

Ausarbeitung

Software-Architektur

an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Kommunikationsinformatik
der Fakultät für Ingenieurwissenschaften

Continuous Integration / Continuous Delivery and Deployment

vorgelegt von

Sayed Mustafa Sajadi

Mhd Yaman Aljazairi

Anas Alajaji

betreut und begutachtet von

Prof. Dr. Markus Esch

Saarbrücken, Tag. Monat Jahr

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“ zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, Tag. Monat Jahr

Sayed Mustafa Sajadi
Mhd Yaman Aljazairi
Anas Alajaji

Zusammenfassung

Kurze Zusammenfassung des Inhaltes in deutscher Sprache, der Umfang beträgt zwischen einer halben und einer ganzen DIN A4-Seite.

Orientieren Sie sich bei der Aufteilung bzw. dem Inhalt Ihrer Zusammenfassung an Kent Becks Artikel: <http://plg.uwaterloo.ca/~migod/research/beck00PSLA.html>.

Inhaltsverzeichnis

1	Einleitung	1
1.0.1	Motivation	1
1.0.2	Aufgabenstellung und Zielsetzung	1
2	Grundlagen	3
3	Release Management Methoden	5
4	Continuous Integration und Continuous Delivery	7
4.1	Continuous Integration	7
4.2	Continuous Delivery und Deployment	9
4.3	CI/CD Tools	10
4.3.1	Concourse CI	10
4.3.2	GitLab	13
4.3.3	CircleCI	16
4.3.4	Jenkins	17
5	Fallstudie	19
5.1	Anforderungen	19
5.1.1	Funktionale Anforderungen	19
5.1.2	Nicht-Funktionale Anforderungen	19
5.2	Konzept	21
5.2.1	Geeignete Software-Update-Strategie der Ampelanlage	21
5.2.2	Software Update-Architektur der Ampelanlage	21
5.2.3	Auswahl der geeigneten Release-Managementmethode	22
5.2.4	Auswahl der geeigneten Release-Strategie	23

5.3	Entwicklungsvorgang	23
5.4	Implementierung	24
5.4.1	Aufbau der experimentellen Umgebung	24
5.4.2	Implementierung des Release-Management-Prozesses	24
6	Evaluation	27
7	Zusammenfassung	29
	Literatur	31
	Abbildungsverzeichnis	35
	Tabellenverzeichnis	35
	Listings	35
	Abkürzungsverzeichnis	37

1 Einleitung

1.0.1 Motivation

Die Notwendigkeit von Software-Updates ist sehr wichtig, um die korrekte Funktion der eingebetteten Systeme fehlerfrei und problemlos zu gewährleisten, denn die Software-Bugs hatten katastrophale Folgen, insbesondere in den Bereichen Luft- und Raumfahrt, Automotive und Medizin. Zwischen Juni 1985 und Januar 1987 verabreichte ein computergesteuertes Therapiegerät namens "Therach-25" sechs Personen eine große Anzahl von Überdosierungen, die zu schweren Verletzungen und zum Tod führten. Den Software-Entwicklern sollten Funktionen zur Verfügung gestellt werden, die den Software-Entwicklungsprozess beschleunigen und Software-Updates über das Internet ermöglichen. Denn ein manuelles Update, zum Beispiel über USB-Sticks, zum einen verlangsamt der Übertragungsprozess und zum anderen kann während der Übertragung zu menschlichen Fehlern führen. Darüber hinaus ist dieser Prozess anfällig, wie Fiat Chrysler 2015 damit begonnen hat, einen Hotfix für Millionen von Fahrzeugen per Post über einen USB-Stick zu verteilen, der es Hackern ermöglicht, Briefe und USB-Sticks zu fälschen.

1.0.2 Aufgabenstellung und Zielsetzung

In dieser Arbeit soll die verschiedenen Release-Management-Methoden erklärt und insbesondere die Methode Continuous Integration, Delivery and Deployment (CICD) herausgearbeitet und die verschiedenen dabei eingesetzten Tools sowie deren Unterschiede erläutert werden. Als nächstes wird auf die Frage eingegangen, welche CI/CD-Auswirkungen die Softwarearchitektur und die Entwicklungsprozesse von haben. Darüber hinaus wird die Anwendung dieser Methode anhand eines Fallbei-

spiels näher untersucht, in dem der Fall der Anwendung eines Software-Updates eines Endgeräts über das Internet tatsächlich betrachtet und unter Verwendung einer Container-Technologie namens Docker umgesetzt wird.

2 Grundlagen

3 Release Management Methoden

4 Continuous Integration und Continuous Delivery

Das Ausführen sich wiederholender Aufgaben führt oft zu Langeweile, was zu Fehlern und Fehlern führt. Ein erfolgreiches Entwicklungsteam vermeidet es, sich wiederholende Aufgaben zu erledigen, um seine Zeit und Kreativität in wichtigere Aufgaben zu investieren. Die Automatisierung ist eine der Lösungen, mit denen dieses Ziel erreicht werden kann. Automatisierung im IT-Bereich zielt darauf ab, Dienste und Anwendungen mit möglichst wenig menschlichem Eingreifen zu produzieren und bereitzustellen. Durch die Implementierung von Automatisierung können die Zuverlässigkeit, Effizienz und Geschwindigkeit der Verarbeitung verschiedener Aufgaben im Vergleich zur Ausführung durch den Menschen erheblich verbessert werden. Nach der Verbreitung automatisierter Prozesse und Methoden hat es begonnen, Softwareentwicklungsmethoden wie *Continuous Integration und Continuous Delivery/Deployment (CI/CD)* zu unterstützen.

4.1 Continuous Integration

Zu Beginn müssen einige von J. Humble und D. Farley [20] genannte Begriffe geklärt werden, um dieses Kapitel zu verstehen:

- Der **Build** ist eine Reihe von Vorgängen, die zum Erzeugen, Testen und Bereitstellen von Software ausgeführt werden.
- **Kontinuierlich (eng. Continuous)** ist ein Prozess, der niemals aufhört, sobald er einmal begonnen hat. Mit anderen Worten, es handelt sich um einen Prozess,

der kontinuierlich abläuft und einen neuen Build durchführt, wenn Änderungen im Versionskontroll-Repository entdeckt werden, z. B. auf GitHub ¹ oder GitLab ².

- **Integration** ist in der Sprache der Prozess der Eingliederung, der als Eingliederung in eine größere Einheit definiert ist. In der Welt der Informatik ist Integration die Aktion, die durchgeführt wird, um separate Quellcodes zu kombinieren, um festzustellen, wie sie als Ganzes funktionieren.

Continuous Integration (Kontinuierliche Integration)(CI) ist eine Reihe von Regeln und Praktiken in der Softwareentwicklung, an denen Entwicklungsteams beteiligt sind [22]. Jedes Teammitglied übernimmt die Verantwortung für die tägliche Integration und Zusammenführung des entwickelten Codes. Unter dem Begriff der Continuous Integration werden viele Perspektiven und Vorteile gesehen. Durch den Einsatz dieser Methode wurde festgestellt, dass sie dazu beiträgt, die Probleme bei der Integration erheblich zu reduzieren, was es einem Team ermöglicht, Software harmonisch und schneller zu entwickeln [14]. Der Grund dafür ist laut M. Shahin und M. Barbar, dass CI Softwareunternehmen aller Größenordnungen kürzere und häufigere Release-Zyklen ermöglicht, die Produktivität ihrer Teams erhöht und die Qualität ihrer Software verbessert. Die Automatisierung der Softwareerstellung und der Unit-/Integrationstests ist in dieser Praxis enthalten [23]. Ein erfolgreicher CI-Prozess bedeutet daher, dass neue Änderungen am Code erstellt, getestet und konsistent in ein gemeinsames Repository in einem Versionskontrollsystem wie GitHub oder GitLab zusammengeführt werden [24]. Kurz gefasst, kontinuierliche Integration legt großen Wert auf automatisierte Tests, um sicherzustellen, dass die Anwendung nicht abbricht, wenn neue Übertragungen in den main-Branch integriert werden [1].

¹<https://github.com/>

²<https://about.gitlab.com/>

4.2 Continuous Delivery und Deployment

Continuous Delivery (Kontinuierliche Bereitstellung)(CD) ist eine Erweiterung der Continuous Integration, da alle Codeänderungen nach der Build- und Test-Phase in Produktionsumgebungen bereitgestellt werden. Das heißt, dass zusätzlich zum automatisierten Testen einen automatisierten Freigabeprozess (Release) gibt, die jederzeit per Mausklick bereitgestellt werden kann. Theoretisch hat der Entwickler die Entscheidung, wann einen neuen Release bereitgestellt werden kann bzw. täglich, wöchentlich, oder monatlich. Wenn der Entwickler jedoch wirklich von den Vorteilen der Continuous Delivery profitieren will, Atlassian empfiehlt, die Delivery für die Produktion so früh wie möglich vorzunehmen, um sicherzustellen, dass kleine Chargen freigegeben wird, die im Falle eines Problems leicht zu beheben sind [1]. Die Continuous Delivery prüft den Code automatisch, erfordert aber einen menschlichen Eingriff, um die Delivery der Änderungen manuell auszulösen [16].

Continuous Deployment (Kontinuierliche Verteilung)(CD) ist ein weiterer Schritt über Continuous Integration hinaus, der ähnlich wie Continuous Delivery ist. Der Unterschied besteht darin, dass die Anwendungen automatisch bereitstellen können, anstatt Anwendungen manuell bereitzustellen. Bei diesem Verfahren wird jede Änderung, die alle Phasen der Produktionspipeline durchläuft, automatisch für die Kunden freigegeben. Es gibt keine menschlichen Eingriffe, und nur ein fehlgeschlagener Test verhindert, dass eine neue Änderung in der Produktion bereitgestellt wird [1]. Nach Bestehen der automatisierten Tests können Änderungen an einer Anwendung innerhalb von Minuten nach deren Erstellung in Betrieb genommen werden [16]. Der größte Vorteil der Continuous Deployment ist, dass die Kunden die Verbesserungen direkt sehen können, in der die Qualität jeden Tag steigt, anstatt jeden Monat oder jedes Jahr [1]. Die Abbildung 4.1 hebt die Zusammenhang von CI/CD hervor.

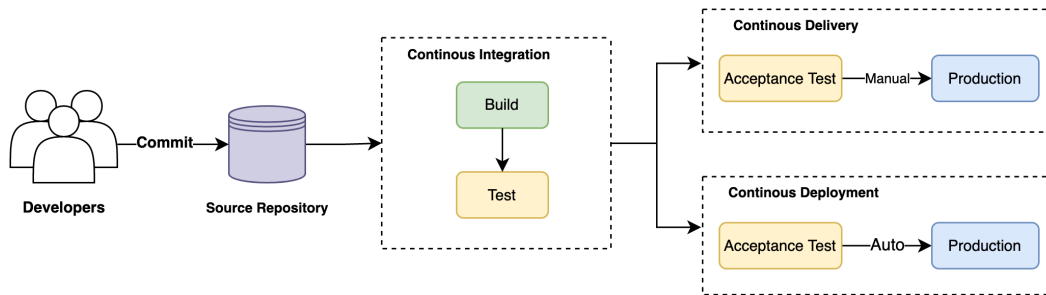


Abbildung 4.1: Die Zusammenhang zwischen CI/CD [23]

4.3 CI/CD Tools

CI/CD-Tools können einem Team helfen, seine Entwicklung, Bereitstellung und Tests zu automatisieren. Einige Tools kümmern sich speziell um die Integration (CI), andere um die Entwicklung und Bereitstellung (CD), wieder andere sind auf kontinuierliche Tests oder verwandte Funktionen spezialisiert.

4.3.1 Concourse CI

Concourse CI ist ein Open-Source-Tool zur kontinuierlichen Integration, das für die Bewältigung sich wiederholender Aufgaben entwickelt wurde. Um diese Aufgaben zu erfüllen, muss eine Automatisierungspipeline mit Hilfe einer YAML-formatierten Konfigurationsdatei erstellt werden. Mit Hilfe von Concourse CI können Aufgaben, Ressourcen und Jobs, die sich von Zeit zu Zeit wiederholen, automatisch definiert und ausgeführt werden [6].

- **Pipelines:** Entwickler müssen Pipelines erstellen, um bestimmte Aufgaben zu automatisieren. Dazu werden die Ressourcen und Jobs deklariert, die zur Erfüllung dieser Aufgaben erforderlich sind. Nach der Konfiguration können diese Pipelines feststellen, ob eine neue Version der definierten Resource veröffentlicht wurde. Wenn dies der Fall ist, stellt Concourse automatisch neue Builds für diese neu eingegebenen Jobs in die Warteschlange [7]. Ein gutes Beispiel für die Erkennung neuer Ressourcen-Aktualisierungen ist, wenn ein Entwickler neue Änderungen an einem Git Branch vornimmt, der in einem

GitHub-Repository definiert ist. Solche Operationen veranlassen die Pipeline, einen neuen Build des Jobs zu erstellen.

- **Ressource (engl. Resource):** Ressourcen in Concourse sind Objekte, die in die Pipeline gezogen oder aus ihr herausgeschoben werden müssen. Nachdem diese Ressourcen richtig konfiguriert sind, überprüft Concourse sie regelmäßig auf neue Versionen und löst einen neuen Build aus, wenn ein Update gefunden wird [8]. Eine der wichtigsten definierten Concourse Ressourcen ist git, die die Commits in einem Git-repository verfolgt [9]. Eine weitere wichtige Ressource time, die steuert, wann die Pipeline ausgelöst wird, indem Zeitintervalle wie jede Stunde oder jede Mitternacht definiert werden [13].
- **Ressourcentypen (engl. Resource Types):** Concourse wird standardmäßig mit git-, time- und s3-Ressourcen ausgeliefert, was in den meisten Fällen ausreichen sollte. Wenn jedoch spezielle Ressourcen erforderlich sind, bietet Concourse eine Möglichkeit, externe Ressourcen zu erstellen, die nicht vom Concourse-Team bereitgestellt werden, sondern als Ressourcentypen definiert werden. Diese Ressourcentypen werden genauso behandelt wie native Ressourcen [11]. Zum Beispiel, CloudFoundry Community erstellte eine externe Ressource namens ³„slack-notification-resource“ für Concourse. Diese Ressource sendet Slack-Benachrichtigungen, wenn der Build erfolgreich und/oder nicht erfolgreich ist.
- **Aufträge (engl. Jobs):** Sie definieren das Verhalten der Pipeline und wie alle definierten Ressourcen durch die Pipeline vorankommen [10].
- **Aufgaben (engl. Tasks):** Concourse definiert Tasks als „die kleinste konfigurierbare Einheit in der Concourse-Pipeline“ [12]. Entwickler können sich Concourse-Aufgaben als Funktionen vorstellen, die Arbeit erledigen.

Listing 4.1 verdeutlicht, wie eine einfache Pipeline mit Concourse CI erstellt werden

³<https://github.com/cloudfoundry-community/slack-notification-resource>

kann, die “Hello, world” ausgibt, wenn die Pipeline ausgelöst wird. Außerdem zeigt die Abbildung 4.2, wie das Ergebnis in Build Nr.01 aussieht. Nach erfolgreicher Ausführung des Jobs wird die Pipeline grün, ansonsten Rot.

Listing 4.1: Concourse-Pipeline drückt ‘Hello, world!’ aus

```
jobs:
- name: hello-world
  plan:
  - task: say-hello
    config:
      platform: linux
      image_resource:
        type: docker-image
        source: {repository: ubuntu}
    run:
      path: echo
      args: ["Hello, world!"]
```

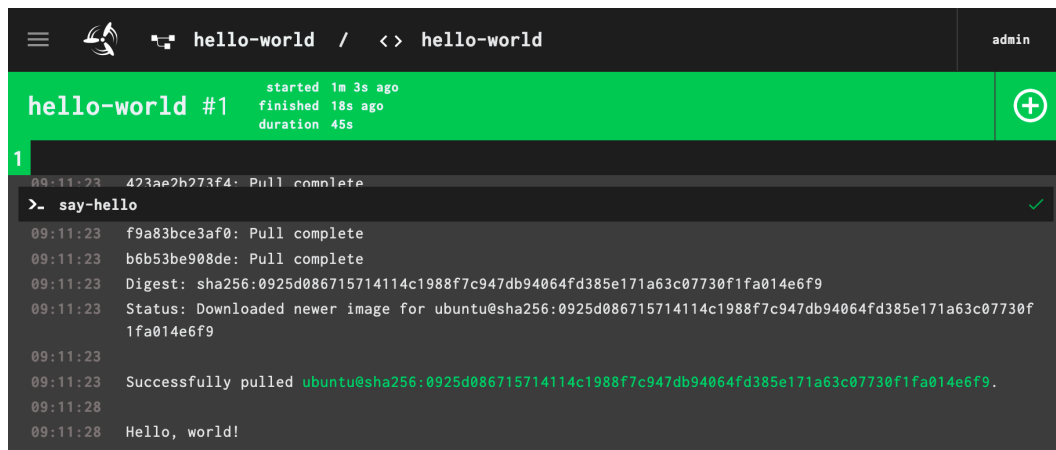


Abbildung 4.2: Drucken von ‘Hello, World!’ nach Auslösen des “hello-world” Auftrag

4.3.2 GitLab

GitLab ist ein webbasiertes Git-Repository, das kostenlose offene und private Repositories bereitstellt. Außerdem ist GitLab eine vollständige DevOps-Plattform, die Entwicklungs-, Betriebs- und Sicherheitsteams in einer einzigen Anwendung vereint. Der Hauptvorteil der Verwendung von GitLab besteht darin, dass alle Teammitglieder in jeder Projektphase zusammenarbeiten können. Es bietet Nachverfolgung von der Planung bis zur Erstellung, um Entwicklern zu helfen, den gesamten DevOps-Lebenszyklus zu automatisieren und die bestmöglichen Ergebnisse zu erzielen. Immer mehr Entwickler verwenden GitLab aufgrund seiner umfangreichen Funktionen und der Verfügbarkeit von Code-Bausteinen. Aus diesem Grund wurde GitLab von verschiedenen großen Unternehmen wie Nvidia, Siemens und Cloud Native Computing Foundation benutzt [15].

Die gesamte Konfiguration der Pipeline wird in der YAML-Konfigurationsdatei *.gitlab-ci.yml* gespeichert und besteht aus folgenden Hauptelementen:

- **Aufträge (engl. Jobs):** Ein Auftrag in GitLab wird oft als "Build-Schritt" bezeichnet, da er der kleinste ausführbare Einheit in GitLab ist. Er kann eine Build- oder Kompilierungs- Deployment-aufgabe sein. Darüber hinaus kann ein Auftrag Unit-Tests ausführen und die Code-Qualitätsprüfungen beinhalten. Ein einzelner Auftrag kann mehrere Befehle (Skripte) enthalten, die ausgeführt werden können [17].
- **Phasen (engl. Stages):** Jeder Auftrag gehört zu einer einzigen Phase. Eine Phase kann null, einen oder mehrere auszuführende Aufträge enthalten. Alle Aufträge in einer einzelnen Phase laufen parallel. Die nächste Phase wird nur dann ausgeführt, wenn alle Aufträge der vorherigen Phase erfolgreich abgeschlossen wurden oder als fehlgeschlagen markiert sind. GitLab hat drei standardmäßigen Phasen nämlich *build*, *testing* and *deploy*. Wenn die Aufträge der Build-Phase erfolgreich abgeschlossen sind, geht GitLab zur Test-Phase über und startet alle Aufträge aus dieser Phase parallel. Wenn die Testphase abgeschlossen ist bzw. wenn alle Aufträge in der Testphase ausgeführt wurden, wird die Bereitstellungsphase ausgeführt. Phasen können manuell hinzugefügt

oder neu definiert werden, in dem man das Stages-Array mit neuen Elementen in `.gitlab-ci.yml` definiert [18].

- **Pipelines:** Pipelines orchestrieren Jobs und Stages und fügen sie alle zusammen. Eine Pipeline wird ausgeführt, wenn eine neue Commit oder ein neues Tag gepusht wird, und führt alle Jobs in ihren Phasen in der richtigen Reihenfolge aus.

Listing 4.2 zeigt eine einfache `gitlab-ci.yml`-Datei, die eine Pipeline zum Drucken von „Hello, World!“ erstellt.

Listing 4.2: Einfache `.gitlab-ci.yml` Datei zum Drucken von “Hello, World!”

```
hello world:
  script: echo "Hello , World!"
```

Nach einem erfolgreichen Lauf der Pipeline wird sie als bestanden (engl. `passed`) markiert, wie auf der Abbildung 4.3 steht.









Status	Pipeline ID	Triggerer	Commit	Stages	Duration
 passed	#458387757 latest		 master -> c45954d9 Create hello world pip...		⌚ 00:00:11 📅 7 minutes ago
 passed	#458382877		 master -> 3a95cd27 Add .gitlab-ci.yml		⌚ 00:00:13 📅 13 minutes ago

Abbildung 4.3: Liste von Pipelines, die als bestanden (engl. `passed`) markiert wurde

Außerdem, wird diese einfache Pipeline wie auf der Abbildung 4.4 aussehen.

Add .gitlab-ci.yml

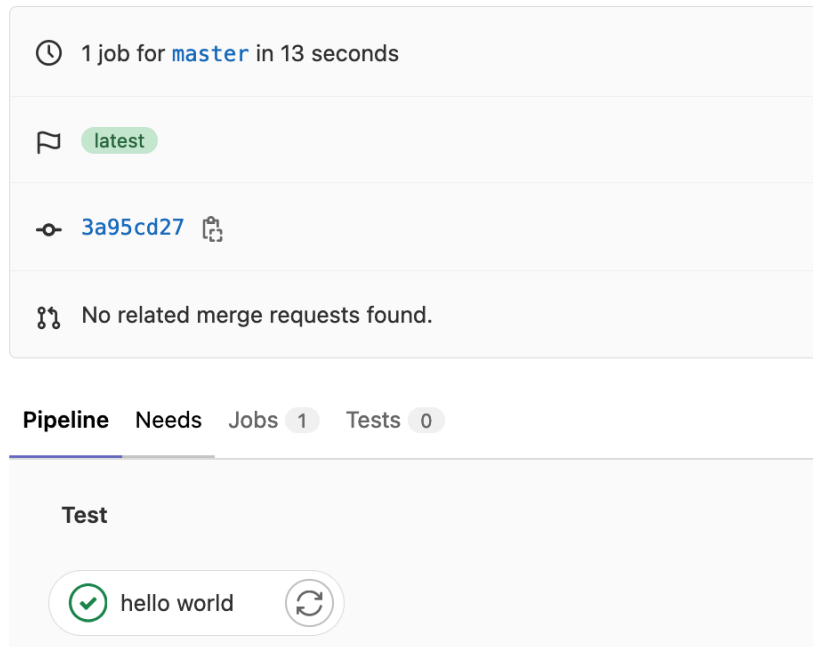


Abbildung 4.4: Einfache hello world pipeline in GitLab

Für weitere Informationen hat der Entwickler die Möglichkeit, einen Blick auf den Pipeline-Build zu werfen, der der Abbildung 4.5 ähnlich sieht.

```
1 Running with gitlab-runner 14.7.0~beta.58.gfa48f33b (fa48f33b)
2   on green-1.shared.runners-manager.gitlab.com/default JLgUopmM
3   ✓ Preparing the "docker+machine" executor 00:05
4   Using Docker executor with image ruby:2.5 ...
5   Pulling docker image ruby:2.5 ...
6   Using docker image sha256:27d049ce98db4e55ddfaec6cd98c7c9cfd195bc7e994493776959db33522383b for r
  uby:2.5 with digest ruby@sha256:ecc3e4f5da13d881a415c9692bb52d2b85b090f38f4ad99ae94f932b3598444b
  ...
7   ✓ Preparing environment 00:01
8   Running on runner-jlguopmm-project-31761671-concurrent-0 via runner-jlguopmm-shared-1643363028-c
  fb8e6d1...
9   ✓ Getting source from Git repository 00:02
10  $ eval "$CI_PRE_CLONE_SCRIPT"
11  Fetching changes with git depth set to 50...
12  Initialized empty Git repository in /builds/yaljazairi/test/.git/
13  Created fresh repository.
14  Checking out c45954d9 as master...
15  Skipping Git submodules setup
16  ✓ Executing "step_script" stage of the job script 00:01
17  Using docker image sha256:27d049ce98db4e55ddfaec6cd98c7c9cfd195bc7e994493776959db33522383b for r
  uby:2.5 with digest ruby@sha256:ecc3e4f5da13d881a415c9692bb52d2b85b090f38f4ad99ae94f932b3598444b
  ...
18  $ echo "Hello, World!"
19  Hello, World!
20  ✓ Cleaning up project directory and file based variables 00:00
21  Job succeeded
```

Abbildung 4.5: GitLab Pipeline-Build vom hello world Job

4.3.3 CircleCI

CircleCI ist eine Plattform für kontinuierliche Integration und Bereitstellung, die Entwicklungsteams bei der schnellen Veröffentlichung von Code und der Automatisierung von Build, Test und Bereitstellung unterstützt. CircleCI kann so konfiguriert werden, dass sehr komplexe Pipelines mit Caching, Docker Layer Caching, Ressourcenklassen und vielem mehr effizient ausgeführt werden [3]. CircleCI sendet auch eine E-Mail- oder Slack-Benachrichtigung über Erfolg oder Misserfolg nach Abschluss der Tests [4]. Führende Unternehmen wie Docker, Heroku, Kickstarter, Nextdoor, Udemy, Coinbase und Cucumber haben ihren Entwicklungsprozess erfolgreich über CircleCI durchgeführt [2]. Auf diese Weise gelang es ihnen, die Lieferung zu beschleunigen und die Produktqualität zu verbessern. Außerdem ist die

Konfigurationsdatei vom CircleCI in YAML geschrieben und das Tool unterstützt viele Programmiersprachen. CircleCI bietet 1.000 Build-Minuten pro Monat und unbegrenzte Repos in einem kostenlosen Plan. Allerdings, stehen Skalierbare bezahlte Pläne zur Verfügung, wenn die kostenlose Plan nicht reicht [5].

4.3.4 Jenkins

Jenkins ist eines der führenden Open-Source-Tools für Continuous Integration / Continuous Delivery and Deployment (CI/CD). Es ist in Java geschrieben und verfügt über 300 Plugins zur Unterstützung der Erstellung und Prüfung jedes Projekts. Außerdem, unterstützt Jenkins alle wichtigen Sprachen und können seine Pipelines über *Jenkinsfile* konfiguriert werden. Allerdings, müssen die Entwickler, die Jenkins benutzen möchten, die Groovy⁴ Programmiersprache beherrschen [19].

⁴<https://groovy-lang.org/>

5 Fallstudie

Es sollen zwei Ampeln nachgebaut werden, die in einer experimentellen Umgebung die Funktionsweise realer Ampeln zeigen. Eine der Ampeln soll die gelbe LED blinken und die andere soll im normalen Betrieb sein. In diesem Projekt soll der CI/CD Methode von Development bis zum Deployment angewendet werden.

5.1 Anforderungen

In diesem Abschnitt werden die funktionale und nicht-funktionale Anforderungen für die Realisierung des Projekts analysiert.

5.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die gewünschte Funktionalität und das Verhalten des Systems, die in Tabelle 5.1 aufgelistet sind.

5.1.2 Nicht-Funktionale Anforderungen

Nicht-funktionale Anforderungen beschreiben die Qualität der oben genannten Funktionen, die erreicht werden müssen. Daher haben sie einen erheblichen Einfluss auf Ressourcenverbrauch, Entwicklung und Wartung. Darüber hinaus tragen diese Anforderungen dazu bei, die Akzeptanz des Systems zu verbessern. Einige dieser Anforderungen werden im Folgenden aufgelistet und erörtert.

- **Zuverlässig:** Zuverlässigkeit stellt die Grundvoraussetzung für die Akzeptanz des Systems dar. Das korrekte Verhalten und der Übergang in einen sicheren

Must-have	<ol style="list-style-type: none">1. Simulierung zwei Verkehrsampeln mithilfe von Raspberry Pi's und LEDs2. Automatische Steuerung der LEDs durch GPIO Schnittstelle3. Entwicklung einer Software für das Blinken der gelben LED4. Entwicklung einer Software für den Normalbetrieb der Ampel5. Automatische Software-Test6. Automatische Bereitstellung (release) der Software7. Implementierung von CI/CD Software-Entwicklungsmethode
Should-have	<ol style="list-style-type: none">1. Automatische Containerisierung der für das Ampelsystem entwickelten Software mithilfe von Docker2. Automatische Veröffentlichung von containerisierten Software auf Docker Hub3. Orchestrierung der Docker-Containers mithilfe von Kubernetes
Could-have	<ol style="list-style-type: none">1. Automatische Übertragung (deploy) der Software an das Endgerät

Tabelle 5.1: Funktionale Anforderungen

Zustand im Fehlerfall muss immer gewährleistet sein. Falls die Übertragung der Software in einem inkorrekten Zustand endet, müssen Entwickler in der Lage sein, das System in den korrekten Zustand wiederherzustellen.

- **Skalierbar:** Das Projekt soll skalierbar sein. Wenn man eine neue Ampel nachbauen möchte, sollte es einfach und nicht kompliziert sein.
- **Fehlertoleranz:** Im Falle eines Fehlers sollte die Pipeline die weitere Ausführung anderer Schritte stoppen, wodurch Fehler vermieden werden, die aufgrund einer nachfolgenden Ausführung auftreten können.
- **Robust:** Der Pull-Request soll nicht gemergt werden, bevor die Fehlerfreiheit des Programms durch bestehenden Unit-Test bestätigt wird.
- **Zeiteffizient:** Der gesamte Prozess, von der Entwicklung bis zur Auslieferung,

muss in kurzer Zeit und ohne Unterbrechung erfolgen.

- **Echtzeitüberwachung:** Während der Übertragung des Software-Updates soll dieses in Echtzeit überwacht werden.

5.2 Konzept

5.2.1 Geeignete Software-Update-Strategie der Ampelanlage

Heutzutage sind mehr Geräte mit dem Internet verbunden als je zuvor, was bedeutet, dass Software-Updates über das Internet und nicht über eine traditionelle Schnittstelle bereitgestellt werden müssen. Das spart viel Zeit und Geld bei der Wartung der Software und entlastet vor allem den Facharbeitern. Darüber hinaus wird die Fehlersuche im Fehlerfall durch den einfachen Fernzugriff auf das betroffene System erheblich vereinfacht. Diese Strategie wird als Software Over The Air (SOTA) benannt. Durch diese Strategie wird die Echtzeitüberwachung der Übertragung ermöglicht, was das ganze Software-Update der Ampelanlage übersichtlicher macht. Darüber hinaus wird durch SOTA Flexibilität bei der Software-Update erreicht, was bedeutet, dass jede Softwareversion mit einem einzigen Klick auf das System übertragen werden kann. Dies macht die Strategie einfach zu handhaben. Neben all diesen Vorteilen hat das System noch einen weiteren Vorteil, nämlich die Anzahl der Updates ist nicht begrenzt und somit nicht zeitabhängig.

5.2.2 Software Update-Architektur der Ampelanlage

Um die oben genannten Vorteile nutzen zu können, ist es notwendig, eine Internet-Schnittstelle zur Außenwelt zu schaffen. In diesem Zusammenhang und aufgrund der im Raspberry Pi integrierten Internetschnittstelle eignet sich der Raspberry Pi für dieses System, und weil es sich um ein Minicomputer handelt, erfüllt die in diesem Projekt aufgelistete Anforderungen. Die Abbildung 5.1 zeigt, wie ein Entwickler, über das im Raspberry Pi integrierte Internetschnittstelle ein Update auf der Ampelanlage übertragen kann, nachdem er die Software auf GitLab hochgeladen hat.

Die Verteilung der korrekten Software-Version an den entsprechenden Ampelanlage erfolgt über Raspberry Pis, die direkt mit den Ampelanlagen angeschlossen sind.

Die Software-Update-Architektur in Abbildung 5.1 zeigt deutlich, dass für die Anforderungen, die im Abschnitt 5.1 aufgelistet sind, eine hierarchische Software-Deployment Topologie dafür geeignet ist.

Die Architektur von Kubernetes, die im Kapitel ?? beschrieben wurde, bietet die Möglichkeit ein hierarchische Firmware-Deployment aufzubauen.

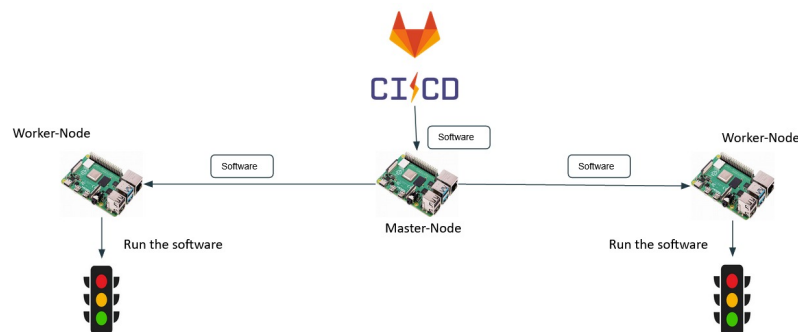


Abbildung 5.1: Übersicht der Software Update

5.2.3 Auswahl der geeigneten Release-Managementmethode

Die Unterschiede der Release-Managementmethoden zeigen, dass die CD-Methode es ermöglicht, qualitativ hochwertige Software-Release in kurze Zeit auf einem Endgerät zu übertragen. CD Methode bietet automatisierte Build-, Test- und Deploymentschritte, um vom Menschen verursachte Fehler zu vermeiden, die bei der manuellen Durchführung der Phasen auftreten können. Außerdem können in jeder Phase Probleme auftreten. Abhängigkeiten zwischen den Schritten können eine schnelle Fehlerkorrektur garantieren. Tritt in einer Phase ein Fehler auf, wird die Pipeline unterbrochen und somit weitere Fehler verhindert. Die Software kann nahtlos, sicher, zuverlässig und wiederholt auf dem Endgerät übertragen werden. Darüber hinaus kann der Entwickler den Fortschritt der Phasen beobachten, was hilft, wenn während

der Phase ein Fehler auftritt, kann der Entwickler nur ab dieser Phase und nicht von allen Phasen aus iterieren. Bei anderen Release-Managementmethoden benötigen Produkthersteller Releasemanager, um Entscheidungen darüber zu treffen, wann Releases erstellt werden, wie einzelne Schritte ausgeführt werden und wann Software ausgeliefert werden soll. Die Methode CD ermöglicht es dem Release Manager, sich mehr auf die operative Seite zu konzentrieren, wie beispielsweise die Erstellung eines automatisierten Prozesses und Workflows zur sicheren Migration von Code in das Endprodukt, anstatt sich auf die Planungs-, Entwicklungs- und Testphasen zu konzentrieren.

5.2.4 Auswahl der geeigneten Release-Strategie

Die Wahl einer geeigneten Release-Strategie hängt vom Hersteller des Produkts und dem Produkt selbst ab. Bei modernen medizinischen Geräten, Fahrzeuge und IoT Geräte sind Fehlerbehebung und neue Funktionen für einen besseren Service für die Kunden unerlässlich. Daher wird in dieser Fallstudie eine flexible Release-Strategie bevorzugt. Die Schritte im Release-Management-Prozess bleiben generisch, sodass der Wechsel zu anderen Release-Strategien einfach sein kann. Eine flexible Release-Strategie sieht laut Dr.-Ing. Kühn in der Regel die Planung von Änderungen an einem bestimmten Produkt vor[21].

5.3 Entwicklungsvorgang

Als Team an der Entwicklung von Software zu arbeiten, ist eine ziemliche Herausforderung, und braucht ein gutes Management. Diesbezüglich soll ein Konzept entwickelt werden, die verteilte Entwicklung von Software der simulierten Ampelanlage zu organisieren.

Für die Umsetzung dieses Konzeptes in der Praxis ist die richtige Wahl eines VCS entscheidend. Aufgrund der Vorteile von GitLab, die in im Abschnitt 4.3.2 beschrieben sind, wird er als das richtige VCS für dieses Konzept bevorzugt. Denn GitLab

vereinfacht nicht nur die Entwicklung und Bereitstellung von Software, sondern ermöglicht auch automatisierte Release-, Build- und Deployment-Schritte.

5.4 Implementierung

Die ausgewählten und entwickelten Konzepte werden im Folgenden umgesetzt und die Details diskutiert.

5.4.1 Aufbau der experimentellen Umgebung

Zur Ausführung der vorher ausgewählten SOTA-Methode werden ein Server und ein Client benötigt. Der Server kommuniziert mit dem Client über die in das Produkt eingebaute Internetschnittstelle. Dadurch wird das Update übertragen und während der Übertragung Nachrichten ausgetauscht. Für diese Aktualisierungsmethode ist eine Testumgebung konfiguriert. Die Umgebung umfasst ein Server-zu-Client-Kommunikationssystem. Als Server und Clients werden die Raspberry Pis zunutze genommen. Ampelanlagen sind mit den Raspberry Pis, die als Client vorgesehen sind, verbunden. In diesem Projekt werden die reale Ampelanlage durch LEDs und Breadboard simuliert, um die serielle Kommunikation zu verwirklichen.

Außerdem wird die Kubernetes-Cluster konfiguriert, der Master-Node wird auf einem Raspberry pi installiert und die Worker-Nodes sind in dieser Architektur die Raspberry Pis, die jeweils mit einer Ampelanlage verbunden sind. Die installierten Worker-Nodes sollen mit dem Master-Node registriert werden. Um Kubernetes-Anwendungen einfach nutzen und verwalten zu können, wird auch Helm-Tool auf den Master-Node heruntergeladen.

5.4.2 Implementierung des Release-Management-Prozesses

Der Release-Management-Prozess enthält alle notwendigen Schritte, bis die Software auf das Endprodukt verteilt wird. Dies sind die Entwicklung-, Build-, Test- und Deployment-Phase. Sobald der Entwicklungsprozess abgeschlossen ist, erfolgt die

automatischen weiteren Phasen wie Build-, Release- und Deployment-Phase. Das Update kann sowohl für alle angeschlossenen Ampelanlagen als auch für verschiedene Ampelanlagen durchgeführt werden.

Im Laufe dieses Kapitels wird den ausgewählten Entwicklungsprozessentwurf umgesetzt und wird darauf eingegangen, wie es in der Realität ein Release erstellt wird. In der Buildphase wird die benötigte Dateien und deren Abhängigkeiten übersetzt und das Ergebnis zur Verfügung gestellt. Diese Phase ist für den gesamten Entwurf und die Implementierung der Architektur erforderlich. Nach der Build- und Release-Phase erfolgt die Deployment-Phase. In dieser Phase wird die Übertragung der Software bis zum Ampelanlage geschehen. Die Ausführung der Phasen erfolgt so weit wie möglich automatisiert und falls während der Ausführung Fehler auftreten, wird die Pipeline unterbrochen, wodurch die Fortsetzung der anderen Phasen verhindert wird.

5.4.2.1 Umsetzung der Continues Integration und Continues Delivery Methode

Nun nach der Entwicklungsphase kommt die CD-Phase zustande. Nach dem Start der CD-Pipeline beginnt die kontinuierliche Deployment-Phase an. Zu CD gehört die Build-,Release- und die Deployment-Phase. Um Continues Integration und Continues Delivery mit GitLab nutzen zu können, wird einen GitLab Runner benötigt. GitLab Runner ist eine Build-Instanz, die verwendet wird, um Aufgaben auf mehreren Computern auszuführen und die Ergebnisse durch „artifacts“ an GitLab zu übertragen.

In diesem Projekt wird der Gitlab Runner auf dem Master-Node konfiguriert und mit einem Executer vorgesehen, der die vordefinierten Aufgaben in Gitlab ausführt. Sobald der Runner registriert ist, initiiert das Runner Tag die Kommunikation zwischen der Gitlab-Plattform und dem Master-Node, auf dem der Runner installiert ist. Die möglichen Executors sind beispielsweise SSH, Shell, Docker, Kubernetes und einige andere. In diesem Projekt wird Kubernetes-Executer benutzt.

Im Folgenden werden die einzelnen Phasendurchführungen erklärt.

Build-Phase

Nun wenn die Pipeline gestartet wird, beginnt die Phasendurchführung der CD. Die erste Phase ist die Build-Phase, da in diesem Projekt Programmiersprache Python verwendet wird, wird diese Phase nicht in der Pipeline-Phasen hinzugefügt.

Test-Phase

Nachdem der Prozess Merge von DevelopBranch mit MasterBranch gestartet wird, folgt die CI/CD-Testphase. Für die Testphase wird die Master-Node verwendet. In diesem Projekt wird unittest für den RaspberryPi GPIO ausgeführt. Der Erfolg des Merge-Prozess hängt von den Ergebnissen von unittest ab. Wenn der Test nicht erfolgreich abgeschlossen werden kann, wird auch der Merge-Prozess abgebrochen, um eine weitere Ausführung der CI/CD-Phase zu verhindern.

Publish and Release Phase

Nach erfolgreichem Merge-Prozess kann die weitere CI/CD-Phase fortgesetzt werden. Nach der Vergabe des Release-Tags wird die Pipeline gestartet und die zweite CI/CD-Phase dieses Projektes ist die „Publish and Release Phase“. In dieser Phase wird die entwickelte Software als Docker-Image gekapselt, auf die Dockerhub-Plattform hochgeladen und zur Übertragung und/oder Installation bereitgestellt. Für die Ausführung dieser Phase wird erneuert der Runner der Master-Node benötigt. Durch das Hochladen des Images wird ein neuer Release erstellt. Außerdem wird ein gewöhnlicher Release auf der GitLab-Plattform erstellt. Die Abbildung 7.4 veranschaulicht die Release-Phase anhand eines Sequenzdiagramms.

6 Evaluation

7 Zusammenfassung

Literatur

- [1] Atlassian. *What are the differences between CI, CD, and CD?* Hrsg. von Sten Pittet. URL: <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment> (besucht am 02. 12. 2021).
- [2] CircleCI. *Customers*. URL: <https://circleci.com/customers/> (besucht am 20. 11. 2021).
- [3] CircleCI. *Homepage*. URL: <https://circleci.com/> (besucht am 20. 11. 2021).
- [4] CircleCI. *Notifications*. URL: <https://circleci.com/docs/2.0/notifications/> (besucht am 20. 11. 2021).
- [5] CircleCI. *Pricing*. URL: <https://circleci.com/pricing/> (besucht am 20. 11. 2021).
- [6] ConcourseCI. *Homepage*. URL: <https://concourse-ci.org/> (besucht am 12. 12. 2021).
- [7] ConcourseCI. *Pipelines*. URL: <https://concourse-ci.org/pipelines.html> (besucht am 01. 11. 2021).
- [8] ConcourseCI. *Resources*. URL: <https://concourse-ci.org/resources> (besucht am 01. 11. 2021).
- [9] ConcourseCI. *git-resource*. URL: <https://github.com/concourse/git-resource> (besucht am 01. 11. 2021).
- [10] ConcourseCI. *jobs*. URL: <https://concourse-ci.org/jobs.html> (besucht am 02. 11. 2021).
- [11] ConcourseCI. *resource-types*. URL: <https://concourse-ci.org/resource-types.html> (besucht am 02. 11. 2021).

- [12] ConcourseCI. *tasks*. URL: <https://concourse-ci.org/tasks.html> (besucht am 02. 11. 2021).
- [13] ConcourseCI. *time-resource*. URL: <https://github.com/concourse/time-resource> (besucht am 01. 11. 2021).
- [14] Martin Fowler. *Continuous Integration*. 1. Mai 2006. URL: https://moodle2019-20.ua.es/moodle/pluginfile.php/2228/mod_resource/content/2/martin-fowler-continuous-integration.pdf (besucht am 05. 12. 2021).
- [15] GitLab. *About Page*. URL: <https://about.gitlab.com/> (besucht am 06. 11. 2021).
- [16] GitLab. *Continuous Delivery*. URL: <https://docs.gitlab.com/ee/ci/introduction/#continuous-deployment> (besucht am 19. 11. 2021).
- [17] GitLab. *Jobs*. URL: <https://docs.gitlab.com/ee/ci/jobs/> (besucht am 06. 11. 2021).
- [18] GitLab. *Stages*. URL: <https://docs.gitlab.com/ee/ci/yaml/#stages> (besucht am 07. 11. 2021).
- [19] Jenkins. *Pipelines*. URL: <https://www.jenkins.io/doc/tutorials/#pipeline/> (besucht am 22. 11. 2021).
- [20] David Farley Jez Humble. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 2010. ISBN: 978-0321601919.
- [21] Dipl.-Wirt.-Ing. Arno Theodor Kühn. „Systematik zur Release-Planung intelligenter technischer Systeme“. Doktor-Thesis. Universitätsbibliothek Paderborn, 27. Okt. 2016. URL: <https://digital.ub.uni-paderborn.de/hs/content/titleinfo/2404085>.
- [22] Mathias Meyer. *Continuous Integration and Its Tools*. 1. Juni 2014. URL: https://ieeexplore.ieee.org/abstract/document/6802994?casa_token=SZ9-UsnQUSEAAAAA:6A0fd8J5Sw7RSixp5_BZFuWCC405q2Ff84BNgx9mGGNUw4A_Lg2ib2FNFbzWbBGJv (besucht am 02. 12. 2021).

- [23] Liming Zhu Mojtaba Shahin Muhammad Ali Babar. *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*. 22. März 2017. URL: <https://ieeexplore.ieee.org/document/7884954> (besucht am 15.12.2021).
- [24] RedHat. *What is CI/CD?* 31. Jan. 2018. URL: <https://www.redhat.com/en/topics/devops/what-is-ci-cd> (besucht am 28.11.2021).

Abbildungsverzeichnis

4.1	Die Zusammenhang zwischen CI/CD [23]	10
4.2	Drucken von 'Hello, World!' nach Auslösen des "hello-world" Auftrag	12
4.3	Liste von Pipelines, die als bestanden (engl. passed) markiert wurde	14
4.4	Einfache hello world pipeline in GitLab	15
4.5	GitLab Pipeline-Build vom hello world Job	16
5.1	Übersicht der Software Update	22

Tabellenverzeichnis

5.1	Funktionale Anforderungen	20
-----	---------------------------	----

Listings

4.1	Concourse-Pipeline drückt 'Hello, world!' aus	12
4.2	Einfache .gitlab-ci.yml Datei zum Drucken von "Hello, World!"	14

Abkürzungsverzeichnis

CI/CD Continuous Integration / Continuous Delivery and Deployment

Anhang

