to Market).

Abbildung 3
Nehmen wir einen Standard-CI-Prozess. Ein Entwickler eines der zuvor geschilderten agilen Teams checkt eine Codeänderung auf das zentrale Repository ein (1) und löst damit den Build eines neuen Applikationspaket aus (2 und 3). Dieses muss auf dem CI mittels automatisierten Tests verifiziert werden. Hier gibt es direkt fünf Haupt-Herausforderungen:

- Eine Daumenregel aus dem Kapitel Zielsetzung zuvor sagte uns ja schon: Die Zeit von der Codeänderung bis diese neue Applikationsversion auf der Demo-Umgebung läuft, sollte unter 2 Stunden liegen. Dauert es länger, kommt es mit der Zeit zu vielen Änderungen, die nicht in der Version-Control sind.

- Damit man so schnell sein kann, müssen sowohl die Testnodes als auch der Test-Applikationsstack in hohem Masse skalierbar sein und das ist alles andere als ein Standardvorgehen beim Capacity Planning auf Stufe Dev. Konkret müsste nicht nur meine Anwendung für Stufe Demo, sondern auch das gesamte Testframwork am besten auf einer Container Platform laufen, um damit an die nötigen Skalierbarkeits Anforderungen zu erfüllen.

- Automated Testing startet auf Stufe CI, sollte das Regression-Testing zusammen mit den automatisierten Tests auf Stufe SIT stattfinden, muss man damit rechnen, dass nach jedem SIT Deployment ein nicht unerheblicher Aufwand in das Maintenance der Testcases geht. Um einen starken Shift-Left zu erreichen, ist eine konsequente Form der Service Entkopplung notwendig. D.h. ich muss auf Stufe CI in der Lage sein, meine Umsysteme einzubeziehen, was klar in Richtung System Integration Testing geht. Jedoch darf es mich nicht aufhalten, wenn eines der Umsysteme aktuell nicht verfügbar ist. Das geht über ein intelligentes Mockinglayer (vergleiche Kapitel Zielsetzung).

  Jedoch muss so ein Konzept konfigurierbar bleiben. Zum Beispiel folgender Fall: ein GUI Service ist nicht verfügbar und ich will Regressionstests des zugehörigen Backends machen. Das Mockinglayer dürfte nun den Ausfall auf der GUI automatisch kompensieren, um mir den Backend Tests durchzuführen, selbst wenn die Teststufe SIT Level wäre. Sollte jedoch ein Testfall so markiert sein, dass scharf mit allen notwendigen externen Services getestet werden soll, dann muss die nicht Verfügbarkeit eines Umsystems in einen Fehler resultieren.

- Testdaten müssen zentral verwaltet werden. Es ist nicht unüblich, dass sich diese auf der Stufe UAT zu CI/SIT unterscheiden, da sie je später, desto ähnlicher zu den Kundendaten werden und nicht selten auf Stufe CI und SIT synthetisiert sind. Jedoch ist gerade die Synchronisierung zwischen CI und SIT daher empfehlenswert, um beim SIT Regressions-Testing so sicherzustellen, dass der Defect als Ursache nicht die Testdaten hat.

- Ist ein Business-Case rund um das Thema Testautomatisierung einmal negativ, sollte man einen Blick auf den Scope werfen. Denn betrachte ich diese Angelegenheit holistisch, dann ist das grösste Potential jeglicher Optimierung einer IT-Organisation das Testing und die Sicherstellung der Funktionsweise des entsprechenden Testframeworks und es ist nicht immer einfach geeignete KPIs zu finden, die diesen Sachverhalt zielführend reflektieren.

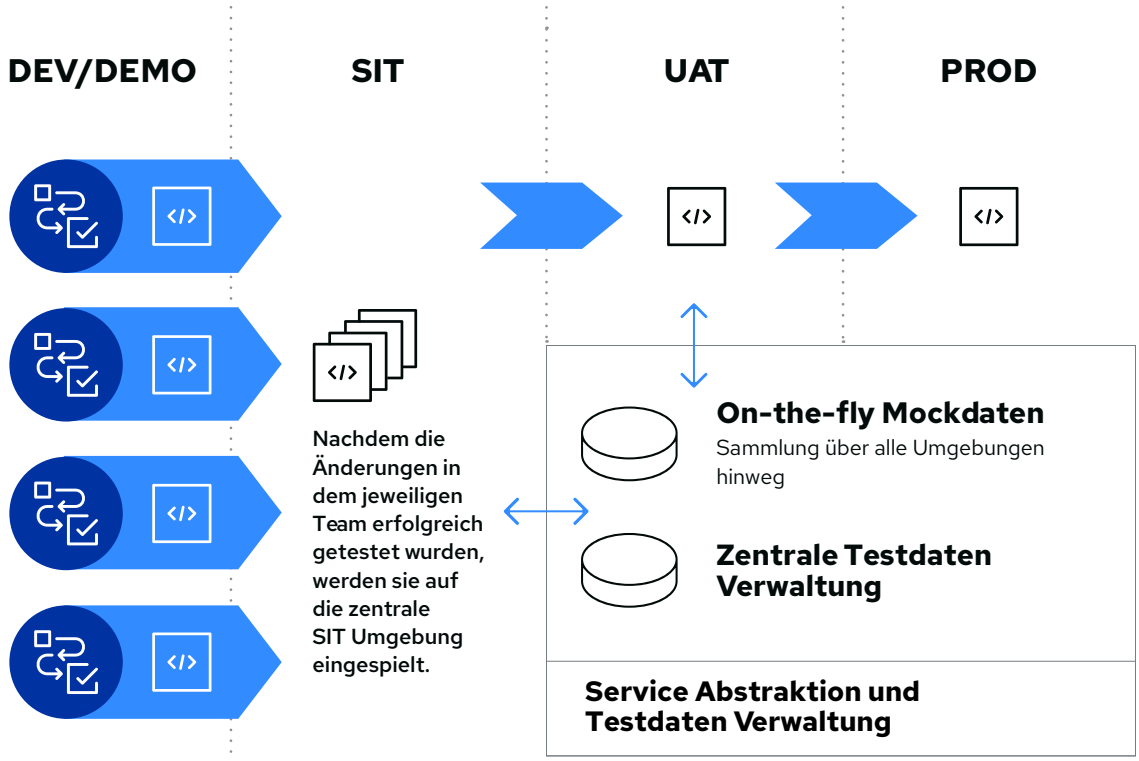

Abb. 4

Abbildung 4
Im weiteren Voranschreiten durch den Software Delivery Lifecycle, kommt die Frage auf, inwieweit ich meine einzelnen Entwicklungsteam voneinander entkoppeln muss, aber trotzdem so stark wie möglich synchron halte. Natürlich ist das nur eine Herangehensweise, abhängig von der Team Disziplin ist es auch möglich diese Schicht zu überspringen und auf einem einzigen gemeinsamen CI-Setup zu laufen. Je nach Setup können hinter z.B. SAFe aber auch mal gerne 20 Scrum-Teams stehen und dann wird es doch sehr unwahrscheinlich, dass man sich da nie ins Gehege kommt.

Jedes Team verbindet seine eigene CI-Umgebung mit der zentralen SIT-Umgebung. Es macht hier wenig Sinn die eigene Demo-Umgebung mit der der anderer Scrum-Teams zu verbinden. Diese sind ja selber andauernd im Refresh bei jeder Code-Änderung und daher müsste man noch weiteren Aufwand betreiben einen Updatemechanismus umzusetzen, der zero Downtime-Updates ermöglicht (auch eine mögliche Variante mit der richtigen Container-Platform). Ansonsten wäre aufgrund der Wechselwirkungen das end-to-end Testing mit den Umsystemen nur erschwert möglich.

Erst wenn eine Änderung sauber auf der eigenen Demo-Umgebung verifiziert wurde, darf diese auf die SIT Umgebung eingespielt werden. Man kann sogar zusätzlich in Maintenance-Windows planen, sollten zu viele externe Abhängigkeiten zu dieser Umgebung bestehen, um die Downtime noch zusätzlich zu managen.

Jeder externe Serviceaufruf läuft über eine zentral verwaltete Service Schicht. Das kann man sich so vorstellen: Anwendung A braucht für einen Business Service X einen externen Service der Anwendung B. Im Normalfall ruft Anwendung A über eine Serviceschicht die Anwendung B auf und erhält die nötige Antwort. Die Serviceschicht speichert automatisch dieses Aufruf-Antwort Pärchen. Im Falle, dass Anwendung B nicht erreichbar ist, spielt die Serviceschicht im Autofailover einfach ab, was vorher aufgenommen wurde. Damit erreicht man ein sehr realistisches Testsetup, was aber dennoch genügend entkoppelt ist, dennoch sich sehr realitätsnah verhält und somit massiv Vorteile gegenüber einem händischen Mock Konzept mit sich bringt. Dieses ist meistens dann doch zu weit weg vom Status Quo, um nicht regelmässig zu grösseren Missverständnissen und Mehraufwand zu führen.

Wichtig ist auch der Inhalt einer Baseline. Diese sollte mindestens immer beinhalten:

- Code
- Daten (etwa Datenbank-Spezifikation, Strukturen, etc) Testcases / Testklassen Testdaten
- API-Konfiguration (was brauche ich, um mich mit meinen Umsystemen zu verbinden) Environment-Konfiguration (die Spezifikation, wie die Umgebung aussehen soll und aufgebaut werden muss)
- Konfigurationsdatei für das Workspace Setup (siehe Kapitel Tooling, CRW)

Solch ein Setup wird relativ schnell sehr komplex, wenn man die verschiedenen Anwendungen nicht durch eine Containerplatform abstrahiert. Das heisst, die verschiedenen Anwendungen werden nicht auf den physischen oder virtuellen Servern direkt installiert, sondern auf einer Zwischenschicht, der Containerplatform. Wählt man hier das richtige Produkt, kommen Dinge, wie: automatische Skalierung, Datenmanagement, Security, Containerrepository, CI, API (Schnittstellenmanagement zu den Umsystemen), Business Continuity Plan, Storage (automatisiertes Kostenmanagement rund um jede Stage) und entsprechende Pipelines, Servicekataloge out of the box mit.

Hier rentiert es sich gegebenenfalls sogar alles mit einem Hersteller abzudecken, da z.B. ein API Management vom selben Hersteller, wie die Containerplatform viel einfacher installiert, gewartet und auch kostengünstiger gehostet werden kann.

Jedoch sollte man immer auf die nötige Abstraktion zu den verschiedenen Vendoren achten. Ist man einmal in einen starken Vendor-Lockin gelaufen und merkt dann, dass man mit dem aktuellen Lösungsansatz nicht weiter kommt, kann es zu massiven Transformationskosten kommen.

Wichtig ist ebenfalls, dass die Automatisierung das Change-Management mit einschliesst. Denn nicht nur das automatisierte Bauen, Verteilen und Testen ist wichtig, sondern ebenso das Erfassen aller Änderungen im Vergleich zur vorherigen Version. Das ist nicht nur wichtig, um die Effizienz der Entwicklerteams bei der Fehleranalyse zu sichern, es kann ebenfalls ein zwingender Aspekt beim Einhalten von Policies oder ISO-Normen sein.

10. Fazit: Keine Agilität ohne Automatisierung

Die Automatisierung von Prozessen wie das Bauen, Testen und Ausliefern von Anwendungen oder das Zusammenwachsen von Dev und Ops sind auch ohne die Einführung von agilen Methoden möglich. Für eine erfolgreiche agile Transformation sind diese Punkte jedoch eine Voraussetzung, da ein agiles Vorgehen nur mit einem hohen Grad an Automatisierung effizient möglich ist.

Eine wirkliche Automatisierung im Software Delivery Lifecycle geht nicht ohne automatisiertes Testen. Hier heisst die Devise (immer abhängig von der Kosten/Nutzenabschätzung): so früh und so weitreichend wie möglich end-to-end zu testen, am besten bereits auf Stufe Dev/CI. Teams mittels Serviceabstraktion zu entkoppeln ist hier genauso eine vorbereitende Massnahme für einen effizienten automatisierten Testansatz, wie die Überführung der Umgebungen auf eine Containerplattform bis hin zur kompletten Abstraktion der Infrastruktur als Code.

Im Grunde sind der Automatisierung rund um das Ausliefern von Software keine Grenzen gesetzt, jedoch muss alles entsprechend finanziell einen Nutzen haben. Die Einführung einer automatisierten Teststrategie gelingt meistens auch ohne signifikantes Einmalinvestment oder laufende Kosten, jedoch nicht ohne Priorisierung der Lieferungsverantwortlichen (Stichwort Purchase Order). Jedoch stehen sämtliche Aufwände in keiner Relation zum manuellen Testen. Bei entsprechender Priorisierung durch agile Teams parallel zum Daily Business, ist zumindest die Maintenance kostenlos machbar. Auf längere Sicht geht es jedoch ohne eine Automatisierung des Testings nicht. Organisationen, die grösstenteils manuelle Tests fahren, werden mit der Zeit immer mehr in einen Konkurrenzkampf geraten, den sie nicht gewinnen können.

Themen, wie agile Transformation, DevOps oder Testautomatisierung sind mittlerweile nicht mehr neu. Es findet sich leicht ein Partner, der vielleicht sogar schon einige Transformationen hinter sich hat, genau weiss, wann das strikte Einhalten von Prozessen und Vorgaben laut Literatur Sinn macht und wann die kreative Interpretation besser ist, um mal einen Erfahrungsaustausch zu suchen. Einer konsequenten agilen Transformation, bei der die Qualität der Ergebnisse nicht auf der Strecke bleibt, steht so nichts mehr im Weg.

Autoren

Michael Siebert
https://www.linkedin.com/in/michael-siebert
Senior Solution Architect bei Red Hat

Pre-Sales Lead für das Enterprise Consulting Business in der Schweiz und die Kooperationen zu AWS, Azure und GCP. Über 16 Jahre Erfahrung im Bereich DevOps und agile Methodiken.

Timo Pfahl
https://www.linkedin.com/in/timopfahl
Head IT SIX Interbank Clearing AG

Leiter der IT-Abteilung, welche die für den Finanzplatz Schweiz kritische Infrastruktur SIC nach agilen Methoden entwickelt und betreibt.

Robert Bonomo
https://www.linkedin.com/in/robertbonomo
Unternehmensbereichsleiter Qualitätssicherung Cognizant für D-A-CH und Osteuropa

Robert ist ein ausgewiesener Fachexperte im Bereich Entwicklungsmethoden, Qualitätssicherung und Test-Automatisierung und verfügt neben diversen Fachzertifikaten über Abschlüsse in Wirtschaftsinformatik und Betriebswirtschaft. Robert ist zudem gewählter Prüfungsexperte für die eidgenössischen Lehrabschlussprüfungen in Applikationsentwicklung.

Über Cognizant

Cognizant (Nasdaq-100: CTSH) ist ein weltweit führendes Professional-Services-Unternehmen, das für seine Kunden Geschäfts-, Betriebs- und Technologiemodelle für das digitale Zeitalter entwickelt. Unser einzigartiger, auf Branchenanforderungen zugeschnittener Beratungsansatz unterstützt Kunden bei der Planung und Umsetzung innovativer und effizienter Geschäftsvorhaben. Cognizant hat seinen Hauptsitz in den USA, rangiert auf Platz 185 der Fortune 500 und zählt zu den renommiertesten Unternehmen der Welt. Erfahren Sie auf www.cognizant.com, wie Cognizant die Digitalisierung seiner Kunden vorantreibt, oder folgen Sie uns @Cognizant.

Cognizant

World Headquarters
300 Frank W. Burr Blvd.
Suite 36, 6th Floor
Teaneck, NJ 07666 USA
Phone: +1 201 801 0233
Fax: +1 201 801 0243
Toll Free: +1 888 937 3277

European Headquarters
1 Kingdom Street
Paddington Central
London W2 6BD England
Phone: +44 (0) 20 7297 7600
Fax: +44 (0) 20 7121 0102

India Operations Headquarters
#5/535 Old Mahabalipuram Road
Okkiyam Pettai, Thoraipakkam
Chennai, 600 096 India
Phone: +91 (0) 44 4209 6000
Fax: +91 (0) 44 4209 6060