



Fakultät Wirtschaft

Studiengang Wirtschaftsinformatik Software Engineering

**Integrations- und
Bereitstellungsautomatisierung von
Cloud-Anwendungen für
Composable-Enterprise-Architekturen im
Kontext der SAP Business Technology Platform**

Bachelorarbeit

Im Rahmen der Prüfung zum Bachelor of Science (B. Sc.)

Verfasser:	Rafael Martin
Kurs:	WI SE-B 2020
Dualer Partner:	SAP SE, Walldorf
Betreuer der Ausbildungsfirma:	Klaus Räwer
Wissenschaftlicher Betreuer:	Herr Ulrich Wolf
Abgabedatum:	08.05.2023

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Bachelorarbeit mit dem Thema:

**Integrations- und Bereitstellungsautomatisierung von
Cloud-Anwendungen für Composable-Enterprise-Architekturen im
Kontext der SAP Business Technology Platform**

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mannheim, jdd.mm.yyyy,

Rafael Martin

Inhaltsverzeichnis

Selbstständigkeitserklärung	II
Abkürzungsverzeichnis	V
Abbildungsverzeichnis	VI
Tabellenverzeichnis	VII
1 Einleitung	1
1.1 Motivation und Problemstellung	1
1.2 Zielsetzung und Abgrenzung	1
1.3 Aufbau der Arbeit	1
2 Grundlagen und Begriffserklärungen	2
2.1 Composable-Enterprise-Architektur	2
2.1.1 Begriffserklärung und Abgrenzung	2
2.1.2 Technologische Konzepte des Composable-Enterprises	2
2.2 Integrations- und Bereitstellungsautomatisierung von Software	3
2.2.1 Agile und DevOps als Anwendungsentwicklungskonzepte	3
2.2.2 CI/CD-Pipelines zur Softwarebereitstellung	6
2.2.3 Strategien zur Bereitstellung von Neuentwicklungen	11
3 Methodische Vorgehensweise	14
3.1 Prototypische Entwicklung der CI/CD-Pipelines	14
3.2 Evaluation der CI/CD-Pipelines unter Anwendung des Analytischen Hierarchieprozess	14
3.3 Semistrukturierte Leitfadeninterviews	14
4 Anwendung der Methodik auf die theoretischen Grundlagen	15
4.1 Prototypische Entwicklung der CI/CD-Pipelines	15

4.2	Evaluation der CI/CD-Pipelines unter Anwendung des Analytischen Hierarchieprozess	15
4.3	Entwicklung einer ganzheitlichen Bereitstellungsstrategie	15
5	Schlussbetrachtung	16
5.1	Fazit und kritische Reflexion	16
5.2	Ausblick	16
	Literatur	VIII
	Anhang	IX

Abkürzungsverzeichnis

Abbildungsverzeichnis

1	Exemplarische Abfolge eines agilen Entwicklungszykluses	3
2	Zeitliche Darstellung des Kundennutzen von IT-Services	5
3	Aktivitäten im CI/CD-Prozess	7
4	Versionskontrollsysteme zur Verwaltung von Quellcode	8
5	Hierarchische Darstellung von Softwaretests	9
6	Strategien zur Bereitstellung von Software	12

Tabellenverzeichnis

1 Einleitung

1.1 Motivation und Problemstellung

1.2 Zielsetzung und Abgrenzung

1.3 Aufbau der Arbeit

2 Grundlagen und Begriffserklärungen

2.1 Composable-Enterprise-Architektur

2.1.1 Begriffserklärung und Abgrenzung

2.1.2 Technologische Konzepte des Composable-Enterprises

2.2 Integrations- und Bereitstellungsautomatisierung von Software

2.2.1 Agile und DevOps als Anwendungsentwicklungskonzepte

Das Hauptaugenmerk eines Composable-Enterprises besteht darin Resilienz und Flexibilität aufrecht zu erhalten. Damit soll sichergestellt werden, dass IT-Leistungen in einem sich stetig ändernden Umfeld schnell und risikoarm bereitgestellt werden können. Das traditionelle Wasserfallmodell, welches eine sequenzielle Abfolge der Projektelemente *Anforderung*, *Design*, *Implementierung*, *Test* und *Betrieb* vorgibt, besitzt dabei signifikante Limitationen. Die in dieser Methodik detailliert durchgeführte Vorabplanung, kann in der Realität aufgrund unvorhersehbarer Externalitäten selten eingehalten werden. Auch die starre Abfolge der Projektphasen mindert besonders zu fortgeschrittenen Zeitpunkten des Vorhabens den Spielraum für Anpassungsmöglichkeiten [9][5]. Dies resultiert nicht nur in einem Anstieg der Kosten sondern führt ebenfalls dazu, dass IT-Projekte länger als geplant ausfallen [8][41]. Als Reaktion haben sich agile Vorgehensmodelle zunehmend innerhalb der Projektmanagementlandschaft etabliert. Im Gegensatz zum Wasserfallmodell, welches eine umfassende Vorab-Planung vorsieht, wird das Vorhaben in einer agilen Entwicklung in viele Teilprojekte, sog. *Sprints*, segmentiert (s. Abb. 1) [4][87].

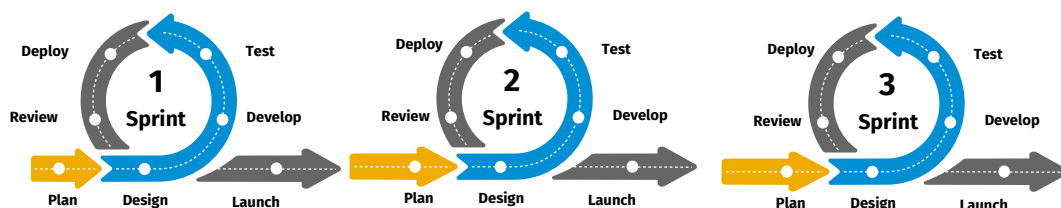


Abbildung 1: Exemplarische Abfolge eines agilen Entwicklungszykluses. In Anlehnung an K&C [KUC.2021].

Sprints sind Durchläufe, welche i.d.R. einen Zeitraum von ein bis vier Wochen umfassen, in welcher die Fertigstellung einer vor dem Teilprojekt definierten Aufgabenkontingente vorgesehen ist. Am Ende jeden Sprints soll dabei ein potenziell an den Kunden auslieferbares Produkt entstehen. Dies erlaubt eine schnelle Bereitstellung funk-

tionsfähiger Software, was neben der Erhöhung der Kundenzufriedenheit ebenfalls in einer Optimierung der Planungsgenauigkeit resultiert. So kann das nach Ablauf des Sprints potenziell an die Stakeholder auslieferbare Artefakt als Feedback-Grundlage für weitere Anpassungen verwendet werden [1][39]. Die dabei bereitgestellten Features können vom Kunden im Regelbetrieb geprüft und auf Anforderungserfüllung untersucht werden [3][180]. Damit kann auf während des Reviews festgestellte Mängel im unmittelbaren Folge-Sprint reagiert werden. Dieser Sprint-Review-Zyklus wird solange durchgeführt, bis alle Kundenbedürfnisse erfüllt sind und das Vorhaben als abgeschlossen gilt. Innerhalb der letzten Dekade haben sich dabei diverse auf agilen Prinzipien basierenden Vorgehensmodelle, wie Scrum, Kanban oder Extreme Programming (XP) in der Softwareentwicklung etabliert. Obwohl einige dieser Methoden zur erfolgreichen Zusammenarbeit innerhalb der Entwicklungsteams beigetragen haben, bleibt das sog. Problem der *Letzten Meile* bestehen [2]. Traditionell erfolgt eine funktionale Trennung der Entwickler- und IT-Betriebs-Teams. Das Problem der Letzten Meile beschreibt dabei, dass aufgrund ausbleibender Kooperation der Entwicklungs- und Betriebsteams der Programmcode nicht auf die Produktivumgebung abgestimmt ist. Die Verzögerung der Markteinführungszeit resultiert damit in einem geminderten Wertschöpfungsprozess und somit einer Abnahme des Ertragspotenzials bzw. in einer Erhöhung der Betriebskosten [5][1]. Und scheint dieses Phänomen kein Einzelfall zu sein. So geht aus der von McKinsey veröffentlichten Studie *The Business Value of Design 2019* hervor, dass durchschnittlich 80 Prozent des Unternehmens-IT-Budgets zur Aufrechterhaltung des Status Quo verwendet wird. Stattdessen fordert das Beratungshaus eine Rationalisierung der Bereitstellung und des Betriebs von Software, um finanzielle Mittel für wertschöpfende Investitionen zu maximieren [6]. Abhilfe schaffen kann dabei das in der Literatur als **Development & Operations (DevOps)** bekannte Aufbrechen organisatorischer Silos zwischen Entwicklung und dem IT-Betrieb [5][1]. Dabei stellt DevOps keine neue Erfindung dar. Stattdessen werden einzelne bereits bewährte Konzepte, wie die agile Softwareentwicklung, zu einem umfassenden Rahmenwerk konsolidiert. Prägnant zusammenfassen lässt sich das DevOps-Konzept durch das Akronym CAMS: *Culture (Kul-*

tur), *Automation* (Automatisierung), *Measurement* (Messung) und *Sharing* (Teilen) [5][5]. Dabei gilt *Kultur* als das wohl wesentlichste DevOps-Erfolgselement. Diese Norm bezweckt eine Zusammenarbeitskultur, welche sich über alle Ebenen eines Unternehmens erstreckt. Operative Entscheidungen sollen dabei auf die Fachebenen herunterdelegiert werden, welche aufgrund ihrer spezifischen Expertise am geeignetsten sind, diese Dispositionen zu verabschieden [5][5]. Eine *Automatisierung* von bei der Softwarebereitstellung anfallenden Prozessen, ermöglicht sich wiederholende manuelle Arbeit zu eliminieren. Eine hochautomatisierte Infrastruktur kann ebenfalls zur Rationalisierung und damit zur Senkung der IT-Betriebskosten beitragen. Der dabei erzielte Einfluss wird anhand verschiedener DevOps-Kennzahlen bemessen (*Messung*). Neben der Systemverfügbarkeit und der Instandsetzungszeit ist insbesondere der *Time-to-Market* eine signifikante Metrik [5][7].

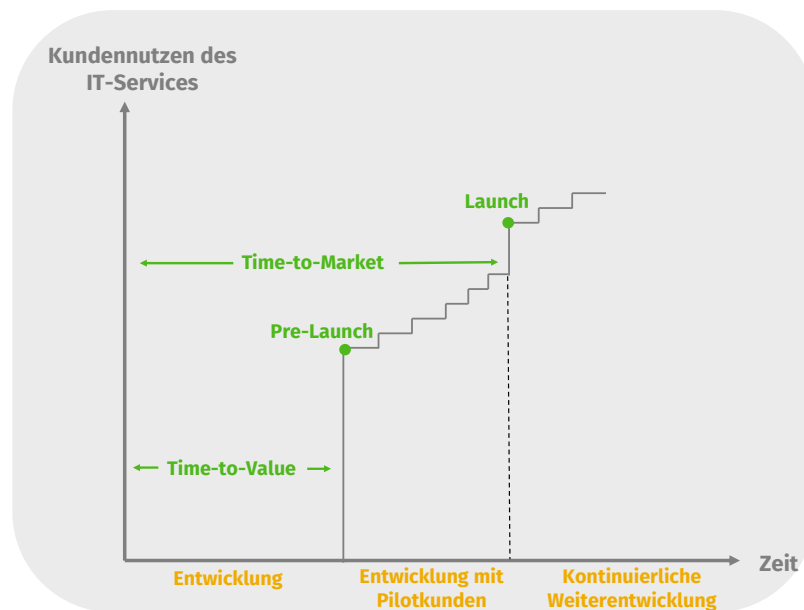


Abbildung 2: Zeitliche Darstellung des Kundennutzen von IT-Services. In Anlehnung an Halstenberg [5][9].

Der Time-To-Market beschreibt die Zeitspanne zwischen Entwicklungsentstehungsprozess und der Markteinführung der IT-Services [7][141]. Auch der *Time-to-Value* erhält zunehmend eine erhöhte Aufmerksamkeit. Im Gegensatz zum Time-to-Market

wird hier nicht die Zeit bis zur Komplett-Einführung, sondern bis zum ersten Kundennutzen bemessen. Obwohl der im Time-to-Value bereitgestellte IT-Service möglicherweise Verbesserungspotenzial bietet, überwiegt der mit der initialen Auslieferung herbeigeführte Mehrwert. Dabei ermöglicht eine solche Früheinführung einen Vorsprung gegenüber anderen Konkurrenten. So ist es dem Softwareunternehmen bereits gelungen erste Kunden zu finden, deren Input und Feedback möglichst rasch erfasst und verarbeitet werden kann [5][9]. Softwareunternehmen können IT-Services ab dem Pre-Launch somit sukzessive und ressourcenoptimiert unter Zusammenarbeit mit den Pilotkunden erweitern. Auch Adam Caplan leitender Strategieberater bei Salesforce empfiehlt angesichts der bei der Integration der IT-Systemen entstehenden Komplexität, Software schnellstmöglichst in Produktivumgebung zu testen [7]. Aus diesen Erfahrungen sollen dann Best Practises entwickelt werden, welche dann innerhalb von Teams und organisationsübergreifend weitergegeben werden sollten (*Teilen*) [5][7].

2.2.2 CI/CD-Pipelines zur Softwarebereitstellung

DevOps beschreibt eine Philosophie, mit welcher die Zusammenarbeit zwischen Entwicklungs- und Betriebsteams gefördert werden soll. Ein integraler Bestandteil des DevOps-Rahmenwerks ist neben verschiedener agiler Entwicklungsmethoden ebenfalls Continuous Integration and Continuous Delivery (CI/CD). CI/CD ist eine Softwareentwicklungsmethode, welche zur Verbesserung der Qualität bzw. zur Senkung der Entwicklungszeit von IT-Services beitragen soll. Abhilfe schaffen soll dabei eine Pipeline, welche alle Schritte von Codeintegration bis Bereitstellung der Software automatisiert. Hauptaugenmerk liegt dabei darauf, Software zuverlässiger und häufiger bereitzustellen. Alle in diesem Prozess anfallenden Aktivitäten sind dabei dem CI/CD-Cycle zu entnehmen (s. Abb. 3).

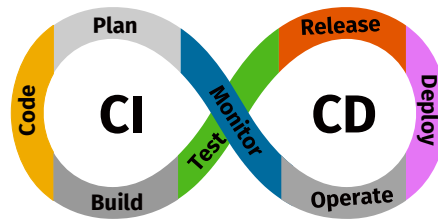


Abbildung 3: Aktivitäten im CI/CD-Prozess. In Anlehnung an.

Der CI-Prozess (Continuous-Integration-Prozess) bezweckt, dass lokale Quellcodeänderungen in kurzen Intervallen und so schnell wie möglich in eine zentrale Codebasis geladen werden. Das frühzeitige Integrieren von Code soll dabei zu einer unmittelbaren und zuverlässigen Fehlererkennung innerhalb des Entwicklungsvorhabens beitragen. Der erste Schritt im CI-Prozess umfasst die *Planung* zu entwickelnder Services (s. Abb. 3). Dabei soll festgestellt werden, welche Anforderungen eine Lösung besitzt bzw. welche Softwarearchitekturen sowie Sicherheitsmaßnahmen implementiert werden sollten. Im Sinne der agilen Entwicklung wird dabei das aus dem *Monitoring* erhobene Feedback berücksichtigt und angewendet. Um sicherzustellen, dass die in der Planung entworfene Anwendungsarchitektur auf das Design des Produktsystems abgestimmt ist, sollte zu jedem Zeitpunkt das Know-How der Betriebsteams einbezogen werden. Nach erfolgreichem Entwurf zu implementieren der Anwendungsfeatures beginnt die Entwicklung der IT-Services (*Code*: s. Abb. 3). Arbeiten hierbei mehrere Entwickler parallel an dem selben IT-Service, wird der entsprechende Quellcode in sog. Repositories wie Github oder Bitbucket ausgelagert. Ein Repository stellt dabei einen zentralen Speicherort dar, welcher das Verfolgen sowie Überprüfen von Änderungen und ein paralleles bzw. konkurrierendes Arbeiten an einer gemeinsamen Codebasis ermöglichen soll. Der in dem Repository archivierte Hauptzweig (Master-Branch) stellt dabei eine aktuelle und funktionsfähige Version des Codes dar. Dieser mit verschiedenen Validierungsprozessen überprüfte Code, kann dabei zu jeder Zeit in der Produktionsumgebung bereitgestellt werden. (s. Abb. 4). Im Sinne der agilen Entwicklung werden dabei große Softwareanforderungen, sog. Epics, in kleine Features segmentiert, welche in separate Feature-Banches ausgelagert werden. Diese sind unabhängige Kopien des Hauptzweiges, in welcher

ein Entwickler Änderungen vornehmen können ohne Konflikte in der gemeinsamen Codebasis zu verursachen.

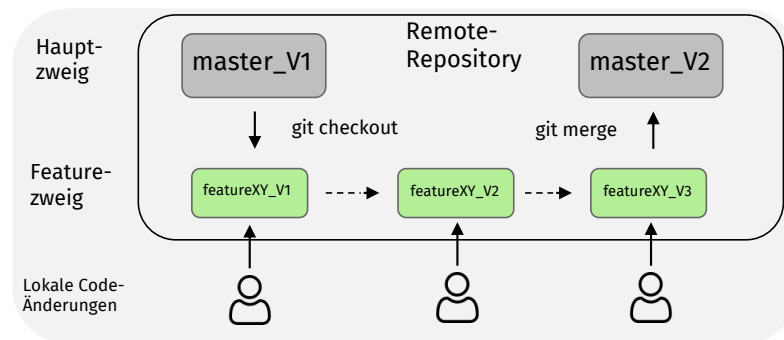


Abbildung 4: Versionskontrollsysteme zur Verwaltung von Quellcode. In Anlehnung an.

Nach Fertigstellung der Funktionalitäten sollte der um die Features erweiterte Quellcode so sichergestellt werden, dass der Code stets stabil, also funktionsfähig ist und keine Konflikte mit dem aktuellen Code des Hauptzweiges aufweist. Dabei wird eine Validierung des Code anhand der Definition of Done (DoD) fokussiert. Die DoD ist eine in der Planungsphase festgelegte Anforderungsspezifikation, welche zur Integration in den Master-Branch erfüllt werden muss. Im Rahmen dieser Norm sind Entwickler dazu angehalten für jedes implementierte Feature einen der DoD entsprechenden Test zu entwerfen. Anschließend wird die Applikation zu einem ausführbaren Programm kompiliert (*Build*) anhand welcher verschiedenste Tests ausgeführt werden. Dafür können je nach Programmiersprache verschiedene Build-Tools, wie Maven für Java oder NPM für Javascript verwendet werden. Die dabei durchgeführten Überprüfungen, auch ‚Smoke-Tests‘, sollen sicherstellen, dass zu jeder Zeit in rudimentärgetesteter Code bereit steht und grundlegende Funktionalitäten sowie Schnittstellen erwartungsgemäß operieren. Der in den Entwicklungszweig bereitgestellte Code wird dabei überwiegend anhand schnell durchführbarer Tests überprüft. Die Durchführung von ressourcenschonenden Tests hat dabei insbesondere zwei Ursachen. Der Hauptaugenmerk von Continuous Integration liegt insbesondere darauf, dass der lokale Entwicklungscode so oft wie möglich in das

gemeinsame Repository geladen wird. Aufwendige Tests mit einer langen Laufzeit würden Entwickler somit hemmen Code in einer hohen Frequenz bereitzustellen. Statt den gesamten Code vor einem Release vorzusammenzuführen (Merge Day), soll sichergestellt werden, dass Fehler und Konflikte so schnell wie möglich entdeckt und behoben werden. Dabei durchgeführte Tests sind i.d.R. Unit- und sowie Integrationstests.

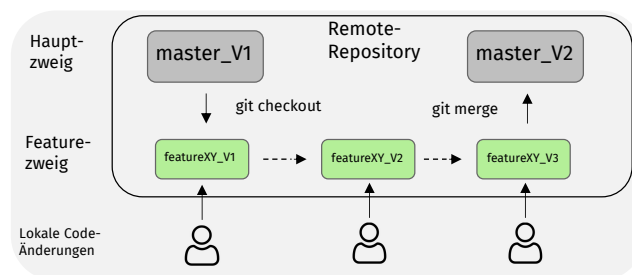


Abbildung 5: Hierarchische Darstellung von Softwaretests. In Anlehnung an.

Unit Tests befinden sich dabei auf unterster Hierarchieebene der Test-Pyramiede (s. Abb. 5). Mit Unit Tests wird die Funktionale Korrektheit kleinster Einheit überprüft. Dies können etwa unabhängige Komponenten wie einzelne Methoden einer Klasse sein. Der Zweck der Unit-Tests besteht dabei in einer von externen Einflüssen und Daten unabhängigen Überprüfung der einzelnen Komponenten. Um bei der Bereitstellung neuer Funktionalitäten ebenfalls das Zusammenspiel einzelner Komponenten zu überprüfen, werden Integration Tests durchgeführt. I.d.R. werden diese in einer White-Boxed-Umgebung abgewickelt. Dies impliziert, dass der Entwickler des Integration-Tests Kenntnis von dem Quellcode, den verwendeten Technologien sowie der Anwendungsarchitektur besitzt. Bei einem solchen Tests, können also gezielt Aspekte, wie der Austausch eines Nachrichtenmodells bei der Kommunikation zweier Web-Services untersucht werden. Nachdem einzelne Funktionalitäten entwickelt und erfolgreich getestet werden, werden diese zu den Änderungen im Hauptzweig

zusammengeführt. Ab diesem Zeitpunkt beginnt das Continuous Delivery. Während Continuous Integration insbesondere den Prozess der kontinuierlichen Integration von Quellcode in ein gemeinsames Repository bezweckt, wird mit Continuous Delivery die Automatisierung der Anwendungsbereitstellung gesteuert. Applikationen sollen somit ohne große Verzögerungen in die Produktivumgebung und somit zum Kunden ausgeliefert werden. In der Theorie sollte der CD-Prozess automatisch und unmittelbar nach Ablauf aller CI-Aktivitäten und somit nach Integration ggdes Codes in den Hauptzweig angestoßen werden. In der Praxis wird hierbei jedoch häufig ein manueller Schritt zwischengestellt, womit sichergestellt werden kann, dass die Anwendung erst in die Produktionsumgebung ausgerollt wird nachdem alle Gesichtspunkte der Bereitstellung überprüft wurden. Zu Beginn des CD-Prozesses wird das in die Produktivumgebung bereitzustellende Artefakt über eine Deployment-Pipeline in eine Staging-Area gelanden. Bei der Staging-Area handelt es sich dabei um ein System, welches zwischen der Entwicklungs- und der Produktivumgebung liegt. Dabei werden Konfigurationen des Staging-Systems möglichst produktionsähnlich angelegt. Neben den Datenbanken, werden hierbei auch Serverkonfigurationen, wie Firewall- oder Netzwerkeinstellungen von dem Produktivsystem übernommen. Somit soll sichergestellt werden, dass eine neue Version der Anwendung unter den Bedingungen die der Produktionsumgebung möglichst getestet wird. Wie im CI-Prozess, werden innerhalb dieser Phase Unit- und Integrationstests durchgeführt. Diese sind i.d.R. deutlich rechenintensiver und besitzen eine längere Ausführungszeit. In der Staging Area werden unterdessen auch in der Test-Pyramide (s. Abb. 5) höher gestufte Tests ausgeführt. Dazu gehören etwa *Functional-Tests*. Mit diesen werden die Planungsphase festgelegten Anforderungen bzw. Funktionen der Anwendung überprüft. So etwa überprüft werden, ob bei der Eingabe einer Benutzer-Passwort-Kennung ein korrekter Authorisierungstoken übergeben wird. Genau wie der Integrationstest, umfasst ein Funct die Überprüfung mehrere Komponenten. Während bei Integrationstests jedoch überprüfung lediglich die generelle Durchführbarkeit einer Operation, wie z.B. ob die Abfrage nach einer Authorisierungstoken überhaupt möglich ist, wird bei einem Functional Test die Korrektheit der spezifischen Ausgabe überprüft.

End-to-End-Tests liegen dabei in der Test-Pyramide eine weitere Hierarchieebene höher. Mit diesen soll sichergestellt werden, dass die gesamten Anforderungen der verschiedenen Stakeholder erfüllt werden. Hierbei wird ein vollständiges Anwenderszenario von Anfang bis Ende getestet. Ein solches Szenario kann im Kontext eines E-Commerce-Webshops etwa das Anmelden mit Benutzernamen, das Suchen eines Produktes und das anschließende Bestellen umfassen. Zusätzlich zu den hier aufgeführten deduktiven Tests, werden in der Deployment-Pipeline i.d.R. auch Codeanalysen durchgeführt. Hier wird etwa überprüft, welcher prozentuale Anteil des Codes durch Unit-Tests überprüft wird oder Qualitätsstandards, bei welchen auf Schwachstellen verwendeter Code-Patterns bzw. Dublikate überprüft wird. Hat die Version alle Tests erfolgreich durchlaufen wird das überprüfte Artefakt auf die Cloud-Plattform geladen (*Deploy*). Je nach Bereitstellungsstrategie (s. ??), wird die Anwendung dann unmittelbar oder erst nach weiteren Überprüfungen für den Kunden zugänglich gemacht. Um eine ordnungsgemäße Ausführung der Anwendung in der Produktionsumgebung sicherzustellen wird die bereitgestellte Anwendung im letzten Schritt des CD-Prozesses überwacht (*Monitoring*). Dabei wird das Applikationstracking i.d.R. unabhängig von der CI/CD-Pipeline auf der Cloud-Plattform betrieben. Wichtige Überwachungselemente sind Infrastruktur sowie Anwendungs-Monitoring. Beim Infrastruktur-Monitoring werden Metriken wie CPU-, Speicher- und Netzwerklast der Server, Datenbanken und Netzwerkkomponenten untersucht. Das Anwendungs-Monitoring umfasst dabei die Überwachung der Funktionalitäten sowie der Applikation selbst. Hierbei werden Informationen wie Anfragen pro Sekunde, die Anzahl der Benutzer oder in Log-Dateien gesammelte Fehlercodes analysiert.

2.2.3 Strategien zur Bereitstellung von Neuentwicklungen

Nachdem das Artefakt auf einer virtuellen Maschine einer Cloud-Instanz installiert und ausgeführt wird entscheiden verschiedene Strategien über die Inbetriebnahme der neuen Softwareversion. Mit dieser Strategie wird festgelegt, wie Nutzeranfragen innerhalb des Produktivsystems von der alten auf die neue Version der Anwendung

übermittelt wird.

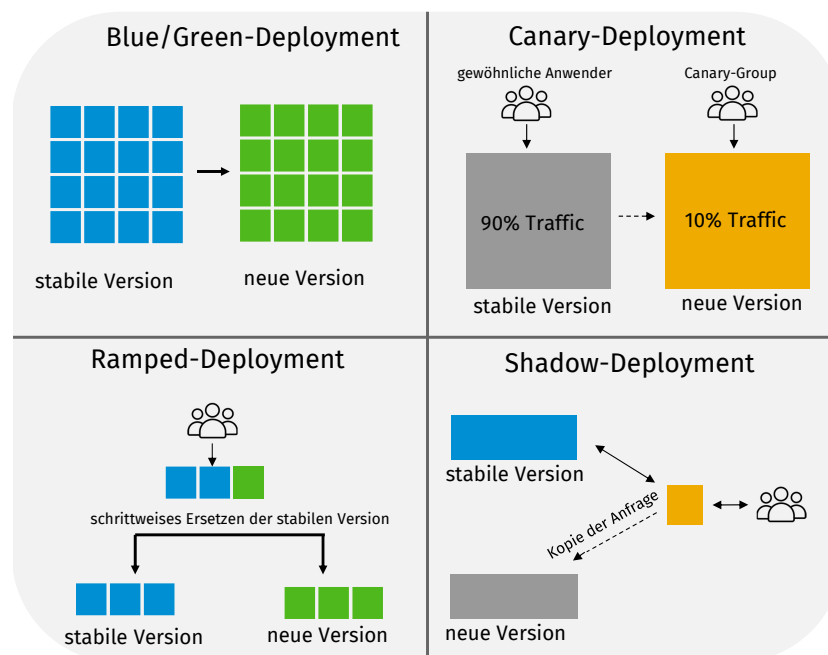


Abbildung 6: Strategien zur Bereitstellung von Software. In Anlehnung an.

Eine häufig verwendete Deployment-Strategie ist dabei das *Blue/Green-Deployment*. Hierbei wird neben der stabilen aktuellen Anwendung (die blaue Version) ebenfalls eine Instanz mit der neuen Anwendung (die grüne Version) betrieben. Die Nutzeranfragen werden dabei von dem Lastverteilungsservice erst nach Validierung aller Tests umgeschaltet. Dazu gehören neben den in der CI/CD-Pipeline definierten Tests ebenfalls Abnahmetest der Qualitätssicherung ausgeführt. Dies sind manuelle Tests, in welchen Funktionen, Benutzeroberfläche sowie die Anwenderfreundlichkeit überprüft werden. Statt wie beim Blue/Green-Deployment die neue Version zu einem Zeitpunkt für die gesamte Nutzerbasis bereitzustellen, zielt das *Canary-Deployment* darauf, die Nutzungslast kontinuierlich und sukzessive auf die neue Anwendung umzuleiten. Hierfür wird die Anwendung vorerst einer kleinen Anzahl an Benutzer bereitgestellt. Eine Canary-Gruppe sollte dabei so zusammengestellt werden, als dass diese die Gesamtzielgruppe möglichst gut repräsentiert. Anhand dieser soll dann der fehlerfreie Betrieb der neuen Anwendung überprüft und ggf. Anpassung vorgenommen werden, bevor diese für alle Nutzer freigeschaltet wird. Für Anwendungen mit

einer hohen Service-Level-Architektur sowie kritischen IT-Systeme wird i.d.R. die *Ramped-Deployment-Strategy* verwendet. Diese ermöglicht eine präzise Kontrolle für horizontalskalierte Services. Bei einer horizontalskalierten IT-Infrastruktur wird die Rechenkapazität einer Anwendung durch das Hinzufügen indentischer Services. Mit dieser Architektur ermöglicht sich eine bessere Lastverteilungen, da Anfragen unabhängig von dem Kontext auf unterschiedliche Instanzen verteilt werden können. Mit Beginn des Ramped-Deployment-Prozesses wird die neue Softwareversion schrittweise auf den horizontalen Instanzen ausgerollt. Dabei ist es möglich, dass zu Beginn des Ausrollens die ersten aktualisierten Instanzen lediglich für bestimmte Anwender, sog. Ramped-Gruppen bereitgestellt wird. Dabei ermöglicht es sich für die DevOps-Teams neben der Analyse der Anwendungsmetriken ebenfalls das Feedback der Ramped-Gruppen zu erfassen. Eine aufwendigere jedoch mit weniger Risiko behaftete Bereitstellungsstrategie stellt das *Shadow-Deployment* dar. Dabei wird neben der Instanz der aktuellen Version ebenfalls ein sog. Shadow-Model auf der Infrastruktur betrieben. Dieses Shadow-Model beinhaltet dabei die neue Version der Anwendung, kann jedoch nicht unmittelbar von den Nutzer verwendet werden, sondern stellt dem Namen entsprechend eine hinter der stabilen Version gelagerte Instanz dar. Anfragen der Anwender werden von dem Cloud-Service stets auf die aktuelle Version weitergeleitet, verarbeitet und beantwortet. Eine Kopie dieser Anfrage wird dabei jedoch ebenfalls an das Shadow-Model weitergeleitet und zur Laufzeit prozessiert. Die Verwendung des in der Produktionsumgebung abgewickelten Netzwerkverkehrs, ermöglicht den Entwickler somit eine anwendungsbezogene Überprüfung entwickelter Features.

3 Methodische Vorgehensweise

3.1 Prototypische Entwicklung der CI/CD-Pipelines

3.2 Evaluation der CI/CD-Pipelines unter Anwendung des Analytischen Hierarchieprozess

3.3 Semistrukturierte Leitfadeninterviews

4 Anwendung der Methodik auf die theoretischen Grundlagen

4.1 Prototypische Entwicklung der CI/CD-Pipelines

4.2 Evaluation der CI/CD-Pipelines unter Anwendung des Analytischen Hierarchieprozess

4.3 Entwicklung einer ganzheitlichen Bereitstellungsstrategie

5 Schlussbetrachtung

5.1 Fazit und kritische Reflexion

5.2 Ausblick

Literatur

- [1] John Adam. *Was ist agile Softwareentwicklung?* Hrsg. von K&C. 2021. URL: <https://kruschecompany.com/de/agile-softwareentwicklung/> (besucht am 05.03.2023).
- [2] Shweta Bhandu und Abby Taylor. *The Evolution from Agile to DevOps to Continuous Delivery — Qentelli*. Hrsg. von Qentelli. 2023-03-05. URL: <https://www.qentelli.com/thought-leadership/insights/evolution-agile-devops-continuous-delivery> (besucht am 05.03.2023).
- [3] Boris Gloger. *Scrum: Produkte zuverlässig und schnell entwickeln*. 5., überarbeitete Auflage. Hanser eLibrary. München: Hanser, 2016. ISBN: 9783446448360.
- [4] Joachim Goll und Daniel Hommel. *Mit Scrum zum gewünschten System*. Wiesbaden: Springer Vieweg, 2015. ISBN: 9783658107208.
- [5] Jürgen Halstenberg. *DevOps: Ein Überblick*. Essentials Ser. Wiesbaden: Springer Fachmedien Wiesbaden GmbH, 2020. ISBN: 9783658314057.
- [6] McKinsey, Hrsg. *The business value of design*. 2019. URL: <https://www.mckinsey.com/capabilities/mckinsey-design/our-insights/the-business-value-of-design> (besucht am 05.03.2023).
- [7] Joseph T. Vesey. „Time-to-market: Put speed in product development“. In: *Industrial Marketing Management* 21.2 (1992), S. 151–158.
- [8] Wolfgang Vieweg. „Agiles (Projekt-)Management“. In: *Management in Komplexität und Unsicherheit*. Springer, Wiesbaden, 2015, S. 41–42.
- [9] Alberto Vivencio, Hrsg. *Testmanagement Bei SAP-Projekten: Erfolgreich Planen * Steuern * Reporten Bei der Einführung Von SAP-Banking*. 1st ed. Wiesbaden: Springer Fachmedien Wiesbaden GmbH, 2013. ISBN: 978-3-8348-1623-8.

Anhang