## Sperrvermerk

Die vorliegende Arbeit beinhaltet interne und vertrauliche Informationen der Firma dotSource GmbH. Die Weitergabe des Inhalts der Arbeit und eventuell beiligenden Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften – auch in digitaler Form – gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma dotSource GmbH. Die Arbeit ist nur Mitgliedern des Prüfungsausschusses zugänglich zu machen.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

AWS …….. Amazon Web Services
CI …….. Continuous Integration
CD ……..... Continuous Delivery
DHGE …… Duale Hochschule Gera-Eisenach
QA …….. Quality Assurance
NPM ……...… Node Package Manager
PROD …… Produktionsumgebung

# 1 Einleitung

> *Wenn du etwas so machst, wie du es seit zehn Jahren gemacht hast, dann sind die Chancen groß, dass du es falsch machst.*
>
> **Charles Kettering**

Dieses Zitat von Charles Kettering unterstreicht, dass IT-Unternehmen in einer sich immer mehr digitalisierenden Gesellschaft niemals stehen bleiben dürfen, denn um mit der wachsenden Konkurrenz mithalten zu können und kompetitiv zu bleiben, muss sich ein Unternehmen stetig weiterentwickeln. Entwicklungsprozesse müssen effizienter gestaltet werden, um Software mit hoher Qualität regelmäßig bereitstellen zu können. Entwicklerteams sollen neue Methoden der Zusammenarbeit lernen, um Kundenprojekte koordiniert und systematisch umzusetzen. Es ist ein stetiges Lernen und Weiterbilden, was den Alltag eines modernen Entwicklers prägt.

Da ein Entwicklungsprozess viele verschiedene Teilaspekte von der Konzeptionierung bis zum fertig ausgelieferten Produkt besitzt, kann ein solcher Prozess auch an vielen Stellen optimiert und effizienter gestaltet werden. Methoden der agilen Softwareentwicklung beschleunigen den Implementierungsprozess, Modelle und Pretotypen unterstützen die Anschaulichkeit von Konzepten, um direkte Rückmeldung vom Kunden bekommen zu können. Doch auch der Prozess der Bereitstellung von Änderungen und neuen Features kann in vielen Punkten durch Automatisierung und weitere Methoden verbessert werden. Dazu dienen DevOps, CI (Continuous Integration) und CD (Continuous Delivery und Continuous Deployment)

Ziel dieser Arbeit ist es, verschiedene Optimierungsmöglichkeiten für solche automatisierten Abläufe zu evaluieren und anhand eines Kundeprojektes zu implementieren, um mit ihrem Einfluss die Laufzeit der DevOps Prozesse im Projekt zu verkürzen.

## 1.1 CI/CD und DevOps

In der Vergangenheit gab es Zeiten, in denen IT-Abteilungen einmal im Jahr eine größere Aktualisierung vornehmen konnten. Trotz agiler Entwicklung mussten Änderungen durch interne Qualitäts- und Betriebsprozesse gebracht werden, und bis die Projekte nach

Kundenakzeptanz auf den Produktionsumgebungen landeten, waren wieder einige Tage oder sogar Wochen vergangen. Doch heutzutage ist damit Schluss – DevOps bringt die Lösung.[1]

Entwicklung (Development) trifft auf IT Betrieb (Operations). Daraus entsteht mit DevOps ein Ansatz, der die Prozesse zwischen Softwareentwicklung und operationalen IT-Teams automatisiert und optimiert, damit Software schneller und zuverlässiger erstellt, getestet und freigegeben werden kann. Dabei sollen Entwicklungs- und operationellen IT-Teams über den gesamten Lebenszyklus von Softwareanwendungen möglichst nah aneinander arbeiten. [2]

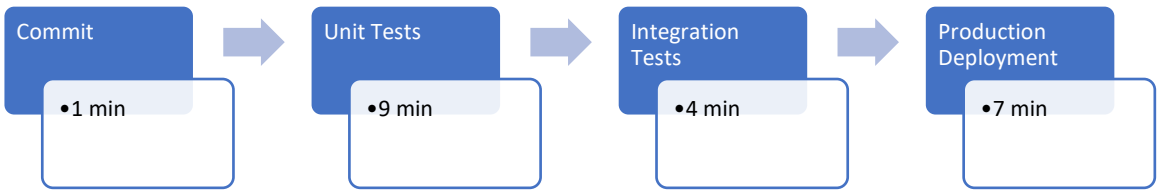


Abbildung 1: Beispiel für eine CI/CD Pipeline

CI/CD sind Praktiken von DevOps. Continuous Integration beschäftigt sich mit der Entwicklung von Quellcode in einer einheitlichen Codebasis, mit Versionskontrolle und automatisierten Tests, während der Entwickler in möglichst kurzen Abständen weniger komplexe Änderungen einreichen soll, um Fehler frühzeitig zu erkennen. Continuous Delivery beschäftigt sich damit, den Code immer in releasefähigem Zustand zu halten und automatisierte Tests einzusetzen, um möglichst schnelles Feedback für die Entwickler zu schaffen. Continuous Deployment schließt daran an und strebt danach, möglichst kleine und weniger komplexe Deployments automatisiert auf den Test- und Produktionsumgebungen verfügbar zu machen, um so die erweiterten Anwendungen den Nutzern bereitzustellen. Der gesamte Prozess vom Einsenden der Codeänderungen bis zum Release der neuen Features wird automatisiert als CI/CD Pipeline bezeichnet.

[1] Vgl. [PIE23]
[2] Vgl. [SAF22]

Eine automatisierte Pipeline kann beliebig viele Schritte je nach Anforderungen des Entwicklerteams besitzen. Jede solche Phase wird dann entweder automatisch oder manuell bestätigt in die nächste Phase übergeleitet, bis das Ende erreicht ist. Werden die Zeitaufwände addiert, die jeder Schritt in der Pipeline erfordert, kann ein vollständigen Durchlauf dieser Pipeline durch die Total-Cycle-Time gemessen werden. In dem abgebildeten Beispiel würde die Total-Cycle-Time 21 Minuten betragen.

Es gibt viele verschiedene Werkzeuge für Entwickler, CI/CD Pipelines zu verwalten. Zu den größten zählen Jenkins, AWS CodePipeline oder auch GitLab, welches ursprünglich zur Versionsverwaltung von Softwareprojekten auf Git-Basis diente, aber auch sehr umfangreiche DevOps Funktionalitäten bietet. Im Verlauf der Praxisarbeit wird ausführlicher auf die Verwendung von GitLab als Tool zur Verwaltung der Pipelines eingegangen.

## 1.2 IST-Zustand

Die dotSource interne CI/CD Pipeline von einem Salesforce B2B Entwicklungsprojekt automatisiert Tests, Validierung sowie die Paketerstellung für die Test- und Produktivumgebung des Kunden und besitzt mit durchschnittlich über 50 Minuten eine hohe Total-Cycle-Time, was die Softwareentwickler wichtige Entwicklungszeit kostet. In dieser Zeit sind auf den Test- und Kundenumgebungen beispielsweise im Vergleich zum Code-Repository veraltete Codestände, was zu Problemen bei Deployments von anderen Entwicklern führen könnte. Außerdem möchte der Entwickler selbst möglichst zeitnahes Feedback zu seinen Änderungen bekommen. Ist die TCT (Total-Cycle-Time) dann sehr lang, erfährt der Entwickler vielleicht erst nach einer Stunde, dass sein Code nicht zu einem produktivfähigem Package zusammengebaut werden kann und muss den Inhalt dann noch einmal anpassen.

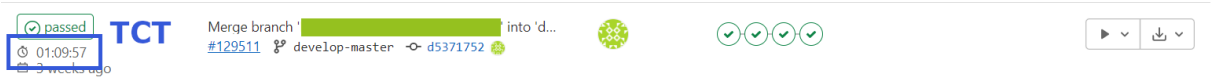


Abbildung 2: Auszug aus der Pipelineübersicht in GitLab

## 1.3 SOLL-Zustand

Ohne die Pipeline in ihren Funktionalitäten einzuschränken, soll die Total-Cycle-Time (TCT) vom Zeitpunkt einer akzeptierten Merge-Request bis zum erfolgreichen Deployment auf die Testumgebung reduziert werden. Dazu ist eine Analyse von potentiellen Verbesserungsmöglichkeiten in den verschiedenen Phasen nötig. Danach sollen mögliche Optimierungsansätze miteinander verglichen werden, gefolgt von der Implementierung einer geeigneten Lösung, um die Pipeline effizienter zu gestalten und so den Zeitaufwand eines Durchlaufs der Pipeline zu reduzieren. Am Ende soll das Entwicklerteam der dotSource schnellere Durchläufe im GitLab verzeichnen können.

# 2 Konzeptionierung

Um die Laufzeit der Deployment Pipeline zu reduzieren zu können, wird im folgenden Kapitel die Pipeline im ersten Schritt analysiert, um danach aus verschiedenen Optimierungsmöglichkeiten einen geeigneten Ansatz zu wählen und zu implementieren. Um dies jedoch erreichen zu können, ist es notwendig zu wissen, an welchen Schritten die Pipeline eher langsam und ineffektiv arbeitet. Diese gefundenen Stellen bieten die Basis für die Wahl des Lösungsansatzes. Selbst wenn durch die Maßnahmen nur die Laufzeit von einer einzelnen Phase angepasst wird, so wird die schon Einfluss auf die Total-Cycle-Time der gesamten Pipeline nehmen.

## 2.1 Analyse der CI/CD Pipeline

Die Deployment Pipeline basiert im zu betrachtenden Fall auf drei verschiedenen Stages, die im Rahmen der Projektarbeit für die Analyse relevant sind. Eine Stage beschreibt einen Arbeitsschritt der Pipeline, der verschiedene kleine Unteraufgaben (genannt Jobs) besitzt, die parallel abgearbeitet werden. Erst wenn alle Jobs erfolgreich abgeschlossen sind, wird die nächste Stage in der Pipeline abgearbeitet.

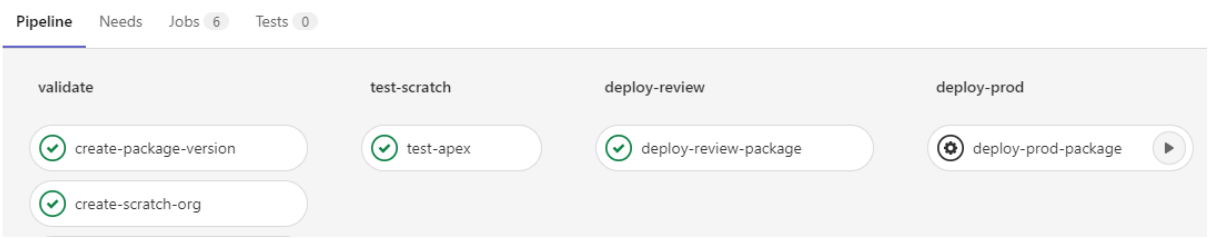


Abbildung 3: Stages und Jobs der Pipeline

In der Salesforce Entwicklung werden als Entwicklungsumgebungen Scratch-Orgs genutzt, auf welchen das aktuelle Feature implementiert wird. Auf solchen Scratch-Orgs kann die Funktionsweise der Änderungen getestet werden, bevor aus dem Codestand ein Package gebaut wird, welches später auf die Produktivumgebung des Kunden deployed wird.

Die erste Stage trägt die Beschriftung „**validate**“ und beinhaltet zwei Jobs, die gleichzeitig abgearbeitet werden. Zum einen wird hier eine verpackte Version der Komponenten erstellt und validiert, zum anderen wird parallel eine neue Scratch-Org erstellt, welche für eine spätere Stage notwendig. In Salesforce ist es im Deploymentprozess notwendig, vor dem Deployment ausführliche Unit-Tests mit mindestens 75% Codeabdeckung der Apex-Klassen durchzuführen. Für diesen Testdurchlauf ist eine vorher erstellte Scratch-Org notwendig. Um

in der zweiten Stage „**test-scratch**“, in welcher die Tests durchgeführt werden, wertvolle Zeit zu sparen, wurde in der aktuellen Konfiguration der Pipeline dieser Schritt zum Erstellen bereits in die erste Stage gelegt, damit eine frisch erstellte Scratch-Org direkt zum Start der Tests bereitsteht. Die dritte Stage „**deploy-review**“ ist nun dafür zuständig, die validierte und getestete Package-Version auf die Review-Sandbox zu deployen, sodass der Kunde Feedback zu den Änderungen geben und ausführlichere Systemtests durchgeführt werden können. Ist diese Phase erfolgreich beendet, so ist die Deployment Pipeline einmal erfolgreich durchlaufen wurden. Im Folgenden werden die verschiedenen Stages nun im Detail betrachtet.

Im Anhang 1 ist eine Statistik der letzten zehn Pipelinedurchläufe zu finden. Es wurden jeweils die Laufzeiten der individuellen Jobs, sowie die Total-Cycle-Time der gesamten Pipeline aufgezeichnet. Zu erwähnen ist, dass in der Statistik zur besseren Überschaubarkeit nur Pipelinedurchläufe betrachtet wurden, bei denen alle Jobs erfolgreich durchgelaufen sind. Sollte ein Job im Deploymentprozess fehlschlagen, so muss der Entwickler den Fehler erst beheben und den Job dann erneut starten, was die TCT, welche im Rahmen dieser Arbeit betrachtet werden soll, stark abweichen lassen würde. Die Analyse dieser Arbeit fokussiert sich auf die automatisierten Prozesse in der Pipeline, nicht auf Fehlerwahrscheinlichkeiten der Entwickler oder auftretende Systemfehler, weshalb bei der Erhebung der Statistiken Pipelinedurchläufe mit fehlgeschlagenen Jobs aussortiert und nicht betrachtet wurden.

### 2.1.1 Betrachtung der Pipelinephase „validate“

Von den beiden Jobs, die gleichzeitig mit Beginn diese Stage parallel starten, benötigt der Job ***create-package-version*** mit einer Durchschnittslaufzeit von 31:07 Minuten deutlich länger als der Job ***create-scratch-org***, welcher durchschnittlich 17:53 Minuten braucht. Daraus folgt, dass der Start der nächsten Phase abhängig vom Job ***create-package-version*** ist, da der parallellaufende Job zu diesem Zeitpunkt schon lange abgeschlossen ist. ***create-package-version*** benötigt viel Zeit, da das Projekt bereits sehr viele Metadaten besitzt, die alle in das eine, große Package verpackt werden. Das Anlegen einer neuen Scratch-Org ist jedoch auch ein aufwendiger Prozess, der etwas Zeit kostet, da Salesforce als Software-as-a-Service Anbieter auf Anfrage hin ein vollständig funktionierendes und umfangreiches System bereitstellen muss.

### 2.1.2 Betrachtung der Pipelinephase „test-scratch“

Diese Stage besitzt nur einen Job, ***test-apex***. Seine Durchschnittslaufzeit beträgt 9:38 Minuten. Auf der erstellten Scratch-Org werden alle Unit-Tests nacheinander durchgeführt, wodurch diese Zeit entsteht. Manche Tests dauern länger, da größere Funktionalitäten getestet werden, und nicht alle Unit-Tests sind optimiert geschrieben. Dennoch ist diese Stage im Vergleich zu den anderen beiden meist die kürzeste.

### 2.1.3 Betrachtung der Pipelinephase „deploy-review“

Auch „**deploy-review**“ hat nur einen Job, der in dieser Phase abzuarbeiten ist. ***deploy-review-package*** dient dazu, sich mit der Testumgebung zu verbinden und dort das erstellte Package zu installieren. Diese Installation dauert mit der vorherigen Authentifizierung zusammen 13:07 Minuten. Ist diese Stage abgeschlossen, wurden die neuen Funktionalitäten erfolgreich auf die Testumgebung deployed.

### 2.1.4 Betrachtung der für die Pipeline relevanten Total-Cycle-Time

Insgesamt hat die Pipeline eine Total-Cycle-Time von durchschnittlich 53:51 min. Es gibt immer mal schwache Abweichungen in beide Richtungen, doch meistens hält sich der Wert ungefähr bei 54 Minuten. Kleinere Abweichungen innerhalb der einzelnen Stages gleichen sich meist durch Abweichungen in einer anderen Stage aus. Der in der Statistik gemessene Bestwert für die TCT beträgt 43:42 Minuten, der schlechteste Wert 1:11:32 Stunden.

## 2.2 Lösungsansätze zum Reduzieren der TCT

### 2.2.1 Auslagern der Pipelinejobs auf automatisierte Auslöser

Der Entwickler hat die letzten Codeänderungen fertiggestellt, ein Kollege hat über die Merge-Request geschaut, bestätigt diese und startet die Pipeline. Die verschiedenen Jobs laufen nun nacheinander Schritt für Schritt durch, bis am Ende endlich der Code auf der Testumgebung des Kunden angekommen ist. Doch dieser langwierige Prozess kann in kleinere Teilprozesse gebrochen werden, welche wiederum zum Sparen von wertvoller Entwicklungszeit ausgelagert werden können. Während der Kollege bereits über die Merge-Request schaut, können bereits im Hintergrund die Apex-Tests laufen. So haben die Entwickler, wenn die Merge-Request akzeptiert wird, bereits die Anzeige, ob der Code wirklich zu einem funktionierenden Package gebaut werden kann.

Grundsätzlich ist der Implementierungaufwand zum Auslagern von Pipeline-Jobs nicht sehr groß, doch es erfordert eine durchdachte Konzeptionierung, um die richtigen Möglichkeiten zu erkennen, durch welche Verschiebung Zeit gewonnen werden könnte. Das macht diesen Ansatz zu einer geeigneten Lösung zur Reduzierung der TCT ohne große Implementierungen.

### 2.2.2 Optimierung der Apex-Tests

Es gibt viele verschiedene Ansätze, mit denen Apex-Tests effizienter gestaltet werden können. Beispielsweise kann durch das Pflegen von Testdaten in einer TastDataFactory sehr viel Zeit gewonnen werden, wenn die meisten der Tests vor einem Testdurchlauf nicht jeder individuell Daten anlegt und danach wieder löscht, sondern Testdaten einheitlich in einer zentralen Klasse von allen Testklassen genutzt werden können.

Auf diese Weise könnte die Laufzeit der Pipeline-Stage „test-scratch“ ein wenig gesenkt werden. Jedoch ist das mit viel Implementierungsaufwand verbunden, da viele Testklassen mit der neuen Funktionalität angepasst werden müssten. Im Vergleich dazu ist der Nutzen jedoch gering, da „test-scratch“ ohnehin aktuell nur durchschnittlich 10 Minuten braucht, um alle Apex-Tests abzuarbeiten. Wird die Zeit dort beispielsweise auf 7 Minuten reduziert, macht das in der Betrachtung der Total-Cicle-Time der gesamten Pipeline wenig Unterschied, abgesehen von einem kleinen Geschwindigkeits-Boost für die Package-Validierung, da dort auch noch einmal die Tests durchlaufen müssen.

### 2.2.3 Diffcheck und Parallelisierung im Build-Prozess

Salesforce unterstützt Modulares Development, auch wenn dessen Anwendung noch nicht weit verbreitet ist. Der Kerngedanke ist, dass statt einem großen „Happy Soup Monolythen“ lieber viele kleine Feature-Packages genutzt werden. Statt also allen Quellcode in einem großen Package zu haben, welches dann jedes Mal komplett gebaut werden muss, kann der Code besser in viele kleine Packages ausgelagert werden, die in sich geschlossen wie ein Micro-Service funktionieren. Der große Vorteil davon ist, dass dann ein Diffcheck eingesetzt werden kann, damit die Pipeline erkennt, welche Packages überhaupt modifiziert wurden und entsprechend auch nur den angepassten Code neu bauen muss. Doch darüber hinaus kann mithilfe von vielen Packages noch mehr Zeit gespart werden: in dem die Pipeline alle modifizierten Packages parallel baut, ist die maximale Laufzeit fast nur noch von der Bauzeit des größten, modifiziertem Package abhängig. Wenn also möglichst viele kleine Packages eingesetzt werden, reduziert das die gesamte Laufzeit des Jobs „create-package-version“, welcher mit einem Durchschnitt von über 30 Minuten der aktuell längste Job der Pipeline ist, drastisch.

Hier ist der Implementierungsaufwand auch hoch, vor allem, da das aktuell bestehende Monolythen-Package zuerst in viele kleine Module aufgeteilt werden muss und eine Technologie benötigt, die Parallelisierung im Bauprozess erlaubt. Jedoch ist mit diesem Lösungsansatz auch ein sehr großer Zeitgewinn möglich, was die TCT drastisch reduzieren kann.

### 2.2.4 Scratch-Org-Pooling

Scratch-Org-Pooling ist ein Prozess, bei dem gleichzeitig mehrere Scratch-Orgs erstellt werden, welche später entweder von Entwicklern oder von der Pipeline genutzt werden können. Der große Vorteil besteht darin, dass die lange Entstehungszeit einer Pipeline und das Deployen des aktuellen Codestandes beim Zugriff auf so eine vorbereitete Scratch-Org nicht mehr nötig ist, da diese bei der Erstellung im Pool ausführlich mit allen Daten bespielt wird. Das Erstellen der Pools kann automatisiert und unabhängig von der Deployment-Pipeline geschehen, wodurch die notwendige Zeit zur Entstehung und dem Deployment der Daten auf die Scratch-Org gespart wird. Dieser Ansatz ist also mit dem Ansatz zum Auslagern der Jobs verbunden.

Die initiale Implementierung von Scratch-Org-Pooling ist mit etwas Aufwand verbunden, aber der Gewinn ist auch außerhalb der Deployment-Pipeline bemerkbar. Es ermöglicht dem Entwickler beim Bearbeiten eines Neuen Tickets direkt mit einer vorbereiteten Scratch-Org zu starten was großen Mehrwert bringt. Aktuell dauert das Erstellen einer Scratch-Org zur Entwicklung genauso lange wie in der Pipeline und danach müssen sogar noch zusätzliche Setupskripte aufgerufen werden. Scratch-Org-Pooling würde also auch außerhalb der Deploymentpipeline Mehrwerte schaffen.

## 2.3 DX@Scale als Werkzeugkasten für die Reduzierung der TCT

DX@Scale ist eine Sammlung von Praktiken, Open-Source-Tools und Frameworks, die auf auf der Erfahrung von qualifizierten Salesforce-Entwicklern basiert und sich der Bereitstellung komplexer Salesforce-Anwendungen widmen. [3]

Sowohl CI/CD Prozesse, die Entwicklererfahrung als auch das Wissen über Modulares Development können mithilfe von DX@Scale erweitert werden. Werkzeuge zum Parallelen Bauen von Packages, zum Source-Tacking für Diffchecks und auch zum Scratch-Org-Pooling werden Salesforce Entwicklern kostenlos angeboten. Dazu kann eine Erweiterung von dem Salesforce Command-Line-Interface namens „sfpowerscripts“ eingesetzt werden, welche alle notwendigen Befehle zum orchestrieren solcher DevOps Prozesse beinhaltet. Somit ist DX@Scale im Rahmen der Projektarbeit gut geeignet, um für die Überarbeitung der CI/CD Pipeline eingesetzt zu werden und verschiedene Lösungsansätze aus den oben analysierten Möglichkeiten zu implementieren. [4]

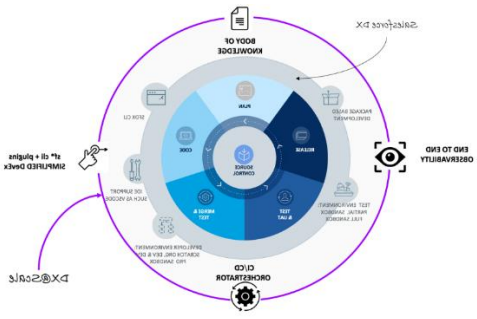

Abbildung 4: DX@Scale

[3] Vgl. [DXS]
[4] Vgl. [DXS]

## 2.4 Planung des Development-Workflows

Auch in der Konzeptionierung für die neue Pipelinearchitektur sollen parallel laufende Jobs eingesetzt werden, um möglichst geringe Wartezeiten für den Entwickler zu schaffen. Die ersten Jobs sollen jedoch bereits vor dem beginn des Deploymentprozesses laufen. Mit Hilfe des DX@Scale Salesforce Orchestrator werden in einem Job, der jede Nacht läuft, in einem Pool Scratch-Orgs für die Entwickler und auch für die CI/CD Prozesse der Deploymentpipeline vorbereitet. So kann ein Entwickler sich zu Tagesbeginn oder bei Bearbeitung eines neuen Tickets direkt eine vorbereitete Scratch-Org mit allen benötigten Packages und Settings aus dem Pool greifen und mit der Entwicklung beginnen. Auf seinem Feature-Branch nimmt er Änderungen am Code vor, und sowie die gewünschte Funktionalität erfolgreich implementiert wurde, erstellt er einen Merge-Request. Diese Erstellung eines Merge-Request wird ein automatisierter Auslöser für die erste Phase der Deploymentpipeline.

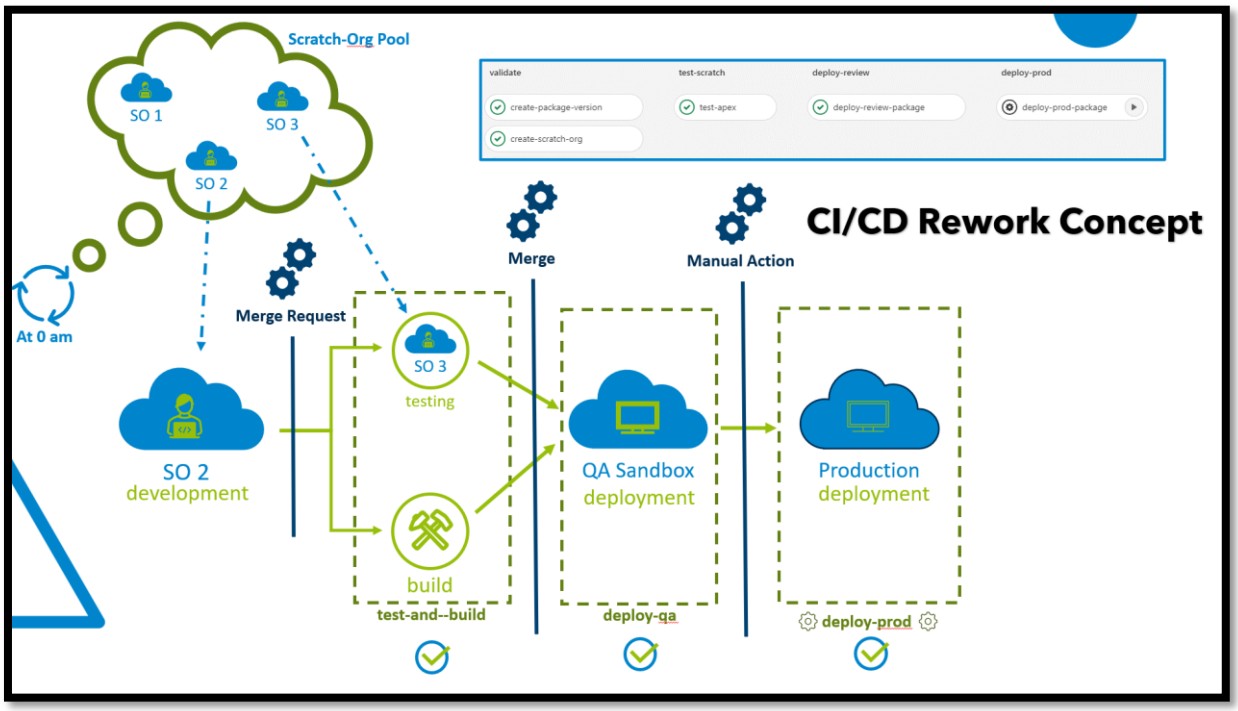


Abbildung 5: Entwurf der neuen Pipeline

Zwei Jobs sollen zu diesem Zeitpunkt parallel starten: zum Einen sollen die Apex-Tests auf einer vorbereiteten Scratch-Org aus dem Pool gestartet und ausgewertet werden, zum Anderen soll der Orchestrator bereits beginnen, ein Package aus den Codeänderungen zu bauen. Auf diese Art und Weise bekommt der Reviewer der Merge-Request direkt Feedback, ob alle Tests erfolgreich durchlaufen konnten, bevor er die Merge-Request bestätigt und somit

die Codeänderungen in den aktuellen Development-Branch aufnimmt. Mit diesem Punkt beginnt die für die Projektarbeit wichtigste Phase: nach dem Merge sollen die Packages auf die Testumgebung veröffentlicht werden, und diese Wartezeit soll möglichst reduziert werden. Das ist der Teil, ab welchem in der aktuellen Pipeline erst die ersten Jobs anlaufen. Das Ziel ist es, hier durch die beschriebene Auslagerung der Jobs nur noch das reine Deployment durchführen zu müssen, um somit die Änderungen zeitnah auf die Testumgebung zu bekommen. Als letztes soll dann nur noch das Deployment auf die Produktionsumgebung vorbereitet werden, jedoch soll dieses nicht automatisiert ausgelöst werden, sondern eine manuelle Bestätigung des Entwicklers erfordern. Auf diese Weise können die Kunden die neuen Funktionalitäten erst ausführlich auf der Testumgebung testen und im Notfall noch einmal Anpassungswünsche an die Entwickler stellen. Sobald jedoch der Kunde die neuen Features bestätigt hat, können die Entwickler mi einem Klick in der Pipeline die Veröffentlichung der gebauten Packages auf die Produktionsumgebung des Kunden starten.

# 3 Umsetzung

## 3.1 Nightly-Job zur automatisierten Erstellung eines Scratch-Org-Pools

Um einen Pipelinejob anzulegen, welcher automatisiert zu bestimmten Zeiten ausgeführt werden kann, ist es zuerst notwendig eine Möglichkeit einzuführen, mit der innerhalb der Pipeline individuelle Jobs direkt angesprochen und gestartet werden können. Dafür können in GitLab globale Variablen in Verbindung mit einem Regelsatz am entsprechenden Job verwendet werden. Zur Implementierung einer individuellen Ansprechmöglichkeit wird eine globale Variable ***TARGETTASKNAME*** angelegt, welche später mit einem Identifikator für den Job befüllt werden kann. In der Description der Variable können den Entwicklern gleich alle möglichen Identifikatoren mitgegeben werden, mit denen später individuelle Jobs angesprochen werden sollen.

```
21  ##########################################
22  #-- DX@Scale global Variables --#
23  variables:
24    TARGETTASKNAME:
25      value: ""
26      description: "The task name to run on demand, please pick ONE from this list:
    start-pipeline, schedule-prepare-ci-pool, schedule-prepare-dev-pool,
    schedule-clean-pool, schedule-report-so-pool, manual-delete-fetched-so,
    manual-release"
```

Abbildung 6: Globale Variable TARGETTASKNAME

Nun kann der Job zum Erstellen des Scratch-Org-Pools angelegt werden. Im Regelsatz wird festgelegt, dass dieser Job nur ausgeführt ist, wenn die Variable ***TARGETTASKNAME*** mit dem Inhalt **„schedule-prepare-ci-pool“** ausgefüllt wurde.

```
prepare-ci-pool:
  stage: dx-prepare-ci-pool
  rules:
    - if: '$TARGETTASKNAME == "schedule-prepare-ci-pool"'
```

Abbildung 7: Regelsatz für individuelle Jobs

Als nächstes erfolgt der Befehl zur automatisierten Erstellung von Scratch-Orgs. Dafür wird vom DX@Scale Orchestrator die Funktionalität ***:prepare*** genutzt. Diese bereitet eine beliebige Anzahl an Scratch-Orgs für ein vorher authentifiziertes DevHub vor. Die Orgs können in einer Konfigurationsdatei angepasst werden, welche Einstellungen wie die Größe des Pools, das Ablaufdatum der Scratch-Orgs, die Möglichkeit zum Source-Tracking und viele weitere Optionen bietet.

Ebenfalls kann in der Konfigurationsdatei, welche Packages auf den Scratch-Orgs vorinstalliert sein sollen. Diese Funktionalität ist essentiell für das Projekt, da sowohl die Entwickler als auch die Apex Tests im Continuous Deployment Step verschiedene Bereiche des Repos benötigen, um operieren zu können. Dafür kann ein Registry angegeben werden, in welchem bereits gebaute Artifakte liegen, die auf jeder Scratch-Org installiert werden sollen. Mehr dazu wird im nächsten Unterabschnitt beschrieben. Außerdem ist eine weitere Kernfunktionalität, dass in der Konfigurationsdatei zwei Scripte angegeben werden können, von denen eins vor dem Deployment der Packages und eins nach dem Deployment der Packages ausgeführt wird. Auch hier ist das Projekt auf die Verwendung von beiden angewiesen, um bereits automatisiert einen Store anzulegen und nach Deployment der Packages dem richtigen Nutzer ein PermissionSet zuzuweisen.

```
{
    "tag": "ci",
    "maxAllocation": 10,
    "expiry": 2,
    "batchSize": 10,
    "configFilePath": "config/project-scratch-def.json",
    "enableSourceTracking": true,
    "preDependencyInstallationScriptPath": "scripts/setup/quickstart-create-store2.sh",
    "postDeploymentScriptPath": "scripts/setup/postDeployment.sh",
    "installAll": true,
    "fetchArtifacts": {
        "npm": {
            "scope": [redacted]
        }
    }
}
```

Abbildung 8: Konfigurationsdatei für Scratch-Org-Pools

## 3.2 Aufsetzen eines NPM Registries für die Artefakte

Der Orchestrator verwendet Artefakte, um den aktuellsten Codestand widerzuspiegeln, welcher in den Development-Branch gemerged wurde. Diese Artefakte liegen in einem internen Registry. Wenn nun eine Scratch-Org-Erstellt wird, können aus diesem Registry die benötigten Artefakte installiert werden. Auch finden die Artefakte Anwendung beim Durchlauf der Apex Tests oder im Build-Prozess, da mithilfe von ihnen Änderungen an individuellen Packages erkannt werden. So baut der Orchestrator später nur die Packages, die sich nicht im Vergleich zu den Artefakten geändert haben.

Die dotSource nutzt intern als NPM Repository Manager die Software Nexus. Gültige Credentials müssen bei jedem Aufruf zur Authentifizierung genutzt werden, bevor auf die dort liegenden Daten zugegriffen werden kann. Dafür wird in GitLab eine Variable ***NEXUS_AUTH*** angelegt, in welcher die benötigten Daten zur Authentifizierung liegen. Um das Repository zu erreichen, muss eine Datei namens **.npmrc** angelegt werden. In dieser wird der Scope und der direkte Link dorthin mitgegeben, sowie die Authentifizierungsdaten. Ein NPM-Scope ist sozusagen der Bereich bzw. die Kategorie, in welche die Artefakte später gelegt werden sollen. Um eine solche Datei in der Pipeline vorzubereiten, wird der **echo** – Befehl genutzt, um die nötigen Inhalte in eine neue Datei zu schreiben. Diese muss nun überall angegeben werden, wo der Orchestrator auf das NPM Package Registry zugreifen soll.

```
- echo "Preparing .npmrc ..."
- echo "" > .npmrc
- echo "[redacted]/:always-auth=true" >> .npmrc
- echo "[redacted]/:_auth=${NEXUS_AUTH}" >> .npmrc
- echo "${NPM_SCOPE}:registry=[redacted]" >> .npmrc
- echo "NPM Scope = ${NPM_SCOPE}"
```

Abbildung 9: Code zur NPM Repository Authentifizierung

Der Orchestrator verwendet bei Angabe einer gültigen **.npmrc** Datei automatisch Befehle wie **npm publish** oder **npm install**. Um während der Pipeline also das NPM Registry zu verwenden, ist kein weiteres Wissen über NPM Package Registry notwendig, da hiervon alles automatisiert stattfindet. Sobald zum ersten Mal nach einem Build die Packages ins Regisry gepublished werden, erstellt sich ein neuer Teilbereich im Nexus unter dem angegebenen Scope. Die Variable für den Scope kann einfach als GitLab Variable angelegt werden, um den Scope notfalls später noch einmal anpassen zu können, ohne ihn im Quellcode hardcoden zu müssen.

## 3.3 Automatisierte Pipelinetrigger für Merge-Requests

```
#-- Automated Pipelinetriggers --#
.run-only-on-merge:
  rules:
    - if: $CI_PIPELINE_SOURCE == "schedule"
      when: never
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
      when: never
    - if: '$TARGETTASKNAME == "manual-delete-fetched-so"'
      when: never
    - if: '$TARGETTASKNAME == "manual-release"'
      when: never
    - if: '($CI_COMMIT_BRANCH == "DX@Scale_Main_Branch" || $CI_COMMIT_BRANCH=~ /^release*$/)'
      when: on_success
    - when: never

.run-on-merge-request:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event" && ($CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "DX@Scale_Main_Branch")'
      when: on_success
    - if: $CI_PIPELINE_SOURCE == "schedule"
      when: never
##########################################
```

Abbildung 10: Automatisierte Pipelinetrigger

Um Phase 1 und Phase 2 der Pipeline automatisiert aufrufen zu können, werden zwei weitere Regelsätze angelegt. In ihnen wird erkannt, ob eine Merge-Request erstellt bzw. akzeptiert wurde. In GitLab kann für einen Job später eingestellt werden, dass er einen solchen Regelsatz **extended**, was so viel heißt wie dass der Job nur ausgeführt wird, wenn genau die angegebenen Bedingungen zutreffen. In den Regelsätzen wird die GitLab Umgebungsvariable ***CI_PIPELINE_SOURCE*** verwendet, um zu erkennen, wodurch die Pipeline gestartet wurde.

## 3.4 Apex-Tests und Build

Sowohl die Implementierung von automatisierten Pipeline-Tests, als auch der Build-Funktion sind mit dem DX@Scale Orchestrator umfangreich konfigurierbar. Das Werkzeug bietet vordefinierte Logik für beide Funktionalitäten und kann ohne große Vorkenntnisse angewandt werden. So wird eine komplexe und zeitaufwändige Implementierung von einer eigenen Logik zum parallelen Bauen und Identifizieren der richtigen Packages gespart. Stattdessen müssen lediglich die richtigen Parameter für den Orchestrator bereitgestellt werden.

Zum Durchlaufen und zur Analyse der Apex-Tests wird der Befehl **:validate** vom Orchestrator verwendet. Hier werden lediglich der Benutzernamen des vorher authentifizierten DevHubs an, sowie den Namen des erstellten Pools, welchen man in der Konfigurationsdatei festlegen konnte, angegeben. Der Befehl **:build** erfordert ein wenig mehr Parameter. Neben dem DevHub wird der aktuelle Branch angegeben, in dem die GitLab Umgabungsvariable ***CI_COMMIT_BRANCH*** genutzt wird. Dazu gehört noch eine Buildnummer, und mit der Option --diffcheck  kann festgelegt werden, dass nur die Packages gebaut wurden, die angepasst wurden. So kann sehr effizient Zeit gespaart werden, da nichtmehr bei jedem Durchlauf der gesamte Sourcecode in Packeges umgewandelt wird.

```
207     # Build and Publish
208     - echo "Build Production Ready Packages and Publish"
209     - sfdx sfpowerscripts:orchestrator:build -v "DEVHUB" --branch "$CI_COMMIT_BRANCH"
    --buildnumber $CI_PIPELINE_ID --diffcheck
210     - sfdx sfpowerscripts:orchestrator:publish -v "DEVHUB" --npm  --npmrcpath .npmrc
    --scope "$NPM_SCOPE" --gittag --pushgittag
211   environment:
212     name: build-env
213   image: ghcr.io/dxatscale/sfpowerscripts
214   resource_group: build
```

Abbildung 11: Build und Publish

Direkt nach dem **:build** Befehl folgt der **:publish** Befehl, welcher die gebauten Packages in Artefakte wandelt und sie in das NPM Packages Registry hochlädt. Hier sollte neben dem Git-Tag unbedingt auch der Pfad zu der erstellten **.npmrc** Datei angegeben werden. Dazu wird ebenfalls der Scope aus der GitLab Variable ausgelesen und als Parameter übermittelt. Sowie der **:publish** Befehl durchgeführt wurde, werden die erfolgreich gebauten Packages im dotSource Nexus gefunden.

## 3.5 QA und PROD Deployment

Der Orchestrator bietet die Funktion :release an, welche anhand einer als Parameter angegebenen Release-Konfigurationsdatei die richtigen Artefakte aus dem NPM Package Registry holt und auf gewünschte Salesforce-Orgs deployed. Hierfür ist selbstverständlich wieder eine funktionstüchtige **.npmrc** Datei anzugeben. Neben dem Scope wird ebenfalls wieder die Org, auf welche deployed werden soll, angegeben. Zum Schluss besteht mittels der Optionen **--generatechangelog** und **--branchname** die Möglichkeit, automatisch einen Changelog zu erstellen und unter einem beliebigen Branchnamen in das GitLab Repository zu laden. Soll auf eine Produktionsumgebung deployed werden, ist es vorher notwendig das Package zu promoten. Dafür kann in der Release-Konfigurationsdatei festgelegt werden, dass die Artefakte vor der Installation unter einer geeigneten Version promoted werden.

```
232     # Release to QA
233     - authenticate QA $DEVQA_AUTH_URL
234     - sfdx sfpowerscripts:orchestrator:release -u "QA" -p release-definitions/
release-1.0.yml --npm --npmrcpath .npmrc --scope "$NPM_SCOPE"  --generatechangelog
--branchname changelog
235   environment:
236     name: QA
237   image: ghcr.io/dxatscale/sfpowerscripts
238   resource_group: QA
```

Abbildung 12: Deployment mit dem Orchestrator

# 4 Analyse der Ergebnisse

## 4.1 Zusammenfassung und Auswertung

Durch die Integration des DX@Scale Salesforce Orchestrators konnten drei von den vier vorgestellten Optimierungsansätzen angewandt werden. Die Implementierung ermöglicht den Entwicklern nun die Anwendung von modularem Development in parallel gebauten Packages, was langfristig ein großer Gewinn durch die Zeitersparnis von Diffchecks und paralleler Abarbeitung ist. Der Grundstein wurde gelegt, damit aus dem aktuellen Monolythen-Repository (auch Happy-Soup-Repo genannt) in der Zukunft ein nach Features aufgeteiltes, packagebasiertes Repository entstehen kann.

Der aktuell größte zu verzeichnende Zeitgewinn ist durch das automatisierte Erstellen von Scratch-Orgs mittels dem implementierten Scratch-Org-Pooling zu verzeichnen. Zum Einen entsteht Mehrwert dadurch, dass Entwickler bei Beginn ihrer Arbeit an einem Feature-Branch direkt mit einer vorbereiteten Scratch-Org arbeiten können, welche normalerweise mit einem c.a. 30 Minuten langen Aufwand zum Erstellen der Org und zum Deployen des aktuellen Codestandes und der Metadaten verbunden ist. Zum Anderen können die notwendigen Apex-Tests in der Deployment-Pipeline direkt auf den vorbereiteten Scratch-Orgs aus dem Pool durchgeführt werden. Damit muss zum Testen nicht mehr wie im aktuellen Deploymentprozess auf das Erstellen einer Scratch-Org gewartet werden, wodurch der Job direkt an den Anfang der Pipeline gezogen werden kann. So kann er parallel zu dem ***build*** Job laufen, und die gesamte Phase ***test-scratch*** aus der aktuellen Pipeline, mit ihrer Durchschnittslaufzeit von 9:38 Minuten, kann erspart werden.

Weiherhin sind Zeitgewinne durch die Diffcheck-Funktionalität zu verzeichnen. Dadurch, dass mit der neuen Pipeline statt dem gesamten Codestand nur noch die Packages gebaut werden, die geändert wurden, profitieren vor allem kleine Packages davon. Finden beispielsweise Änderungen in einem kleinen Metadaten-Package, statt dem großen Monolythen-Package statt, zeigt die Deploymentpipeline anstelle der ehemalig notwendigen Durchschnittslaufzeit von 31:07 Minuten nun nur noch 05:34 Minuten an. Doch nicht nur der Build-Prozess profitiert von der Diffcheck-Funktionalität, sondern die Deployment-Jobs auf die Testumgebung und die Produktionsumgebung des Kunden ebenfalls.

## 4.2 Ausblick

Um den maximalen Mehrwert aus dem Projekt zu holen ist es nun im nächsten Schritt notwendig, das aktuell bestehende Code-Repository in viele kleine, Business-Unit-orientierte Packages aufzubrechen. Ein solches Refactoring ist jedoch mit etwas Aufwand verbunden, da für jedes entstehende Package streng auf die Dependencies und die mögliche Dopplung von Metadaten in verschiedenen Packages geachtet werden muss. Nur wenn die Umstrukturierung gut geplant und durchdacht durchgeführt wird, können Fehler und Probleme vermieden werden. Dafür ist jedoch der Gewinn aus dem Refactoring groß: je früher angefangen wird, mit kleinen Packages zu arbeiten, desto weniger muss langfristig refactored werden und desto mehr Laufzeit kann ab diesem Zeitpunkt schon gespart werden.

## 4.3 Wiederverwendbarkeit der gewonnenen Erkenntnisse

Die gewonnenen Erkenntnisse sind nicht nur an des Projekt gebunden, sondern können für alle zukünftigen Salesforce Lightning Projekte der dotSource verwendet werden. Gesammelte Erfahrung und die geschaffene Infrastruktur für modulares Development können für neue Projekte bereits von Anfang an Eingesetzt werden, um den Deploymentprozess und den Development-Workflow zu optimieren. Selbst wenn nicht die komplette CI/CD Pipeline übernommen wird, können Teilprozesse wie das automatisierte Scratch-Org Pooling eingesetzt werden, um an verschiedenen Stellen wertvolle Entwicklerzeit zu sparen.

## 4.4 Fazit

Im Rahmen der Projektarbeit konnte nicht nur viel neues Wissen zu DevOps Prozessen für das Unternehmen gewonnen werden, sondern auch effiziente neue Automatisierungsprozesse wie Zahnräder in das bestehende, komplexe System integriert werden. Die Zielsetzung der wissenschaftlichen Arbeit, Optimierungsmöglichkeiten zu identifizieren und aktiv in das Projekt zu integrieren wurde erfolgreich erreicht und wie gewünscht ist ein Geschwindigkeitsgewinn zu erkennen. Vor allem konnte das Vorhaben trotz der vielen bestehenden, voneinander abhängigen Abläufen im Projekt umgesetzt werde. Damit wurde eine neue, moderne Technologie für das Entwicklerteam erschlossen, welches fortan im Kundenprojekt und weiteren dotSource Projekten eingesetzt wird.

# 5 Abschließende Worte

Die Projektarbeit zeigt, dass eine Umstellung manchmal viel Aufwand und große Änderungen für die Entwickler bedeuten kann. Manchmal kann ein großer Faktor bei solch einer Überarbeitung auch die Motivation von sich selbst und den Teamkollegen sein, und es wird umso wichtiger, dass man sich stehts vor Augen hält, wofür man diese Arbeit macht. Jedoch wachsen wir Entwicklerteams nur, in dem wir uns weiterentwickeln und bereit sind, neue Dinge auszuprobieren. Wir streben danach, besser und effizienter zu werden, und gerade weil wir daran glauben sind wir bereit, solche bestehenden Prozesse auch immer wieder zu überdenken und neu zu phantasieren. Als Entwickler ist es nicht nur unsere Aufgabe, neues Wissen zu erlangen, sondern auch darüber nachzudenken, wie man diese neuen Erkenntnisse in bereits bestehende Prozesse integrieren kann. Gerade das spiegelt sich in dieser Projektarbeit wieder, und sie soll eine Motivation für andere Entwickler sein, selbst diesen Schritt zu gehen. Denn nur wenn wir bereit sind uns in das Unbekannte zu begeben und daraus zu lernen, können wir unseren eigenen Horizont erweitern. Wir stehen nicht still. Denn der wichtigste Schritt im Leben eines Entwicklers, ist der **nächste**.

## Literaturverzeichnis


[SAF22] Milad Safar: *Was verbirgt sich hinter dem Begriff DevOps,* 2022, https://weissenberg-group.de/was-ist-devops/

[PIE23] Frank Pientka: *Wie DevOps die IT beschleunigen*, 2023, https://www.computerwoche.de/a/wie-devops-die-it-beschleunigen,3071433

[DXS] DX@Scale online Dokumentation, 2023, https://docs.dxatscale.io/


## Bildquellen


https://3917811566-files.gitbook.io/~/files/v0/b/gitbook-x-prod.appspot.com/o/spaces%2FIT89vt0q5am7VFg1WIGJ%2Fuploads%2F4gPPu9rr8Xr3pAzANg49%2Fdxatscale.png?alt=media&token=53580433-3d76-4219-a469-25de5eb65f01

# Anlagenverzeichnis

Anlage 1: CI/CD Pipeline Runtimes

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| validate | validate | test-scratch | deploy-review | | Pipeline |
| create-package-version | create-scratch-org | test-apex | deploy-review-package | | TCT |
| 0:31:07 | 00:17:53 | 00:09:38 | 00:13:07 | | 0:53:51 |
| | | | | | |
| 0:29:07 | 00:17:08 | 00:09:38 | 00:14:38 | | 0:53:23 |
| 00:29:37 | 00:17:45 | 00:09:40 | 00:13:06 | | 0:52:23 |
| 00:31:40 | 00:15:53 | 00:08:49 | 00:10:37 | | 0:51:06 |
| 00:32:40 | 00:16:02 | 00:09:57 | 00:11:57 | | 0:54:34 |
| 00:25:37 | 00:21:25 | 00:09:29 | 00:10:43 | | 0:45:49 |
| 00:28:38 | 00:17:17 | 00:04:07 | 00:10:57 | | 0:43:42 |
| 00:33:44 | 00:18:12 | 00:09:53 | 00:13:11 | | 0:56:48 |
| 00:33:42 | 00:18:31 | 00:10:50 | 00:10:25 | | 0:54:57 |
| 00:32:07 | 00:19:36 | 00:14:16 | 00:07:56 | | 0:54:19 |
| 00:34:17 | 00:16:57 | 00:09:38 | 00:27:37 | | 1:11:32 |

Ehrenwörtliche Erklärung

# Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Projektarbeit/Studienarbeit/Bachelorarbeit mit dem Thema:

**Reduzierung der Laufzeit einer Salesforce Deployment Pipeline**

ohne fremde Hilfe angefertigt habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und

3. dass ich meine Projektarbeit/Studienarbeit/Bachelorarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

_____ _____

Ort, Datum Unterschrift