

## Continuous Integration/Continuous Deployment in cloud-basierten Umgebungen: Architekturprinzipien und Best Practices

Dr. Marcus Zinn

### Berufliche Tätigkeiten

- 5,5 Jahre als Leiter für mobile Anwendungen im IIoT (Industrial Internet of Things)
- 5 Jahre Leiter für Patente (Schwerpunkt KI und Softwarepatente)
- 12 Jahre Software-/Systemarchitektur

### Hobbys

- Dozent seit 2004 (Mobile Apps, Informatik, Software Engineering, usw.)
- Betreuung von Bachelor- / Masterarbeiten: 28 (einschließlich 6 von der THAB)
- IIoT-Imkerei KI-basierte Bildgenerierung (Stabile Diffusion / ComfyUI)

# Inhalt der Vorlesung

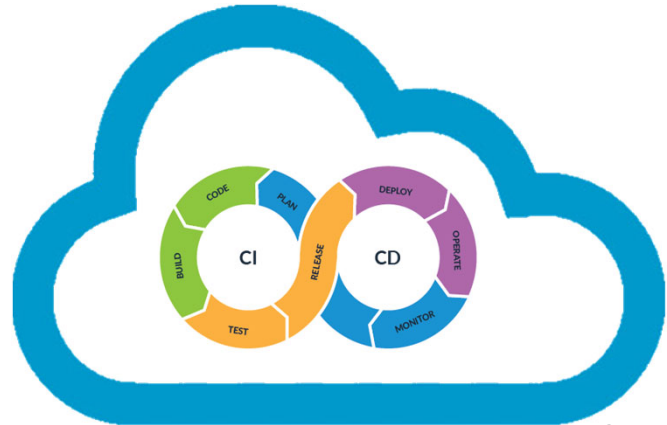
Teil 1 - Einführung

Teil 2 - Grundlegende Architekturprinzipien

Teil 3 - Architekturmusterbeispiele für CI/CD in der Cloud

Teil 4 - Best Practices für CI/CD in der Cloud

Teil 5 - Schlussfolgerung und Ausblick



Internal

2

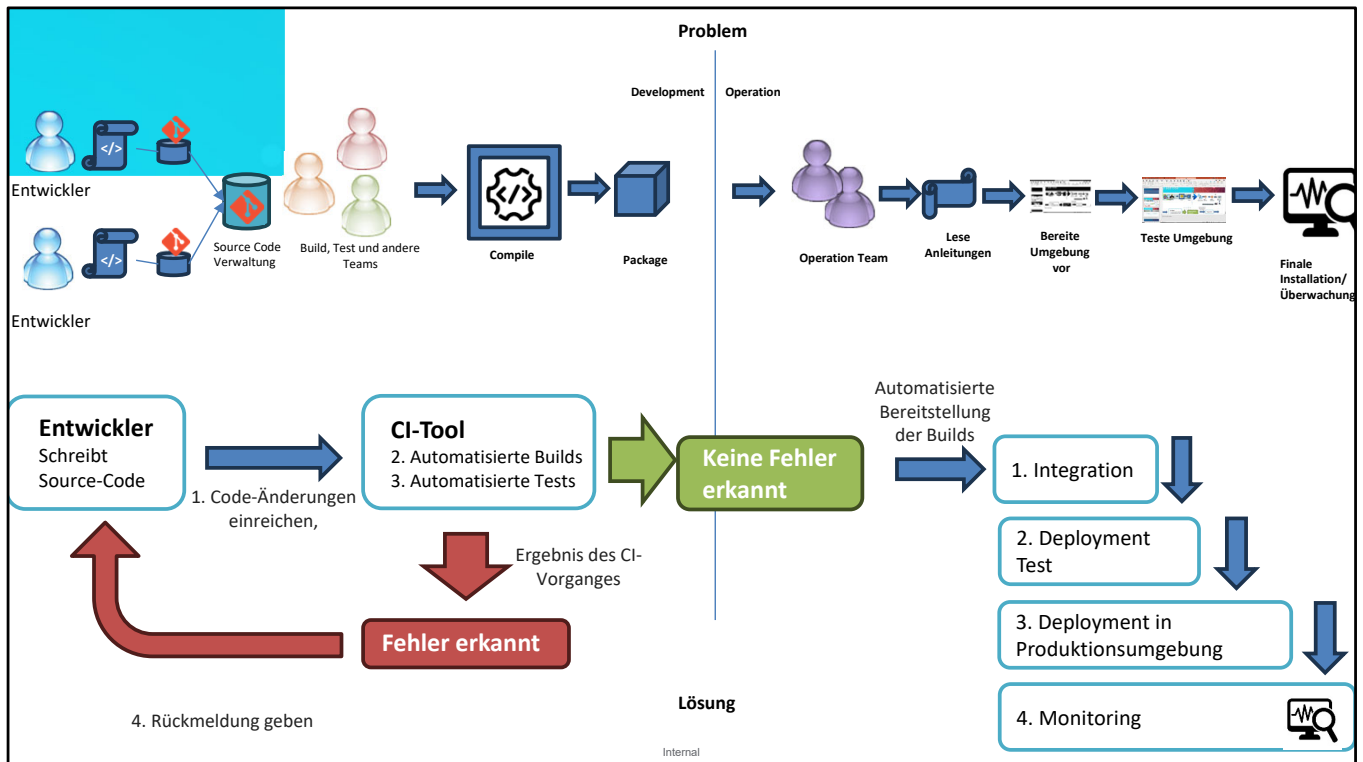
# Einleitung

Continues Integration / Continues Deployment

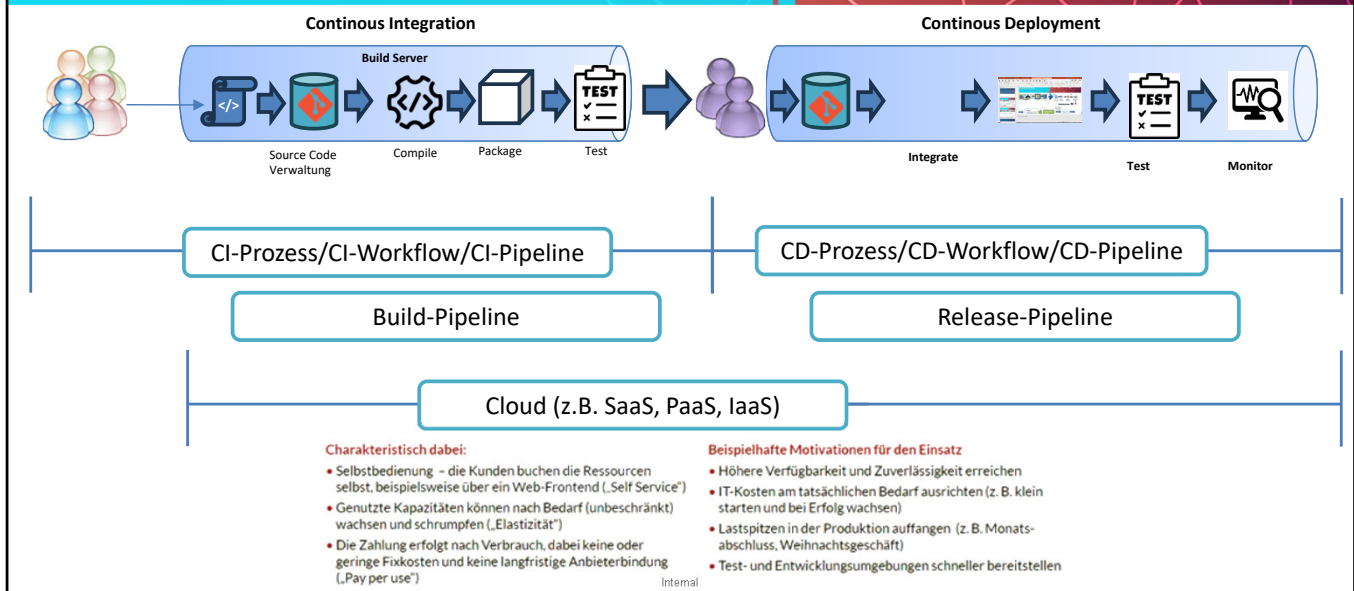
Dr. Marcus Zinn | 180 into 3 | Introduction

Internal

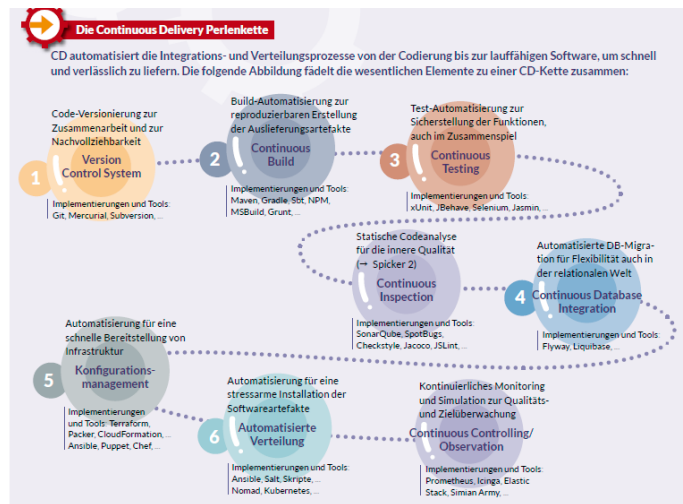
**Kapitel 1: Einprägsame Einführung (2 Minuten): Stelle dich kurz vor und erläutere die Herausforderung, 180 verschiedene Apps in 3 IIoT-Apps zu konsolidieren. Betone die Relevanz von Softwarearchitektur in diesem Kontext.**



# Grundlegende Konzepte erklären Pipeline und Cloud



# Pipeline-Beispiele



Continuous Delivery Perlenkette

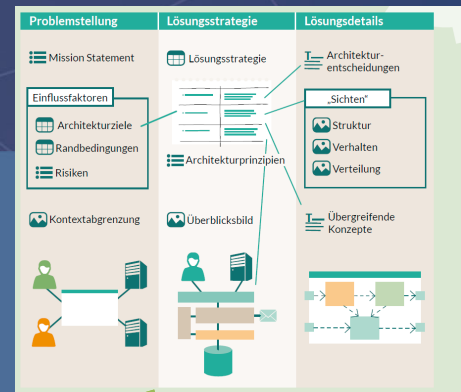
(Quelle: Embarc, 2024)




# Grundlegende Architekturprinzipien

Continues Integration / Continues Deployment

Modularität  
Kohäsion  
Skalierbarkeit  
Automatisierung

Architektur Vorgehensweisen  
(Quelle: Embarc-1, 2024)

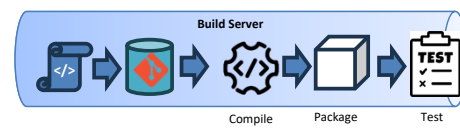
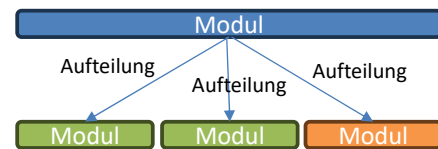


Zutat und typisches Format	Beschreibung
 Lösungsstrategie (Tabelle)	Zweispaltige Tabelle mit Architekturzielen und zugeordneten Architekturansätzen, mit Verweisen auf Überblicksbild und Lösungsdetails
 Architekturprinzipien	Grundsätze, an denen sich alle Entscheidungen orientieren (z.B. Präferenzen, „Bevorzuge XY vor Z“).
 Informelles Überblicksbild	Visualisierung der Lösung mit Betonung der zentralen Architekturansätze (z.B. Stil, Muster, Struktur ...) – eher kein UML

# Architekturprinzipien und ihrer Bedeutung für CI/CD in der Cloud

- **Modularität**

- Unterteilung in unabhängige Module oder Komponenten
- Schnellere Entwicklung, Tests und Bereitstellung
- Verbesserung der Agilität und Flexibilität in CI/CD-Umgebungen

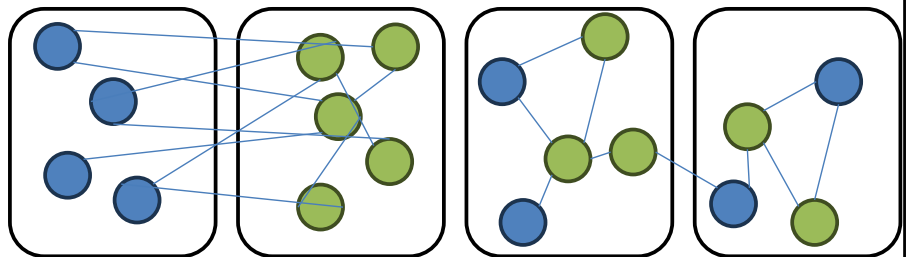
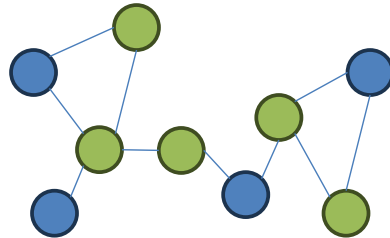




# Architekturprinzipien und ihrer Bedeutung für CI/CD in der Cloud

- **Kohäsion**

- Grad der Zusammengehörigkeit innerhalb eines Moduls oder einer Komponente
- Wichtig für Wartbarkeit und Testbarkeit in CI/CD-Umgebungen
- Gute kohäsive Module sind leichter zu verstehen, zu testen und zu ändern
- Führt zu schnelleren Iterationen im Entwicklungsprozess

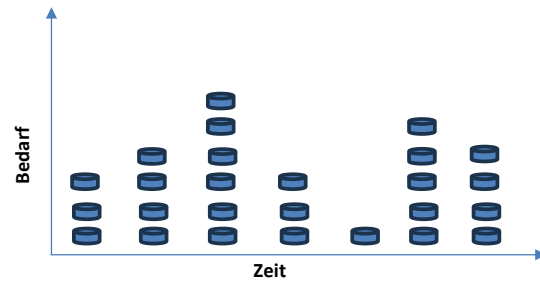


Internal

# Architekturprinzipien und ihrer Bedeutung für CI/CD in der Cloud

- **Skalierbarkeit**

- Reaktion auf steigende Anforderungen durch effiziente Ressourcennutzung
- Horizontale oder vertikale Skalierung in der Cloud-Architektur je nach Anwendungsanforderungen
- Vorteile für CI/CD durch schnelle Bereitstellung und Testumgebungen für neue Funktionen ohne Leistungseinbußen



# Architekturprinzipien und ihrer Bedeutung für CI/CD in der Cloud

- **Automatisierung**

- Entwicklerteams sparen Zeit und minimieren menschliche Fehler
- Gewährleistung einer konsistenten Bereitstellung
- Nahtlose Integration von Automatisierungstools und -services in der Cloud für den gesamten Entwicklungs- und Bereitstellungszyklus

<https://community.dataminer.services/ci-cd-and-the-agil>



Continuous Integration & Continuous Delivery pipeline

(Quelle: Dataminer, 2021)

## Relevanz für die Effizienz und Qualität von Softwareentwicklungsprozessen

- **Modularität**
  - Aufteilung in unabhängige Module
  - Effizienteres Arbeiten für Entwickler
  - Schnellere Entwicklung, Tests und Bereitstellung
  - Geringere Auswirkungen von Änderungen
  - Erleichterte Wartbarkeit und Skalierbarkeit
- **Kohäsion:**
  - Klare und spezifische Aufgaben in Modulen
  - Reduzierung unerwünschter Abhängigkeiten und Komplexität
  - Steigerung der Effizienz
  - Besser strukturierter Code und einfachere Testverfahren
  - Erhöhung der Qualität und Wartbarkeit der Software
- **Skalierbarkeit:**
  - Reaktion auf steigende Anforderungen
  - Schnelle Anpassung an Lastspitzen
  - Erhaltung von Leistung und Stabilität
  - Zuverlässige Funktion unter zunehmender Belastung
  - Aufrechterhaltung hoher Qualität
- **Automatisierung:**
  - Reduzierung manuellen Aufwands und menschlicher Fehler
  - Steigerung der Effizienz und konsistente Ergebnisse
  - Freisetzung von Zeit für Entwicklung neuer Funktionen und Qualitätsverbesserungen
  - Beschleunigung von Entwicklungsprozessen
  - Schnellere Markteinführung von Softwareprodukten



# Architekturmuster für CI/CD in der Clou

Continues Integration / Continues Deployment

Serverless-Architektur  
Mikroservice-Architektur  
Container-Orchestrierung

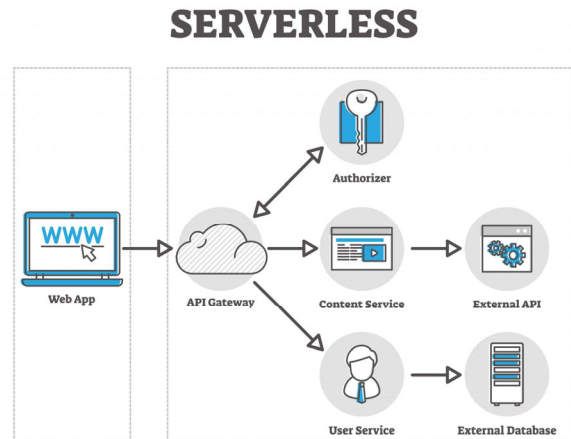
## Kurze Vorstellung der verschiedenen Architekturmuster wie Serverless, Mikroservices und Container-Orchestrierung

- **Serverless-Architektur:**

- Bereitstellung von Funktionen oder Mikroservices in unabhängigen Einheiten, ohne sich um die zugrunde liegende Infrastruktur kümmern zu müssen
- Granulare Skalierbarkeit und präzise Ressourcennutzung
- geeignet für Event-basierte oder skalierbare Anwendungen, bietet hohe Flexibilität

- **Vorteile für CI/CD:**

- Schnelle Bereitstellung (von Services)
- kann können leicht in CI/CD-Pipelines integriert werden, da sie aus kleinen, unabhängigen Funktionen bestehen.
- Automatisierte Build- und Deployment-Skripte können verwendet werden, um Funktionen bereitzustellen.
- CI/CD-Pipelines können automatische Tests für jede Funktion durchführen, um die Qualität zu gewährleisten.
- Kostenoptimierung (pay-as-you-go-Preismodell)
- Skalierbarkeit (granulare Skalierung auf Funktionsebene)



Serverless-Architektur (Quelle Kofi-group, 2024)

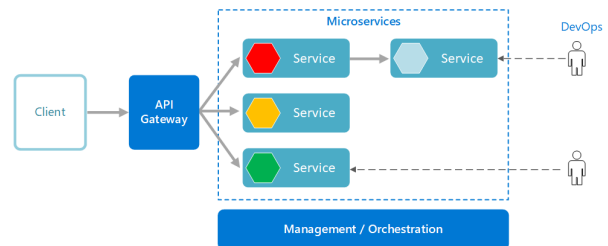
# Kurze Vorstellung der verschiedenen Architekturmuster wie Serverless, Mikroservices und Container-Orchestrierung

- **Mikroservice-Architektur:**

- Bei der Mikroservice-Architektur wird eine Anwendung in kleinere, eigenständige Dienste oder Mikroservices aufgeteilt, die jeweils eine spezifische Funktion erfüllen.
- Unabhängige Entwicklung und Bereitstellung: Jeder Mikroservice kann unabhängig entwickelt, getestet, deployt und skaliert werden, was die Wartbarkeit, Skalierbarkeit und Flexibilität der Anwendung verbessert.
- Standardisierte Kommunikation: Mikroservices kommunizieren über standardisierte Schnittstellen wie APIs und können in verschiedenen Programmiersprachen und Technologien implementiert werden.

- **Vorteile für CI/CD**

- **Unabhängige Bereitstellung**  
Durch die Aufteilung in Mikroservices können Updates oder neue Funktionen unabhängig voneinander bereitgestellt werden, ohne dass andere Teile des Systems beeinträchtigt werden.
- **Skalierbarkeit**  
Mikroservices können separat skaliert werden, um auf veränderte Anforderungen oder Lastspitzen zu reagieren, was eine effiziente Ressourcennutzung ermöglicht.
- **Technologische Vielfalt:**  
Jeder Mikroservice kann in einer anderen Technologie oder Programmiersprache implementiert werden, was Flexibilität und Innovation ermöglicht.



MicroService Architektur (Quelle: Microsoft, 2024)

<https://learn.microsoft.com/de-de/azure/architecture/guide/architecture-styles/microservices>

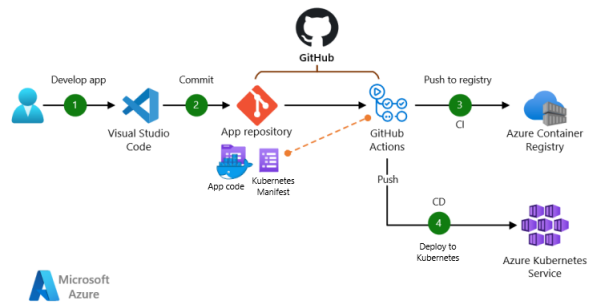
## Kurze Vorstellung der verschiedenen Architekturmuster wie Serverless, Mikroservices und Container-Orchestrierung

- **Container-Orchestrierung:**

- Container-Orchestrierungssysteme wie Kubernetes ermöglichen die Verwaltung und Orchestrierung von Containern in verteilten Umgebungen.
- Container bieten eine konsistente Bereitstellung von Anwendungen in verschiedenen Umgebungen und reduzieren die Abhängigkeit von der zugrunde liegenden Infrastruktur.
- Kubernetes ermöglicht die Automatisierung von Bereitstellung, Skalierung, Lastausgleich und Wiederherstellung von Containern, was die Effizienz und Zuverlässigkeit des Betriebs verbessert.

- **Vorteile für CI/CD:**

- Container-Orchestrierungssysteme wie Kubernetes erleichtern die Integration von Containern in CI/CD-Pipelines.
- Entwickler können automatisierte Skripte verwenden, um Containerimages zu erstellen, zu testen und in Kubernetes-Cluster bereitzustellen.
- CI/CD-Pipelines können auch automatische Rollbacks durchführen, wenn Tests fehlschlagen oder Probleme auftreten.
- Konsistente Bereitstellung
- Automatisierte Skalierung
- Hohe Verfügbarkeit



Push-basierte Architektur mit GitHub Actions für CI und CD.

(Quelle: Microsoft, 2024) <sup>16</sup>

Internal



## Architekturprinzipien und Architekturmuster

	Serverless	Microservice	Container Orchestrierung
Modularisierung	✓✓	✓✓	✓✓
Kohäsion		✓✓	✓✓✓
Skalierbarkeit	✓✓	✓✓	✓✓✓
Automatisierung	✓	✓✓✓	✓✓



# Best Practices für CI/CD in der Cloud

Continues Integration / Continues Deployment

# Bedeutung von Automatisierung, kontinuierlichem Feedback und Monitoring

## 1. Automatisierung:

1. grundlegendes Prinzip von CI/CD und ermöglicht wiederholbare, effiziente und konsistente Prozesse.
2. spart Entwicklern Zeit, minimiert menschliche Fehler und beschleunigt die Softwarebereitstellung.
3. Automatisierte Build-, Test- und Deployment-Pipelines gewährleisten Softwarequalität und Zuverlässigkeit durch Reduzierung manueller Aufwände und Fehler.
4. In cloud-basierten Umgebungen nahtlose Integration für den gesamten Entwicklungs- und Bereitstellungszyklus sowie effiziente Skalierung.

## 2. Kontinuierliches Feedback:

1. entscheidend, um die Qualität und Leistung der Software kontinuierlich zu verbessern.
2. Entwickler können schnell auf Probleme reagieren, Verbesserungsmöglichkeiten identifizieren und den Entwicklungsprozess optimieren.
3. Automatisierte Tests, Code-Reviews und Nutzerfeedback sind wichtige Quellen für kontinuierliches Feedback, die in CI/CD-Pipelines integriert werden können.
4. In cloud-basierten Umgebungen können Monitoring- und Logging-Tools verwendet werden, um Echtzeitdaten über die Leistung und Verfügbarkeit von Anwendungen zu sammeln und Entwicklerteams mit relevanten Informationen zu versorgen.

## 3. Monitoring:

1. kontinuierliche Überwachung von Leistung, Verfügbarkeit und Sicherheit von Anwendungen, um potenzielle Probleme frühzeitig zu erkennen.
2. Entwickler können schnell auf Leistungsprobleme, Ausfälle oder Sicherheitsvorfälle reagieren und Maßnahmen ergreifen, um die Qualität und Zuverlässigkeit der Software zu gewährleisten.
3. In cloud-basierten Umgebungen können Monitoring-Tools und -Dienste Metriken, Protokolle und Traces sammeln und analysieren, um Einblicke in die Leistung und das Verhalten von Anwendungen zu gewinnen.
4. Ermöglicht es Trends zu identifizieren, Kapazitätsplanung durchzuführen und die Skalierung von Ressourcen entsprechend anzupassen

# Bewährten Methoden für Build-, Test- und Deployment-Automatisierung.

## **Build-Automatisierung:**

1. Build-Management-Tools wie Apache Maven, Gradle oder npm, um den Build-Prozess zu automatisieren.
2. Konfiguration von Build-Skripten, die alle notwendigen Schritte zum Kompilieren von Quellcode, Erstellen von Artefakten und Verwalten von Abhängigkeiten enthalten.
3. Einrichtung von Trigger-Mechanismen, um automatisch Builds auszulösen, wenn Änderungen im Versionskontrollsystem vorgenommen werden.
4. Nutzung von Continuous Integration-Servern wie Jenkins, Travis CI oder CircleCI, um den Build-Prozess zu automatisieren und zu planen.

## **Testautomatisierung:**

1. Entwicklung von automatisierten Testsuites, die verschiedene Aspekte der Anwendung abdecken, einschließlich Einheitstests, Integrationstests und End-to-End-Tests.
2. Verwendung von Test-Frameworks und Bibliotheken wie JUnit, Selenium, pytest oder Mocha, um automatisierte Tests zu schreiben und auszuführen.
3. Integration von automatisierten Tests in den CI/CD-Pipeline, um sicherzustellen, dass jede Änderung am Code automatisch getestet wird, bevor sie in die Produktionsumgebung gelangt.
4. Implementierung von Regressionstests, um sicherzustellen, dass vorhandene Funktionalität nach Änderungen weiterhin ordnungsgemäß funktioniert.

## **Deployment-Automatisierung:**

1. Verwendung von Konfigurationsmanagement-Tools, um die Bereitstellung von Anwendungen und Infrastruktur zu automatisieren (Ansible, Chef oder Puppet).
2. Erstellung von Skripten oder Playbooks, die die Installation, Konfiguration und Aktualisierung von Anwendungen und Diensten automatisieren.
3. Implementierung von Continuous Deployment-Pipelines, um Änderungen automatisch in Produktionsumgebungen zu deployen, nachdem sie erfolgreich getestet wurden.
4. Verwendung von Deployment-Strategien, um die Auswirkungen von Änderungen auf die Benutzer zu minimieren und die Verfügbarkeit der Anwendung zu gewährleisten (Canary-Deployments oder Blue-Green-Deployments).



# Schlussfolgerung und Ausblick

Continues Integration / Continues Deployment

# Zusammenfassung der wichtigsten Punkte aus der Vorlesung

## **Architekturprinzipien**

Modularität, Kohäsion, Skalierbarkeit und Automatisierung sind entscheidende Architekturprinzipien, die die Effizienz und Qualität von CI/CD-Prozessen in der Cloud verbessern.

## **Architekturmuster**

Serverless, Mikroservices und Container-Orchestrierung sind wichtige Architekturmuster, die CI/CD in cloud-basierten Umgebungen unterstützen und verschiedene Vorteile bieten, darunter Flexibilität, Skalierbarkeit und Konsistenz.

## **Bedeutung von Automatisierung**

Automatisierung von Build-, Test- und Deployment-Prozessen ermöglicht eine schnellere Bereitstellung von Software, eine höhere Produktivität der Entwicklerteams und eine verbesserte Softwarequalität.

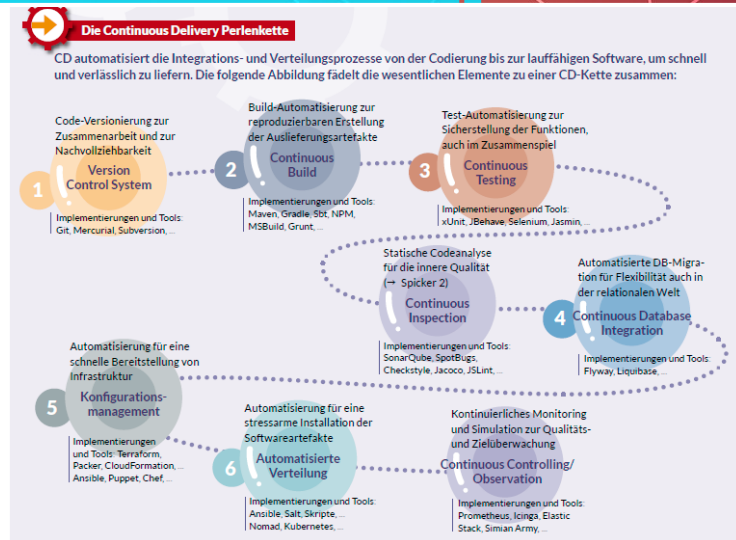
## **Kontinuierliches Feedback und Monitoring**

Kontinuierliches Feedback und Monitoring sind entscheidend, um die Qualität und Leistung von Anwendungen kontinuierlich zu verbessern, Probleme frühzeitig zu erkennen und die Benutzererfahrung zu optimieren.

## **Praktische Anwendbarkeit und Auswirkungen von CI/CD**

CI/CD beschleunigt die Bereitstellungszyklen, verbessert die Softwarequalität, erhöht die Entwicklerproduktivität, fördert eine DevOps-Kultur und bietet Skalierbarkeit und Flexibilität für Unternehmen.

# Ausblick auf zukünftige Entwicklungen und Herausforderungen im Bereich CI/CD in der Cloud.



Continuous Delivery Perlenkette

Internal

Quelle: Embarc, 2024

23

## Ausblick auf zukünftige Entwicklungen und Herausforderungen im Bereich CI/CD in der Cloud.

### Weiterentwicklung von Cloud-Technologien

Cloud-Plattformen werden sich weiterentwickeln, um mehr Funktionen und Dienste anzubieten, die die Automatisierung, Skalierbarkeit und Sicherheit von CI/CD-Prozessen verbessern.

### Integration von KI und ML

Künstliche Intelligenz (KI) und maschinelles Lernen (ML) werden vermehrt in CI/CD-Pipelines integriert, um automatische Fehlererkennung, Optimierung von Bereitstellungen und Vorhersage von Leistungsproblemen zu ermöglichen.

### Micro-Frontends

Ähnlich wie bei Mikroservices werden Micro-Frontends aufkommen, um Frontend-Anwendungen in kleine, unabhängige Komponenten aufzuteilen, was zu flexibleren Bereitstellungs- und Aktualisierungsprozessen führt.

### Infrastruktur als Code (IaC)

IaC wird bereits in vielen CI/CD-Pipelines eingesetzt, aber dieser Trend wird sich voraussichtlich weiter verstärken. Die Verwendung von Tools wie Terraform oder Ansible zur Automatisierung der Infrastrukturkonfiguration ermöglicht eine konsistente Bereitstellung von Infrastrukturrressourcen in verschiedenen Umgebungen.



Continues Integration & Continues Deployment in  
Cloudumgebungen

Folien verfügbar unter



Vielen Dank für Ihre Aufmerksamkeit

Kontakt Daten  
Dr Marcus Zinn  
mail@marcuszinn.de  
[www.linkedin.com/in/marcuszinn](https://www.linkedin.com/in/marcuszinn)

## Quellenverzeichnis

Quellenreferenz	Titel	Link
Embarc, 2024	Continuous Delivery	<a href="https://www.embarc.de/architektur-spicker/07-continuous-delivery/">https://www.embarc.de/architektur-spicker/07-continuous-delivery/</a>
Embarc-1, 2024	Der Architekturüberblick	<a href="https://www.embarc.de/architektur-spicker/01-der-architekturueberblick/">https://www.embarc.de/architektur-spicker/01-der-architekturueberblick/</a>
Dataminer, 2021	Continuous Integration & Continuous Delivery pipeline	<a href="https://community.dataminer.services/ci-cd-and-the-agile-principles/">https://community.dataminer.services/ci-cd-and-the-agile-principles/</a>
Kofi-group, 2024	Serverless-Architektur	<a href="https://www.kofi-group.com/serverless-architecture-explained-in-10-minutes/">https://www.kofi-group.com/serverless-architecture-explained-in-10-minutes/</a>
Microsoft, 2024	MicroService Architektur	<a href="https://learn.microsoft.com/de-de/azure/architecture/guide/architecture-styles/microservices">https://learn.microsoft.com/de-de/azure/architecture/guide/architecture-styles/microservices</a>
Microsoft, 2024	<i>Push-basierte Architektur mit GitHub Actions für CI und CD.</i>	<a href="https://learn.microsoft.com/de-de/azure/architecture/guide/aks/aks-cicd-github-actions-and-gitops">https://learn.microsoft.com/de-de/azure/architecture/guide/aks/aks-cicd-github-actions-and-gitops</a>

# Live Showcase „GitHub Actions“

The screenshot displays a GitHub repository interface for a project named "Test2 project". The left pane shows the workflow file `Test2 project` with the following content:

```
1 name: Test2 project      # Name of the job
2 on: workflow_dispatch    # User manual starts the workflow
3 jobs:                    # Define jobs
4   test-job:              # Name of the first job
5     runs-on: windows-latest # Define OS to run
6     steps:                # Steps to run for a job
7       - name: Get Code    # Name of the step
8         uses: actions/checkout@v4
9       - name: Install NodeJS
10        uses: actions/setup-node@v4
11        with:
12          node-version: 18
13       - name: Install dependencies
14         run: npm install
15       - name: Run tests
16         run: npm test
```

The right pane shows the "Test2 project #4" workflow run. It includes a "Summary" section with links to "Jobs", "Run details", "Usage", and "Workflow file". The "Jobs" section lists the steps of the workflow:

- Set up job (1s)
- Get Code (11s)
- Install NodeJS (27s)
- Install dependencies (6s)
- Run tests (0s)
- Post Install NodeJS (0s)
- Post Get Code (1s)
- Complete job (0s)

Internal

27