HTWK Hochschule für Technik, Wirtschaft und Kultur Leipzig

Abschlussarbeit zur Erlangung des akademischen

Grades Bachelor of Engineering (B.Eng.)

im Studiengang Telekommunikationsinformatik
der Fakultät Digitale Transformation
der Hochschule für Technik, Wirtschaft und Kultur Leipzig

Umsetzung von Continuous Integration im Kontext bestehender Anwendungen

vorgelegt von

PHILIPP BRAUN, 73035, 18TIB-1
Magdeburg, den 5. März 2022

Erstgutachter: Prof. Dr.-Ing. Christian-Alexander Bunge, HTWK Leipzig
Zweitgutachter: M. Eng. Eric Schmieder, Deutsche Telekom IoT GmbH

Kurzfassung

Software-Testing ist ein Kernbestandteil der modernen Softwareentwicklung. Immer größere Entwicklerteams und Projekte benötigen sichere und zuverlässige Produktveröffentlichungen. Um diesen Wunsch gleich zu werden, müssen häufig und programmatisch Tests an der implementierten Anwendung durchgeführt werden. Diese Tests manuell durchzuführen, benötigt jedoch viel Zeit und ist inhärent Fehleranfällig.

Mithilfe von Continuous Integration kann dieser Schritt vollständig automatisiert werden. Dieser Prozess wird in der Literatur umfassend beschrieben, jedoch wird sich hierbei meist lediglich mit idealen Ausgangspunkten befasst. Zu einem Projekt werden entwicklungsbegleitend Softwaretests implementiert nach dem Prinzip des Test-Driven-Development. In der Praxis werden jedoch Ansätze benötigt, die für große existierende Anwendungen ohne bestehender Testsuite, anwendbar sind.

Abstract

Software testing is a core component of modern software development. Ever growing development teams and projects require safe and reliable product releases. Frequent and programmatic tests are needed to make this requirement possible. Doing these tests manually requires a lot of time and is inherently error prone.

Using Continuous Integration can completely automate this step. Relevant literature explores this process to great extent, but mostly relies on ideal starting points. Software tests are created before the implementation, following the test-driven development process. In practice, there need to be approaches applicable to existing projects without a test suite.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API ..... *Application Programming Interface*
BS ..... *Betriebssystem*
CD ..... *Continuous Deployment*
CDE ..... *Continuous Delivery*
CI ..... *Continuous Integration*
CSR ..... *Change success ratio*
DBMS ..... *Datenbankmanagementsystem*
DSCp ..... *Diagnostic support capability*
FR ..... *Full Reliability*
GUI ..... *Graphical User Interface*
SPOF ..... *Single point of failure*
TDD ..... *Test-Driven-Development*
TUR ..... *Total Unreliability*
UI ..... *User Interface*
VCS ..... *Version Control System*
VitR ..... *Vital Reliability*

Glossar

Begriff	Erläuterung
Defekt	„Ein statischer Mangel/Schwäche in einer Software. Führt, wenn ausgeführt, zu einer Fehlfunktion“ [1, S. 29].
Django	Django ist ein Webframework zur Entwicklung von Webanwendungen. Es basiert auf der Programmiersprache Python. [2]
Dump	Ein Dump ist ein Auszug aus einer Datenbank. Diese werden hautsächlich zur Migration oder Datensicherung genutzt.
Fehler	„Ein inkorrekter interner Zustand der Software als Manifestation eines Defekts“ [1, S. 29].
Fehlfunktion	„Ein inkorrektes externes Verhalten bezüglich gewisser Anforderungen bzw. anderen spezifizierten erwarteten Verhaltens“ [1, S. 29].
Fixture	Fixtures werden zur einheitlichen Initialisierung von Systemen genutzt, um Softwaretests wiederholbar zu machen. Beispiel einer solchen Fixture sind vorgegebene Daten, die zur Befüllung einer Testdatenbank vor jedem Test genutzt werden [3, S. 99–106].
Version Control System	Version Control Systeme (VCS) vereinfachen und beschleunigen den Softwareentwicklungsprozess. Dateien und ihre Historie werden verfolgt und für viele Teammitglieder gleichzeitig zur Bearbeitung verfügbar gemacht. Zusätzlich erlauben VCS das Einrichten diverser Workflows zur Realisierung von CI/CD Prinzipien [4].

1 Einleitung

1.1 Relevanz & Motivation

In der heutigen Zeit und besonders im Zuge der drastisch voranschreitenden Digitalisierung kommen bei Anwendungen oftmals ähnliche Mängel auf. Das liegt daran, dass auf Grund unseres dynamischen Lebens, neue Software besonders schnell ausgeliefert werden muss, um zu vermeiden, dass deren eigentlicher Nutzen noch vor Release obsolet wird. Dieser enorme zeitliche Druck sorgt vor allem bei der Entwicklung für eine reduzierte Qualität der Anwendung, da ein Entwickler meist sehr viele verschiedene Wünsche umsetzen muss.

Zu häufig werden daher Softwaretests nicht entwicklungsbegleitend erstellt, da sie anfänglich nur die Entwicklung verlangsamen. Die positiven Effekte solcher Softwaretests zeigen sich erst später, wenn das Projekt einen produktionsreichen Status erreicht hat. In vielen Projekten wird sich daher nicht die Zeit genommen, um korrekt Testgetriebene Anwendungsentwicklung zu betreiben. Dies wirkt sich dann negativ in der späteren Entwicklung und Instandhaltung der Anwendung aus.

Besonders aus der Sicht agiler DevOps-Teams ist es wichtig in kurzen und regelmäßigen Abständen eine neue betriebsbereite Applikation zu erstellen. Der Continuous Integration / Continuous Deployment (CI/CD) Prozess ermöglicht es, robuste und vollständige Applikationen in kurzen Zeiträumen automatisiert zur Verfügung zu stellen. Die Umsetzung dieses Ansatzes muss jedoch individuell an die Anforderungen und Funktionalitäten der Anwendung angepasst werden.

Da CI/CD-Prozesse auf eine Anwendung zugeschnitten sein müssen, werden sie traditionell zeitgleich mit der Anwendung in Form einer „Testgetriebenen Anwendungsentwicklung“ erstellt. Diese Prozesse nachträglich einzuführen ist jedoch arbeitsintensiv, da sehr viel Zeit benötigt wird, um sie zu implementieren.

1.2 Zielstellung und Forschungsfragen der Arbeit

Das Ziel dieser Arbeit ist es, mögliche Methoden zur Implementierung von Softwaretests in bestehenden Anwendungen zu analysieren und die unten gegebenen Forschungsfragen zu beantworten. Diese Analyse soll anhand eines betriebsinternen Projekts der Deutschen Telekom AG durchgeführt und mit vorhandener Literatur verglichen werden. Hierbei sollen exemplarisch die Vor- und Nachteile verschiedener Herangehensweisen bei der Erstellung von Softwaretests dokumentiert und dargestellt werden. Daraus bilden sich die folgenden Forschungsfragen für diese Arbeit:

- Welche Ansatzweise eignet sich am besten zur schnellen und effektiven Abdeckung von Testfällen?
- Wie groß ist die Auswirkung von Softwaretests im Bezug zur Fehleranfälligkeit in Produktivsystemen?
- Ist eine vollständige Testabdeckung bei der nachträglichen Implementierung von Softwaretests empfehlenswert?
- Wie geeignet und effektiv sind existierende, theoretische Ansätze in der Implementierung von Softwaretests, um die Zuverlässigkeit und Geschwindigkeit von Releases in bestehenden Anwendungen zu garantieren?

Davon ausgehend leiten sich folgende Hypothesen ab:

- Wenn die gewünschte Zuverlässigkeit erreicht ist, dann ist eine vollständige Testabdeckung nicht benötigt.
- Je mehr Testfälle benötigt werden, um eine hohe Testabdeckung zu erreichen, desto weniger eignet sich der Ansatz zur schnellen und effektiven Abdeckung.
- Die Auswirkungen von Tests auf die Fehleranfälligkeit sind Signifikant.

2 Forschungsgrundlage

Im folgenden Kapitel werden die grundlegenden Begriffe und fundamentale Kenntnisse dieser Arbeit betrachtet. Es wird genauer auf die Themen Continuous Integration, Continuous Delivery, Continuous Deployment sowie DevOps eingegangen und in ein Verhältnis mit dem Thema dieser Arbeit gebracht.

2.1 Definition von Continuous Integration

Continuous Integration (CI) ist ein Softwareentwicklungsprozess bei dem Code kontinuierlich in ein zentrales Repository zusammengeführt und getestet wird. Dieses Repository agiert gleichzeitig als Version Control System. Jede Codeintegration wird durch automatische Test- und Build-Prozesse verifiziert. Frühes und häufiges testen kann zur Prävention von Integrationsproblemen bzw. kritischer Fehler zwischen Code-Updates beitragen [5, S. 467]. Testprozesse müssen robust und tiefgreifend sein und werden mit bestimmten Definitionen voneinander abgegrenzt. Diese werden im Folgenden weiter erläutert:

Black-Box Testing

Der Black-Box-Ansatz behandelt Software wie eine „Black-Box“, Funktionalitäten werden folglich ohne Wissen über die Implementation und ohne Zugriff auf den Quellcode geprüft. Die zuständigen Tester kennen lediglich die Ausgaben, welche das Programm erzeugen soll [6, S. 54–55].

White-Box Testing

Im Vergleich zum Black-Box-Ansatz, ist im White-Box-Ansatz der Quellcode und das Wissen über die Implementation von Funktionalitäten verfügbar. Hierbei existieren unterschiedliche Level an Tests, abhängig vom Umfang des zu testenden Codesegments:

Unit Tests

Unit Tests sind automatisierte Tests, welche eine einzelne Sektion (Unit) auf ihre Funktionalität überprüft. Sie stellen die Grundlage für weitere Testarten dar, da sie den Standort von Fehlern und Fehlfunktionen genau aufzeigen können [7, S. 75].

Integration Tests

Ein Integration Test kombiniert mehrere einzelne Units und testet sie als Gruppe auf Compliance und korrekte Funktionalität. Unit Tests bilden die Grundlage für Integration Tests, da ohne sie ein Fehler in Integration Tests nur schwer auffindbar sind. Ohne sie kann nicht gesagt werden, welche Unit einen Defekt aufweist [8, S. 71].

Regression Tests

Ein Regression Test soll sicherstellen, dass bereits entwickelte und getestete Software auch nach Änderungen immer noch korrekt funktioniert. Dies wird durch wiederholtes Testen von bereits getesteten Komponenten mithilfe von Unit & Integration Tests bewältigt [9].

White-Box-Testing-Methoden finden in der Continuous Integration breite Akzeptanz, da sie leicht automatisierbar und wiederholbar sind. Zudem geben sie Konfidenz in die Integrität der Software.

Alleinstehend gibt Continuous Integration lediglich Feedback an Entwickler ob Softwaretests, nach Änderungen der Entwickler, erfolgreich sind oder fehlschlagen.

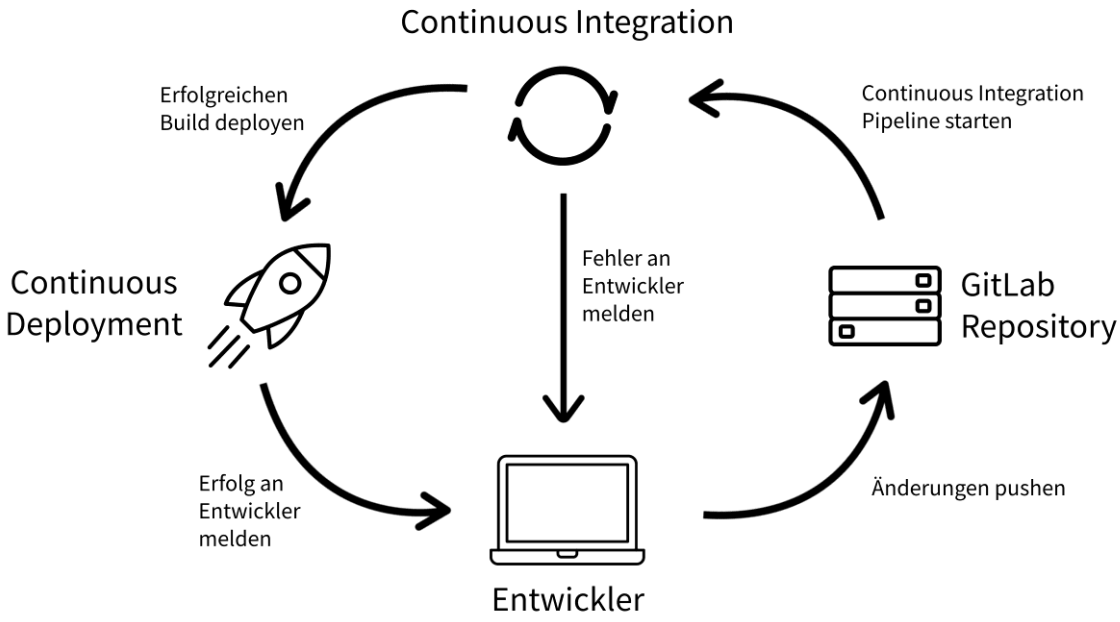

Abbildung 1: Continuous Integration Zyklus

In Kombination mit Continuous Delivery/Deployment bildet die CI/CD-Chain jedoch, eine solide Pipeline zum automatischen Building, Testing und Deployment. Abbildung 1 zeigt einen vereinfachten Aufbau einer CI/CD-Pipeline. Hierbei startet ein Entwickler den Prozess durch das Pushen einer Änderung am Code auf das genutzte Version Control System (VCS). In diesem Beispiel werden die Änderungen auf ein GitLab-Repository gepusht. Hiervon ausgehend wird dann die CI/CD-Pipeline mithilfe eines Pipeline-Workflows gestartet. Daraufhin wird im Continuous Integration-Schritt die Anwendung getestet. Hierzu wird zuerst eine Umgebung aufgesetzt, die identisch zur produktiven Umgebung ist. In dieser werden anschließend vorher definierte Testfälle durchgeführt. Bei einem Fehler wird die Pipeline abgebrochen und der Fehler an den Entwickler beziehungsweise das Entwicklerteam gemeldet. Bei einem erfolgreichen Build wird der Continuous Deployment Schritt initiiert. Hierbei wird die Software auf der Produktivumgebung auf die neue Version aktualisiert. Nachdem dieser Schritt fertiggestellt ist, wird der Erfolg dem Entwickler mitgeteilt. Existierende Projekte müssen sich jedoch einzigartigen Herausforderungen bei der Implementierung dieser Methoden stellen. Diese werden im Verlauf des nächsten Kapitels erläutert.

2.1.1 Herausforderungen existierender Anwendungen

Eine große Herausforderung bei der Implementierung von CI/CD-Prinzipien ist der Mangel einer Testsuite. Ein Fehler in der Software muss sich nicht extern als Fehlfunktion bemerkbar machen. Ein momentan korrekter Zustand bedeutet nicht, dass in Zukunft kein fehlerhafter Zustand auftreten kann. Wenn keine Softwaretests vorhanden sind, gestaltet sich die Bestimmung der Ursache eines Fehlers oder einer Fehlfunktion als sehr schwierig, da nicht gesagt werden kann ob neue Änderungen an der Software fehlerhaft sind oder ob sie einen Mangel an einer anderen Softwarekomponente hervorrufen.

Laut Sahin et al. [10, S. 3926–3927] ist dies eine der am weitverbreitetsten Hürden bei der Implementierung von Continuous Integration, da es unter anderem ein zeitaufwändiger und mühsamer Prozess ist, aussagekräftige und automatisierte Tests zu erstellen.

Weitergehend werden folgende Probleme bei der Übernahme von CI/CD-Praktiken beschrieben:

1. *Mangel an Wahrnehmung & Transparenz:* Continuous Delivery Prozesse sollten so entworfen sein, dass der Status des Projekts, die Anzahl der Fehler und die Qualität der Features sichtbar und transparent für alle Teammitglieder sind.
2. *Koordinierungs- und Kollaborationsherausforderungen:* Eine erfolgreiche Implementierung von CI/CD-Praktiken erfordert mehr Kollaboration und Koordinierung zwischen einzelnen Teammitgliedern. Im Vergleich zu seltenen Releases, benötigen Continuous Releases mehr Kommunikation und Koordinierung mit Operations-Teams.
3. *Mangel an Investment:* Dies umfasst sowohl die Kosten zur Umstellung auf CI/CD als auch den Mangel an Expertise und den entsprechenden Tools sowie Technologien.
4. *Organisatorische Prozesse, Strukturen und Richtlinien:* Die effektive Nutzung von CI/CD benötigt agilere Arbeitsweisen als klassische Methoden zur Softwareentwicklung. Schwierigkeiten kommen auf, wenn organisatorische Strukturen, welche auf einen großen Zeitraum zwischen Releases ausgelegt waren, sich nicht entsprechend anpassen können.

Anhand der genannten Punkte ist ersichtlich, dass eine Umstellung auf CI/CD sich sehr schwierig gestalten kann. Dies ist speziell der Fall bei einem zu geringen Investment an Zeit und Geld zur Implementierung.

Umfangreiche infrastrukturelle Verbesserungen müssen durchgeführt werden, um Continuous Integration / Continuous Deployment Praktiken zu unterstützen. Teammitglieder benötigen Schulungen und die Tools / Technologien müssen auf den neusten Stand gebracht werden, um CI/CD effektiv umsetzen zu können.

Ein weiteres großes Problem mit Projekten, welche nicht mit Testfällen erstellt wurden, ist, dass nicht darauf geachtet wird, wie bestimmte Entscheidungen den Prozess der Implementation von Testfällen erschweren. Bei einer entwicklungsbegleitenden Testerstellung fällt schnell auf, wenn eine Änderung problematisch wird. Diese kann somit frühzeitig zurückgerollt und mit einer nachhaltigeren Lösung ersetzt werden. Ein System ohne Testfälle kann weniger flexibel angepasst werden. Dies ist der Fall da bei der Entwicklung von Tests sich stark an die Anwendung angepasst wird, da Features meist schon feststehen und ohne großen Zeitaufwand nicht mehr umstrukturiert werden können.

So ist beispielweise die Architektur der Anwendung festgelegt und es existiert keine geeignete Möglichkeit extern Anfragen an die Datenbank zu senden, da kein explizites „Application Programming Interface“ (API) zwischen Programm und Datenbank vorhanden ist, das genutzt werden könnte. Für eine simple Anwendung ohne weitere Abhängigkeiten ist das legitim, aber für große Enterprise-Anwendung ist das problematisch. Es gibt entweder keine Möglichkeit, außerhalb des Programms mit den Daten zu arbeiten oder die Datenbank muss direkt angesprochen werden.

Probleme mit Datenbankintegrität sowie inkonsistente und fragile Tests würden nicht existieren, wenn eine API in den Tests aufgerufen werden könnte. Testdaten können hier Abhilfe schaffen, wenn die Datenbankarchitektur determiniert wurde. Ansonsten führen sie jedoch zu vielen Problemen wie bspw. Fehlalarme, da sie manuell für jede Datenbankmodelländerung angepasst werden müssen, um konform mit dem neuen Modell zu sein.

Um diese Herausforderungen zu vermeiden, wurden bestimmte Maßnahmen bei der Erstellung der Softwarearchitektur getroffen. Wie diese Architektur aufgebaut ist, wird vereinfacht im nachfolgenden Unterpunkt beschrieben.

2.1.2 Telekom Software Implementierung

Die Implementierung von Continuous Integration bzw. Software-Testing wurde anhand einer Telekom-internen Anwendung durchgeführt. Hierbei handelt es sich um eine auf Django (Python) basierte Webanwendung zur Suche bzw. Sammlung Telekom-zertifizierter Geräte. Diese Anwendung wird seit Anfang 2021 mithilfe von Continuous Delivery automatisch auf einer Produktivumgebung eingesetzt und auf Fehler und Defekte überprüft. Ohne Softwaretests besteht hier ein großes Risiko, dass bei einem Deployment das gesamte System abstürzt. Um dieses Szenario zu vermeiden, wurden mehrere Betriebsumgebungen erstellt, auf denen Änderungen manuell überprüft werden können. Dadurch können einzelne Softwareversionen überprüft werden, ohne die Produktivumgebung einem unnötigen Risiko auszusetzen. Die Erstellung mehrerer Umgebungen kann jedoch häufig zu Problemen führen, da Unterschiede in der Einrichtung zu unsteten Fehlern führen können. Beispielweise müssen aus Datenschutzgründen alle personenbezogenen Daten auf Testumgebungen entfernt bzw. mit Pseudodaten ersetzt werden. Dieser Unterschied in Daten kann daraufhin zur Folge haben, dass auf Testumgebungen kein Fehler entdeckt wird, aber auf Produktivumgebungen ein Fehler durch einen ganz bestimmten Datensatz zustande kommt.

Dieses Problem wurde mithilfe von Docker beseitigt. Jede Umgebung kann mithilfe von Containerisierung vollständig voneinander isoliert, auf die exakt gleiche Weise erstellt und betrieben werden. Wie in der folgenden Abbildung 2 zu erkennen ist wurde das Systemdesign in drei identische Branches unterteilt. Diese stehen in Abhängigkeit zu den Branches in GitLab. Zusätzlich zu den Services wird „Ofelia“ als Docker-basierte Cron-Engine und „Adminer“ als Web-Database-Viewer genutzt. Ofelia ermöglicht es automatisch in bestimmten Zeitintervallen Aktionen auszuführen wie bspw. das Erstellen von Datenbankbackups, währen Adminer es erleichtert mit der Datenbank zu interagieren.

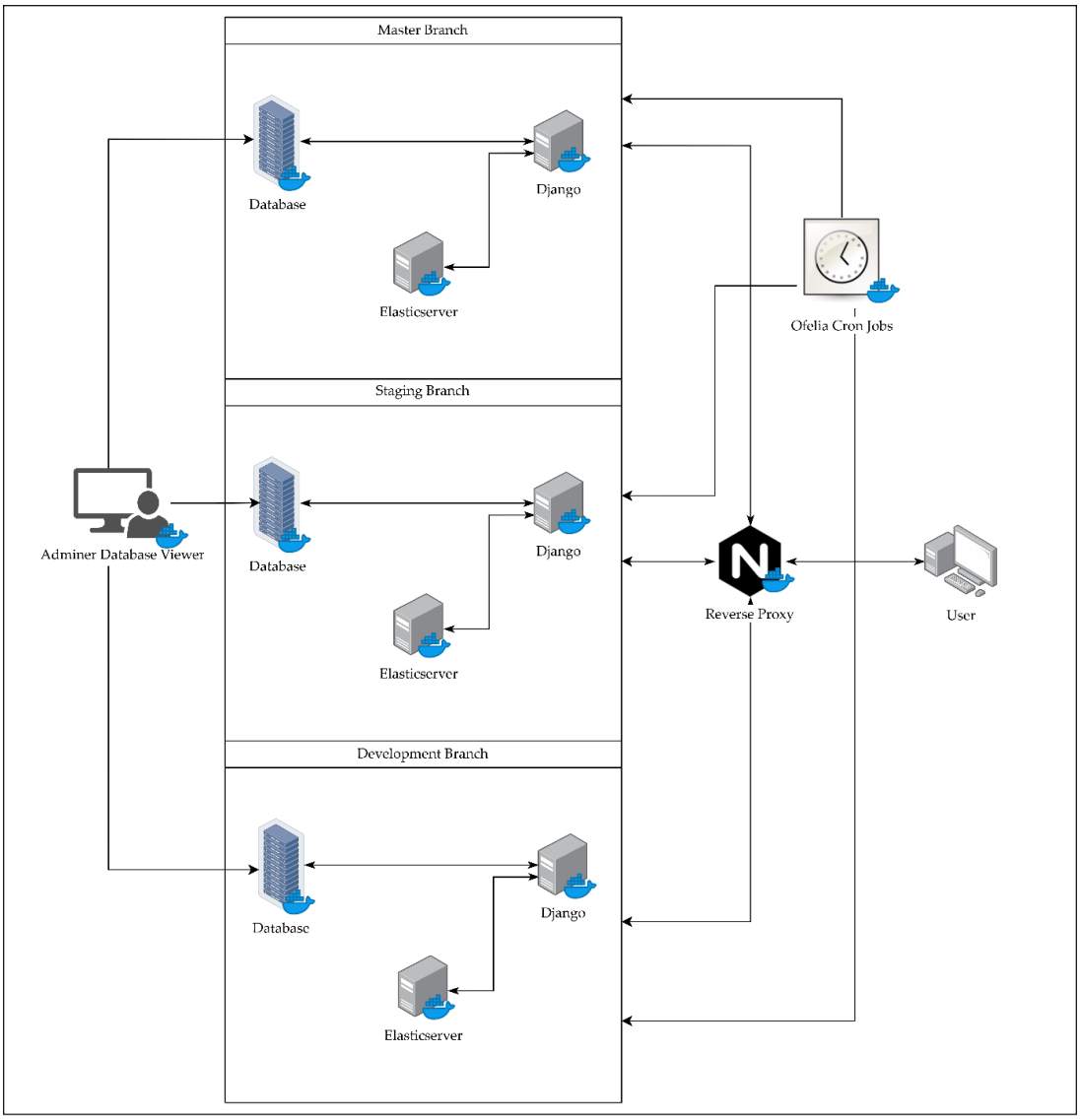

Abbildung 2: Vereinfachte Anwendungsinfrastruktur

Von dieser Architektur ausgehend wurde Continuous Integration beziehungsweise Softwaretests implementiert, um die Stabilität und Zuverlässigkeit der Produktivumgebung zu erhöhen. Wie in der Abbildung 2 zu erkennen, werden eine Vielzahl an Services betrieben. Einen neuen Softwarerelease manuell zu veröffentlichen, ist zeitintensiv und fehleranfällig. Mithilfe von Continuous Delivery beziehungsweise Continuous Deployment wird dieser Schritt automatisiert. CI/CD ist integraler Bestandteil in der Automatisierung von Deployment-Prozessen und daher auch kritisch in der Umsetzung von DevOps-Prinzipien. Worum es sich bei Continuous Delivery / Deployment handelt, wird anknüpfend erläutert.

2.2 Continuous Delivery

Continuous Delivery (CDE) ist eine Software Engineering Methode, die die Softwareentwicklungsdauer reduzieren soll. Hierbei werden Softwarereleases zuverlässig und automatisch auf eine produktions-ähnliche Umgebung veröffentlicht.

Um die Zuverlässigkeit zu gewährleisten, wird Continuous Delivery meist zusammen mit CI betrieben [10, S. 3910–3911]. CDE und DevOps sind sehr ähnlich in ihrer Bedeutung und streben eine schnellere und häufigere Ausführung von Prozessen an. Continuous Delivery erreicht dies durch die Automatisierung von Deployment-Prozessen. Gekoppelt mit CI entsteht eine Pipeline die sehr häufig, schnell und robust Software baut, testet und veröffentlicht.

2.3 Continuous Deployment

Continuous Deployment (CD) erweitert CDE um das automatische Deployment auf Produktionsumgebungen. Continuous Delivery setzt einen letzten manuellen Schritt für das Deployment auf Produktionsumgebungen voraus. Continuous Deployment dagegen automatisiert diesen Schritt zusätzlich. Im Bereich der Microservices und verteilten Anwendungen dient dieser Prozess dazu schnell und sicher Updates in das System einzuspielen [11], [12]. Zusammen mit Continuous Integration bildet die Continuous Integration / Continuous Deployment Chain (CI/CD-Chain) ein robustes System zur Automatisierung von häufig wiederkehrenden Aufgaben im Testing und Deployment von Anwendungen. Für die Umsetzung von DevOps sind diese Prinzipien von höchster Bedeutung. In welchen Zusammenhang sie zu DevOps stehen beziehungsweise was der Zweck von DevOps ist, wird im folgenden Unterpunkt näher betrachtet.

2.4 DevOps

Development & Operations (DevOps) beschreibt einen Ansatz in dem Softwareentwicklung und IT-Operations miteinander verknüpft werden. Das bedeutet, eine Verbindung zwischen Entwicklung und Betrieb von Software zu realisieren. Im Unterschied zu traditionellen Verfahren, arbeiten Softwareentwickler und Betriebsexperten zusammen an einem Projekt. Crossfunktionale Teams werden dadurch immer bedeutender, da ein intensiver Wissensaustausch innerhalb eines Teams sehr wichtig ist.

DevOps-Praktiken nehmen im gesamten Softwareentwicklungsprozess Einfluss auf die Entwickler laut Zhu et al. [13]:

- Softwareentwickler müssen verifizieren, dass die entwickelte Software die Systemstandards für den späteren Betrieb erfüllt
- Ein Entwickler kann ohne Absprachen mit anderen Entwicklungsteams neue Funktionalitäten zum Livesystem hinzufügen
- Änderungen durchlaufen mehrere Umgebungen, bevor diese zu Produktivsystemen hinzugefügt werden. Dadurch können schwerwiegende Fehler & Fehlfunktionen überprüft und getestet werden, bevor es zu Schäden auf Livesystemen kommt
- Systeme werden nach dem Deployment überwacht und können im Falle eines Fehlers auf einen vorherigen Stand zurückgebracht werden

Das Ziel von DevOps ist es, den Softwareentwicklungszyklus zu verkürzen. DevOps ist komplementär zur agilen Softwareentwicklung und setzt auf ein hohes Level an Kollaboration zwischen Entwicklungs- und Operationspersonal [12, S. 4].

Um diese Praktiken umsetzen zu können, werden flexible und portable Anwendungen benötigt. Anforderungen, wie das Durchlaufen mehrerer Umgebungen, bevor Änderungen auf Produktivsysteme hinzugefügt werden, sind ideale Anwendungsfälle für Containerisierung. Die Umsetzung von CI/CD wird bedeutsam vereinfacht durch die Nutzung von Containerisierungstechnologien wie beispielweise: „Docker“. Wie diese funktionieren und sich auf CI/CD auswirken wird im Folgenden beschrieben.

2.5 Containerisierung

Eine Kerntechnologie, die zur Umsetzung dieser Arbeit genutzt wurde, ist Docker. Hierbei handelt es sich um eine auf Betriebssystem (BS) -Ebene operierende Virtualisierungssoftware die eine Paravirtualisierung betreibt, d.h. ein Kern-BS abstrahiert Hardwareressourcen für mehrere Gastumgebungen, ohne sie speziell für jede Gastumgebung einzeln zu virtualisieren [14, S. 68]. Mithilfe von Docker ist es möglich, unabhängig vom genutzten Host-BS, Anwendungen zu betreiben. Docker nutzt zum Erstellen der Container-Images sogenannte „Dockerfiles“. Ein Container beschreibt hierbei eine Laufende Instanz eines Images. Dockerfiles sind Textdokumente, die alle Anweisungen enthalten, welche ein Nutzer in einer Kommandozeile zum Erstellen von Images ausführen kann. Ein Image ist eine vorgefertigte Betriebsumgebung, die alle benötigten Abhängigkeiten der bestimmten Softwareversion enthält. Dadurch wird sichergestellt, dass ein Softwarerelease immer eine korrekte dazugehörige Konfiguration besitzt. Zur Automatisierung mit CI/CD ist dies vorteilhaft, da Images bei jedem Codeupdate erstellt, gespeichert und getestet werden können.

Ein beispielhaftes Szenario zeigt eine weitere Stärke von Containern mit CI/CD. Es wird angenommen, dass ein manuell verwaltetes System ein neues Update erhält. Änderungen werden in das Livesystem übernommen, daraufhin tritt ein kritischer Fehler auf und das System ist nicht mehr im Betrieb.

Ein normaler Ablauf in solch einem System wäre:

1. Störungsmeldung wird an Kollegen geschickt
2. Repository muss zurückgerollt werden
3. Rollback-Commit muss von zuständiger Person genehmigt werden
4. Die Daten müssen auf den Server kopiert werden
5. Services werden manuell neu gestartet

Ohne eine Virtualisierung mit Docker ist ein Livesystem abhängig vom Stand des Master-Branches eines Repositories. Der letzte funktionsfähige Stand muss wieder hergestellt werden. Diese Behebung des Problems erfordert einen hohen Zeitaufwand und somit größere Wartungsfenster.

Dasselbe Szenario mit Docker wäre:

1. Container weist eine Störung auf und wird gestoppt
2. Es wird versucht den Container neu zu starten
3. Fehler tritt weiterhin auf
4. Vorheriges Docker Image wird genutzt

Docker macht es einfach zu erkennen, ob ein Service in Betrieb ist. Tritt ein Fehler auf, wird versucht den Container erneut zu starten. Tritt der Fehler weiterhin auf, kann ein vorheriges Docker-Image von der Container-Registry geladen werden, ohne das Repository bearbeiten zu müssen.

Laut Bosshard [15] werden so folgende Probleme behoben:

1. *Mangel an Konsistenz:* Enterprise-Anwendungen werden meist in Teams entwickelt. Es ist wahrscheinlich, dass Teammitglieder verschiedene Betriebssysteme nutzen oder ihre Maschinen unterschiedlich konfiguriert sind. Das bedeutet, dass sich die lokalen Entwicklungsumgebungen der Teammitglieder voneinander und daher auch von der Live-Umgebung unterscheiden. Es ist daher möglich, dass alle Softwaretests lokal erfolgreich sind, aber auf der Produktivumgebung fehlschlagen.
2. *Hoher Zeitanspruch & Fehleranfälligkeit:* Ohne automatisierte Deployments müssen jedes Mal, wenn eine neue Umgebung oder die gleiche Umgebung an mehreren Standorten bzw. Servern eingerichtet werden soll, manuell die exakt gleichen Konfigurationsschritte befolgt werden und die benötigten Pakete installiert werden. Dieser Prozess kann oft mehrere Stunden dauern und ist fehleranfällig, da er immer von Menschen manuell ausgeführt wird und dadurch Fehler entstehen können. Diese Probleme skalieren mit der Komplexität der Applikation. Kleine Projekte mit minimaler Konfiguration sind auf diese Weise verwaltbar, große Applikationen bestehend aus vielen Microservices jedoch nicht.

3. *Riskante Deployments:* Da Konfiguration-, Update- und Build-Prozesse nur zur Zeit des Deployments stattfinden, erhöht sich das Risiko, das ein Fehler, während eines Deployments auftritt. Rollbacks auf vorherige Versionen garantieren zudem nicht, dass das System wieder perfekt auf dem vorherigen Stand ist. Versionsänderungen und Updates von externen Softwarebibliotheken werden nicht automatisch mit einem Repository-Rollback zurückgenommen und müssen stattdessen manuell durchgeführt werden. Docker behebt dieses Problem mithilfe von Images.
4. *Schwierige Maintainability:* Updates der Applikation müssen manuell in einen Server eingespielt und angewandt werden, was erneut Zeitintensiv und Fehleranfällig ist.
5. *Downtime:* Eine Applikation auf einen einzelnen Server zu deployen stellt einen Single point of failure (SPOF) dar. Muss eine Applikation durch ein Update neugestartet werden, würde sie in dieser Zeit nicht verfügbar sein. Es ist einfacher und sicherer mehrere Server automatisch zu deployen. Die Portabilität und Flexibilität von Docker-Images erleichtert solche Konfigurationen.

3 Methodik

Zur Bearbeitung der genannten Zielsetzung wurde in Form einer Vorstudie, qualitativ die Umsetzung von CI/CD an einem existierenden Projekt durchgeführt. Hierbei wurden mögliche Probleme, Übereinstimmungen und Abweichungen zur Literatur ermittelt. Dementsprechend wurde zur Analyse des Falls und der Forschungsfragen deduktiv vorgegangen.

Um eine Einsicht in die CI/CD-Thematik zu erhalten, wurde mithilfe von „IEEE Xplore“ und anderen einschlägigen Datenbanken, wissenschaftliche Quellen zu den Stichworten „CI/CD“, „Continuous Integration“, „Software Testing“, „DevOps“ und verwandten Begriffen recherchiert. Im Fokus standen Möglichkeiten, Schwierigkeiten, Vorteile und Nachteile verschiedener Prozesse der Softwareentwicklung.

Mithilfe von „Backward reference searching“ auch bekannt als „Chain searching“ wurden weitere Quellen gefunden und für diese Arbeit genutzt. „Backward reference searching“ ist ein Vorgang in dem weiterführende Suchen nach relevanter Literatur anhand von zitierten Quellen eines wissenschaftlichen Werkes getätigt werden [16, S. XVI].

Da das Thema hochaktuell und im ständigen Wandel ist, wurde versucht, hauptsächlich aktuelle Publikationen einzubeziehen. Verschiedene Softwaredokumentationen, Blogartikel, Forum-Posts und weitere Quellenarten wurden in der Entwicklung des Projektes genutzt. Im Kontext der Literaturrecherche wurden diese jedoch nicht in der Analyse berücksichtigt, da sie lediglich die Entwicklung des Projektes unterstützten.

Der methodische Ablauf richtet sich dabei nach der Empfehlung von Yin [17] und beinhaltet drei Stufen:

1. Fallstudie entwerfen
2. Fallstudie durchführen
3. Fallstudie analysieren

Diese Stufen werden im folgenden Teil näher erläutert.

3.1 Entwurf

Die erste Stufe des methodischen Ablaufes befasst sich mit der Definition des zu untersuchenden Falles. Hierbei werden Hypothesen, Theorien und relevante Probleme gefunden und aufgestellt [17, S. 58]. Dabei wird, mit den vorher genannten Mitteln, Literatur gesammelt, um bekannte Theorien und Probleme aus den Bereichen der Continuous Integration zu erhalten und als Basis der späteren Evaluation zu nutzen.

Hinzu kommt die Auswahl der Kriterien [17, S. 60–61], mit denen die Auswertung und der Vergleich durchgeführt werden. Da es sich um eine Fallstudie mit Samplegröße eins handelt, können Kriterien wie „Umsetzungsdauer“ und „Effizienz“ nur bedingt genutzt werden, welche sich zwischen Projekten stark unterscheiden können. Die Anforderung aus dem Betrieb war eine Implementierung von Tests, um automatische Deployments abzusichern und ein Upload von defekten Builds zu verhindern. Bei der Auswahl von Kriterien wird versucht, diesen ursprünglichen Zweck der Implementierung im Betrieb zu berücksichtigen. Ein großes Augenmerk wird daher auf die Anzahl kritischer Fehler in der Produktivumgebung gelegt. Dies kann objektiv anhand von Pipeline-Logs und Jira-Tickets zu kritischen Problemen gesammelt, protokolliert und bewertet werden.

3.2 Durchführung

Auf Basis der genannten Schritte im Entwurf und der gesammelten Literatur wird anhand der „Telekom IoT Hardware Datenbank“ die Fallstudie durchgeführt. Die einzelnen Durchführungsschritte werden detailliert dokumentiert und die daraus resultierenden Ansätze mit der Literatur abgeglichen. Bei der Durchführung wird sich auf zwei Ansätze beschränkt, um den Horizont der Arbeit zu begrenzen. Der Fokus der Arbeit liegt hierbei auf den Ansätzen der „Top-down Integration“ und der „Bottom-up Integration“ gelegt [18, S. 372]. Diese Ansätze wurden gewählt, da sie sehr allgemeingültig und zueinander exakte Gegensätze sind und daher sehr gut im Vergleich der gewählten Entwurfskriterien genutzt werden können. Zudem können dadurch trotz der Beobachtung eines einzigen Projektes, mehrere Sichtweisen auf die Problematik gefunden werden.

Die Daten zur Analyse der Effektivität der Fallstudie werden mithilfe von GitLab CI/CD-Pipelines und Jira-Tickets erfasst. Die Pipelines geben dabei einen Einblick in die Anzahl erfolgreicher und abgebrochener Deployments, während die Jira-Tickets aussagen, wie viele Fehler letztendlich auf Produktivumgebungen erfasst wurden. Speziell bei Jira-Tickets wird sich auf Tickets mit den Typen „Bug“ oder „Impediment“ und einer Priorität von „hoch“ oder „sehr hoch“ bezogen. Mit diesen Daten lässt sich approximieren, wie fehleranfällig das System mit einem CI/CD Workflow ist.

3.2.1 Testumgebung

Wie durch das vorherige Kapitel thematisiert, wurde das Projekt mit Hilfe des Django-Frameworks für Python entwickelt. Dieses Framework bietet eingebaute Testing-Module, die für die Zwecke dieses Projektes genutzt wurden.

Eine große Schwierigkeit bei der Einrichtung einer automatisierten Testumgebung ist der Bedarf einer Datenbank. Viele Prozesse benötigen einen Austausch mit der Datenbank, um ein korrektes Verhalten zu testen. Bei der Automatisierung von Tests wird es jedoch schwierig, eine identische Datenbankinstanz für eine Liveumgebung bereitzustellen und als Mock-Datenbank einzurichten. Es ist einfacher, die Datenbank des Livesystems direkt zu erreichen und keine Mock-Datenbank zu erstellen. Dies ist jedoch mit Blick auf Datenschutz und Testfragilität suboptimal. Speziell Datenschutz ist immer ein wichtiger Fall, da die meisten Anwendungen personenbezogene Daten in Form von Accounts der Nutzer speichern.

Die zweite Möglichkeit besteht darin, einen Dump der Datenbank zu erstellen und diese als Fixture in die Testdatenbank zu laden.

```
{
  "model": "contact.company",
  "pk": 160,
  "fields": {
    "created_date": "2020-01-13T08:05:46.328Z",
    "modified_date": "2021-01-25T10:56:05.292Z",
    "created_by": 40,
    "modified_by": 37,
    "name": "Deutsche Telekom AG",
    "preferred_partner": false,
    "contract_level": 0,
    "contract_date": null,
    "contract_status": 0,
    "website": "https://www.telekom.de/start",
    "order_email": "",
    "short_name": "",
    "contact_page": ""
  }
},
```


Abbildung 3: Fixture Ausschnitt

Abbildung 3 zeigt einen Ausschnitt solch einer Fixture, im Datenbankmodell der IoT Hardware Datenbank. Diese Fixture-Daten wurden aus einer SQL-Datenbank exportiert und liegen im JSON-Format vor. Eine problemfreie Handhabung dieser Daten wird vor allem durch die enorme Größe der Datenbank erschwert. Eine kleine Datenbank hingegen kann vollständig repliziert werden und separat für jeden Testfall neu geladen werden. Je größer die Datenbank wird, desto weniger realisierbar wird dieser Ansatz. Daher muss ein Subset der Datenbank erstellt werden, welches zum Testen genutzt wird. Hier stellt sich die Frage: Welches Subset der kompletten Datenbank soll für Testzwecke genutzt werden? Es kann nicht im Vorhinein erkannt werden, welche Datensätze problematisch sein könnten. Wird nur ein Teil aller Daten verwendet, besteht das Risiko, dass Tests erfolgreich durchlaufen, die Produktivumgebung aber nach dem Update abstürzt, weil ein bestimmter nichtgetesteter Datensatz einen Fehler hervorruft. Im Rahmen dieser Arbeit wurde ein angepasster Datenbank-Dump für die Erstellung der Fixtures genutzt. Der Fokus der Anpassung war die Reduzierung von Dopplungen. Beispielweise liegt in Abbildung 3 ein Ausschnitt eines Company Eintrags vor. Einträge dieser Art werden genutzt, um einzelnen Produkten das entsprechende Unternehmen zuzuweisen und den Status der Telekom mit diesem Unternehmen zu protokollieren. Viele Einträge unterscheiden sich an dieser Stelle lediglich im Namen und Erstellungsdatum. Dopplungen dieser Art wurden entfernt, da die besprochenen Attribute nicht weiter evaluiert werden. Weitere Maßnahmen zur Erstellung der Fixtures werden im folgenden Kapitel mitunter beschrieben.

3.2.2 Datenbankskalierung

Schwierigkeiten bezüglich der Skalierung von Anwendungen, können während der Entwicklung sehr schnell vernachlässigt werden, da Entwicklungsumgebungen typischerweise nur wenige Datensätze besitzen. Bei der Implementation von Tests werden Probleme mit der Datenbankstruktur sehr auffällig, da es vorteilhaft ist, Mock-Datenbanken für die Tests zu nutzen. Eine schlechte Datenbankstruktur erschwert das manuelle Erstellen von Datensätzen und kann den Speicheraufwand der Datenbank exponentiell wachsen lassen. Komplexe Datenstrukturen erschweren daher die Umsetzung von Testfällen [19, S. 28].

Die Datenbank der genutzten Anwendung besitzt eine Tabelle, welche alle Werte verschiedener Attribute von verschiedenen IoT-Geräten speichert. Es existieren insgesamt 285 Geräte mit insgesamt 51.024 Attributwerten. Das bedeutet, dass im Durchschnitt jedes Gerät 179 verschiedene Attribute besitzt. Attribute können hier jegliche Informationen wie bspw. der Name, Verbindungsmöglichkeiten, Wasserdichte und Betriebstemperatur sein. Dadurch entsteht eine große Menge an Datensätzen. Dies ist an sich noch kein Problem, da es lediglich eine hohe Fülle an Informationen für jedes Gerät gibt. Ein hoher Informationsgehalt liegt hier jedoch nicht vor. Beim Filtern nach den Strings „N/A“ (Not Available) und „ “ (leer) fällt auf, dass 38.822 der 51.024 Werte (76,09%) irrelevante Werte speichern. Insbesondere „N/A“ und Leere Strings sind dafür geeignet durch das Frontend automatisch befüllt zu werden, anstatt sie in der Datenbank zu speichern. Bei dieser Suche hat sich herausgestellt, dass sich ein Großteil der Werte häufig wiederholt. Bei einer Suche nach allen distinkten Strings wurde herausgefunden, dass nur 1.730 Werte einzigartig sind. Folglich sind 96,99% aller Werte irrelevant. Bei der Erstellung eines angepassten Datenbankdumps konnten daher eine große Menge an Werten verworfen werden.

Insgesamt ist die Datenbank mit 285 Geräten 24,41 MB groß. Allein durch das Entfernen von „N/A“ und leeren Strings reduziert sich die Größe auf 5,84 MB. Wird dieser Wert nun auf beispielweise 2000 Geräte skaliert bedeutet das, dass die Datenbank 171,3 MB nicht optimiert und 40,96 MB optimiert groß wäre. Das Datenbankmanagementsystem (DBMS) wird dadurch nicht beeinträchtigt, da es dafür optimiert mit hoher Geschwindigkeit, eine große Anzahl an Daten zu verarbeiten. Eine Webanwendung ist das jedoch nicht. Anstatt programmatisch auf der Clientseite Standard-Werte zu setzten, müssen diese separat verschickt werden und verbrauchen mehr Bandbreite, wodurch die Anwendung nachhaltig verlangsamt wird.

Eine performante Ethernet-Verbindung benötigt im schlechtesten Fall neun Sekunden, um die Webanwendung vollständig zu laden und verbraucht dabei 6,81 MB an Daten. Dies führt auch zu Problemen beim Testing, da die Datenbank für jeden Test neu aufgesetzt wird, um sicherzustellen, dass immer derselbe Ausgangspunkt vorliegt. Wird nun dieser vollständige Dump genutzt, so verlängert sich die Testdauer auf mehr als eine Stunde.

Das große Problem mit der nachträglichen Entwicklung von Testfällen ist, dass viele Verfahrensweisen so weit in der Entwicklung sind, dass sie nicht mehr geändert werden können, da viele Komponenten abhängig von anderen sind und eine Änderung konform mit anderen Komponenten sein muss. Das Musterbeispiel ist hier der Datenbankaufbau. Wie bereits erwähnt gibt es viele Probleme mit der vorhandenen Datenbank, die bei der Implementierung von Tests zu Tage traten. Diese Probleme sind allerdings kaum behebbar, da bei der Fülle von Änderungen es effektiver wäre neu zu beginnen. Ein weiterer problematischer Umstand kommt zum Tragen, wenn eines der bisher beschriebenen Probleme nicht durch eigens entwickelte Software verursacht wird. In einem solchen Fall müsste der ursprüngliche Entwickler kontaktiert werden, was einem erheblichen Mehraufwand gleichkommt.

3.2.3 End-to-End Tests

Anstatt viele kleine Unit Tests zu schreiben, kann auf holistische Weise die gesamte Anwendung auf ihre allgemeine Funktionalität getestet werden. End-to-End (E2E) Tests nutzen dafür einen Browser und führen Anweisungen ähnlich wie ein Mensch aus. Dadurch kann getestet werden, ob das gesamte System wie erwartet funktioniert. So können sehr schnell grobe Tests erstellt werden, die überprüfen ob alle Teile der Anwendung navigierbar / funktionsfähig sind. Die Granularität der Tests kann dabei allmählich erhöht werden.

Diese Art von Tests ist jedoch sehr hardwarelastig. Für den Kontext: Bei einem Laptop ausgestattet mit einem Intel i5 8350U 4 Kern Prozessor, wurde der Prozessor zu 100% ausgelastet und erreichte eine Höchsttemperatur von 96°C. Die Ausführung der kompletten Testsuite konnte hierbei bis zu vier Minuten dauern.

Hochperformante GitLab-Runner mit acht Kernen benötigen dagegen nur circa 90 Sekunden. Es wird gezeigt wie abhängig die Ausführungszeit von Hardwareperformance ist. Bei einer Wartezeit von vier Minuten ist es sinnvoll die Tests vor einem Push auszuführen, da es zu lange dauert, um zwischen jeder Änderung zu testen.

Um mit einem Browser automatisiert interagieren zu können, wird die Softwaresuite „Selenium“ genutzt. Diese erlaubt es mithilfe von sehr einfachem Code, automatisiert Aktionen auf einer Website zu tätigen. Hierbei soll der durchschnittliche Nutzer der Anwendung und sein Interagieren mit der Anwendung emuliert werden.

In einem Zeitraum von jeweils einer Arbeitswoche wurde das Projekt für automatisiertes Testen eingerichtet und Testfälle erstellt. Ein großer Faktor bei der Implementierung von E2E-Testfällen im Vergleich zu Unit Tests ist der Einrichtungsaufwand. Selenium als Tool benötigt einen Browser. Um diesen Browser steuern zu können, wird zusätzlich ein Driver benötigt, der als Schnittstelle zwischen Anwendung und Browser fungiert. Um Tests mit Selenium automatisch durchzuführen, müssen Browser und Treiber in der Testumgebung des CI/CD-Prozesses installiert werden und eine Netzwerkverbindung zwischen ihnen eingerichtet werden. Diese Änderungen an der CI/CD-Konfiguration sind sehr zeitintensiv, da vorerst ein Runner bzw. eine virtuelle Maschine bereitgestellt werden muss, die den gewünschten Job ausführt. Dieser Prozess hat im Falle dieses Projektes zwei der fünf Tage für die Implementierung benötigt.

In den restlichen drei Tagen wurden insgesamt neun Testfälle für E2E-Testing implementiert. Diese Tests sind in der Lage, großflächige Teile der Anwendung, welche zu jeder Zeit funktionieren müssen, abzudecken.

```
def test_DevicePageLoadsCorrectly(self):
    driver = self.driver
    # Choose your url to visit
    driver.get(self.live_server_url + '/device/standard/')
    assert 'Digi Transport Router WR11' in driver.page_source
```

Abbildung 4: E2E Test Auszug

Die Abbildung 4 illustriert einen der genutzten Testfälle. Hier wird auf eine bestimmte Route einer Website navigiert und im gesamten Quellcode nach einem bestimmten String gesucht. In diesem Fall wird nach dem Produktnamen „Digi Transport Router WR11“ gesucht. Wenn dieser String erfolgreich gefunden werden kann, dann bedeutet das, dass die Webanwendung in Betrieb ist, die Datenbank korrekt abgefragt wird und beim Nutzer die Daten abgebildet werden. Mit nur drei Zeilen Code werden hier große Teile der gesamten Anwendung auf Funktionalität überprüft. Dem Zweck der Anwendung geschuldet, sammeln sich eine große Anzahl an Funktionen an zentralen Stellen in der Anwendung, wodurch E2E noch effektiver eingesetzt werden kann.

3.2.4 Unit Tests

Im Gegensatz zum E2E-Testing wurden 23 Unit Tests im Bearbeitungszeitraum erstellt. Die höhere Anzahl an Testfällen ist auf einen geringen Einrichtungsaufwand zurückzuführen. Hier auftretende Schwierigkeiten bei der Implementierung von mehr Testfällen, war die in Kapiteln 3.2.2 und 3.2.3 genannte Komplexität der Datensätze und Größe der Datenbank. Als Entwickler ist zudem die Einschätzung des Zwecks und der Aufbau unbekannter bzw. fremder Funktionen schwierig nachzuvollziehen. Da ein hoher Grad an Verständnis einzelner Funktionalitäten benötigt wird, gestaltete sich das Erstellen sinnvoller Testfälle als eine Herausforderung. Ein weiteres Problem ist die Anzahl an Tests, die benötigt werden, um ein hohes Maß an Konfidenz in die Fehlerfreiheit der Anwendung zu erreichen. Das Verhältnis der Gesamtheit an Unit Tests zu der gesamten Anzahl an Funktionen / Methoden sollte mindestens 20% oder größer sein [20, S. 113]. Idealerweise sollte dieses Verhältnis jedoch 1:1 sein [21, S. 25]. Ein 1:1 Verhältnis bedeutet, dass linear wachsend mit der Anwendung Tests erstellt werden müssen, um dem Idealfall zu entsprechen. Zudem wird hierbei mehr Zeit benötigt, um die Testfälle zu pflegen. Mit sich ändernden Anforderungen müssen sowohl die eigentlichen Funktionen als auch die Testfälle angepasst werden, um den Änderungen gerecht zu werden.

```
def test_create_group_no_parent(self):
    """
    this test makes sure the validation error is not
    thrown if parent is none
    """
    new_group = AttributeGroup.objects.create(name='name')
    self.assertEqual(new_group.name, 'name')
```

Abbildung 5: Unit Test Auszug

Die Abbildung 5 zeigt einen einfachen Unit Test indem mithilfe von Instanziierung einer Klasse getestet wird, ob für eine Instanz ohne die Variable „parent“ ein Validierungsfehler auftritt. Anhand dieses Beispiels ist erkennbar, wie lokalisiert Unit Tests genutzt werden. Ein Fehler in diesem Unit Test zeigt eindeutig, dass ein Fehler in der Klasse „AttributeGroup“ vorliegt.

Die vorliegenden Unterschiede zwischen E2E und Unit Tests bezüglich der Granularität, Effektivität, Wartbarkeit und Umsetzbarkeit werden im folgenden Kapitel mithilfe von gesammelten Daten aus sowohl dem zentralen GitLab-Repository als auch dem projektbegleitenden Jira-System betrachtet und ausgewertet.

3.3 Analyse

Zur Analyse der Fallstudie werden die genannten Forschungskriterien im Entwurf näher betrachtet und es wird versucht, mithilfe von Pattern Matching [17, S. 224] die auftretenden Beobachtungen zu erklären. Zusätzlich werden die erfassten Daten mithilfe von statistischen Tests analysiert um die Signifikanz der Messergebnisse ermitteln und auswerten zu können. Die mithilfe dieser Prinzipien erstellten Hypothesen, werden im letzten Teil der Fallstudie validiert bzw. widerlegt werden und als Vergleichsbasis mit der Literatur dienen.

3.3.1 Datensammlung

Die gesammelten GitLab- und Jira-Daten bieten jeweils einen sehr unterschiedlichen Aufbau und dienen sehr anderen Zwecken. Daher wird vorerst dieser Aufbau näher erläutert und sein Zweck beschrieben.

Sowohl die gesammelten GitLab-Pipeline-Logs als auch Jira-Tickets wurden von der Deutschen Telekom IoT GmbH zur Verfügung gestellt. Die Datensammlung der Pipeline-Logs begann im Januar 2021, während die Jira-Tickets seit Januar 2020 gesammelt wurden. Daten wurden nicht in einem festen Intervall gesammelt, sondern jedes Mal, wenn ein Event manuell ausgeführt wurde. Ein „Event“ bezieht sich hier auf Aktionen, wie Updates am Produktivsystem und manuell eingestellte Tickets.

GitLab-Pipeline-Logs sind in vier Schritte aufgeteilt, in dem das Projekt als Docker-Image verpackt, getestet, versioniert und auf der Produktivumgebung eingesetzt wird. Durch die Nutzung von Docker ist es möglich, eine identische Instanz der Anwendung, wie sie auf der Produktivumgebung betrieben wird, zu testen. Mithilfe von Docker wird das Vorkommen von Fehlern durch unterschiedliche Betriebsumgebungen vermieden. Für die Datensammlung relevant ist der Testing-Schritt, der direkt nach dem Erstellen eines Docker-Images durchgeführt wird. Da das Projekt Docker nutzt, wird in diesem Schritt zuerst das erstellte Docker-Image instanziiert.

Mithilfe der eingebauten Testfunktionalität des Django-Frameworks werden daraufhin die erstellten Tests durchgeführt.

Gesammelt werden diese Daten, um zu protokollieren, welche Anzahl an Fehlern verhindert wurden. Eine abgebrochene Pipeline ist in diesem Fall ein Erfolg für den jeweiligen Testfall, da Fehler von Produktivumgebungen ferngehalten werden konnten. Um nun jedoch herauszufinden, wie viele Fehler es unbemerkt in die Produktivumgebung geschafft haben, werden zusätzlich Jira-Tickets herangezogen. Hierbei werden lediglich Tickets vom Typ „Bug“ oder „Impediment“ mit einer Priorität von „hoch“ oder „sehr hoch“ betrachtet. Dadurch wird sich auf kritische Fehler, welche die gesamte Funktionalität der Anwendung beeinträchtigen, beschränkt. Eine Priorität von „hoch“ bzw. „sehr hoch“ muss nicht generell aussagen, dass ein kritischer Fehler vorliegt. Im Kontext dieses Projektes war dies jedoch der Fall.

Die genannten Datensätze wurden daraufhin händisch in einer Excel-Tabelle protokolliert. Hierbei wurden die Daten kategorisiert in erfolgreiche Runs und fehlgeschlagene Runs. Zusätzlich wurden die Datensätze in Zeitintervalle vor und nach der Implementierung von Continuous Integration gruppiert. Mithilfe dieser Daten wurden daraufhin Signifikanztests und Metriken zur Bewertung der Softwarequalität erstellt. Wie diese zustande gekommen sind, wird in den Punkten 3.3.2, 3.3.3 und 3.3.4 näher erläutert.

3.3.2 Analyse mithilfe von Signifikanztests

Statistiken können auf dem ersten Blick eindeutig und verlässlich erscheinen. Häufig ist es jedoch möglich, dass Ergebnisse aus Zufall entstanden sind und bei wiederholter Erstellung von Datensätzen ein anderes Ergebnis zustande kommen würde.

Signifikanztests sollen hier zeigen, mit welcher Wahrscheinlichkeit ein bestimmtes Ergebnis aus Zufall entstanden sein könnte. Desto geringer der Wert eines Signifikanztests, desto unwahrscheinlicher ist es, dass ein Ergebnis aus Zufall entstanden ist. Der Zielwert ist hier ein Chi-Quadrat bzw. exakter Test nach Fischer von $\alpha < 0.05$ [22, S. 1].

Daten hierzu wurden auf verschiedenen Ebenen gesammelt, wodurch bei der Analyse mithilfe von Signifikanztests zwischen sieben verschiedenen Ebenen unterschieden wird.

Tabelle 1: Gesammelte Signifikanzebenen und deren Bedeutung

Ebene	Bedeutung
Black-Box Development	Zeigt wie signifikant die Anzahl der gefundenen Fehler auf **Entwicklungsumgebungen**, im Vergleich zu vor der Implementierung von **Black-Box Tests**, gesunken bzw. gestiegen ist.
Black-Box Master	Zeigt wie signifikant die Anzahl der gefundenen Fehler auf **Produktivumgebung**, im Vergleich zu vor der Implementierung von **Black-Box Tests**, gesunken bzw. gestiegen ist.
White-Box Development	Zeigt wie signifikant die Anzahl der gefundenen Fehler auf **Entwicklungsumgebungen**, im Vergleich zu vor der Implementierung von **White-Box Tests**, gesunken bzw. gestiegen ist.
White-Box Master	Zeigt wie signifikant die Anzahl der gefundenen Fehler auf **Produktivumgebung**, im Vergleich zu vor der Implementierung von **White-Box Tests**, gesunken bzw. gestiegen ist.
White-Box vs. Black-Box	Zeigt wie signifikant **Black-Box-** und **White-Box-**Ansätze sich anhand gefundener Fehler **unterscheiden**.
Jira Development	Zeigt wie signifikant die Anzahl der **unentdeckten Fehler** auf **Entwicklungsumgebungen**, im Vergleich zu vor der Implementierung von Continuous Integration, gesunken bzw. gestiegen ist.
Jira Master	Zeigt wie signifikant die Anzahl der **unentdeckten Fehler** auf **Produktivumgebungen**, im Vergleich zu vor der Implementierung von Continuous Integration, gesunken bzw. gestiegen ist.

Die in Tabelle 1 beschriebenen Ebenen werden im Zusammenhang zueinander betrachtet, analysiert und bewertet. Mithilfe dieser Ebenen wird gezeigt, wie viele Fehler der jeweilige Testing-Ansatz abfängt und an welcher Stelle ein Fehler auffällt. Zudem wird die Anzahl der unentdeckten Fehler und die Effektivität der Ansätze untereinander bestimmt.

Ähnliche Signifikanzwerte zwischen diesen Ebenen sind jedoch verschieden interpretierbar. Ein signifikant geringerer Wert im Bereich Black-Box Master deutet beispielweise auf eine Verbesserung der Fehlerrate durch die erstellten Tests hin, während ein signifikant geringerer Wert im Bereich Black-Box Development auf eine generell geringere Anzahl an Fehlern hinweist. Diese können nicht auf Software-Testing zurückgeführt werden. Dies wird in der Auswertung der Ergebnisse näher betrachtet.

3.3.3 Analyse mithilfe von Softwarequalitätsmetriken

Eine zweite Art der Metriken, die in dieser Arbeit gesammelt und analysiert wurden, sind jene für die Bewertung der Softwarequalität. Es werden Metriken aus den Gruppen der Zeitanalyse, Reliability und Maintainability nach Empfehlung von Galin [23] berechnet und ausgewertet. Nachfolgend werden diese Metrikgruppen näher betrachtet.

Die Zeitanalyse als erste betrachtete Metrik ist die einfachste und dient der Aufgabe die Anzahl der auftretenden Fehler, unabhängig von der Länge des Ausfalls, in einem bestimmten Zeitintervall zu summieren. Als Ausgangslage wird hierfür die Anzahl an Fehler pro Jahr gesammelt. Dieser Wert kann dann bei Bedarf umgerechnet werden, um einen Überblick über das Risiko einer Anwendung in einem bestimmten Zeitraum zu erhalten. Eine Beispielrechnung hier wäre: Bei einer Anwendung treten 12 Fehler im Jahr auf. Das bedeutet die Anwendung hat 12 „Flaws“ pro Jahr. Ist die Anzahl der Flaws pro Monat benötigt, so wird dieser Wert durch 12 geteilt. Wird ein kleinerer Zeitraum betrachtet wird auf dieselbe Weise vorgegangen.

Die zweite betrachtete Gruppe ist die Reliability. In der Praxis ist sie eine der am häufigsten genutzten Metriken, besonders im Cloud-Bereich. Hier wird die eigentliche „Downtime“ (Ausfallzeit) einer Anwendung beachtet und daraufhin prozentual auf ein Jahr abgebildet.

Die folgende Formel beschreibt die „Full Reliability“ (Vollständige Zuverlässigkeit) FR einer Anwendung, wobei $NYSerH$ (Notated Yearly Service Hours) die Anzahl der Betriebsstunden im Jahr und $NYFH$ (Notated Yearly Failure Hours) die Anzahl der Stunden im Jahr in der mindestens eine Funktion fehlgeschlagen ist beschreibt.

$$FR = \frac{NYSerH - NYFH}{NYSerH} \quad [1.1]$$

Diese Funktion [1.1] repräsentiert den Zustand, in dem alle Softwaresysteme korrekt funktionieren und bildet einen Prozentanteil der Betriebszeit. Der Industriestandard von 99,99%+ Uptime [24] wird mithilfe der „Full Reliability“ bestimmt.

Der schwächere Grad der „Full Reliability“ ist die „Vital Reliability“ $VitR$. Hier wird lediglich auf die Funktionalität von kritischen Funktionen geachtet. Unkritische Funktionen dürfen ohne Einbußen in der Reliability fehlschlagen. Bestandteil dieser Formel sind daher erneut die $NYSerH$ und die $NYVitFH$ (Notated Yearly Vital Failure Hours), welche die Anzahl an Stunden pro Jahr, in der mindestens eine kritische Funktion fehlschlug, repräsentiert.

$$VitR = \frac{NYSerH - NYVitFH}{NYSerH} \quad [1.2]$$

Der resultierende Prozentsatz muss zudem größer als die „Full Reliability“ sein.

Die letzte betrachtete Formel der Reliability-Gruppe ist die „Total Unreliability“ TUR. Sie reflektiert, wie lange ein Softwaresystem einen kompletten Ausfall bzw. ein Fehlschlagen aller Funktionen erleidet. $NYTFH$ (Notated Yearly Total Failure Hours) repräsentiert hierbei die Anzahl aller totalen Ausfälle im Jahr.

$$TUR = \frac{NYTFH}{NYSerH} \quad [1.3]$$

Die „Total Unreliability“ muss zudem so gering sein, dass $1 - TUR > VitR$ ist. Insgesamt gilt daher: „Full Reliability“ < „Vital Reliability“ < 1 – „Total Unreliability“. Mithilfe dieser Metriken ist es nun möglich, die Konfidenz in das Produkt abzuschätzen und Wartungszeiten einzuplanen. Aufgrund dieser Metriken kann zudem besser geplant werden, wie viel mehr Zeit und Geld in die Verbesserung von Testfällen und Fehlerabsicherungen gesteckt werden muss.

Als letzte große Gruppe an Metriken wird die Maintainability (Wartbarkeit) der Anwendung geprüft. Hierbei wird ein Fokus auf Aspekte gelegt, die es vereinfachen,

Fehler innerhalb einer Anwendung zu analysieren und zu beheben. Als Metriken werden hier die „Diagnostic support capability“ (DSCp) und das „Change success ratio“ (CSR) genutzt. Die DSCp sagt aus, wie viele Fehler TNF von Diagnostic / Test Funktionen NFMDF gefunden wurden. Ein höheres Verhältnis bedeutet hier eine hohe Testabdeckung bzw. umfassende Fehlerabfangmechanismen.

$$DSCp = \frac{NFMDF}{TNF} \quad [1.4]$$

Zusätzlich zum DSCp wird das CSR herangezogen. Hierbei werden die Anzahl an Änderungsfehlern $NCnF$ (number of change failures) in ein Verhältnis zu der Gesamtheit aller Änderungen $TNCn$ (total number of changes) gebracht. Ein Änderungsfehler bedeutet, dass auf einer fehlerhaften Änderung trotz Nutzung und Hilfe von Diagnose-Funktionen eine weitere Änderung erneut fehlerhaft ist. Die Formel bildet sich daher wie folgt:

$$CSR = 1 - \frac{NCnF}{TNCn} \quad [1.5]$$

3.3.4 Analyse anhand der Forschungskriterien

Zuletzt wird die Analyse durch eine explizite Betrachtung der Forschungsfragen/-Kriterien erweitert. Die Frage: „Welche Ansatzweise eignet sich am besten zur schnellen und effektiven Abdeckung von Testfällen?“, soll durch die Kombination von Signifikanztests und Reliability-Metriken beantwortet werden. Durch diese Kombination kann die Effektivität der Umsetzung sehr gut beurteilt werden. Da die Umsetzungsdauer jeweils eine Woche betrug, kann von einer schnellen Implementierung bzw. Abdeckung gesprochen werden. Orientiert wurde sich bei dieser Zeitspanne an „Scrum Sprints“. Diese sind in der Regel ein bis zwei Wochen lang und nicht länger als vier Wochen [25]. Im Rahmen dieser Arbeit bedeutet „schnelle Implementierung“, innerhalb eines Sprints.

Ein wichtiger Faktor dieser Arbeit ist die Dichotomie zwischen dem theoretischen Ansatz und den existierenden Anwendungen. Sowohl der Arbeitstitel als auch die Forschungsfrage verlangen eine Auseinandersetzung mit verschiedensten theoretischen als auch praktischen Umsetzungen. Die Erkenntnisse aus diesen Quellen sollen folglich mit den Ergebnissen der praktischen Umsetzung des internen Projektes der Telekom verglichen werden.

4 Ergebnisse & Auswertung

In einer Zeitspanne von drei Monaten nach der Implementierung von Continuous Integration, wurden die vorher genannten Datensätze gesammelt. Insgesamt wurden in diesem Zeitraum 252 einzigartige Datensätze zusammengetragen. Vor der Implementierung von Continuous Integration wurden dagegen 381 Datensätze in einem Zeitraum von einem Jahr aggregiert.

Das folgende Diagramm zeigt die Aufteilung dieser Datensätze in fehlgeschlagene und erfolgreiche Durchläufe.

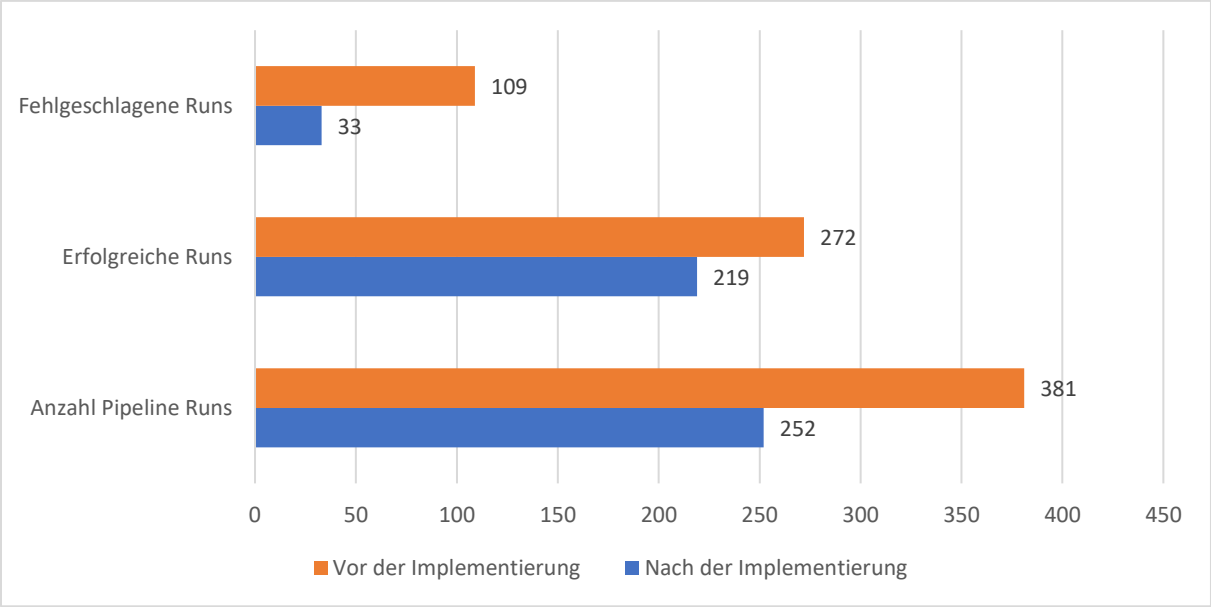

Abbildung 6: Anteil erfolgreicher und fehlgeschlagener Pipelines

Bei Betrachtung der Intervalle vor und nach der Implementierung liegen insgesamt 633 Datensätze vor. Davon waren 491 erfolgreich und 142 fehlgeschlagen. Auf den ersten Blick ist hier eine starke Verringerung an aufgetretenen Fehlern zu erkennen. In diesem Diagramm wird keine Rücksicht darauf genommen, weshalb eine GitLab-Pipeline fehlgeschlagen ist. Das bedeutet, dass Fehler die durch inkorrekte Pipeline-Konfigurationen, defekte Dockerfiles und fehlende Berechtigungen entstehen, ebenfalls protokolliert wurden.

Diese Art von Fehlern sind jedoch im Kontext von Continuous Integration irrelevant, da sie nicht aus der eigentlichen Anwendung hervorgehen, sondern aus den Rahmenbedingungen einer CI/CD-Pipeline. Fehlgeschlagene Runs, die nicht durch einen Testschritt fehlgeschlagen sind, werden daher ignoriert.

Auf unrelevante Datensätze bereinigt bleiben erwartungsgemäß keine fehlerhaften Runs vor der Implementierung übrig, da zu diesem Zeitpunkt noch keine Testfälle existierten.

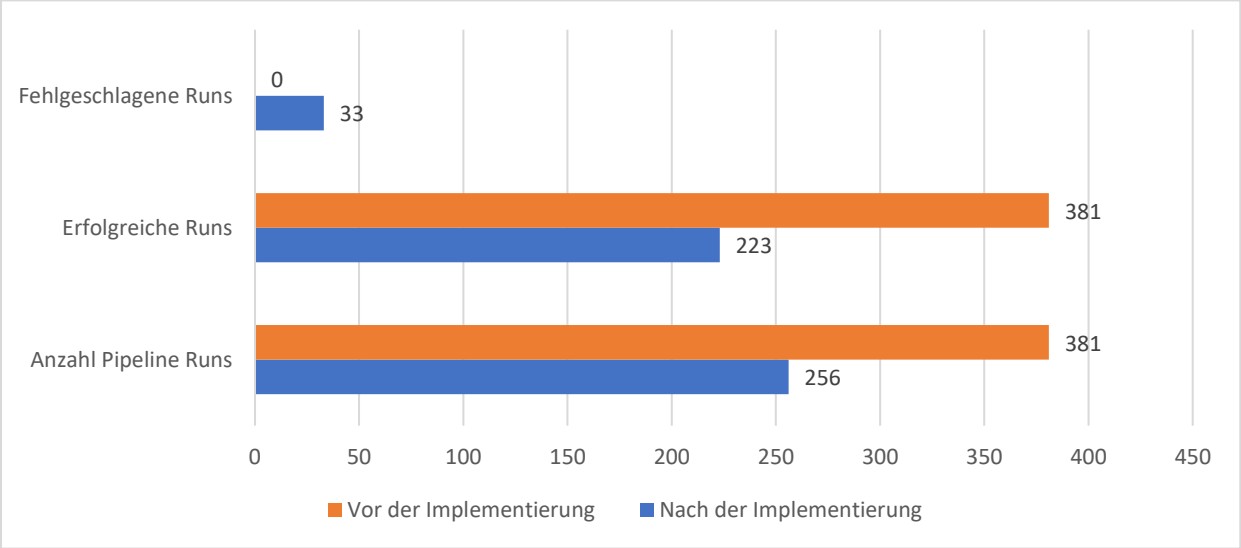

Abbildung 7: Bereinigter Anteil erfolgreicher und fehlgeschlagener Pipelines

Offenkundig liegt ein signifikanter Unterschied zwischen dem Vorhandensein und dem Fehlen von Continuous Integration vor. Die eingeführten Tests waren in der Lage, mehrere Fehler innerhalb der Anwendung zu entdecken und ein Deployment auf Produktivumgebung zu verhindern.

Mithilfe der gesammelten Jira-Tickets kann eindeutig gezeigt werden, dass sich die Anzahl kritischer Fehler auf der Produktivumgebung verringert hat.

Vor der Implementierung von CI wurden 15 fehlerhafte Deployments getätigt, welche im Anschluss durch Jira-Tickets dokumentiert wurden. Die Nutzung von CI hat dies auf zwei fehlerhafte Deployments reduziert.

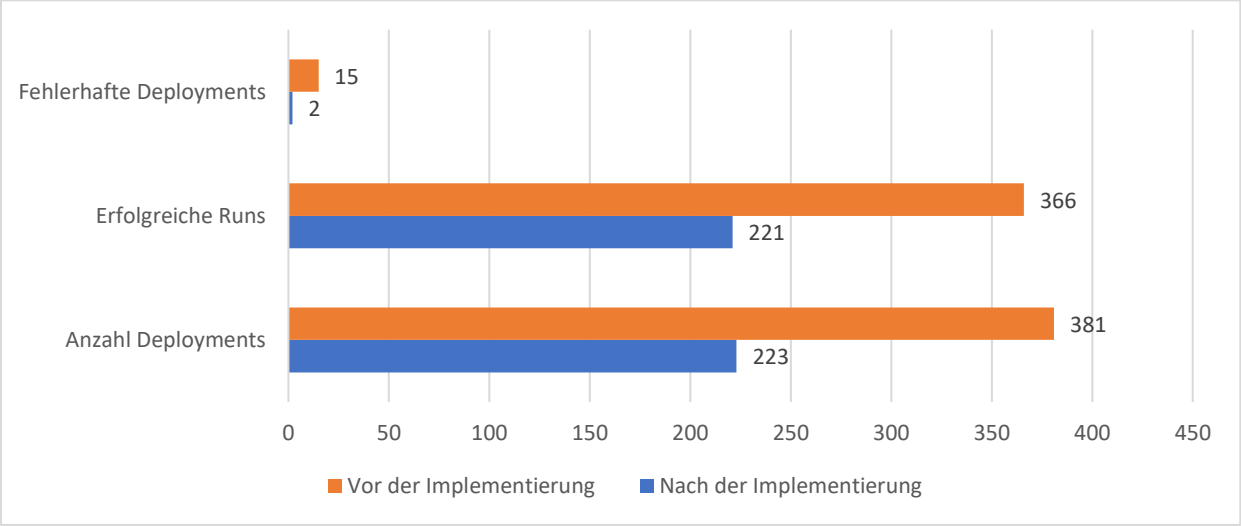

Abbildung 8: Anteil unentdeckter Fehler mithilfe von Jira-Tickets

4.1 Signifikanzwertanalyse

Die Signifikanzwertanalyse dieser Werte zeigt, dass sowohl GitLab-Pipeline-Logs als auch Jira-Tickets eine signifikante Abweichung zwischen vor und nach der Implementierung von CI aufweisen.

Tabelle 2: Signifikanz von Systemtests

Ebene	Signifikanz	Bedeutung
GitLab-Pipeline	0%*	$p < 5\%$ signifikante Abweichung
Jira-Tickets	0,998%	$p < 5\%$ signifikante Abweichung

Anmerkung: *Werte $< 1 * 10^{-10}$ auf 0 gerundet

Zusätzlich zu der Umsetzung von Systemtests (Black-Box Tests) wurden außerdem Unit Tests (White-Box Tests) implementiert. Diese Tests wurden parallel zu den Systemtests ausgeführt, wodurch immer erkennbar war, ob eine der Tests Fehler entdeckt hat. Bei den 23 umgesetzten Testfällen wurden ebenfalls 256 GitLab-Pipeline-Runs durchgeführt und protokolliert. Die Unit Tests waren jedoch nicht in der Lage, einen Fehler innerhalb der Anwendung zu ermitteln. Die Systemtests haben im selben Zeitraum gezeigt, dass Fehler im System auftraten. Wie zu erwarten konnten Unit Tests diese jedoch nicht ermitteln, da die Testabdeckung der Anwendung in einer Woche

nicht ausreichend mithilfe von Unit Tests erreicht werden konnte. Die Gründe hiervon, werden im Unterpunkt 4.3 näher betrachtet.

Der Signifikanztest zeigt hier einen Signifikanzwert von 100%, d.h. es liegt keine Signifikanz im Vergleich zu vor der Implementierung vor, da der Wert über der Signifikanzschwelle von 5% liegt. Die Signifikanz wurde hier mithilfe des Exakten Test nach Fischer berechnet, da für das Chi-Quadrat mindestens fünf Datensätze in jeder zu analysierenden Reihe benötigt werden.

Im direkten Vergleich zwischen White-Box und Black-Box Tests liegt ein signifikanter Unterschied vor.

Tabelle 3: Signifikanz von Unit Tests im Vergleich

Ebene	Signifikanz	Bedeutung
Unit Test vs. Vor der Implementierung	100%	$p > 5\%$ keine Signifikanz
Unit Test vs. Systemtest	0%*	$p < 5\%$ Signifikante Abweichung

Anmerkung: *Werte $< 1 * 10^{-10}$ auf 0 gerundet

Eine generelle Problematik von Unit Tests kommt an dieser Stelle zum Vorschein. Unit Test sind sehr feingranular, da sie sehr kleine Elemente einer Anwendung einzeln testen. Dadurch weiß ein Entwickler, wenn ein Fehler gefunden wurde, exakt an welcher Stelle und warum der Fehler aufgetreten ist. Zum Diagnostizieren und Ausbessern von Fehlern ist dies ein großer Vorteil. Die Konsequenz ist jedoch, dass eine große Anzahl von Testfällen erstellt werden müssen, um große Teile einer Anwendung abzudecken. Wie bereits in den Grundlagen dieser Arbeit erläutert wurde, ist das ideale Verhältnis von Tests zu Funktionen 1:1. Der Implementierungsaufwand von Unit Tests steigt also linear mit dem Wachstum der Anwendung.

An den Signifikanztests ist hier klar erkennbar, dass die Abdeckung der Anwendung viel zu gering ist, da kein signifikanter Unterschied zu vor der Implementierung jeglicher Tests erkennbar war.

Im Gegensatz dazu waren die Systemtests in der Lage, eine Vielzahl von Fehlern zu erkennen und fehlerhafte Deployments zu verhindern. Die Anzahl der Tests, die hier implementiert werden müssen, wächst viel langsamer, da ein einziger Systemtest hunderte Funktionen der Anwendung auf einmal abdeckt. Der Nachteil ist hierbei, dass die feine Granularität der Unit Tests fehlt und es schwieriger ist, Fehler ausfindig zu machen.

4.2 Softwarequalitätsmetriken

Zusätzlich zu Signifikanztests wurden Softwarequalitätsmetriken genutzt. Unterschieden wird bei diesen Metriken zwischen vor und nach der Implementierung. Da keine Fehler mithilfe von White-Box Tests erkannt werden konnten, sind diese deckungsgleich mit den Werten vor der Implementierung. Nach der Implementierung beschreibt hier folglich ausschließlich Black-Box Tests.

Als erste Metrik wurde die Zeitanalyse genutzt. Eine Reduzierung der Flaws (Fehler) pro Monat ist hier klar erkennbar. Vor der Implementierung waren 1,25 Flaws pro Monat zu erwarten. Nach der Implementierung hat sich dieser Wert auf 0,166 Flaws reduziert. Die folgende Tabelle 4 zeigt die Werte hoch und heruntergerechnet auf verschiedene Zeitspannen.

Tabelle 4: Zeitanalyse

Zeitspanne	Nach der Implementierung	Vor der Implementierung
Flaws pro Jahr	2	15
Flaws pro Monat	0,166666667	1,25
Flaws pro Tag	0,005555556	0,041666667
Flaws pro Stunde	0,000231481	0,001736111
Flaws pro Minute	3,85802E-06	2,89352E-05
Flaws pro Sekunde	6,43004E-08	4,82253E-07

Wenn davon ausgegangen wird, dass ein Flaw bzw. Fehler eine Standarddowntime von vier Stunden verursacht, dann ergeben sich folgende Downtime-Werte:

Tabelle 5: Zeitanalyse in Downtime

Zeitspanne	Nach der Implementierung	Vor der Implementierung
Downtime pro Jahr	28800s	216000s
Downtime pro Monat	2400s	18000s
Downtime pro Tag	80s	600s
Downtime pro Stunde	3,333333333s	25s
Downtime pro Minute	0,055555556s	0,416666667s
Downtime pro Sekunde	0,000925926s	0,006944444s

Im Kontext von Kosten spielt die insgesamte Downtime eine große Rolle. Eine Umfrage aus 2020 hat gezeigt, dass jede Stunde an Server-Downtime mehr als 300.000 Dollar in 88% aller Unternehmen verursacht [24, S. 31]. Diese Umfrage bezieht sich zwar auf den Betrieb von Server-Hardware und nicht Softwareprojekten, sie stellt jedoch dar, wie wichtig eine minimale Downtime ist. Zudem sind gerade bei großen Shops wie Telekom.de, Verluste in dieser Höhe möglich. Zum Beispiel eine Verbesserung der Downtime von 60 Stunden pro Jahr auf acht Stunden, würde bei dem Richtwert von 300.000 Dollar pro Stunde eine Verbesserung von 15,6 Millionen Dollar bedeuten.

Als zweite Metrik wurde die Reliability betrachtet. 87% der befragten Unternehmen setzen „four nines“ also 99,99% Reliability als ein Minimum voraus [24, S. 32]. Downtime beschreibt in diesem Kontext Vital- bzw. Total Reliability. Vor der Implementierung von CI ist hier zu erkennen, dass eine Total Reliability von 99,65% beziehungsweise „two nines“ vorliegt. Die Nutzung von CI hat die Total Reliability auf 100% verbessert, da in dem gesamten Überprüfungszeitraum kein totaler Systemzusammenbruch beobachtet wurde. Realistischer für die Bewertung der Reliability ist hier die Vital Reliability da hier der Ausfall einzelner kritischer Funktionen betrachtet wird. Hierbei zeigt die Implementierung von CI ebenfalls eine Verbesserung auf. Die Reliability liegt hier bei 99,95%, während vorher eine Reliability von 99,4% vorlag.

Tabelle 6: Reliability Metriken

Reliability	Nach der Implementierung	Vor der Implementierung
Full Reliability	0,999084249	0,993131868
Vital Reliability	0,999542125	0,994848901
Total Reliability	1	0,996565934

Wie in Tabelle 6 zu erkennen ist, hat die Implementierung von CI die Vital Reliability von „two nines“ auf „three nines“ verbessert.

Die letzte genutzte Metrik zur Bewertung der Softwarequalität ist die Maintainability. Hierbei wurden die Diagnostic support capability (DSCp) und das Change success ratio (CSR) betrachtet. Mithilfe des DSCp ist erkennbar, dass Continuous Integration sehr effektiv darin ist, Fehler aufzuzeigen. 94% aller Fehler wurden mithilfe von Black-Box Tests erkannt. Vor der Implementierung beziehungsweise White-Box Tests waren nicht in der Lage, Fehler ausfindig zu machen, wodurch eine DSCp von 0% in Tabelle 7 vorliegt.

Tabelle 7: Maintainability Metriken

Maintainability	Nach der Implementierung	Vor der Implementierung
DSCp	0,942857143	0
CSR	0,979591837	1

Das CSR zeigt zudem, dass Fehler sehr häufig direkt erkannt werden und Änderungsfehler vermieden werden können. Hier liegt nach der Implementierung von Black-Box Tests ein CSR von 97,95% vor. Vor der Implementierung bzw. bei White-Box Tests liegt ein CSR von 100% vor, da nie ein Änderungsfehler vermerkt werden konnte.

4.3 Bewertung anhand der Forschungskriterien

Signifikanztests und Softwarequalitätsmetriken gewähren sehr gute Einblicke in die Effektivität von Softwaretests. Beide Ansätze wurden in einem Zeitraum von jeweils einer Woche entwickelt. Hierbei hat sich gezeigt, dass Black-Box Tests ideal für eine schnelle und dennoch effektive Abdeckung von Testfällen sind. Black-Box Tests konnten eine hohe signifikante Abweichung zu vor der Implementierung von Testfällen aufzeigen. Zudem ist eine Verbesserung in jeder der betrachteten Qualitätsmetriken zu verzeichnen. Im Durchschnitt haben diese Tests die gesamte Downtime im Jahr um mehr als 52 Stunden reduziert. Ein Implementierungsaufwand von einer Woche à 40 Stunden ist in diesem Fall sehr kosteneffektiv.

Im Gegensatz dazu haben White-Box Tests gezeigt, dass ein sehr viel größerer Aufwand benötigt wird, um einen gleichwertigen Unterschied zu erreichen. Signifikanztests und Softwarequalitätsmetriken haben hier aufgezeigt, dass White-Box Tests keinen signifikanten Unterschied zu vor der Implementierung erreichen konnten.

Die generelle Effektivität von Softwaretests bleibt unangefochten.

Eine Anwendung mit Softwaretests wird maximal genauso viele Fehler wie eine ohne Tests auf Produktivumgebungen gestatten. Die im Rahmen dieses Projekts umgesetzte Anwendung hat jedoch eine Verbesserung von 68,13% auf Produktivumgebungen erfahren.

Die Umsetzungsdauer und Projektgröße haben bei der Implementierung eine sehr große Rolle gespielt. Mit größer werdenden Projekten wird auch der Aufwand für die Implementierung von Testfällen größer. Das lineare Wachstum von White-Box Tests mit der Anzahl der Funktionen war hier ein gewichtiger Nachteil, da das vorliegende Projekt hunderte Funktionen besitzt und daher nur eine sehr beschränkte Testabdeckung in dem gegebenen Zeitraum erreicht werden konnte. Black-Box Tests dagegen hatten einen klaren Vorteil, da der Aufwand bei ihnen logarithmisch verläuft. Eine sehr kleine Anzahl an Testfällen sind in der Lage, große Teile einer Anwendung abzudecken.

Ein weiterer wichtiger Faktor ist der Release-Zyklus der Software. Ein Projekt, welches nur sehr wenige Releases im Jahr veröffentlicht, ist in der Lage, über einen längeren Zeitraum Unit Tests zu implementieren, da nicht tagtäglich vollständige Releases ausgerollt werden müssen. Releases können bei langen Zyklen manuell getestet werden, bis die Implementierung von White-Box Tests fertiggestellt ist.

Projekte mit kurzen Entwicklungszyklen, wie z.B. Projekte die agil entwickelt werden, können nicht für jeden Release manuell alle Funktionalitäten der Anwendung überprüfen. Hier wird meist eine sofortige Abhilfe durch Testsuites benötigt. Wie in den Ergebnissen dieses Projektes gezeigt wurde, konnten Black-Box Tests mit großer Effektivität die Anzahl der Fehler mit einem nur minimalen Umsetzungsaufwand reduzieren.

Laut Islam et al. [26, S. 3] wird im Wasserfallmodell bis zu 30% des Entwicklungszyklus für die Erstellung von „Testing Deliverables“ genutzt. Diese umfassen Test-Scripts, Defect-Reports und User-Feedback. Der eigentliche Coding-Schritt dagegen ist auf 35% ausgelegt. Die in diesem Projekt implementierte Software wird seit zwei Jahren aktiv entwickelt. Die Implementierung von White-Box und Black-Box Tests wurde jeweils in einer Woche durchgeführt. Anteilig bedeutet das, dass das Testing bisher insgesamt 1,92% der Entwicklungsdauer beansprucht hat. Black-Box Tests haben eine spürbare Verbesserung der Zuverlässigkeit der Anwendung hervorgerufen. Davon ausgehend kann festgehalten werden, dass Black-Box Testing eine kosten- und

zeiteffektive Methode zur Abdeckung von Testfällen und der Absicherung einer Anwendung ist.

Pan [27, S. 5] beschreibt Testing als „endlos“ und nennt zwei Ansätze als Stopppunkt von Testing. Es ist unmöglich alle Defekte ermitteln und beheben zu können. Realistisch gesehen ist Testing ein Kompromiss zwischen Budget, Zeit und Qualität. Für diesen Kompromiss gibt es verschiedene Ansätze. Der häufigste Ansatz ist der Pessimistische, mit dem Testen aufzuhören, wenn eine der gegebenen Ressourcen ausgereizt ist. Der optimistische Ansatz dagegen ist es aufzuhören, wenn entweder die Zuverlässigkeit den Anforderungen entspricht oder die Kosten der Implementierung von weiteren Tests nicht mehr gerechtfertigt werden können.

Die in dieser Arbeit betrachtete Anwendung wird als unkritisch betrachtet. Das bedeutet, dass sie lediglich supplementär zu den Kernanwendungen der Deutschen Telekom IoT GmbH genutzt wird. Genaue Zielwerte für die Zuverlässigkeit lagen nicht vor. Eine Verbesserung der Zuverlässigkeit auf mindestens 99,9% war jedoch wünschenswert.

Diese Verbesserung konnte ohne eine vollständige Testabdeckung erreicht werden. Implementierung weiterer Testfälle wäre redundant, da diese 24/7 betriebene Anwendung bereits die gewünschte Mindestzuverlässigkeit erfüllt und weitere Testfälle keine Verbesserung garantieren können. Eine bessere Allokation von Ressourcen ist hier bei fortschreitender Entwicklung eine korrekte testgetriebene Anwendungsentwicklung zu betreiben. Das bedeutet für jede neue Funktionalität, die implementiert werden soll, sollten vorerst Testfälle in Form von Unit Tests (White-Box Tests) erstellt werden.

5 Literarischer Vergleich

Das folgende Kapitel setzt sich explizit mit den Unterschieden bezüglich der theoretischen und praktischen Umsetzung von Continuous Integration auseinander. Zudem werden Beobachtungen vorheriger Kapitel mit Ergebnissen ähnlicher Projekte beziehungsweise Fallstudien verglichen.

5.1 Vergleich der Effektivität

Ein Kernkritikpunkt von Continuous Integration ist, dass es keine einheitliche Methode beziehungsweise „Best Practices“ gibt, mit denen immer eine ideale Pipeline aufgesetzt werden kann. Ståhl und Bosch [28, S. 58], [29] beschreiben eine Uneinigkeit zwischen Softwareentwicklern, bezüglich der Vorteile von Continuous Integration. Zudem wurde demonstriert, dass Continuous Integration-Ansätze sich stark voneinander unterscheiden können, wodurch ein Unterschied hinsichtlich der Effektivität möglich sein kann.

Die in diesem Projekt implementierte Anwendung hat gezeigt, dass in sehr kurzer Zeit eine sehr effektive Testsuite mithilfe von funktionalen Tests (Black-Box Tests) erstellt werden kann. Viele der betrachteten Beispiele in der Literatur stimmen zu, dass diese Art von Tests sehr schnell implementierbar sind [30, S. 5], [31], [32]. Ein Problem, dass in diesem Projekt nicht betrachtet werden konnte, ist das Auftreten von Fehlalarmen. Laut Holmes und Kellog [30, S. 5] ist eines der fundamentalsten Probleme von Graphical User Interface (GUI) Testing, dass triviale Änderungen an der Anwendung zum fehlschlagen von Tests führen können, ohne das ein Fehler vorliegt. So reicht es beispielweise aus den Namen einer Schaltfläche zu ändern. Tests, die diese Schaltfläche nutzen, würden fehlschlagen, da sie eine Schaltfläche mit anderen Namen erwarten. Diese Fragilität im Frontend Testing führte zu einem misstrauen der Testfälle als sofortiges Feedback. Innerhalb eines Zeitraums einer Iteration hatten sich diese jedoch stabilisiert. Dieses Problem konnte bisher nicht in dem betrachteten Projekt beobachtet werden. Eine mögliche Erklärung dafür ist, dass sich das Frontend der Anwendung im Zeitraum der Testdurchführung nur sehr minimal geändert hatte und der Großteil der Änderungen an den Funktionalitäten abseits des User Interface (UI) gemacht wurden. Zudem muss hier auch der Zweck der Testfälle näher betrachtet werden. Verschiedene Nutzungszwecke werden daher im folgenden Unterpunkt behandelt.

5.2 Vergleich von Nutzungszwecken

Eine Beobachtung, die in der Literatur gemacht wurde, ist, dass Continuous Integration genutzt wird, um die Produktivität von Entwicklern zu erhöhen und die Vorhersehbarkeit von Anwendungsfehlern zu verbessern [29, S. 7–8]. Abhängig davon, wie stark diese Zwecke gewichtet werden, können sehr unterschiedliche Implementierungen Zustandekommen.

Der Zweck des hier implementierten Projektes lag hauptsächlich auf der Verbesserung der Vorhersehbarkeit von Anwendungsfehlern und dem Vermeiden jeglicher Fehler auf Produktivumgebungen. Der Fokus war folglich die Optimierung der Zuverlässigkeit des gegebenen Systems. Ein Fehlalarm, wie in der Literatur erwähnt, ist in diesem Fall unkritisch, da keine direkten Änderungen an der Produktivumgebung gemacht werden. Die Zeit die durch die Analyse fälschlicher Fehler verloren geht, wird geringer gewichtet als die verringerte Entwicklungsdauer von Tests und die verbesserte Uptime der Anwendung.

Große Unterschiede in der Effektivität wurden bei Beispielen bemerkt, in welchen ein großer Fokus auf Fehleranalyse beziehungsweise Erhöhung der Entwicklerproduktivität gelegt wurde. Aus der Literatur ist zu erkennen, dass die Meinungen der Experten sich in diesem Punkt stark variieren [29, S. 7]. Ein Ansatz, der hier sehr viel Anklang findet, ist der des Test-Driven-Development (TDD) [28, S. 54], [33]. Hierbei werden zuerst Testfälle erstellt und daraufhin Funktionen implementiert. Die Testfälle sollen hierbei so lange fehlschlagen, bis die Funktion korrekt funktioniert und die Anforderungen erfüllt. Um diesen Ansatz zu verfolgen, werden Unit Tests (White-Box Tests) benötigt. Die Ergebnisse dieses Projektes haben gezeigt, dass Unit Tests einen sehr großen Implementierungsaufwand mit sich bringen. Generell werden 30-50% der Entwicklungszeit im Softwareentwicklungszyklus für Testing eingeplant [34]. Unit Tests als zeitaufwändigste Testart nehmen hierbei einen großen Anteil dieser Zeit ein. Wenn die gegebenen Anforderungen einer Anwendung es zulassen, dass Unit Tests ausgelassen werden können, ist ein sehr großer Zeitgewinn im Entwicklungszeitraum möglich. Das in dieser Arbeit betrachtete Telekom-Projekt, hatte keine Anforderungen hinsichtlich Unit Tests. Zudem war nur eine Zuverlässigkeit von 99,9% gewünscht. In diesem Fall ist eine zusätzliche Implementierung von Unit Tests überflüssig, da das gewünschte Maß an Zuverlässigkeit bereits erreicht wurde. Sollte sich diese Anforderung in Zukunft ändern, müssten Unit Tests ausführlich für die gesamte Anwendung implementiert werden, um Grenzfälle besser testen zu können.

5.3 Verallgemeinerung der Ergebnisse

Die Ergebnisse können den Anschein erwecken, dass funktionale Tests ausreichen, um eine Anwendung in befriedigendem Maße zu testen. Diese sind jedoch den Anforderungen und dem Umfang des hier betrachteten Projektes geschuldet. Test-Driven-Development, Unit Testing und umfangreiche Testbibliotheken werden häufig in der Literatur empfohlen, da Umfang, Kritikalität und Anforderungen sich in jeder Anwendung unterscheiden. Beginnt man mit der Entwicklung eines neuen Projektes, ist es ratsam, diese Praktiken umzusetzen, da Umfang und Anforderungen sich häufig ändern. Trotz dessen muss abgewogen werden, wie umfangreich eine Anwendung werden soll. Eine kleine, unkritische Anwendung ohne Anforderungen beziehungsweise Kunden (wie ein privates Hobbyprojekt), benötigt keine umfangreiche Testsuite. Im Gegensatz dazu sollte eine große, kritische Anwendung wie das Navigationssystem für das autonome Fahren, in vollem Maße getestet werden.

Wie von Ellims et al. [34] beschrieben werden 30-50% der Entwicklungszeit im Softwareentwicklungszyklus für Testing eingeplant. Testing ist einer der Schritte im Entwicklungszyklus, der den möglichen Umfang einer Anwendung, durch seinen großen Zeitaufwand, stark eindämmt. Daher wäre es vorteilhaft, die aufgewendete Zeit für diesen Schritt zu minimieren. Eine weitere Forschung in diesem Bereich wird jedoch benötigt, um das optimale Verhältnis zwischen Projektumfang und Testaufwand zu bestimmen.

Die Ergebnisse dieser Arbeit haben gezeigt, dass bei einem mittelgroßen Projekt mit geringer Kritikalität die benötigte Entwicklungsdauer von Tests, unter der Nutzung von Black-Box Tests auf 1,92% reduziert werden konnte, ohne die benötigte Zuverlässigkeit zu beeinträchtigen. Von diesen Ergebnissen ausgehend, kann jedoch keine allgemeine Aussage für jede Art von Softwareprojekt getätigt werden. Die große Fülle von Unterschieden hinsichtlich des Projektaufbaus, Unternehmensstrukturen, Softwareentwicklungsprozessen, Zeitspannen, Ressourcen und der Teamgröße sind nicht anhand eines einzelnen Projektes auswertbar.

5.4 Methodischer Rückblick

Im Rahmen dieser wissenschaftlichen Arbeit wurde eine Fallstudie anhand einer Anwendung der Deutschen Telekom sowie eine Literaturanalyse durchgeführt. Diese methodischen Herangehensweisen wurden gewählt, um einen holistischen Überblick bezüglich der Continuous Integration zu gewähren.

Um diesen Einblick zu schaffen, wurden von der Hochschule für Technik, Wirtschaft und Kultur Leipzig bereitgestellte Bibliotheken genutzt und deren Literaturquellen analysiert. Diese Quellen wurden herangezogen um verschiedene theoretische als auch praktische Methoden der Implementierung von Continuous Integration vergleichen und darstellen zu können.

Da es sich hierbei um ein hochaktuelles Thema handelt, welches sich einem ständigen Wandel unterzieht, wurde anhand eines eigenen Projektes dieser Prozess implementiert, um Unterschiede, Übereinstimmungen, Probleme und Erfolge mit verschiedenen Quellen vergleichen zu können. Von den in Kapitel 1.3 erwähnten Zielen der Arbeit, war eines der Ziele herauszufinden, welcher Ansatz sich am besten zur schnellen und effektiven Abdeckung von Testfällen eignet. Hierbei wurde sich auf zwei größere Ansätze beschränkt.

Eine Konkretisierung der spezifischen Ansätze wurde bewusst nicht getroffen, da kein Mehrwert aus der Nutzung programmiersprachen-spezifischer Testmöglichkeiten erkannt werden konnte. Zum Zwecke allgemeingültiger Aussagen zu den Forschungsfragen wurde sich daher auf Ansätze beschränkt, welche in jeder modernen Programmiersprache bzw. Softwareprojekt anwendbar sind. Im Sinne dieser Rahmenbedingungen wurden für die Implementierung von Continuous Integration ausschließlich für die Öffentlichkeit frei zugängliche Technologien genutzt.

Rückblickend ist jedoch anzumerken, dass die Ergebnisse anhand eines einzelnen Projektes keinesfalls allgemeingültig anwendbar sind. In weiteren Forschungen müssten daher größere Stichproben von Projekten, unterschiedlicher Größe und Struktur analysiert und verglichen werden. Jenes Vorhaben würde sich jedoch schwierig umsetzen lassen, da es nahezu unmöglich ist, hunderte Projekte mittels eines standardisierten Tests und Untersuchungsverfahrens umzusetzen. Zudem wäre die Kooperation einer Vielzahl von Unternehmen unterschiedlicher Größe erforderlich, welche keinen direkten Vorteil durch ihre Kooperation erhalten würden.

Im Hinblick auf die genutzten Literaturquellen wurden hauptsächlich englischsprachige Quellen verwendet. Da diese Arbeit in der deutschen Sprache verfasst wurde, mussten eine Vielzahl von Auszügen übersetzt werden. Ein großer Wert wurde daher auf die korrekte und sinngemäße Übersetzung der Inhalte gelegt. Ein Fokus auf deutschsprachige Quellen wäre für eine konsistente Sprachverwendung vorteilhaft, jedoch würde dies eine Limitierung darstellen, da das Feld der Informationstechnologie ein überwiegend englischsprachiger, internationaler Raum ist.

Zuletzt ist anzumerken, dass zur Prävention von subjektiv-geprägten Aussagen, die Auswertung der Ergebnisse dieser Arbeit im Vier-Augen-Prinzip überprüft wurde. Da es sich hierbei um eine Bachelorarbeit handelt, wurde auf weitere methodische Maßnahmen in dieser Hinsicht verzichtet.

6 Fazit

Continuous Integration als Maßnahme zur Automatisierung von Software-Testing befindet sich in einem ständigen Wandel. Mit immer größeren Projekten und Anforderungen muss die Integrität und Zuverlässigkeit von Anwendungen in sehr kurzer Zeit gewährleistet werden. In vielen Bereichen der Industrie ist es bereits ein integraler Bestandteil des Softwareentwicklungsprozess und gewinnt in der immer agiler werdenden Welt der Softwareentwicklung mehr und mehr an Bedeutung. Das ist mit Blick auf analysierte Literaturquellen, als auch den Bedürfnissen innerhalb der Deutschen Telekom IoT GmbH zu erkennen.

Eine sich stets weiterentwickelnde kompetitive Softwarelandschaft und immer agiler werdende Entwicklungsumgebungen setzen auf häufige, robuste und zuverlässige Software-Releases. Anhand dieser Erkenntnisse lässt sich eine Prognose künftiger Entwicklungen aufstellen:

Zukünftige Softwareprojekte werden vermehrt in agilen Teams mit automatisierten Tests und Deployments erstellt, um die Entwicklungsgeschwindigkeit zu erhöhen und das Ausfall- beziehungsweise Fehlerrisiko zu minimieren. Die enge Verbindung zwischen Developer und Operations macht kommunikative und kollaborative Fähigkeiten innerhalb des Teams immer bedeutsamer.

Generell muss der Entwicklungsprozess von Continuous Integration zukünftig vereinheitlicht werden. Viele der heutigen Ansätze beruhen darauf, dass zu Beginn des Entwicklungszeitraums mit Test-Driven-Development unter der Nutzung von Unit Tests begonnen und dabei eine Testabdeckung von nahezu 100% beibehalten wird. Wie in dieser Arbeit erkennbar, muss dieser Ansatz nicht in jedem Fall implementiert werden. Der Vorteil einer sehr feingranularen Testabdeckung mithilfe von Unit Tests ist mit einem großen Zeitaufwand verbunden. Sehr große Projekte, welche bereits über längere Zeit ohne Testsuite ausgekommen sind, sollten allein durch Black-Box Tests eine zufriedenstellende Zuverlässigkeit bei den bereits existierenden Funktionalitäten erreichen können.

Dennoch sollen diese Ergebnisse keinen Entwickler in die Verlegenheit bringen weniger zu Testen denn,

"Es ist schwer genug Fehler in Code zu finden, wenn man danach sucht;

Es ist noch schwerer, wenn man davon ausgeht, dass der Code Fehlerfrei ist.".

Literaturverzeichnis

[1] P. Ammann und J. Offutt, *Introduction to software testing*. New York: Cambridge University Press, 2008.

[2] „Django Documentation“, *djangoproject.com*. https://buildmedia.readthedocs.org/media/pdf/django/latest/django.pdf (zugegriffen 20. Februar 2022).

[3] L. P. da Silva und P. Vilain, „Execution and code reuse between test classes“, in *2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA)*, Juni 2016, S. 99–106. doi: 10.1109/SERA.2016.7516134.

[4] S. Otte, „Version Control Systems“. https://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2008-09_WS/S_19565_Proseminar_Technische_Informatik/otte09version.pdf (zugegriffen 10. August 2021).

[5] S. Garg und S. Garg, „Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security“, in *2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, März 2019, S. 467–470. doi: 10.1109/MIPR.2019.00094.

[6] R. Patton, *Software testing*. Indianapolis, IN : Sams Pub., 2006. Zugegriffen: 11. August 2021. [Online]. Verfügbar unter: http://archive.org/details/softwaretesting0000patt

[7] A. Kolawa und D. Huizinga, *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[8] B. C. S. W. G. on Testing, *Testing in Software Development*. Cambridge University Press, 1986.

[9] B. ANIRBAN, *SOFTWARE QUALITY ASSURANCE, TESTING AND METRICS*. PHI Learning Pvt. Ltd., 2015.

[10] M. Shahin, M. Ali Babar, und L. Zhu, „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“, *IEEE Access*, Bd. 5, S. 3909–3943, 2017, doi: 10.1109/ACCESS.2017.2685629.

[11] M. Shahin, M. A. Babar, M. Zahedi, und L. Zhu, „Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges“, in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Nov. 2017, S. 111–120. doi: 10.1109/ESEM.2017.18.

[12] F. H. Vera-Rivera, „A development process of enterprise applications with microservices“, *J. Phys. Conf. Ser.*, Bd. 1126, S. 012017, Nov. 2018, doi: 10.1088/1742-6596/1126/1/012017.

[13] L. Zhu, L. Bass, und G. Champlin-Scharff, „DevOps and Its Practices“, *IEEE Softw.*, Bd. 33, Nr. 3, S. 32–34, Mai 2016, doi: 10.1109/MS.2016.81.

[14] *Digital Practitioner Body of Knowledge Standard*. The Open Group, 2019.

[15] „Automated software testing in Continuous Integration (CI) and Continuous Delivery (CD) – Continuous Improvement“. https://pepgotesting.com/continuous-integration/ (zugegriffen 11. August 2021).

[16] J. Webster und R. T. Watson, „Analyzing the Past to Prepare for the Future: Writing a Literature Review“, *MIS Q.*, Bd. 26, Nr. 2, S. xiii–xxiii, 2002.

[17] R. K. Yin, *Case study research and applications: design and methods*. 2018.

[18] Y. Singh, *Software testing*. Cambridge ; New York: Cambridge University Press, 2011.

[19] P. Runeson, „A survey of unit testing practices“, *IEEE Softw.*, Bd. 23, Nr. 4, S. 22–29, Juli 2006, doi: 10.1109/MS.2006.91.

[20] M. F. Aniche, G. A. Oliva, und M. A. Gerosa, „What Do the Asserts in a Unit Test Tell Us about Code Quality? A Study on Open Source and Industrial Projects“, in *2013 17th European Conference on Software Maintenance and Reengineering*, März 2013, S. 111–120. doi: 10.1109/CSMR.2013.21.

[21] R. Ramler, M. Moser, und J. Pichler, „Automated Static Analysis of Unit Test Code“, in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, März 2016, Bd. 2, S. 25–28. doi: 10.1109/SANER.2016.102.

[22] E. L. Lehmann, „Significance Level and Power“, *Ann. Math. Stat.*, Bd. 29, Nr. 4, S. 1167–1176, Dezember 1958, doi: 10.1214/aoms/1177706448.

[23] „Software Product Quality Metrics“, in *Software Quality: Concepts and Practice*, Hoboken, NJ, USA: John Wiley & Sons, Inc., 2018, S. 346–374. doi: 10.1002/9781119134527.ch16.

[24] IBM und Lenovo, Hrsg., „ITIC 2020 Global Server Hardware, Server OS Reliability Report“. April 2020. Zugegriffen: 13. Dezember 2021. [Online]. Verfügbar unter: https://www.ibm.com/downloads/cas/DV0XZV6R

[25] K. Schwaber, „The Scrum Guide“. November 2020. Zugegriffen: 27. Januar 2022. [Online]. Verfügbar unter: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf#zoom=100

[26] A. K. M. Z. Islam und Dr. A. Ferworn, „A Comparison between Agile and Traditional Software Development Methodologies“, *Glob. J. Comput. Sci. Technol.*, S. 7–42, Dez. 2020, doi: 10.34257/GJCSTCVOL20IS2PG7.

[27] J. Pan, „Software Testing“. 1999. Zugegriffen: 7. Februar 2022. [Online]. Verfügbar unter: http://www.sci.brooklyn.cuny.edu/~sklar/teaching/s08/cis20.2/papers/software-testing.pdf

[28] D. Ståhl und J. Bosch, „Modeling continuous integration practice differences in industry software development“, *J. Syst. Softw.*, Bd. 87, S. 48–59, 2014, doi: 10.1016/j.jss.2013.08.032.

[29] D. Ståhl und J. Bosch, „Experienced Benefits of Continuous Integration in Industry Software Product Development: A Case Study“, März 2013, doi: 10.2316/P.2013.796-012.

[30] A. Holmes und M. Kellogg, „Automating functional tests using Selenium“, in *AGILE 2006 (AGILE'06)*, Juli 2006, S. 6 pp. – 275. doi: 10.1109/AGILE.2006.19.

[31] „Continuous Integration“, *martinfowler.com*. https://martinfowler.com/articles/continuousIntegration.html (zugegriffen 8. Februar 2022).

[32] S. Stolberg, „Enabling Agile Testing through Continuous Integration“, in *2009 Agile Conference*, Aug. 2009, S. 369–374. doi: 10.1109/AGILE.2009.16.

[33] P. Gestwicki, „The entity system architecture and its application in an undergraduate game development studio“, in *Proceedings of the International Conference on the Foundations of Digital Games - FDG '12*, Raleigh, North Carolina, 2012, S. 73. doi: 10.1145/2282338.2282356.

[34] M. Ellims, J. Bridges, und D. C. Ince, „The Economics of Unit Testing“, *Empir. Softw. Eng.*, Bd. 11, Nr. 1, S. 5–31, März 2006, doi: 10.1007/s10664-006-5964-9.

Danksagung

An dieser Stelle möchte ich mich bei den Leuten bedanken, die mich während der Erstellung dieser Arbeit unterstützten und motivierten.

Zuerst danke ich meinem ersten Betreuer Herrn Prof. Dr. Christian-Alexander Bunge. Herr Bunge war sowohl in der Vermittlung von Vorlesungsinhalten als auch der Betreuung dieser Arbeit immer hilfreich und engagiert. Für unsere häufigen Termine und die konstruktive Kritik zu der Erstellung dieser Arbeit möchte ich mich herzlich bedanken.

Zusätzlich möchte ich meinem zweiten Betreuer und Business Experten Herrn Eric Schmieder danken. Schon weit vor Beginn dieser Arbeit nahm er sich die Zeit, um mit mir Ideen und Themen zu finden. Bei Problemen und Fragen war er stets verfügbar, um zu helfen. Dafür bin ich ihm besonders dankbar.

Ebenfalls möchte ich mich bei meinen Kommilitonen Jannis Mende und Hannes Norbert Göring für das Korrekturlesen meiner Bachelorarbeit bedanken. Deren zahlreichen Ideen und Anmerkungen haben in großem Maß dazu beigetragen, dass diese Arbeit in ihrer jetzigen Form vorliegt.

Abschließend möchte ich mich bei meinen Eltern sowie meinen Großeltern bedanken, die durch ihre allgegenwärtige Unterstützung mir mein Studium ermöglicht haben und stets ermutigende Worte für mich hatten.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die an der Hochschule für Technik, Wirtschaft und Kultur Leipzig, konkret an der Fakultät Digitale Transformation, eingereichte Arbeit zum Thema „Umsetzung von Continuous Integration im Kontext bestehender Anwendungen“ selbstständig, ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe.

Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht. Die Abbildungen in dieser Arbeit wurden von mir selbst erstellt oder mit einem entsprechenden Hinweis auf die Quelle versehen.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

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

Unterschrift PHILIPP BRAUN

Magdeburg, den 5. März 2022