Westfälische Hochschule

University of Applied Sciences
Gelsenkirchen Bocholt Recklinghausen

Bachelorarbeit

Titel der Arbeit // Title of Thesis

Konzeption und Entwicklung einer Continuous-Integration-Strategie für Kundenprojekte auf Basis der Shopware-Platform

---

Akademischer Abschlussgrad: Grad, Fachrichtung (Abkürzung) // Degree

Bachelor of Science (B.Sc.)

Autorenname, Geburtsort // Name, Place of Birth

Frederik Bußmann, Coesfeld

Studiengang // Course of Study

Informatik.Softwaresysteme

Fachbereich // Department

Wirtschaft und Informationstechnik

Erstprüferin/Erstprüfer // First Examiner

Prof. Dr.-Ing. Martin Schulten

Zweitprüferin/Zweitprüfer // Second Examiner

Martin Knoop

Abgabedatum // Date of Submission

31.08.2023

Eidesstattliche Versicherung

Bußmann, Frederik

Name, Vorname // Name, First Name

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem Titel

Konzeption und Entwicklung einer Continuous-Integration-Strategie für Kundenprojekte auf Basis der Shopware-Platform

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Stadtlohn, den 31.08.2023

Ort, Datum, Unterschrift // Place, Date, Signature

Abstract

Das Ziel dieser Bachelorarbeit ist die Erarbeitung eines geeigneten Konzepts für das Einbinden von Continuous-Development-Techniken in Shopware-Projekten. Insbesondere die Praktiken und Methoden der Continuous Integration (CI) werden im Verlauf der Arbeit untersucht und erläutert. Dabei wird die Bedeutung von CI in der modernen Softwareentwicklung hervorgehoben, besonders im Kontext von Shopware, eine der führenden E-Commerce-Plattformen in Deutschland. Zunächst werden die Konzepte des Continuous Software Engineering vorgestellt und der fachliche Hintergrund für weitere Themen und Aspekte der Arbeit definiert. Die Arbeit beleuchtet außerdem die Herausforderungen und Vorteile, die mit der Implementierung von CI in Shopware-Projekten einhergehen. Ein besonderer Fokus liegt auf der Automatisierung des Build-Prozesses, wodurch eine schnelle und effiziente Integration von Softwarekomponenten ermöglicht wird. Automatisierte Tests und QA-Tools werden eingesetzt, um Standards vorzugeben und die Funktionalität der Software sicherzustellen. Darüber hinaus werden die positiven Auswirkungen von CI auf die Entwicklungszeit von Software und die allgemeine Softwarequalität diskutiert. Neben der theoretischen Grundlagen wird ein praktischer Leitfaden für Entwickler und Unternehmen erstellt, um CI-Techniken in neue und bestehende Shopware-Projekte zu integrieren. Dieser wird anschließend für die erstellte Strategie evaluiert und es werden einige Erkenntnisse und Schlussfolgerungen dargelegt. Insgesamt wird die Implementierung von Continuous Integration in Shopware-Projekten in der Arbeit untersucht, durchgeführt und kritisch bewertet.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API . . . . . Application Programming Interface

AST . . . . . Abstract Syntax Tree

CD . . . . . Continuous Deployment

CDE . . . . . Continuous Delivery

CI . . . . . Continuous Integration

CMS . . . . . Content Management System

CSS . . . . . Cascading Style Sheets

NPM . . . . . Node Package Manager

OS . . . . . Operating System

PHPCS . . . . . PHP_CodeSniffer

PHPMD . . . . . PHP Mess Detector

QA . . . . . Quality Assurance

UI . . . . . User Interface

VCS . . . . . Version Control System

VM . . . . . Virtual Machine

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Rahmen dieser Arbeit werden verschiedene Aspekte betrachtet, um ein Konzept für das Einbinden von Continuous Integration (CI) in Shopware-basierten Projekten zu erarbeiten. Shopware bietet als eine führende E-Commerce-Plattform und eine der bevorzugten Online-Shop-Lösungen in Deutschland[1] eine solide Grundlage für Unternehmen, um im digitalen Raum erfolgreich zu agieren. Durch die gezielte Implementierung von CI-Praktiken in solchen Projekten kann der Entwicklungszyklus effizienter gestaltet und die Qualität des Endprodukts gesteigert werden. Dies unterstützt Unternehmen dabei, ihre Wettbewerbsfähigkeit zu erhöhen und eine agile und reaktionsschnelle Entwicklungsumgebung zu etablieren.

1.1 Motivation

In der heutigen schnelllebigen digitalen Welt ist die Fähigkeit, qualitativ hochwertige Softwareprodukte schnell auf den Markt zu bringen, nicht nur wünschenswert, sondern oft entscheidend für den Geschäftserfolg. Die E-Commerce-Branche, geprägt durch ihre intensive Wettbewerbsdynamik, erfordert von Unternehmen eine kontinuierliche Anpassung und Innovation, um im Markt bestehen zu können. Hierbei nimmt die Effizienz und Effektivität der eingesetzten Softwareentwicklungsmethoden eine zentrale Rolle ein. Um eine möglichst reaktionsschnelle und effektive Umgebung für Entwicklerteams in Shopware-basierten Projekten zu schaffen, können Methodiken des Continuous-Software-Engineering verwendet werden. Die Entwicklung einer robusten, jedoch flexiblen CI-Strategie ist im Hinblick auf sich ständig weiterentwickelnder Technologien und variierender Anforderungen besonders wichtig für die Sicherstellung der Softwarequalität und wird in Zukunft immer relevanter.

1.2 Zielsetzung

Die Entwicklung einer Continuous-Integration-Strategie für auf Shopware basierende Projekte ist das primäre Ziel dieser Arbeit. Die Strategie soll dazu beitragen, die Qualität der Software zu verbessern, die Effizienz des Entwicklungsprozesses zu steigern und letztendlich die Kundenzufriedenheit zu erhöhen. Die nachfolgend definierten Ziele Z_n dienen als Leitfaden für die Konzeption und Entwicklung der CI-Strategie und stellen die geschäftsseitigen Anforderungen von Unternehmen in der E-Commerce-Branche an den Entwicklungsprozess mit Shopware dar:

- Z_1 **Hohe Entwicklungsgeschwindigkeit**

  Die Einführung einer umfangreichen CI-Strategie soll die Effizienz der Softwareentwicklungsteams verbessern und die Zeit bis zum Produkt-Release senken. Eine hohe Entwicklungsgeschwindigkeit sorgt für eine schnellere Auslieferung neuer Features und Fehlerbehebungen und somit zu einer niedrigeren Wartezeit für Kunden.

- Z_2 **Niedrige Fehlerrate**

  CI soll dazu beitragen, Fehler frühzeitig im Entwicklungsprozess erkennen und beheben zu können, was die Qualität des Endprodukts verbessert. Die Stabilität und Qualität der

[1] Vgl. eCommerceDB. *The Most Commonly Used Shop Software Among Online Shops in Germany - Shopware and Salesforce Share Rank No. 1.* 2023.

ausgelieferten Anwendung wirkt sich durch weniger Ausfälle und eine geringere Supportzeit auf die Zufriedenheit von Kunden aus.

- Z_3 **Kontinuierliche Auslieferung neuer Software**

  Die CI-Strategie und die damit verbundenen Prozesse, die sich für Entwicklerteams ergeben, sollen zu einer anpassbaren Entwicklungsumgebung führen. Diese Umgebung soll durch kontinuierliche Weiterentwicklung an die ständig wechselnden Anforderungen der modernen Softwareentwicklung angepasst werden können, was die Wettbewerbsfähigkeit fördert.

Bei der Konzeption sollen diese Ziele verfolgt und die Maßnahmen der zu erarbeiteten CI-Strategie dementsprechend ergriffen werden. Die Strategie soll dabei nicht nur die technischen Aspekte von Continuous Integration berücksichtigen, sondern auch die organisatorischen Veränderungen, die mit der Einführung von CI einhergehen. Darüber hinaus soll die Strategie flexibel genug sein, um sich an zukünftige Veränderungen und Entwicklungen anpassen zu können.

1.3 Struktur der Arbeit

Im Laufe dieser Arbeit wird eine CI-Strategie für Shopware-basierte Kundenprojekte konzeptioniert und entwickelt. Die Arbeit ist in fünf Hauptabschnitte unterteilt, die jeweils unterschiedliche Aspekte des Prozesses abdecken.

Fachlicher Hintergrund

In diesem Abschnitt wird zunächst der theoretische Rahmen für die Arbeit festgelegt. Dies umfasst eine Einführung in das Continuous-Software-Engineering und die Prinzipien und Praktiken von Continuous Integration sowie eine Übersicht über die Shopware-Platform. Der Abschnitt dient dazu, ein grundlegendes Verständnis für die Themen und Technologien zu schaffen, die in der Arbeit behandelt werden.

Analyse und Konzept

Dieser Abschnitt befasst sich mit der Analyse der aktuellen Situation und der Entwicklung eines Konzepts für die CI-Strategie. Dies beinhaltet die Identifizierung von Herausforderungen und Anforderungen, die Berücksichtigung von Best Practices und die Ausarbeitung eines Plans für die Implementierung der Strategie. Die Konzeptionierung stützt sich dabei auf die im vorherigen Abschnitt aufgezeigte Fachliteratur. Der Abschnitt dient als Brücke zwischen Theorie und Praxis und stellt sicher, dass die entwickelte Strategie sowohl fundiert als auch anwendbar ist.

Umsetzung der CI-Strategie

In diesem Abschnitt wird die Umsetzung der CI-Strategie als Fallbeispiel beschrieben. Dies umfasst die Auswahl und Konfiguration der benötigten Tools, die Definition von Prozessen und Workflows, die Implementierung von Automatisierungen und Tests sowie das automatisierte Ausliefern der Software. Der Schwerpunkt liegt hierbei auf der praktischen Umsetzung des zuvor entwickelten Konzepts und dessen Integration in reale Shopware-Projekte.

Evaluierung

Die Auswertung der implementierten CI-Strategie wird im folgenden Abschnitt behandelt. Dabei wird die umgesetzte Strategie im Hinblick auf die im fachlichen Hintergrund aufgezeigten Prinzipien geprüft. Die Ergebnisse dieser Evaluierung werden analysiert und interpretiert, um Rückschlüsse auf den Erfolg der Strategie zu ziehen.

Schlussfolgerung und Ausblick

Der letzte Abschnitt fasst die Ergebnisse der Arbeit zusammen und es werden Schlussfolgerungen über die CI-Strategie und dessen Anwendbarkeit in Shopware-Projekten gezogen. Darüber hinaus wird ein Ausblick auf mögliche zukünftige Entwicklungen und Verbesserungen gegeben. Dieser Abschnitt dient dazu, die Arbeit abzurunden und einen Ausblick auf weitere Forschungs- und Entwicklungsarbeiten in diesem Bereich zu geben.

2 Fachlicher Hintergrund

Für die Erarbeitung einer geeigneten CI-Strategie wird zunächst ein Einblick in die Disziplin des Continuous-Software-Engineering gegeben. Anschließend werden die Begrifflichkeiten und Prinzipien von Continuous Integration definiert und weitere relevante Technologien und Bereiche für die Nutzung von CI erläutert. Darüber hinaus wird eine Übersicht über die Funktionen und Mechanismen der Shopware-Platform gegeben.

2.1 Continuous-Software-Engineering

Continuous-Software-Engineering fasst die Prinzipien der Continuous Integration (CI), Continuous Delivery (CDE) und Continuous Deployment (CD) zusammen. Shahin et al. definieren den Begriff als einen Bereich der Softwareentwicklung, bei dem es um die Entwicklung, Auslieferung und das schnelle Feedback von Software und Kunde geht. Die Disziplin umfasst Geschäftsstrategie und Planung sowie Entwicklung und den Betrieb der Software.[2] Diese kontinuierliche Integration von Software ist sehr kompatibel mit den häufigen Iterationen in der agilen Softwareentwicklung und wurde unter anderem durch die agile Methodik des „Extreme Programming“ bekannt.[3] Nachfolgend werden die Bereiche der agilen Softwareentwicklung und der CI, CDE und CD kurz erläutert.

Agile Software Development

Agile Softwareentwicklung ist ein Ansatz zur Softwareentwicklung, der auf Flexibilität und Kundeninteraktion setzt. Im Gegensatz zu traditionellen, plangetriebenen Methoden, die die Anforderungen und Lösungen am Anfang des Projekts festlegen, erlaubt die agile Methodik Änderungen und Anpassungen während des gesamten Entwicklungsprozesses. Dies wird durch iterative Entwicklung und regelmäßiges Feedback erreicht. Zu den wichtigsten Prinzipien der agilen Softwareentwicklung gehören die kontinuierliche Auslieferung von Software, Offenheit für sich ändernde Anforderungen und enge Zusammenarbeit zwischen Teams und Entwicklern.[4] Gren und Lenberg fassen „Agile“ als die Reaktionsfähigkeit im Hinblick auf sich ständig ändernde Anforderungen und Umgebungen zusammen.[5]

Continuous Integration

Continuous Integration (CI) ist ein Softwareentwicklungsprozess, bei dem Entwickler ihre Änderungen regelmäßig - oft mehrmals täglich - in eine gemeinsam geführte Hauptversion der Software integrieren. Jede dieser Integrationen wird dann von einem automatisierten Build-System überprüft, um sicherzustellen, dass die Änderungen mit der bestehenden Codebase kompatibel sind und keine Fehler verursachen. Dieser Prozess ermöglicht es Teams, Probleme frühzeitig zu erkennen und zu beheben, was die Qualität der Software verbessert und die Zeit bis zur Auslieferung der Software reduziert.[6]

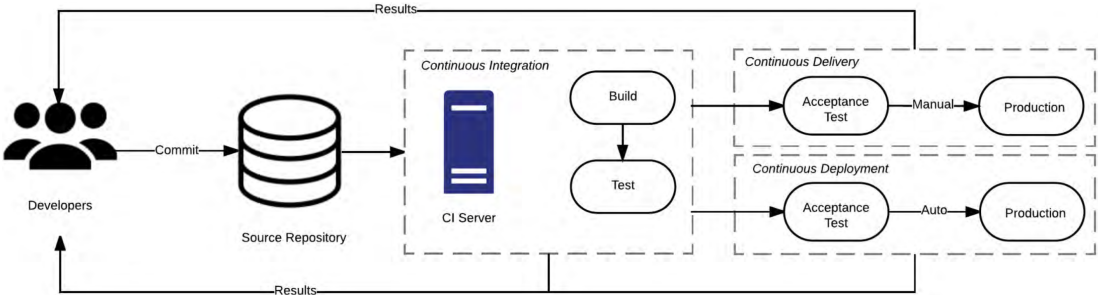
[2] Vgl. Shahin, Ali Babar und Zhu. „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“. 2017, S. 3910–3911.
[3] Vgl. Fitzgerald und Stol. „Continuous software engineering: A roadmap and agenda“. 2017, S. 181.
[4] Vgl. Beck, Beedle, Bennekum et al. *Manifesto for Agile Software Development*. 2001.
[5] Vgl. Gren und Lenberg. „Agility is responsiveness to change“. 2020.
[6] Vgl. Fowler. *Continuous Integration*. 2006.

Continuous Delivery

Continuous Delivery (CDE) erweitert das Konzept der Continuous Integration, indem es sicherstellt, dass die Software stetig in einem Zustand ist, der sicher in die Produktionsumgebung ausgerollt werden kann. Dies wird durch das Einführen von Integrationstests, welche die Funktionalität der vollständigen Software inklusive aller Module testen, erreicht. Das Ziel von CDE ist es, den Prozess der Softwareauslieferung zu beschleunigen und zuverlässiger zu machen, indem menschliche Fehler minimiert und schnelles Feedback über Probleme in der Produktionsumgebung ermöglicht werden.[7]

Continuous Deployment

Continuous Deployment (CD) ist der nächste Schritt nach Continuous Delivery. Bei CD wird jede Änderung, die den automatisierten Testprozess besteht, automatisch in die Produktionsumgebung eingespielt.[8] Dies bedeutet, dass neue Features und Updates mehrmals täglich an die Endbenutzer ausgeliefert werden können, was eine schnelle Reaktion auf Marktbedingungen und Kundenfeedback ermöglicht. Es ist jedoch zu beachten, dass CD eine hohe Reife der Entwicklungsprozesse und Testautomatisierung erfordert, um die Anzahl der in der Produktionsumgebung aufkommenden Fehler zu minimieren. Der Begriff „Deployment“ wird außerdem als Synonym für das Ausliefern von Code an eine Umgebung verwendet. Der Zusammenhang zwischen Continuous Integration, Delivery und Deployment wird in Abbildung 1 aufgezeigt.



Quelle: Übernommen von Shahin, Ali Babar und Zhu (2017)

Abbildung 1: Zusammenhang zwischen CI, CDE und CD

2.2 Begrifflichkeiten und Prinzipien von Continuous Integration

Um die Bedeutung und den Nutzen von Continuous Integration und des Continuous-Software-Engineering für die Softwareentwicklung nachvollziehen zu können, ist es hilfreich, einen Blick auf die historische Entwicklung und die grundlegenden Prinzipien der CI zu werfen. Ein Kernprinzip hinter Continuous Integration wurde bereits im Jahr 1991 von Grady Booch definiert. Hierbei werden Software-Releases nicht als ein großes Ereignis betrachtet, sondern regelmäßig durchgeführt, wobei die vollständige Software stetig größer wird.[9] Kent Beck popularisierte im

[7] Vgl. Humble und Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 2010.
[8] Vgl. Shahin, Ali Babar und Zhu. „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“. 2017, S. 3911.
[9] Vgl. Booch. *Object oriented design with applications*. 1991.

Jahr 1998 die Disziplin des „Extreme Programming“, wobei großer Wert auf das frühe und regelmäßige Testen und Integrieren der entwickelten Komponenten einer Software gelegt wird. Beck behauptet hierbei, dass ein Feature, für welches es keine automatisierten Tests gibt, auch nicht funktioniert.[10] Im Jahr 2006 fasste Software-Entwickler Martin Fowler einige Bereiche dieser Methodiken in dem Artikel „Continuous Integration“ unter dem gleichnamigen Begriff zusammen. Fowler beschreibt CI als einen Prozess, bei dem Teammitglieder ihre Arbeit regelmäßig Integrieren, wobei Integration als der Build-Prozess inklusive automatisierter Tests für die vollständige Software mitsamt der erarbeiteten Änderungen zu verstehen ist.[11] Ein grundlegendes Ziel dieser Vorgehensweise ist neben der frühzeitigen Erkennung von Fehlern im Quellcode durch automatisierte Tests und QA-Prüfungen, die Reduzierung der „Cycle Time“, welche die Zeitspanne von der Entwicklung eines Features bis zum Erhalten des Kundenfeedbacks nach dessen Auslieferung beschreibt.[12] In 2010 betonen Humble und Farley die Wichtigkeit von CI und behaupten, dass Software ohne Continuous Integration als defekt gilt, bis sie als funktionierend nachgewiesen wird, während mit CI Software mit jeder erfolgreich integrierten Änderung als funktionierend bewiesen wird.[13] Über Zeit hat sich CI als essenzielle Praktik in der Software-Welt etabliert. Im Jahr 2016 zeigte eine Analyse von 34.544 auf der Plattform GitHub verwalteten Open-Source-Projekten, dass 40% der Projekte CI einsetzen. Bei den populärsten 500 analysierten Projekten lag dieser Anteil bei 70%.[14] Da die erfolgreiche Implementierung von Continuous Integration auf der Kombination von verschiedenen Methoden zur Verwaltung von Softwareprojekten fußt, werden im Folgenden einige dieser Schlüsselaspekte aufgezeigt:

- **Regelmäßige Integration**

  Die namensgebende Methodik der CI ist das regelmäßige Integrieren von Software und damit verbunden ist der Software-Release. In der traditionellen Softwareentwicklung wird der Release als ein einmaliges großes Ereignis betrachtet. Als Integration wird der Prozess des Einbindens einer einzeln entwickelten Komponente in die bisher bestehende Gesamtheit einer Software bezeichnet, wobei der Release das Zusammenfinden aller Komponenten und das Ausführen des Build-Prozesses bis hin zur fertigen, ausführbaren und auslieferbaren Software beschreibt. In der Vergangenheit wurde bei einem Release jede Einzelkomponente der Software manuell integriert und getestet, wobei dies als eigene Phase in der Entwicklung einer Applikation galt. In einem CI-gestützten Projekt wird der Prozess des Software-Releases vollständig automatisiert, sodass jedes Teammitglied eine entwickelte Komponente schnell integrieren und einen Software-Build erzeugen kann, was sich positiv auf die Cycle Time auswirkt. In einem Build-Prozess erzeugte Dateien, welche die fertig gebaute Software ausmachen, werden dabei als „Artifact“ betitelt.[15] Fowler empfiehlt hierbei, dass ein entwickeltes Feature erst nach der vollständigen Integration durch einen erfolgreichen Build als fertig angesehen werden soll.[16] In der Regel finden das Bauen und

[10] Vgl. Beck. „Extreme programming: A humanistic discipline of software development“. 1998, S. 2–4.
[11] Vgl. Fowler. *Continuous Integration*. 2006.
[12] Vgl. Elazhary, Werner, Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices“. 2022, S. 2580.
[13] Vgl. Humble und Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 2010, S. 56.
[14] Vgl. Hilton, Tunnell, Huang et al. „Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects“. 2016, S. 428–429.
[15] Vgl. Elazhary, Werner, Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices“. 2022, S. 2580.
[16] Vgl. Fowler. *Continuous Integration*. 2006.

Testen der Software in einer „Pipeline“ statt, welche die genutzten CI-Tools wie Test-Suites und weitere Analyse-Software ausführt.[17]

- **Automatisierte Tests**

  Neben der regelmäßigen Integration von Code gelten automatisierte Software-Tests und Quality-Checks als ein wichtiger Aspekt von CI. Hierbei werden oftmals Unit-Tests verwendet, welche eine einzelne Softwarekomponente auf ihre Funktion prüfen, abgekapselt von anderen Komponenten.[18] Wenn ein Test fehlschlägt, wird in der Regel die Pipeline unterbrochen. Neben den typischerweise schnell durchlaufenden Unit-Tests, die aufgrund ihrer isolierten Natur spezifische Komponenten prüfen, können nach dem Build-Prozess umfassende End-To-End-Tests für die gesamte Software durchgeführt werden. Diese systemübergreifenden Tests, welche das Zusammenspiel verschiedener Komponenten testen, erhöhen zwar die Dauer des Test-Prozesses, können aber wichtige Einsicht in die Qualität der Software gewähren.[19] Automatisierten Tests in einer CI-Pipeline können insgesamt dabei helfen, Fehler in eingeführtem Quellcode zu finden.[20] Oftmals kommen auch Static-Code-Analysis Tools zum Einsatz, welche den Code eines Projekts anhand gegebener Regeln prüfen können, um gegebenenfalls die Pipeline abzubrechen. Diese Programme können zusätzliche Qualitätsprüfungen und Coding-Standards in ein Projekt einführen.[21]

- **Reproduzierbarkeit**

  Ein weiterer zentraler Aspekt von Continuous Integration ist die Reproduzierbarkeit der CI-Pipeline. In einem CI-Umfeld werden der Build- und Testing-Prozess vollständig automatisiert und standardisiert, was bedeutet, dass diese unter den gleichen Bedingungen und mit den gleichen Ergebnissen wiederholt werden können. Dies ist von entscheidender Bedeutung, um sicherzustellen, dass die Software in verschiedenen Umgebungen (z. B. Entwicklung, Test, Produktion) konsistent funktioniert. Durch das Zentralisieren der Software und der CI-Tools in einer Versionierungssoftware in Verbindung mit dem Automatisieren des Build- und Testing-Prozesses, wird das Wiederherstellen von früheren Ständen in der Entwicklung und das Nachvollziehen von Fehlern im Entwicklungsprozess vereinfacht.[22]

- **Transparenz**

  Transparenz in einem CI-Prozess bedeutet, dass alle Aspekte des Entwicklungsprozesses sichtbar und verständlich sind, sowohl innerhalb des Entwicklungsteams als auch für andere Stakeholder. Sie ermöglicht eine klare Sicht auf den aktuellen Stand des Projekts, einschließlich der Qualität des Codes, der Fortschritte und möglicher Probleme. Durch den Einsatz von CI und der Automatisierung des Build- und Test-Prozesses können Teammitglieder schnell Feedback über den Status von Änderungen an der Codebase erhalten.

[17] Vgl. Elazhary, Werner, Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices“. 2022, S. 2571.
[18] Vgl. Ahmad, Haleem und Beg. „Test Driven Development with Continuous Integration: A Literature Review“. 2013, S. 280.
[19] Vgl. Fowler. *Continuous Integration*. 2006.
[20] Vgl. Elazhary, Werner, Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices“. 2022, S. 2572.
[21] Vgl. Zampetti, Scalabrino, Oliveto et al. „How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines“. 2017, S. 338.
[22] Vgl. Duvall, Matyas und Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007, S. 74–75.

> Wenn ein Problem auftritt, z. B. ein Test fehlschlägt oder ein Build abbricht, wird das Team sofort benachrichtigt, sodass das Problem schnell behoben werden kann. Dies reduziert die Zeit und den Aufwand, die benötigt werden, um Fehler zu finden und zu beheben, und verbessert die Qualität der Software. Darüber hinaus fördert das schnelle Feedback die Kommunikation und Zusammenarbeit im Team, da alle Mitglieder ständig über den Status des Projekts informiert sind.[23,24]

Die vorgestellten CI-Praktiken bauen auf unterschiedlichen Technologien und Konzepten auf, die für das Betreiben von Continuous Integration eine Rolle spielen. Um die Praktiken der Continuous Integration vollständig zu verstehen und effektiv umzusetzen, ist es unerlässlich, die zugrunde liegenden Technologien und Konzepte zu betrachten, die die Basis für die Implementierung von CI in einem Softwareprojekt bilden. Dazu gehören Themen wie Version-Control-Systems, Pipelines, Containerization, Software-Testing und Static-Code-Analysis. Im Folgenden werden einige dieser Themen untersucht, um ein umfassendes Verständnis der Mechanismen und Werkzeuge zu vermitteln, die für die erfolgreiche Anwendung von Continuous Integration erforderlich sind.

Version-Control-Systems

Das Verwalten verschiedener Versionsstände von Software wird durch sogenannte Version-Control-Systems (VCS) ermöglicht. Es gibt verschiedene Implementierungen von VCS, namentlich unter anderem Git, Mercurial oder Subversion. Ein VCS speichert den gesamten Quellcode eines Projekts in einem eigenen „Repository“.[25] Innerhalb desselben Repositories können verschiedene Versions-Historien in sogenannten „Branches“ gespeichert werden. Der aktuelle Stand eines Branches kann dabei kopiert werden und daran vorgenommene Revisionen können anschließend wieder in einen Branch des Repositories integriert werden. Versionskontroll-Software ermöglicht es außerdem, verschiedene Versionen in Form von Branches zusammenzuführen. Der Prozess des Zusammenführens verschiedener Branches wird hierbei als „Merge“ bezeichnet. In vielen VCS können Merges zunächst durch sogenannten Merge-Requests angefragt werden, bevor diese final durchgeführt werden. Die Strukturierung von Branches kann anhand gewisser Vorgaben in Form einer Branching-Strategie definiert werden, wobei Branches für gewisse Aufgaben innerhalb des Projekts einer Namenskonvention folgen und Regeln für Merges vorgeschrieben werden. Die Nutzung eines VCS, in welchem verschiedene Branches angelegt werden können, ermöglicht Teammitgliedern die gleichzeitige Arbeit an einer einzelnen Codebase, ohne sich dabei gegenseitig zu beeinflussen.[26] Fowler setzt für die Nutzung von CI das Führen eines Source-Repositories voraus, welches die Projekt-Software inklusive aller dazugehörigen Build- und Testing-Konfigurationen, verwaltet.[27]

[23] Vgl. Elazhary, Werner, Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices“. 2022, S. 2573.
[24] Vgl. Elazhary, Storey, Ernst et al. „ADEPT: A Socio-Technical Theory of Continuous Integration“. 2021, S. 26–27.
[25] Vgl. Duvall, Matyas und Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007, S. 381–385.
[26] Vgl. Duvall, Matyas und Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007, S. 388–390.
[27] Vgl. Fowler. *Continuous Integration*. 2006.

Pipelines

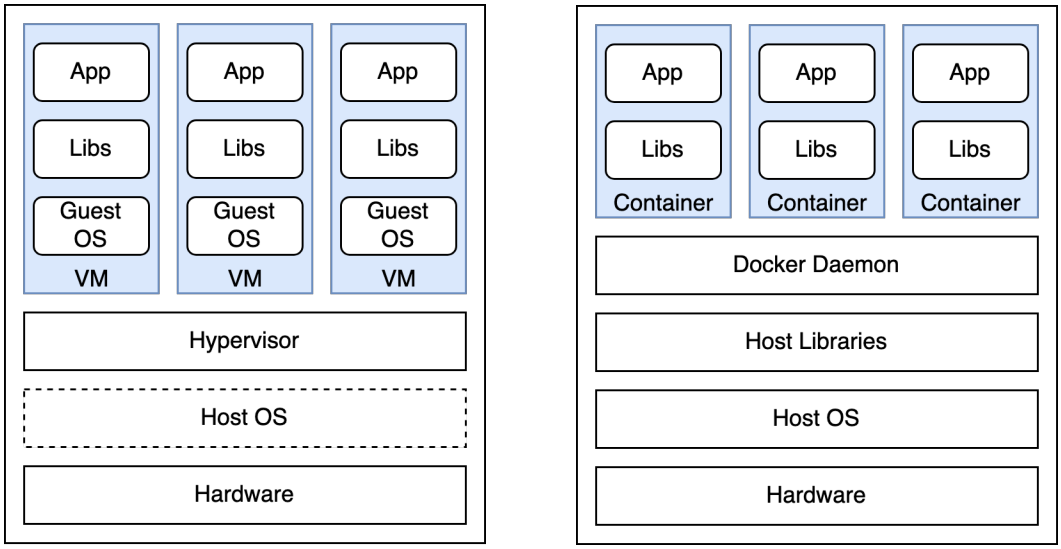
Um die in einer Versionskontroll-Software eingeführten Code-Änderungen automatisiert bauen, testen und ausliefern zu können, wird eine ausführende Umgebung für diese Prozesse benötigt. In einem CI-Prozess werden hierfür Pipelines eingesetzt, welche zum Ausführen von Befehlen und Prozessen zur automatischen Integration und dem Testen von Code genutzt werden.[28] Sie werden durch sogenannte Pipeline- oder Task-Runner verwaltet und können verschiedene Aufgaben verrichten, welche durch das Ausführen von Befehlen innerhalb sogenannter „Jobs“ durchgeführt werden. Pipelines können mehrere Jobs starten, teilweise auch parallel, wobei nacheinander laufende Jobs voneinander abhängig sein und Artifacts für nachfolgende Jobs zwischengespeichert werden können. Pipelines spielen durch das kontinuierliche Integrieren von entwickeltem Code eine integrale Rolle bei der Reduzierung der Cycle Time[29] und sind somit wichtig zum Erreichen des Ziels Z_1 der Arbeit. Eine Pipeline in einem CI-Prozess wird in der Regel bei jeder Änderung am Quellcode des Source-Repositories gestartet.[30] Die Durchlaufzeit einer CI-Pipeline sollte dabei möglichst gering bleiben, da viele Entwickler auf das erfolgreiche Durchlaufen der Pipeline warten, bevor sie mit der nächsten Aufgabe beginnen. Eine grobe Richtlinie für die maximale Dauer einer Pipeline ist die Zehn-Minuten-Marke.[31]

Containerization

Um die zum Bauen, Testen und Ausliefern von Software in einem CI-Projekt benötigten Pipelines nutzen zu können, wird eine isolierte und reproduzierbare ausführende Umgebung benötigt. In der Vergangenheit wurden für solche isolierten Umgebungen hauptsächlich „Virtual Machines“ (VMs) verwendet. VMs nutzen hierbei einen sogenannten „Hypervisor“, welcher die Hardware eines physischen Computers emulieren kann, sodass mehrere Betriebssysteminstanzen und die darin verwalteten Applikationen gleichzeitig auf einem einzigen System ausführbar sind.[32] Containerization ist eine modernere Technologie, die einen Performance-Zuwachs gegenüber VMs bieten kann.[33] Die hierbei genutzten „Container“ sind eigenständige, ausführbare Pakete, die alles enthalten, was für den Betrieb einer Anwendung erforderlich ist, einschließlich Code, Laufzeitumgebung, Bibliotheken und Systemtools. Während VMs zur Ausführung einen Hypervisor und vollständige Betriebssysteminstanzen benötigen, teilen sich Container die Ressourcen eines einzelnen Betriebssystems und isolieren so die Anwendung von der zugrunde liegenden Infrastruktur. Ein Container ist hierbei die laufende Instanz eines Container-Images, welches den Aufbau des Containers inklusive Abhängigkeiten und installierter Software speichert. Containerization erleichtert also das Vereinheitlichen verschiedener Umgebungen in CI-Projekten, indem es Abhängigkeiten und Konfigurationen in einem Container kapselt.[34] Dies bietet Entwicklern und Testern eine erhöhte Effizienz und Flexibilität, da sie sich darauf verlassen können, dass die

[28] Vgl. Elazhary, Werner, Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices“. 2022, S. 2571.
[29] Vgl. Fowler. *Continuous Integration*. 2006.
[30] Vgl. Duvall, Matyas und Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007, S. 3–4.
[31] Vgl. Duvall, Matyas und Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007, S. 87–88.
[32] Vgl. Combe, Martin und Di Pietro. „To Docker or Not to Docker: A Security Perspective“. 2016, S. 54–55.
[33] Vgl. Spoiala, Calinciuc, Turcu et al. „Performance comparison of a WebRTC server on Docker versus virtual machine“. 2016.
[34] Vgl. Elazhary, Werner, Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices“. 2022, S. 2576.

Software in jeder Umgebung identisch funktioniert.



Quelle: Eigene Darstellung nach Combe, Martin und Di Pietro (2016)

Abbildung 2: Visualisierung der Hypervisor- und Container-Architektur

In Abbildung 2 werden die Architekturen von Hypervisor und Container gegenübergestellt. Die Visualisierung zeigt im linken Teil den Aufbau eines Systems, in dem ein Hypervisor zum emulieren von Hardware-Ressourcen verwendet wird (a). Die Technologie kann hierbei auf einem bereits installierten Host-Betriebssystem (Host OS) oder ganz ohne OS betrieben werden. Einzelne VMs, die durch den Hypervisor betrieben werden können, nutzen hierbei jeweils ein eigenes Gast-Betriebssystem (Guest OS). Im Kontrast dazu steht die Container-Architektur, welche das Betreiben eines Host OS voraussetzt und dessen Ressourcen zur Ausführung isolierter Container-Instanzen verwendet (b). Das Ausführen der Container wird dabei durch eine Container-Engine übernommen, im Falle der Docker-Technologie wird dies durch einen eigenen Hintergrundprozess (Docker Daemon) ermöglicht.

Software-Testing

Wenn eine ausführende Umgebung für das Bauen und Testen von Software besteht, können Software-Tests in den CI-Prozess eingeführt werden. Durch manuelle und automatisierte Tests kann ein Software-Produkt auf verschiedene Funktionsbereiche überprüft werden. Da manuelle Tests einen hohen Zeitaufwand darstellen können, wird Testautomatisierung in der agilen Softwareentwicklung als wichtige Aktivität angesehen. Besonders bei repetitiven Aufgaben zum Reproduzieren vordefinierter Systemfunktionalitäten wirken automatisierte Tests effizienzsteigernd.[35] Sie müssen hierbei nur einmal angelegt werden und können anschließend beliebig oft und ohne weitere Aufwände erneut ausgeführt werden. In CI-Projekten werden automatisierte Tests oftmals eingesetzt, um Probleme bei der Einführung von Features mit vorher entwickelter

[35] Vgl. Collins, Dias-Neto und Lucena Jr. „Strategies for Agile Software Testing Automation: An Industrial Experience“. 2012, S. 440.

Logik zu vermeiden und um Fehler im Entwicklungsprozess früher entdecken zu können. Häufiges automatisiertes Testing in der CI kann sich positiv auf die Qualität der entwickelten Software auswirken.[36, 37]
Nachfolgend werden verschiedene Arten von Tests in der Softwareentwicklung vorgestellt:[38]

- **Unit-Tests**

  Unit-Tests prüfen das Verhalten von einzelnen Programm-Einheiten. Eine Programmeinheit oder Prozedur stellt eine oder mehrere zusammenhängende Anweisungen in Form von Code dar, welche durch andere Teile der Software namentlich aufgerufen werden können. Durch Unit-Tests werden also die Implementierungsschritte eines Software-Programms einzeln geprüft.

- **Module-Tests**

  Ein Modul besteht aus einer Sammlung zusammengehöriger Programmeinheiten, die in einer Datei, einem Paket oder einer Klasse zusammengefasst sind. Module-Tests, auch Component-Tests genannt, zielen darauf ab, diese Module isoliert zu bewerten, im Hinblick auf die Interaktionen zwischen den Einheiten und ihren dazugehörigen Datenstrukturen. In der Praxis können Unit- und Module-Tests zusammengefasst als Prüfung der eigentlichen Implementierung eines Programms betrachtet werden.[39]

- **Integration-Tests**

  Integration-Tests fokussieren sich auf die Integration verschiedener Module einer Software. Sie prüfen das Zusammenspiel von unterschiedlichen Komponenten des Programms und stellen so sicher, dass die Kommunikation zwischen verschiedenen Bereichen der Applikation korrekt verläuft. Integrationstests stützen sich dabei auf die Annahme, dass die Module selbst bereits vollständig funktionieren.

- **System-Tests**

  Ähnlich wie Integration-Tests prüfen System-Tests mehrere Komponenten eines einzelnen Systems auf ihre Zusammenarbeit. System-Tests grenzen sich hierbei allerdings durch ihren Bezug auf vorher definierte Produktanforderungen ab. Sie prüfen, ob ein Programm in Gänze funktioniert, wobei der Erfolg der Tests aus der Erfüllung von vordefinierten, geschäftsseitigen Anforderungen an das Programm besteht. Diese Art von Test wird auch als „Functional Test“ bezeichnet.[40]

- **Acceptance-Tests**

  Acceptance-Tests werden manuell ausgeführt. Sie prüfen, ob die Bedürfnisse des Projektkunden durch die fertiggestellte Software abgedeckt werden. Hierbei werden also Tests aus

[36] Vgl. Ahmad, Haleem und Beg. „Test Driven Development with Continuous Integration: A Literature Review“. 2013, S. 281.
[37] Vgl. Elazhary, Werner, Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices“. 2022, S. 2570.
[38] Vgl. Ammann und Offutt. *Introduction to Software Testing*. 2016, S. 23–24.
[39] Vgl. Collins, Dias-Neto und Lucena Jr. „Strategies for Agile Software Testing Automation: An Industrial Experience“. 2012, S. 441.
[40] Vgl. Collins, Dias-Neto und Lucena Jr. „Strategies for Agile Software Testing Automation: An Industrial Experience“. 2012, S. 441.

der Sicht der Kunden oder vom Kunden selbst durchgeführt, welche fehlschlagen, wenn dessen Anforderungen an das Programm nicht erfüllt werden.

Durch diese verschiedenen Test-Arten kann ein großer Bereich der zu testenden Software und dessen Codebase abgedeckt werden. Da bereits erstellte Tests immer wieder verwendet werden können, werden diese oftmals eingesetzt, um zu prüfen, ob neu eingeführter Code diese bestehenden Tests fehlschlagen lässt. Dieser Einsatz von existierenden Tests zur Sicherstellung der Funktionalität des Systems wird Regression-Testing genannt und stellt einen wichtigen Aspekt des automatisierten Prüfens von Software in einem CI-Kontext dar.[41] Um beim Ausführen von Tests etwaige Abhängigkeiten, die zur Ausführung des Codes innerhalb Test benötigt werden, zu umgehen, werden „Mocks“ eingesetzt. Ein Mock stellt eine Nachahmung der Vorgehensweise einer Abhängigkeit dar, so kann zum Beispiel das Verhalten einer Datenbank oder eines anderen System-Moduls simuliert werden, ohne diese als Service im Testing-Prozess zu benötigen.[42] Zur Bewertung der Effektivität und Qualität von erstellten Software-Tests gibt es verschiedene Ansätze, ein wichtiger Indikator ist jedoch die Test-Abdeckung. Der Anteil von durch Tests abgedeckten Klassen und Funktionen im Kontrast zu der Anzahl an ungetesteten Komponenten der Software wird als Abdeckung oder „Coverage“ bezeichnet.[43] Neben Test-Coverage können noch weitere Metriken wie Mutation-Tests zur Bestimmung der Qualität von Tests genutzt werden. Hierbei wird der zu testende Source-Code verändert oder mutiert und dieser abgeänderte Code mit einem bestehenden Test-Set geprüft. Wenn die mutierte Variante des Codes durch das Test-Set aufgedeckt wurde, indem der Test fehlschlägt, wird der mutierte Code, auch Mutant genannt, als eliminiert betrachtet. Durch Mutation-Testing können Schwachstellen in Software-Tests gefunden werden und die Qualität der bestehenden Tests eines Projekts gemessen werden.[44]

Static-Code-Analysis

Neben automatisiertem Testing ist das statische Analysieren des Codes ein weiterer wichtiger Aspekt für die Qualitätskontrolle von Software in einer CI-Umgebung. Wo Tests nur die Ergebnisse des Ausführens von Komponenten und Funktionen messen, können statische Code-Analysis-Tools die Struktur des Codes und die Einhaltung von vorgegebenen Standards bewerten. Eine laufende CI-Pipeline kann so bei der Einführung von Code, welcher nicht dem vorgegebenen Coding-Standard entspricht, abgebrochen werden. Dies zwingt die Entwickler eines Projekts dazu, eine einheitliche Code-Struktur zu verwenden und kann das Einführen von Fehlern verringern. Außerdem können Static-Code-Analysis-Tools durch Warnungen verhindern, dass unoptimierter Code in die Produktionsumgebung gerät.[45] Statische Code-Analyse basiert oftmals auf der Erstellung von einem sogenannten „Abstract Syntax Tree“ (AST) welcher eine hierarchische Struktur des Quellcodes darstellt. Der AST zerlegt den Code in seine grundlegenden Bestandteile und stellt Beziehungen zwischen diesen Teilen her.[46] Dies ermöglicht es

[41] Vgl. Duvall, Matyas und Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007, S. 53–54.
[42] Vgl. Ammann und Offutt. *Introduction to Software Testing*. 2016, S. 299–300.
[43] Vgl. Duvall, Matyas und Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007, S. 132–133.
[44] Vgl. Ammann und Offutt. *Introduction to Software Testing*. 2016, S. 242.
[45] Vgl. Zampetti, Scalabrino, Oliveto et al. „How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines“. 2017.
[46] Vgl. Baxter, Yahin, Moura et al. „Clone detection using abstract syntax trees“. 1998.

Analysewerkzeugen, den Code systematisch zu durchsuchen, Muster zu erkennen und potenzielle Fehler, Schwachstellen, Dopplungen oder Abweichungen von gegebenen Code-Standards zu identifizieren, ohne diesen auszuführen.

Monitoring

Monitoring, also das Überwachen von Software und dessen Metriken, ist ein unerlässlicher Bestandteil moderner Softwareentwicklungs- und Betriebsprozesse. Die Ergebnisse von Tests und statischen Code-Analysen eines CI-Prozesses können zur kontinuierlichen Überwachung von Qualitätsmetriken und Code-Standards genutzt werden.[47] Neben der Überwachung von Test- und QA-Ergebnissen können auch weitere Software-Bereiche überwacht werden, zum Beispiel durch das Ausführen von Performance- und Lasttests innerhalb der CI-Pipeline.[48] Hierbei wird die Geschwindigkeit des Durchführens von Transaktionen oder Aufrufzeiten beim Ausführen der Software mit vielen gleichzeitigen Nutzerzugriffen überwacht.

2.3 Übersicht über die Shopware-Platform

Shopware wurde als Online-Shop-Software im Jahr 2000 durch Stefan Hamann ins Leben gerufen[49] und hat sich über Zeit zu einer umfangreichen Lösung für den digitalen Handel entwickelt. Neben E-Commerce-Systemen wie Spryker, Commercetools oder Shopify hat sich Shopware durch seine Flexibilität und Anpassbarkeit in der Branche einen Namen gemacht. Die Software bietet heute in ihrer aktuellen Major-Version 6 eine moderne E-Commerce-Plattform auf Basis des PHP-Frameworks „Symfony“. Das Symfony-Framework wird neben Shopware noch von anderen PHP-basierten Projekten wie dem Content-Management-System (CMS) „Drupal“, dem Shop-System „Magento“ und einigen weiteren Programmen[50] als Grundlage genutzt und bildet somit ein erprobtes Fundament für die Shopware-Platform. Shopware selbst ist nach der Installation bereits voll funktionsfähig und kann mit einem Backend und optional mit einem Frontend oder für das Konsumieren der mitgelieferten API eingerichtet werden. Ein Application Programming Interface (API) ist hierbei eine Schnittstelle, welche die standardisierte Kommunikation zwischen Programmen ermöglicht. Die Software kann auf verschiedenen Plattformen gehostet werden, darunter Linux-Server und containerisierte Umgebungen. Darüber hinaus bietet Shopware als Unternehmen auch eine eigene Hosting-Lösung an, die speziell auf die Anforderungen der Software zugeschnitten ist. Die Plattform kann sowohl im Einzelbetrieb als auch als Cluster genutzt werden, wobei mehrere, oftmals geografisch verteilte Instanzen der Software gleichzeitig betrieben werden, um eine hohe Verfügbarkeit und Skalierbarkeit zu gewährleisten.[51] Nachfolgend wird der Aufbau des Symfony-Frameworks dargestellt und die Architektur der darauf basierenden Shopware-Platform aufgezeigt.

[47] Vgl. Duvall, Matyas und Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 2007, S. 17–18.
[48] Vgl. Collins, Dias-Neto und Lucena Jr. „Strategies for Agile Software Testing Automation: An Industrial Experience“. 2012, S. 441.
[49] Vgl. Shopware. *The story behind Shopware AG*. 2023.
[50] Vgl. Symfony SAS. *Projects using Symfony - Popular PHP projects using Symfony components or based on the Symfony framework*. 2023.
[51] Vgl. Shopware AG. *Cluster Setup*. 2023.

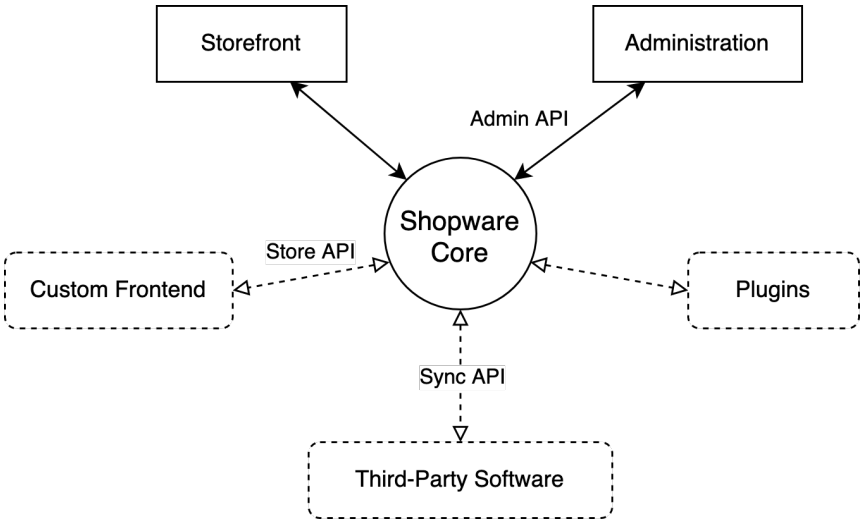
Symfony-Framework

Symfony ist ein im Jahr 2005 von Fabien Potencier entwickeltes Full-Stack-Framework. Das PHP-basierte Projekt besteht aus verschiedenen Einzelkomponenten, welche unabhängig voneinander verwendet werden können, somit ist es sehr flexibel. Gleichzeitig bietet Symfony eine Reihe von Konventionen und Best Practises für das Erstellen und Nutzen von Komponenten, welche die Entwicklung von Anwendungen erleichtern und beschleunigen. Das Framework ist weit verbreitet und stellt eine robuste Grundlage für die Entwicklung umfangreicher Softwareprojekte dar. Es beinhaltet essenzielle Funktionen wie Abhängigkeiten-Verwaltung, Performance-Reviews, Datenverwaltung, Handling von Nutzer-Sessions, Routing, Formularverwaltung und eine Template-Engine. Durch den Einsatz des Paket-Managers „Composer“ können diese Grundfunktionen erweitert und zusätzliche Features in ein Projekt importiert werden. Symfony bietet eine Reihe von Konsolenbefehlen, die das Erstellen von eigenen Komponenten, die Durchführung von Datenbankmigrationen, das Ausführen von Tests und vieles mehr erleichtern. Die umfangreiche Dokumentation und die aktive Entwicklercommunity des Frameworks bieten einen hervorragenden Support für Entwickler. Zudem gewährleistet Symfony eine Langzeitunterstützung für jede Hauptversion mit einem Support-Zeitraum von drei Jahren, was die Zuverlässigkeit und Stabilität des Frameworks unterstreicht.[52]

Architektur der Shopware-Platform

Shopware 6 bietet eine modulare Architektur. Die Plattform besteht aus drei Kernkomponenten, dem Shopware-Core, der Administrationsoberfläche und der Storefront. Der Core bildet hierbei die grundlegenden Shop-Funktionen und Ressourcen und stellt für diese verschiedene Schnittstellen zur Nutzung bereit. Die Verwaltung der Shop-Daten, Produkte und weiterer Betriebsfunktionen wird mit der Administrations-Oberfläche vorgenommen. Diese bildet eine eigene Komponente und kommuniziert mit dem Shopware-Core über die Admin-API. Die Administration bietet neben der Konfiguration des Shops und dessen Daten und Produkten auch eine CMS-Funktion und kann beliebigen Content verwalten. Letztlich wird das Frontend durch die Shopware-Storefront dargestellt, welche mithilfe von Symfonys Template-Engine „Twig“ und der direkten Anbindung an den Shopware-Core das Aufrufen von Produkten, Authentifizieren von Usern und Durchführen von Zahlungen ermöglicht. Die Storefront ist eine Komponente, welche optional auch deaktiviert und auf Wunsch durch die Nutzung der Store-API mit einer eigenen Applikation ersetzt werden kann. Im Standardumfang liefert die Shopware-Platform alle drei Hauptkomponenten aus. Diese stehen als einzelnes Repository mit dem Namen `shopware/platform` auf GitHub zur Verfügung.[53] Neben den drei Hauptkomponenten bietet die Shopware-Platform verschiedene Anbindungsmöglichkeiten zur Anpassung des Shop-Systems. Plugins können direkt an den Core angebunden werden und ermöglichen die Erweiterung der Shopware-Logik, das Templating der Storefront oder Anpassungen an der Administration. Zusätzlich kann die Sync-API für die Anbindung externer Anwendungen verwendet werden, wie zum Beispiel Zahlungsanbieter oder E-Mail-Dienste.[54]

[52] Vgl. Engebreth und Sahu. „Introduction to Symfony“. 2023, S. 273–278.
[53] Vgl. Shopware. *Einblicke in die Core Architektur von Shopware 6*. 2019.
[54] Vgl. Pickware GmbH. *Shopware 6 – Ein Blick auf die neue Architektur*. 2019.



Quelle: Eigene Darstellung nach Pickware GmbH (2019)

Abbildung 3: Visalisierung der Shopware-Architektur

Eine grobe Zusammenfassung der Architektur kann aus Abbildung 3 entnommen werden. Hierbei werden die Kernkomponenten von Shopware aufgezeigt und optionale Services in gestrichelt dargestellt. Für die Entwicklung der CI-Strategie wird der Fokus auf Shopware-Plugins gesetzt. Durch Plugins können der Shopware-Core, die Administrationsoberfläche und die Storefront angepasst werden, wobei diese als abgekapselte Module entwickelt oder als Monolith zusammen mit dem Shopware-Projekt in einem gemeinsamen Repository verwaltet werden können. Shopwares Plugin-System baut auf dem Bundle-System von Symfony auf, um standardisierte, modulare Erweiterungen der Software zu ermöglichen, wobei das Bundle-System Funktionen wie Plugin-Lifecycle-Verwaltung bereitstellt. Symfony nutzt teils für eigene Core-Features das Bundle-System, um die einzelnen Teilbereiche des Frameworks modular zu gestalten.[55] Durch die solide Basis der Symfony-Bundles bieten Shopware-Plugins eine klare Struktur und eine hohe Erweiterbarkeit und fördern die Wiederverwendbarkeit von entwickelter Software für das Shop-System.

[55] Vgl. Shopware. *Plugins for Symfony developers*. 2023.

3 Analyse und Konzept

Im folgenden Kapitel werden verschiedene Techniken zur kontinuierlichen Integration von Code analysiert und eine Konzeption für Shopware-basierte Projekte erarbeitet. Zunächst werden technische Anforderungen an die zu entwickelnde Strategie im Hinblick auf die geschäftsseitigen Vorgaben durch die Ziele Z_n der Arbeit definiert. Anschließend wird die Ausgangssituation in Shopware-Projekten und dessen Umgebungen und kundenspezifischer Anpassungen beleuchtet, wobei der Fokus auf den gemeinsamen Grundvoraussetzungen und der Anpassungsfähigkeit der CI-Strategie an verschiedene Architekturen liegt. Zuletzt wird die eigentliche Konzeption der Strategie anhand der zuvor erarbeiteten Methoden und Praktiken und den definierten Anforderungen vorgenommen.

3.1 Technische Anforderungen

Um ein vollumfängliches Konzept erstellen zu können, müssen einige technische Anforderungen an die Strategie definiert werden. Diese Anforderungen orientieren sich sowohl an der Zieltechnologie – in diesem Fall Shopware – als auch an den im fachlichen Hintergrund erörterten Methoden und Praktiken. Da die Strategie das Nutzen von CI in Shopware-Projekten ermöglichen soll, werden nachfolgend die technischen Anforderungen an das Konzept im Hinblick auf die Gegebenheiten der Shop-Software aufgezeigt:

- **Vollautomatisierter Software-Build**

  Der Prozess des Software-Builds muss durchweg automatisiert stattfinden. Dies umfasst das Installieren der Software aus der Versionskontrolle, dem Herunterladen und Installieren sämtlicher Abhängigkeiten und dem Erzeugen von Build-Dateien aus dem gegebenen Quellcode.

- **Automatisierte Software-Tests und Quality-Checks**

  Da Testing einen integralen Bestandteil der CI bildet, muss die zu konzipierende Strategie ein umfassendes Test-Konzept bieten. Dies schließt sowohl Unit- und Module-Tests als auch programmübergreifende Prüfungen wie Integration- und System-Tests ein. Zudem soll die Qualitätssicherung der zu entwickelnden Software eines Projekts durch statische Code-Analyse-Tools in die CI-Strategie integriert werden.

- **Reproduzierbare CI-Umgebungen**

  Es ist von entscheidender Bedeutung, dass die CI-Umgebung konsistent und reproduzierbar ist. Dies gewährleistet, dass jeder Build und Test unter identischen Bedingungen durchgeführt wird, unabhängig davon, wann und wo der CI-Prozess initiiert wird. Die Strategie sollte daher die Verwendung von Containerisierungstechnologien, wie Docker, in Betracht ziehen, um eine einheitliche, isolierte und wiederholbare Umgebung für den Build- und Testprozess zu schaffen. Dies minimiert potenzielle Inkonsistenzen und Fehler, die durch unterschiedliche Umgebungsbedingungen entstehen könnten.

- **Hohe Anpassbarkeit und Skalierbarkeit**

  Die CI-Strategie sollte flexibel genug sein, um sich an veränderte Anforderungen und wachsende Projektgrößen anzupassen. Dies bedeutet, dass sowohl die Infrastruktur als auch die

Prozesse skalierbar gestaltet werden müssen, um mit der Evolution des Projekts Schritt zu halten. Die Möglichkeit, neue Tools und Technologien nahtlos zu integrieren, sollte ebenfalls gegeben sein, um die kontinuierliche Anpassung und Optimierung der CI-Pipeline zu gewährleisten.

- **Einheitliche Daten-Verwaltung**

  Um Konsistenz und Nachvollziehbarkeit zu gewährleisten, sollte alles von Code über Konfigurationen bis hin zu Datenbank-Skripten und Tests in einem Versionskontrollsystem verwaltet werden. Dies ermöglicht eine klare Historie von Änderungen und erleichtert das Rollback im Falle von Problemen. Ein zentralisiertes VCS stellt sicher, dass alle Teammitglieder stets mit der aktuellsten Version arbeiten und Änderungen nachvollziehbar sind.

- **Kontinuierliche Bereitstellung und Deployments**

  Neben dem Build- und Testing-Prozess ist es auch wichtig, dass die CI-Strategie Mechanismen für die kontinuierliche Bereitstellung und das Deployment der Software bietet. Dies ermöglicht es, Änderungen schnell in die Produktionsumgebung zu übertragen und sicherzustellen, dass die Software stets aktuell und einsatzbereit ist. Automatisierte Deployments sollten daher in den Pipelines mit eingeplant werden, um den Prozess der Softwareauslieferung zu optimieren und zu beschleunigen.

- **Hohe System-Verfügbarkeit**

  Die CI-Infrastruktur sollte so konzipiert sein, dass sie stets verfügbar ist und minimale Ausfallzeiten aufweist. Dies ist entscheidend, um den kontinuierlichen Entwicklungsfluss nicht zu unterbrechen und eine ständige Feedback-Schleife zu gewährleisten. Redundanzen und regelmäßige Backups sollten implementiert werden, um die Systemverfügbarkeit auch bei unerwarteten Problemen zu gewährleisten.

- **Transparenter CI-Prozess**

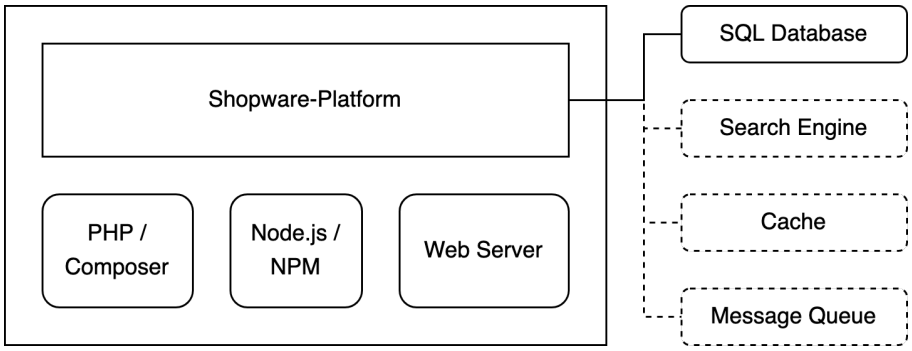
  Die CI-Umgebung sollte so gestaltet sein, dass alle Prozesse, Ergebnisse und Aktivitäten für alle Teammitglieder transparent und nachvollziehbar sind. Dies beinhaltet eine klare Darstellung des aktuellen Status der CI-Pipeline, einschließlich aller durchgeführten Tests, ihrer Ergebnisse und eventuell aufgetretener Fehler. Dadurch wird es dem Team ermöglicht, schnell auf Probleme zu reagieren, den Fortschritt zu überwachen und sicherzustellen, dass alle Beteiligten stets auf dem gleichen Stand sind.

Diese technischen Anforderungen sollen das Entwickeln einer vollumfänglichen CI-Strategie im Hinblick auf die definierten Ziele Z_n der Arbeit ermöglichen.

3.2 Analyse der Ausgangssituation

Die aktuelle Situation stellt sich wie folgt dar: In E-Commerce-Unternehmen, die Shopware-Projekte betreiben, gibt es in der Regel verschiedene Kunden, die mit unterschiedlichen Versionen von der Software arbeiten. Diese Kunden können bei verschiedenen Hosting-Anbietern untergebracht sein, was zu einer Vielfalt an technischen Umgebungen führt, in denen die Software betrieben wird. Darüber hinaus verwenden Kunden meist eine individuelle Kombination aus Plugins und Eigenentwicklungen, die auf dessen spezifische Bedürfnisse zugeschnitten sind. Diese

Diversität stellt eine Herausforderung dar, da sie eine Vielzahl von Variablen in die Entwicklung und Wartung der Software einbringt. Um eine generalisierte Strategie erstellen zu können, wird sich also zunächst auf die Gemeinsamkeiten von Shopware-Projekten konzentriert. Shopware besteht aus einer Vielzahl von Verzeichnissen und Dateien, jede Version der Software und jedes Shopware-basierte Projekt unterscheiden sich voneinander. Ungeachtet von Projekt und Version gibt es jedoch einige Grundvoraussetzungen für das Ausführen von Shopware.[56]



Quelle: Eigene Darstellung nach Shopware (2023)

Abbildung 4: Umgebung und Abhängigkeiten der Shopware-Platform

In Abbildung 4 werden die grundsätzlichen Abhängigkeiten der Shopware-Platform und die verschiedenen Services für den Betrieb der Software aufgezeigt. Hierbei werden optionale Services in gestrichelt dargestellt. Um ein Projekt installieren und ausführen zu können oder um Tests auf der Codebase durchzuführen, wird also eine Umgebung vorausgesetzt, in der die benötigten Services und Tools installiert sind. Nachfolgend werden die in der Grafik aufgezeigten Abhängigkeiten und Services erläutert:

- **PHP**

  Da Symfony und somit auch Shopware auf der Programmiersprache PHP basieren, muss diese in der ausführenden Umgebung installiert sein.

  - **PHP-Extensions**

    Shopware benötigt für den Betrieb einige PHP-Extensions, zum Beispiel zum Erstellen von Archiven oder dem Lesen von XML-Dateien.

  - **Paketmanager „Composer“**

    Durch Composer werden sowohl die Abhängigkeiten von Symfony und Shopware als auch Third-Party-Plugins und eigene Entwicklungen im Projekt verwaltet und zur Nutzung im PHP-Code bereitgestellt.

- **JavaScript**

  JavaScript wird bei der Nutzung der Shopware-Platform sowohl im Browser als auch serverseitig ausgeführt.

[56] Vgl. Shopware. *Requirements*. 2023.

  - **JavaScript-Runtime „Node“**

    Zur serverseitigen Ausführung von JavaScript-Code wird die Node-Runtime als Abhängigkeit vorausgesetzt.

  - **Node-Paketmanager „NPM“**

    Der Paketmanager NPM ist eine weitere Abhängigkeit von Shopware. Dieser verwaltet JavaScript-Pakete, welche zur Ausführung der Software benötigt werden.

- **Webserver**

  Um die Shop-Instanz im Browser aufrufen und API-Anfragen verarbeiten zu können, wird eine Webserver-Software benötigt.

- **Datenbank**

  Shopware schreibt zur Nutzung der Software eine Datenbank vor, welche die eigentlichen Shop-Konfigurationen, Produkte, Bestellungen, Kunden und weitere Daten verwaltet.

- **Search Engine**

  Um den Besuchern des Online-Shops eine Textsuche für Produkte und Hersteller zu bieten, unterstützt Shopware das optionale Anbinden verschiedener Suchmaschinen.

- **Cache**

  Zur Optimierung der Shop-Performance kann zudem optional ein Zwischenspeicher, auch „Cache“ genannt, für Warenkorb-Inhalte und aktuelle Nutzer-Sitzungen eingeführt werden.

- **Message Queue**

  Um viele gleichzeitige Zugriffe von Usern effizient verwalten zu können, erlaubt Shopware das Anbinden einer Message Queue. Diese ermöglichen das Sammeln von Nutzer-Anfragen, welche dann in Reihe verarbeitet werden können.

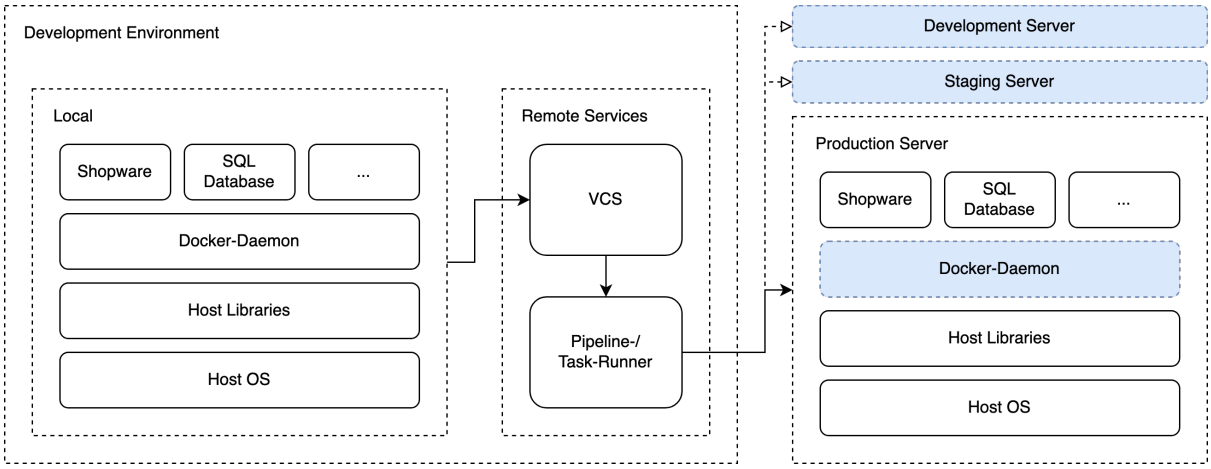
Durch das Installieren dieser grundlegenden Abhängigkeiten und Services wird das Betreiben von Shopware 6 ermöglicht. Da diese sich in zukünftigen Versionen der Software ändern können, sollte bei neuen Projekten immer geprüft werden, welche Abhängigkeiten für die gewünschte Shopware-Version vorgegeben sind. Alte Projekte sollten außerdem kontinuierlich auf aktuelle Versionen upgedatet und auf die Erfüllung von dessen Voraussetzungen geprüft werden.

3.3 Konzeption der CI-Strategie

Im Folgenden wird die Konzeption der CI-Strategie für Kundenprojekte auf Basis der Shopware-Platform durchgeführt. Hierbei wird zunächst die ausführende Umgebung geplant, sowohl für das lokale Entwickeln mit Shopware als auch für die Integration und Ausführung der verschiedenen CI-Tools innerhalb einer Pipeline. Anschließend wird die Struktur der angedachten Pipeline aufgestellt, wobei dessen einzelne Aufgaben systematisch in eigene Phasen und Jobs unterteilt werden.

Projekt-Struktur und Umgebung

Die Anforderungen des Projekts setzten eine Umgebung, in der die Shopware-Platform ausgeführt, getestet und analysiert werden kann, voraus. Eine Umgebung muss in diesem Fall sowohl lokal als auch in einer CI-Pipeline ausführbar sein und die Grundbedürfnisse der Shopware-Instanz bereitstellen. Dies beinhaltet die von Shopware genutzte PHP-Version inklusive benötigter PHP-Erweiterungen sowie Composer, Node, eine Webserver-Software und weitere Voraussetzungen. Da spätere Umgebungen, an welche die Software ausgeliefert werden soll, nicht zwangsläufig unter der Kontrolle des Entwicklerteams stehen, wird sich in der Strategie auf die Vereinheitlichung der lokalen Entwicklungsumgebung und der Pipeline beschränkt. Hierzu empfiehlt sich die Nutzung von Containerization, wobei die Abhängigkeiten zum Ausführen der Shopware-Platform und der CI-Tools gebündelt und wiederverwendet werden können. Ein lokal entwickeltes Feature kann so nach der Fertigstellung über ein VCS integriert werden, welches dann die Pipeline-Orchestration anstößt. Die Pipeline soll zur Ausführung der einzelnen Jobs das gegebene Image instanziieren und als Ausführende Umgebung verwenden, sodass diese möglichst der lokalen Entwicklungsumgebung gleicht. Nach dem erfolgreichen Durchlaufen des Software-Builds, Tests und Qualitätschecks soll die Software letztlich durch einen Deployment-Job innerhalb der Pipeline automatisiert an verschiedene Umgebungen ausgeliefert werden können. Hierbei kann es sich um die Produktionsumgebung oder andere Infrastrukturen handeln, wie zum Beispiel einen Development-Server zum Testen von Features oder einen Staging-Server für die Abnahme von neuen Änderungen durch Projekt-Kunden.



Quelle: Eigene Darstellung nach Combe, Martin und Di Pietro (2016)

Abbildung 5: Geplante Architektur der CI-Strategie

Eine Visualisierung der geplanten Architektur für die CI-Strategie kann in Abbildung 5 eingesehen werden. Die Darstellung zeigt die Struktur der lokalen Entwicklungsumgebung so wie der späteren möglichen Deployment-Umgebungen in Zusammenhang mit dem VCS und des Pipeline-Runners auf. Optionale Services und Umgebungen werden in Blau dargestellt, der Lebenszyklus von Integrationen in der Strategie wird durch die Pfeile zwischen den Teilbereichen verdeutlicht. Die lokale und die Deployment-Umgebungen basieren dabei auf Hardware-Ressourcen, auf denen jeweils ein Betriebssystem (Host OS) installiert ist, welches das Nutzen von Host Libraries ermöglicht. Als Host Libraries werden in diesem Fall die installierten Programme und Services

auf dem Betriebssystem bezeichnet, welche zum Beispiel für das Ausführen des Docker-Daemons und dessen Containern oder für das direkte Ausführen der Shopware-Services und dessen Abhängigkeiten genutzt werden.

Aufbau der Pipeline

Im Mittelpunkt der CI-Strategie steht die Pipeline zum automatischen Bauen, Testen und Ausliefern der Software. Nachdem die Umgebung für Projekte und der grundsätzliche Ablauf von Integrationen in der Strategie definiert wurde, wird nun die eigentliche Pipeline entworfen. Da diese verschiedene Phasen durchläuft, werden nachfolgend die für die Pipeline angedachten Stages und dessen Aufbau erläutert.

- **Build-Stage**

  Ungeachtet der späteren ausführenden Umgebung muss in der CI-Pipeline eines Shopware-Projekts zur Durchführung weiterer Schritte eine Build-Phase durchlaufen werden. Hierbei werden die Abhängigkeiten der Shopware-Platform selbst zusammen mit weiteren Paketen wie Testing- und QA-Tools installiert und somit für die Testing-Phase vorbereitet. Die hierbei stattfindende Integration wird also zunächst durch den Status des Software-Builds geprüft. Ist der Build-Job erfolgreich, wird ein Artifact erzeugt, welches, um die Durchlaufzeit der Pipeline insgesamt möglichst gering zu halten, in einem Cache zur weiteren Nutzung zwischengespeichert wird.

- **Testing-Stage**

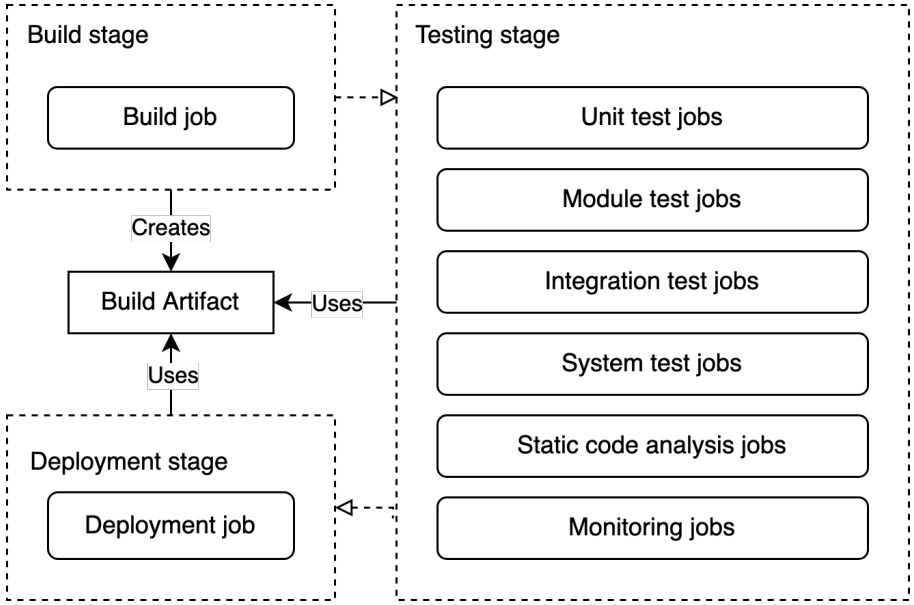
  Wenn die Build-Stage erfolgreich durchgelaufen ist und ein Artifact erzeugt wurde, beginnt die Testing-Phase. In der Strategie ist hierbei das parallele Ausführen von Tests und QA-Tools auf der bestehenden Codebase angedacht. Um Tests parallel auszuführen, werden mehrere Jobs gestartet, welchen bereits alle installierten Tools und der zu testende Quellcode durch das zwischengespeicherte Artifact des Build-Jobs zur Verfügung stehen. Hierbei sollte eine möglichst große Abdeckung des Codes durch verschiedene Arten von Software-Tests bestehen. Durch Unit-, Module- und Integration-Tests kann an dieser Stelle die Funktionalität der einzelnen Anwendungskomponenten überprüft werden, während System-Tests die Korrektheit des Gesamtsystems anhand der Geschäftsanforderungen abdecken. Besondere Anforderungen einzelner Testing-Tools, wie zum Beispiel ein Datenbank-Service innerhalb der Pipeline zum Ausführen von System-Tests, sollten dabei nicht in der Build-Phase, sondern vor der Ausführung des Tools im jeweiligen Job installiert werden. Parallel zu den verschiedenen Test-Arten können in diesem Schritt Tools zur Static-Code-Analysis ausgeführt werden. Die Testing-Stage umfasst somit die automatisierte Qualitäts- und Funktionalitätsprüfung der Shop-Software in der CI-Pipeline. Dabei anfallende Metriken und Status sollen kontinuierlich überwacht werden, wobei auch Monitoring-Tools zur weiteren Prüfung der Applikationsperformance und anderen Qualitätsindikatoren eingesetzt werden können.

- **Deployment-Stage**

  Wenn der Software-Build inklusive aller automatisierten Tests und Qualitätschecks erfolgreich durchlaufen, kann eine Deployment-Stage gestartet werden. In der Strategie ist dessen

Ausführung für Branches angedacht, bei denen eine Ziel-Umgebung zur Auslieferung des gebauten Artifacts existiert, wie zum Beispiel die Produktionsumgebung. Der hierbei ausgeführte Deployment-Job liefert die relevanten Daten der Shop-Software, Testing-Tools und CI-Abhängigkeiten ausgeschlossen an die definierte Umgebung aus. Hierbei muss beachtet werden, dass die Ausfallzeit der Applikation bei der automatischen Auslieferung so gering wie möglich bleibt, um die technischen Anforderungen der Strategie zu erfüllen.

Eine Zusammenfassung des Aufbaus und der Phasen der Pipeline kann in Abbildung 6 eingesehen werden. Hierbei werden die einzelnen Phasen in gestrichelt dargestellt, diese enthalten die Arten von Jobs, welche in der jeweiligen Phase aufgerufen werden. Der Ablauf der Pipeline wird durch gestrichelte Pfeile dargestellt, welche die Reihenfolge der Phasen verdeutlichen. Außerdem wird aufgezeigt, dass der Build-Job ein Software-Artifact erzeugt, auf welchem die weiteren Jobs der Test- und Deployment-Stage basieren. Die in einem Kundenprojekt entwickelte Pipeline soll dieser Struktur folgen, wobei allerdings keine festen Tools oder Versionen vorgegeben sind. Bei jedem neuen Projekt sollte also vorher eine Analyse der verfügbaren Testing und QA-Tools durchgeführt werden, um immer auf dem aktuellsten Stand zu sein.



Quelle: Eigene Darstellung

Abbildung 6: Phasen und Abhängigkeiten der konzipierten CI-Pipeline

Diese Struktur ist in der Strategie zunächst für die Durchführung der Pipeline bei allen Ausgangs-Branches angedacht. Da Jobs allerdings eine lange Durchlaufzeit haben können, soll die Möglichkeit bestehen, diese nur unter bestimmten Bedingungen auszuführen. Um die Durchlaufzeit gering zu halten ($<= 10min$), können lang-andauernde System-Tests oder Performance-Monitoring-Jobs nur bei der Integration in einen Branch vorgenommen werden, für den eine Deployment-Umgebung definiert ist. Die Strategie soll somit Flexibilität bieten und an die individuellen Bedürfnisse eines Projekts angepasst werden können.

4 Umsetzung der CI-Strategie

Im Folgenden wird die zuvor konzipierte CI-Strategie in die Praxis umgesetzt und dabei in einer Fallstudie exemplarisch auf ein Projekt angewandt. Zunächst wird das ausgewählte Projekt vorgestellt, um einen Kontext für die anschließende Implementierung zu schaffen. Dabei werden die spezifischen Anforderungen und Besonderheiten des Projekts hervorgehoben. Im Anschluss daran wird die praktische Umsetzung der CI-Strategie beschrieben, wobei sowohl die technischen Aspekte als auch die organisatorischen Prozesse berücksichtigt werden. Hierbei werden CI-Tools für die Strategie ausgewählt, eine Projekt-Struktur und Pipeline definiert und der eigentliche Shop inklusive angedachter Eigenentwicklungen implementiert.

4.1 Projekthintergrund

Bei dem Shopware-Projekt, in welchem die CI-Strategie implementiert werden soll, handelt es sich um einen Anbieter von Kochutensilien. Der Shop soll dabei auf der Basis von Shopware 6 neu entwickelt werden und das zuvor erstellte CI-Konzept in der Praxis einsetzen. Das Ziel ist hierbei, durch die Integration der untersuchten CI-Praktiken eine robuste, skalierbare und effiziente Entwicklungsumgebung zu schaffen, die den Anforderungen moderner Online-Shops gerecht wird. Dabei wird das zuvor ausgearbeitete CI-Konzept als Leitfaden für die Entwicklung, Optimierung und kontinuierliche Qualitätssicherung des Shops herangezogen. Hierzu wird zunächst eine Shopware-Instanz aufgesetzt, welche anschließend über den Standardumfang der Software hinaus um ein eigenes Theme und einige Plugins zur Anpassung verschiedener Bereiche des Shops ergänzt wird. Die für den Shop angedachten Eigenentwicklungen umfassen folgende Plugins:

- **Eigene Produktkacheln auf Listing-Seiten**

  Bei Varianten-Artikeln wird keine direkte Auswahlmöglichkeit angeboten, stattdessen werden individuell gestaltete Produktkacheln präsentiert.

- **Eigener Produkt-Typ „Rezept“**

  Dieser spezielle Produkttyp kann vom Kunden betrachtet, aber nicht gekauft werden. Er dient zur Präsentation von Rezepten, die mit den im Shop erhältlichen Produkten zubereitet werden können.

Als Produkt-Listing-Seiten werden Seiten der Shopware-Instanz bezeichnet, welche die Produkte einer bestimmten Kategorie oder Gruppierung gesammelt in einer Liste anzeigen. Produkte werden auf diesen Seiten im Standardumfang der Software als rechteckige Kacheln angezeigt, welche verschiedene Informationen anzeigen, wie zum Beispiel eine Auswahl an Varianten des Artikels. Durch das Klicken auf eine der Kacheln wird auf die jeweilige Produkt-Detail-Seite verwiesen. Diese Seiten zeigen die Detail-Ansicht der Produkte auf und bieten weitere Informationen und Varianten- und Kaufoptionen. Für einige weitere benötigte Features des Shops wie eine Filialsuche und Social-Media-Feeds wurde der Einsatz von Drittanbieter-Plugins eingeplant, um Entwicklungszeit einzusparen.

4.2 Implementierung des Konzepts

Nachfolgend wird das als Fallstudie vorgesehene Shopware-Projekt anhand der untersuchten CI-Praktiken implementiert. Zunächst wird gemäß dem Konzept eine Analyse der zur Verfügung stehenden CI-Tools durchgeführt, woraufhin Technologien für die Nutzung innerhalb des Projekts ausgewählt werden. Anschließend wird die Projekt-Umgebung für die Nutzung von CI aufgesetzt und das Shopware-Projekt installiert und vorbereitet. Daraufhin wird die Pipeline für das erstellte Projekt angelegt. Hierbei werden die Phasen und Jobs erstellt und CI-Tools konfiguriert, um automatisierte Tests und Analysen durchführen zu können. Nachdem das Projekt initialisiert wurde und die Pipeline funktionsfähig ist, wird die Implementierung der eigentlichen Shop-Funktionen erläutert, wobei der Fokus auf die kontinuierliche Prüfung der Features durch die Pipeline gerichtet ist.

Auswahl von CI-Tools

Bevor mit der Implementierung des Projekts begonnen werden konnte, musste zunächst eine Auswahl an Tools getroffen werden, die zur Umsetzung der CI-Strategie eingesetzt werden können. Hierfür wurden populäre Tools aus verschiedenen Bereichen, die in dem Projekt abgedeckt werden sollen, herangezogen:

Version Control System

- Git / GitLab

Pipeline-Runner

- GitLab CI/CD

PHP Testing Tools

- PHPUnit
- Infection

JavaScript Testing Tools

- Jest
- Cypress

PHP Static-Code-Analysis Tools

- PHP_CodeSniffer
- PHP Mess Detector
- PHPStan
- Deptrac
- License Checker
- Security Checker

JavaScript Static-Code-Analysis Tools

- Eslint
- Danger JS
- License Checker
- AuditJS

Deployment Tools

- Deployer

Eine vollständige Übersicht und Beschreibung der eingesetzten CI-Tools kann aus Anhang A entnommen werden.

Aufsetzen der Projekt-Umgebung

Nach dem Festlegen der zu verwendenden Technologien wurde mit der Implementierung des Projekts begonnen, wobei zuerst eine Umgebung für die Ausführung von Shopware und der gewählten CI-Tools erstellt wurde. Hierbei wurde sich für die Nutzung von Docker als Containerization-Technologie entschieden, da Shopware selbst ein Docker-Image für das Betreiben der Plattform

bereitstellt.[57] Auf der Basis des bereitgestellten Images wurde eine Shopware-Instanz aufgesetzt und anschließend eine Datenbank angebunden. Diese wurde, um in der CI-Pipeline wiederverwendet werden zu können, auch aus einem Docker-Image instanziiert. Nach der Shopware-Installation wurden Grundkonfigurationen vorgenommen und erste manuelle Tests durchgeführt, um sicherzustellen, dass die Instanz funktional ist. Um die verschiedenen lokalen Container-Konfigurationen zentralisiert verwalten zu können, wurde beim Aufsetzen der lokalen Umgebung das Tool „Docker-Compose“ verwendet. Dieses Tool eignet sich besonders für Multi-Container Docker-Applikationen, da in der Konfiguration hinterlegte Container mit je einem Konsolenbefehl initialisiert, gestartet, gestoppt oder entfernt werden können.[58] Hierfür wurde eine Konfigurationsdatei im YAML-Format angelegt, welches zur Konfiguration von Programm-Daten genutzt wird:

```yaml
version: "3"

services:
  shop:
    container_name: shop
    image: shopware/development:8.2-composer-2
    ports:
      - "80:80"
    volumes:
      - ./shopware:/app
    networks:
      - web

  database:
    container_name: database
    image: mariadb:10.3
    env_file:
      - .env
    ports:
      - "3306:3306"
    volumes:
      - "database_volume:/var/lib/mysql"
    networks:
      - web

volumes:
  database_volume:
    driver: local

networks:
  web:
    external: false
```

Die aufgezeigte Konfigurationsdatei definiert zwei Container: `shop` und `database`. Diese sind über ein angegebenes Netzwerk `web` miteinander verbunden und haben jeweils offene Ports zur Kommunikation hinterlegt. Außerdem wurde das Docker-Volume `database_volume` für das Persistieren der lokalen Datenbank angelegt, welches sich um dessen Datenverwaltung kümmert. Der Shop-Container nutzt dabei auch eine Volume-Anweisung in der Konfigurationsdatei, um die Projekt-Dateien und Verzeichnisse in die Image-Instanz zu laden. Darüber hinaus wurde für die Datenbank die zuvor angelegte Konfigurationsdatei `.env` angegeben, in der Umgebungsvariablen in Form von Tabellen-Konfiguration und Nutzerzugangsdaten hinterlegt werden konnten.

[57] Vgl. Shopware. *shopware/development - Docker Image*. 2023.
[58] Vgl. Docker Inc. *Docker Compose overview*. 2023.

Nachdem die lokale Entwicklungsumgebung erstellt wurde, konnte mit der Vorbereitung für die Umgebung der Pipelines begonnen werden. Hierzu wurde das gesamte Projekt in GitLab hinterlegt und ein Branching-Modell eingeführt, um sicherzustellen, dass neue Features und Bugfixes vor der Integration in den Hauptzweig (`main`) in isolierten Branches entwickelt werden. Neben dem `main` -Branch, welcher als Deployment-Ziel die Produktionsumgebung hinterlegt hat, wurden auch ein Development-Branch (`development`) und -Server angelegt und eine Staging-Umgebung aufgesetzt. Der Staging-Server ist dabei ein variables Deployment-Ziel, für welches die Auslieferung von `release` -Branches vorgesehen ist. Die hierbei verwendete Branching-Strategie nennt sich „Git-Flow“. Diese führt Namenskonventionen und verschiedene Arten von Branches ein, wobei neue Features und Anpassungen zunächst in `feature` -Branches entwickelt werden. Anschließend werden diese in den `development` -Branch gemerged, wo die Änderungen gesammelt werden. Die gesammelten Features können daraufhin in einen `release` -Branch überführt werden, in welchem keine substanziellen Änderungen, sondern nur noch Fehlerbehebungen vorgenommen werden. `release` -Branches können zuletzt in den `main` -Branch integriert werden. Dieser stellt den Haupt-Versionsstand des Software-Projekts dar und speichert die aktuell ausgelieferte Version der Software. Neben diesen vier Arten von Branches gibt es noch die Möglichkeit, `hoftix` -Branches für Fehlerbehebungen, welche am Haupt-Versionsstand vorgenommen werden müssen.[59]

Erstellen der Pipeline

Nachdem eine ausführende Umgebung definiert wurde, konnte die eigentliche Pipeline für das Projekt erstellt werden. Diese soll zunächst bei jeder Integration in einen Branch ausgeführt werden und durchläuft dabei eine Build-, Testing- und Deployment-Stage. Im Build-Prozess werden die CI-Tools und die Abhängigkeiten der Shopware-Platform installiert. Der Testing-Prozess umfasst zunächst alle für die Strategie ausgewählten Testing- und QA-Tools, inklusive lang-andauernder Tests durch End-to-End-Testing. Im späteren Verlauf der Entwicklung besteht durch Gitlab CI/CD die Möglichkeit, diese Tests nur in Branches mit definierter Deployment-Umgebung durchführen zu lassen, falls dessen Laufzeit zu groß wird. Abschließend liefern Jobs in der Deployment-Stage den entwickelten Code an die jeweilig für den Branch definierte Umgebung aus.

Die hierbei implementierten Phasen und Jobs der Pipeline wurden zur Ausführung in GitLab CI/CD in Form von YAML-Dateien angelegt. Erstellte Pipeline-Konfigurationsdateien werden dabei im Repository des Projekts hinterlegt, welches diese ausliest und die Anweisungen an GitLab CI/CD weitergibt. Somit konnten die einzelnen Stages der Pipeline und darin durchzuführende Jobs angelegt werden, sowie dessen Ausführungsregeln, Beziehungen zueinander, anfallende Artifacts und Abhängigkeiten in Form von Services. Nachfolgend wird ein kurzer Überblick der Implementierung einzelner Phasen der Pipeline in GitLab CI/CD gegeben:

- **Build-Stage**

  Die Build-Stage besteht aus einem einzelnen Build-Job, welcher die Installation der Abhängigkeiten von Shopware und der CI-Tools durch Composer und NPM vornimmt. Im

[59] Vgl. Driessen. *A successful Git branching model.* 2010.

Folgenden wird die erstellte Konfiguration dieses Jobs für GitLab CI/CD aufgezeigt:

```yaml
build-shopware-project:
  stage: build
  image: shopware/development:8.2-composer-2
  script:
    # Install shopware
    - cd "$PROJECT_ROOT"
    - composer install --no-interaction --optimize-autoloader --no-scripts

    # Install ci-tools
    - cd "$CI_PROJECT_DIR" && composer install --no-interaction && npm install
  rules:
    - !reference [ .rules, run-always ]
  artifacts:
    paths:
      - vendor/
      - node_modules/
      - cache/Cypress/
      - shopware/vendor/
```

Die hierbei erzeugten Dateien werden als Artifacts für die Nutzung durch darauffolgende Jobs hinterlegt. Außerdem referenziert der Build-Job die Regel „run-always“, welche zuvor angelegt wurde und dafür sorgt, dass der Job bei jeder Integration in das Repository und auch bei Merge-Requests gestartet wird.

- **Testing-Stage**

  In der Testing-Stage werden die für das Projekt vorgesehenen Test- und QA-Tools durchgeführt. Die Test-Jobs nutzen dabei die in der Build-Stage erstellten Artifacts, anstatt diese selbst erneut zu installieren, um die Ausführung der Pipeline insgesamt zu beschleunigen. Als Beispiel wird nachfolgend die Job-Konfiguration für das Unit-Testing-Tool `PHPUnit` aufgezeigt:

```yaml
shopware-php-unit-tests:
  stage: test
  dependencies:
    - build-shopware-project
  script:
    # Execute phpunit
    - vendor/bin/phpunit --log-junit=reports/phpunit_report.xml
  rules:
    - !reference [ .rules, run-always ]
  artifacts:
    paths:
      - test-results/
    reports:
      junit: reports/phpunit_report.xml
```

Dieser Job gibt als Abhängigkeit den zuvor definierten Build-Job an und kann somit dessen installierte Artifacts nutzen. Da Unit-Tests in der Regel keine besonders zeitaufwändigen Tests darstellen, referenziert dieser Job auch die „run-always“-Regel. Außerdem generiert der Job eigene Artifacts in Form von Test-Reports, welche in GitLab CI/CD ausgewertet und in der jeweiligen Pipeline angezeigt werden. Tests, die weitere Abhängigkeiten voraussetzen, so wie Functional-Tests durch das Tool `Cypress`, folgen dabei dem gleichen Konfigurationsprinzip. Sie erweitern lediglich die Job-Konfiguration, um die benötigten

Services, Umgebungsvariablen und weitere Abhängigkeiten, die spezifisch für den jeweiligen Test benötigt werden, zu installieren. Im weiteren Verlauf des Projekts können eigene spezielle Regeln für lang-andauernde Test-Jobs erstellt werden, sodass diese nur bei bestimmten Branches oder Aktionen im VCS ausgeführt werden.

- **Deployment-Stage**

  Nach dem Erstellen der Build- und Testing-Stage wurde zuletzt die Deployment-Stage angelegt. Hierbei wurden verschiedene Jobs definiert, die jeweils zur Ausführung bei einer Integration in einen bestimmten Ziel-Branch konfiguriert wurden. Nachfolgend wird der Deployment-Job des `main`-Branches aufgezeigt:

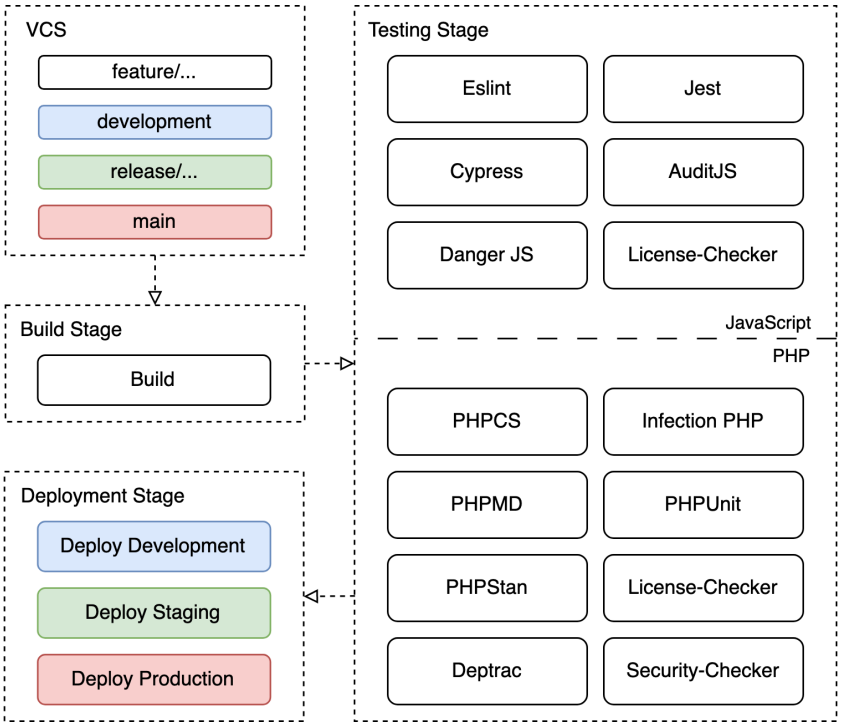
```
deploy-shopware-production:
  stage: deploy
  dependencies:
    - build-shopware-project
  script:
    # Deploy to production
    - vendor/bin/dep deploy_production
  only:
    - main
  when: manual
```

  Der Job wurde so konfiguriert, dass die finale Ausführung zwar automatisiert abläuft, dabei aber manuell angestoßen werden muss, womit die umgesetzte Strategie effektiv Continuous Delivery betreibt. Dies kann je nach den Bedürfnissen des Projekts angepasst werden und ermöglicht somit optional auch die Nutzung von CD durch das vollautomatische Ausliefern nach dem erfolgreichen Durchlaufen der Build- und Testing-Jobs. Für den `development`- und die `release`-Branches, für welche auch Deployment-Umgebungen angelegt wurden, sind jeweils eigene Jobs definiert worden. Diese Jobs folgen dabei demselben Prinzip und nutzen das Tool `Deployer`, welches das Ausliefern der Software an die jeweilige Umgebung anhand von eigens vorgegebenen Konfigurationen vornimmt. Die hierbei angelegten `Deployer`-Konfigurationen für die verschiedenen Umgebungen wurden in Form von PHP-Code im Repository persistiert.

Die grundlegenden Phasen und Jobs der Pipeline konnten somit anhand der erarbeiteten Strategie modelliert und implementiert werden. Sowohl der Build-Prozess als auch das automatisierte Testen und das Ausliefern des integrierten Codes wurden hierbei mit einbezogen und umgesetzt. Die Pipeline-Konfiguration bietet dabei eine gewisse Flexibilität und ermöglicht die Anpassung an verschiedene Vorgaben und Anforderungen, welche sich im Verlauf des Projekts ändern können.

In Abbildung 7 wird die für das Projekt erstellte CI-Pipeline visualisiert. Hierbei werden verschiedene Arten von Branches in einem VCS dargestellt, welche zunächst die gleiche Aktion auslösen, den Build-Prozess. Dort werden die CI-Tools und die Abhängigkeiten von Shopware für die anschließende Nutzung in der Testing-Stage installiert. Die Testing-Stage wird nach erfolgreichem Durchlaufen der Build-Stage ausgeführt und ist in der Abbildung in zwei Teile separiert. Im oberen Teil werden die gewählten Testing- und QA-Tools für JavaScript-Code der Strategie aufgezeigt, während der untere Teil die Tools für PHP darstellt. Wenn alle Jobs der Testing-Stage erfolgreich durchgelaufen sind, wird je nach Art des Ausgangs-Branches ein

Deployment-Prozess für die jeweilige Umgebung angestoßen. Die Deployment-Stage wird nur für Branches mit definierter Umgebung ausgeführt und liefert dabei den Code des Ausgangs-Branches an den Development-, Staging- oder Produktionsserver aus.



Quelle: Eigene Darstellung

Abbildung 7: Visualisierung der implementierten CI-Pipeline

Eine visuelle Ausgabe einzelner durchgeführter Pipelines des Projekts in GitLab CI/CD kann aus Anhang B entnommen werden.

Implementierung der Shop-Funktionen

Durch die zuvor erstellten Umgebungen und die angelegte Pipeline-Konfiguration konnten nun die geplanten Funktionen des Shops unter der Nutzung der erarbeiteten CI-Praktiken implementiert werden. Diese können grob in drei verschiedene Bereiche eingeteilt werden:

- **Entwicklung des Shop-Themes**

  Bei der Entwicklung des Themes für die Shopware-Instanz handelt es sich um viele kleine Anpassungen an dem bestehenden User Interface (UI) des Standardumfangs. Da UI-Anpassungen aufwendig zu testen sind und die Standard-UI durch Shopwares eigene Tests abgedeckt ist, wurde das Theme ohne weitere Tests angelegt. Für Anpassungen, welche substanziell die Funktion von UI-Komponenten verändern, können jedoch System-Tests angelegt werden, wobei geprüft wird, ob sich die angepasste UI nach den gegebenen Vorschriften verhält. Dies hat den Vorteil, dass Bugs, die durch Updates der Standard-UI mit bestehenden Anpassungen entstehen, schneller erkannt und gefixed werden können.

- **Entwicklung des Plugins zur Anpassung der Produktkacheln**

  Um die angepassten Produktkacheln zu entwickeln, wurde ein eigenes Plugin angedacht, welches bestimmte Produkt-Daten in den Kacheln auf Produkt-Listing-Seiten des Shops anzeigt. Hier wird im Standard-Umfang der Shopware-Platform eine Auswahl an Varianten angeboten, welche durch das Plugin ersetzt werden soll. Statt der Variantenauswahl soll an dieser Stelle der Preis der günstigsten Variante angezeigt werden. Bei der Integration dieses Plugins konnten verschiedene Bereiche durch den Einsatz der Pipeline auf ihre Richtigkeit geprüft werden. Da die günstigste Artikelvariante zunächst Backend-seitig bestimmt und dem Frontend zur Verfügung gestellt werden musste, wurde bei der Entwicklung PHP-Code produziert. Für diesen konnten Unit-, Module- und Integration-Tests angelegt und dazugehörige UI-Anpassungen im Frontend durch System-Tests abgedeckt werden. Die generelle Einhaltung gesetzter Code-Standards und weitere Qualitätsmerkmale wurden hierbei durch die gewählten statischen Code-Analyse-Tools überwacht.

- **Entwicklung des Plugins für Rezept-Produkte**

  Das Plugin für die Erstellung von Rezepten legt eine eigene Produkt-Art an, welche nicht aktiv im Shop gekauft werden kann und dabei lediglich die hinterlegten Rezept-Informationen anzeigt. Da dieses Feature sowohl Backend- als auch Frontend-Anpassungen benötigt, konnte der entwickelte PHP- und JavaScript-Code ebenfalls durch Unit-, Module- und Integration-Tests abdeckt werden. Die frontend-seitigen Erweiterungen bestimmter Produkt-Kacheln im Listing und der Produkt-Detail-Seite, welche nun Rezepte statt Produkten anzeigen, wurden dabei auch durch System-Tests abgedeckt. Vorteilhaft bei dieser Art der automatisierten Prüfung der erstellten Features ist, dass mit neuen Entwicklungen eingeführte Wechselwirkungen und Fehler schneller entdeckt werden können. So wurde zum Beispiel durch Regression- und System-Tests verhindert, dass das Anpassen der Produktkacheln durch beide erstellten Plugins zu unentdeckten Problemen nach der Integrationsphase führt.

Bei der Entwicklung der Anpassungen wurden verschiedene CI-Praktiken und Tools eingesetzt. Der für die Anpassungen entwickelte PHP- und JavaScript-Code konnte hierbei durch automatisierte Unit-, Module- und Integration-Tests der Tools `PHPUnit` und `Jest` abgedeckt werden. Erstellte Unit-Tests in PHP wurden durch das `Infection`-Framework mit Mutation-Tests kontinuierlich evaluiert. System-Tests konnten mit dem JavaScript-basierten Tool `Cypress` erstellt und durchgeführt werden. Durch `Deptrac` konnte die Architektur der erstellten PHP-Klassen und Module geprüft werden. Die Einhaltung gesetzter Coding-Standards und weitere Qualitätsmetriken konnten durch die Nutzung von `PHPCS`, `PHPMD` und `PHPStan` für PHP-Code und von `Eslint` für JavaScript-Code erhoben werden. `AuditJS` und der PHP `Security Checker` konnten installierte Pakete automatisch auf Sicherheitslücken überprüfen. Um die Lizenzen der im Projekt installierten Pakete für das Projekt zu überwachen, wurden `License Checker` eingesetzt, welche die Lizenzen der von Composer und NPM installierten Pakete beobachten. Außerdem wurde `Danger JS` zur Steigerung der Transparenz des CI-Prozesses eingesetzt, welches den Status der einzelnen Jobs einer durchgelaufenen Pipeline in Form eines Kommentars im VCS hinterlegt. Letztlich wurden die Deployments für die jeweiligen Umgebungen durch das PHP-basierte Tool `Deployer` durchgeführt, wodurch die Ausfallzeiten beim Ausliefern der Software minimiert werden konnten.

Drittanbieter-Plugins und Shop-Konfigurationen wurden hierbei nicht weiter getestet, da es sich bei ihnen nicht um Eigenentwicklungen handelt und diese durch die jeweiligen Hersteller oder die Shopware-Platform selbst geprüft werden. Des Weiteren wurden im Verlauf des Projekts neben den automatisierten Tests durch die CI-Pipeline auch manuelle Tests von Entwicklern, Projektleitern und den Kunden durchgeführt, um die System-Funktionalität sicherzustellen. Darüber hinaus wurde durch die Verwendung einer flexiblen Pipeline die Möglichkeit eingeräumt, das Projekt durch Performance- und Last-Tests zu überwachen oder weitere Tools und Services zum Testen der Applikation anzubinden. Dank des parallelen Ausführens der Test-Jobs konnte die Durchlaufzeit der Pipeline minimiert werden, wobei die Strategie auch die Möglichkeit bietet, Jobs konditionell auszuführen, sollte deren Dauer die gesetzte Zehn-Minuten-Marke übersteigen.

5 Evaluierung

Nach der Implementierung der CI-Strategie im vorherigen Abschnitt folgt nun die kritische Evaluierung des Konzepts anhand der entwickelten Fallstudie. In diesem Kapitel wird die Effektivität und Anwendbarkeit der umgesetzten CI-Strategie im Kontext des erstellten Shopware-Projekts untersucht. Dabei wird nicht nur die technische Umsetzung in den Blick genommen, sondern auch die praktischen Auswirkungen der Strategie auf den Entwicklungsprozess und das Endprodukt. Zunächst wird eine Analyse der Durchlaufzeit verwendeter Tools und durchgeführter Phasen der Strategie durchgeführt. Dabei wird sowohl die lokale als auch die CI-Umgebung betrachtet. Anschließend wird anhand der durchgeführten Analyse eine Bewertung der Strategie vorgenommen, wobei Rückschlüsse auf dessen Qualität und Effektivität gezogen, Erkenntnisse dargelegt und Auswirkungen diskutiert werden.

5.1 Analyse und Vergleich

Um die Effektivität der eingesetzten Strategie bewerten zu können, wurde zunächst eine Analyse der Durchlaufzeit eingesetzter Tools durchgeführt. Um eine vergleichbare Grundlage zu bieten, wurde die Zeit, die für die Durchführung einzelner Tools benötigt wird, je dreimal lokal und in einer CI-Pipeline gemessen und daraus die durchschnittliche Ausführungszeit ermittelt.

Name des Tools	Durchlaufzeit CI	Durchlaufzeit lokal
AuditJS	$47s$	$19s$
Cypress	$2m\ 16s$	$1m\ 32s$
Danger JS	$1m\ 1s$	–
Eslint	$51s$	$12s$
Jest	$54s$	$11s$
(JS) License Checker	$55s$	$14s$
Deptrac	$36s$	$5s$
Infection	$1m\ 11s$	$47s$
PHPCS	$44s$	$2s$
PHPMD	$42s$	$3s$
PHPUnit	$1m\ 7s$	$14s$
PHPStan	$38s$	$19s$
PHP Security Checker	$41s$	$9s$
(PHP) License Checker	$44s$	$7s$
Deployer	$5m\ 18s$	$5m\ 58s$

Tabelle 1: Durchlaufzeit der Test-, QA- und Deployment-Tools

In Tabelle 1 werden die durchschnittlichen Zeiten für die Ausführung einzelner Test-, QA- und Deployment-Tools in der lokalen Entwicklungsumgebung und in der CI-Pipeline aufgezeigt. Es zeigt sich, dass das lokale Durchführen der Tools in der Regel schneller ist als in der CI-Umgebung. Dies lässt sich auf verschiedene Faktoren zurückschließen, wie der Mehraufwand durch das Initialisieren der einzelnen Jobs und dessen Images im Pipeline-Runner, Wartezeiten durch die Speicherung von Job-Artifacts, Unterschiede in der genutzten Hardware oder weiteren Umständen. Da in der Tabelle nur die rohen Ausführungszeiten der Tools dargestellt werden,

muss außerdem beachtet werden, dass bei der manuellen lokalen Durchführung zwischen jedem Tool eine Unterbrechung stattfindet. Dies ist dadurch bedingt, dass die testende Person die einzelnen Befehle zur Ausführung der Tools heraussuchen, in die Kommandozeile einfügen und somit von Hand durchführen muss. Da `Danger JS` nur in einer CI-Pipeline genutzt werden kann, wurde dieses bei den lokalen Tools ausgelassen. Bei der Ausführung der Tools in der Pipeline werden diese teils zeitgleich gestartet und laufen parallel, was die Durchlaufzeit der Phasen der Pipeline insgesamt verkürzt. Die Testing-Phase, welche Tools in Jobs parallel ausführt, dauert also nur so lange wie der langsamste Job. Anhand der gemessenen Ausführungszeiten der Tools wurden anschließend Rückschlüsse auf die Durchlaufzeit der Integrationsphasen des Projekts gezogen.

Name der Phase	Durchlaufzeit CI	Durchlaufzeit lokal
Build	$2m\ 12s$	$2m\ 47s$
Test	$2m\ 16s$	$4m\ 14s$
Deploy	$5m\ 6s$	$5m\ 58s$
Gesamt	$9m\ 34s$	$12m\ 59s$

Tabelle 2: Dauer der Phasen einer Integration

In Tabelle 2 werden die errechneten Durchlaufzeiten der Integrationsphasen des Projekts aufgezeigt. Die aufgelistete Zeit für die Build-Phase der CI-Umgebung wurde hierbei aus dem Durchschnittswert von drei in Pipelines ausgeführten Build-Jobs errechnet. Für die Build-Phase in der lokalen Umgebung wurden dreimal die einzelnen Schritte des Vorgangs gemessen und summiert, woraus anschließend auch der Durchschnitt ermittelt wurde. Die Zeiten der Test- und Deploy-Phasen stützen sich auf die zuvor gemessenen Ausführungszeiten der im Projekt verwendeten Tools. Aus den Daten kann entnommen werden, dass eine Integration aus der lokalen Entwicklungsumgebung heraus theoretisch nicht signifikant länger andauert als eine Integration durch die CI-Umgebung. Hierbei wird allerdings außer acht gelassen, dass die benötigte Zeit zur lokalen Ausführung des Build-Prozesses sowie der Tests, QA-Tools und der Deployment-Lösung in der Praxis viel höher sein kann. Da Entwickler diese aktiv ausführen müssen, kommt es zwischen jedem Tool zu Verzögerungen. Somit stellt die zur lokalen Durchführung errechnete Zeit gleichzeitig auch die bestmöglich zu erreichende Zeit für den Integrationsprozess aus dieser Umgebung heraus dar. Außerdem stützt sich die dargestellte Durchlaufzeit für lokale Deployments auf der Annahme, dass `Deployer` bereits vor der Implementierung der CI-Strategie zur automatisierten Auslieferung genutzt wurde. Ein vollständig manuell durchgeführtes Deployment würde hierbei wieder für Verzögerungen zwischen einzelnen Schritten führen und die Durchlaufzeit der Phase erhöhen.

Durch die Analyse-Tools von GitLab CI/CD konnte außerdem ermittelt werden, dass im Verlauf des Projekts insgesamt 132 Pipelines gestartet wurden, wovon 56 durch Fehler im Integrationsprozess abgebrochen wurden. Die Fehlerrate von Integrationen in der durchgeführten Fallstudie beträgt somit circa 42%. Diese hohe Fehlerzahl lässt sich zum Teil durch das häufige Fehlschlagen von Pipelines im frühen Entwicklungsstadium des Projekts erklären. Da anfangs die Erstellung einzelner Jobs und die ersten Konfigurationen von Test-, QA- und Deployment-Tools vorgenommen wurden, fielen in dieser Phase des Projekts vermehrt Fehler bei dessen Ausführung an.

Insgesamt geben die in Tabelle 1 und 2 aufgeführten Zeiten nur eine grobe Richtung vor, da verschiedene Aspekte wie die Qualität der vorhandenen Internetverbindung und Hardwareressourcen, Wartezeiten von Jobs im Pipeline-Runner und weitere Faktoren die gemessenen Zeiten beeinflussen können. Sie können allerdings in Betrachtung der zuvor aufgeführten Umstände einen Indikator für die weitere Bewertung des Erfolgs der Strategie bieten. Die ermittelte Fehlerrate ausgeführter Integrationen ist dabei ein weiterer hilfreicher Anhaltspunkt.

5.2 Erkenntnisse und Auswirkungen der Strategie

Die Evaluierung der im vorherigen Abschnitt aufgezeigten Durchführungszeiten liefert einige Erkenntnisse über die CI-Strategie und hilft dabei, dessen Auswirkungen auf ein Projekt nachvollziehen zu können. Es wird ersichtlich, dass die Nutzung einer CI-Pipeline zur Automatisierung von Integrationen zumindest einen kleinen zeitlichen Vorteil bietet. Betrachtet man darüber hinaus den Aspekt, dass bei der manuellen Durchführung der einzelnen Schritte einer Integration immer menschliche Verzögerungen und potenzielle Fehlerquellen hinzukommen, wird dieser Vorteil noch deutlicher. Da Integrationen in modernen und agilen Softwareprojekten oft mehrmals stattfinden, summiert sich diese Zeitersparnis mit jeder weiteren Durchführung.

Durch die automatisierte und parallele Ausführung von Integrationsphasen in einer CI-Pipeline konnte der Entwicklungsprozess insgesamt beschleunigt werden. Erstellte Features und Anpassungen konnten somit zügig und verlässlich integriert werden. Außerdem verringerte dessen Einsatz den manuellen Aufwand des Integrationsprozesses, wodurch diese Zeit für andere Aufgaben verwendet werden konnte. Durch die Steigerung der Entwicklungsgeschwindigkeit unter der Verwendung der erstellten CI-Strategie konnte somit die geschäftsseitige Anforderung und Ziel der Arbeit Z_1 erreicht werden.

Neben der Beschleunigung des Entwicklungsprozesses führte die Verwendung der CI-Strategie in der Fallstudie zu der Findung von Fehlern beim Integrationsprozess. Im Verlauf des Projekts wurden einige Pipelines aufgrund von Fehlern in der eingeführten Integration abgebrochen und konnten somit die Entwickler des Projekts auf die zugrunde liegenden Probleme hinweisen. Fehler und Inkonsistenzen konnten somit frühzeitig im Entwicklungszyklus erkannt und behoben werden. Dieser Ansatz trug zur Erfüllung des Ziels Z_2 bei, indem er die Anzahl an Fehlern, welche die ausführenden Deployment-Umgebungen der Software und somit den Endkunden erreichen, mindert.

Die CI-Strategie hat außerdem die kontinuierliche Auslieferung neuer Softwareversionen erleichtert. Durch die Automatisierung der Build- und Deployment-Prozesse konnten neue Features und Bugfixes schneller und konsistenter an die Kunden ausgeliefert werden. Hierdurch wurde die Fähigkeit des Entwicklungsteams gestärkt, auf sich ändernde Geschäftsanforderungen und Marktbedingungen zu reagieren. Die Strategie konnte somit auch zur Erreichung des Ziels Z_3 beigetragen, indem sie eine flexible und anpassbare Entwicklungsumgebung einführt, welche die kontinuierliche Weiterentwicklung und Auslieferung von Software unterstützt.

Zusammenfassend lässt sich sagen, dass die Implementierung der CI-Strategie in der Fallstudie die Effizienz und Qualität der entwickelten Software positiv beeinflusst hat. Die Vorteile in

Bezug auf Zeitersparnis, Fehlerreduktion und kontinuierliche Auslieferung haben dabei direkt zu der Erreichung der zuvor definierten geschäftsseitigen Ziele Z_n beigetragen und den Wert einer ganzheitlichen CI-Strategie weiter unterstrichen.

6 Schlussfolgerungen und Ausblick

Im folgenden abschließenden Kapitel werden die zentralen Erkenntnisse und Resultate dieser Arbeit reflektiert. Es werden Schlussfolgerungen zur konzipierten CI-Strategie aufgestellt und ein Fazit zur erstellten Arbeit gezogen, wobei der Fokus auf der praktischen Umsetzung den erreichten Zielen und den Herausforderungen liegt, die während der Implementierung aufgetreten sind. Letztlich wird ein Ausblick über mögliche zukünftige Erweiterungen und Anpassungen der Strategie gegeben, um diese noch effizienter und anpassungsfähiger im Hinblick auf zukünftige Projekte und dessen Anforderungen zu gestalten.

6.1 Fazit

Die Entwicklung und Implementierung einer CI-Strategie ist ein komplexer Prozess, der sowohl technische als auch organisatorische Herausforderungen mit sich bringt. Die konzipierte CI-Strategie, die im Laufe dieser Arbeit entwickelt und evaluiert wurde, hat gezeigt, wie effektiv der Einsatz von Continuous-Development-Techniken in modernen Softwareprojekten sein kann. Durch die Automatisierung von Build-, Test- und Deployment-Prozessen konnte die Qualität des Endprodukts insgesamt sichergestellt werden. Fehler wurden frühzeitig erkannt und behoben, wodurch die Auslieferung an den Endnutzer reibungsloser und zuverlässiger wurde. Die Implementierung umfangreicher automatisierter Tests hat nicht nur dazu beigetragen, die Softwarequalität zu erhöhen, sondern auch das Vertrauen des Entwicklerteams in den eigenen Code gestärkt. Die Transparenz, die durch die CI-Strategie geschaffen wurde, hat die Zusammenarbeit im Team unterstützt und den Entwicklungsprozess beschleunigt. Es wurde jedoch auch deutlich, dass die Einführung einer solchen Strategie eine sorgfältige Planung und Anpassung an die spezifischen Anforderungen des Projekts erfordert und die Komplexität des Entwicklungsprozesses steigert. Nicht jede CI-Lösung ist für jedes Projekt geeignet, und es ist wichtig, die richtigen Tools und Prozesse für die jeweilige Aufgabe auszuwählen.

Abschließend lässt sich sagen, dass die CI-Strategie trotz ihrer Komplexität und der damit verbundenen Herausforderungen einen positiven Einfluss auf die Entwicklung von Shopware-Projekten und dessen Effizienz und Qualität haben kann. Sie ermöglicht es den Entwicklerteams, agil auf Veränderungen zu reagieren, den Entwicklungsprozess zu optimieren und gleichzeitig die Qualität des Endprodukts zu gewährleisten. Die kontinuierliche Integration und Auslieferung fördert zudem eine Kultur der ständigen Verbesserung, in der Feedbackschleifen verkürzt und Innovationen schneller umgesetzt werden können. Die in dieser Arbeit vorgestellte CI-Strategie ist ein Beispiel dafür, wie solch ein Ansatz in der Praxis umgesetzt und optimiert werden kann. Es ist jedoch wichtig zu betonen, dass die kontinuierliche Integration und Auslieferung kein starres Konzept ist, sondern vielmehr ein dynamischer Prozess, welcher eine ständige Anpassung und Weiterentwicklung erfordert. Die Technologielandschaft, die Anforderungen der Stakeholder und die Bedürfnisse des Marktes ändern sich ständig, und es ist die Aufgabe des Entwicklerteams, darauf flexibel zu reagieren und die CI-Strategie entsprechend anzupassen. Insgesamt bietet die erstellte CI-Strategie nicht nur technische Vorteile, sondern auch wirtschaftliche und organisatorische. Sie unterstützt Unternehmen dabei, wettbewerbsfähig zu bleiben und Kundenanforderungen an Shopware-Projekte effizient zu erfüllen.

6.2 Ausblick

Die in der Arbeit erstellte CI-Strategie deckt einige wichtige Aspekte der modernen Softwareentwicklung ab. Hierbei bieten sich dennoch einige Möglichkeiten für zukünftige Erweiterungen und Optimierungen. Ein zentrales Thema, das in zukünftigen Arbeiten weiter vertieft werden könnte, ist die Integration des „DevOps“-Konzepts. Das Prinzip entsprang aus der steigenden Kluft zwischen der Entwicklung und dem Betreiben von Software in großen Unternehmen. Es befasst sich mit dem Vereinheitlichen des Prozesses der Softwareentwicklung und dessen Auslieferung an dafür vorgesehene Umgebungen.[60] Die Arbeit beschränkte sich auf das Konzipieren reproduzierbarer Entwicklungs- und Pipeline-Umgebungen. Hierbei könnte die weitere Vereinheitlichung der ausführenden Deployment-Infrastruktur eine nahtlose Integration zwischen Entwicklung, Testing und Produktion ermöglichen. Dies würde nicht nur die lokale Entwicklungsumgebung und die Pipelines betreffen, sondern auch die Produktionsumgebung, wodurch die genutzte Infrastruktur des Projekts ganzheitlich abgedeckt würde.

Ein weiterer interessanter Ansatzpunkt ist die Evaluierung der CI-Strategie in anderen technologischen Kontexten oder für größere Shopware-Projekte. Dies würde wertvolle Erkenntnisse über die Skalierbarkeit und Anpassungsfähigkeit des erstellten Konzepts liefern. Shopware bietet die Möglichkeit, in einem Cluster betrieben zu werden.[61] Hierbei werden mehrere Instanzen der Software gleichzeitig im Produktionsbetrieb angewandt, oftmals an verschiedenen Orten, um eine erhöhte Verfügbarkeit zu bieten und die Zugriffsgeschwindigkeit für Nutzer über große Flächen sicherzustellen. In Zukunft könnte die Strategie dazu für das Ausführen verteilter Systeme untersucht und erweitert werden.

Insgesamt zeigt sich, dass die Evolution von Continuous-Software-Engineering in der modernen und agilen Softwareentwicklung von zentraler Bedeutung ist. Mit der stetigen Weiterentwicklung von Technologien und Methoden wird die Bedeutung einer robusten und flexiblen CI-Strategie in den kommenden Jahren weiter zunehmen, um den Anforderungen von Unternehmen und Entwicklern gerecht zu werden.

[60] Vgl. Fitzgerald und Stol. „Continuous software engineering: A roadmap and agenda“. 2017, S. 178.
[61] Vgl. Shopware AG. *Cluster Setup*. 2023.

A Anhang I: Übersicht der verwendeten CI-Tools

In Kapitel 4.2 wurden verschiedene CI-Tools und Services vorgestellt, welche für die Fallstudie zur Umsetzung der Strategie eingesetzt wurden. Im Folgenden wird eine Übersicht der genutzten Tools und dessen Funktionen und Merkmale gegeben:

Version-Control-System und Pipeline-Runner

- **GitLab**

  GitLab ist ein webbasierter Service für Versionierung mit dem VCS „Git“. Es bietet alle verteilten Versionskontroll- und Quellcodeverwaltungsfunktionen von Git sowie einige zusätzliche Features. Dies umfasst zum Beispiel eine Zugriffskontrolle und mehrere Kollaborationsfunktionen wie Aufgabenverwaltung, Fehlerverfolgung und Feature-Anfragen für Projekte sowie eine integrierte Wiki-Funktion für die Dokumentation. Der Service bietet außerdem CI-Tools und Pipeline-Verwaltung an. Dieser wird unter einer Open-Source-Lizenz vertrieben und dessen Self-Managed-Version ermöglicht es, GitLab auf eigenen Servern zu installieren und zu verwalten.[62]

- **GitLab CI/CD**

  GitLab CI/CD ist der Pipeline-Runner von GitLab und stellt eine eingebaute Pipeline-Lösung für das Ausführen von CI- und CD-Prozessen dar. Das Tool ist vollständig in GitLab integriert und verwendet eine Konfigurationsdatei innerhalb des Repositories, um die Phasen und Jobs der CI-Pipelines zu definieren. Es bietet eine breite Palette von Funktionen, einschließlich der Parallelisierung von Jobs, Pipelines für unterschiedliche Branches und das Verwalten von Umgebungsvariablen. GitLab CI/CD kann auf eigenen Servern oder in der Cloud ausgeführt werden, und sowohl für bei GitLab, als auch für extern gehostete Repositorys verwendet werden, was es zu einer flexiblen Lösung macht. Die für das Ausführen der Pipeline benötigten Umgebungsvariablen, Passwörter und andere vertrauliche Daten können dabei in einer von GitLab zur Verfügung gestellten, sicheren Datenbank verwaltet werden.[63]

Software-Testing

- **PHPUnit**

  PHPUnit ist ein weitverbreitetes Open-Source Unit-Testing-Framework, welches zum Erstellen von Unit-Tests in PHP-basierten Applikationen verwendet wird. Das Framework ermöglicht das Erstellen von Mocks und das Auswerten der Test-Coverage für das zu testende Projekt.[64]

- **Infection**

  Infection ist eine Mutation-Testing-Library für PHP. Sie ändert den Quellcode automatisch und führt dann bestehende Tests erneut aus, um zu sehen, ob diese fehlschlagen. Wenn

[62] Vgl. GitLab B.V. *The DevSecOps Platform | GitLab*. 2023.
[63] Vgl. GitLab B.V. *GitLab CI/CD | GitLab*. 2023.
[64] Vgl. Bergmann. *PHPUnit – The PHP Testing Framework*. 2023.

die Tests bestehen, obwohl der Code mutiert wurde, zeigt Infection diese Stellen im Code an, damit diese verbessert werden können. Die Library unterstützt die Integration mit PHPUnit und weiteren Testing-Tools.[65]

- **Jest**

  Jest ist ein Javascript Testing-Framework und ermöglicht das Einbringen von Unit-Tests in JavaScript-basierten Applikationen. Ähnlich wie PHPUnit unterstützt Jest das Erstellen von Mocks und bietet Funktionen zur Coverage an. Das Framework parallelisiert laufende Tests, um eine möglichst hohe Performance zu bieten.[66]

- **Cypress**

  Cypress ist ein Framework für Webapplikationen, um Functional-Tests durchzuführen. Es ermöglicht das automatisierte Testen von Benutzerinteraktionen innerhalb einer Webapplikation in einem realen Browser. Durch Cypress kann das Zusammenspiels des gesamten Projekts prüfen und zur Automatisierung von Testfällen genutzt werden, bei denen normalerweise Nutzer-Interaktion erforderlich ist.[67] Shopware bietet eine eigene Test-Suite für System-Tests durch Cypress an.[68]

Static-Code-Analysis

- **PHP_CodeSniffer**

  PHP_CodeSniffer (PHPCS) ist ein Set aus zwei Kommandozeilen-Befehlen für das Analysieren von PHP- und JavaScript-Code und von Cascading Style Sheets (CSS) nach einem gegebenen Coding-Standard und für das automatische Korrigieren von Abweichungen dieses Standards. Das Tool kann mit vorgegebenen und eigens erstellten oder angepassten Regelsätzen betrieben werden.[69]

- **PHP Mess Detector**

  PHP Mess Detector (PHPMD) ist eine Software, die nach vorgegebenen Problemen und Unstimmigkeiten in PHP-Code sucht. Das Tool kann zum Verhindern des Einführens von überflüssigen Variablen und Methoden, unoptimiertem Code und möglichen Fehlern in die Zielumgebung genutzt werden. PHPMD kann ergänzend zu Danger und PHPCS zur statischen Analyse von PHP-Code verwendet werden.[70]

- **PHPStan**

  PHPStan ist ein weiteres QA-Tool für das statische Analysieren von PHP-Code. Es konzentriert sich auf die Erkennung von Fehlern, die im laufenden Betrieb zu Problemen führen können, wie z. B. Aufrufe von nicht existierenden Methoden, ungenutzte Variablen etc. PHPStan kann in den Entwicklungsprozess integriert werden, um die Codequalität kontinuierlich zu verbessern.[71]

[65] Vgl. Rafalko. *Introduction - Infection PHP*. 2023.
[66] Vgl. Meta Platforms, Inc. *Jest · Delightful JavaScript Testing*. 2023.
[67] Vgl. Cypress.io, Inc. *JavaScript Component Testing and E2E Testing Framework | Cypress*. 2023.
[68] Vgl. Shopware. *E2E Platform Testsuite for Shopware 6*. 2023.
[69] Vgl. Squiz Labs. *PHP_ CodeSniffer*. 2023.
[70] Vgl. Pichler. *PHPMD - PHP Mess Detector*. 2023.
[71] Vgl. Mirtes. *Find Bugs Without Writing Tests | PHPStan*. 2023.

- **Deptrac**

  Deptrac ist ein statisches Code-Analyse-Tool, das dabei hilft, die Architektur eines PHP-Projekts zu verstehen und zu überprüfen. Es stellt sicher, dass die Abhängigkeiten zwischen den Modulen eines Projekts den definierten Architekturregeln entsprechen. Deptrac kann in den Entwicklungsprozess integriert werden, um die Architektur des Projekts kontinuierlich zu überwachen und zu verbessern.[72]

- **Eslint**

  ESLint ist ein statisches Tool zur Analyse von JavaScript-Code. Das Tool ist sehr anpassbar und kann für das Entwickeln mit verschiedenen Frameworks und Libraries genutzt werden. Viele Text- und Code-Editoren unterstützen ESLint und zeigen Analyse-Resultate bereits bei der Entwicklung im Editor an. Das Tool unterstützt außerdem das automatische Beheben von bestimmten Syntax-Fehlern mit einem eingebauten Kommandozeilenbefehl.[73]

- **Danger JS**

  Danger ist ein statisches Analyse-Tool, welches direkt in VCS angebunden werden kann. Das Tool läuft während des CI-Prozesses und kann zur Automatisierung von Code-Review-Aufgaben genutzt werden. Mit Danger können Warnungen und Fehler in der Pipeline geworfen werden, wenn geplante Integrationen zu groß werden, kein Changelog-Eintrag für Änderungen angelegt wurde, Abhängigkeiten nicht up to date sind und vieles mehr. Danger kann außerdem Job-Ergebnisse einer Pipeline in Form von Kommentaren in Merge-Requests hinterlegen, um Entwickler schneller über den Status des Projekts zu informieren. Die Software wird für die Programmiersprachen JavaScript, Ruby, Kotlin, Python und Swift angeboten.[74]

- **License Checker**

  License Checker sind Tools, die Lizenzen von Abhängigkeiten in einem Projekt überprüfen. Sie können dabei helfen, Lizenzprobleme bei Plugins und Erweiterungen von Drittanbietern zu identifizieren und zu vermeiden. Für das Prüfen von Composer- und NPM-Paketen existieren jeweils eigene PHP- und JavaScript-basierte Tools.[75, 76]

- **Security Checker / AuditJS**

  Security Checker sind Tools, die überprüfen, ob die in einem Projekt verwendeten Abhängigkeiten bekannte Sicherheitslücken aufweisen. Diese Tools können dabei helfen, Projekte kontinuierlich zu überwachen und sicher zu halten. Symfony bietet für das automatische Prüfen von Applikationen einen eigenen Security Checker.[77] Darüber hinaus wurde zur Prüfung von JavaScript-Abhängigkeiten in der Fallstudie das JavaScript-basierte Sicherheitsprüfungs-Tool AuditJS verwendet.[78]

[72] Vgl. QOSSMIC GmbH. *Deptrac*. 2023.
[73] Vgl. OpenJS Foundation. *Find and fix problems in your JavaScript code - ESLint - Pluggable JavaScript Linter*. 2023.
[74] Vgl. Therox und Danger JS contributors. *Danger JS*. 2023.
[75] Vgl. madewithlove BV. *CLI Licence checker for composer dependencies*. 2023.
[76] Vgl. Glass. *NPM License Checker*. 2023.
[77] Vgl. Symfony SAS. *Installing & Setting up the Symfony Framework - Checking Security Vulnerabilities*. 2023.
[78] Vgl. Sonatype Community. *AuditJS*. 2023.

Deployment

- **Deployer**

  Deployer ist ein PHP-basiertes Deployment-Tool, welches zur automatisierten Auslieferung von Software an verschiedene Umgebungen verwendet werden kann. Das Werkzeug bietet vordefinierte Setups für das Deployment verschiedener Services und weitere wichtige Features wie minimale Ausfallzeiten und die Möglichkeit, ausgelieferte Software auf einen früheren Stand zurückzurollen.[79] Shopware stellt eine Anleitung für das Einsetzen von Deployer in Projekten bereit, wobei auch die Nutzung mit GitLab CI/CD abgedeckt wird.[80]

[79] Vgl. Deployer. *Deployer - The deployment tool for PHP | Deployer*. 2023.
[80] Vgl. Shopware. *Deployment with Deployer*. 2023.

B Anhang II: Darstellung ausgeführter Pipelines

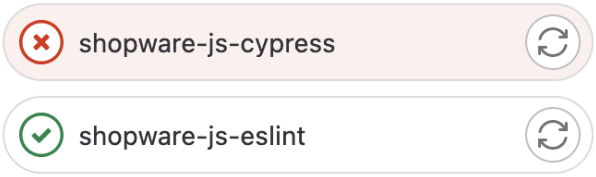
Der für die Umsetzung der Fallstudie gewählte Pipeline-Runner GitLab CI/CD bietet eine Übersicht aller Pipeline-Instanzen, die im Verlauf des Projekts durchgeführt wurden. Neben dem Titel und der Art des Ausgangs-Branches der Pipeline wird hierbei auch der Status der Pipeline aufgezeigt:

Quelle: GitLab B.V.

Abbildung 8: Übersicht ausgeführter Pipelines in GitLab CI/CD

Abbildung 8 stellt einen Ausschnitt der Pipeline-Übersicht dar. Es werden zwei verschiedene Pipelines angezeigt, wobei eine erfolgreich durchgeführt und die andere aufgrund von Fehlern in einem der Jobs abgebrochen wurde. Die fehlgeschlagene Pipeline wird dabei durch ein rotes X mit dem Schriftzug „failed“ markiert, während die erfolgreich durchgelaufene Pipeline einen grünen Haken und das Wort „passed“ ausgibt.

Navigiert man in der Pipeline-Auflistung von GitLab CI/CD zu der Detail-Ansicht einer der gezeigten Pipelines, werden dessen Phasen und Jobs dargestellt. Die hierbei angezeigten Jobs können entweder als erfolgreich oder fehlerhaft markiert werden:



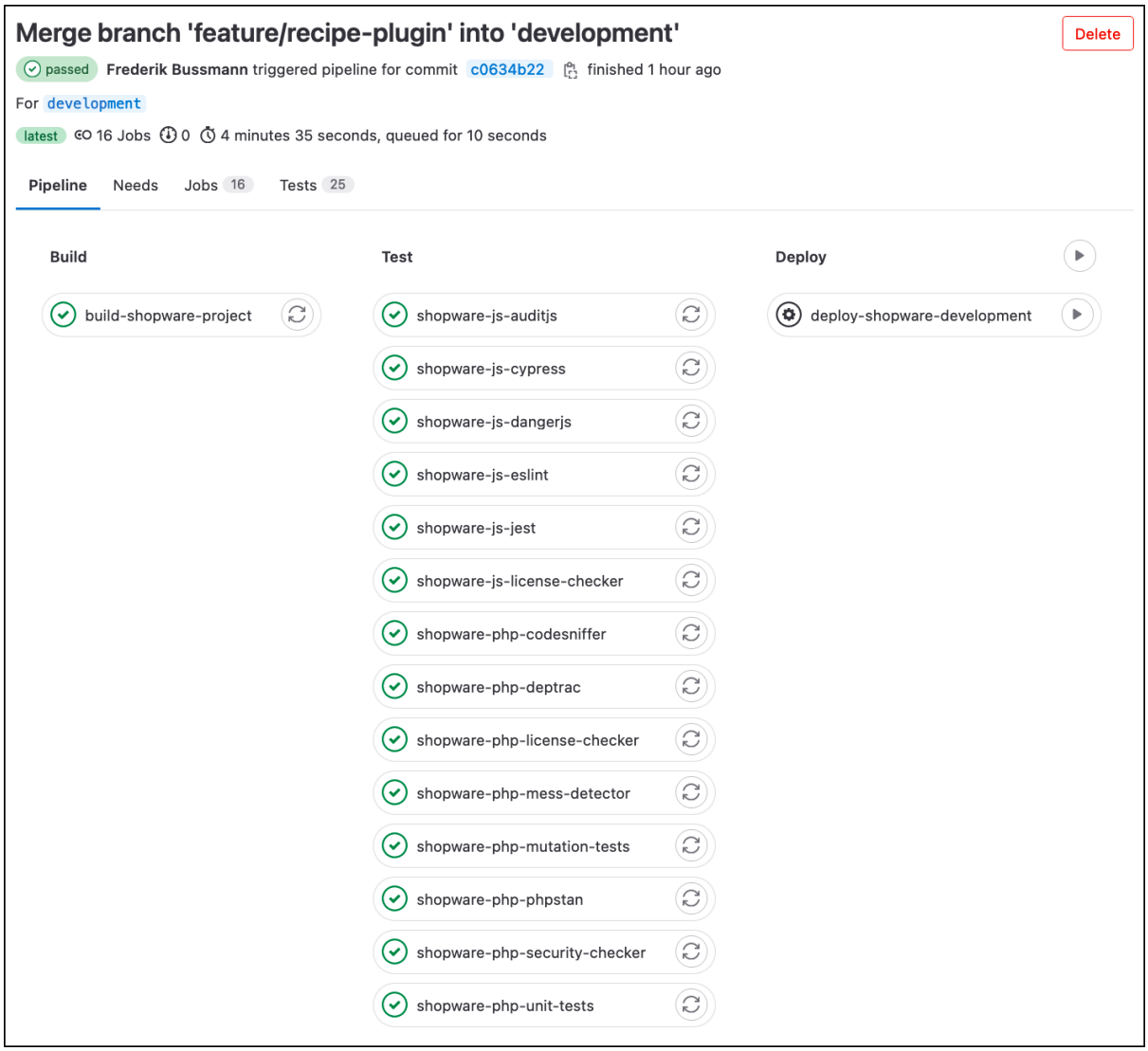
Quelle: GitLab B.V.

Abbildung 9: Status-Anzeige der Jobs einer Pipeline

In Abbildung 9 werden verschiedene Jobs aufgezeigt, welche jeweils einen unterschiedlichen Status berichten. Während der Job für das statische Code-Analyse-Tool Eslint erfolgreich durchgelaufen ist, wurde der Job zur Durchführung von System-Tests durch Cypress abgebrochen. Der erfolgreiche Job wird hierbei mit einem grünen Haken markiert, während der abgebrochene Job mit einem roten X gekennzeichnet wurde.

Neben den Job-Informationen werden in der Pipeline-Detail-Ansicht noch weitere Metriken wie die Anzahl der in den Jobs durchgeführten Tests oder die Dauer der gesamten Pipeline

ausgegeben. Eine Übersicht aller Jobs einer ausgewählten Pipeline wird in Abbildung 10 angezeigt. Diese implementiert die in Kapitel 4.2 ausgewählten Tools der CI-Strategie, welche nach dem erfolgreichen Build-Job in der Testing-Phase der Pipeline stattfinden. Jobs können in dieser Detail-Ansicht wiederholt und manche manuell angestoßen werden, wie zum Beispiel der Deployment-Job für die Development-Umgebung, welcher in der Deployment-Phase der Pipeline ausführbar ist. Dieser kann von den Entwicklern auf der Web-Oberfläche gestartet werden, um die gebaute und getestete Integration an die hinterlegte Umgebung auszuliefern.



Quelle: GitLab B.V.

Abbildung 10: Detail-Ansicht einer durchgeführten Pipeline

Literaturverzeichnis

[1] eCommerceDB. *The Most Commonly Used Shop Software Among Online Shops in Germany - Shopware and Salesforce Share Rank No. 1.* eCommerceDB GmbH. 2023. URL: https://ecommercedb.com/insights/the-most-commonly-used-shop-softwares-among-online-shops-in-germany-shopware-and-salesforce-share-rank-no-1/4210 (aufgerufen am 22. 06. 2023).

[2] 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. DOI: 10.1109/ACCESS.2017.2685629.

[3] Brian Fitzgerald und Klaas-Jan Stol. „Continuous software engineering: A roadmap and agenda". In: *The Journal of Systems and Software* 123 (2017), S. 176–189. DOI: 10.1016/j.jss.2015.06.063.

[4] Kent Beck, Mike Beedle, Arie van Bennekum et al. *Manifesto for Agile Software Development.* 2001. URL: https://agilemanifesto.org/ (aufgerufen am 30. 06. 2023).

[5] Lucas Gren und Per Lenberg. „Agility is responsiveness to change". In: *Proceedings of the Evaluation and Assessment in Software Engineering.* ACM, Apr. 2020. DOI: 10.1145/3383219.3383265.

[6] Martin Fowler. *Continuous Integration.* Fowler, Martin. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (aufgerufen am 26. 06. 2023).

[7] Jez Humble und David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN: 978-0-321-67022-9.

[8] Grady Booch. *Object oriented design with applications.* 1. Aufl. Benjamin/Cummings Pub. Co, 1991. ISBN: 978-0-805-30091-8.

[9] Kent Beck. „Extreme programming: A humanistic discipline of software development". In: *Fundamental Approaches to Software Engineering.* Springer Berlin Heidelberg, 1998. DOI: 10.1007/bfb0053579.

[10] Omar Elazhary, Colin Werner, Ze Shi Li et al. „Uncovering the Benefits and Challenges of Continuous Integration Practices". In: *IEEE Transactions on Software Engineering* 48.7 (Juli 2022), S. 2570–2583. DOI: 10.1109/tse.2021.3064953.

[11] Michael Hilton, Timothy Tunnell, Kai Huang et al. „Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ASE '16. Singapore: Association for Computing Machinery, 2016, S. 426–437. ISBN: 978-1-450-33845-5. DOI: 10.1145/2970276.2970358.

[12] Sheikh Fahad Ahmad, Mohd Haleem und Mohd Beg. „Test Driven Development with Continuous Integration: A Literature Review". In: *International Journal of Computer Applications Technology and Research* 2 (Mai 2013), S. 281–285. DOI: 10.7753/IJCATR0203.1013.

[13] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto et al. „How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines“. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, S. 334–344. DOI: 10.1109/MSR.2017.2.

[14] Paul M. Duvall, Steve Matyas und Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. 1. Aufl. Addison-Wesley Signature Series. Pearson Education, 2007. ISBN: 978-0-321-63014-8.

[15] Omar Elazhary, Margaret-Anne Storey, Neil A. Ernst et al. „ADEPT: A Socio-Technical Theory of Continuous Integration“. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 2021, S. 26–30. DOI: 10.1109/ICSE-NIER52604.2021.00014.

[16] Theo Combe, Antony Martin und Roberto Di Pietro. „To Docker or Not to Docker: A Security Perspective“. In: *IEEE Cloud Computing* 3.5 (2016), S. 54–62. DOI: 10.1109/MCC.2016.100.

[17] Cristian Constantin Spoiala, Alin Calinciuc, Corneliu Octavian Turcu et al. „Performance comparison of a WebRTC server on Docker versus virtual machine“. In: *2016 International Conference on Development and Application Systems (DAS)*. 2016, S. 295–298. DOI: 10.1109/DAAS.2016.7492590.

[18] Eliane Collins, Arilo Dias-Neto und Vicente F. de Lucena Jr. „Strategies for Agile Software Testing Automation: An Industrial Experience“. In: *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. 2012, S. 440–445. DOI: 10.1109/COMPSACW.2012.84.

[19] Paul Ammann und Jeff Offutt. *Introduction to Software Testing*. 2. Aufl. Cambridge University Press, 2016. DOI: 10.1017/9781316771273.

[20] Ira D. Baxter, Andrew Yahin, Leonardo. Moura et al. „Clone detection using abstract syntax trees“. In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 1998, S. 368–377. DOI: 10.1109/ICSM.1998.738528.

[21] Shopware. *The story behind Shopware AG*. Shopware AG. 2023. URL: https://www.shopware.com/en/company/story/ (aufgerufen am 25. 06. 2023).

[22] Symfony SAS. *Projects using Symfony - Popular PHP projects using Symfony components or based on the Symfony framework*. Symfony SAS. 2023. URL: https://symfony.com/projects (aufgerufen am 24. 06. 2023).

[23] Shopware AG. *Cluster Setup*. 2023. URL: https://developer.shopware.com/docs/guides/hosting/installation-updates/cluster-setup (aufgerufen am 05. 08. 2023).

[24] Gunnard Engebreth und Satej Kumar Sahu. „Introduction to Symfony“. In: *PHP 8 Basics: For Programming and Web Development*. Berkeley, CA: Apress, 2023, S. 273–283. ISBN: 978-1-4842-8082-9. DOI: 10.1007/978-1-4842-8082-9_15.

[25] Shopware. *Einblicke in die Core Architektur von Shopware 6*. Shopware AG. 2019. URL: https://www.shopware.com/de/news/einblicke-in-die-core-architektur-von-shopware-6/ (aufgerufen am 03. 07. 2023).

[26] Pickware GmbH. *Shopware 6 – Ein Blick auf die neue Architektur*. Pickware GmbH. 2019. URL: https://www.pickware.com/de/blog/shopware-6-neue-architektur (aufgerufen am 03. 07. 2023).

[27] Shopware. *Plugins for Symfony developers*. Shopware AG. 2023. URL: https://developer.shopware.com/docs/guides/plugins/plugins/plugins-for-symfony-developers (aufgerufen am 04. 07. 2023).

[28] Shopware. *Requirements*. Shopware AG. 2023. URL: https://developer.shopware.com/docs/guides/installation/requirements (aufgerufen am 06. 07. 2023).

[29] Shopware. *shopware/development - Docker Image*. Shopware AG. 2023. URL: https://hub.docker.com/r/shopware/development (aufgerufen am 12. 07. 2023).

[30] Docker Inc. *Docker Compose overview*. 2023. URL: https://docs.docker.com/compose/ (aufgerufen am 24. 08. 2023).

[31] Vincent Driessen. *A successful Git branching model*. 2010. URL: https://nvie.com/posts/a-successful-git-branching-model/ (aufgerufen am 12. 08. 2023).

[32] GitLab B.V. *The DevSecOps Platform | GitLab*. 2023. URL: https://about.gitlab.com/ (aufgerufen am 06. 07. 2023).

[33] GitLab B.V. *GitLab CI/CD | GitLab*. 2023. URL: https://docs.gitlab.com/ee/ci/ (aufgerufen am 07. 07. 2023).

[34] Sebastian Bergmann. *PHPUnit – The PHP Testing Framework*. 2023. URL: https://phpunit.de/ (aufgerufen am 08. 07. 2023).

[35] Maks Rafalko. *Introduction - Infection PHP*. 2023. URL: https://infection.github.io/guide/ (aufgerufen am 08. 07. 2023).

[36] Meta Platforms, Inc. *Jest · Delightful JavaScript Testing*. 2023. URL: https://jestjs.io/ (aufgerufen am 08. 07. 2023).

[37] Cypress.io, Inc. *JavaScript Component Testing and E2E Testing Framework | Cypress*. 2023. URL: https://www.cypress.io/ (aufgerufen am 08. 07. 2023).

[38] Shopware. *E2E Platform Testsuite for Shopware 6*. Shopware AG. 2023. URL: https://github.com/shopware/e2e-testsuite-platform (aufgerufen am 19. 07. 2023).

[39] Squiz Labs. *PHP_CodeSniffer*. 2023. URL: https://github.com/squizlabs/PHP_CodeSniffer (aufgerufen am 09. 07. 2023).

[40] Manuel Pichler. *PHPMD - PHP Mess Detector*. 2023. URL: https://phpmd.org/ (aufgerufen am 09. 07. 2023).

[41] Ondřej Mirtes. *Find Bugs Without Writing Tests | PHPStan*. 2023. URL: https://phpstan.org/ (aufgerufen am 10. 07. 2023).

[42] QOSSMIC GmbH. *Deptrac*. 2023. URL: https://github.com/qossmic/deptrac (aufgerufen am 10. 07. 2023).

[43] OpenJS Foundation. *Find and fix problems in your JavaScript code - ESLint - Pluggable JavaScript Linter*. 2023. URL: https://eslint.org/ (aufgerufen am 09. 07. 2023).

[44] Orta Therox und Danger JS contributors. *Danger JS*. 2023. URL: https://danger.systems/js/ (aufgerufen am 09. 07. 2023).

[45] madewithlove BV. *CLI Licence checker for composer dependencies*. 2023. URL: https://github.com/madewithlove/license-checker-php (aufgerufen am 10. 07. 2023).

[46] Dav Glass. *NPM License Checker*. 2023. URL: https://www.npmjs.com/package/license-checker (aufgerufen am 12. 07. 2023).

[47] Symfony SAS. *Installing & Setting up the Symfony Framework - Checking Security Vulnerabilities*. 2023. URL: https://symfony.com/doc/current/setup.html#checking-security-vulnerabilities (aufgerufen am 10. 07. 2023).

[48] Sonatype Community. *AuditJS*. 2023. URL: https://github.com/sonatype-nexus-community/auditjs (aufgerufen am 12. 07. 2023).

[49] Deployer. *Deployer - The deployment tool for PHP | Deployer*. 2023. URL: https://deployer.org/ (aufgerufen am 15. 07. 2023).

[50] Shopware. *Deployment with Deployer*. Shopware AG. 2023. URL: https://developer.shopware.com/docs/guides/hosting/installation-updates/deployments/deployment-with-deployer (aufgerufen am 15. 07. 2023).