Methoden zur systematischen Implementierung der Continuous Practices

Carl Schünemann
Informatik, B.Sc.
Technische Hochschule Ingolstadt
Matr. 00107827

***Zusammenfassung*—Continuous Practices gelten als Standard für Automatisierungstechniken in der Softwareindustrie. Sie ermöglichen es, kontinuierlich hochwertige Softwareprodukte entwickeln und ausliefern zu können. Dabei ist die technische Teilpraktik Continous Delivery besonders kostenintensiv. Automatisierte Schritte erzeugen in Continuous-Delivery-Pipelines aus Codeänderungen auslieferbare Softwareprodukte. Diese Pipelines werden in der Regel in Job Execution Systemen umgesetzt und ausgeführt. Diese Arbeit stellt im Rahmen einer Literaturübersicht kostensparende Möglichkeiten zur systematischen Planung von Continous Delivery Pipelines und zur objektiven Auswahl eines dazu gehörigen, anforderungsgerechten Job Execution Systems vor.**

***Index Terms*—Continuous Practices, Continuous Delivery, UML, Cinders, Job Execution System**

I. Einführung

In der Vergangenheit war die Umwandlung von Quellcodezeilen in funktionierende, ausgereifte Produkte in der Regel ein manueller Prozess, abhängig von organisatorischen Zuständigkeiten und der Unternehmensarchitektur. [19] Heute gelten *Continuous Practices* als Standard der modernen Software-Entwicklung. [21] Mithilfe organisatorischer und technischer Maßnahmen wird die Qualität eines Software-Produkts kontinuierlich verbessert und automatisch dem Kunden bereitgestellt. [9, 21]

Auf organisatorischer Ebene erfordert dies die Einführung von *Continuous Integration*, eine Sammlung von Verhaltensweisen für den Arbeitsalltag der Teammitglieder. Die technische Komponente der Prozessoptimierung durch Continuous Practices ist *Continuous Delivery*. In *Continuous-Delivery-Pipelines* findet die Umwandlung von neuen Quellcodeänderungen in verifizierte Softwareprodukte statt. Um die Continuous Practices technisch in einem Projekt zu implementieren, müssen zunächst die notwendigen Pipeline-Schritte, sowie deren Reihenfolge und Abhängigkeiten definiert werden. Basierend darauf müssen die entwickelten Tools und Skripte in einem ausgewählten *Job Execution System* zu konkreten Pipelines umgesetzt werden, beispielsweise in *Jenkins*.

Demnach ist die Einführung dieser Praktiken keine triviale Aufgabe und sowohl von technischen als auch organisatorischen Rahmenbedingungen abhängig. [13, 17] Trotzdem investieren Unternehmen Ressourcen in die Einführung von Continuous Practices. So schätzen Google und Mozilla die Kosten ihrer Continuous-Delivery-Pipelines auf mehrere Millionen Dollar. [5] Ståhl und Bosch identifizierten dabei eine unkoordinierte technische Implementierung der Continuous Practices als Kostenfaktor. [19]

Diese Arbeit bietet eine Literaturübersicht über Möglichkeiten, die technischen Komponenten der Continuous Practices systematisch zu implementieren. Dazu werden im ersten Teil Möglichkeiten und Frameworks zur systematischen Planung der Abläufe in CD-Pipelines identifiziert. Im zweiten Teil dieser Arbeit werden Kriterien zur systematischen und objektiven Auswahl eines Job Execution Systems gesammelt, um basierend auf den Projektanforderungen dort CD-Pipelines umzusetzen.

II. Begriffsdefinitionen

Continuous Practices entstanden Ende der 1990er Jahre als neue Methode der Softwareentwicklung. Eine Reihe von Praktiken ermöglichen es Unternehmen, neue Funktionen kontinuierlich als verifizierte Softwareprodukt zu veröffentlichen. [17] Sowohl in der Literatur als auch in Unternehmen werden diese Praktiken unterschiedlich interpretiert und umgesetzt. Die nachfolgenden Definitionen wurden von Ståhl, Martensson und Bosch erarbeitetet und sind in der Forschung akzeptiert. [21] Der Zusammenhang der Praktiken ist in Abbildung 1 dargestellt.

A. Continuous Integration und Continuous Delivery

Bei der *Continuous Integration*, kurz *CI* handelt es sich um eine Entwicklerpraxis, die den täglichen Arbeitsablauf von Entwicklungsteams bestimmt. [9, 15, 21]. Teammitglieder integrieren dabei häufig ihre Arbeitsergebnisse, beispielsweise mit einem zentral zugänglichen Repository. [5]

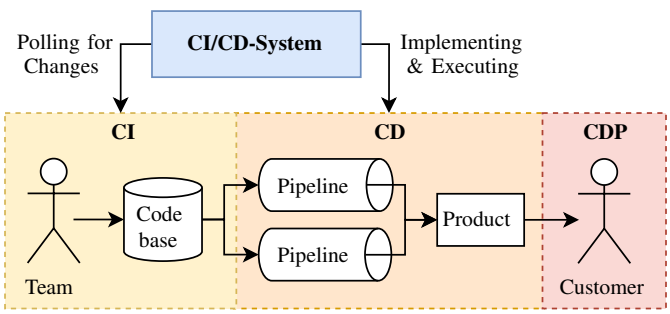

Abbildung 1. Zusammenhang der *Continuous Practices*

Im Gegensatz dazu ist *Continuous Delivery*, kurz *CD* eine Entwicklungspraxis. Dabei handelt es sich um eine Continuous Practice, die allein durch technische Maßnahmen umgesetzt werden kann. Eine Reihe automatisierter Tools und Skripte transformiert neue Codeänderungen in ein auslieferbares Produkt für den Kunden. Dieser Prozess findet in einer sogenannten *Continuous-Delivery-Pipeline*, kurz *CD-Pipeline* oder *Pipeline* statt. Im Rahmen der Continous Delivery besitzt das Unternehmensmanagement stets die diskrete Entscheidungsgewalt, ob eine Produkt tatsächlich an den Kunden ausgeliefert wird. Sobald auch diese Entscheidung automatisiert wird, spricht man von *Continuous Deployment*, kurz *CDP*. [21]. Die Grenze zwischen CI und CD wird dort definiert, wo die Aktivitäten des Entwicklers aufhören und die automatisierten Pipelineschritte beginnen. [10, 13, 21]

Die Bestandteile von CD-Pipelines variieren für jedes Projekt. Das Zusammensetzen der Pipeline-Bestandteile zu spezifischen CD-Pipelines findet im *Continuous Integration and Delivery System*, kurz *CI/CD-System* statt. Dabei wird die Codebasis regelmäßig auf neue Integrationen überprüft und Pipelines für unterschiedliche Anwendungsfälle ausgeführt. [5, 19]

B. Job Execution System

Eine mögliche Implementierung eines CI/CD-Systems ist das in Abbildung 2 dargestellte *Job Execution System*, eines der wichtigsten Werkzeuge für moderne Softwareentwicklungsteams.

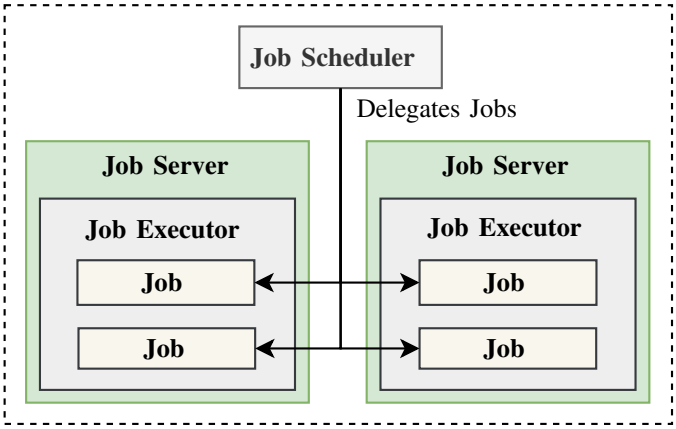

Abbildung 2. Ein *Job Execution System* bestehend aus *Job Scheduler* und zwei *Job Servern* mit jeweils einem *Job Executor*. [12]

Dabei muss ein *Job*, hier eine CD-Pipeline, von einem *Job Executor* ausgeführt werden. Bei einem manuellen Job ist der Job Executor ein menschlicher Benutzer. Job Executor können auf einem entfernten Rechner laufen und parallelisiert werden. Der Rechner, auf dem ein Job Executor installiert ist, wird als *Job Server* bezeichnet. Die Entscheidung, wann und von welchem Job Executor ein Job ausgeführt werden soll, wird vom *Job Scheduler* getroffen. Die Kombination dieser drei Elemente bildet das *Job Execution System*. Job Execution Systeme, wie *Jenkins* und *TeamCity* gibt es von unterschiedlichen Anbietern; sie werden selten selbst entwickelt. [12]

III. Systematische Planung von CD-Pipelines

Continous Delivery ist die technisch umsetzbare Praktik der Continuous Practices. [21] Die Schritte einer Continuous Delivery Pipeline wandeln den Sourcecode in ein auslieferbares Produkt um. Vor Entwicklung der dazu notwendigen Skripte und Tools muss der Aufbau der Pipeline strukturiert werden. Dabei gilt es insbesondere die technischen Rahmenbedingungen des Projekts zu berücksichtigen. Wenn die Entwicklung von CD-Pipelines nicht koordiniert vorbereitet wird, entstehen bei komplexen Projekten zusätzliche Kosten. [19] Nachfolgend werden im Rahmen einer Literaturrecherche Möglichkeiten zur systematischen Planung und Modellierung von CD-Pipelines identifiziert und bezüglich ihrer praktischen Relevanz bewertet.

A. Berücksichtigung von Erfahrungswerten

Bei der Entwicklung der ersten CD-Pipelines orientierte man sich insbesondere an in der Praxis gesammelten Erfahrungswerten. Duvall et al. veröffentlichten 2007 mit dem sogenannten *Integrationsknopf* eine der ersten *CD Best Practices*. Das Drücken dieses Knopfes symbolisiert das Auslösen einer CD-Pipeline. Wie in Abbildunug 3 dargestellt, umfasst dies das Erstellen des Software-Produkts inklusive Integration von Datenbanken, das Testen des Produkts, Codeanalysen, Generierung von Dokumenten sowie Rückmeldung und Bereitstellen der Software für das Team. [5] Das Konzept des

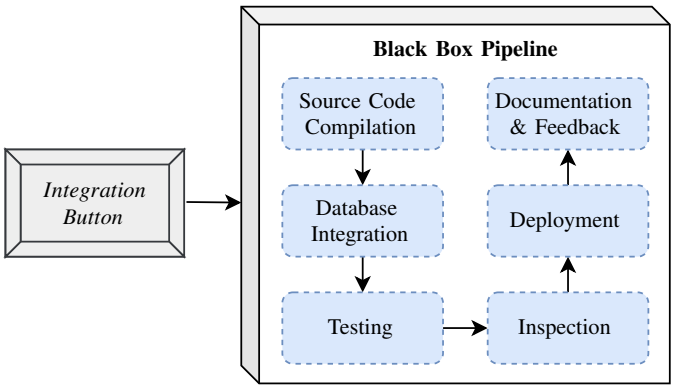

Abbildung 3. Das Drücken des *Integrationsknopfes* löst die Pipeline aus. [5]

Integrationsknopfes beschreibt jedoch keine konkrete Herangehensweise zur Entwicklung dieser kritischen Pipeline-Bestandteile. Zudem wird ausschließlich die Einführung von CD in neuen Projekten behandelt. Diese Lücke versuchen Gupta et al. durch ein Berwertungsframework für Pipelines in laufenden Projekten zu schließen. Dabei wird deren funktionale Vollständigkeit anhand 18 in der Literatur identifizierter Kategorien bewertet. [7]

Beide Ansätze unterstützen die Entwicklung von CD-Pipelines mithilfe von Richtlinien und empfohlenen Herangehensweisen. Elemente zur Planung komplexer Abläufe in CD-Pipelines fehlen jedoch. Somit eignen sich diese Ansätze nur für Projekte mit begrenztem Umfang.

B. Modellierung mit UML

Grundelement des Software-Engineerings zur visuellen Modellierung komplexer Abläufe in Systemen ist die *Unified Modeling Language*, kurz *UML*. Entstanden in den frühen 1990er Jahren, ist UML heute der de-facto Standard für die Modellierung objektorientierter Prozesse. Mittels UML können Prozesse innerhalb eines Systems durch unterschiedliche Diagramme dargestellt werden. Für jeden UML-Diagrammtyp gibt es standardisierte Notationen. [14] In der Praxis wird jedoch beobachtet, dass die UML-Notation an den Kontext und die Problemdomäne des Systems angepasst wird. [4]

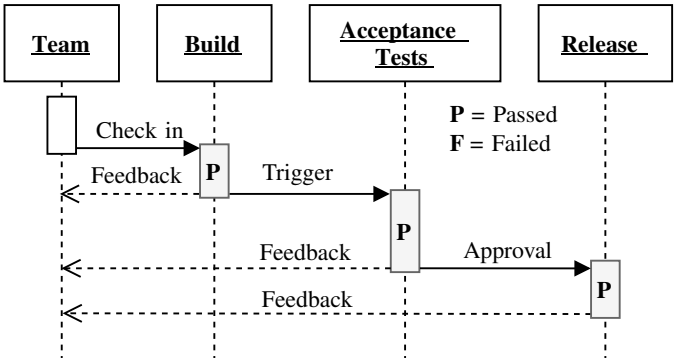

Abbildung 4. Pipeline-Modellierung mittels UML-Sequenzdiagramm nach Humble [10].

Gleiches lässt sich bei der Modellierung von CD-Pipelines beobachten: Humble vermittelt in seinem Buch *Continuous Delivery* den grundlegenden Aufbau von CD-Pipelines mithilfe von UML-Sequenzdiagrammen. Anhand von Abbildung 4 kann man erkennen, wie Humble von der UML-Notation abweicht. [10] Entgegen der standardisierten Notation verwendet Humble keine konkreten Instanzen als kommunizierende Einheiten, sondern allgemeingültige Klassen. So löst im Sequenzdiagramm in Abbildung 4 das *Team* über einen *Check in* den *Build* aus. Es wird nicht konkretisiert, ob es sich beispielsweise um einen vollständigen Systembuild handelt oder nur eine Teilintegration der Komponenten. Weiterhin fügt Humble den Sequenzdiagrammen neue Elemente hinzu. So ergänzt er für jede Phase der CD-Pipeline einen Erfolgsstatus mit den Werten *Passed* oder *Failed*.

Auch Sommerville orientiert sich im Buch *Software Engineering* bei der Modellierung von Pipeline-Abläufen an der UML-Notation. Eine Abweichung ist beispielsweise die Kombination von UML-Aktivitätsdiagramm und Klassendiagramm. [18]

C. Modellierung mit Cinders

Aus der Beobachtung, dass UML nicht unverändert zur Modellierung von CD-Pipelines verwendet wird, entwickelte eine von Ståhl und Bosch geführte Forschergruppe die Modellierungsframeworks *Automated Software Integration Flows*, kurz *ASIF* und *Continuous Integration Visualization Technique*, kurz *CIViT*. [3, 16] Mit diesen Frameworks können sowohl die funktionale Implementierung als auch die Evaluierungsaktivitäten von CD-Pipelines dokumentiert werden. Die Funktionalitäten beider Konzepte wurden im Modellierungsframework *Cinders* kombiniert und erweitert. [19]

Bestandteile

Grundelemente bilden die von ASIF übernommenen *Nodes*. *Activity*- (Rechteck) und *Task Node* (Oktagon) stellen die ausgeführten Schritte in der CD-Pipeline dar. Diese können von externen Ereignissen, den *Trigger Nodes* (Kreis), ausgelöst werden. Datenquellen werden durch den *Repository Node* (Dreieck) dargestellt. [19]

Diese Entitäten sind in vier sogenannten *Viewpoints* gruppiert, wobei jeder Viewpoint die Pipeline aus einem anderen Gesichtspunkt betrachtet, darstellt. Im *Causality Viewpoint* werden die kausalen Abhängigkeiten zwischen den Nodes als Pfeile dargestellt. Datenflüsse werden im *Production Line Viewpoint* durch gestrichelte Pfeile modelliert. In beiden Viewpoints stehen die Nodes für allgemeingültige Klassen. Erst beim *Instances Viewpoint* sind die Nodes vom Kontext abhängige, konkrete Instanzen. Wie in Abbildung 5 links dargestellt, kombiniert dieser Viewpoint die Informationen des Causality- und Product Line Viewpoints. Im *Test Capabilities Viewpoint*, eine Vereinfachung des CIViT-Frameworks, werden die Testaktivitäten der CD-Pipeline visualisiert. Wie rechts in Abbildung 5 dargestellt, werden dazu die Activity-Nodes in einem zweidimensionalen Koordinatensystem auf die Achsen *Systemvollständigkeit* und *Testdauer* abgebildet. [19]

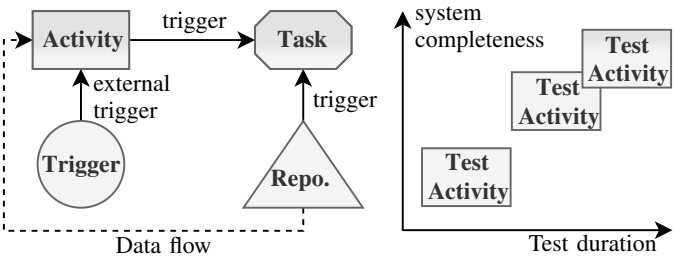

Abbildung 5. *Instances Viewpoint* (links) und *Test Capabilities Viewpoint* (rechts).

Vergleichbar mit den Viewpoints sind die UML-Diagrammtypen. Durch unterschiedliche UML-Diagramme können die Prozesse eines Systems aus unterschiedlichen Sichtweisen abgebildet werden.

Jeder Viewpoint erhält über sogenannte *Layers* optionale Informationen. In Abbildung 6 sind drei fundamentale Layers dargestellt. Die *Attributes Layer* fügt einem Node quantitative und qualitative Attribute hinzu. Durch die *Physical Layer* erhalten die Nodes je nach physikalischer Umgebung farbige Hintergründe. Schließlich wird durch die *Automation Layer* die Kontur der Nodes je nach Automatisierungsgrad rot (vollständig automatisiert), gelb (teilautomatisiert) oder grün (manuell) gefärbt. [19]

D. Praktische Relevanz

Für Projekte mit begrenztem Umfang ist es ausreichend, sich bei der Entwicklung von Continuous-Delivery-Pipelines

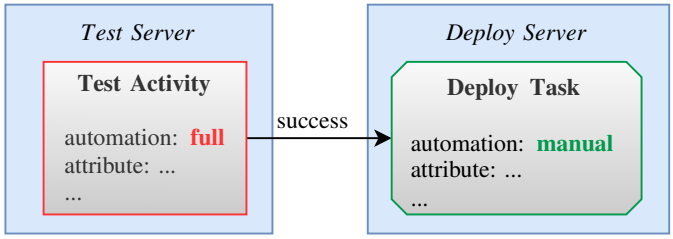

Abbildung 6. Ein Activity- und Task-Node mit *Attributes-*, *Physical-* und *Automation Layer*.

an gesammelten Erfahrungswerten zu orientieren, wie es Duvall et al. mit dem Konzept des Integrationsknopfs vorschlagen. Projekte mit komplexem Auslieferungsprozess erfordern hingegen vor der Umsetzung der Pipelines eine systematische Planung der Pipeline-Schritte. Dazu wird in der Literatur auf UML zurückgegriffen, ein etabliertes Werkzeug des Software-Engineerings. In der Regel wird von der standardisierten UML-Notation abgewichen. [10, 18] Eine von Ståhl und Bosch geführte Forschungsgruppe nahm dies als Begründung, um für CD neue Modellierungsmethoden zu entwickeln. So entstanden die Frameworks ASIF, CIViT und Cinders — die einzigen in der Literatur bekannten Continuous Delivery-Modellierungsframeworks. [3, 16, 19]

Alle drei Frameworks wurden auf gleiche Weise evaluiert. Die Autoren führten in Projekten unterschiedlicher Unternehmen Workshops durch. Jedes Team sollte die CD-Pipeline ihres Projekts mit dem jeweiligen Framework modellieren. Anschließend wurden mithilfe von Fragebögen Verbesserungsvorschläge eingeholt. Basierend darauf wurden die Frameworks angepasst sowie Vor- und Nachteile identifiziert. [3, 16, 19] Darüber hinaus wurde Cinders in zwei Masterarbeiten der Technischen Hochschule Chalmers angewendet und bewertet. [1, 6]

Hingegen gibt es in der Literatur keinen Nachweis über den praktischen Einsatz dieser Frameworks in der Industrie. Aufgrund der spärlichen Verwendung von Cinders lassen sich die in den Workshops und Masterarbeiten identifizierten Eigenschaften daher nicht bestätigen. Weiterhin ist anzumerken, dass an ASIF, CIViT und Cinders die gleichen Autoren gearbeitet haben. Somit wurden die Frameworks nicht unabhängig voneinander entwickelt.

Mit Cinders sollte ein Framework ausschließlich zur Modellierung von CD-Pipelines geschaffen werden. Im Unterschied dazu nutzten Humble und Sommerville unabhängig voneinander das bereits im Software-Engineering etablierte Modellierungsframework UML. Sie verwendeten unterschiedliche UML-Diagramme und passten die UML-Notation basierend auf den gegebenen Rahmenbedingungen für spezifische Pipelines an.

Zwar kritisieren Ståhl et al. die Abweichung von der UML-Notation zur Modellierung von CD-Pipelines, die generische Anwendbarkeit von UML auf eine Vielzahl von Problemdomänen identifizierten sie jedoch als Stärke. Sie bezeichnen UML-Diagramme deshalb als Alternative zu ihren Modellierungsframeworks. [19] Somit können sowohl mit UML als auch mit Cinders CD-Pipelines basierend auf den Anforderungen eines Projekts modelliert werden.

Um die Relevanz von Cinders zu bewerten, motivieren Ståhl et al. die CI/CD-Gemeinschaft des Software Engineerings, unabhängige empirische Fallstudien oder Umfragen durchzuführen, die bis dato jedoch nicht erfolgt sind. [20]

IV. Systematische Auswahl eines Job Execution Systems

Nachdem ein Team basierend auf einem strukturierten Plan die Tools und Skripte der Pipeline-Bestandteile entwickelt hat, müssen diese zu konkreten Conitnuous-Delivery Pipelines zusammengefügt werden. Dies findet typischerweise in einem Job Execution System statt. Analog zum systematischen Vorgehen bei der Pipeline-Modellierung, muss das Job Execution System für die Anforderungen des Projekts systematisch und objektiv ausgewählt werden.

Auf Internetseiten werden oft die *Besten CI/CD Pipeline Tools & Frameworks* [11] oder *Top Continuous Integration Tools* [22] vorgestellt. Die Auswahl basiert dabei häufig auf willkürlichen oder nicht offengelegten Kriterien. Shahin et al. stellen fest, dass nur 25 von 69 untersuchten Literaturstudien erklären, wie die Tools für CI/CD-Lösungen ausgewählt werden. [17].

Um die CD-Pipelines umzusetzen, muss ein Team jedoch objektiv beurteilen, ob ein bestimmtes Job Execution System die Randbedingungen des Projekts erfüllt. Laut Duvall et al. sollte ein Job Execution System optimal zur Umgebung des Projekts passen und den Entwicklungsprozess langfristig unterstützen [5]. Um die objektive Auswahl eines optimalen Job Execution System zu unterstützen, wird in diesem Kapitel ein Kriterienkatalog als Basis für die Entscheidungsfindung entwickelt. Dessen insgesamt 14 Kategorien können jeweils im Bezug auf die Rahmenbedingungen eines konkreten Projekts angepasst werden.

A. Methodik zur Definition der Kriterien

Für die Erarbeitung des Kriterienkatalogs wurde zunächst relevante Literatur recherchiert, in der die objektive Auswahl von JES thematisiert wird. Dabei wurden folgende Literaturquellen identifiziert:

- Duvall et al. widmeten im Buch *Continuous Integration* ein Kapitel der Auswahl von CI/CD-Tools [5].
- Henschel untersucht in einer wissenschaftlichen Veröffentlichung die Unterschiede zwischen Job Execution Systems [8].
- Andreassen et al. beschreiben bei der Implementierung von CD-Pipelines ausführlich den Auswahlprozess der Tools [2].
- Ståhl und Bosch beschreiben in ihrer Arbeit über das Modellierungsframework Cinders relevante Eigenschaften von CI/CD-Tools [19].

Von den untersuchten Quellen begründen Duvall et al. die Auswahl ihrer vorgeschlagenen Kriterien sowie den Zusammenhang zwischen den Kriterien am ausführlichsten. Die Kri-

Tabelle I
KATEGORIEN ZUR AUSWAHL EINES JOB EXECUTION SYSTEMS

CLI Support	**Test Kompatibilität**	**Build Tool Integration**	**Feedback**	*Grundfunktionalitäten*
Bash, Powershell, ...	JUnit, Performance testing, Code analysis tools, ...	CMake, Maven, Gradle, ...	E-Mail, Slack, ...	

VCS	**Checkout-Strategie**	*Versionskontrolle*
Git, SVN, ...	Trunk only, Branch X, ...	

Pipeline Trigger	**Artefaktspeicherung**	**Build Isolation**	*Pipeline-Funktionalitäten*
Poll-, Schedule- oder Event-Driven	intern oder extern	Lokal, Container, VM, Kubernetes, ...	

Job Executor	**Job Scheduler**	**Pipeline Konfiguration**	*Verwaltungseigenschaften*
Self-/Cloud-hosted, OS	Self-/Cloud-hosted, OS	GUI, Pipeline-as-code, ...	

Extensibility	**Kosten**	*Sonstige Eigenschaften*
API, Plug-Ins, ...	Komplett kostenfrei, open-core, ...	

terien von Duvall et al. [5] bildeten deshalb die übergeordneten Kategorien zur Einordnung der Kriterien der anderen Quellen.

Alle Kriterien einer Kategorie wurden daraufhin zu einem neuen Überbegriff zusammengefasst. Durch diese Zusammenfassung erhielten die 14 Kategorien teilweise neue Bedeutung. Beispielsweise wird in der Auswahlkategorie *Program execution* von Duvall et al. festgelegt, welche Programme im JES ausgeführt werden sollen. [5] Andreassen et al. konkretisieren diese Kategorie. Sie empfehlen, sowohl die vom JES zu unterstützenden Skriptsprachen, als auch die auf dem Dateisystem des Job Executors durchzuführenden Operationen, festzulegen. [2] Die Gedanken beider Quellen wurden in der neuen Kategorie *CLI Support* kombiniert. Die Kommandozeile des Rechners, das *Command Line Interface*, kann sowohl Programme (Duvall et al.), als auch Skripte unterschiedlicher Programmiersprachen und Befehle zur Manipulation des Dateisystems (Andreassen et al.) ausführen.

Schließlich wurden die 14 zusammengefassten Kategorien in die fünf identifizierten Gruppen *Grundfunktionalitäten*, *Versionskontrolle*, *Pipeline-Funktionalitäten*, *Verwaltungseigenschaften* und *Sonstige Eigenschaften* unterteilt.

B. Kriterienkatalog

Der in Tabelle I dargestellte Katalog zur Auswahl eines Job Execution Systems besteht aus 14 allgemeinen Kategorien in fünf Überkategorien. Damit ein Job Execution System bestmöglich zu einem Projekt passt, müssen diese Kategorien anhand der technischen Randbedingungen des Projekts konkretisiert werden.

Grundfunktionalitäten

Zu den elementaren Funktionalitäten eines Job Execution Systems gehört das Ausführen von *CLI Kommandos*. Dies umfasst das Ausführen von Programmen und Skripten in diversen Sprachen [2, 5] sowie Operationen zur Dateimanipulation [5]. Weiterhin muss die Ausführung von Software-Tests durch eine ausreichende *Test Kompatibilität* ermöglicht werden. Dies umfasst auch andere Validierungsmethoden, wie Codeanalysen. [5] Um das Softwareprodukt herstellen zu können, muss durch eine *Build Tool Integration* der Quellcode kompiliert, gelinkt und in auslieferbare Komponenten verpackt werden. [5] Schließlich ist ein aussagekräftiges *Feedback* essentiell für das Team. Je nach Bedarf unterstützen Job Execution Systeme automatische Benachrichtigungen über aufgetretene Ereignisse, bis hin zur Generierung von Berichten über die Pipeline-Ergebnisse. Dazu werden Technologien wie E-Mail oder Instant-Messaging-Plattformen verwendet. [2, 8]

Versionskontrolle

Um in einer CD-Pipeline auf die integrierten Änderungen des Teams zugreifen zu können, muss das Job Execution System mit dem *Version Control System*, kurz *VCS*, des Projekts kompatibel sein [2, 5, 8]. Zudem muss durch eine *Checkout-Strategie* definiert werden, welche Dateien des Repositories auf Änderungen überprüft werden sollen [5, 8].

Pipeline-Funktionalitäten

Bei der Umsetzung von CD-Pipelines in einem Job Execution System muss festgelegt werden, welche Ereignisse eine Pipeline-Ausführung auslösen. Diese *Pipeline Trigger* lassen sich in drei Kategorien unterteilen. [5, 8]

- Eine *Poll-Driven* Pipeline wird sofort ausgelöst, nachdem Änderungen im Repository festgestellt wurden.
- Im Gegensatz dazu wird eine *Schedule-Driven* Pipeline nach einem expliziten Zeitplan ausgeführt.
- Eine *Event-Driven* Pipeline wird von internen Ereignissen anderer Pipelines ausgelöst.

Die Produkte einer Pipeline werden als Artefakte bezeichnet. Relevant für den Nutzer ist die *Artefaktspeicherung*. [5] Artefakte können intern im Job Execution System oder im Versionsverwaltungssystem gespeichert werden. Alternativ kann der Nutzer externe Infrastruktur zum Hochladen der Artefakte verwenden. Artefakte, wie das Softwareprodukt, müssen nicht auf dem Betriebssystem des Job Servers hergestellt werden,

sondern können in Containern oder in virtuelle Maschinen isoliert werden. Je nach Art der *Build Isolation* wird dadurch eine Reproduzierbarkeit und Skalierbarkeit der Builds ermöglicht. [8]

Verwaltungseigenschaften

Job Execution Systeme bieten unterschiedliche Möglichkeiten zur Umsetzung der CD-Pipelines an. Über beispielsweise eine grafische Nutzerschnittstelle kann die *Pipelinekonfiguration* visuell dargestellt werden. [19] Für komplexe Anforderungen können CD-Pipelines in Job Execution Systeme mit Skriptsprachen beschrieben werden und als Datei im Repository gespeichert werden. [8, 19] In Abbildung 7 ist der *YAML* Code zum Bauen und Testen eines Softwareprodukts in GitLab gezeigt.

```
stages:
   - build
   - test

build-code-job:
   stage: build
   script:
      - echo "Build a Ruby project."
      - make ...

test-code-job1:
   stage: test
   script:
      - echo "Test the built product."
      - junit ...
```

Abbildung 7. *Pipeline as Code* am Beispiel YAML nach der GitLab Syntax

Job Execution Systeme ermöglichen verteilte Architekturen. Um Pipelines schnell und isoliert auszuführen, bietet sich eine self-hosted Installation der *Job Executors* direkt auf der im Projekt zur Verfügung stehenden Hardware an. Der *Job Scheduler* kann beispielsweise die Jobs hingegen als Cloud-Anwendung koordinieren. Um die Kompatibilität mit der Projektumgebung zu gewährleisten, müssen die Instanzen das entsprechende Betriebssystem unterstützen.

Sonstige Eigenschaften

Die Funktionalität von Job Execution Systemen kann mit Plug-Ins und APIs erweitert werden. Neben dieser *Erweiterbarkeit* unterscheiden sich die Systeme bezüglich der anfallenden *Kosten*. Je nach Kostenmodell ist das Job Execution System komplett oder nur teilweise kostenfrei nutzbar. [2, 5]

C. Benutzung des Kriterienkatalogs

Um ein Job Execution System für ein Projekt auszuwählen, müssen diese Kategorien in Auswahlkriterien umgewandelt werden. Dazu muss basierend auf den Rahmenbedingungen des Projekts für jede Kategorie ein konkreter Wert definiert werden. Die Erfüllung eines konkreten Kriteriums sollte dabei mit *ja* oder *nein* beantwortet werden können. Sollte eine Kategorie das Projekt nicht betreffen, wird diese ausgelassen. Weiterhin können *kritische Kriterien* festgelegt werden. Sollte ein Job Execution System ein kritisches Kriterium nicht erfüllen, scheidet das System aus.

Die Kriterien werden anschließend als Spalten in eine Tabelle geschrieben. Jedes auszuwählende Job Execution System entspricht einer Zeile. Wenn ein System ein Kriterium erfüllt, wird die entsprechende Spalte mit einem Kreuz markiert. Zum Schluss wird für jede Zeile die Summe aller Kreuze gebildet. Nach Formel 1 erfüllt das Job Execution System mit der höchsten Summe die meisten Kriterien.

$$max \sum_{c \in C} \begin{cases} 1 & \text{if criterion met} \\ 0 & \text{otherwise} \end{cases} \quad C := \text{criterions} \qquad (1)$$

In Tabelle II werden die Job Execution Systeme *Jenkins*, *TeamCity*, *CircleCI* und *GitLab* verglichen. Nach den Rahmenbedingungen des Projekts muss das Job Execution System das Versionsverwaltungsprogramm SVN unterstützen, die CD-Pipelines müssen sofort nach neuen Integrationen ausgeführt werden und das Team muss per E-Mail benachrichtigt werden. Gemäß Formel 1 erfüllen Jenkins und TeamCity mit $\sum_{c \in C} = 3$ alle Kriterien.

Tabelle II
VERGLEICH VON VIER JOB EXECUTION SYSTEMS MITHILFE DES KRITERIENKATALOGS.

Job Execution System	**VCS**	**Pipeline Trigger**	**Feedback**	$\sum$
	SVN	Poll-Driven	E-Mail	
Jenkins	x	x	x	3
TeamCity	x	x	x	3
CircleCI		x	x	2
GitLab				0

V. FAZIT

Ziel dieser Arbeit war es, eine Literaturübersicht zur systematischen Implementierung der technischen Komponenten der Continuous Practices zu erstellen.

Im ersten Teil der Arbeit wurden existierende Methoden und Ansätze zur systematischen Planung von CD-Pipelines für unterschiedliche Projektkomplexitäten identifiziert. Diese Planung findet vor der Entwicklung der eigentlichen Pipeline-Bestandteile statt. Dabei wurde festgestellt, dass es bei der Entwicklung von Pipelines in einfachen Projekten mit begrenztem Umfang ausreichend ist, sich an in der Praxis gesammelten Erfahrungswerten zu orientieren. Beispielsweise im Falle von privaten Projekten oder Forschungsprojekten. [5, 7] Zur visuellen Modellierung von Pipelines komplexerer Projekte wurde UML als geeignetes Werkzeug identifiziert. Dabei wird typischerweise die standardisierte UML-Notation an die Rahmenbedingungen des Projekts angepasst. [10, 18] Alternativ dazu bietet das Continous-Delivery-Modellierungsframework Cinders die Möglichkeit zur ausführlichen Planung der Pipeline-Schritte. Verglichen mit UML findet dieses Framework in der Praxis wenig Verwendung. [19]

Im zweiten Teil dieser Arbeit wurde die systematische Auswahl eines Job Execution Systems zur Umsetzung der CD-Pipelines in der Literatur thematisiert. Dazu wurde im Rahmen einer Literaturrecherche eine Auswahlhilfe bestehend aus 14

Auswahlkriterien, gegliedert in fünf Gruppen erstellt. Basierend auf den Anforderungen des Projekts kann damit objektiv und systematisch ein Job Execution System zur Umsetzung der CD-Pipelines ausgewählt werden.

Mithilfe der systematischen Planung von CD-Pipelines und der objektiven Auswahl eines Job Execution Systems können die technischen Aspekte der Continuous Practices systematisch in einem Projekt implementiert werden — eine Notwendigkeit, die Ståhl und Bosch als bedeutende Kostenersparnis in Softwareprojekten identifizierten. [19]

LITERATUR

[1] Evio Abazi. "Practicing Continuous Integration in a Multi-Supplier Environment for the Development of Automotive Software". Magisterarb. 2019.

[2] O.O. Andreassen und A. Tarasenko. "Continuous Integration Using LabVIEW, SVN and Hudson". In: *International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2013)*. 2014, S. 74–76.

[3] Jan Bosch u. a. *Accelerating digital transformation: 10 years of Software Center*. Cham: Springer, 2022. ISBN: 978-3-031-10872-3.

[4] David Budgen u. a. "Empirical evidence about the UML: a systematic literature review". In: *Software: Practice and Experience* 41.4 (2011), S. 363–392.

[5] Paul M. Duvall, Steve Matyas und Andrew Glover. *Continuous integration: Improving software quality and reducing risk*. 8. print. A Martin Fowler signature book. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 978-0-321-33638-5.

[6] Ívar Gautsson und Thórhildur Hafsteinsdóttir. "Continuous Integration in Component-Based Embedded Software Development: Problems and Causes". Magisterarb. 2017.

[7] Viral Gupta, Parmod Kumar Kapur und Deepak Kumar. "Modeling and measuring attributes influencing DevOps implementation in an enterprise using structural equation modeling". In: *Information and software technology* 92 (2017), S. 75–91.

[8] Jack Henschel. "A comparison study of managed CI/CD solutions". In: (2020).

[9] Alena Hramyka und Martin Winqvist. *Traceability in continuous integration pipelines using the Eiffel protocol*. 2019.

[10] Jez Humble und David Farley. *Continuous delivery: [reliable software releases through build, test, and deployment automation]*. A Martin Fowler signature book. Upper Saddle River, NJ: Addison-Wesley, 2011. ISBN: 978-0-321-60191-9.

[11] Inc. Katalon. *Best 14 CI/CD Tools You Must Know: Updated for 2022*. © 2022. URL: https://katalon.com/resources-center/blog/ci-cd-tools (besucht am 13. 12. 2022).

[12] Kirill Shirinkin. *Job Execution Systems: What is the difference between Jenkins, Rundeck, Airflow, Gitlab CI and others*. Hrsg. von mkdev. München, © 2022. URL: https://mkdev.me/posts/job-execution-systems-what-is-the-difference-between-jenkins-rundeck-airflow-gitlab-ci-and-others (besucht am 13. 12. 2022).

[13] Eero Laukkanen, Juha Itkonen und Casper Lassenius. "Problems, causes and solutions when adopting continuous delivery—A systematic literature review". In: *Information and Software Technology* 82 (2017), S. 55–79.

[14] Francisco J Lucas, Fernando Molina und Ambrosio Toval. "A systematic review of UML model consistency management". In: *Information and Software technology* 51.12 (2009), S. 1631–1645.

[15] Mathias Meyer. "Continuous integration and its tools". In: *IEEE software* 31.3 (2014), S. 14–16.

[16] Agneta Nilsson, Jan Bosch und Christian Berger. "Visualizing testing activities to support continuous integration: A multiple case study". In: *International Conference on Agile Software Development*. Springer. 2014, S. 171–186.

[17] Mojtaba Shahin, Muhammad Ali Babar und Liming Zhu. "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices". In: *IEEE Access* 5 (2017), S. 3909–3943.

[18] Ian Sommerville. *Software engineering*. 6. Aufl., [2. Nachdr.] Informatik. München: Pearson Studium, 2003. ISBN: 9783827370013.

[19] Daniel Ståhl und Jan Bosch. "Cinders: The continuous integration and delivery architecture framework". In: *Information and Software Technology* 83 (2017), S. 76–93. ISSN: 0950-5849.

[20] Daniel Ståhl und Jan Bosch. "Cinders: The Continuous Integration and Delivery Architecture Framework: Extended Abstract". In: *Proceedings of the 2018 International Conference on Software and System Process*. ICSSP '18. New York, NY, USA: Association for Computing Machinery, 2018, S. 128–129. ISBN: 9781450364591.

[21] Daniel Ståhl, Torvald Martensson und Jan Bosch. "Continuous practices and devops: beyond the buzz, what does it all mean?" In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2017, S. 440–448.

[22] Victoria Bezsmolna. *Top 7 Continuous Integration Tools for DevOps*. 2019. URL: https://smartbear.com/blog/top-continuous-

integration-tools-for-devops/ (besucht am 13. 12. 2022).