Duale Hochschule Sachsen
Staatliche Studienakademie Leipzig

Effiziente Strategien zur kontinuierlichen Integration langfristiger Feature-Branches

Bachelorthesis

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

in der Studienrichtung Informatik

Eingereicht von: Marcel Schmied
William-Zipperer-Straße 40
04177 Leipzig
CS22-1
Matrikelnr.: 5002241

Erstgutachter: Herr Dipl.-Wi. Inf. Thomas Max
mgm technology partners (QFS)

Zweitgutachter: Herr M.Sc. Klein
Duale Hochschule Sachsen
Staatliche Studienakademie Leipzig

Leipzig, 1. September 2025

Inhaltsverzeichnis

Abkürzungsverzeichnis

AST Abstract Syntax Tree

bat Startbefehl buildAndTest

CR Conflict-Resolution

CI kontinuierliche Integration (engl. Continuous Integration)

CD kontinuierliche Auslieferung (engl. Continuous Delivery)

CT kontinuierliches Testen (engl. Continuous Testing)

CVS Concurrent Versions System

DEV Entwicklung

DOKU Dokumentation

DVCS Distributed Version Control Systems

pp20 Name des Hauptservers im Unternehmen

QFS Quality First Software GmbH

rerere reuse recorded resolution

sdev Name des Buildservers im Unternehmen

SMTP Simple Mail Transfer Protocol

ssh secureShell

SVN Apache Subversion

VCS Versionsverwaltungssysteme (engl. Version Control Systems)

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

1 Einleitung

Die kontinuierliche Integration von Feature-Branches in einem Versionsverwaltungssystem ist heutzutage ein zentraler Bestandteil der modernen Softwareentwicklung. Sie ermöglicht es, Änderungen aus parallelen Entwicklungszweigen regelmäßig zusammenzuführen und automatisiert zu testen, um mögliche Integrationsprobleme und Wechselwirkungen der Branches frühzeitig zu erkennen. Das funktioniert bei einer kurzen Entwicklungszeit eines Branches in den meisten Fällen ohne große Probleme, stellt jedoch bei langfristigen Feature-Branches immer noch eine Herausforderung dar: Es wird über Monate oder sogar Jahre hinweg an einem komplexen Feature auf einem Branch entwickelt, dessen wachsender Code sich nicht ohne weiteres kontinuierlich in die Basisversion integrieren lässt. Mit zunehmender Zeigt steigt das Risiko von Merge-Konflikten und schwerwiegenden Wechselwirkungen mit anderen Branches erheblich.

Im Unternehmen Quality First Software GmbH (QFS) wurde zur Lösung dieses Problems ein neuer innovativer Ansatz, die `Bouquet-Integration` (zu deutsch: Blumenstrauß) entwickelt. Sie ermöglicht es, eine dynamische Auswahl relevanter Feature-Branches (die `Blumen`) zu einem testbaren Integration-Branch (dem `Blumenstrauß`) zusammenzuführen. Auf diese Weise können Wechselwirkungen zwischen langfristigen Branches frühzeitig sichtbar gemacht und eine beliebige Anzahl an Feature Branches untereinander getestet werden. Das führt zu einer Verbesserung der Qualitätssicherung und Reduzierung von Integrationsproblemen. Die Bouquet-Integration ist eine vielversprechende Eigenentwicklung in Bezug auf kontinuierliche Integration. Bis heute ist diese jedoch nur innerhalb des Unternehmens verfügbar, was zum einen auf ihre Komplexität zurückzuführen ist und zum anderen auf die Verzahnung mit unternehmensinternen Code.

1.1 Motivation

Die `Bouquet-Integration` soll in der vorliegenden Arbeit zum ersten Mal umfassend dokumentiert und damit verständlich dargestellt werden. Hierbei sollen sowohl die technischen Abläufe als auch die zugehörigen Workflows sinnvoll aufbereitet werden, womit zugleich eine Grundlage für zukünftige Weiterentwicklungen geschaffen wird. Im Zuge dieser Arbeit soll die Buildpipeline innerhalb des Unternehmens weiter optimiert und entkoppelt werden, um diese gegebenenfalls künftig in Form einer universell einsetzbaren modularen Toolchain der Entwickler-Community zugänglich zu machen. Dadurch würde eine Möglichkeit geschaffen, die `Bouquet-Integration` erstmals auch außerhalb des Unternehmens QFS einzusetzen, das heißt einen allgemeinen Beitrag zur Lösung der Herausforderungen bei der kontinuierlichen Integration langfristiger Feature-Branches zu leisten.

1.2 Zielsetzung

Aus dieser Motivation heraus ergeben sich die folgenden vier zentralen Thesen:

H1 `Feature-Branches`: Wechselwirkungen langfristiger Feature-Branches können mit bestehenden Ansätzen der kontinuierlichen Integration nicht ausreichend getestet werden.

H2 `Bouquet-Integration`: Die Bouquet-Integration ermöglicht es, Wechselwirkungen zwischen langfristigen Feature-Branches frühzeitig zu erkennen und damit die Qualitätssicherung zu verbessern.

H3 `Optimierungen`: Eine gezielte empirische Umfrage im Entwicklerteam deckt konkrete Optimierungspotenziale in der Buildpipeline auf, mit denen die Buildpipeline verbessert wird.

H4 `Toolchain`: Die Bouquet-Integration kann mithilfe einer modularen Toolchain auch außerhalb des Unternehmens QFS verbreitet und eingesetzt werden.

Sie sind ausformuliert im Thesenpapier in Anhang A. Um diese Thesen zu überprüfen, verfolgt die Arbeit vier zentrale Ziele:

- Analyse bestehender Ansätze zur Umsetzung der kontinuierlichen Integration langfristiger Feature-Branches.
- Untersuchung und Dokumentation der firmeneigenen „Bouquet-Integration“ als alternativer Ansatz, ergänzt durch eine empirische Befragung des Entwicklerteams.
- Ableitung, Implementierung und Evaluation von Optimierungen der bestehenden Buildskripte.
- Konzeption einer eigenständigen modularen Toolchain, welche die Bouquet-Integration implementiert und universell einsetzbar macht.

Zusammenfassend soll die komplexe, unternehmensinterne Buildpipeline analysiert, dokumentiert und mit bestehenden Ansätzen verglichen werden, um die zentrale Frage zu evaluieren, ob die Bouquet-Integration eine Möglichkeit bietet, die Nachteile bestehender Ansätze zu verbessern und damit einen Beitrag zur Weiterentwicklung der Multi-Branch-Integration zu leisten.

1.3 Abgrenzung

Diese Arbeit konzentriert sich ausschließlich auf die technischen Aspekte und Prozesse der Multi-Branch-Integration. Dabei werden organisatorische Methoden des Projektmanagements, wie Scrum oder ökonomische Fragestellungen wie Kosten-Nutzen-Analysen von kontinuierlicher Integration/Auslieferung (CI/CD) nicht betrachtet. Es wird sich ausschließlich auf den Prozess fokussiert, eine beliebige Anzahl an Feature-Branches zu einem Integration-Branch zu integrieren. Alle weiteren Vorgänge einer Buildpipeline wie Build- und Testprozesse werden außen vorgelassen, da diese in den meisten großen Entwicklungsteams bereits vorhanden sind. Ebenfalls nicht Gegenstand sind Funktionen der Buildpipeline jenseits der Bouquet-Integration, z. B. das Erstellen von Builds für einzelne Branches oder die Verwaltung von Testumgebungen.

Die Konzipierung der Toolchain wird bewusst auf die Umsetzung der Bouquet-Integration beschränkt, um sie möglichst verständlich und universell einsetzbar zu gestalten. Weitere Features der firmeninternen Buildskripte sind kein Bestandteil dieser Arbeit.

1.4 Aufbau der Arbeit

Im ersten Teil der Arbeit werden die theoretischen Grundlagen für die kontinuierliche Integration langfristiger Feature-Branches geschaffen, anhand deren die firmeninterne Buildpipeline, insbesondere die Bouquet-Integration evaluiert werden kann. Nach der Aufarbeitung der bestehenden Ansätze zur Umsetzung von CI/CD, wird der Workflow und der technische Aufbau der Buildpipeline ausführlich dokumentiert mit dem Fokus auf den Buildskripten, die für die Umsetzung der Bouquet-Integration zuständig sind. Um diese weiter zu optimieren, wird eine empirische Untersuchung im Entwicklerteam der Firma QFS in Form einer Umfrage durchgeführt, um mögliche Optimierungspotentiale zu ermitteln. Die ermittelten Optimierungen werden zur Verbesserung der Buildpipeline anschließend implementiert. Um die Einsatzmöglichkeiten der Bouquet-Integration über die Grenzen der Firma zu erweitern, wird aus der optimierten Buildpipeline eine modulare Toolchain entwickelt. Diese wird im Zusammenhang mit der Bouquet-Integration abschließend hinsichtlich ihrer Einsatzmöglichkeiten und Verbesserungen aufgrund der Ergebnisse der Entwicklerumfrage evaluiert.

2 Theoretische Grundlagen

Heutzutage gewinnt die parallele Entwicklung voneinander unabhängiger Aufgaben immer mehr an Bedeutung in der Softwareentwicklung, um den steigenden Anforderungen hinsichtlich Time-to-market und Flexibilität gerecht zu werden. Um das zu ermöglichen, werden bei Softwareentwicklungen mit Versionsverwaltungssystemen sogenannte Feature-Branches verwendet, insbesondere beim Einsatz von verteilten Versionsverwaltungssystemen wie Git [1, Kap. 1][2].

2.1 Softwareentwicklung mit Feature-Branches

Unter Feature-Branches versteht man sowohl kurzlebige, als auch langlebige Branches, in denen einzelne Funktionen (Features), Verbesserungen oder Bugfixes entwickelt werden. Dieser Branch ist hierbei eine eigenständige Kopie vom Hauptentwicklungszweig, der häufig `master`, `main` oder `trunk` genannt wird, so dass Änderungen nicht direkt auf diesem implementiert werden. Das Ziel dabei ist, den Entwicklungsprozess modular und unabhängig von anderen Änderungen zu gestalten, Konflikte zu minimieren und den Hauptbranch vor unfertigen oder instabilen Features zu schützen [3].

In der Regel arbeitet ein Teammitglied an einem neuen Feature und erstellt hierfür einen neuen separaten Branch. Dieser Feature-Branch bleibt solange bestehen, bis die Entwicklung abgeschlossen und das Feature getestet wurde. Anschließend wird dieser wieder mit dem Hauptbranch zusammengeführt, was als Merge bezeichnet wird. Vor dem Merge werden im Idealfall alle Änderungen des Hauptbranches auf den Feature-Branch erneut integriert (Rebase), was eine saubere Historie und das Aufdecken von Konflikten ermöglicht. Durch diese Vorgehensweise können beliebig viele Features in einem Team unabhängig voneinander entwickelt und implementiert werden, ohne dabei den Hauptbranch, also den aktuellen Stand des Softwareproduktes zu destabilisieren.

Abbildung 1 stellt ein typisches Branching-Modell mit mehreren Feature-Branches dar, welche zum Schluss wieder zusammengeführt werden, um ein fertiges Endprodukt zu bilden.

In der modernen Softwareentwicklung werden Feature-Branches in jeglichen Projektgrößen eingesetzt. Für kleine Teams reicht meist ein einfaches Branching-Modell, in großen Teams jedoch werden komplexe Strategien wie Git Flow [4], GitHub Flow oder Trunk-Based Development [5](Unterabschnitt 3.2, um eine reibungslose Entwicklung mit vielen Feature-Branches zu koordinieren.

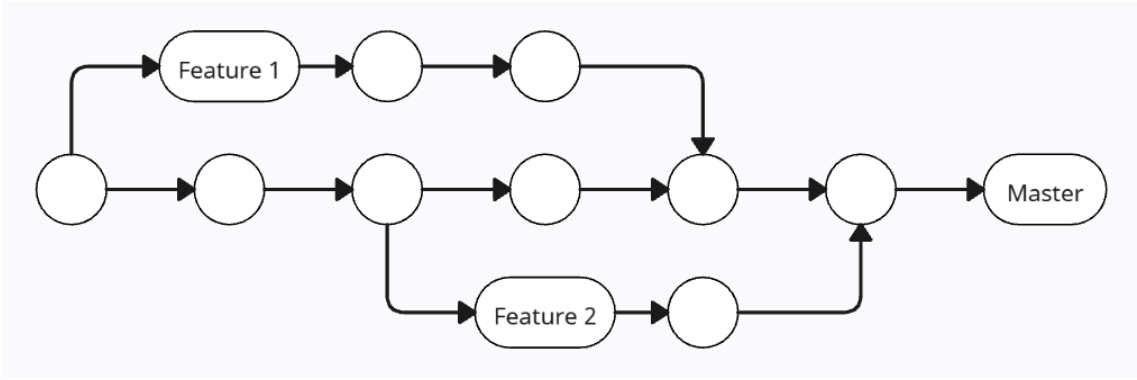


Abbildung 1: Branching-Modell mit Hauptbranch `master` und mehreren Feature-Branches (eigene Darstellung nach [1, Kap. 3].

2.2 Continuous Integration und Continuous Delivery

Die Verwendung von Feature-Branches ist jedoch auch mit Herausforderungen verbunden, welche mithilfe der Prinzipien Continuous Integration (CI) und Continuous Delivery (CD) adressiert werden.

Continuous Integration beschreibt den Prozess, in dem das Zusammenführen der Branches mit dem Hauptbranch nicht erst am Ende des Entwickelns, sondern in regelmäßigen Abständen, idealerweise sogar mehrmals täglich passiert. Das Integrieren der Branches läuft über automatisierte Build- und Testpipelines, welche den Entwicklern und Entwicklerinnen regelmäßiges Feedback zu Integrationsproblemen und fehlschlagenden Tests liefern [2][6, Kap. 2]. Ziel ist es, die Zeit bis zur Fehlererkennung und -behebung zu verkürzen sowie die Softwarequalität und Integrationsfähigkeit zu erhöhen.

Continuous Delivery geht noch einen Schritt weiter und setzt sich als Ziel, dass die Software nach jedem erfolgreichem Integrationsschritt ausgeliefert werden kann. Dafür werden zusätzlich zu der CI weitere Schritte eingeführt, die dafür sorgen, dass nachdem alle Branches im Hauptbranch integriert sind, ein fertiges Build entsteht. Dieses fertige Build soll mit möglichst wenig manuellem Aufwand in die gewünschte Zielumgebung deployt werden können [7]. Durch diverse Prozesse wie automatisierte Build-, Test- und Deployment-Pipelines (z.B. mit Jenkins oder Gitlab CI) kann hiermit die Auslieferungsgeschwindigkeit und Qualität deutlich erhöht werden [3].

Neben den bekannten Prozessen CI und CD ist Continuous Testing (CT) ein wichtiger aber nicht so bekannter Bestandteil. Testen wird traditionell meistens erst durchgeführt, wenn Features fertig entwickelt sind und gemergt werden sollen, oder vor dem Release einer neuen Software (-version). Continuous Testing dagegen verfolgt das Ziel, alle relevanten Testtypen wie Unit- und Systemtests so früh wie möglich und anschließend durchgehend über den gesamten Entwicklungszeitraum automatisiert durchzuführen. Durch die Integration von Tests bei jeder Änderung, sowie in allen Pipeline-Stufen werden Fehler häufiger und schneller aufgedeckt, was zu einer Steigerung der Codequalität führt [3]. Somit stellt Continuous Testing einen

wichtigen Baustein dar, um die Ziele von CI und CD hinsichtlich Qualität und Geschwindigkeit zu erreichen.

Gerade in der Zusammenarbeit mit langlebigen Feature-Branches ist die Verwendung von CI und CD essentiell, da diese Prinzipien dabei helfen, die Codebasen frühzeitig zusammenzuführen. Dadurch wird das Risiko durch spätes Zusammenführen minimiert und sonstige Wechselwirkungen oder Integrationsprobleme so früh wie möglich aufgedeckt, so dass sie rechtzeitig behoben werden können [3].

2.3 Herausforderungen bei langfristigen Feature-Branches

Der Einsatz von Feature-Branches bietet zahlreiche Vorteile, führt jedoch insbesondere bei lang laufenden Branches zu spezifischen Herausforderungen. Ein Beispiel dafür ist die Gefahr des Abdriftens des Hauptbranches von den Feature Branches was zu erheblichen Merge-Konflikten und damit Integrationsschwierigkeiten führen kann [8][9].

Hierdurch kann das Risiko entstehen, in die sogenannte „Integration Hell“ zu kommen, also eine Situation, in der Entwickler und Entwicklerinnen aufgrund langer Entwicklungszeiten viele Änderungen auf dem isoliertem Branch vornehmen und dadurch die Integration mit dem Hauptbranch zunehmend komplexer und aufwendiger werden. Die damit verbundenen Merge-Konflikte können aufgrund ihrer Komplexität den Entwicklungsprozess erheblich verzögern und die Fehleranfälligkeit erhöhen [6, Kap. 3].

Typische Ursachen von Merge-Konflikten bei langlebigen Feature-Branches sind:

- Divergierende Architektur- oder API-Änderungen, die im Feature-Branch genutzt werden,
- Parallele Anpassungen an den gleichen Dateien oder Codezeilen,
- Refaktorisierungen im Hauptbranch während der Feature-Entwicklung.

Eine empirische Untersuchung zu diesem Thema von Bird [8] zeigt auf, dass Merge-Konflikte nicht so häufig wie erwartet auftreten, aber wenn, dann ist die Lösung meistens zeitaufwändiger und fehleranfälliger als erwartet ist. Gerade bei lang laufenden Feature-Branches führen sie somit zu erhöhtem Aufwand.

Eine zentrale Herausforderung ist somit das Testen von Zwischenständen, damit noch in der Entwicklung befindliche Feature-Branches regelmäßig in automatisierte Tests einbezogen werden können. Wenn es eine automatisierte Integration, sowie Tests für alle relevanten Branches gibt, werden Wechselwirkungen frühzeitig erkannt und deren Behebung nimmt deutlich weniger Zeit in Anspruch [3]. Deshalb haben sich die folgenden Strategien für eine koordinierte Nutzung von Feature-Branches, sowie deren regelmäßiges Mergen durchgesetzt:

- Regelmäßiges Aktualisieren des Feature-Branches aus dem Hauptbranch (Mergen/Rebasen)
- Einheitliche Konventionen für Branch Bezeichnungen und Commit-Strukturen
- Automatisierte Überprüfung und Tests mittels CI/CD

Für die meisten Projekte hat sich gezeigt, dass regelmäßiges Integrieren, auch bekannt als „Merge early, merge often“[7] und der Einsatz automatisierter Tests entscheidende Punkte sind, um die negativen Effekte lang laufender Feature-Branches zu minimieren [7].

2.4 Überblick über gängige Versionsverwaltungssysteme

Um eine kontinuierliche Integration, sowie das frühzeitige Testen paralleler Branches realisieren zu können, sind Werkzeuge zur Versionsverwaltung eine grundlegende Voraussetzung. Gerade bei komplexen Projekten, in denen an vielen Feature-Branches gleichzeitig gearbeitet wird, müssen Entwickler und Entwicklerinnen die Möglichkeit haben, unterschiedliche Arbeitsstände effizient zu verwalten, sie zu verzweigen und wieder zu vereinen und die daraus entstehenden Zwischenstände für automatisierte Tests und Buildvorgänge zugänglich zu machen. Ohne den Einsatz eines Versionsverwaltungssystems wären Prinzipien wie CI unvorstellbar.

Versionsverwaltungssysteme (Version Control Systems, VCS) sind heutzutage unverzichtbare Werkzeuge in der professionellen Softwareentwicklung. Ihre Hauptaufgabe besteht darin, alle Änderungen am Quellcode systematisch zu erfassen und als wiederherstellbare Zustände auf mehreren Entwicklungssträngen nachvollziehbar abzuspeichern. Diese abgespeicherten Zustände können jederzeit wieder abgerufen werden. VCS sind damit das Fundament, um als Team gemeinsam an einem Softwareprodukt entwickeln zu können, aber auch für Qualitätssicherung, Fehlerdiagnosen und die Rückverfolgung von Änderungen [1, Kap. 1].

Ursprünglich wurden in den meisten Projekten zentrale Systeme wie das Concurrent Versions System (CVS), oder Apache Subversion (SVN) eingesetzt. Hauptbestandteil dieser zentralisierten Systeme ist ein Hauptrepository auf einem Server, mit dem alle Entwickler verbunden sind, um Änderungen durchzuführen. Der Vorteil zentraler Systeme ist eine einfache Topologie, welche jedoch bei größeren Teams,

in denen viele Entwickler parallel arbeiten, an ihre Grenze stößt. Das Entwickeln mit Branches ist oft schwerfällig, weil ein Entwickler lokal nur einen Zustand besitzt und seine Änderungen immer sofort mit dem Server teilen muss, bevor er sich einen anderen Zustand holen kann. Diese häufige Integration der vielen parallelen Entwicklungsstränge verursacht Konflikte und Bottlenecks, da nur ein Entwickler gleichzeitig Änderung an einer Datei hochladen kann und ein Serverausfall die komplette Entwicklung lahm legt. Diese Limitierungen führten dazu, dass ab Mitte der 2000er Jahre verteilte Systeme immer populärer wurden [1, Kap. 1].

Verteilte Versionsverwaltungssysteme (Distributed Version Control Systems, DVCS), wie Mercurial und Git zeichnet aus, dass jeder Entwickler eine vollständige lokale Kopie des Remote-Repositorys, einschließlich aller Branches und deren Historien besitzt. Dieser Aufbau ermöglicht es, nahezu alle Entwicklungs- und Verwaltungsaufgaben lokal durchzuführen. Erst wenn man seinen lokalen Stand mit anderen Entwicklern teilen will, wird eine Verbindung zum Server benötigt [1, Kap. 9].

Mercurial und Git sind die beiden bekanntesten DVCS und entstanden nahezu zeitgleich. Sie verfolgen beide ähnliche Grundprinzipen, unterscheiden sich jedoch in einigen technischen Details. Mercurial zeichnet eine sehr konsistente und vergleichsweise einfache Bedienung aus. Die Versionierung ist linear und für Einsteiger oft leichter nachvollziehbar als bei Git [1, Kap. 9]. Dennoch hat sich heutzutage Git weitestgehend durchgesetzt, sichtbar daran, dass die meisten großen Plattformen wie GitHub, oder GitLab ihren Fokus ausschließlich auf Git gelegt haben und für Git weltweit eine deutlich größere Community- und Dienstleistungslandschaft mit umfassenden Schulungen sowie Supportangebot existiert [1, Kap. 9].

Git wurde ursprünglich von Linus Torvalds für die Entwicklung des Linux-Kernels konzipiert, hat sich aber wenige Jahre später zum unumstrittenen Standard unter modernen Versionsverwaltungssystemen entwickelt. Es überzeugt insbesondere durch seine Leistungsfähigkeit, Skalierbarkeit und Flexibilität, durch die zentralen Konzepte wie das einfache und effiziente Anlegen und Verwalten von Branches sowie deren Zusammenführung und die Möglichkeit, komplexe Entwicklungshistorien nachzuvollziehen oder umzuschreiben [1, Kap. 1]. Ein weiterer Vorteil von Git ist seine hohe Anpassungsfähigkeit an unterschiedliche Entwicklungsprozesse. So kann es von einem einzelnen Hauptbranch, über komplexe Branching-Modelle wie GitFlow bis hin zu Trunk-Based Development oder Pull/Merge-Request-Workflows (erläutert in Unterabschnitt 3.2), wie bei GitHub und GitLab in der Softwareentwicklung eingesetzt werden [5][3].

2.4.1 Technische Funktionsweise und zentrale Begriffe von Git

Als verteiltes VCS stellt Git eine Vielzahl von Funktionen bereit, um ein möglichst effizientes paralleles und nachvollziehbares Arbeiten im Team zu ermöglichen. Die wichtigsten Grundbegriffe und Operationen, mit denen in der Buildpipeline gear-

beitet wird, werden im Folgenden näher erläutert:

Branch Ein Branch (Zweig) ist in Git ein leichtgewichtiger Zeiger auf einen bestimmten Zustand in der Entwicklung eines Projektes. Branches ermöglichen es, mehrere Dinge parallel zu entwickeln, wie zum Beispiel Features oder Bugfixes. Das Erstellen eines Branches, sowie das Wechseln auf einen anderen (checkout) sind in Git besonders performant, da keine komplett neue Projektkopie erstellt wird, sondern stattdessen lediglich Metadaten angepasst werden [1, Kap. 3]. Branching-Strategien wie ein Hauptbranch (main/master) oder Feature-Branches bieten eine klare Struktur in großen Projekten.

Commit Ein Commit speichert einen bestimmten Zustand des Projektes, auch Snapshot genannt. Zusätzlich zu dem Stand der Dateien enthält er eine kurze Beschreibung, Metadaten zum Autor und Zeitstempel, sowie einen Verweis auf den Vorgänger-Commit. Somit ergibt sich eine Verkettung aller Commits, was wiederum einen nachvollziehbaren Verlauf aller Änderungen im Projekt garantiert [1, Kap. 2].

Push Bei einem Push werden lokale Commits auf eine Remote Repository wie einem zentralen GitHub oder GitLab-Server übertragen. Erst durch das Pushen von Commits, werden diese für andere Entwickler im Team sichtbar und zugänglich gemacht. Um Änderungen pushen zu können, muss das lokale Repository, bzw. der Branch, auf dem die Änderungen gepusht werden sollen, auf dem selben Stand oder aktueller als das Remote Repository sein. Ist das nicht der Fall, muss vorher ein Merge oder Rebase durchgeführt werden. Beide Methoden werden in diesem Abschnitt ausführlich erläutert.

Pull Mit einem Pull wird der Branch, der aktuell lokal ausgecheckt ist, mit dem Remote Repository verglichen, alle dort vorhandenen Änderungen geladen und lokal integriert. Diese Operation setzt sich aus den zwei Schritten Fetch und Merge (oder Rebase) zusammen:

Fetch Bei einem Fetch werden alle neuen Änderungen aus dem Remote Repository lokal geladen, wie neue Commits oder entfernte Referenzen. Diese werden jedoch nicht in den aktuell ausgecheckten Branch integriert, so dass der Entwickler die Möglichkeit besitzt, vor dem eigentlichem Merge die Unterschiede gezielt zu analysieren.

Merge Beim Merge werden zwei Branches zusammengeführt. Hierzu analysiert Git die Historien beider Branches und kombiniert die Änderungen aller Dateien beider Seiten soweit wie möglich automatisch. Es können bei überschneidenden Änderungen (Änderungen in der gleichen Zeile) Merge-Konflikte auftreten. Diese muss der Entwickler manuell lösen, bevor der Merge abgeschlossen werden kann [8]. Ein erfolgreicher Merge wird als ein neuer Commit in der Historie abgespeichert. In Abbildung 2 ist ein typischer Ablauf eines neuen Feature-Branches, in diesem Fall `experiment` dargestellt, der vom Hauptbranch `experiment` abgezweigt und später wieder gemergt wird.

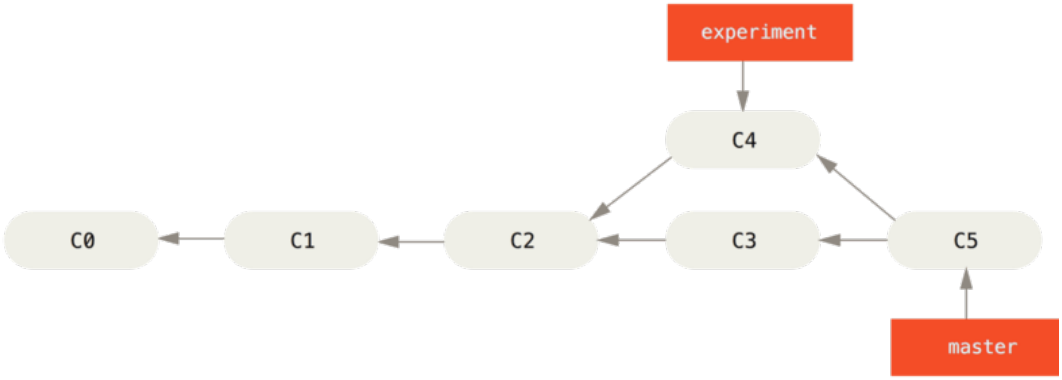


Abbildung 2: Schema eines Merge-Vorgangs [1, Kap. 3]

Rebase Eine Alternative zum Merge ist der Rebase. Hierbei werden alle Commits eines ausgewählten Basis-Branches so umgeschrieben, dass sie auf dem aktuellen Stand eines Ziel-Branches basieren. Technisch gesehen wird jeder Commit des Basis-Branches einzeln auf den neusten Stand des Ziel-Branches wiederholt ausgeführt. Dadurch entsteht eine lineare Historie, welche einfacher nachzuvollziehen ist, aber die Commit-Historie wird dabei verändert, weshalb Rebasen nur auf Branches durchgeführten werden sollte, von denen kein weiterer Branch abgezweigt wurde. Auch hierbei können Merge-Konflikte auftreten, welche der Entwickler lösen muss. [1, Kap. 3]. Durch das Verändern der Historie können diese Änderungen nicht einfach gepusht werden, sondern es wird ein Force Push benötigt. Dieser erzwingt das Überschreiben des Remote-Branches mit den lokalen Änderungen, wodurch Commits, die remote, aber nicht lokal vorhanden sind, verloren gehen.

Weitere relevante Begriffe sind die `Staging Area` oder `Labels`. Die Staging Area ist ein lokaler temporärer Bereich, in dem Änderungen vor dem Commit explizit übernommen werden. So kann der Entwickler entscheiden, ob er alle Änderungen oder nur bestimmte Teile committet. Mithilfe von Labels können Commits markiert werden, indem ihnen selbst angelegte Eigenschaften für Versionen oder spezielle Varianten hinzugefügt werden. Folgendes Codebeispiel stellt die beschriebenen Git Operationen praktisch dar. Es wird in einem Remote-Repository mit dem lokalen

Standardnamen `origin` gearbeitet. Die Befehle sind beispielhaft für die wichtigsten Schritte im Lebenszyklus einer Änderung:

Listing 1: Feature-Branch-Workflow

```
#Hauptbranch wechseln und aktualisieren
git checkout main
git pull

#Neuen Feature-Branch erstellen und in diesen wechseln
git checkout -b feature/neues-feature

#Entwicklung und schrittweises Committen
git add .
git commit -m "Teilfunktionalitt implementiert"

#Zwischendurch den Feature-Branch regelmaeig mit main abgleichen
git fetch origin
git rebase origin/main # Alternative: git merge origin/main

#Nach Abschluss den Feature-Branch in main mergen
git checkout main
git merge feature/neues-feature

#Aenderungen zum Remote-Repository pushen
git push
```

Während dieses Ablaufs werden iterativ Commits erstellt, Branches verwaltet und regelmäßig mit dem Hauptbranch synchronisiert, sodass beim Zusammenführen (Merge) oder Rebasen auftretende Konflikte kontinuierlich und nicht erst am Ende der Entwicklung gelöst werden. Durch diese klare und einheitliche Gliederung der Kernoperationen sowie die damit unterstützten Workflows trägt Git maßgeblich zur Produktivität, Transparenz und Fehlerrobustheit moderner Softwareentwicklung bei [1, Kap. 9][8]. Mit diesen Möglichkeiten der verteilten Versionskontrolle sind die technischen Voraussetzungen für die parallele Entwicklung auf Feature-Branches und dessen kontinuierliche Integration geschaffen. Dennoch stoßen größere Projekte mit vielen Feature-Branches und teilweise langen Entwicklungszeiten auf eine neue Herausforderung: Neben der Verwaltung der einzelnen Branches, muss ein Verfahren gefunden werden, wie mehrere parallele Branches regelmäßig und automatisiert zu einem testbaren Integrations-Build zusammengeführt werden können.

3 Stand der Technik

In diesem Kapitel werden die herausgearbeiteten theoretischen Grundlagen aus dem vorherigen Kapitel auf bestehende wissenschaftliche und industrielle Ansätze zur kontinuierlichen Integration von Feature-Branches angewendet.

3.1 Problem und wissenschaftliche Relevanz

Die zunehmende Komplexität moderner Softwareprojekte führt dazu, dass es nicht mehr ausreicht, Features einzeln zu testen, sondern es sind in regelmäßigen Abständen zahlreiche parallele Branches gemeinsam zu einem Build zusammenzuführen und umfassend zu testen [3][10]. Dadurch wird sichergestellt, dass mögliche Komplikationen und unerwünschte Nebeneffekte verschiedener Features frühzeitig erkannt und gelöst werden können [8]. Ein reines individuelles Integrieren jedes Feature-Branches mit dem aktuellen Stand des Hauptbranches reicht somit häufig nicht mehr aus, da dabei jegliche Wechselwirkungen durch Überschneidungen mit anderen Feature-Branches nicht getestet werden können.

Werden diese Risiken ignoriert, besteht die Gefahr, dass erst beim finalen Merge auffallende Fehler erheblich mehr Korrektur- und somit Kostenaufwand verursachen, als frühzeitig gefundene Fehler. Diese Abhängigkeit konnte bereits von Barry Boehm mit seiner `Cost of Change Curve` empirisch nachgewiesen werden, welche besagt, dass der Aufwand zum Beheben von Fehlern exponentiell ansteigt, je später diese im Entwicklungsprozess aufgedeckt werden [11]. Um das zu vermeiden, benötigt es spezielle Strategien und technische Lösungen, die eine kontinuierliche, automatisierte Zusammenführung mehrerer Feature-Branches zuverlässig ermöglichen [12][9].

In diesem Kapitel werden wissenschaftliche Ansätze, Arbeitsprozesse und aktuelle Werkzeuge für die sogenannte Multi-Branch-Integration vorgestellt und verglichen. Der Schwerpunkt hierbei liegt insbesondere auf den Methoden, die ein automatisiertes Zusammenführen und anschließendes Testen von vielen Feature-Branches ermöglichen und somit den Entwicklungsprozess und die Qualitätssicherung unterstützen, dabei aber den Entwicklungsfluss nicht unnötig belasten.

3.2 Bestehende Werkzeuge und Ansätze

Die am weitesten verbreiteten Branching-Strategien sind die Workflows Git-Flow [4] und GitHub Flow sowie das Trunk-Based Development. Alle drei sind Ansätze, um möglichst effizient mit vielen Branches entwickeln zu können [5].

Hierfür werden beim Git-Flow zahlreiche langlebige Feature- Release- und Hotfix-Branches eingesetzt. Wenn diese Branches nicht zeitnah gemergt werden, also eine längere Entwicklungsdauer besitzen, kann es zu Integrationsproblemen kommen, was insbesondere passiert, wenn Feature-Branches über Wochen nicht mit dem Hauptbranch synchronisiert werden.

Beim GitHub Flow wird dieses Problem durch kurzlebige Feature-Branches gelöst. Diese werden möglichst schnell mithilfe von Pull Requests in dem Hauptbranch integriert. Das führt zu weniger Merge-Konflikten, ist aber mit vielen gleichzeitigen Features und komplexen Abhängigkeiten schwer umzusetzen. Gerade wenn nicht alle Branches sofort in den Hauptbranch integriert werden können, weil sie zum Beispiel erst in einer späteren Version ausgeliefert werden sollen, oder bei komplexen Features, die eine längere Entwicklungszeit benötigen, kommt es zu Problemen.

Trunk-Based Development versucht die Anzahl und Lebensdauer von Branches möglichst gering zu halten. Der Workflow sieht vor, dass Entwickler ihre Änderungen möglichst häufig und früh in den Trunk (Hauptbranch) integrieren. Auch diese Methode reduziert Merge-Konflikte und fördert automatisiertes Testen durch eine frühe Integration der Feature-Branches. Bei vielen unfertigen Feature Branches, welche trotzdem so früh wie möglich integriert werden sollen, ist die Fehlergefahr gerade in großen Teams stark erhöht, was zu vermehrten Fehlerzuständen der Software und einer gesenkten Softwarequalität führen kann [5].

Die Entwicklung anhand von Workflows ist bei der Benutzung von Feature Branches essentiell und wird durch moderne Git-Plattformen wie GitHub, GitLab oder Bitbucket unterstützt. Diese Plattformen stellen Tools für Pull/Merge-Requests, Code-Reviews und automatisierte Pipeline-Infrastruktur bereit. Jedoch bleibt das Problem bestehen, dass immer nur einzelne Branches gemergt und getestet werden können, während sich diese in der Entwicklungsphase befinden. Somit bleiben die Wechselwirkungen vieler paralleler Feature Branches ungetestet, solange nicht alle zu testenden Branches in einen gemeinsamen Build gemergt werden können.

3.2.1 Automatisierte Pre-Integration und Merge-Queues

Eine mögliche erste Erweiterung des Workflows, sind Pre-Integration-Pipelines und Merge-Queues, wie sie z.B. von GitHub Merge-Queue, oder GitLab Merge-Train angeboten werden [13]. Hierbei werden alle Merge-Requests an den Hauptbranch als erstes in eine Warteschlange aufgenommen und anschließend nacheinander in einer isolierten Umgebung mit dem Hauptbranch gemergt. Für jeden Fall werden abschließend Unit- und Integrationstests durchgeführt.

Der Workflow ist wie folgt aufgebaut: Alle Feature-Branches, die den Master als Basis haben, werden nach der fertigen Entwicklung in eine Merge-Queue eingereiht. In der Abbildung 3 sind das die Branches 1 - 3. Sie müssen jedoch nicht den gleichen Commit als Ausgangslage besitzen. Wenn die Entwicklung eines Branches abgeschlossen ist, rebaset der Entwickler, um die Merge-Konflikte mit dem Master zu lösen. Anschließend reiht er mit dem Starten des Merges den Branch in die Merge-Queue ein. Hier werden Integrations- und Softwaretests durchgeführt, um die Wechselwirkungen der neuen Feature-Branches untereinander zu testen. Wenn dabei ein Merge fehlschlägt, egal ob wegen Build- oder Konfliktfehlern, wird dieser Branch abgewiesen wie beim zweiten Branch in der Abbildung 3 oder wieder an das Ende der Queue versetzt. Nur Änderungen, die ohne fehlgeschlagene Merges und Tests durchgelaufen sind, werden direkt danach in den Hauptbranch (Master) gemergt, bevor der nächste Branch getestet wird, in der Abbildung 3 der erste und dritte Branch. Dieses Verfahren wird angewendet, um Integrationsprobleme und Konflikte der Feature-Branches untereinander aufzudecken und erhöht damit die Build Stabilität [13]. Es werden jedoch keine Integrationsprobleme von Feature-Branches untereinander während der Entwicklungsphase festgestellt.

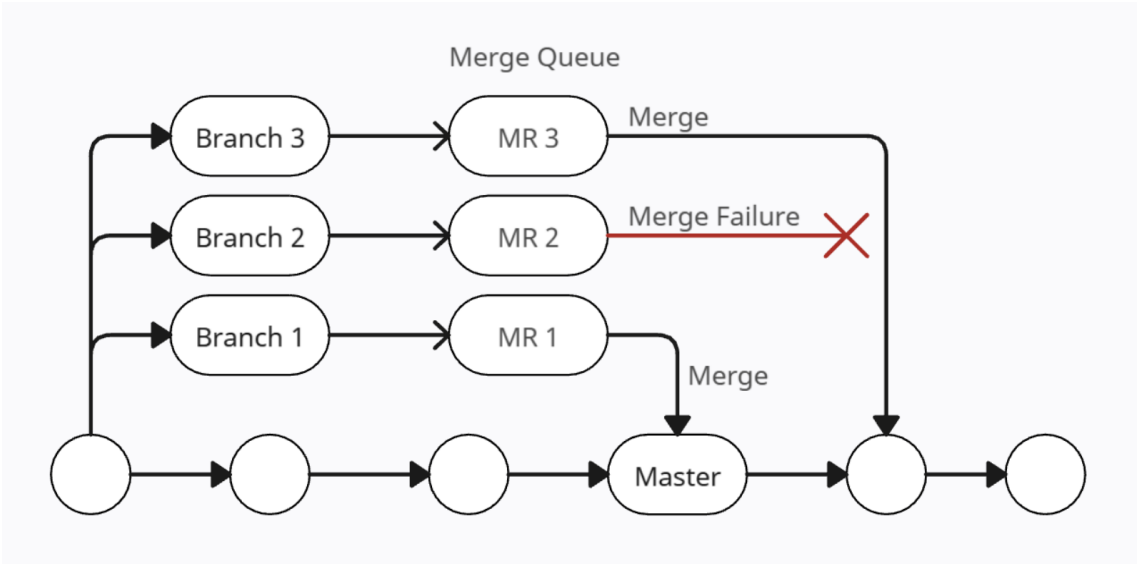


Abbildung 3: Abläufe einer Merge Queue, MR = Merge-Request

Mögliche Weiterentwicklungen erlauben das Kombinieren von mehreren Branches im Batch, was die Effizienz steigert und Wechselwirkungen aufdecken kann, jedoch die Fehlerzuordnung erschwert.

3.2.2 Batch-Merge und Staging-Branches

Eine Merge-Queue wird in vielen modernen Entwicklungs-Workflows bereits eingesetzt, sodass Branches ohne das erfolgreiche Durchlaufen von Unit-Tests, oder Fehlern beim Build nicht gemergt werden. Um jedoch auch schon während der Entwicklungsphase eines Feature-Branchs Test- und Buildvorgänge zu überprüfen, werden häufig Batch-Merge-Verfahren angewendet. Bei ihnen werden in regelmäßigen Abständen auf temporären Staging Branches, alle Features für die nächste Build-Iteration wie Release oder Nightly Build zusammengeführt [10]. Diese Builds können anschließend automatisiert getestet werden, um Fehlerzustände aufzudecken und die Wechselwirkungen der einzelnen integrierten Branches zu testen.

Wenn der Buildprozess mit sämtlichen zugehörigen Branches erfolgreich ist und die nachfolgenden Tests abgeschlossen sind, kann mit dem neuen Build weitergearbeitet werden. Es werden entweder einzelne Branches in den Hauptbranch übernommen oder sogar alle Änderungen, je nachdem wie der eingesetzte Workflow abläuft und ob die integrierten Branches keine Fehlerzustände mehr besitzen. Bei diesem Ansatz können Inkompatibilitäten zwischen mehreren Branches frühzeitig aufgedeckt werden, es können jedoch längere Feedbackzeiten auftreten, verursacht durch komplexere Buildvorgänge oder das Fehlschlagen eines Builds durch einen fehlerhaften Branch. Dennoch eignet sich dieser Ansatz, besonders wenn es bei der Entwicklung der Software mehrere verschiedene Release-Versionen für unterschiedliche Kunden oder fest definierte regelmäßige Testabläufe gibt [10].

3.2.3 Sandbox-Integration

Noch flexibler ist die Sandbox-Integration, bei der auf Anfrage eines Entwicklers oder automatisiert im Hintergrund für jede relevante Kombination aus Hauptbranch und mehreren Feature-Branches temporäre Integrationsumgebungen (sogenannte `Sandboxes`) erstellt werden. Das bedeutet, dass die gewünschten Branches in einen neuen Integration-Branch, basierend auf dem Hauptbranch gemergt und anschließend zu einem neuen funktionsfähigen Build zusammengebaut werden. Mithilfe der Sandboxes können Builds und Tests unabhängig vom Hauptbranch ausgeführt werden [3]. So können sowohl gezielte Problemkonstellationen als auch beliebige Feature-Sets je nach Bedarf getestet werden, beispielsweise wenn Teams an voneinander abhängigen Funktionalitäten entwickeln.

Dieses Konzept kann auch als so genannte `Bouquet-Integration` bezeichnet werden: Bei ihr gibt es zum Start eine beliebig große Menge an Blumen; angewandt auf die Softwareentwicklung sind die Blumen alle aktuell comitteten Feature-Branches (und Bugfix-Branches). Aus dieser Menge wird nun ein neuer Blumenstrauß (Bouquet) gebildet, was bedeutet, dass alle relevanten Feature-Branches ermittelt und auf einen neuen Integration-Branch, der auf den Hauptbranch basiert. gemergt werden. Das Endprodukt ist ein neuer alleinstehender Integration-Branch.

Die Umsetzung dieses Ansatzes ist jedoch aufwendig, da es zu erheblichen Infrastruktur und Wartungsaufwänden kommen kann, gerade wenn es sehr viele Kombinationen zu bauender Branches gibt und somit sehr viele verschiedene Branches und Builds durchgeführt werden müssen [3]. Der größte Aufwand ist jedoch die Entwicklung der Build-Pipeline und das Aufsetzten der Infrastruktur. Wenn diese Punkte zuverlässig funktionieren, ist der Wartungsaufwand im Vergleich zum Nutzen vergleichsweise gering [6, Kap. 4].

3.2.4 Automatisierte Konfliktlösung

Ein zentrales Problem bei allen genannten Ansätzen, welche mehrere Branches zu einem neuen zusammenführen, sind potentielle Merge-Konflikte. Sie entstehen, wenn Änderungen aus unterschiedlichen Branches die gleichen Zeilen Code betreffen oder Änderungen an Daten, Build, oder Infrastruktur vornehmen, ohne dass alle erforderlichen Anpassungen auf sämtlichen Feature-Branches vorgenommen wurden [9]. Git selbst besitzt hierfür ein System, welches auf einen dreifachen Merge zwischen Basis-, Entwicklungs- und Zielbranch aufbaut und auf Zeilendifferenzen basiert. Moderne Merge Verfahren ergänzen diesen Ablauf durch semantische Analysen, auch `Semantic Merge` genannt, bei denen Rollen und Kontext von Änderungen in der Programmlogik erkannt und berücksichtigt werden [8].

Während traditionelle Merge-Verfahren rein textbasiert sind, werden diese mithilfe von Semantic Merge um die Fähigkeit erweitert, auch syntaktische und semantische Strukturen des Programmcodes zu analysieren. Anstatt lediglich Zeilenvergleiche durchzuführen, wird als Erstes ein Abstract Syntax Tree (AST) der zu vereinigenden Codebasen generiert.

Listing 2: einfache Hello World Funktion

```
def main():
    print("Hello, world!")

main()
```

Der aus dieser einfachen Hello-World-Funktion erstellte Abstract Syntax Tree ist in der Abbildung 4 dargestellt. Der sogenannte Wurzelknoten ist `Modul`, er repräsentiert die gesamte Python-Datei. Von ihm direkt ausgehend sind die Funktion main und der Aufruf (`call`) dieser. Der Aufruf setzt sich zusammen aus dem Namen der aufgerufenen Funktion sowie dem übergebenen Argument, in diesem Fall `none` (keine). Die Funktion besitzt einen `body`, in dem alle Schritte enthalten sind, die beim Aufruf der Funktion ausgeführt werden und in `args` stehen die Argumente, die diese Funktion beim Aufruf einliest. Im `body` ist der Aufruf (`call`) der Funktion `print`, mit der Konstante `Hello,World!` als Argument.

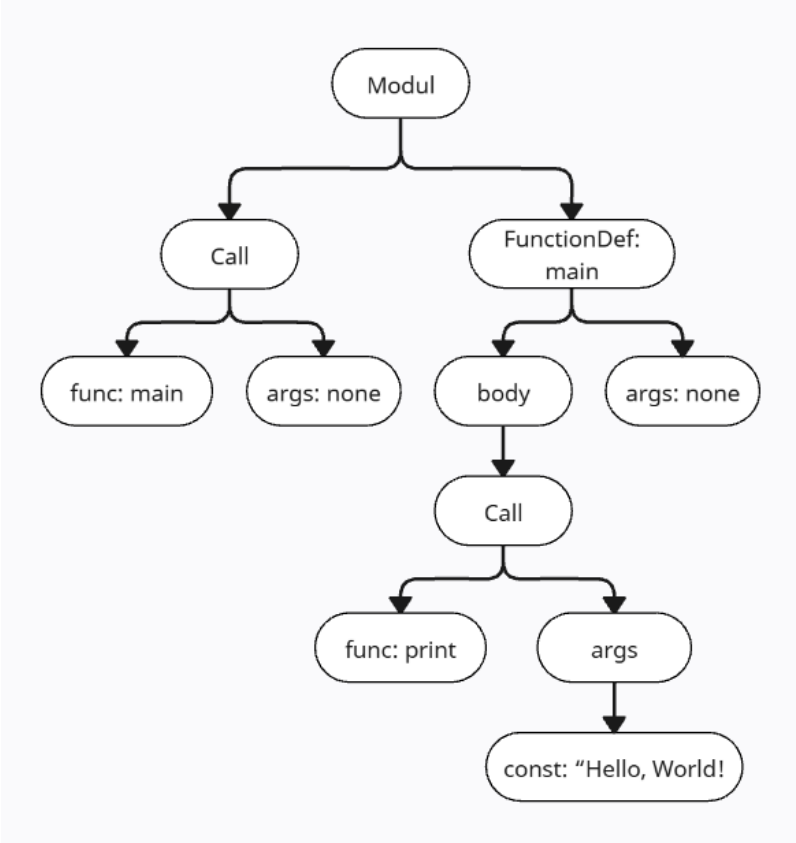


Abbildung 4: Abstract Syntax Tree für Listing 2

Wie in Abbildung 4 sichtbar besitzt der AST eine Baumstruktur, worüber Verschiebungen von Funktionen, deren Umbenennungen oder andere semantische Änderungen erkannt und zusammengeführt werden. Dadurch können mehr Merge-Konflikte wahrscheinlicher automatisch gelöst werden und wenn es trotzdem noch fehlschlägt, werden sie besser aufbereitet für das anschließende manuelle Lösen durch einen Entwickler. Folglich werden Fehlermeldungen und manueller Aufwand reduziert und die Integrationssicherheit gesteigert [8].

3.3 Umsetzungen in der Praxis

Nach der Betrachtung der theoretischen Ansätze und Grundlagen zur Integration langfristiger Feature-Branches wird im folgenden Abschnitt ein Überblick über

praktische Umsetzungen in der Industrie, sowie in Open-Source-Projekten gegeben. Anhand ausgewählter Beispiele wird dargestellt, welche konkreten Prozesse, Tools und Abläufe sich etabliert haben, um die kontinuierliche Integration vieler paralleler Entwicklungszweige zu ermöglichen.

3.3.1 Meta: Pre-Merge Testing in skalierbaren Monorepos

Meta besitzt für seine Produkte wie Facebook, Instagram oder WhatsApp eine der weltweit größten und aktivsten Codebasen im Monorepo-Stil, was bedeutet, dass der gesamte Code in einem Repository liegt und nicht auf mehrere aufgeteilt wird. Damit die mehr als tausend Entwickler gemeinsam arbeiten können, verwendet Meta kurze Zyklen von Feature-Branches. Diese werden jedoch nach Abschluss nicht sofort in den Hauptbranch gemerged, sondern durchlaufen eine automatisierte Pre-Merge-Testing Pipeline mit komplexer Sandbox-Infrastruktur, um die Softwarequalität sicherzustellen.

Neue Änderungen werden zunächst als Diffs vorgeschlagen und gelangen in eine Warteschlange (Merge-Queue), in der sie anschließend im Meta internen Build-System `Sandcastle` isoliert getestet werden. Hierbei werden alle zuvor eingereichten Änderungen berücksichtigt, auch wenn sie noch nicht im Hauptbranch integriert sind und alle zusammen in einer eigens dafür generierten Sandbox getestet. Erst nachdem alle auftretenden Merge-Konflikte zwischen den neuen Änderungen und den gestapelten Patches - also den vorherigen aufeinander aufbauenden Commits - gelöst sind und die Tests erfolgreich sind, wird die jeweilige Änderung in den Hauptbranch übernommen. Damit werden mögliche unerwünschte Wechselwirkungen und Integrationsprobleme schon vor dem Merge erkannt und ausgeschlossen [14]. Die Feature-Branches werden jedoch erst vor dem Merge getestet und nicht schon während der Entwicklung, weshalb nur mit kurzlebigen Branches gearbeitet wird.

3.3.2 Google: Feature Flags und Pre-Submit-Builds

Google verwendet Trunk-Based-Development als Entwicklungsmodell, kombiniert mit kurzlebigen Feature Branches. Entwicklungen, die eine längere Zeit benötigen, werden über Feature-Flags schrittweise im Code aktiviert. Dadurch werden komplexe Merge-Konflikte stark vereinfacht, neue Features aber trotzdem möglichst frühzeitig auf den Hauptbranch integriert. Um die Releases vor dem Veröffentlichen des Features nicht zu beeinträchtigen, werden diese vor Fertigstellung deaktiviert ausgeliefert [15].

Für jede gepushte Änderung wird ein Pre-Submit-Build ausgelöst, dafür gibt es interne Tools wie `Piper` und `Copybara`. Diese mergen automatisiert die neuen Änderungen in einer temporären Umgebung mit dem Hauptbranch und testen sie anschließend. Die zugehörige Buildpipeline ist sehr komplex, aber komplett in den Entwicklungs-Workflow integriert, ohne Mehraufwand für die Entwickler. Dieses

Verfahren hilft, Integrationsprobleme frühzeitig zu erkennen, jedoch nur für einzelne Branches [15].

3.3.3 Open-Source-Projekte

In großen Open-Source-Projekten wie zum Beispiel Kubernetes ist die Integration vieler konkurrierender Branches eine Herausforderung. Es wird exemplarisch der Workflow von Kubernetes betrachtet. Hier wird für jeden Pull-Request mithilfe des Tools `Prow` automatisiert eine Sandbox Umgebung erstellt [16]. Wenn mehrere Pull-Requests gleichzeitig offen sind, kommt eine Merge-Queue zum Einsatz. Mit Hilfe des Tools `bors-ng` werden alle Pull Requests in eine Reihe gebracht und einzeln oder sogar im Batch in einem Staging-Bereich (Testumgebung) zusammengebaut und getestet. Erst wenn die Tests erfolgreich sind, werden die Änderungen automatisch auf den Hauptbranch übertragen. Dadurch wird verhindert, dass ein anderer Entwickler einen fehlerhaften Stand des Projekts bekommt, wenn er sich alle neuen Änderungen fetcht [17]. Des Weiteren wird `Zuul CI` eingesetzt, welches ermöglicht, dass mehrere Branches gemeinsam getestet werden können.

Zuul ist ein Open-Source-System für die Umsetzung von CI, welches von der OpenStack-Community entwickelt wurde. Es ist speziell für das Testen und Integrieren paralleler Branches in großen verteilten Projekten geeignet. Im Gegensatz zu den vorherigen Systemen kann Zuul mehrere Branches parallel in einem Build testen, indem es sie zu sogenannten `depend-on-chains` zusammenführt. Das bedeutet, dass es alle zu integrierenden Änderungen in temporären Umgebungen zusammenführt und zu einem Build kombiniert. Ein wichtiger Bestandteil davon ist eine `Gating Pipeline`, die wie eine Merge-Queue funktioniert und alle relevanten Branches auf den aktuellen Hauptbranch nacheinander mergt, auch wenn diese noch nicht final gemergt sind und sich noch in der Entwicklung befinden, aber mit getestet werden sollen. Das ermöglicht nicht nur, die Integration einzelner Branches zu testen, sondern auch zwischen mehreren Branches und sogar Projekten [18].

3.3.4 Cloudbasierte Pipelines

Die meisten großen Plattformen für Versionsverwaltungen wie zum Beispiel GitHub besitzen heutzutage Features, um CI und CD zu ermöglichen oder unterstützen. In GitHub gibt es hierfür das Automatisierungs- und CI/CD-System GitHub Actions. Es bietet Funktionen wie das automatisierte Integrieren von Branches in Testumgebungen, diese müssen jedoch von den Benutzern eigenständig definiert werden. Eine realisierbare Funktion ist zum Beispiel, dass für jeden Feature-Branch oder Pull-Request automatisch eine Testumgebung erzeugt wird, in der Buildprozesse und Softwaretests durchgeführt werden. GitHub Actions stellt hierbei die Schnittstellen für Server, Buildpipelines und Tests bereit, um diese einfach mit dem Git Repository zu verbinden. Sie müssen jedoch vom Nutzer selbst bereitgestellt und entwickelt werden [19]. Bei anderen Plattformen wie Azure Pipelines oder Bitbucket ist eine ähnliche Infrastruktur vorhanden [20][21].

3.4 Herausforderungen der praktischen Umsetzung

Bei den vorgestellten praktischen Umsetzungen wird ersichtlich, dass trotz der fast identischen Zielsetzung die Ausgestaltung der Prozesse je nach Unternehmensgröße, Infrastruktur und den schon vorhandenen Abläufen deutlich variiert. Der Implementierungsaufwand der Buildpipelines nimmt mit der Menge der gewünschten Funktionen an die Pipeline zu. Gerade das Testen der Wechselwirkungen paralleler Branches untereinander ist sehr komplex und steigt mit der Anzahl der Feature Branches exponentiell an, wenn alle theoretisch denkbaren Kombinationen getestet werden sollen [9].

Deswegen werden in der Praxis meistens nur die derzeit im Review befindlichen Branches getestet und auch erst vor dem Merge mit dem Hauptbranch. Hierbei wird versucht, möglichst viel zu automatisieren. Unterstützt durch Tools wie `Jenkins`, `Zuul`, oder `GitHub Actions`, welche heutzutage flexible APIs und Konfigurationen bereitstellen, ist das auch zu großen Teilen möglich. Die Integration langfristiger Feature-Branches ist jedoch in den meisten Softwareprojekten nicht möglich und wenn, dann mit sehr viel Aufwand verbunden. Damit ist das frühzeitige Erkennen und Beheben von unerwünschten Wechselwirkungen solcher Branches ein Problem, für das bislang keine vollständig automatisierte Lösung bereit steht [9]. Als möglicher Lösungsansatz wurde in der Firma Quality First Software GmbH eine Buildpipeline entwickelt, die auf dem Prinzip der Bouquet-Integration basiert, die im folgenden Kapitel vorgestellt wird.

4 Dokumentation der Buildpipeline

Die in der Firma QFS eingesetzte Buildpipeline ist in dieser Form (soweit bekannt) bislang einzigartig. Sie bietet erstmals eine Möglichkeit, die Softwareentwicklung mit Feature-Branches, Continuos-Integration, -Development und -Testing vollständig miteinander zu verbinden. Die Grundlage bildet das verteilte Versionsverwaltungssystem Git zusammen mit der Plattform GitLab und diversen Buildskripten. Diese Bestandteile und deren Zusammenhänge werden im folgenden Kapitel dokumentiert.

4.1 Methodik der Dokumentation

Um die Buildpipeline dokumentieren zu können, wurden in dem GitLab-Projekt des Unternehmens QFS mehrere Test-Branches und ein neuer Test-Integration-Branch erstellt. Diese Branches wurden anschließend mit der Buildpipeline zu dem neuen Integration-Branch gemergt. Hierbei wurden bewusst Merge-Konflikte erzeugt, damit sowohl das erfolgreiche Mergen von Branches ohne Merge-Konflikte, als auch die Verfahrensweise beim Auftreten dieser analysiert werden konnte. Durch diese Vorgehensweise konnte die Buildpipeline in ihrer Funktionsweise geprüft und die Abläufe parallel dazu dokumentiert werden.

4.2 Überblick über die Eigenentwicklung

Die Entwicklungsabläufe der Firma QFS orientieren sich am GitLab Workflow, der viele Elemente aus dem GitHub Flow übernimmt, jedoch einige signifikante Unterschiede aufweist. Ein zentraler Unterschied ist das Feature-driven Development, das heißt das funktionsorientierte Entwickeln auf einzelnen Feature-Branches. Für jede neue Funktion oder Bugfix wird ein neuer Entwicklungs-Branch angelegt, auf welchem entwickelt wird. Nach Abschluss der Entwicklungen wird ein Merge-Request erstellt, vergleichbar mit dem Pull-Request bei GitHub. Mithilfe von Merge-Requests werden die Änderungen des Feature-Branchs mit dem Hauptbranch zusammengeführt. GitLab besitzt eine integrierte CI/CD-Pipeline, in diese können jegliche Buildpipelines integriert werden oder es wird der Auto-DevOps Modus genutzt, der automatisiert ein Build, sowie einige Test durchführen kann, um mögliche Fehler frühzeitig aufzudecken. Letzterer funktioniert ohne Anpassungen jedoch meistens nur in kleineren Standardprojekten [22].

Für die Entwicklung bei QFS wurde der GitLab-Workflow wie folgt angepasst: Für jede neue Funktion, Bugfix oder sonstige Änderung wird ein Ticket in der Projektmanagement-Software `Jira` angelegt. Passend zum Titel und zur Nummer des Tickets wird anschließend ein neuer Branch erstellt, auf welchem die gesamte Entwicklung für das Ticket stattfinden soll. Der Merge-Request für diesen Feature-

Branch wird jedoch nicht erst beim Mergen des Branches, sondern direkt am Anfang mit erstellt. Diesem Merge-Request werden GitLab-Labels mit den zugehörigen Versionen hinzugefügt. Anhand dieser Labels werden später die Feature-Branches für einen Build ermittelt. Schon während der Entwicklung wird der Branch in täglichen Builds mit gebaut und dementsprechend auch getestet. Die Testergebnisse werden dem jeweiligen Entwickler in `Confluence`, einer Wissensdatenbank für Teams und Unternehmen, bereitgestellt. Wenn die Entwicklung abgeschlossen ist, wird von einem weiteren Entwickler ein Code-Review durchgeführt und die Funktionen getestet. Sobald dabei keine Fehler oder Unstimmigkeiten gefunden werden, wird der Branch in den zugehörigen Hauptbranch gemergt.

4.3 Technischer Aufbau der Infrastruktur

Die gesamte CI-Pipeline wird aktuell auf fünf dedizierten Servern betrieben, die jeweils unterschiedliche Aufgaben übernehmen. Die Funktionen und Verbindungen der einzelnen Server sind in Abbildung 5 dargestellt. Ein Entwickler, der einen neuen Build startet, interagiert hierbei ausschließlich mit dem Hauptserver namens pp20. Dieser fungiert als zentrale Schnittstelle zwischen der CI-Pipeline und den Entwicklern. Von hier aus können sowohl manuelle Builds als auch regelmäßig ausgeführte Nachtbuilds gestartet werden. Beim Starten eines neuen Builds lassen sich verschiedene Parameter angeben, welche den Ablauf sowie das Ergebnis der Buildpipeline beeinflussen, vgl. Unterabschnitt 4.5.

Vom Hauptserver aus wird der Start-Befehl direkt an den Buildserver (sdev) weitergereicht. Auf diesem Server befinden sich die Buildskripte der Buildpipeline, auf ihnen liegt der Hauptfokus in dieser Arbeit. Der Buildprozess läuft wie folgt ab: Zuerst wird für komplexe Builds wie „Maintenance-“ oder „Medium-Versionen“, die dass Mergen mehrerer Branches erfordern, ein neuer Branch erstellt, auf dem die einzelnen Branches zusammengeführt werden. Soll nur ein einzelner Branch gebaut werden, ist dieser Schritt nicht nötig. Anschließend wird ein neuer Build für die gewünschte Version oder den Branch erstellt. Das Ergebnis dieses Prozesses ist ein neuer funktionsfähiger Build, welcher auf dem Fileserver mit einem layered Filesystem abgespeichert wird.

Wurde der Buildprozess mit einem zusätzlichem Testparameter gestartet - was der Standard beim initialen Kommando buildAndTest ist - so initiiert der Hauptserver nach Abschluss der Buildprozesse automatisierte Tests auf einer zuvor definierten Anzahl an Testmaschinen. Dabei koordiniert der Hauptserver sowohl das Vorbereiten der Testumgebung, als auch das Durchführen der Tests und das Verarbeiten der Ergebnisse. Die Testmaschinen bekommen die zu testende Version der Software und auch die aktuellsten Versionen der Tests vom Fileserver automatisch gesynct.

Besitzt der neue Build ein bestimmtes Attribut, beginnt der Package-Server mit dem Erstellen der Installer-Pakete für Windows, Linux und macOS. Sobald dieser

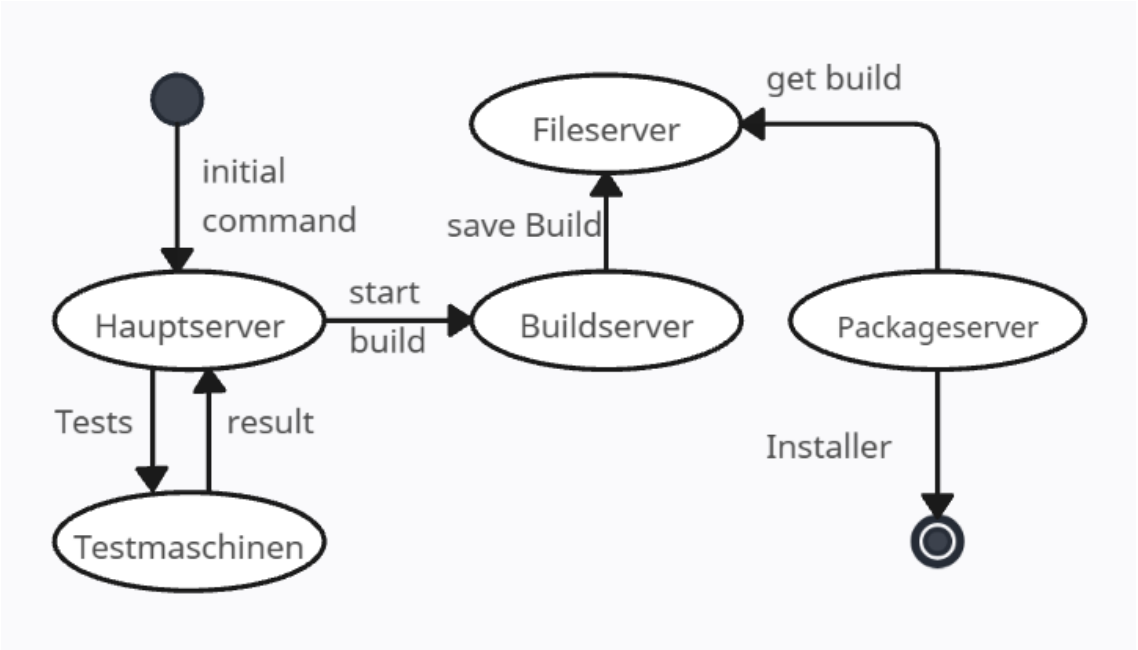


Abbildung 5: Aufbau der Infrastruktur

Prozess abgeschlossen ist, sind die Installer-Pakete für alle Entwickler auf einem Dashboard zum Download abrufbar. Die neueste Version des Branches ist auch im Remote-Repository verfügbar, welches wie im folgenden Abschnitt beschrieben aufgebaut ist.

4.3.1 Struktur des Git-Repositorys

Das zentrale Remote-Repository trägt den Namen `nprj`. Es enthält alle Entwicklungs-Branches sowie Branches, die bestimmte Zustände der Software dokumentieren wie Releases oder Zustände aktueller Versionen. Strukturiert sind sie in mehrere Kategorien, für die Pipeline relevant sind folgende:

- master - der aktuelle Hauptbranch,
- /dev - enthält alle aktiven Entwicklungs-Branches,
- /integration - dient zur Ablage aktueller Versionen, z.B. Maintenance, Medium oder Milestone.
- /releases - enthält alle in der Vergangenheit releasten Branches
- /maintenance - enthält alle Maintenance-Änderungen für alle releasten Versionen

Zu jedem Entwicklungs-Branch im Dev-Verzeichnis gibt es einen zugehörigen Merge-Request. Diesem Merge-Request können Labels hinzugefügt werden, welche im Repository definiert und verwaltet werden. In diesem Repository gibt es die Label: `Maintenance`, `Medium` sowie Bezeichner diverser Engines und Editionen, die

für bestimmte Kunden benötigt werden. Während des Buildprozesses wird anhand dieser Labels automatisch ermittelt, welche Branches für einen Integration-Build relevant sind, damit diese zu einer lauffähigen Anwendung zusammengeführt werden.

4.4 Abläufe in den Buildskripten

Der gesamte Buildprozess findet auf dem Buildserver `sdev` statt und arbeitet eng mit dem Remote-Repository `nprj` zusammen. Dieser Ablauf ist detailliert und in Abhängigkeit mit der Zusammenarbeit des Remote-Repositorys, als Sequenzdiagramm im Anhang D dargestellt.
Als erstes wird das Logging vorbereitet, welches alle ausgeführten Git-Befehle und sonstige Aktionen loggt, sodass der Buildprozess bei unerwünschten Ergebnissen leicht nachvollzogen und potentielle Fehler schnell gefunden werden können. Anschließend werden die Parameter, die vom Hauptserver übertragen werden, eingelesen, aufgelöst und entsprechende Vorbereitungen durchgeführt. Das Einlesen und Auflösen eines Parameters ist anhand des Beispiel-Parameters `noinflux` im Listing Listing 3 dargestellt. Beim Start mit dem Parameter -noinflux wird das Ergebnis und auch der Fortschritt des Buildprozesses nicht auf dem Build-Dashboard angezeigt. Andernfalls werden auf dem Build-Dashboard in `Grafana` aktuelle Statusberichte zu den Prozessen auf den einzelnen Servern und eine Liste der letzten Buildergebnisse dargestellt. Weitere Buildparameter werden in Unterabschnitt 4.5 und Anhang C aufgelistet und erklärt.

Listing 3: Parameter einlesen

```
noinflux = False

for k in list(ap.options.keys()):
    if k == "noinflux":
            noinflux = True
if not noinflux:
    writeBuildInfo()
```

Nachdem alle Parameter eingelesen wurden, wird der Name des aktuellen ausgecheckten Git Branches mit `git rev-parse --abbrev-ref HEAD` ermittelt, um diesen anschließend sperren zu können. Damit wird verhindert, dass während des Buildprozesses Änderungen auf dem Branch von außerhalb der Buildpipeline vorgenommen werden. Hierzu wird ein lockToken erzeugt, der zusammen mit dem zu sperrenden Verzeichnis abgespeichert wird. Das Ganze wird mit einer bestimmten Sperrdauer verbunden, so dass bis zum Ende dieser Zeit oder dem Entsperren des Verzeichnisses durch den Buildserver, in diesem nichts geändert werden kann, selbst nicht mit `--force`. Sobald das Buildverzeichnis gesperrt ist, werden vom Remote Repository alle neuen Änderungen mit `fetch` eingeholt und zusätzlich mit dem Parameter `-p` (prune) alle lokalen Referenzen auf Remote-Tracking-Branches, die dort nicht mehr

existieren, entfernt. Im nächsten Schritt wird von dem zu bauendem Branch der letzte Commit mit dem Befehl `git log --pretty=%H -1 branch` abgerufen, wobei als Branch der zu bauende Branch eingesetzt wird. Der Befehl liefert den formatierten Hash des neuesten Commits zurück. Dieser wird nach Abschluss des Buildprozesses mit dem des neuen Builds verglichen, um zu ermitteln, ob sich etwas geändert hat. Wenn das nicht der Fall ist, wird der neue Build verworfen und mit dem alten weitergearbeitet, denn dies spart Zeit sowie Speicherplatz auf dem Fileserver.

Wenn das lokale Buildverzeichnis Namens branchprj vorbereitet ist, rufen wir die Buildpipeline rekursive mit folgendem Befehl erneut auf:

```
buildQftest -branch merge -createbranch -nobuild -nosave -checkout checkout.
```

Hierbei ist buildqftest der Name der Python Builddatei. Die weiteren Parameter werden wie folgt aufgelöst:

- `-branch merge` es werden die zugehörigen Merge-Methoden ausgeführt
- `-createbranch` mehrere Branches werden zu einem neuen Integration-Branch gemergt
- `-nobuild`, `-nosave` nach dem Mergen der Branches werden weitere Build Schritte ausgelassen
- `-checkout Branch` gibt den zu bauenden Integration-Branch an

Alle bis jetzt genannten Befehle werden erneut ausgeführt, jedoch teilweise mit anderen Parametern, was zu anderen Ergebnissen führt. Die wichtigste Änderung ist, dass das Arbeitsverzeichnis zu `mergeprj` geändert wurde, in welchem weiter gearbeitet wird. Zu dem angegebenem Integration-Branch werden daraufhin aus der qftest.branch.integration Datei die jeweils zugehörigen Labels ausgelesen. Für jeden Integration-Branch enthält diese Datei eine Zeile wie folgt:

Listing 4: Zeile aus qftest.branch.integration

```
# Medium
integration/medium: Medium Integration,!Tests failing
```

An erster Stelle ist der Branch-Name definiert, gefolgt von beliebig vielen Labels. Alle Labels die mit einem ! beginnen, werden bei einem Build ausgelassen. Das bedeutet, dass in einem Medium-Build (vgl. Listing 4) alle Merge-Requests, die ein `Medium Integration`, aber kein `Tests failing` Label besitzen, gemergt werden. Nachdem alle zugehörigen Merge-Requests ermittelt sind, wird mit `git fetch -p` das lokale Git Verzeichnis aktualisiert und geprüft, ob es die Basis des zu mergenden Branches überhaupt gibt. Dieser ist in der Datei `qftest.branch.master` definiert und ist bis auf Ausnahmen immer der aktuelle Hauptbranch. Hierzu werden mit `git branch -r` alle Branches ausgegeben und anschließend mithilfe eines Vergleiches des gesuchten Branches (vgl. Listing 5) versucht, eine Übereinstimmung zu finden. Wenn

das der Fall ist, wird das Git Repository mit `git reset --hard`, gefolgt von `git clean -d -f` resettet. Zum Abschluss wird der gefundene Branch mit `git checkout branch` ausgecheckt. Wenn kein passender Branch gefunden wird, bricht die Buildpipeline hier ab.

Die Befehle werden, wie in Listing 5 sichtbar, nicht direkt ausgeführt, sondern Mithilfe der Hilfsfunktion `runAndLog` oder `runAndCollect`. Beide Methoden ermöglichen ein ausführliches Logging der Abläufe, wobei bei runAndCollect noch die zurückgegebenen Werte in eine Variable geschrieben werden.

Listing 5: guess Branch Methode

```
def gitGuessBranch(prj, branch, buf=None):
    try:
        os.chdir(prj)
    except:
        return -1
    runAndLog("git fetch -p", buf=buf)
    cmd = "git branch -r"
    ret, out = runAndCollect(cmd, buf=buf)
    ret = None
    hits = 0
    if out:
        for line in out.split("\n"):
            br = line.strip()
            if br.startswith("origin/"):
                br = br[7:]
                if br == branch:
                    return br
                if br.find(branch) > 0:
                    hits += 1
                    ret = br
    if hits == 1:
        return ret
    else:
    return None
```

Hiermit wurde der erste Durchlauf der `gitCheckout`-Methode abgeschlossen. Sie ist eine der zentralen Methoden, die während des Build-Vorganges insgesamt dreimal durchlaufen, aber jeweils mit unterschiedlichen Parametern aufgerufen wird. Im ersten Durchlauf leitet sie das Zusammenbauen des Integration-Branches ein, in welchem sie sich selbst nochmals aufruft, um den Basis-Branch für die kommenden Merges mit der `guessBranch()` Methode (vgl. Listing 5) zu ermitteln und das lokale Verzeichnis vorzubereiten. Nach Abschluss dieses Vorganges wird mit `git pull` das lokale Verzeichnis aktualisiert. Damit der Integration-Branch sauber neu gebaut werden kann, wird der alte mit `git branch -d -f branch` gelöscht und anschließend wieder neu erzeugt mit `git branch Branch`. Dieser Branch, dessen Basis jetzt der angegebene Basis-Branch (in den meisten fällen master) ist, wird jetzt ausgecheckt.

Damit sind alle Vorbereitungen für die kommenden Mergeprozesse abgeschlossen. Wir haben uns alle Änderungen aus dem Remote-Repository geholt und veraltete Dateien entfernt, einen neuen Branch für den Integration-Branch erstellt und ausgecheckt, der als Basis die neueste Version des Master-Branch verwendet.

4.4.1 Merge-Konflikte und automatisierte Konfliktlösung

Die Ausgangslage für das Mergen der Feature-Branches ist der neue Interims-Branch, dessen Basis wiederum der aktuelle master ist, sowie eine Liste aller Merge-Requests der zu mergenden Branches. Aus dem Merge Request (vgl. Listing G.5) werden zuerst der Basis und Ziel Branch ermittelt. Anschließend wird zwischen den folgenden beiden Fällen unterschieden:

- 1. Die Basis des zu mergenden Branches ist dieselbe wie die des Interims-Branches
- 2. Die Basis des zu mergenden Branches ist nicht dieselbe wie die des Interims-Branches

Es wird sich zunächst auf den ersten Fall konzentriert. Als erstes wird versucht, den zu mergenden Branch mit `git merge -Xignore-space-at-eol --no-ff --no-edit branch` automatisch zu mergen. Der Befehl setzt sich aus folgenden Bestandteilen zusammen: `git merge` ist der Grundbefehl, um zwei Branches zu mergen, `-Xignore -space-at-eol` ist ein Argument für die Standard-Merge-Strategie, um jegliche Änderungen von Leerzeichen oder Tabulatoren am Zeilenende zu ignorieren, da durch diese keine relevanten Konflikte entstehen. `--no-ff` verhindert einen fast-forward-Merge, damit immer ein Merge-Commit vorhanden ist. Damit der Merge noch komplett automatisiert durchführt werden kann, wird abschließend noch das Argument `--no-edit` hinzugefügt, welches den Commit Dialog unterdrückt und die Standard Merge-Commit Nachricht akzeptiert. Zuletzt wird der zu mergenden Branch angegeben. Wenn dieser automatische Merge fehlschlägt, wird versucht, den Konflikt mithilfe von Conflict-Resolutions, kurz CR zu lösen.

Die Conflict-Resolutions sind Hauptbestandteil eines Verfahrens, bei dem die Lösung eines Merge-Konfliktes als Textdatei für den späteren Merge abgespeichert wird. Dass eine Conflict-Resolution benötigt wird, erfährt der Entwickler, wenn bei einem Build der Merge seines Branches fehlschlägt. Wenn dieser Fall eintritt, wird eine ungelöste Conflict-Resolution mit den Informationen aus dem fehlgeschlagenen Merge erstellt und alle Entwickler, die an dem Merge-Request vermerkt sind, erhalten eine Email als Benachrichtigung. Die Conflict-Resolution besteht aus drei Dateien und wird in einem Conflict-Resolution-Ordner angelegt. In diesem befindet sich zunächst ein neuer Unterordner, dessen Name dem Hash der beim Merge fehlgeschlagenen Datei entspricht. Darin werden drei Dateien angelegt, welche alle drei Eingangszustände des Merges darstellen: Die `base`-Datei enthält die Basis der beiden

zu mergenden Branches, `remote` enthält den aktuellen Stand des Interim-Branches und `local` den Stand des zu mergenden Branches der Datei. Der betroffene Entwickler muss jetzt den Merge-Konflikt manuell lösen und die daraus resultierende Conflict-Resolution (Zielzustand) unter dem ursprünglichen Dateipfad commiten.

Wenn im Mergeprozess ein Merge-Konflikt auftritt, wird zuerst geprüft, ob genau dieser Konflikt in der Vergangenheit schon einmal gelöst worden ist. Das passiert durch das Setzten des Git-Parameters `rerere` (reuse recorded resolution). Dieses merkt sich im git Cache alle Lösungen von vergangenen Merge-Konflikten und wenn exakt der gleiche Merge-Konflikt noch einmal auftritt, löst es ihn automatisch mit der ihm schon bekannten Lösung. Sobald eine Conflict-Resolution in einem Build einmal abgerufen worden ist, kann der Entwickler sie aus dem Branch aber wieder entfernen, damit sie später beim Mergen des Branches nicht mit gemergt wird.

Die manuell entwickelte Lösung wird als neuer Stand der Datei dem Interim-Branch mit `git add datei` hinzugefügt. Für den Fall, dass ein Merge-Konflikt auf eine neue Weise gelöst werden muss, d. h. wenn dieser zwar durch eine Conflict-Resolution bereits in der Vergangenheit gelöst worden ist, jedoch der Lösungsweg inzwischen geändert wurde, gibt es am Ende des Mergeprozesses einen Check, ob eine neuere Conflict-Resolution für diesen Merge-Konflikt vorliegt. Wenn das der Fall ist, muss die alte Lösung mit `git rerere forget Datei` gezielt gelöscht werden, so dass die neue Conflict-Resolution greifen kann. Anschließend wird mit dem Merge-Request weitergemacht, bis wieder ein nicht lösbarer Merge-Fehler auftritt oder alle Dateien erfolgreich gemergt worden sind, so dass sie auf den Interim-Branch mithilfe von `git commit -n --no-edit` committed werden können. Mit `-n` werden alle aktiven pre-commit und commit hooks umgangen und `--no-edit` akzeptiert die standardmäßig generierte Commit-Nachricht.

Im zweiten Fall ist die Basis des zu mergenden Branches nicht dieselbe wie die Basis des zu bauenden Interim-Branches. Um diesen Fall trotzdem automatisch mergen zu können, wird der zu mergende Branch zuerst auf einen temporären Branch kopiert mit dem Befehl `git branch tmp/branch origin/branch` und dieser anschließend ausgecheckt. Im nächsten Schritt wird ein sogenannter Rebase durchgeführt. Mit dem Befehl `git rebase --onto branch origin/master tmp/branch` werden alle Commits, die in dem aktuell ausgecheckten temporären Branch, aber nicht im master enthalten sind, auf den Integration-Branch gerebaset. Falls es dabei zu Merge-Konflikten kommt, greift erneut die Logik der Conflict-Resolution und wenn diese erfolgreich ist, wird der Rebase mit `git rebase --continue` fortgesetzt. Abschließend wird der Integration-Branch ausgecheckt, so dass der temporäre Branch in den Integration-Branch mit dem Befehl `git merge -Xignore-space-at-eol --no-commit --no-ff tmp/branch` gemergt und die Änderungen commited werden können. Der temporäre Branch wird nicht mehr benötigt und gelöscht.

Nachdem alle Merge-Requests durchgelaufen sind, wird der daraus resultierende

Integration-Branch in das Remote Repository mit Hilfe von `git push --force` gepusht und zusätzlich auf dem Fileserver gespeichert. Damit ist das Buildscript, welches den Integration-Branch zusammen baut, abgeschlossen. Es gibt jedoch noch ein weiteres Skript auf dem Buildserver mit dem Namen buildQftest.

In diesem Skript wird das Zusammenbauen des Integration-Branches initialisiert. Auch in diesem wird als erstes das Logging aufgesetzt und anschließend die Startparameter eingelesen, welche später zum Teil an das Buildscript weitergereicht werden. Im nächsten Schritt werden alle angegebenen Testmaschinen gelockt, damit diese beim späteren Starten der Tests verfügbar sind. Anschließend wird der Integration-Branch gemergt und eine lauffähige Anwendung daraus gebaut, danach die angegebenen Tests mit dieser durchgeführt. Darauf folgend werden die Testergebnisse in Confluence veröffentlicht und alle Testmaschinen wieder freigegeben. Das Packaging wird für Linux, Windows und MacOS durchgeführt und dauert relativ lange, weshalb es unabhängig vom Buildprozess automatisch startet, wenn auf dem Fileserver ein neuer Build gepusht wird. Dadurch ist der Buildserver während dieser Zeit schon wieder für andere Buildprozesse verfügbar. Das ist wichtig, da über diesen auch Prozesse wie das Starten von Tests ohne das bauen eines Builds vorher oder das Bauen von Handbuchversionen ausgeführt werden.

4.5 Workflows und Anwendungsfälle

Mithilfe der oben beschriebenen Pipeline kann ein Entwickler einen neuen Integrations-Build aus mehreren Feature-Branches erstellen oder auch nur aus einen einzelnen Branch mit dem Hauptbranch. Diese neuen Builds können nach Belieben mit allen oder auch nur einer Auswahl der verfügbaren Tests getestet werden. Die Tests können sowohl automatisch als auch interaktiv auf diversen Testmaschinen gestartet, überprüft, repariert oder weiterentwickelt werden.

Alle Prozesse werden von dem Hauptserver `pp20` mit dem Befehl `buildAndTest`, kurz `bat` gestartet. Abhängig von den folgenden Parametern werden verschiedene Aktionen ausgeführt. Typische Anwendungsfälle für das Arbeiten mit der Buildpipeline als Entwickler sind folgende Szenarien:

- Das Erstellen eines Builds für einen Integrations-Branch:
  bat -checkout integration/branch -createbranch -syncm
- Das Mergen eines Integrations-Branches ohne Build:
  bat -checkout integration/branch -createbranch -nobuild -nosave
- Das Ausführen eines Tests auf bestimmten Testmaschinen:
  bat -l linuxHost -w windowsHost -test – -suite"Testsuite"
- Das Bauen und Testen eines Integrations-Branches:
  bat -checkout integration/milestone -createbranch -sync -syncm -test -l linux-

Host -w windowsHost -m macOSHost -report confluence – -suite webtests

Die verwendeten Parameter haben folgende Bedeutungen: `createbranch` wird beim Bauen eines Integration-Branches benötigt, `syncm` gibt an, dass das Ergebnis des Buildprozesses auf dem Fileserver gespeichert wird. `nobuild` und `nosave` verhindern das Packaging und Speichern des fertigen Builds auf dem Fileserver. Für das Ausführen von Tests können Testmaschinen mit `-l` (Linux) `-w` (Windows und `-m` (MacOS) angegeben werden, dahinter folgt jeweils eine kommaseparierte Liste der Namen oder eine Zahl für die Anzahl der Testmaschinen des jeweiligen Betriebssystems. Ganz am Ende wird nach -- mit `-suite` eine oder mehrere Testsuiten angegeben, die ausgeführt werden sollen. Des Weiteren gibt es vordefinierte Schlagwörter, die eine Gruppe an Testsuiten ausführt, wie zum Beispiel Webtests. Mit `-report` werden html Reports generiert, welche mit `-confluence` dort veröffentlicht werden können, um die Ergebnisse mit allen Entwicklern zu teilen. Dies ist nur ein kleiner Ausschnitt aller Parameter, jedoch die, die am häufigsten verwendet werden.

Zum Stand der Dokumentation der Buildpipeline gibt es insgesamt über 100 verschiedene Parameter, mit denen die Buildpipeline konfiguriert werden kann. Diese sind im Anhang C dokumentiert. Die Anzahl der Buildparameter zeigt einerseits die Komplexität der Buildpipeline und andererseits das Potential zur Optimierung, da nur ein Bruchteil der Parameter dokumentiert und einige sogar nicht funktionsfähig sind.

5 Empirische Untersuchung im Entwicklungsteam

Um die vorhandene Buildpipeline zu optimieren, wurden im ersten Schritt im Rahmen der Dokumentation jegliche Unstimmigkeiten und veraltete optimierbare Code bereits analysiert. Die dabei gefundenen Punkte (vgl. Unterabschnitt 6.1) werden in diesem Kapitel durch eine empirische Umfrage des Entwicklungsteams erweitert. Im Rahmen der Bachelorarbeit stellt dies einen zentralen methodischen Schritt dar, um weitere Optimierungspotenziale der komplexen Buildpipeline zu identifizieren und praxisnah zu evaluieren.

5.1 Konzeption und Zielsetzung der Entwicklerbefragung

Die Zielsetzung der Umfrage setzt sich einerseits zusammen aus dem Bedürfnis, Schwachstellen des bestehenden Buildprozesses bezüglich der Funktionalität und Benutzerfreundlichkeit zu erfassen, sowie anderseits dem Anspruch, die erfahrenen Entwickler im Unternehmen `QFS` in den Verbesserungsprozess mit einzubinden. Es sollen sowohl verborgene (implizite) Kenntnisse über die Buildpipeline, als auch Nutzererfahrungen systematisch erfasst werden. [23, Kap. 6].

Für die Befragung wurden die Entwickler in zwei relevante Nutzergruppen aufgeteilt, die teilweise getrennt voneinander betrachtet werden, um möglichst aussagekräftige Ergebnisse zu erzielen. Die Gruppen sind: `Benutzer`, sie sind die reinen Anwender der Buildpipeline und `Pipeline-Entwickler`, sie sind die Personen die aktiv an der Buildpipeline mit entwickeln, also nicht nur mit ihr, sondern auch an ihr arbeiten. Dadurch können spezifische Probleme und Verbesserungsvorschläge sowohl aus Nutzerperspektive als auch aus Sicht der Pipeline-Entwicklung herausgearbeitet werden.

Der inhaltliche Aufbau der Umfrage orientiert sich an den Qualitätskriterien für Software, konkret der ISO 25010 [24], die weltweit anerkannt ist und die Eigenschaften funktionale Eignung, Leistungsfähigkeit, Kompatibilität, Benutzbarkeit (Usability), Zuverlässigkeit, Sicherheit, Wartbarkeit, Flexibilität und Funktionssicherheit umfasst. Um diese zentralen Aspekte der Softwarequalität sicherzustellen, wurde für jedes relevante Kriterium mindestens eine Frage konzipiert.

5.2 Theoretischer Aufbau des Fragebogens

Der Aufbau der Umfrage folgt den klassischen Prinzipien der sozialwissenschaftlichen Fragebogenkonstruktion nach Fietz und Friedrichs [25]. Sie beginnt mit einer Einleitung, die relativ kurz gehalten wurde und hauptsächlich das Ziel der Umfrage beschreibt, da alle teilnehmenden Entwickler bereits eine Erläuterung und Einweisung zur Umfrage in einem Meeting erhalten haben. Alle Entwickler wurden darüber informiert, dass die Ergebnisse wenn gewünscht anonymisiert ausgewertet werden.

Es gab jedoch zum Abschluss die Möglichkeit, freiwillig den Namen anzugeben, um sich bei komplexeren Vorschlägen oder Rückfragen direkt mit dem jeweiligen Entwickler abstimmen zu können und deren Anregungen optimal umzusetzen. Damit wird dem Prinzip der informierten Einwilligung und den Anforderungen der DSGVO entsprochen [26].

Auf die Einleitung folgt der Hauptteil der Umfrage, in dem alle Fragen zur Erhebung der Informationen enthalten sind [25]. Alle Fragen des Hauptteils wurden unter der Berücksichtigung der `10 Gebote` für die Frageformulierung von Porst konzipiert [27]. Es wurden folgende Punkte beachtet, so dass es nicht zu semantischen oder pragmatischen Problemen kommt, d. h. dass jede Frage sowohl inhaltlich an sich, als auch im Kontext verstanden wird [28]:

- einfache und unmissverständliche Sprache verwenden, die dem Wissensstand der befragten Entwickler entspricht
- kurze und prägnante Fragen benutzen
- die Befragten nicht durch Unterstellungen in eine Richtung drängen
- unklare Begriffe definieren
- überschneidungsfreie Antwortmöglichkeiten verwenden
- hypothetische Fragen vermeiden
- die Reihenfolge der Fragen kontrollieren, um Kontexteinflüsse zu vermeiden [27]

Es wurden gezielt sowohl geschlossene als auch offene Fragen eingesetzt, um ein möglichst umfassendes Bild der Nutzererfahrungen zu bekommen. Bei sogenannten Fragen sind die Antworten in Form von Auswahloptionen oder numerischen Skalen bereits vorgegeben [23, Kap. 10]. Bei diesen wurde darauf geachtet, dass es immer mindestens eine passende Antwortmöglichkeit für jeden Teilnehmer gibt und dass sich die Antwortmöglichkeiten nicht überschneiden [27]. Geschlossene Fragen eignen sich für eine strukturierte und leicht quantifizierbare Erfassung zentraler Aspekte wie z. B. der Nutzungshäufigkeit oder Zufriedenheit im Zusammenhang mit der Buildpipeline. Sie ermöglichen einen Vergleich zwischen den Teilnehmern und damit eine statistische Auswertung [23, Kap. 10].

Offene Fragen wurden hingegen eingesetzt, um individuelle Erfahrungsberichte, unerwartete Probleme oder kreative Verbesserungsvorschläge der Entwickler zu erheben. Durch die Möglichkeit, eigene Erfahrungen, Hindernisse oder Lösungsideen frei zu formulieren, wird es den Befragten ermöglicht, weitere Aspekte einzubringen. Diese Vorgehensweise wurde besonders bei der zweiten Gruppe angewandt, da diese über tiefgehende explorative Kenntnisse zur Buildpipeline verfügt, welche nicht allein durch geschlossene Fragen erfasst werden können [29].

Am Ende der Umfrage gibt es einen Schlussteil. In diesem wird den Befragten die Möglichkeit gegeben, für Rückfragen ihren Namen anzugeben. Des Weiteren wird für die Teilnahme an der Umfrage gedankt und ein Link zu einer Seite in `Confluence`, dem firmeninternen Wiki genannt, auf der die Ergebnisse der Umfrage nach der Auswertung veröffentlicht werden.

5.3 Umsetzung der Umfrage

Die empirische Untersuchung wurde als Online-Umfrage mit Hilfe des Tools LimeSurvey umgesetzt. Die Verwendung eines strukturierten Online-Fragebogens bietet den Vorteil, eine systematische und zeiteffiziente Erhebung der benötigten Datensätze aus dem Entwicklerteam zu ermöglichen [30] [23, Kap. 10]. Hierbei bietet die Methode eine Integration in den Arbeitsalltag der befragten Entwickler, ohne in diesen zu stark einzugreifen, da die Teilnahme asynchron und mittels den Entwicklern vertrauten Software-Programmen stattgefunden hat.

Zum Beginn der Umfrage stellt die erste Frage des Fragebogens einen Filter dar, welcher alle Teilnehmer in die zwei vorher definierten Gruppen differenziert. Die erste Gruppe umfasst alle Entwickler, die mit der Pipeline arbeiten, sie aber nur benutzen. Die zweite Gruppe sind alle Entwickler, die a) mit der Pipeline arbeiten, aber auch b) aktiv daran mitentwickeln und deshalb zur Hauptzielgruppe der Umfrage gehören. Anhand der Ergebnisse der zweiten Gruppe können konkrete Optimierungsmöglichkeiten der Buildskripte ermittelt werden. Die Ergebnisse der ersten Gruppe sind jedoch nicht uninteressant, da sie aufzeigen, ob die Buildpipeline im allgemeinen noch Verbesserungspotential im Alltag der sie benutzenden Entwickler bietet.

Alle Befragten, die bei der ersten Frage `Wie kommst du als Entwickler*in mit der Buildpipeline in Kontakt?` die Antwortmöglichkeit `über sdev` angeben, sind Entwickler, die an der Pipeline mitentwickeln. Auf diesem Server wird die Pipeline zwar technisch gehostet, aber nur bei der Entwicklung an der Pipeline wird auf diesen Server direkt zugegriffen. Wenn der Befragte eine oder mehrere der anderen Antwortmöglichkeiten auswählt, werden ihm anschließend die Fragen für die `Benutzer` der Buildpipeline angezeigt. Des Weiteren gibt es `Gar nicht` als Antwortmöglichkeit, falls einer der Befragten überhaupt nicht mit der Buildpipeline arbeitet.

Wenn einer der befragten Entwickler gar nicht mit der Buildpipeline in Kontakt kommt, wird nach den Ursachen dafür gefragt. Diese Frage dient wiederum als Filter, um zu unterscheiden, ob sich die Buildpipeline überhaupt im Arbeitsbereich des Entwicklers befindet. Wenn das der Fall ist, folgt eine weitere Frage nach den Gründen für die Nicht-Verwendung der Buildpipeline und es werden Verbesserungsmöglichkeiten vorgeschlagen, um in Zukunft eine Arbeit mit der Pipeline zu ermöglichen.

Der Hauptteil für die restlichen Befragten startet für beide Gruppen mit den gleichen drei geschlossenen Fragen, lediglich die Antwortmöglichkeiten unterscheiden sich. Diese Fragen dienen dazu, die Zufriedenheit und den Umgang mit der Buildpipeline zu erheben. Für die Benutzer folgen zwei offene Fragen, um Frustrationspunkte und mögliche Verbesserungen zu ermitteln. Für die Nutzergruppe der `Pipeline-Entwickler` folgen mehrere Fragen, welche sich nun speziell auf die Buildskripte auf dem Server `sdev` beziehen. Auch hier wird zuerst nach Frustrationspunkten und möglichen Verbesserungen gefragt. Die Verbesserungen sind in funktionale und nicht funktionale Funktionen unterteilt und werden abschließend durch zwei Fragen zu möglichen Optimierungen der in Abschnitt 4 beschriebenen Buildskripte ergänzt. Die abschließende Frage sowie die abschließende Seite der Umfrage sind für alle Nutzergruppen gleich und in Unterabschnitt 5.2 bereits beschrieben.

Der Fragebogen wurde vor dem Start der Erhebung ausführlich mit einem Entwickler getestet, wie von Döring und Bortz empfohlen [23, Kap. 10]. Hierbei wurden alle möglichen Pfade durchlaufen und alle Fragen auf ihre Verständlichkeit und Funktionsfähigkeit überprüft. Anschließend wurde sie dem Entwicklerteam der Firma `QFS` im wöchentlichem Entwickler-Meeting vorgestellt und als Link per E-Mail an alle Entwickler verteilt. Nachdem alle Befragten geantwortet hatten, wurde mit der Auswertung begonnen.

5.4 Methodik der Auswertung

Um eine möglichst umfassende Auswertung der Daten zu ermöglichen, erfolgt diese sowohl qualitativ, als auch quantitativ. Im ersten Schritt wurden alle Rohdaten der Ergebnisse aus `LimeSurvey` als Excel-Datei (.xlsx Format) exportiert und für die weitere Verarbeitung aufbereitet. Dies umfasste das Löschen technischer Hilfsspalten, die inhaltliche Trennung der einzelnen Nutzergruppen gemäß der Zuordnung aus der ersten Frage und eine Plausibilitätsprüfung. Die Plausibilitätsprüfung wurde manuell durchgeführt, da die Menge der befragten Entwickler mit `n = 13` vergleichsweise gering ist. Sie umfasst die Überprüfung der Ergebnisse auf Logik und Nachvollziehbarkeit, um potentielle Fehler frühzeitig zu identifizieren [23, Kap. 10].

Alle geschlossenen Fragen mit festen Antwortmöglichkeiten oder Skalen wurden mithilfe der explorativen Datenanalyse unter Einsatz von grafischen Darstellungen ausgewertet. Hierbei wurden absolute und relative Häufigkeiten berechnet und, wo es sinnvoll erschien, deskriptive Statistiken wie Mittelwert und Standardabweichung ermittelt [23, Kap. 7]. Die so ermittelten Werte wurden durch geeignete Diagramme visualisiert, um mögliche Trends und Schwerpunkte darzustellen. Sowohl die relativen Häufigkeiten, als auch der Mittelwert können bei einer kleinen Datenmenge, wie sie in diesem Fall vorliegt, allerdings stark durch einzelne Ausreißer beeinflusst werden. Daher sind sie nur mit Bedacht zu interpretieren. Die Standardabweichung verdeutlicht diese Ausreißer entsprechend.

Die verbleibenden offenen Fragen wurden gemäß der qualitativen Inhaltsanalyse nach Mayring [29] ausgewertet. Das Ziel dabei war es, die erhaltenen Antworten anhand von wiederkehrenden Themen zu passenden Problemen und Optimierungsvorschlägen systematisch zu sortieren und in Kategorien zu bündeln. Anschließend wurden die für diese Arbeit relevanten Optimierungen ermittelt. Die Bearbeitung erfolgte in folgenden Schritten:

1. Extraktion: Die offenen Antworten wurden zuerst einzeln in Excel extrahiert und gesichtet.

2. Kategorisierung: Die Inhalte wurden zusammengefasst und anhand inhaltlicher Übereinstimmungen in Kategorien gegliedert. Ähnliche oder gleiche Vorschläge wurden zu einer Optimierungskategorie wie `Dokumentation` (DOKU) oder `Entwicklung` (DEV) zusammengeführt

3. Bewertung: Jede herausgearbeitete Optimierung wurde dahingehend geprüft, ob sie mit ihr angemessenem und gerechtfertigtem Aufwand umgesetzt werden kann. Des Weiteren wurde geschaut, ob sie sich konkret auf die Buildskripte oder einen anderen Teil der Buildpipeline bezieht und damit für diese Arbeit nicht relevant ist (Unterabschnitt 4.3).

4. Übersicht: Die Ergebnisse wurden in einer eigenen Tabelle dokumentiert und im firmeninternen `Confluence` veröffentlicht

Mit dieser Vorgehensweise wurde überprüft, welche Optimierungen für diese Arbeit und welche Verbesserungsvorschläge zwar außerhalb der Arbeit, aber dennoch für das Unternehmen relevant sind. Somit stellt die Umfrage für das Unternehmen auch abseits der Buildskripte einen praktischen Mehrwert dar, da auch Optimierungsmöglichkeiten für andere Komponenten der Buildpipeline ermittelt wurden. Des Weiteren helfen sie dabei, ein besseres Verständnis dafür zu entwickeln, wie die Buildpipeline optimal eingesetzt werden muss, damit die Entwickler bestmöglich mit ihr arbeiten können.

5.5 Auswertung der Umfrage

Im folgenden Abschnitt werden die wichtigsten Ergebnisse der empirischen Umfrage systematisch analysiert und vorgestellt. Hierbei werden zuerst die geschlossenen und anschließend die offenen Fragen für die differenzierten Gruppen betrachtet.

5.5.1 Quantitative Ergebnisse

Anhand der ersten vier geschlossenen Fragen können Annahmen zum allgemeinen Verhältnis zwischen dem Entwicklerteam und der Buildpipeline getroffen werden.

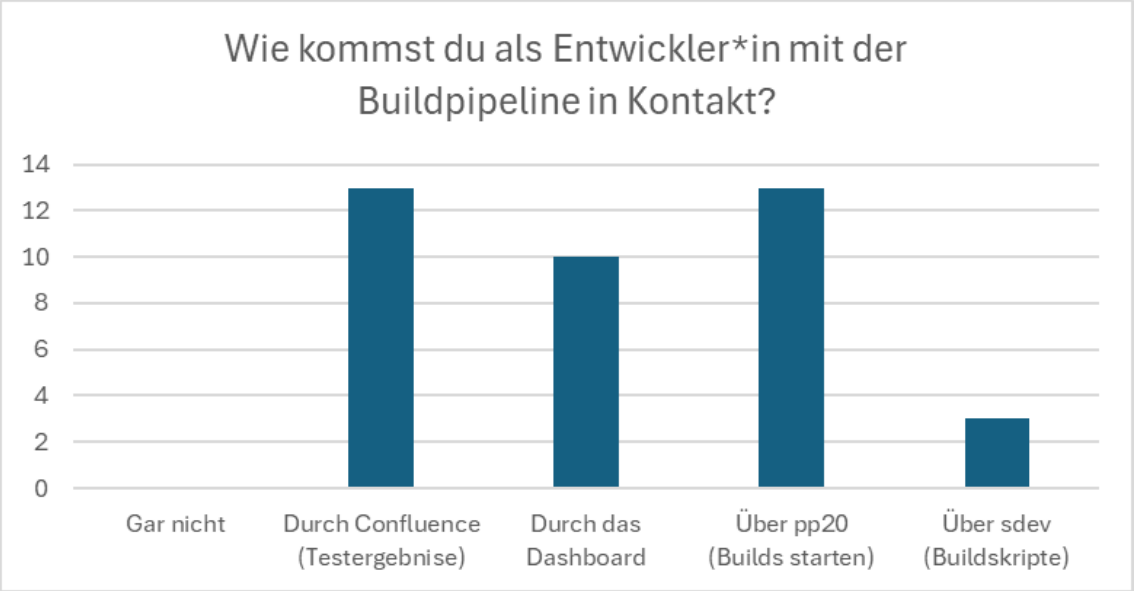


Abbildung 6: Nutzung der Buildpipeline-Komponenten

Die Nutzung der einzelnen Komponenten der Buildpipeline durch die Entwickler ist anhand der absoluten Häufigkeiten im Diagramm in Abbildung 6 dargestellt.

Es ist erkennbar, dass alle der 13 befragten Entwickler der Firma QFS die Buildpipeline benutzen. Jedoch arbeiten nur drei der Entwickler auf dem Entwicklungsserver `sdev` an den Buildskripten, was bedeutet, dass die restlichen zehn von ihnen der ersten Nutzergruppe und nur diese drei der zweiten Nutzergruppe zugeordnet werden. Bei den kommenden Fragen werden sowohl die Gesamtheit aller Entwickler, als auch ein Vergleich der beiden Nutzergruppen betrachtet.

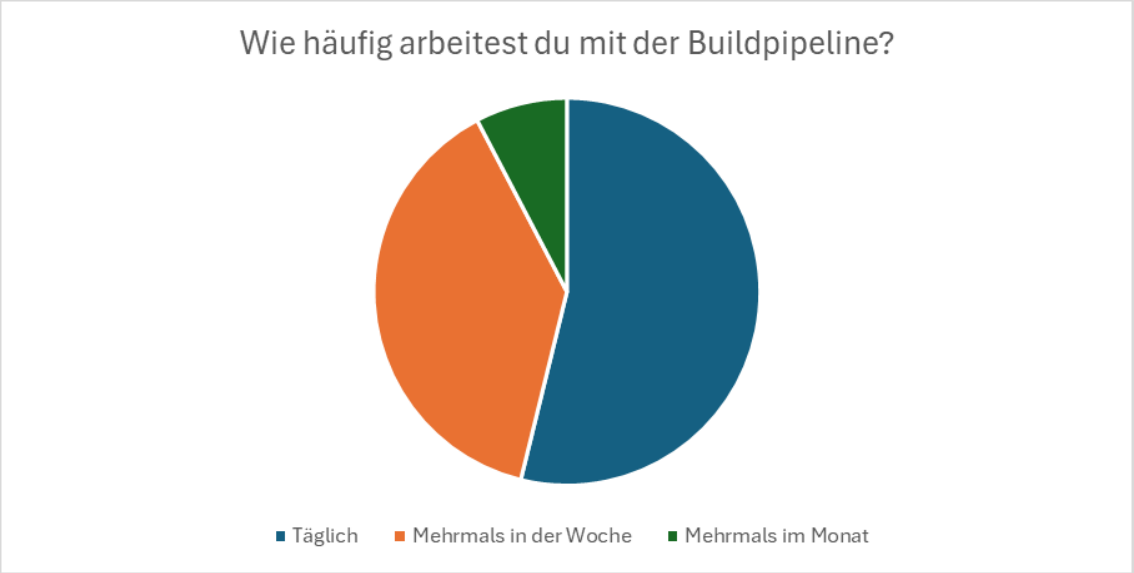


Abbildung 7: Häufigkeit der Nutzung der Buildpipeline

Von den 13 befragten Entwicklern benutzen 12 von ihnen die Buildpipeline mehrmals die Woche und 7 davon sogar täglich, dargestellt als relative Häufigkeiten in Abbildung 7. Lediglich eine Person sticht hierbei heraus und hat angegeben nur

mehrmals im Monat mit ihr zu arbeiten. Das zeigt, dass die Buildpipeline ein zentrales Tool im Entwicklungszyklus der Firma QFS ist und bis auf einzelne Ausnahmen jeder der Entwickler mehrmals in der Woche mit ihr arbeitet

Obwohl die Buildpipeline eine essentielle Komponente im Entwicklungsalltag ist, zeigt die dritte Frage auf, dass die Benutzer nicht vollkommen zufrieden mit ihr sind. Mit den in Abbildung 8 dargestellten absoluten Häufigkeiten, können wir hierzu den Mittelwert der Zufriedenheit auf einer Skala von 0 (nicht zufrieden) bis 10 (voll zufrieden) berechnen:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i \cdot h_i}{N} = \frac{(5 \times 3) + (6 \times 2) + (7 \times 5) + (8 \times 2)}{12} = 6,5$$

Der Mittelwert ist mit 6,5 etwas besser als der Mittelwert der Skala ,der sich bei 5,5 befindet, aber er bewegt sich trotzdem nur im Bereich von mittel bis gut. Die Standardabweichung vom Mittelwert beträgt:

$$s = \sqrt{\frac{\sum_{i=1}^{k} h_i \cdot (x_i - \bar{x})^2}{N - 1}}$$

$$s = \sqrt{\frac{3 \cdot (5 - 6{,}5)^2 + 2 \cdot (6 - 6{,}5)^2 + 5 \cdot (7 - 6{,}5)^2 + 2 \cdot (8 - 6{,}5)^2}{12 \ - \ 1}} \approx 1{,}09$$

Diese relativ niedrige Standardabweichung von ca. 1,09 deutet darauf hin, dass die Mehrheit der Benutzer die Buildpipeline ähnlich bewertet, es gibt also nur eine moderate Streuung um den Mittelwert. Es zeigt jedoch auch, dass Optimierungen der Buildpipeline sinnvoll sind, dass es zwar keine Ausschläge in die negative Richtung, aber auch keine in die sehr positive Richtung (Skalenwerte 9, 10) gibt.

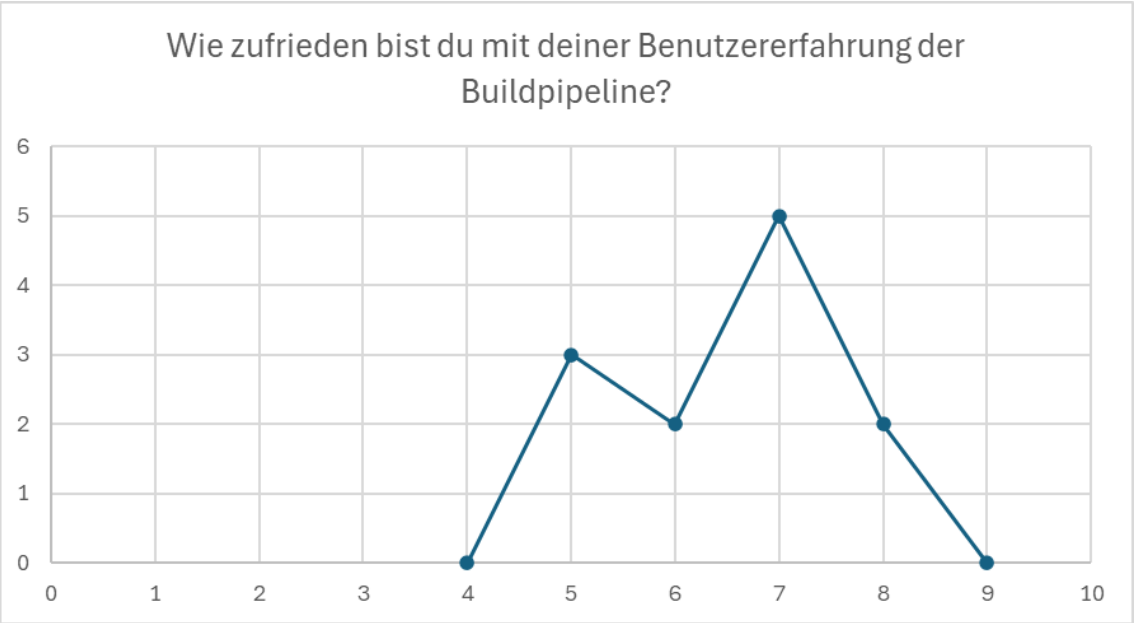


Abbildung 8: Zufriedenheit mit der Benutzererfahrung

Während diese Frage die allgemeine Zufriedenheit mit der Buildpipeline ermittelt

hat, konzentrierte sich die nächste Frage auf die Schwierigkeiten mit den einzelnen Komponenten. Die Antwortmöglichkeiten sind wie bei der ersten Frage die sechs Komponenten der Buildpipeline. Für diese gab es wiederum jeweils eine Skala mit vier Optionen von Nie bis Oft. Es wurden vier Antwortmöglichkeiten gewählt, damit sich die Befragten in eine positive oder negative Richtung entscheiden müssen, da viele Menschen dazu neigen, ansonsten einfach die Mitte auszuwählen [23, Kap. 10]. Die Ergebnisse dieser Matrix-Frage sind im Diagramm in Abbildung 9 dargestellt.

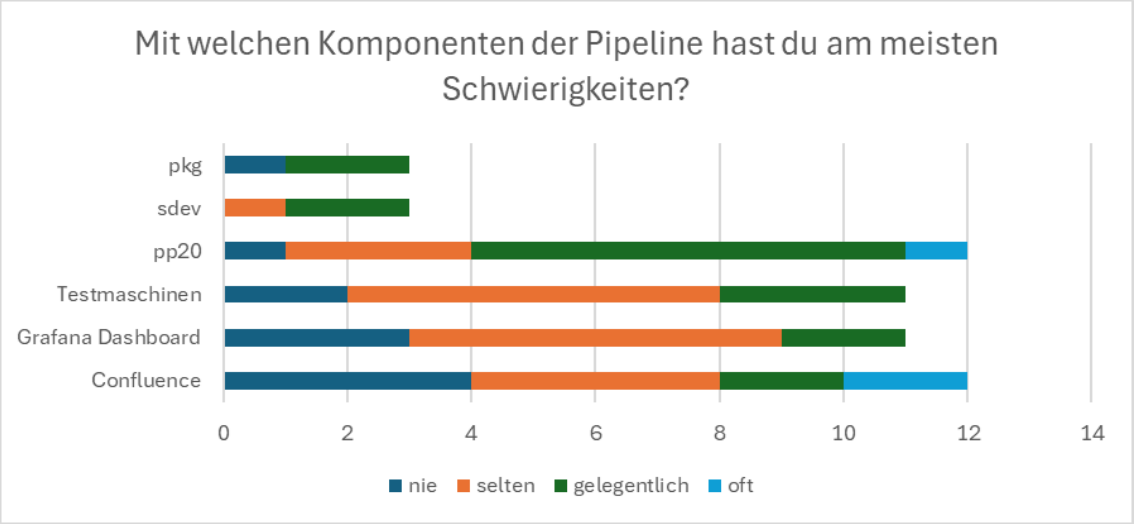


Abbildung 9: Schwierigkeiten mit den Buildpipeline-Komponenten

Hierbei ist zu beachten, dass nur die Nutzergruppe der `Pipeline-Entwickler` die Optionen `pkg` und `sdev` zur Auswahl gehabt hat, da die Nutzergruppe der Benutzer mit diesen beiden nicht in Kontakt kommt. Um die Ergebnisse besser vergleichen zu können, wurde aus den Daten eine Tabelle erstellt (Tabelle 1) und für alle Komponenten der Mittelwert berechnet. Damit das überhaupt möglich ist, wurden für die Schwierigkeitsstufen `Nie` bis `Oft` die Zahlen 0 bis 3 angenommen. Gestartet wurde bei 0 für `Nie`, da wenn man niemals Schwierigkeiten mit der Komponente hat, sie rechnerisch nicht ins Gewicht fallen sollte.

Der Vergleich der Mittelwerte zeigt auf, dass die technisch komplexen Komponenten wie `pp20` (Hauptserver, siehe Unterabschnitt 4.3), `sdev` (Buildserver) und `pkg` (packaging Server) mehr Schwierigkeiten bereiten als die rein grafischen Oberflächen wie Confluence oder das Grafana Dashboard. Jedoch gibt es auch hier Entwickler, die insbesondere bei Confluence sogar oft auf Schwierigkeiten stoßen, so dass auch hier Optimierungspotential besteht. Der Durchschnitt liegt bei den ersten drei Komponenten ca. beim Wert `Selten` und bei den letzten drei zwischen `Selten` und `Gelegentlich`, mit einer Tendenz zu `Gelegentlich`.

Komponente	Nie	Selten	Gelegentlich	Oft	N	$\bar{x}$
Confluence	4	4	2	2	12	1,17
Grafana Dashboard	3	6	2	0	11	0,91
Testmaschinen	2	6	3	0	11	1,09
pp20	1	3	7	1	12	1,67
sdev	0	1	2	0	3	1,67
pkg	1	0	2	0	3	1,33

Tabelle 1: Schwierigkeit der Buildpipeline-Komponenten aller Befragten (0 = nie, ..., 3 = oft)

Wenn man jetzt nur die Nutzergruppe der `Pipeline-Entwickler` betrachtet (Tabelle 2), wird sichtbar, dass die Abstände der Mittelwerte zwischen den ersten drei Komponenten und den letzten drei Komponenten noch größer werden. Das bedeutet, die Entwickler, die tiefer gehendes Wissen zu der Buildpipeline besitzen, haben weniger Probleme mit den vermeintlich einfacheren Komponenten, aber trotzdem noch welche mit den technisch anspruchsvolleren Komponenten wie `sdev` und pp20. Da `sdev` der Buildserver ist, auf dem die Buildskripte ausgeführt werden, und `pp20` der Hauptserver, von dem diese angesteuert werden, sind diese beiden die zentralen Komponenten, auf die sich die vorliegenden Arbeit fokussiert. Diese beiden Komponenten besitzen die höchsten Mittelwerte und verursachen dementsprechend die meisten Schwierigkeiten bei der Benutzung, was auf ein besonders hohes Optimierungspotential hindeutet.

Komponente	Nie	Selten	Gelegentlich	Oft	N	$\bar{x}$
Confluence	1	2	0	0	3	0,67
Grafana Dashboard	1	2	0	0	3	0,67
Testmaschinen	0	3	0	0	3	1,00
pp20	0	1	1	1	3	2,00
sdev	1	0	2	0	3	1,33
pkg	0	1	2	0	3	1,67

Tabelle 2: Schwierigkeit der Buildpipeline-Komponenten, Pipeline-Entwickler (0 = nie, ..., 3 = oft)

5.5.2 Qualitative Ergebnisse

Um die konkreten Optimierungen zu ermitteln, wurden diverse offene Fragen an die beiden Nutzergruppen gestellt. Während die Fragen an die erste Nutzergruppe auf die gesamte Buildpipeline mit allen Komponenten ausgelegt waren, sind die Fragen an die zweite Nutzergruppe speziell auf die Buildskripte bezogen. Die erhaltenen Antworten wurden wie in Unterabschnitt 5.4 beschrieben ausgewertet. Das vollständige Antwortmaterial aus allen geschlossenen Fragen wurde mehrfach analysiert, um daraus einzelne Verbesserungsvorschläge herauszuarbeiten. Diese wurden dahingehend überprüft, ob sie a) umsetzbar sind und b) relevant für die Buildskripte. Wenn Beides der Fall ist, wurden sie einer Kategorie zugeordnet. Beim Analysieren der Antworten haben sich die Kategorien Dokumentation und Entwicklung ergeben. In DOKU sind alle Optimierungen zusammengefasst, die sich der Dokumentation des Codes oder der Benutzerführung zuordnen lassen, in DEV alle Optimierungen, die im Code implementiert werden müssen. Letztere können entweder das Entwickeln der Buildskripte vereinfachen oder fehlende Funktionen in ihnen ergänzen.

Wenn eine Optimierung schon vorhanden ist, also ein Feature gewünscht wurde, welches bereits zum Zeitpunkt der Umfrage implementiert war, dem Gefragten aber einfach nicht bekannt, wurden diese Optimierung zwar abgelehnt, aber in der Dokumentation ergänzt. Des Weiteren wurden sie dem Befragten vorgeführt, sofern dieser seinen Namen am Ende der Umfrage angegeben hatte. Ein Beispiel hierfür ist das Lösen von Conflict-Resolutions außerhalb des Texteditors `Notepad++`. Diese Optimierung wurde von einem Benutzer gewünscht, obwohl sie schon möglich ist, ihm jedoch nicht bekannt. Die fehlende Dokumentation wurde ergänzt und dem Benutzer die Funktion gezeigt.

Die Ergebnisse der Antworten der ersten Nutzergruppe waren für die Optimierungen nicht relevant, da sie die Buildskripte nicht betreffen. Jedoch zeigen sie auf, worauf geachtet werden sollte, wenn die Buildpipeline in einem Projekt oder Entwicklerteam einsetzt wird, damit die Entwickler optimal mit ihr arbeiten können. Die Ergebnisse aus diesen Fragen sind trotzdem in der Tabelle 3 enthalten, wurden aber als nicht relevant gekennzeichnet.

Die Antworten der zweiten Gruppe, welche speziell für die Buildskripte der Buildpipeline ausgelegt sind, wurden zu acht prägnanten Optimierungen zusammengefasst. Sie sind aufgelistet in der Tabelle 3 und werden in den zugehörigen Abschnitten im nächsten Kapitel näher erläutert.

Optimierung	**Umsetzbar**	**Relevant**	**Kategorie**
bessere, ausführlichere und aktuelle (gepflegte) Dokumentation	JA	JA	DOKU
Benutzerguide zur Steuerung der Pipeline über pp20	JA	JA	DOKU
bestehende Dokumentation aufräumen (alte Inhalte entfernen)	JA	JA	DOKU
Aliase dokumentieren und in pp20, sdev hinterlegen	JA	JA	DOKU
GUI für das Starten von buildAndTest	JA	NEIN	-
buildAndTest allgemein vereinfachen	NEIN	-	-
Builds aus Grafana starten	JA	NEIN	-
bessere Fehlermeldungen -> besseres Fehlerfeedback in der E-Mail	JA	JA	DEV
bessere Fortschritts/Statusanzeigen im Dashboard	JA	NEIN	-
Conflict Resolutions vereinfachen, ist aber nicht wirklich möglich - eigenes Repo für die CR ist eine Option	NEIN JA	- JA	- DEV
einfache Option Builds/Tests zu beenden	JA	JA	DEV
Fehlerhistorie eines Testfalls in Confluence umständlich	JA	NEIN	-
Grafana Dashboard längere Historie -> gibt es schon	-	-	-
Mergeprozesse über Mergetool -> gibt es schon	-	-	-
wenn minismoke (Testbuild) erfolgreich und Hauptbuild danach fehlschlägt minismoke verwenden -> Fallback Parameter	JA	NEIN	-
automatische Jobs pflegeleichter machen	JA	NEIN	-
Buildskripte modularer machen	JA	JA	DEV
Testumgebung zum Entwickeln der Pipeline	JA	NEIN	-

Tabelle 3: Alle ermittelten und ausgewerteten Optimierungen

6 Optimierung des Buildprozesses

Nach der systematischen Dokumentation und empirischen Analyse der bestehenden Buildpipeline, wurden diverse Optimierungspotentiale herausgearbeitet. Das folgende Kapitel widmet sich der praktischen Umsetzung der ermittelten Verbesserungen und beschreibt die methodischen Grundlagen und die technische Vorgehensweise sowie deren Ergebnisse.

6.1 Optimierungen

Durch die empirische Analyse wurden acht potentielle Optimierungen ermittelt. Diese lassen sich in jeweils vier Optimierungen für die Dokumentation der Buildpipeline und vier für die Funktionalität und Benutzerfreundlichkeit der Buildskripte unterteilen. Zusätzlich dazu haben sich bei der Analyse und Dokumentation der Buildskripte folgende Optimierungsmöglichkeiten aufgezeigt:

- Entfernen alter ungenutzter Methoden und Variablen (DEV)
- Aufteilen der komplexen Buildskripte in thematisch passende Teilskripte (DEV)
- Ergänzen eines Parameters, der bei Merge-Fehlern den gestarteten buildAndTest Prozess abbricht (DEV)
- Erstellen einer README für die Benutzung der Buildpipeline (DOKU)

Ein Teil dieser Punkte überschneidet sich mit den bereits ermittelten Optimierungen aus der Umfrage und wird zu einem gemeinsamen Punkt zusammengeführt, genauso wie die Dokumentations-Optimierungsvorschläge, die aus der Umfrage hervorgegangen sind. Sie werden alle zusammen in einem neuen, ausführlichen Benutzerguide zusammen umgesetzt. Zwei der Punkte, die sich bei der Dokumentation aufgezeigt haben, sind jedoch inhaltlich komplett neue Verbesserungen und werden in einen eigenen Punkt genannt. Die finalen Optimierungen sind:

1. Refaktorisierung: Entfernen alter ungenutzter Methoden und Variablen (DEV)
2. Modularisierung: Aufteilen der komplexen Buildskripte in thematisch passende Teilskripte (DEV)
3. Merge-Fehler Handhabung: Ergänzen eines Arguments, der bei Merge-Fehlern den gestarteten Build abbricht (DEV)
4. Fehlermeldungen: Überarbeitung des Fehlerfeedbacks für die Benutzer der Buildpipeline (DEV)
5. Stoppen der Buildpipeline: Implementierung eines neuen Arguments, der das Stoppen eines Builds erlaubt (DEV)

6. Conflict-Resolutions: Auslagern der Conflict-Resolutions in ein eigenes Git-Repository (DEV)

7. Benutzerguide: Erstellen eines neuen aktuellen Benutzerguides für die Benutzung der Buildpipeline (DOKU)

Die einzige Optimierung bezüglich der Dokumentation umfasst, dass Erstellen eines neuen, ausführlichen Benutzerguides für die Benutzung der Buildpipeline und das Arbeiten mit den Buildskripten, dies kann direkt im firmeninternen Wiki umgesetzt werden. Sie bildet auch die Grundlage für die spätere Anleitung (README) der Toolchain. Um die weiteren Optimierungen umzusetzen, ist ein komplexerer Workflow nötig.

6.2 Implementierungs- und Testprozess

Die Buildskripte befinden sich in einem eigenen Git-Repository mit dem Namen `qtools`. In diesem wird für jede der `DEV` Optimierungen ein neuer Branch erstellt. Auf diesem Branch wird das zugehörige Feature entwickelt und sobald die Optimierung vollständig implementiert ist, wird sie direkt auf dem Buildserver `sdev` getestet.

Es gibt zum Zeitpunkt dieser Arbeit keine Möglichkeit, die Buildskripte mit ihren Funktionen in einer Testumgebung auf einem anderen Server oder lokal zu testen, da es für sie noch keine Container-basierte Testumgebung gibt, wie sie für einige der anderen Komponenten schon vorhanden ist. Des Weiteren werden beim Ermitteln der Branches und dem resetten des Git Repositorys (Unterabschnitt 4.4) sehr große Datenmengen verarbeitet, welches beim Entwickeln und Testen eines lokalen Build-Containers zu extrem langen Wartezeiten führte. So hat das Ausführen von `git reset --hard` auf dem Buildserver wenige Sekunden und im lokalen Build-Container mehrere Minuten benötigt, weshalb die Entwicklung eines lokalen Build-Containers für die Buildskripte im Rahmen dieser Arbeit nach einigen Versuchen abgebrochen werden musste. Es gibt jedoch ein Projekt in der Firma, in dem an einer komplett Container-basierten DEV-Umgebung für die Entwicklung der Buildpipeline und damit auch den Buildskripten gearbeitet wird. Dieses befindet sich aber aktuell noch in der Entwicklung.

Damit direkt auf dem Buildserver entwickelt werden kann, muss vor dem Start eines Tests überprüft werden, ob der Buildserver aktuell belegt ist, da sonst bereits gestartete Builds unbrauchbar werden würden. Hierzu müssen folgende Punkte überprüft werden:

1. Sind auf dem Build-Dashboard aktuell laufende Buildprozesse aufgelistet?

2. Ist per `top` auf dem Buildserver erkennbar, ob derzeit ein Build läuft? (Der Aufruf von `top` zeigt alle aktuell laufenden Prozesse auf dem Buildserver an.)

3. Startet in der nächsten Zeit ein geplanter Test? (Dies ist abzulesen aus einer Übersicht auf dem Build-Dashboard für die Testplanung.)

Erst wenn all diese Punkte nicht zutreffen, kann auf dem Buildserver entwickelt werden. Hierzu wird mithilfe von `ssh`(secureShell) auf den Buildserver zugegriffen. `ssh` ist ein kryptografisches Netzwerkprotokoll, welches einen einfachen, sicheren und verschlüsselten Terminal-Zugriff auf den Server ermöglicht [31]. Auf dem Server wird nun von dem aktuellen Hauptbranch, der standardmäßig ausgecheckt ist, auf den zu testenden Feature-Branch der Optimierung gewechselt. Sobald dieser nun der aktive Branch ist, können die gewünschten Tests, wie z.B. das Starten eines Test-Builds durchgeführt werden. Ein Beispiel hierfür ist:

```
builQftest -branch merge -createbranch -nosave -noinflux -checkout integration
/test
```

Mit diesem Befehl wird der Test-Integration-Branch `test` gebaut. Dabei wird ein neuer Branch erstellt (-createbranch), auf dem alle Branches, die zu dem Test-Integration-Branch gehören, zusammen gemergt werden (-branch merge). Dieser wird jedoch nicht auf dem Fileserver gespeichert (-nosave) und auch keine Statusupdates des Buildvorgangs auf dem Dashboard angezeigt (-noinflux), damit der Workflow der anderen Entwickler nicht durch den Buildvorgang beeinträchtigt wird. Sobald dieser Vorgang abgeschlossen ist, muss der Branch sofort wieder auf den aktuellen Hauptbranch, in diesem Fall `master` gewechselt werden. Wenn das nicht der Fall ist, kann es zu schwerwiegenden Folgen kommen, da wichtige Nightly-Builds auf die die Firma angewiesen ist, Fehler enthalten oder sogar völlig defekt sein können, da auf `qtools` ein noch in der Entwicklung befindender Feature-Branch ausgecheckt ist. Im Entwicklungsteam der Firma QFS wird dieser gesamte Prozess metaphorisch auch als „Operation am offenen Herzen“ bezeichnet. Um die Weiterentwicklung zu vereinfachen und vom Produktivsystem zu entkoppeln, wird in der Firma an einer Testumgebung in Form eines DEV-Containers gearbeitet.

Um den Buildvorgang im Anschluss zu analysieren, gibt es Build-Protokolle. Diese werden für alle Builds der letzten vier Monate in einem Ordner auf dem Fileserver des Unternehmens gespeichert, so dass alle Entwickler auf sie zugreifen können. Um noch ausführlichere Protokolle zu erhalten, die nicht nur die Vorgänge aus den Buildskripten, sondern auch der anderen Server der Buildpipeline und deren Kommunikation untereinander protokollieren, muss allerdings wieder ein Zugriff über ssh auf den Buildserver sdev erfolgen, wo diese in einem eigenen Ordner gespeichert sind (hier jedoch nur für die letzten 14 Tage).

Wenn eine Optimierung vollständig implementiert und getestet wurde, muss sie von einem fachkundigem Entwickler in einem Review überprüft werden. Bei erfolgreichem Review wird sie anschließend in den Hauptbranch gemergt. Je nachdem, welche Anforderungen eine Optimierung stellt, wurden zudem am Entwicklungsprozess ggf. leichte Änderungen vorgenommen.

6.3 Refaktorisierung der Buildskripte

Die Buildskripte sind zum Zeitpunkt der Dokumentation teilweise über 20 Jahre alt. Mit der Zeit ist der Code immer komplexer und größer geworden, jedoch wurde alter Code selten entfernt, so dass es heutzutage schwierig ist, den bereits bestehenden Code komplett zu verstehen und nachzuvollziehen zu können. Um die Verständlichkeit der aktuellen Codebasis zu verbessern, sollen veraltete Methoden und Variablen sowie Startargumente, die nicht mehr verwendet werden oder gar nicht mehr funktionieren, entfernt werden.

Um die ungenutzten Methoden zu ermitteln, wurden alle Methoden dahingehend überprüft, ob sie im Code mindestens einmal aufgerufen werden. Des Weiteren sind bei der Dokumentation mehrere Methoden aufgefallen, die in ihrer aktuellen Implementierung keinen Nutzen haben oder nur Dummy-Funktionen sind. Beispiele hierfür sind:

Listing 6: Fehlerhafte Funktionen

```
if update or build:
            # Must lock
            acquireLock() # This is currently a fake

def acquireLock():
    pass

def parseConstraints(constraints):
    logger.MTD("constraints: %s", constraints)
    #... what to do? parse each String constraint and look up the result. Split
         by ',' to iterate
    return constraints
```

Diese beiden Funktionen wurden nie komplett implementiert, aber schon in den Hauptbranch integriert. Im Fall der zweiten Funktion wird der mitgegebene Parameter ohne Änderungen wieder zurückgegeben, was zu Fehlern führen kann, wenn ein anderes Verhalten von der Funktion erwartet wird. Beide können ohne einen Funktionsverlust entfernt und, wenn benötigt, später wieder funktionierend implementiert werden. Bei den Variablen sowie Startargumenten für buildAndTest gibt es ähnliche Fälle. Insgesamt 5 der 96 Parameter werden zwar eingelesen, jedoch wird mit ihnen nie weitergearbeitet. Durch die Entfernung dieser ungenutzten Methoden und Variablen können über 50 Zeilen Code eingespart werden. Um weitere veraltete Startargumente herauszufinden, die zwar funktionieren, aber nicht mehr verwendet werden, wurden mithilfe eines Log-Parsing-Skripts alte Logdateien sowie aktuell verwendete Testbefehle auf dem Hauptserver analysiert und ausgewertet.

6.3.1 Analyse der Startargumente

In dem Ordner, in dem sämtliche Protokolldateien der letzten Builds gespeichert werden, befinden sich zum aktuellen Stand dieser Arbeit 1398 verschiedene Protokolle aus dem Zeitraum vom 07.04.2025 bis zum 07.08.2025. In jedem dieser Protokolle steht in der ersten Zeile der ausgeführte Startbefehl, wie zum Beispiel:

```
5 (12:45:03.988) MainThread qfs.scripts.buildqftest.<module>():
Command: buildQftest -block -branch branch -forcelock -checkout
integration/milestone -createbranch -saveall -build -syncm
```

Diese Zeilen können einfach in einem Skript ausgelesen und daraus alle benutzten Argumente ermittelt werden. Um einen genauen Überblick über alle aktuell verwendeten Argumente zu ermitteln, wurde ein Log-Parsing-Skript implementiert. Ein Log-Parsing-Skript ist ein Programm, welches automatisiert alte Logdateien durchsucht und bestimmte relevante Informationen in strukturierter Form herausschreibt. In diesem Fall wurden für alle 1398 Logdateien, die sich nochmals in eigenen Ordnern befinden, jeweils die erste Zeile eingelesen und alle Argumente herausgeschrieben. Für jedes Argument, welches bereits herausgeschrieben wurde, wird eine Zählervariable für das Argument inkrementiert. Das Ergebnis ist eine Datei mit allen benutzten Argumenten sowie deren Anzahl der Aufrufe vgl. (Unterabschnitt E.1). Diese Liste ist jedoch noch nicht ganz korrekt, da einige `False-Positives` enthalten sind. Das sind Werte, die zwar dem Filter entsprochen haben und als Argumente identifiziert und herausgeschrieben wurden, aber eigentlich gar keine sind. Um sie zu entfernen, wurden die Argumente der Liste in Unterabschnitt E.1 mit den akzeptierten Parametern der Buildpipeline verglichen und alle, die nicht akzeptiert werden, entfernt. Wenn ein unbekanntes Argument mitgegeben, welches von der Buildpipeline nicht akzeptiert wird, schlägt diese sofort fehl und es würde kein Log erstellt. In diesem Fall wurden die fehlerhaften Argumente aus den Namen von Tests oder ähnlichen Variablen, die beim Start mitgegeben wurden, fälschlicherweise ermittelt.

Die Analyse der ermittelten Werte kommt zu folgenden Ergebnissen:
Es werden von den 133 verschiedenen akzeptierten Argumenten nur 48 genutzt, was ca. 36% sind, davon auch nur 38 regelmäßig (10 Argumente unter 5 Benutzungen) was sogar nur 29% sind. Das heißt über zwei Drittel der in der Buildpipeline enthaltenen Startparameter werden nie oder fast gar nicht verwendet. Eine Liste der ungenutzten Argumente ist im Unterabschnitt E.3 aufgeführt. Argumente besitzen in einigen Fällen Aliasse, also weitere Namensvarianten, unter denen sie aufgerufen werden können. Diese sind in Klammern hinter dem ersten Argument vermerkt. Wenn man nun die Aliasse bei der Aufzählung der Argumente herauslässt, also für jede Funktion nur ein Argument zählt, sind es nur 96 verschiedene Argumente, von denen 46 verwendet werden. Das sind mit 47% sogar fast die Hälfe. Es gibt nur zwei Fälle, in denen ein Alias verwendet wurde. Ansonsten wird immer nur ein und die selbe Schreibweise angewandt. Das lässt sich damit begründen, dass in den meis-

ten Fällen nur eines der Aliasse wie z. B. der Shell-Befehl I (Alias für Is -I) unter Linux dokumentiert ist und die anderen Verwendungsmöglichkeiten wie `linhost`, `linhosts`, `lin`, `u` und `lh` für den Entwickler unbekannt sind. Die Liste der ungenutzten Argumente wurde den Entwicklern der Buildpipeline vorgelegt, um sie mit ihnen zusammen zu analysieren und zu entscheiden, welche Parameter noch nützlich sind, aber einfach vergessen wurden und welche keinen Nutzen mehr haben und somit entfernt werden können. Alle Parameter und ihre Funktionsweise wurden in Anhang C dokumentiert.

6.4 Modularisierung der Buildskripte

Die verschiedenen Bestandteile der Buildpipeline wie der Buildserver, der Fileserver oder der Packageserver benötigen alle ihre eigenen Skripte, anhand der sie verschiedene Funktionen ausführen können. Bei der Analyse und Dokumentation der Buildskripte wurde jedoch festgestellt, dass in einzelnen Skripten wie `buildQftest` die Skripte mehrerer Komponenten in nur einer Datei implementiert wurden. Dies führt zu einer vermeidbaren Erhöhung der Komplexität bei der Wartung und Weiterentwicklung einzelner Prozesse. Im Fall des Fileservers gibt es für alle Funktionen eine eigene Builddatei, die verständlich und übersichtlich ist. Die unterschiedlichen Funktionen zum Paketieren der Anwendung, also die Vorgänge, in denen die Betriebssystem-spezifischen Installer für die Installation gebaut werden, sind jedoch alle im Hauptskript des Buildvorgangs implementiert. Um die Verständlichkeit der Buildskripte zu erhöhen, wurden alle Funktionen und Abhängigkeiten, die zu dem Packaging-Vorgang gehören, in eine eigene Datei verschoben.

Hierzu wurde zuerst eine neue `pkgUtil` Datei angelegt, in welche alle zugehörigen Methoden verschoben werden können. Anschließend mussten alle Methodenaufrufe angepasst werden. Bei der Auslagerung der Methoden fiel auf, dass zwar die Helferskripte wie `BuildUtil` Python-Dateien sind, jedoch das Hauptskript `buildQftest` noch als Shell-Skript initialisiert ist. In seiner ersten Zeile (`#!/usr/bin/env qpython3`) wurde der Interpreter, `qpython3` für das restliche Skript gesetzt. Das ist jedoch ein Problem, da aus einem Shell-Skript nur schwierig Variablen oder Funktionen in die `pkgUtil` importiert werden können, da es keine Python-Datei ist. Alle neueren Buildskripte bei `QFS` werden mithilfe eines `Wrappers` aufgerufen, der den Aufruf an die jeweilige Datei weiterleitet. Um die `pkgUtil` sauber implementieren zu können, wurde nun auch hier eine solches kurzes Wrapper-Skript ergänzt:

Listing 7: buildQftest Wrapper Skript

```
#!/bin/bash
qpython3 ../../python/qfs/misc/buildQftest.py "$@"
```

Dieses Skript leitet den Aufruf von `buildQftest` an die neue buildQftest.py Datei weiter und setzt davor den Interpreter für die Ausführung der Datei auf `qpython3`.

Durch diese Änderung können nun jegliche Variablen und Funktionen aus buildQftest einfach exportiert und in den Helper-Skripten importiert werden. Das ist nötig, da zentrale Funktionen für das Logging auf dem Dashboard oder der aktuelle Wert von diversen Variablen in ihnen benötigt werden. Nachdem alle Packaging-Prozesse ausgelagert wurden, konnte die Länge des Hauptskriptes `buildQftest` um über 1000 Zeilen reduziert werden, was ca. 1/3 der gesamten Datei entsprach. Dadurch ist nun sowohl das Hauptskript weniger komplex (und somit auch verständlicher), als auch der Packaging-Prozess.

6.5 Handhabung der Merge-Fehler

Wenn bei dem Mergen des Integration-Branches ein Merge-Fehler auftritt, der auch nicht durch eine Conflict-Resolution gelöst werden kann, wurde dieser Branch bislang nicht mit integriert, sondern es wurden nur die zugehörigen Entwickler des Merge-Requests benachrichtigt. Das hatte zur Folge, dass, wenn ein Entwickler einen Build gestartet hat, er selbst keine Benachrichtigung darüber erhalten hat, wenn ein Branch fehlschlägt, an dem er nicht vermerkt ist. Somit gibt es zwar am Ende einen fertigen Build der Software, jedoch ohne einen Branch, der unter Umständen aber benötigt wurde, und es wissen nur die am Merge-Request beteiligten Entwickler davon. Alle nachfolgenden Tests werden ebenso ohne den Branch durchgeführt. Wenn diese erfolgreich sind, kann fälschlicherweise angenommen werden, dass der Branch keine Testfehler verursacht, obwohl er gar nicht mitgelaufen ist. Um dieses Problem zu lösen, wurde ein neuer Parameter implementiert.

Zusammen mit dem neuen Parameter wird auch das standardmäßige Verhalten der Buildskripte geändert. Ab jetzt wird bei einem Merge-Fehler, der sich nicht lösen lässt, der Buildprozess sofort abgebrochen sowie die Entwickler, die für den fehlschlagenden Branch verantwortlich sind, benachrichtigt. Es wird jedoch eine Möglichkeit benötigt, dieses Verhalten zu ändern, da für alle regelmäßigen gestarteten Builds immer ein fertiger Build sowie dessen Testergebnisse benötigt werden, mit denen das Entwicklungsteam weiterarbeiten kann. Hierfür wurde der Parameter `ignoremergefailure` ergänzt, welcher das alte Verhalten wiederherstellt, aber per Default auf `false` gesetzt ist.

Mit diesem neuen Verhalten werden nicht mit integrierte Branches aufgezeigt und der Build bewusst abgebrochen. Der Entwickler hat jedoch selbst die Möglichkeit, mithilfe des neuen Parameters dennoch einen Build zu erzeugen. Alle regelmäßigen Builds und Tests werden davon nicht beeinträchtigt.

6.5.1 Verbesserung der Fehlermeldungen

Bei der Analyse des Verhaltens von Merge-Fehlern, wurde festgestellt, dass die gesendeten Fehlermeldungen, die die Entwickler erhalten, welche den Build gestartet haben, in einigen Fällen nicht ausreichend sind. In ihnen wurden lediglich die letzten

Zeilen des Protokolls mitgeschickt, die ohne Kontext nur schwer oder sogar falsch zu verstehen sind. Eine Beispiel-E-Mail befindet sich sich in Anhang F. In der ersten Zeile ist die für den Entwickler wichtigste Meldung enthalten, dass der Merge eines Branches fehlgeschlagen ist. In der zweiten Zeile steht jedoch als vermeintliche Begründung, dass der Branch nicht gefunden wurde, also gar nicht vorhanden ist. Das stimmt jedoch nicht und bezieht sich in diesem Fall nur auf das Ergebnis einer lokale Suche nach einer bereits vorhandenen Version des Branches auf dem Buildserver. Dies ist aber für einen Entwickler, der sich nicht mit den Buildskripten auskennt, nicht erkennbar. Die eigentlich wichtigste Information steht am Ende und besagt, dass eine Conflict-Resolution erstellt wurde, welche zu lösen ist. Es ist jedoch nicht aufgeführt, welche Entwickler diese lösen müssen. Für den Entwickler, der den Build gestartet hat und diese E-Mail als Fehlerfeedback erhält, gibt es keine Möglichkeit, daraus zu erfahren, an welchen Entwickler er sich wenden muss, um das Problem möglichst schnell zu beheben.

Um die Fehlermeldung zu verbessern, wurden die fehlenden Informationen in zukünftigen Fehler-E-Mails ergänzt. Der Log an sich bleibt so bestehen, aber es wird darin oben eine kurze Erklärung ergänzt, dass eine Conflict-Resolution für einen fehlschlagenden Branch angelegt wurde und welche Entwickler für diese verantwortlich sind. Somit weiß der Entwickler, der diese Fehlermeldung erhält, sofort, welches Problem vorliegt und an wen er sich wenden muss.

6.6 Stoppen der Buildpipeline

Zum Zeitpunkt der Dokumentation der Buildpipeline gibt es keine Möglichkeit, die Buildpipeline sauber zu beenden, wenn sie einmal gestartet ist. Ein Entwickler kann zwar mit `strg C`, dem sogenannten SIGINT Signal, welches alle laufenden Prozesse in der Shell beendet [32], den aktuellen Buildvorgang von dem Hauptserver pp20 aus beenden, jedoch kann danach kein Build mehr richtig starten. Der Grund dafür ist der in Unterabschnitt 4.4 beschriebene Lock-Mechanismus des Repositorys. Dieser wird beim Beenden über SIGINT nicht wieder entfernt, weshalb alle weiteren Builds sich aufhängen. Das bedeutet, dass ein Build nicht gestoppt werden kann und dass, wenn es ein Entwickler instinktiv mit `strg C` doch tut, die gesamte Buildpipeline ins Stocken gerät. Dieses Verhalten ist auch bekannt und in der Dokumentation zur Buildpipeline auf Confluence in einem kleinen Absatz ganz unten erklärt. In diesem wird auch beschrieben, welche Handlungen ein Entwickler ergreifen muss, falls er `strg C` gedrückt hat. Im ersten Schritt muss er sich auf dem Buildserver `sdev` einloggen und dort die beiden Befehle `killall java make python3` und `rm /build/values/lock.repository.branchprj` ausführen, welche den aktuellen Build komplett beendet und alle Locks entfernt. Der normale Benutzer der Buildpipeline war jedoch noch nie auf sdev und weiß auch nicht unbedingt, wie er genau dorthin kommt. Unabhängig davon erweist sich diese Vorgehensweise im Entwicklungs-Workflow als

sehr umständlich. Um dieses Problem zu lösen, gibt es zwei Möglichkeiten:

Die erste Möglichkeit ist die Implementierung eines neuen Startarguments `stop` . Dieses kann dann von pp20 aus mit `buildAndTest -stop` aufgerufen werden und führt auf dem Buildserver automatisch die oben genannt Befehle aus. Die Implementierung der `stop` Methode und deren Aufruf ist in Listing 8 dargestellt. Mit ihrer Hilfe kann ein Build einfach direkt mit dem Befehl`-stop` beendet oder nach dem beenden mit `strg C` sauber aufgeräumt werden, ohne dass der Benutzter der Buildpipeline mit sdev direkt interagiert.

Listing 8: Einfache stop Methode

```
def stopBuild():
    cmds = [
        "killall java make python3",
        "rm /build/values/lock.repository.branchprj"
    ]
    for cmd in cmds:
        ret = BuildUtil.sshCommand("sdev", cmd)
        if (ret != 0):
            print("Failure during stop")
            sys.exit(1)
    print("stopping build and reseting locks.")
    sys.exit(0)

if __name__ == "__main__":
    for k in list(ap.options.keys()):
            if k == "stop":
                stopBuild()
```

Die zweite Möglichkeit ist die Implementierung eines SIGINT Handlers. Dieser Handler hätte die Aufgabe, alle Aufrufe von SIGINT (`strg C`) auf dem Hauptserver `pp20` nach dem Starten eines Builprozesses abzufangen und anschließend dem Entwickler, der `strg C` aufgerufen hat, die Möglichkeit zu geben, den Prozess wieder weiterlaufen zu lassen oder sauber zu beenden. Hierfür muss der Handler direkt beim Start der Buildprozesse initialisiert werden und die gesamte Zeit auf dem MainThread laufen, um pp20 abhören zu können. Die genaue Implementierung ist in Listing G.2 dokumentiert. Für das Stoppen des Buildvorgangs kann die gleiche Methode wie bei der ersten Möglichkeit verwendet werden. Außerdem müssen Methoden für den Handler an sich, nämlich eine für das Setup und eine für das Aufräumen, ergänzt werden. Die letzten beiden sind nötig, um den bisherigen SIGINT Handler vor der Überschreibung mit dem neuen abzuspeichern und nach den Buildprozessen wiederherzustellen. Des Weiteren wird ein neuer Parameter ergänzt, mit dem diese Logik für einen Build deaktiviert werden kann.

Es wurden beide Varianten der Umsetzung für die Optimierung konzipiert und in jeweils eigenen Branches implementiert. Jedoch wurde vorerst nur die erste Variante in die Buildpipeline integriert, da sie weniger komplex und für die Anforderung der Optimierung ausreichend ist. Sollte sich herausstellen, dass die zweite Variante doch noch benötigt wird, kann sie einfach in den Hauptbranch gemergt werden.

6.7 Auslagern der Conflict-Resolutions

Ein weiterer Punkt, der optimiert werden kann, ist der Speicherort der Conflict-Resolution-Dateien. Diese werden zum Stand der Dokumentation einfach in das Hauptrepository in einem dafür neu erstelltem Ordner `conflict-resolution` unter dem File-hash der fehlschlagenden Datei sowie unter deren Namen abgespeichert. Diese Conflict-Resolutions werden von den zugehörigen Entwicklern zwar immer gelöst, damit die Integration des Branches wieder funktioniert, aber danach häufig nicht entfernt und am Ende mit in den Hauptbranch gemergt. Des Weiteren wäre das Erstellen und Einfügen von Conflict-Resolutions in Fremd-Repositorys, in welchen die Buildskripte in Form einer Toolchain eingesetzt werden, nicht akzeptabel. Deswegen soll die Logik so umgebaut werden, dass die Conflict-Resolutions in einem eigenen Repository abgespeichert werden. Die hierfür neu implementierte Logik ist wie folgt aufgebaut:

Beim Erstellen einer Conflict-Resolution wird versucht, diese in das zugehörige Git-Repository `conflict-resolutions` abzuspeichern. Wenn es dieses noch nicht gibt, muss es von einem Entwickler neben dem angegebenen Hauptrepositorys neu erstellt werden. Dieses wird dann automatisch an die Stelle, wo die Buildskripte ausgeführt werden, geklont. Alle zukünftigen Conflict-Resolutions werden hier gespeichert und anschließend in das Remote-Repository gepusht. Die zugehörigen Entwickler bekommen eine Benachrichtigung, dass diese CR sich dort befindet und gelöst werden muss. Beim nächsten Buildvorgang wird in dem CR-Repository überprüft, ob sich in dem Repository eine Lösung für den Merge-Konflikt befindet, und wenn ja, wird diese eingelesen und mit `rerere` wie in Unterabschnitt 4.4 beschrieben abgespeichert. Die CR kann anschließend nach Belieben gelöscht werden, da sie im Cache des anderen Git-Repositorys als Lösung für den Merge-Konflikt abgespeichert wurde. Die dafür benötigten Methoden sind in Unterabschnitt G.3 aufgelistet. Die alte Logik im Buildskript wurde überarbeitet durch das Einsetzen von Aufrufen dieser Methoden an allen benötigten Stellen.

6.8 Buildpipeline Benutzerguide

Um sowohl die neuen, als auch die bereits vorhandenen Funktionen für die Benutzer und Entwickler der Buildpipeline besser zu dokumentieren, wurde ein neuer ausführlicher Benutzerguide zur Handhabung der Buildpipeline erstellt. Dieser basiert

auf der Dokumentation der Buildpipeline in dieser Arbeit, der bereits vorhandenen firmeninternen Dokumentation sowie den Ergebnissen der empirischen Umfrage.

Die Dokumentation in Abschnitt 4 ist für eine Anleitung technisch zu detailliert, enthält jedoch auch Abschnitte, die den Workflow und die Benutzung der Buildpipeline gut zusammengefasst darstellen. Diese werden ergänzt durch die bestehende Dokumentation auf Confluence. Letztere ist vielmehr eine Ansammlung an Informationen und Beispielen zur Benutzung der Buildpipeline und ist über die Zeit immer mehr gewachsen. Sie beinhaltet zwar die meisten wichtigen Informationen, ist jedoch für den Einstieg zu unübersichtlich und bietet keinen roten Faden. Beide Dokumentationen werden in einem neuen strukturierten Benutzerguide zusammengeführt und mit den Anforderungen aus den Umfrageergebnissen ergänzt. Der neue Aufbau des Benutzerguides ist wie folgt:

Übersicht Die Übersicht ist die Hauptseite des Benutzerguides. In ihr wird ein Überblick über den Nutzen und die Funktionen der Buildpipeline gegeben. Sie bietet einen Einstieg für alle neuen Benutzer oder alte, die etwas nachschauen müssen. In ihr wird ein Überblick über den Workflow sowie verschiedene Anwendungsfälle der Buildpipeline gegeben. Des weiteren werden die Befehlssyntax von buildAndTest und die dazugehörigen am häufigsten verwendeten Startparameter erklärt. Zum Abschluss werden häufige Probleme und deren Lösungen aufgelistet, die man für die Benutzung der Buildpipeline wissen muss. Alle folgenden Unterseiten sind oben in der Hauptseite verlinkt.

Build & Test Anleitung Die erste Unterseite bietet eine Schritt-für-Schritt Anleitung, wie man die Buildpipeline lokal oder auf dem Server ansteuert und benutzt. Es werden diverse Anwendungsbeispiele vorgestellt, die ein Benutzer der Buildpipeline benötigen kann, um Build oder Testdurchläufe zu starten und Installer zu erstellen und zu veröffentlichen.

Befehle und Aliasse Auf der zweiten Unterseite sind alle Befehle mit Anwendungsbeispielen dokumentiert. Ergänzt werden sie durch die Auflistung von vorhandenen Aliasen, welche auf dem Hauptserver `pp20` vorhanden sind und die Befehlssyntax deutlich vereinfachen.

Buildskripte Auf der dritten und letzten Unterseite sind die Abläufe zum Entwickeln und Instandhalten der Buildpipeline bzw. genauer gesagt der Buildskripte festgehalten. Sie ist dafür geeignet, Buildpipeline-Benutzern, die selbst etwas an den Buildskripten weiterentwickeln müssen, einen einfachen Einstieg zu ermöglichen.

Sowohl die Hauptseite als auch alle drei Unterseiten sind im firmeninternen Confluence mit Markdown-Formatierung umgesetzt. Dort sind sie mit jeweils passenden Tags markiert, so dass sie von allen Entwicklern einfach gefunden und verwendet

werden können. Der Benutzerguide bildet die Grundlage für die Dokumentation und Verwendung der Toolchain, welche aus den Buildskripten im nächsten Kapitel konzipiert wird.

7 Konzeption der Toolchain

Nachdem die bestehende Build-Pipeline sowohl analysiert und dokumentiert als auch in zentralen Funktionen und Eigenschaften optimiert wurde, ist damit die Grundlage geschaffen, aus ihr eine eigenständige und klar strukturierte Toolchain zu entwerfen. Ziel dieses Kapitels ist es, einen Einblick in die Konzipierung einer Toolchain aus theoretischer Sicht und der zugehörigen praktischen Umsetzung zu geben. Damit wird der Prozess der Entwicklung von einer komplexen, firmeninternen Pipeline zu einer modularen, weiter verwendbaren Lösung dargestellt.

Unter einer Toolchain versteht man in der Softwareentwicklung eine Aneinanderreihung von mehreren Werkzeugen zu einer geordneten Prozesskette. Mit ihrer Hilfe können komplexe Abläufe wie das kontinuierliche Bauen, Testen und Bereitstellen automatisiert werden. Eine klassische Toolchain besteht auf technischer Ebene aus mehreren Komponenten, wie z. B. der statischen Codeprüfung, Compilern und Laufzeitsystemen, die durch genaue Schnittstellen miteinander verbunden und gesteuert werden. Wichtig hierbei ist die Modularität: Einzelne Komponenten einer Toolchain müssen möglichst einfach untereinander verbunden und ausgetauscht werden können, ohne dass die gesamte Kette neu entworfen werden muss [33].

7.1 Zielsetzung

Die meisten Toolchains werden bei der Entwicklung von komplexer Software eingesetzt, damit die zugehörigen Entwicklungsteams möglichst effizient zusammenarbeiten können. Damit werden sowohl kurze Release-Zyklen ermöglicht als auch durch eine voranschreitende Automatisierung aller Buildprozesse die Qualitätssicherung verbessert [33].

So verhält es sich auch im Falle der zu konzipierenden Toolchain. Sie soll einen neuen entscheidenden Baustein in dem Buildprozess einer Software darstellen. Mit ihrer Hilfe sollen Entwickler ohne großen Aufwand in einem Git-Repository einen Integration-Branch aus beliebig vielen Feature-Branches erstellen können. Ein Ziel es ist hierbei, die Toolchain modular zu halten, so dass sie einfach in komplexen Buildprozessen als ein neuer Schritt eingebaut werden kann, ohne die bereits bestehende Infrastruktur stark anpassen zu müssen. Damit könnte eine Buildpipeline, welche bereits einzelne Branches bauen und testen kann, so erweitert werden, dass vollständige Versionen, basierend auf Integration-Branches als Grundlage für die Builds und Tests herangezogen werden können und dies vollständig automatisiert. Wesentliche Anforderungen an die Toolchain sind also, dass sie so minimalistisch wie möglich gehalten wird und dennoch alle wichtigen Funktionen der Bouquet-Integration beinhaltet und dass sie eine einfache verständliche Schnittstelle für den Benutzer und die Integration in einen bestehenden Prozess bietet.

7.2 Aufbau und Funktion

Nach Appel umfasst eine Toolchain für die Softwareentwicklung in ihrem Kern drei funktionale Schichten: a) ein Analysetool, das die Eigenschaften eines Programms überprüft, b) ein Compiler, der das Programm in eine andere Darstellung überführt, sowie c) eine Laufzeitumgebung oder ein Betriebssystem, welche als Kontext für die Ausführung dient. Entscheidend ist, dass die Schnittstellen zwischen diesen Komponenten präzise spezifiziert sind, da nur so die Wiederverwendbarkeit und Austauschbarkeit gewährleistet werden kann.

Im Zusammenhang mit den verschiedenen Buildprozessen wird meistens von DevOps-Toolchains gesprochen. Unter DevOps sind diverse Praktiken und Tools zu verstehen, mit denen die verschiedenen Prozesse zwischen der Softwareentwicklung und den Administratoren automatisiert und integriert werden können. Bei einer DevOps-Toolchain tritt an die Stelle von Compiler und Betriebssystem die Pipeline als Orchestrator. Sie verbindet verschiedene Werkzeuge wie Versionsverwaltungssysteme, Buildskripte, Test- und Deployment-Umgebungen zu einem weitgehend automatisierten Ablauf [33].

Die im Rahmen dieser Arbeit skizzierte Toolchain beschränkt sich auf die Verbindung von einem Versionsverwaltungssystem und den Buildskripten zur Umsetzung der Bouquet-Integration, da die weiteren Komponenten wie Test- und Deployment-Umgebungen in den meisten großen Projekten und Firmen schon vorhanden sind.

7.2.1 Umsetzung

Die Grundlage für die Toolchain bilden die optimierten Buildskripte. Im ersten Schritt müssen jedoch alle, für die Umsetzung der Bouquet-Integration benötigten Funktionalitäten extrahiert werden, da die Buildskripte mehrere tausend Zeilen Code umfassen. Die Vorgehensweise hierbei war, die bestehenden Buildskripte in ein neues Git-Repository zu klonen und alle nicht benötigten Parameter, Variablen, Methoden und Abhängigkeiten zu entfernen. Im Anschluss musste der verbleibende Code überarbeitet werden, so dass er auch ohne die firmeninternen Abhängigkeiten funktionsfähig ist. Aus dieser Codebasis wurde die neue Toolchain entwickelt.

Dieser Prozess entspricht dem wissenschaftlichen Grundprinzip der Reduktion auf das Wesentliche. Anstatt alle Funktionen einer komplexen Pipeline zu übernehmen, wird eine modulare Kernlösung entwickelt. Sie muss klar definierte Schnittstellen bereitstellen und unabhängig von internen Prozessen lauffähig sein. [34]

Um den Anforderungen aus Unterabschnitt 7.1 gerecht zu werden, wird die Toolchain in mehrere Bestandteile eingeteilt:

Schnittstelle: Als Schnittstelle zwischen der Toolchain und weiteren Buildprozessen wird eine neue zentrale Config-Datei angelegt. In ihr können alle wichtigen Variablen wie der Pfad zum GitLab-API-Endpunkt, der Git-Token zur Authentifizierung

oder der Ordner, in dem alle Buildprozesse ausgeführt werden , erfasst werden. Des Weiteren können hier die Default-Werte der Startparameter gesetzt werden, wie z. B. der Detailgrad des Loggings. Da das statische Hinterlegen von sensiblen Informationen wie einem Git-Token ein mögliches Sicherheitsrisiko für Benutzer darstellen kann, lassen sich alle vertraulichen Informationen mithilfe von Umgebungsvariablen setzten. Diese können beim Start der Buildpipeline an den Buildserver oder Container, wo diese ausgeführt wird, zusammen mit dem Startbefehl übergeben werden. Ein Ausschnitt der Config-Datei befindet sich im Unterabschnitt G.4.

Buildskripte: Der Hauptbestandteil der Toolchain sind mehrere Buildskripte, in denen die benötigte Logik für die Umsetzung der Bouquet-Integration enthalten ist. Sie beinhalten die gesamte Logik für das Ermitteln und Mergen von bestimmten Branches. Des Weiteren werden die akzeptierten Startargumente definiert und verarbeitet.

Git-Verknüpfung: Mithilfe der angegeben Git-Variablen wird beim Starten der Toolchain zunächst der Name des Git-Repositorys, in welchem mithilfe der Bouquet-Integration ein neuer Integration-Branch zusammengeführt werden soll, sowie der Name des Git-Repositorys für die Conflict-Resolutions durch die angegebene Git-API-Schnittstelle ermittelt. Dadurch, dass die GitLab-Api benötigt wird, ist die Toolchain jedoch auf eine Benutzung in GitLab beschränkt. Anschließend wird überprüft, ob im Buildverzeichnis diese Repositorys lokal ausgecheckt sind. Wenn das nicht der Fall ist, wird für das jeweils fehlende Repository die URl zum Klonen ermittelt und dieses in das Buildverzeichnis geklont. Durch diese Vorgehensweise kann die Toolchain auf einem Buildserver oder in einem Container für beliebig viele Projekte verwendet werden. Im weiteren Verlauf der Buildskripte werden mithilfe der GitLab-API alle für den Integration-Branch zugehörigen Merge-Requests ermittelt.

Logging: Das firmeninterne Logging-System wurde beibehalten, da es fest in den Buildprozess eingearbeitet ist und ein gutes Logging-System auch für die Toolchain benötigt wird. Dadurch wird den Benutzern der Toolchain eine Möglichkeit gegeben, die ablaufenden Buildprozesse im Nachhinein in einer Log-Datei analysieren zu können. Das Logging-Level, d. h. wie viel genau protokolliert wird, lässt sich mithilfe eines Parameters angeben. Auch der Name und Speicherort der Logfiles kann mithilfe von mehreren Startparametern gesetzt werden.

Conflict-Resolutions: Für die Conflict-Resolutions muss in der Config-Datei ein weiteres Git-Repository angegeben werden. In diesem werden die zu lösenden CRs erstellt und für alle betroffenen Entwickler zugänglich gemacht. Es gibt die Möglichkeit, das gleiche Git-Repository wie das zu mergende Projekt anzugeben, dann werden die CRs in diesem in einem extra Ordner mit dem Namen `conflict-resolutions`

abgespeichert, so wie es auch zum Stand der Dokumentation der Buildpipeline der Fall war.

Während die Schnittstelle zum Starten der Toolchain durch die Config-Datei gut bereitgestellt werden konnte, stellt die Schnittstelle zum Weiterreichen der Ergebnisse bzw. der zugehörigen Informationen ein größeres Problem dar. Der fertig gemergte Integration-Branch und die Conflict-Resolutions können einfach in die angegebenen Git-Repositorys gepusht werden. Die dazugehörige Benachrichtigung an die Entwickler, welche die CR lösen müssen, jedoch nicht. Diese wurden in der firmeninternen Buildpipeline mithilfe eines eigenen lokalen SMTP-Servers (Simple Mail Transfer Protocol Server) an alle Entwickler per E-Mail geschickt. Die Kommunikation ist jedoch in jedem Entwicklerteam unterschiedlich, weshalb ein SMTP-Server nicht standardmäßig vorausgesetzt werden kann. Um die Toolchain universell einsetzbar zu lassen, wurden die Informationen in das Log eingebaut. An der Stelle, wo die E-Mail an den Entwickler geschickt werden würde, werden jetzt alle Namen der Entwickler und die zugehörigen E-Mail-Adressen (ermittelt über der GitLab-API), als auch der Text der zugehörigen Benachrichtigung protokolliert. Die Log-Dateien werden immer an der gleichen Stelle gespeichert und können mit geringem Aufwand automatisiert eingelesen werden. Dadurch kann jedes Entwicklerteam die Namen und E-Mail-Adressen der Entwickler ermitteln und anschließend selbst entscheiden, wie es weiter vorgeht, um diese zu benachrichtigen.

Die Toolchain und alle ihre Bestandteile wurden während der Entwicklung kontinuierlich auf ihre korrekte Funktionsweise getestet. Des Weiteren wurde sie nach Abschluss der Entwicklung in einer unternehmensunabhängigen Testumgebung überprüft, ob die einzelnen konzipierten Bestandteile die gestellten Anforderungen erfüllen und die Toolchain die Bouquet-Integration umsetzt.

7.3 Möglichkeiten zur Veröffentlichung

Ein zentrales Ziel der Arbeit besteht darin, die entwickelte Toolchain nicht nur firmenintern zu optimieren, sondern auch extern verfügbar zu machen. Wissenschaftliche Studien betonen die Bedeutung von wiederverwendbaren Open-Source Toolchains, um die Verbreitung von neuen DevOps-Praktiken in der Entwickler-Community zu fördern [35].

Die geplante Veröffentlichung könnte durch Bereitstellung der Toolchain als GitLab-Repository erfolgen, sodass Anwender sie wie ein normales Git-Projekt klonen und direkt in eine bestehende Buildpipeline integrieren können. Des Weiteren könnte die Toolchain von GitLab aus über ssh einfach auf einen Buildserver geklont und dort direkt eingesetzt werden.

Es bestehen allerdings zwei entscheidende Einschränkungen:

Reifegrad: Die Toolchain wurde zwar auf deren Funktionalität ausführlich getestet, jedoch nur in einem kleinen Softwareprojekte mit einer geringen Anzahl an Feature-Branches und somit wenigen Mergekonflikten. Dass dieses Grundprinzip der Bouquet-Integration auch in großen Git-Repositorys funktioniert, ist zwar durch die Funktionalität der firmeninternen Buildpipeline gegeben. Das bedeutet jedoch nicht, dass dies auch für die extrahierte Toolchain gilt. Um die Skalierbarkeit auf große Softwarevorhaben zu überprüfen, fehlen jedoch die benötigten Testprojekte, anhand derer die Funktionalität sichergestellt werden kann. Erst nach der Durchführung entsprechender Tests kann die Toolchain als hinreichend erprobt und ohne Einschränkungen freigegeben werden.

Rechtliche Rahmenbedingungen: Die Grundlage der Toolchain sind Buildskripte, die in der Firma QFS entwickelt worden sind. Sie sind zwar angepasst und überarbeitet worden, enthalten jedoch immer noch große Teile des originalen Codes. Dieser proprietäre Code gehört der Firma QFS und kann nicht ohne eine Freigabe des Unternehmens erfolgen, welche zum Zeitpunkt der Arbeit noch nicht erteilt wurde.

8 Evaluation der Ergebnisse

In diesem Kapitel werden die herausgearbeiteten Ergebnisse der letzten Kapitel analysiert und beurteilt. Hierbei werden für die einzelnen Schritte die Ergebnisse jeweils kurz zusammengefasst und anschließend evaluiert.

8.1 Dokumentation

Im ersten Schritt wurde die Buildpipeline der Firma QFS mit dem Schwerpunkt auf den Buildskripten ausführlich dokumentiert. Es wurden sowohl die technischen Abläufe der Bouquet-Integration als auch der zugehörige Workflow zum Arbeiten mit dieser dargestellt.

Die Bouquet-Integration bietet im Vergleich zu anderen Verfahren für die kontinuierliche Integration eine Möglichkeit, Feature-Branches dynamisch in regelmäßigen Builds gemeinsam zu testen. In den analysierten bestehenden Ansätzen werden die Feature-Branches entweder nur einzeln gebaut und getestet oder direkt vor dem Merge mithilfe einer Merge-Queue (Unterunterabschnitt 3.3.1).

Die Dokumentation der Bouquet-Integration hat gezeigt, dass der Ansatz zwar funktioniert, aber sehr auf die spezifischen Bedürfnisse des Unternehmens zugeschnitten ist. Durch diese ausführliche Dokumentation wird die Einarbeitung zukünftiger Entwickler in die Buildpipeline erheblich erleichtert und eine Basis für Weiterentwicklungen geschaffen. Des Weiteren wurde durch die Dokumentation in dieser Arbeit das Verfahren das erste Mal für Entwickler außerhalb des Unternehmens QFS zugänglich gemacht.

8.2 Empirische Umfrage und Optimierungen

Die Ergebnisse der empirischen Umfrage im Entwicklerteam haben eine Reihe von Optimierungen für die Buildpipeline erbracht. Diese wurden entweder direkt im Zusammenhang mit dieser Arbeit durchgeführt oder können in der Zukunft durch die Firma QFS noch durchgeführt werden. Die Ergebnisse der Umfrage mit besonderem Fokus auf den Änderungen an der Buildpipeline sind den Entwicklern in einer DEV-Schulung vorgestellt worden. Dadurch konnte allen Entwicklern die neuen Funktionen und Änderungen an der Buildpipeline vorgeführt sowie direktes Feedback gegeben werden, ob die neuen Funktionalitäten die in der Umfrage geforderten Änderungen erfüllen.

Das Feedback der Entwickler ergab, dass durch die Optimierung der Buildpipeline die Benutzung in diversen Aspekten vereinfacht und verbessert wurde. Einige Beispiele hierfür sind:

Fehlende Dokumentation: Mehrere Benutzer haben angemerkt, dass die Dokumentation an vielen Stellen unzureichend oder unübersichtlich ist und teilweise sogar ganz fehlt. Hierfür wurde eine neuer Benutzerguide erstellt, der die Informationen zur Benutzung der Pipeline in sinnvolle Abschnitte gliedert. Mithilfe des neuen Benutzerguides lassen sich sich bestimmte Informationen durch eine bessere Benutzerführung schneller finden und neue Benutzer können durch eine genaue Schritt-für-Schritt-Anleitung und praxisnahen Beispielen ohne die Hilfe anderer Entwickler die Buildpipeline benutzen.

Stoppen der Pipeline: Einige Benutzer haben angegeben, dass die Buildpipeline nicht ohne Probleme gestoppt werden kann und ein instinktives Stoppen mit `strg C` die Buildpipeline so beschädigt, dass sie ohne weiteres Handeln nicht mehr funktioniert. Dieses Problem wurde mithilfe eines neuen `-stop` Arguments gelöst, mit welchem sich die Buildpipeline sauber beenden lässt. Die Benutzung dieses neuen Befehls ist sehr trivial und wurde sofort in den Workflow beim Arbeiten mit der Pipeline übernommen. Diese Optimierung hat ausschließlich positives Feedback bekommen und verhindert Angst- und Stresszustände beim versehentlichen Drücken von `strg C`.

Buildskripte vereinfachen: Die Entwickler, die an den Buildskripten mitentwickeln, gaben an, dass die Arbeit an diesen schwierig ist, da sie über die Zeit sehr groß und komplex geworden sind. Durch eine Modularisierung und Refaktorierung der Buildskripte wurde versucht, das Problem zu verbessern. Dadurch sind die Buildskripte kürzer geworden, da alter ungenutzter Code sowie Startargumente entfernt und die Buildskipte thematisch passend weiter unterteilt wurden. Durch diese Änderungen wurden sie verständlicher und die Arbeit an einzelnen Komponenten wurde verbessert. Die Buildskripte der Buildpipeline sind jedoch weiterhin sehr komplex und relativ schwer zu verstehen. Um sie für neue Entwickler noch zugänglicher zu machen und die Wartbarkeit deutlich zu erhöhen, müssen ein Großteil der ungenutzten Startargumente und deren zugehörige Logik entfernt werden. Um das zu ermöglichen, wurden die genutzten Startargumente statistisch analysiert und der Firma QFS präsentiert. Es wurden jedoch zum derzeitigen Stand der Arbeit vorerst keine weiteren Startargumente entfernt.

Alle Optimierungen, die in dieser Arbeit nicht umgesetzt wurden, weil sie entweder nicht umsetzbar oder thematisch nicht passend sind, konnten dem Unternehmen QFS weitere Verbesserungspotentiale der Buildpipeline aufzeigen. Insgesamt wurden sich deutlich mehr Verbesserungen an den anderen Komponenten der Buildpipeline, als an den Buildskripten gewünscht, was sich aber auch auf das Verhältnis der Nutzergruppen zurückführen lässt. Insbesondere bei der Bedienung der Buildpipeline über `pp20` wurde sich von mehreren Benutzern eine Grafikoberfläche statt der Befehlseingabe in der Shell gewünscht, was die Bedienung deutlich erleichtern könn-

te. Das zeigt, dass zwar die Funktionsweise und Dokumentation der Buildskripte zentrale Bestandteile einer guten Buildpipeline sind, die Schnittstellen für die Benutzer aber nicht vernachlässigt werden sollten, da sie die Personen sind, die mit ihr arbeiten müssen.

8.3 Toolchain

Die aus den Buildskripten extrahierte Toolchain besitzt alle benötigten Funktionen, um die Bouquet-Integration auch außerhalb des Unternehmens einsetzen zu können. Es wurden Schnittstellen konzipiert, die eine nahtlose Einbindung in bereits bestehende externe Buildpipelines ermöglicht. Diese sind gut dokumentiert, so dass sowohl die Integration der Toolchain, als auch die Benutzung einfach möglich ist. Der Code ist im Vergleich zu den komplexen Buildskripten sehr minimalistisch, wodurch eine verbesserte Wartbarkeit erreicht wird.

Durch die zum Stand der Arbeit aktuellen Einschränkungen aus Unterabschnitt 7.3, kann die extrahierte Toolchain jedoch noch nicht veröffentlicht werden. Sie kann dennoch als Referenzarchitektur für eine modulare CI/CD-Toolchain dienen, mit der eine kontinuierliche Integration von langfristigen Feature-Branches ermöglicht wird.

Zusammenfassend konnte aus den Buildskripten der Firma QFS eine alleinstehende, modulare Toolchain konzipiert werden, die alle Funktionen der Bouquet-Integration beinhaltet. Durch eine klare Definition der Schnittstellen ist eine einfache Einbindung in bestehende Buildprozesse möglich. Eine Veröffentlichung der Toolchain ist jedoch zum jetzigen Stand der Arbeit durch fehlende Testmöglichkeiten und rechtliche Hürden beschränkt.

9 Fazit und Ausblick

Anhand der Dokumentation und Optimierung der Bouquet-Integration sowie der Konzipierung einer zugehörigen Toolchain setzte sich diese Arbeit mit einem neuen Ansatz zur kontinuierliche Integration von langfristigen Feature-Branches auseinander. Im folgenden Kapitel werden die daraus resultierenden Schlussfolgerungen erläutert.

9.1 Chancen und Grenzen der Bouquet-Integration

Die Bouquet-Integration ist ein fester Bestandteil der Buildpipeline des Unternehmens QFS und bietet eine einzigartige Möglichkeit, langfristige Feature-Branches kontinuierlich zu integrieren. Das wird zum aktuellen Stand der Technik in den vielen großen Softwareentwicklungen nicht gemacht, da sowohl die Verfahren fehlen, als auch eine komplexe Infrastruktur benötigt wird. Die gängigsten Verfahren greifen auf Prozesse wie Merge-Queues zurück, wobei die Feature-Branches nicht kontinuierlich, sondern lediglich am Ende ihrer Entwicklung vor dem Mergen getestet werden. Es findet jedoch kein Testen von Wechselwirkungen zwischen den Feature-Branches während der Entwicklung statt, was zu einem drastischen Mehraufwand beim Mergen des Branches führen kann. Es gibt jedoch mit `zuul` auch ein Tool, welches ähnlich wie die Bouquet-Integration funktioniert und genau dieses Testen der Wechselwirkungen ermöglicht. Es ist scheint allerdings nicht weit verbreitet zu sein und wird hauptsächlich in Open-Source-Projekten eingesetzt. Somit stellt die Bouquet-Integration eine weitere, aber teilweise auch eine neue Möglichkeit dar, langfristige Feature-Branches zu ermöglichen und nicht wie Meta und Google sich auf kurzlebige Feature-Branches zu beschränken.

Zusammenfassend wurde festgestellt, dass es zwar Möglichkeiten für die kontinuierliche Integration langfristiger Feature-Branches gibt, diese jedoch nur wenig verbreitet sind. Die These `H1 Feature-Branches` konnte damit bis auf einer Ausnahme bestätigt werden und es wurde gezeigt, dass ein Verfahren wie die Bouquet-Integration auch außerhalb des Unternehmens QFS zur Verbesserung der Qualitätssicherung beitragen kann.

Die Dokumentation der Buildpipeline hat aufgezeigt, dass durch einen geschickten Einsatz von `git rerere` und das Verwenden von Conflict-Resolutions eine Möglichkeit geschaffen wurde, einen Merge-Konflikt für einen Branch nur einmal lösen zu müssen und ihn anschließend in beliebig viele Builds zu integrieren. Dadurch können Feature-Branches schon frühzeitig ohne einen Mehraufwand mit integriert werden. Das wiederum ermöglicht es, Merge-Konflikte und Wechselwirkungen mit anderen Branches beim Bauen und Testen aufzuzeigen, womit die These `H2 Bouquet-Integration` belegt ist.

Die Methode der empirischen Umfrage hat sich als erfolgreich erwiesen, da aufgezeigt wurde, dass diverse Optimierungen für die Buildpipeline benötigt werden, offensichtlich eine Nachfrage besteht und konkrete Optimierungspotentiale aus den Ergebnissen ermittelt werden konnten. Diese konkreten Optimierungsvorschläge wurden anschließend konzipiert und in der Buildpipeline implementiert. Auch wenn nicht alle Optimierungen implementiert werden konnten, da einige entweder Komponenten außerhalb des Kontextes dieser Arbeit betroffen haben oder nicht umsetzbar sind, haben die Optimierungen, die umgesetzt wurden, zu einer Verbesserung der Buildpipeline beigetragen. Es wurde das Arbeiten mit und das Entwickeln an den Buildskripten deutlich erleichtert, welches die These `H3 Umfrage` belegt.

Damit die These `H4 Toolchain` belegt werden kann, wurde die Toolchain für die Extrahierung der Bouquet-Integration aus den Buildskripten nicht nur konzipiert, sondern auch implementiert. Dadurch konnten die Funktionalität der entwickelten Konzepte in einer unternehmensexternen Testumgebung und somit eine erfolgreiche Extrahierung der Bouquet-Integration nachgewiesen werden. Obwohl die grundlegende Funktionalität durch Tests sichergestellt wurde, kann die Toolchain zum Stand der Arbeit jedoch nicht in anderen Unternehmen als QFS eingesetzt werden, da unzureichende Testumgebungen und rechtliche Anforderungen eine Veröffentlichung einschränken. Zusammengefasst wurde die These `H4 Toolchain` bestätigt. Das entworfene Konzept der Toolchain kann auch ohne eine Veröffentlichung zusammen mit der ausführlichen Dokumentation der Buildprozesse als Referenz für andere Toolchains dienen, mit denen die `Bouquet-Integration` und damit die kontinuierliche Integration von langfristigen Feature-Branches ermöglicht wird.

9.2 Ausblick auf zukünftige Entwicklungen

Auch wenn die Toolchain zum heutigen Stand der Arbeit nicht veröffentlicht werden kann, wird eine Veröffentlichung und Verbreitung weiterhin angestrebt und soll in Zusammenarbeit mit dem Unternehmen QFS in der Zukunft umgesetzt werden. Hierfür ist eine Testphase für ausgewählte interessierte Kunden des Unternehmens denkbar, um die Funktionalität und Robustheit der Toolchain weiter zu prüfen.

Die Toolchain ist zum Stand dieser Arbeit auf GitLab als Plattform des Versionsverwaltungssystems beschränkt, da die GitLab-API zum Abrufen diverser Informationen benötigt wird. Eine Erweiterung der Toolchain für eine GitHub Unterstützung ist jedoch vorgesehen, damit die Benutzung nicht durch vorhandene Infrastrukturen wie der benutzten Plattform für Git eingeschränkt wird.

Des Weiteren können Funktionen aus den Buildskripten, die für die Bouquet-Integration nicht notwendig sind, aber trotzdem beim Entwickeln benötigt, in der Toolchain ergänzt werden. Ein Beispiel hierfür ist das Bauen und Testen eines einzelnen Branches. Dadurch kann die Toolchain zu einem Tool erweitert werden, welches nicht nur die Bouquet-Integration umsetzt, sondern alle Prozesse, die beim Arbeiten

mit Feature-Branches benötigt werden.

Die Optimierungen, die bisher nicht umgesetzt wurden, können sowohl im Rahmen des Unternehmens als auch in der Toolchain ergänzt werden, um die Benutzung weiter zu verbessern. Ein Beispiel hierfür wäre die Integration einer grafischen Oberfläche zur Steuerung der Buildpipeline im Unternehmen sowie zur Steuerung der Toolchain. Dadurch würde die Benutzung der Bouquet-Integration weiter vereinfacht und in Folge dessen die Menge der möglichen Nutzer erhöht.

Um die Bouquet-Integration weiter zu verbreiten, sind das Veröffentlichen von Artikeln in technischen Zeitschriften auf Grundlage dieser Arbeit sowie Präsentationen auf Entwickler-Konferenzen denkbare Optionen.

A Thesenpapier

H1 Feature-Branches: Es wird angenommen, dass die Wechselwirkungen langfristiger Feature-Branches mit bestehenden Ansätzen der kontinuierlichen Integration (CI) nicht ausreichend getestet werden können. Um diese These zu überprüfen, werden zunächst die theoretischen Grundlagen der CI ermittelt und anschließend wissenschaftliche Ansätze und aktuelle Verfahren in der Industrie in Bezug auf die Möglichkeit analysiert, langfristige Feature-Branches kontinuierlich zu integrieren.

H2 Bouquet-Integration: Es wird angenommen, dass die im Unternehmen entwickelte Buildpipeline es ermöglicht, basierend auf dem Verfahren der Bouquet-Integration Wechselwirkungen zwischen langfristigen Feature-Branches frühzeitig zu erkennen und zu beheben. Diese Annahme wird überprüft durch die Dokumentation des Aufbaus sowie der Funktionsweise der Buildpipeline und des dazugehörigen Workflows. Die Informationen für die Dokumentation werden während Testdurchläufen und auf Basis der Buildskripte ermittelt.

H3 Umfrage: Das Entwicklerteam des Unternehmens QFS besitzt langjährige Erfahrungen bei der Benutzung und Entwicklung der Buildpipeline zur Umsetzung der Bouquet-Integration. Daher wird angenommen, dass eine gezielte empirische Umfrage im Entwicklerteam konkrete Optimierungspotenziale in der Buildpipeline aufdeckt, mit denen die Buildpipeline verbessert werden kann. Um das zu überprüfen, wird eine auf das Entwicklerteam zugeschnittene empirische Umfrage konzipiert und durchgeführt. Die Ergebnisse werden anschließend sowohl statistisch analysiert, als auch inhaltlich hinsichtlich möglicher Optimierungen ausgewertet. Die ermittelten Optimierungen werden in der Buildpipeline integriert, um eine mögliche Verbesserung der Buildpipeline aufzuzeigen.

H4 Toolchain: Die Bouquet-Integration kann mithilfe einer modularen Toolchain auch außerhalb des Unternehmens QFS eingesetzt werden. Um diese These zu überprüfen, wird zunächst eine modulare Toolchain mit der Zielsetzung konzipiert, die Bouquet-Integration aus den komplexen unternehmensinternen Buildskripten zu extrahieren. Anschließend wird analysiert, wie diese Toolchain eingesetzt werden kann und ob sie die Bouquet-Integration in externen Git-Projekten ermöglicht. Dies wird mit lokalen Tests überprüft.

B Literaturverzeichnis

Literatur

[1] Scott Chacon und Ben Straub. *Pro git*. Springer Nature, 2014.

[2] Martin Fowler und Matthew Foemmel. *Continuous integration*. `https://martinfowler.com/articles/continuousIntegration.html`. [Online, zugegriffen am 01.07.2025].

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

[4] Vincent Driessen. *A successful Git branching model*. `http://nvie.com/posts/a-successful-git-branching-model`. [Online, zugegriffen am 02.07.2025].

[5] Martin Fowler. *Branching Patterns - Trunk Based Development*. [Online, zugegriffen am 02.07.2025]. URL: `%5Curl%7Bhttps://martinfowler.com/articles/branching-patterns.html%7D`.

[6] Paul M Duvall, Steve Matyas und Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[7] Jez Humble und David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Kapitel 2. Pearson Education, 2010.

[8] Christian Bird u. a. "The promises and perils of mining git". In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE. 2009, S. 1–10.

[9] Tom Mens. "A state-of-the-art survey on software merging". In: *IEEE transactions on software engineering* 28.5 (2002), S. 449–462.

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

[11] Barry Boehm und Victor R Basili. "Software defect reduction top 10 list". In: *Foundations of empirical software engineering: the legacy of Victor R. Basili* 426.37 (2005), S. 426–431.

[12] Michael Hilton u. a. "Usage, costs, and benefits of continuous integration in open-source projects". In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. Association for Computing Machinery, 2016, S. 426–437.

[13] Maximilian Jungwirth, Martin Gruber und Gordon Fraser. “Improving Merge Pipeline Throughput in Continuous Integration via Pull Request Prioritization”. In: *arXiv preprint arXiv:2508.08342* (2025).

[14] Meta. *A Meta developer's workflow: Exploring the tools used to code at scale*. https://developers.facebook.com/blog/post/2022/11/15/meta-developers-workflow-exploring-tools-used-to-code/. [Online, zugegriffen am 16.07.2025].

[15] Rachel Potvin und Josh Levenberg. “Why Google Stores Billions of Lines of Code in a Single Repository Google's monolithic repository provides a common source of truth for tens of thousands of developers around the world.” In: *Software and Systems Modeling* 59.7 (2016), S. 78–87.

[16] The Kubernetes Authors. *A brief guide about Prow and its ecosystem*. https://developers.facebook.com/blog/post/2022/11/15/meta-developers-workflow-exploring-tools-used-to-code/. [Online, zugegriffen am 17.07.2025].

[17] Bors-NG. *Setting up bors*. https://docs.prow.k8s.io/docs/overview/. [Online, zugegriffen am 17.07.2025].

[18] Zuul project contributors. *Zuul - A Project Gating System*. https://zuul-ci.org/docs/zuul/latest/. [Online, zugegriffen am 17.07.2025].

[19] Timothy Kinsman u. a. “How do software developers use github actions to automate their workflows?” In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, S. 420–431.

[20] Microsoft Corporation. *Was ist Azure Pipelines?* https://learn.microsoft.com/de-de/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops. [Online, zugegriffen am 17.07.2025].

[21] Atlassian. *Get started with Bitbucket Pipelines*. https://support.atlassian.com/bitbucket-cloud/docs/get-started-with-bitbucket-pipelines/. [Online, zugegriffen am 17.07.2025].

[22] GitLab Docs. *CI/CD pipelines*. https://docs.gitlab.com/ci/pipelines/. [Online, zugegriffen am 20.07.2025].

[23] Nicola Döring und Jürgen Bortz. *Forschungsmethoden und Evaluation*. Springer, 2016.

[24] International Organization for Standardization (ISO). *ISO/IEC 25010:2023 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. https://www.iso.org/standard/78176.html. [Online, zugegriffen am 07.07.2025].

[25] Jennifer Fietz und Jürgen Friedrichs. “Gesamtgestaltung des Fragebogens”. In: *Handbuch Methoden der empirischen Sozialforschung*. Springer, 2019, S. 813–828.

[26] Pia Wagner-Schelewsky und Linda Hering. “Online-Befragung”. In: *Handbuch Methoden der empirischen Sozialforschung*. Springer, 2019, S. 787–800.

[27] Rolf Porst. “Frageformulierung”. In: *Handbuch Methoden der empirischen Sozialforschung*. Springer, 2014, S. 687–699.

[28] *Forschungsbereich Semantik und Pragmatik*. `https://www.leibniz-zas.de/de/forschung/forschungsbereiche/semantik-pragmatik`. [Online, zugegriffen am 22.07.2025].

[29] Philipp Mayring und Thomas Fenzl. “Qualitative Inhaltsanalyse”. In: *Handbuch Methoden der empirischen Sozialforschung*. Springer, 2019, S. 633–648.

[30] Mario Callegaro, Katja Lozar Manfreda und Vasja Vehovar. *Web survey methodology*. Kapitel 7. Sage, 2015.

[31] Daniel J Barrett und Richard E Silverman. *SSH, the Secure Shell: the definitive guide*. Kapitel 1. O’Reilly Media, Inc., 2001.

[32] Inc. Free Software Foundation. *GNU Bash Reference Manual (Edition 5.3)*. `https://www.gnu.org/software/bash/manual/bash.html`. [Online, zugegriffen am 11.08.2025].

[33] Jörn Guy Süß, Samantha Swift und Eban Escott. “Using DevOps toolchains in Agile model-driven engineering”. In: *Communications of the ACM* 21.4 (2022), S. 1495–1510.

[34] David Lorge Parnas, Paul C Clements und David M Weiss. “The modular structure of complex systems”. In: *IEEE Transactions on software Engineering* 3 (2006), S. 259–266.

[35] Deepika Badampudi, Muhammad Usman und Xingru Chen. “Large scale reuse of microservices using CI/CD and InnerSource practices-a case study”. In: *Empirical Software Engineering* 30.2 (2025).

C Liste der Build Parameter

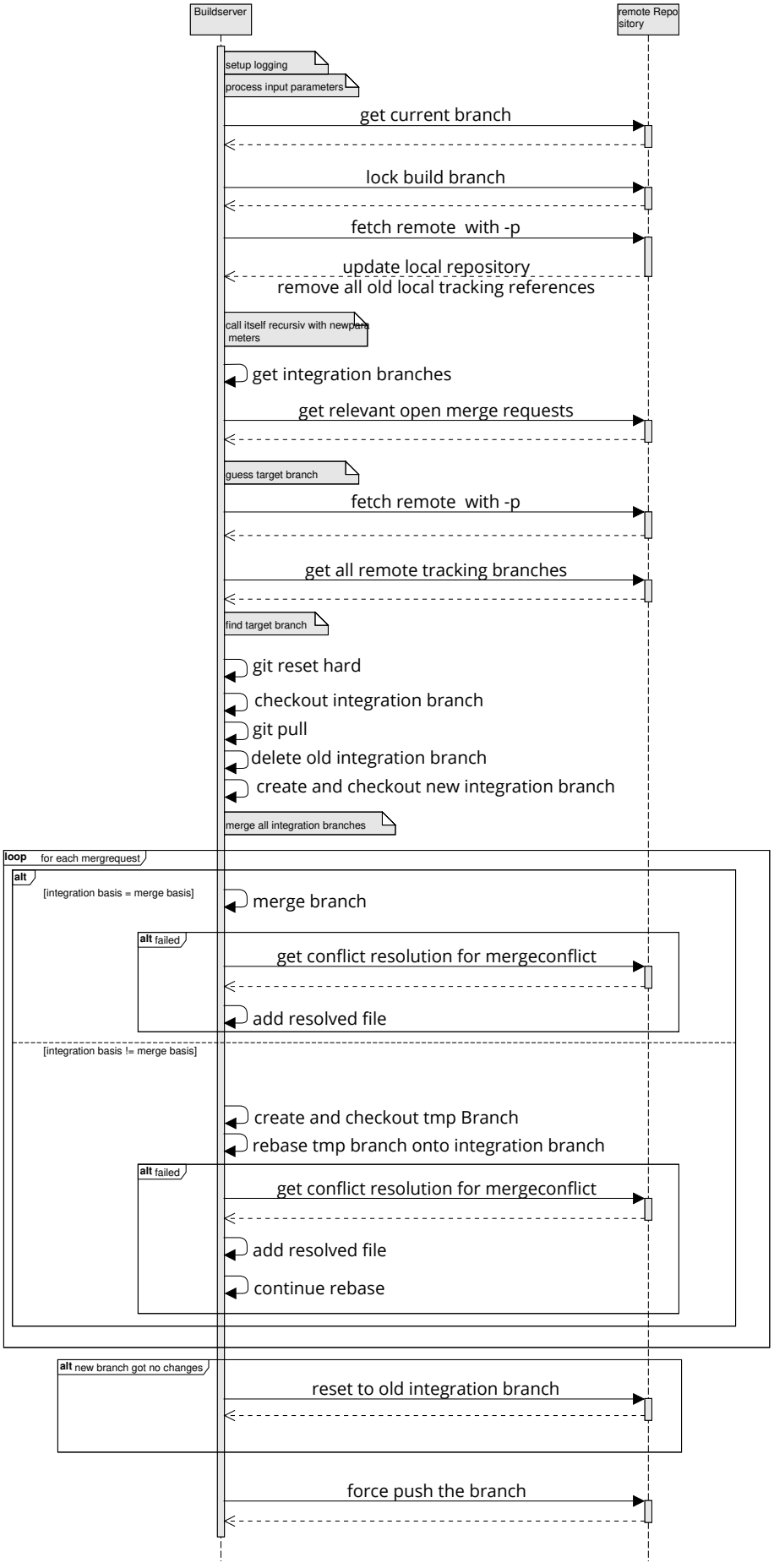
Parameter	Funktion
prj	Name des lokalen Projektverzeichnisses. Wird zur Bestimmung des Buildpfades genutzt.
branchtag, branch	Name des Branches/Tag, der gebaut bzw. getestet werden soll.
display, nodisplay	Legt das Display für GUI-Tests fest.
winhost, winhosts, win, w, wh	Liste oder Anzahl der zu verwendenden Windows-Testhosts. Wird entweder als kommaseparierte Liste oder als Zahl (Anzahl) angegeben.
nowin, nowinhost, nw	Deaktiviert den Einsatz von Windows-Testhosts.
linhost, linhosts, lin, l, u, lh	Liste oder Anzahl der zu verwendenden Linux/Unix-Testhosts.
nolin, nolinhost, nl	Deaktiviert den Einsatz von Linux/Unix-Testhosts.
linfirst	Legt fest, dass Linux-Hosts bei der Verteilung der Tests bevorzugt bzw. zuerst verwendet werden.
machost, machosts, mac, m, mh	Liste oder Anzahl der zu verwendenden Mac-Testhosts.
containerhost, containerhosts, chost, chosts, c, ch	Liste der Container-Hosts für Docker-basierte Testumgebungen.
forcelock	Erzwingt den Build/Test-Vorgang, auch wenn bereits ein Host-Lock besteht.
waitforlock	Wartezeit (in Sekunden), wie lange auf das Freigeben eines gesperrten Hosts gewartet werden soll.
nolock	Deaktiviert das Locking der Testmaschinen.
passlock, nopasslock	Steuert den Lock für passwortbezogene parallele Suite-Ausführungen.
noparts, nopart	keine Verteilung einzelner Suiten auf mehrere Hosts, alle werden auf jeder ausgeführt.
nokillproc	Prozesse, die im Rahmen eines Testlaufs gestartet wurden, werden nach Abschluss nicht beendet.
killtest, killtests, kill	Beendet laufende Tests auf den Hosts.
killtestsfirst	Zuerst werden alle laufenden Tests beendet, danach wird 30 Sekunden gewartet, bevor neue Tests gestartet werden.
runid	Eindeutige Kennung für einen Testlauf (z.B. für Reports); kann automatisch als Zeitstempel generiert oder explizit gesetzt werden.
help	Zeigt eine Hilfemeldung bzw. Usage-Information zu den Parametern an.
update	Führt ein Update der Arbeitskopie (SVN/Git) durch; (teilweise veraltet).

status	Zeigt den aktuellen Status des Builds an.
sync, nosync	Synchronisiert die Testsuiten und gegebenenfalls das „common“-Verzeichnis auf den Testmaschinen.
synccommon, nosynccommon	Synchronisiert ausschließlich das „common“-Verzeichnis. (bzw. unterdrückt es)
synctest, synctests, nosynctest, nosynctests	Aktiviert (oder deaktiviert) die Synchronisation der Testsuiten.
synctestsfrom	Synchronisiert nur diejenigen Testsuiten, die seit einem bestimmten Commit verändert wurden.
build	Führt einen Build durch und kompiliert QF-Test.
nobuild	Verwendet bereits vorhandene Build-Artefakte statt einen neuen Build zu erstellen.
buildhost	Gibt den Host an, auf dem gebaut werden soll (typischerweise sdev18).
devenv	Aktiviert die Entwicklervariante, die eine lokale Build- und Testumgebung einrichtet.
pkghost	Gibt den Host an, der für die Paketerstellung (Erzeugen von Installern etc.) zuständig ist (z.B. pkg.qfs.de).
layerhost	Bestimmt den Host für den Layer-Sync (Synchronisation von Quell- und Dokumentationsdaten mit dem Layer).
nosave	Verhindert das Aktualisieren des Layers.
checkout	Checkt einen bestimmten Branch aus – bestimmt, welcher Branch gebaut wird.
createbranch	Legt einen Integration-Branch an.
abortonmergefailure	Bricht den Build bei Merge-Konflikten ab. (funktioniert nicht)
clean	Löscht temporäre und veraltete Builddaten.
noclean	Führt keine Bereinigung der temporären Daten durch.
cclean	Führt einen „Cache Clean“ durch – löscht nur zwischengespeicherte Inhalte.
doclayer	Verwendet den Dokumentenlayer (z.B. für Release-Dokumentation).
tfg, notfg	Setzt die Zeit der letzten Änderung aller Dateien in einem Git-Arbeitsverzeichnis, auf die des letzten Commits zurück.
savemapandjars	Speichert die Proguard-Buildmap sowie die Original-JAR-Dateien ab.
syncm	Startet nach dem Build das Packaging (Erzeugung von ZIPs, TARs, Installern etc.) und den Upload.
withimage	Bezieht zusätzlich ein Docker-Image oder ein anderes Container-Image mit in den Build ein.
upload	Lädt den Build nach dem Packaging auf einen Server hoch.
artifactory	Legt den Build in Artifactory ab – im Falle eines Release-Builds werden hierfür oft spezielle Einstellungen verwendet.
withlstar	Baut den Build unter Verwendung des LSTAR-Moduls.
withjunit, nojunit, nowithjunit	Erzeugt (bzw. unterdrückt) einen JUnit-Testreport.
frombuild	Verwendet einen bestehenden Build anstelle eines neuen Builds.
fromcommit	Verwendet Build- oder Test-Artefakte aus einem bestimmten Commit.
fromversion	Erlaubt die Angabe einer expliziten Version für Test oder Build.
clearappdata	Löscht AppData-Verzeichnisse.
cleardotswt	Löscht das `.swt`-Verzeichnis.
clearwebdriverdirs	Löscht alle WebDriver-Verzeichnisse auf den Testmaschinen.

clearprofiledirs	Löscht Profilverzeichnisse (z.B. Browserprofile) auf den Testmaschinen.
copyjre	Kopiert die JRE auf die Testmaschinen; optional kann eine spezifische JRE-Variante angegeben werden.
install	Installiert bzw. richtet QF-Test auf den Testmaschinen ein.
installwin, noinstallwin	Installiert (oder unterdrückt die Installation) auf Windows-Maschinen.
installlin, noinstalllin	Installiert (oder unterdrückt die Installation) auf Linux/Unix-Maschinen.
installmac, noinstallmac	Installiert (oder unterdrückt die Installation) auf Mac-Maschinen.
noinstall	Führt auf allen Testmaschinen keine Installation oder Aktualisierung von QF-Test durch.
instrument	Aktiviert die Testinstrumentierung (um beispielsweise Laufzeitdaten zu erfassen).
clearcachedir	Leert das Cache-Verzeichnis.
deltmp, nodeltmp	Gibt an (in Tagen), wie lange temporäre Dateien aufbewahrt werden (0 = nicht löschen).
deltmpwin	Löscht temporäre Verzeichnisse auf Windows-Maschinen.
ziptmp	Zippt temporäre Testverzeichnisse, wenn diese älter als die angegebene Zahl von Tagen sind.
rotatelogs, norotatelogs	Führt (bzw. unterdrückt) eine Logrotation der Testlogs durch.
test	Führt die Tests aus.
notest	Führt keine Tests aus, sondern nur den Build bzw. andere Aktionen.
report	Erzeugt einen HTML/XML-Testreport basierend auf den Testergebnissen.
reporterrorsonly, fullreport	Gibt im Testreport nur Fehler oder den vollständigen Bericht aus.
thumbnails	Sichert Screenshots als Thumbnails für den Report. Falls ein numerischer Wert übergeben wird, ist dieser der Skalierungsfaktor.
nolinklatest	Erzeugt keinen Symlink „latest“ auf den aktuellen Build.
confluence	Lädt die Testberichte in ein Confluence-System hoch.
confluence-OutputStyle	Legt den Ausgabestil (z.B. table) für die Confluence-Notizen fest.
justconfluence	Aktualisiert ausschließlich die Confluence-Berichte, ohne einen neuen lokalen Report zu erzeugen.
showexpected, noshowexpected	Zeigt (bzw. unterdrückt) Suiten, bei denen erwartete Fehler auftreten, im Testreport.
showgood	Listet auch fehlerfreie Suiten im Report bzw. in Confluence auf.
emphasizewin	Hebt Testergebnisse von Windows-Hosts im Bericht besonders hervor.
testdoc	Erzeugt eine Test-Dokumentation (basierend auf den Testfällen).
pkgdoc	Erzeugt eine Dokumentation zur Paketerstellung (z.B. für Installer).
mail, email	Versendet die Testergebnisse bzw. den Build-Report per E-Mail an angegebene Empfänger (als kommaseparierte Liste).
noinflux	Verhindert den Upload von Build- und Testdaten an InfluxDB.
logfile	Gibt den Pfad zur Logdatei an bzw. legt diese fest, falls nicht automatisch erzeugt.
t, dryrun	Führt einen Probelaufdurch, bei dem keine echten Aktionen ausgeführt werden.
rerun	Startet nur bestimmte zuvor ausgeführte Tests erneut (Filter oder Range angeben).

rerunfromconfluence	Startet Testläufe erneut basierend auf den zuvor in Confluence hinterlegten Ergebnissen.
rerunforbranch	Führt einen Rerun für einen bestimmten Branch durch.
rerunwithhost	Erzwingt, dass beim Rerun die Tests auf bestimmten Hosts ausgeführt werden.
rerunall	Führt alle Tests erneut aus, nicht nur die fehlerhaften.
noinitializetests	Überspringt die Initialisierung der Tests (z.B. Setzen von Umgebungsvariablen etc.).
daily	Aktiviert einen speziellen Daily-Testlauf-Modus, der weitere Flags (wie z.B. automatische Bereinigung) setzt.
dailyrerun	Aktiviert einen Daily-Rerun-Modus für Tests.
iterate	Führt ausgewählte Tests mehrfach/iterativ aus.
checkdesktop	Führt eine Desktopzugriffsprüfung durch (wird als Testlauf gestartet).
checkdesktopafter, nocheckdesktopafter	Schaltet die Überprüfung des Desktop-Zugriffs nach dem Test ein oder aus.
param:*	Ermöglicht das Übergeben beliebiger zusätzlicher Parameter an einzelne Hosts; Syntax: `param:key=value`.

D Aufbau des Buildserverskripts



E Verwendete Start Argumente

E.1 Alle ermittelten Argumente unbereinigt

Argument	Anzahl
4	7
5257	1
5689_set	3
6063	4
6063_wda_update_9	1
6101	3
6105_smart_check	1
6111	1
6245	2
64	6
artifactory	5
branch	44
buildhost	1366
checkout	1395
clean	1324
clearappdata	757
clearcachedir	1320
cleardotswt	757
clearprofiledirs	555
clearwebdriverdirs	757
compact	560
confluence	1353
createbranch	1360
daily	145
dailyrerun	151
dev	1
email	1333
emphasizewin	395
forcelock	702
fromcommit	14

Argument	Anzahl
groovydir	1322
groovyrun	1322
i	7
ignoremergefailure	391
image	3
ios	1
iterate	360
jdk	536
justconfluence	1
killtest	1
killtests	2
killtestsfirst	702
l	610
m	221
mgm	79
mobile	83
nobuild	9
noinstall	3
nojunit	695
nopart	105
nosave	9
noshowexpected	142
override	2
performance	105
report	1353
rerun	534
rerunforbranch	2
rerunfromconfluence	267
rotatelogs	595
runid	1059

Argument	**Anzahl**
showgood	721
sorted	241
step	3
suite	800
sync	879
syncm	1360
synctest	4
test	1347
thumbnails	931
u	2
upload	29
variable	1224
w	697
webapi	25
websaga	79

E.2 Alle ermittelten Argumente bereinigt

Argument	Anzahl
artifactory	5
branch	44
buildhost	1366
checkout	1395
clean	1324
clearappdata	757
clearcachedir	1320
cleardotswt	757
clearprofiledirs	555
clearwebdriverdirs	757
confluence	1353
createbranch	1360
daily	145
dailyrerun	151
email	1333
emphasizewin	395
forcelock	702
fromcommit	14
groovydir	1322
groovyrun	1322
ignoremergefailure	391
iterate	360
jdk	536
justconfluence	1

Argument	Anzahl
killtest	1
killtests	2
killtestsfirst	702
l	610
m	221
noinstall	3
nojunit	695
nopart	105
noshowexpected	142
report	1353
rerun	534
rerunforbranch	2
rerunfromconfluence	267
rotatelogs	595
runid	1059
showgood	721
sync	879
syncm	1360
synctest	4
test	1347
thumbnails	931
u	2
upload	29
w	697

E.3 Alle ungenutzten Argumente

cclean
checkdesktop
checkdesktopafter
confluenceOutputStyle
containerhost (containerhosts, chost, chosts, c, ch)
copyjre
deltmp
deltmpwin
devenv
doclayer
frombuild
frompkg
fromversion
help
installlin
installmac
installwin
instrument
linfirst
linhost (linhosts, lh)
linhost2 (lin2, l2, lh2)
linhost3 (lin3, l3, lh3)
linhost4 (lin4, l4, lh4)
logfile
mail
machost (machosts, mac, mh)
logfile
nodeltmp
nodisplay
nolin (nolinhost, nl)
noinitializetests
noinstalllin
noinstallmac
noinstallwin
nolock
nolinklatest
nopasslock
noparts (nopart)
notest
notfg
nosync
nosynccommon
nosynctest (nosynctests)
nowithjunit
nowin (nowinhost, nw)
passlock
pkgdoc
pkghost
prj
reporterrorsonly
rerunall
rerunwithhost
savemapandjars
status
synccommon
synctests
synctestsfrom
tfg
t (dryrun)
update
withimage
withjunit
withlstar

F Fehler-E-Mail an einen Entwickler

Merge of branch TestBranch-02 failed:
error: branch 'TestBranch-02' not found.
Branch 'TestBranch-02' set up to track remote branch 'TestBranch-02' from 'origin'.
Switched to branch 'tmp/priv/marcel/TestBranch-02'
Your branch is up to date with 'origin/priv/marcel/TestBranch-02'.
First, rewinding head to replay your work on top of it...
Applying: TEST-branch for Bachelorthesis
Using index info to reconstruct a base tree...
M test/qftest/gradle-plugin/gradlePlugin.qft
Falling back to patching base and 3-way merge...
Auto-merging test/qftest/gradle-plugin/gradlePlugin.qft

CONFLICT (content): Merge conflict in test/qftest/gradle-plugin/gradlePlugin.qft
error: Failed to merge in the changes.
hint: Use 'git am –show-current-patch' to see the failed patch
Patch failed at 0001 TEST-branch for Bachelorthesis
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase –continue".
You can instead skip this commit: run "git rebase –skip".
To abort and get back to the state before "git rebase", run "git rebase –abort".
Switched to branch 'integration/marcel'
[integration/marcel 6a6] QFT-0 Conflict resolution files for branch TestBranch-02
4 files changed, 1048 insertions(+)

create mode 100644 conflict-resolution/e5cb6ce/base_gradlePlugin.qft
create mode 100644 conflict-resolution/e5cb6ce/gradlePlugin.qft
create mode 100644 conflict-resolution/e5cb6ce/local_gradlePlugin.qft
create mode 100644 conflict-resolution/e5cb6ce/remote_gradlePlugin.qft

G Listings

G.1 Log-Parsing-Skript

```
import os, re
from collections import defaultdict

OUTPUT_FILE = "argumente.txt"

def parse_arguments(line):
    #find 'buildAndTest'
    pattern = r'buildAndTest(.*)'
    match = re.search(pattern, line)
    if not match:
        return []
    after_buildandtest = match.group(1)
    #find all arguments
    arg_pattern = r'-(\w+)\b'
    return re.findall(arg_pattern, after_buildandtest)

def read_log_first_line(log_path):
    try:
        with open(log_path, encoding='utf-8') as f:
            return f.readline().strip()
    except Exception:
        return ""

def main():
    argument_counter = defaultdict(int)
    current_folder = os.getcwd()
    # find the log files
    for entry in os.listdir(current_folder):
        if os.path.isdir(entry) and entry.startswith("250"):
            log_path = os.path.join(entry, "000.log")
            # get the arguments
            if os.path.isfile(log_path):
                first_line = read_log_first_line(log_path)
                args = parse_arguments(first_line)
                for arg in args:
                    argument_counter[arg] += 1
    # write all arguments with their count in the output file
    with open(OUTPUT_FILE, "w", encoding='utf-8') as f:
        f.write("Argumente:\n")
        for arg, count in sorted(argument_counter.items()):
            f.write(f"{count} {arg}\n")

if __name__ == "__main__":
    main()
```

G.2 SIGINT Handler für den Buildprozess

```
import signal

def setup_sigint():
    global oldSigintHandler
    oldSigintHandler = signal.getsignal(signal.SIGINT)
    signal.signal(signal.SIGINT, sigint_handler)

def restore_sigint():
    if oldSigintHandler:
        signal.signal(signal.SIGINT, oldSigintHandler)

def sigint_handler(signum, frame):
    print("\n-----------------------------------------")
    print("STRG-C detected. Do you realy want to stop the build process?")
    answer = input("Terminate build process (y/n)? ")
    if answer.strip().lower() in ("y", "j", "yes"):
        print("\nStart Clean-up-Process")
        stopBuild()
        sys.exit(0)
    else:
        print("Continue")

if __name__ == "__main__":
    nosigint = false
    for k in list(ap.options.keys()):
           if k == "nosigint":
               nosigint = true

    if !nosigint:
        setup_sigint()
    try:
        #...
        #build process
        #...
    finally:
        if !nosigint:
            restore_sigint()
```

G.3 Methoden für die Integration eines CR-Repositorys

```
GIT_REPO_CR = "https://git.company/path/conflict-resolutions.git"

def getConflictResolutionRepo():
    """
    Find the repository for the conflict resolutions.
    If there is no lokal repo, clone it from the remote repo.
    """
    crrepopath = os.path.join(os.path.dirname(os.path.abspath(os.getcwd())), "
        conflict-resolutions")
    if not os.path.exists(crrepopath):
        if not GIT_REPO_CR:
            logger.ERR("Path to conflict-resolution repo is missing or empty!")
            raise Exception("Pls create a new empty git repository for the
                conflict resolutions and add the URL to GIT_REPO_CR.")
        cmd = "git clone %s" % GIT_REPO_CR
        logger.MSG("Cloning repository for conflict resolutions with command %s",
            cmd)
        runAndLog(cmd)
    return crrepopath

def lookupConflictResolutionFile(hash, filename, crrepopath):
    crfile = os.path.join(crrepopath, hash, filename)
    if os.path.isfile(crfile):
        return crfile
    return None

def saveConflictResolutionFiles(filename, basefile, localfile, remotefile,
    conflicted, crrepopath):
    conflictdir = os.path.join(crrepopath, hash)
    if not os.path.exists(conflictdir):
        os.makedirs(conflictdir)

    shutil.copyfile(conflicted, os.path.join(conflictdir, filename))
    shutil.copyfile(basefile, os.path.join(conflictdir, "base_" + filename))
    shutil.copyfile(localfile, os.path.join(conflictdir, "local_" + filename))
    shutil.copyfile(remotefile, os.path.join(conflictdir, "remote_" + filename))

    cmd = "git -C %s add ." % crrepopath
    logger.MSG("Adding files to CR-Repo with command %s", cmd)
    runAndLog(cmd)
    cmd = "git -C %s commit -n --no-edit" % crrepopath
    logger.MSG("Committing conflict-resolution to CR-Repo with command %s", cmd)
    runAndLog(cmd)
    cmd = "git -C %s push" % crrepopath
    logger.MSG("Committing conflict-resolution to CR-Repo with command %s", cmd)
    runAndLog(cmd)
```

G.4 Toolchain Config-Datei

```
"""
Config file to define all default values
"""
# --- Git Settings ---
# All Git Variables can be set through environment variables, with the syntax: BQ_VAR
# Example: ${BQ_GIT_TOKEN}
# URL to the GitLab API endpoint
# Example: "https://gitlab.example.com/api/v4"
GIT_PATH =
# ID of the GitLab project
# Can be found on the "Edit Project" page in GitLab at the top-right under "Project ID".
GIT_PROJECT =
# ID of the GitLab project for the conflict-resolutions (cr)
# GIT_PROJECT_CR = GIT_PROJECT if you want the cr in the same repo as the project
GIT_PROJECT_CR =
# Token to read project merge requests, needs read_api scope
# Example: "glpat-xxxxxxxxxxxxxxxxxxxx"
GIT_TOKEN = ""
# Token to read user info; needs read_api scope
# Example: "glpat-xxxxxxxxxxxxxxxxxxxx"
GIT_TOKEN_BOB = ""
# master Branch in the GIT_PROJECT; base of the integration branch
masterBranch = "master"

# --- Build directory ---
# leave empty for default:
#   - Windows: C:\build
#   - Linux/Unix: /var/tmp/build
# or set an absolute path, e.g. /opt/build
buildDir = ""

# --- Logging ---
# high detail of logging via qflog
logViaQflog = True
logLevel = 10
```

G.5 Beispiel Merge-Konflikt


```
{
  "id": 1,
  "iid": 1,
  "project_id": 1,
  "title": "Test merge for Test branch for Bachelorthesis",
  "description": "",
  "state": "opened",
  "created_at": "2025-06-30T16:38:27.227+02:00",
  "updated_at": "2025-07-02T09:20:40.953+02:00",
  "merged_by": null,
  "merge_user": null,
  "merged_at": null,
  "closed_by": null,
  "closed_at": null,
  "target_branch": "integration/branch",
  "source_branch": "priv/assigne/TestBranch-01",
  "user_notes_count": 0,
  "upvotes": 0,
  "downvotes": 0,
  "author": {
    "id": 54,
    "username": "assigne",
    "public_email": null,
    "name": "Marcel Schmied",
    "state": "active",
    "locked": false,
    "avatar_url": "https://secure.gravatar.com/avatar/1234",
    "web_url": "https://git.url/assigne"
  },
  "assignees": [
    {
      "id": 1,
      "username": "assigne",
      "public_email": null,
      "name": "assigne",
      "state": "active",
      "locked": false,
      "avatar_url": "https://secure.gravatar.com/avatar/1234",
      "web_url": "https://git.url/assigne"
    }
  ],
  "assignee": {
    "id": 1,
    "username": "assigne",
    "public_email": null,
    "name": "assigne",
    "state": "active",
```

```
    "locked": false,
    "avatar_url": "https://secure.gravatar.com/avatar/1234",
    "web_url": "https://git.qfs.de/assigne"
  },
  "reviewers": [],
  "source_project_id": 1,
  "target_project_id": 1,
  "labels": [
    "assigne",
    "No Auto Rebase"
  ],
  "draft": false,
  "imported": false,
  "imported_from": "none",
  "work_in_progress": false,
  "milestone": null,
  "merge_when_pipeline_succeeds": false,
  "merge_status": "can_be_merged",
  "detailed_merge_status": "need_rebase",
  "merge_after": null,
  "sha": "1234",
  "merge_commit_sha": null,
  "squash_commit_sha": null,
  "discussion_locked": null,
  "should_remove_source_branch": null,
  "force_remove_source_branch": true,
  "prepared_at": "2025-06-30T16:38:29.170+02:00",
  "reference": "!1234",
  "references": {
    "short": "!1234",
    "relative": "!1234",
    "full": "qfs/project!1234"
  },
  "web_url": "https://git.url/project/-/merge_requests/1",
  "time_stats": {
    "time_estimate": 0,
    "total_time_spent": 0,
    "human_time_estimate": null,
    "human_total_time_spent": null
  },
  "has_conflicts": false,
  "blocking_discussions_resolved": true
}
```

H Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Bachelorarbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

Leipzig, 01. September 2025 Marcel Schmied

_____

Unterschrift