Qualysoft

WHITEPAPER

STANDARD DELIVERY PIPELINE

www.qualysoft.com

Continuous Delivery

Qualysoft

Bei Continuous Delivery eines Software-Projektes geht es darum, über den gesamten Projektlebenszyklus ständig lauffähige Versionen zur Verfügung zu stellen. Eine weitgehende Automatisierung ermöglicht dabei, ohne Zusatzaufwand der Entwickler immer eine aktuelle Version für Stakeholder zugänglich zu machen. Continuous Delivery ist eine perfekte Ergänzung zur agilen Arbeitsweise, ist aber auch bei klassischer Entwicklung eine große Hilfe für das Team.

Continuous Delivery Pipeline

Eine Continuous Delivery Pipeline ist eine individuelle Abbildung des verwendeten Software-Entwicklungsprozesses. Sie bietet mehrere Vorteile, u.a.:

- **Übersicht:** Gemeinsames Verständnis für Projekt-Teams und Stakeholder über Ablauf, Entwicklungsstand und verfügbaren Versionen
- **Standardisierung:** Einheitliche Software-Qualität auf den einzelnen Stufen der Pipeline
- **Automatisierung:** Die meisten Pipeline-Schritte lassen sich komplett automatisieren
- **Feedback:** Hilft Entwicklern dabei, Fehler so früh wie möglich zu beheben

Ablauf

Jede erfolgreiche Änderung muss die gesamte Pipeline durchlaufen. Nach dem Build wird die Software dabei auf unterschiedliche Umgebungen deployed, um dort getestet zu werden. Jedes Deployment ist voll automatisiert und setzt die jeweilige Umgebung zuerst auf einen definierten Stand zurück. Quality Gates definieren dabei Kriterien, mit deren Erfüllung erst auf die nächste Umgebung deployed wird.

Schlagen Unit Tests, Integration Tests oder Quality Gates fehl, wird die Pipeline unterbrochen. Dadurch bleiben spätere Umgebungen weiterhin mit einer älteren Version verwendbar, trotz etwaiger Fehler auf früheren Umgebungen.

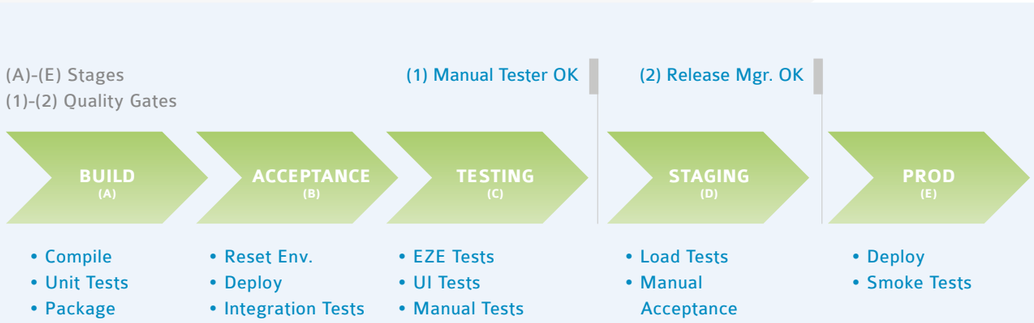

Abbildung 1: Stufen, Schritte und Quality Gates einer allgemeinen Pipeline

System

Die Pipeline wird von einem zentralen Dienst bereitgestellt, der den gesamten Prozess automatisiert und visualisiert. Dieser Dienst wartet auf Änderungen im Source Repository, verwendet die Build Agents zum Kompilieren, legt die erzeugten Software-Artefakte ab, startet Deployments auf den verschiedenen Umgebungen und sammelt die Test-Ergebnisse.

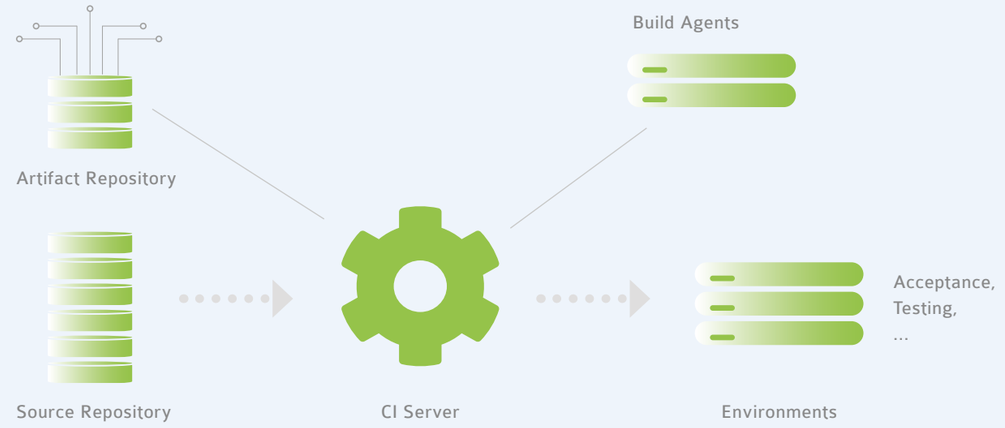

Abbildung 2: Allgemeine Komponenten einer Continuous Delivery Pipeline

Bestandteile

Zu jeder Stufe der Pipeline gehören Schritte, eine Umgebung, und ein Datenstand.

Stufen (Stages)

Build: Baut alle Module, führt Unit Tests aus, und erstellt ein installierbares Paket.
Build Agents stellen dafür alle nötigen Build-Tools bereit.

Acceptance: Installiert die Anwendung in die einfachste mögliche Umgebung und führt Integration Tests aus.
Als Abhängigkeiten stellt diese Umgebung statt komplexer/teurer Services (z.B. Zahlungsdienste, Industrieanlagen) virtuelle Services bereit, diese sind durch das Team kontrollierbar. Ein minimaler Datenstand ermöglicht das Testen der einzelnen Funktionen.

Testing: Installiert die Anwendung und führt End-to-End Tests und UI Tests aus.
Diese Umgebung ermöglicht den Zugriff für manuelle Tester. Geben diese ihr OK, läuft die Pipeline weiter.

Staging: Installiert die Anwendung und führt Lasttests aus.
Diese Umgebung besitzt der Produktion ähnliche Hardware-Spezifikationen und einen ähnlichen Datenstand, um diese Tests realitätsnah durchführen zu können.

Prod: Installiert die Anwendung in Produktion.
Smoke-Tests stellen sicher, dass einige häufig verwendete Funktionen verfügbar sind und die Anwendung damit lauffähig ist.

Umgebungen (Environments)

Jede Umgebung stellt die aktuelle Software-Version für eine bestimmte Zielgruppe und einen bestimmten Zweck zur Verfügung. Die gleiche Software soll mit geringen Änderungen der Konfiguration auf jeder Umgebung lauffähig sein. Umgebungen können auf physischen wie virtuellen Maschinen beheimatet sein.

- **Build:** Nur für die Automatisierung; beinhaltet alle für das Kompilieren notwendigen Tools
- **Acceptance:** Für Entwickler; stellt sicher, dass die einzelnen Komponenten der Software ihre technischen Ziele erfüllen
- **Testing:** Für manuelle Tester; ermöglicht normale Benutzer-Szenarien manuell durchzugehen
- **Staging:** Für Product Owner und Release Manager; ermöglicht Lasttests und Abnahmetests
- **Prod:** Für Benutzer; stellt die Software für die eigentlichen Benutzer zur Verfügung

Jede Umgebung beinhaltet alle für den Betrieb der Software notwendigen Abhängigkeiten. Zur Vereinfachung oder Kostenreduktion können Abhängigkeiten dabei auch als virtuelle Services bzw. Service-Stubs ausgeführt sein.

Steuerung und Visualisierung

Normalerweise wird die Pipeline automatisch durch Änderungen im Source Repository gestartet. Zusätzlich kann man für Testzwecke über ein Web-Interface auch manuell eine beliebige Version auf eine gewünschte Umgebung deployen.

Das Web-Interface stellt weiterhin den aktuellen Pipeline-Zustand, inklusive etwaiger fehlgeschlagener Tests sowie dem Versionsstand auf den einzelnen Umgebungen, dar.

Abbildung 3: Versionsübersicht auf verschiedenen Umgebungen

Im Beispiel in Abbildung 3 sieht man, dass die aktuelle Entwicklerversion 1.0.3 die Tests auf der Acceptance-Umgebung nicht bestanden hat. Deswegen bleiben die späteren Umgebungen auf ihrer alten Version und sind weiterhin funktionsfähig. Die Version 1.0.2 wartet hingegen gerade auf die Abnahme. Die Produktion wird noch durch die alte Version 1.0.1 bedient.

Pipelines für Microservices

Besteht die Anwendung aus mehreren separat deploybaren Services, liegt es nahe, diese jeweils mit einer separaten Pipeline auszuliefern. Das hilft dabei, die einzelnen Stufen möglichst schnell ausführbar zu halten. Bei jedem Service-Deployment werden dabei nur die für das Service relevanten Tests ausgeführt.

Für eine hohe Anzahl an Microservices in einer komplexen Anwendung ist es unerlässlich, die Pipeline für alle Services identisch zu halten. Erst die Standardisierung der Services ermöglicht die problemlose service-übergreifende Beobachtbarkeit und Kontrollierbarkeit.

In einem typischen Szenario gibt es dafür ein oder mehrere parametrierbare Pipeline- Templates, welches alle prozeduralen Schritte beinhaltet. Jedes Service stellt dann nur einige wenige deklarative Parameter (u.a. Pipeline-Template, Artefakt-Name, Ziel-Umgebungen) individuell ein.

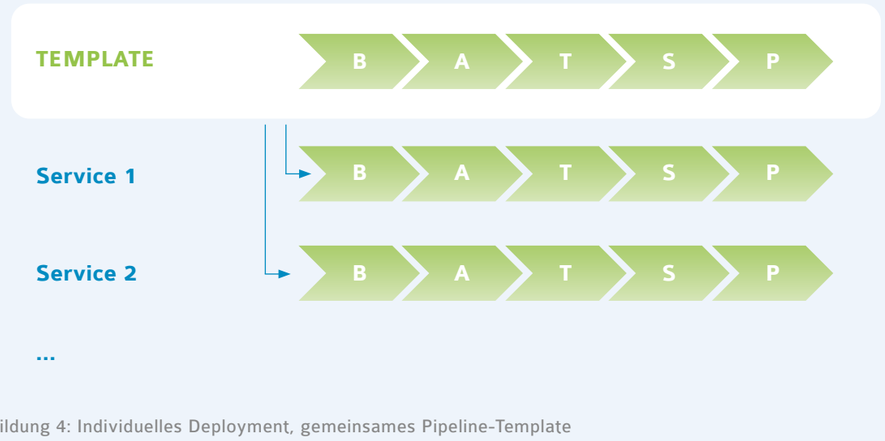

Abbildung 4: Individuelles Deployment, gemeinsames Pipeline-Template

Jedes Service durchläuft unabhängig von den anderen seine eigene Pipeline, eine Änderung wird nacheinander auf die einzelnen Umgebungen deployed. Bei neuen Features mit Service-Abhängigkeiten laufen die Tests auf einer Umgebung erst durch, wenn alle Abhängigkeiten ebenfalls deployed worden sind.

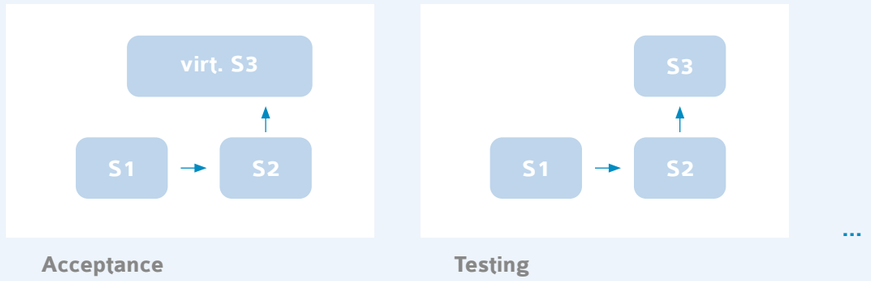

Abbildung 5: Umgebungen mit Microservices. Nicht alle müssen auf jede Umgebung deployed werden

Planung

Soll eine Pipeline von mehreren Entwicklerteams verwendet werden, ist es sinnvoll, im Vorfeld die Anforderungen der Teams zu erheben und ein gemeinsames Pipeline-Konzept zu erstellen. Dies hilft dabei, die gemeinsamen Teile der Pipeline für alle verwendbar zu gestalten und mehrfache Arbeiten in den einzelnen Teams zu vermeiden.

Bei wesentlich größeren Vorhaben ist ein Berechtigungskonzept unerlässlich. Dieses definiert, welche Mitarbeiter-Rollen welche Teile der Pipeline verändern dürfen. Berechtigungen reduzieren Nebeneffekte, die Pipeline-Änderungen auf andere Teams haben können, und helfen dabei, ein übermäßiges Abdriften der Pipeline-Implementierung vom Gesamtkonzept zu vermeiden.

Implementierung und Qualitätskriterien

Wie eine konkrete Pipeline-Umsetzung letztendlich ausfällt, wird von folgenden Aspekten wesentlich beeinflusst. Sie lassen sich nach den Anforderungen an die Pipeline gruppieren.

Performance

Je schneller Entwickler Feedback bekommen, desto effektiver ist ihre Arbeit.
Bei einer konkreten Pipeline ist die Performance abhängig von:

- **Software-Komplexität:** Je komplexer die Software, desto länger dauern Build und Tests im Allgemeinen. Sind das mehr als etwa 10 Minuten, verlangsamt das die Entwicklung [1]. Aber: Der Aufwand zum Kompilieren lässt sich meist verteilen, und Tests sind meist ebenfalls parallelisierbar. Schließlich kann man auch die Software in Module unterteilen, um dafür getrennte Pipelines zu erstellen.

- **Software-Architektur:** Bei monolithischer Software wird typischerweise ein Build-Tool aufgerufen, welches alle Abhängigkeiten einzeln kompiliert und zusammenfügt. Auch kleine Änderungen verursachen dann einen kompletten Rebuild und das Ausführen aller Tests. Eine Microservice-Architektur lässt sich hingegen vom Design her natürlich in einzelne Service-Pipelines unterteilen. Builds und Tests sind Service-spezifisch und laufen deshalb zügiger durch.

- **Hardware:** Sollte so dimensioniert sein, dass sich die durch geänderte Arbeitsweise häufigeren Builds aller Projekte bzw. Services darauf ausgehen. Bei Projekten in der Public Cloud kann dabei beispielsweise auch bedarfsabhängig skaliert werden, indem automatisch weitere Build Agents erstellt werden.

[1] Martin Fowler: "Continuous Integration", Artikel auf https://www.martinfowler.com/articles/continuousIntegration.html

Wartbarkeit

Eine gute Pipeline trifft aus Entwicklersicht das ideale Verhältnis zwischen prozeduralem Code und deklarativer Konfiguration. Wichtige Einflussfaktoren sind dabei:

- **Anzahl der Projekte oder Services mit unterschiedlichen Technologien:** Für jede Basis-Technologie (Programmiersprache/Framework) wird eine eigene Variante der Standard-Pipeline benötigt. Spätere Anpassungen betreffen alle durch die Pipeline-Variante bedienten Projekte/Services.

- **Komplexität der Umgebungen:** Ist das Deployment mittels einfacher Scripts ausreichend wartbar, oder sind Deployment-Tools mit einer statischen Umgebungskonfiguration besser geeignet, um alle notwendigen Abhängigkeiten der Software entsprechend abzubilden?

Dokumentation

Das Anlegen neuer Projekte sowie Änderungen an Umgebungen sollte durch die Dokumentation so weit unterstützt werden, dass sich Entwickler bzw. Administratoren schnell in der Konfiguration der Pipeline zurechtfinden.

FAZIT

Eine Continuous Delivery Pipeline hilft dabei, durch Automatisierung und Standardisierung des Software-Entwicklungsprozesses eine einheitliche Software-Qualität zu schaffen. Ständige Verfügbarkeit lauffähiger Versionen sowie die frühzeitige Erkennung von Fehlern stehen dabei im Mittelpunkt.

Die meisten Fehler werden bei diesem Ablauf durch automatische oder manuelle Tests auf separaten Umgebungen erkannt und noch vor dem Deployment in Produktion behoben. Stakeholder haben immer aktuelle Versionen zum Testen, ihre frühzeitigen Rückmeldungen steigern die Produktivität der Entwickler.

Eine Continuous Delivery Pipeline sollte den Gegebenheiten eines Unternehmens und dessen konkreten Projekten entsprechen, damit sie wartbar und performant bleibt. Beim Design und Aufbau einer Pipeline werden viele Themen angesprochen, welche auch für die Qualität der Anwendung selbst entscheidend sind.