HOCHSCHULE MITTWEIDA
University of Applied Sciences

MASTERARBEIT

Herr
Jan Schneider, B.Sc.

Konzeption und Umsetzung eines automatisierten Build- und Deployment-Systems für eine Java-Desktop-Anwendung

Mittweida, September 2024

Fakultät **Angewandte Computer- und Biowissenschaften**

MASTERARBEIT

Konzeption und Umsetzung eines automatisierten Build- und Deployment-Systems für eine Java-Desktop-Anwendung

Autor:
Jan Schneider

Studiengang:
Cybercrime/Cybersecurity

Seminargruppe:
CY22wC-M

Erstprüfer:
Prof. Dr. rer. pol. Dirk Pawlaszczyk

Zweitprüfer:
Rico Herlt, M.Sc.

Einreichung:
Mittweida, 27.09.2024

Verteidigung/Bewertung:
Mittweida, 2024

Faculty of **Applied Computer Sciences and Biosciences**

MASTER THESIS

Design and implementation of an automated build and deployment system for a Java desktop application

Author:
Jan Schneider

Course of Study:
Cybercrime/Cybersecurity

Seminar Group:
CY22wC-M

First Examiner:
Prof. Dr. rer. pol. Dirk Pawlaszczyk

Second Examiner:
Rico Herlt, M.Sc.

Submission:
Mittweida, 27.09.2024

Defense/Evaluation:
Mittweida, 2024

Bibliografische Beschreibung

Schneider, Jan:
Konzeption und Umsetzung eines automatisierten Build- und Deployment-Systems für eine Java-Desktop-Anwendung. – 2024. – 50 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften, Masterarbeit, 2024.

Referat

Ziel dieser Arbeit ist es, den Build- und Bereitstellungsprozess einer bestehenden Java-Anwendung durch den Einsatz von Gradle und GitHub Actions zu automatisieren. Im Fokus steht die Implementierung eines plattformübergreifenden Build-Prozesses, der es ermöglicht, Installationspakete für Windows, macOS und Linux zu erstellen. Zusätzlich sollen Pakete für die gängigen Paketmanager Chocolatey und Homebrew bereitgestellt werden, um die Installation zu erleichtern. Ein weiterer Schwerpunkt liegt auf der Gestaltung einer sicheren CI/CD-Pipeline gemäß den OWASP Best Practices, um Sicherheitsrisiken während des Build-Prozesses zu minimieren.

Abstract

The aim of this thesis is to automate the build and deployment process of an existing Java application by using Gradle and GitHub Actions. The focus lies on implementing a cross-platform build process that makes it possible to create installation packages for Windows, macOS and Linux. In addition, packages for the common package managers Chocolatey and Homebrew will be provided to make installation easier. Another focus lies on designing a secure CI/CD pipeline according to OWASP best practices to minimize security risks during the build process.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CD ..... Continuous Deployment

CI ..... Continuous Integration

CI/CD ..... Continuous Integration/Continuous Deployment

CLI ..... Command-Line Interface

DSL ..... Domain Specific Language

JDK ..... Java Development Kit

JRE ..... Java Runtime Environment

JVM ..... Java Virtual Machine

PPE ..... Poisoned Pipeline Execution

SCM ..... Source Code Management

SLSA ..... Supply Chain Levels for Software Artifacts

VCS ..... Version Control System

1 Einleitung

Die Entwicklung von Software ist sehr teuer; so stiegen die stündlichen Kosten für einen IT-Freelancer in Deutschland von 79€ in 2013 auf über 89€ in 2017 [1]. Dabei entstehen die Kosten nicht nur bei der Entwicklung von Software, sondern auch bei der Bereitstellung. Sei es eine Webseite, welche auf einem Server gehostet oder eine App, welche im Store veröffentlicht wird – es entsteht zusätzliche Arbeit, welche mitunter preisintensiv werden kann. In diesem Zusammenhang spielen Build-Systeme eine zentrale Rolle. Sie sind dafür verantwortlich, den Build-Prozess einer Software zu steuern und zu beschleunigen, indem sie die notwendigen Schritte zur Kompilierung, zum Testen und zur Paketierung automatisieren. Im Java-Ökosystem haben sich Gradle und Apache Maven als die dominierenden Build-Systeme etabliert. Neben den traditionellen Build-Systemen haben sich in den letzten Jahren auch neuere Werkzeuge wie Bazel und Buck2 etabliert, die eine plattformübergreifende Unterstützung bieten und von Technologie-Giganten wie Google und Meta entwickelt wurden. Ein gut gewähltes Build-System sorgt dafür, dass diese Prozesse konsistent und wiederholbar sind, was die Qualität und Zuverlässigkeit des Endprodukts erheblich steigert.

Ein weiterer Aspekt, der in modernen Entwicklungsprozessen immer mehr an Bedeutung gewinnt, ist die Sicherheit der Software und ihrer Abhängigkeiten. Um sicherzustellen, dass nur geprüfte und vertrauenswürdige Pakete in einer Anwendung verwendet werden, ist eine präzise Verwaltung der Abhängigkeiten unerlässlich.

Außerdem relevant für die moderne Softwareentwicklung ist die plattformübergreifende Kompilierung und Bereitstellung von Anwendungen. Während die Entwicklung einer Anwendung häufig auf einem spezifischen Betriebssystem erfolgt, muss die fertige Software oft auf verschiedenen Plattformen wie Windows, macOS und Linux lauffähig sein. Jede dieser Plattformen hat ihre eigenen Anforderungen und Besonderheiten, die berücksichtigt werden müssen, um sicherzustellen, dass die Software reibungslos funktioniert.

Die manuelle Kompilierung und Bereitstellung auf mehreren Systemen ist jedoch ein zeitaufwändiger und fehleranfälliger Prozess. In vielen Fällen ist es nicht möglich, auf einem einzigen System die Anwendung für verschiedene Zielplattformen zu erstellen, was zusätzliche Herausforderungen mit sich bringt. Entwickler müssen entweder mehrere Maschinen betreiben oder auf virtuelle Maschinen zurückgreifen, um die Software für unterschiedliche Umgebungen zu bauen. Dies erhöht nicht nur den Arbeitsaufwand, sondern auch die Komplexität des gesamten Entwicklungsprozesses.

Plattformübergreifende Build-Systeme und CI/CD-Pipelines bieten hier eine Lösung, indem sie die Kompilierung und Bereitstellung für mehrere Plattformen automatisieren und optimieren. Diese Systeme können so konfiguriert werden, dass sie den Code automatisch für die gewünschten Zielplattformen bauen und die resultierenden Artefakte bereitstellen. Dadurch wird der Aufwand für die plattformübergreifende Kompilierung erheblich reduziert, was nicht nur Zeit spart, sondern auch die Konsistenz und Zuverlässigkeit der veröffentlichten Software erhöht. In dieser Arbeit wird untersucht, wie solche Systeme effektiv eingesetzt werden können, um die Herausforderungen der plattformübergreifenden Entwicklung zu bewältigen und eine effiziente Bereitstellung der Software zu gewährleisten.

1.1 Zielstellung

Ziel dieser Arbeit ist die Entwicklung einer kontinuierlichen Integrations- und Bereitstellungspipeline (CI/CD) zur automatisierten Bereitstellung einer Java-Desktop-Anwendung nach einer Änderung im Quellcode. Diese Pipeline soll sicherstellen, dass jede Änderung im Quellcode automatisch zu einer neuen, funktionalen und plattformübergreifenden Version der Anwendung führt, die bereitgestellt werden kann.

Die Arbeit umfasst die Nutzung einer CI/CD-Plattform zur Automatisierung dieses Prozesses. Dabei soll die Pipeline so gestaltet werden, dass sie alle notwendigen Schritte von der Quellcodeverwaltung über die Kompilierung bis hin zur Bereitstellung der Anwendung abdeckt. Dies beinhaltet die Konfiguration und Integration verschiedener Tools und Technologien, um eine effiziente und zuverlässige Automatisierung zu gewährleisten.

Konkret werden folgende Ziele verfolgt:

- **Automatisierte Kompilierung**: Der Quellcode der Java-Desktop-Anwendung soll nach jeder Änderung automatisch kompiliert werden. Dabei wird sichergestellt, dass die Anwendung stets auf dem neuesten Stand ist und alle Änderungen korrekt in die Anwendung integriert werden.
- **Plattformübergreifende Bereitstellung**: Die Pipeline soll in der Lage sein, plattformspezifische Abbilder der Anwendung für Windows, Linux und macOS zu erstellen. Dies gewährleistet, dass die Anwendung auf allen gängigen Betriebssystemen verfügbar ist.
- **Einsatz von CI/CD-Best Practices**: Die Arbeit soll bewährte Praktiken der CI/CD implementieren, um die Zuverlässigkeit und Sicherheit der Pipeline zu maximieren. Dazu gehört die Nutzung von Branch-Protection-Regeln, die Durchführung automatisierter Tests und die Implementierung von Sicherheitsmechanismen.
- **Effiziente und sichere Bereitstellung**: Die Pipeline soll die Anwendung nach erfolgreicher Kompilierung und Testdurchführung automatisiert bereitstellen. Dabei wird besonders Wert auf die Sicherheit und Integrität der bereitgestellten Anwendung gelegt, um unbefugte Änderungen und potenzielle Sicherheitslücken zu verhindern.
- **Dokumentation und Nachvollziehbarkeit**: Alle Schritte und Konfigurationen der Pipeline sollen umfassend dokumentiert werden. Dies gewährleistet, dass der Prozess nachvollziehbar und reproduzierbar ist und bei Bedarf angepasst werden kann.

Die erfolgreiche Umsetzung dieser Ziele soll sicherstellen, dass die Java-Desktop-Anwendung kontinuierlich und zuverlässig aktualisiert und bereitgestellt wird, wodurch die Effizienz und Produktivität im Entwicklungsprozess erheblich gesteigert werden.

1.2 Die Anwendung *FQLite*

Die Open-Source-Software *FQLite* (`https://github.com/pawlaszczyk/fqlite`) von Prof. Dr. rer. pol. Dirk Pawlaszczyk wird auf *GitHub* bereitgestellt. Diese Software „ist ein Werkzeug zum Auffinden und Wiederherstellen gelöschter Datensätze in SQLite-Datenbanken“ [2]. Der Quellcode dieser Anwendung wurde in der Programmiersprache Java (siehe Kapitel 2.3) geschrieben, und diese kann sowohl als Kommandozeilenanwendung als auch mit einer grafischen Benutzeroberfläche genutzt werden [3]. Ein Ausschnitt der grafischen Benutzeroberfläche ist in Abbildung 1.1 abgebildet.

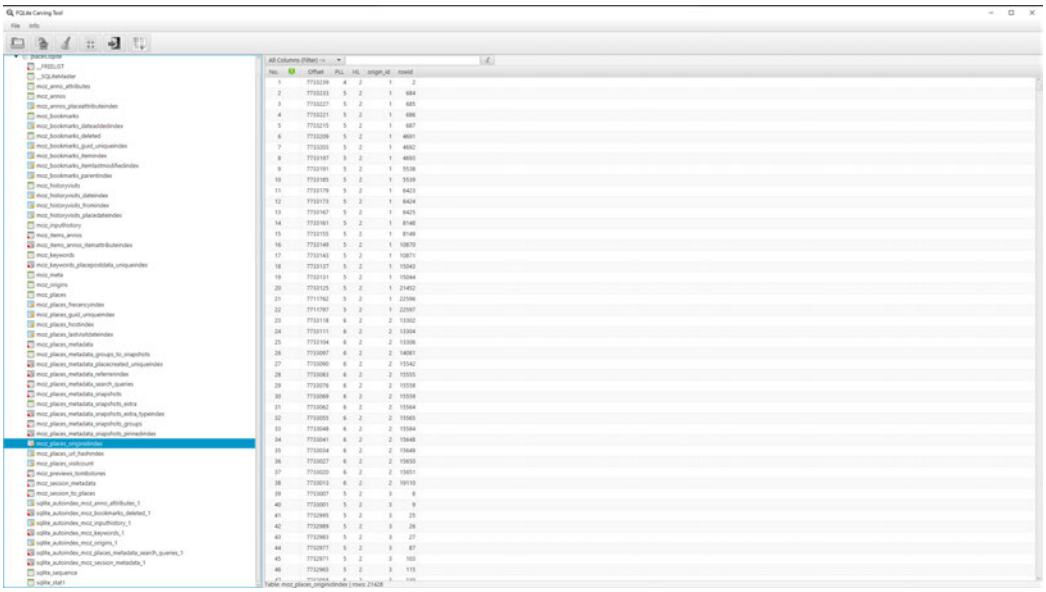

Abbildung 1.1: FQLite Benutzeroberfläche

Gelöschte Daten in SQLite-Datenbanken können häufig wiederhergestellt werden, da diese Daten oft noch nicht überschrieben wurden. In SQLite werden Daten lediglich als gelöscht markiert und als freier Speicher bereitgestellt, um durch neue Daten überschrieben zu werden. Daher ist es in vielen Fällen möglich, diese gelöschten Daten wiederherzustellen. Selbst wenn die gelöschten Einträge teilweise überschrieben wurden, lassen sich oft noch Artefakte dieser Daten rekonstruieren. [4]

Zu Beginn dieser Arbeit wurde die Anwendung in der Version 2.6.5 auf *GitHub* veröffentlicht. Um die erforderlichen Umstrukturierungen und Quellcodeänderungen vorzunehmen, wurde ein Fork-Repository erstellt. Dieses Fork-Repository stellt eine Kopie des ursprünglichen *Git*-Verlaufs dar und ermöglicht es, Änderungen unabhängig vom Hauptprojekt durchzuführen. Dieses Fork-Repository kann unter `https://github.com/bocian67/fqlite` eingesehen werden. Durch diese Vorgehensweise können die vorgenommenen Anpassungen später unter Berücksichtigung der zwischenzeitlich vom ursprünglichen Autor vorgenommenen Änderungen in das Haupt-Repository integriert werden.

1.3 Rahmenbedingungen

Ziel dieser Arbeit ist die Implementierung einer automatisierten CI/CD-Pipeline für den Bau und die Bereitstellung von der Open-Source-Anwendung *FQLite*. Diese Automatisierung soll mithilfe von *GitHub Actions* realisiert werden, um eine kontinuierliche und zuverlässige Bereitstellung der Software zu gewährleisten. Der gesamte Prozess umfasst mehrere wesentliche Schritte:

- **Quellcodeverwaltung**: *FQLite* wird auf *GitHub* gehostet, was die zentrale Plattform für die Quellcodeverwaltung darstellt. Änderungen und neue Features werden über Pull-Requests in das Repository eingebracht und versioniert.
- **Automatisierter Build-Prozess**: Durch *GitHub Actions* wird ein Workflow definiert, der bei Änderungen im Quellcode ausgelöst wird. Dieser Workflow kompiliert die Software automatisch und stellt sicher, dass die neuesten Änderungen korrekt integriert und fehlerfrei sind.

- **Plattformspezifische Abbilder**: Die Pipeline wird so konfiguriert, dass sie plattformspezifische Installationspakete für Windows, Linux und macOS erstellt. Dies gewährleistet, dass die Software auf allen gängigen Betriebssystemen verfügbar ist und optimal funktioniert.
- **Bereitstellung der Software**: Nach erfolgreicher Kompilierung und Paketierung wird die Software automatisiert auf einem FTP-Server bereitgestellt. Dies ermöglicht Nutzern den einfachen Download der neuesten Version von *FQLite*.
- **Sicherheits- und Integritätsmaßnahmen**: Um die Sicherheit und Integrität der bereitgestellten Software zu gewährleisten, werden verschiedene Sicherheitsmechanismen implementiert. Dazu gehören signierte Commits, automatisierte Tests und die Überprüfung der Integrität der erzeugten Artefakte.
- **Automatisierte Abhängigkeitsverwaltung**: Um die Sicherheit und Stabilität der Anwendung sicherzustellen und gleichzeitig eine hohe Flexibilität zu gewährleisten, wird das Build-System so konfiguriert, dass es automatisch alle benötigten Abhängigkeiten bezieht und verwaltet. Dadurch wird sichergestellt, dass stets die korrekten und geprüften Versionen der Abhängigkeiten verwendet werden, ohne manuelle Eingriffe erforderlich zu machen.

Die Implementierung dieser CI/CD-Pipeline erfordert eine sorgfältige Planung und Konfiguration des *GitHub Action* Workflows. Besonderer Fokus wird dabei auf die Zuverlässigkeit und Sicherheit des gesamten Prozesses gelegt. Durch die Automatisierung soll die Effizienz des Entwicklungsprozesses erhöht und die Qualität der Software kontinuierlich sichergestellt werden. Die Bereitstellung über einen FTP-Server ermöglicht zudem eine einfache Verteilung der Software an die Endnutzer.

2 Grundlagen

Nach der Entwicklung von Software werden üblicherweise umfassende Tests zur Qualitätssicherung durchgeführt. Diese Tests dienen dazu, die Funktionalität der Anwendung sicherzustellen und zu überprüfen, ob sie den vorgesehenen Anforderungen entspricht. Wenn Tests fehlschlagen, wird der Entwicklungsprozess wiederholt, bis die definierten Tests erfolgreich absolviert wurden. Nach erfolgreichem Bestehen der Tests können die neuen Quellcodeänderungen in das zentrale Quellcode-Repository integriert und anschließend kompiliert werden. Das resultierende Softwarepaket muss dann bereitgestellt werden, indem es entweder auf einem Server für die Anwender zugänglich gemacht wird, sodass diese oder die Anwendung selbst ein Update durchführen können, oder indem es als Webanwendung direkt auf dem Server für alle Nutzer installiert wird. [5, S. 2f.]

In der modernen Softwareentwicklung spielen Continuous Integration (CI) und Continuous Deployment (CD) daher eine zentrale Rolle, da sie den Entwicklungs- und Veröffentlichungsprozess erheblich optimieren und beschleunigen. CI bezieht sich auf die kontinuierliche Integration von Codeänderungen, begleitet von automatisierten Tests, in ein gemeinsames Repository. Dies soll sicherstellen, dass keine neuen Fehler eingeführt werden und der Code konsistent und fehlerfrei bleibt. Durch CI wird die Zusammenarbeit im Entwicklungsteam verbessert, da Integrationsprobleme und Konflikte frühzeitig erkannt und behoben werden können.
CD baut auf CI auf und automatisiert den Prozess der Bereitstellung der integrierten und getesteten Software in produktionsähnliche Umgebungen. Dies ermöglicht es, Softwareänderungen schnell und zuverlässig zu veröffentlichen und die Zeit von der Konzeption bis zur Markteinführung erheblich zu verkürzen. *Continuous Deployment*, eine Erweiterung von *Continuous Delivery*, geht noch einen Schritt weiter, indem es die Freigabe der Software in die Produktionsumgebung automatisiert, sobald alle Tests erfolgreich bestanden wurden. [6]

Die Implementierung von CI/CD-Pipelines bietet zahlreiche Vorteile, darunter eine verbesserte Softwarequalität, kürzere Veröffentlichungszyklen und eine höhere Reaktionsfähigkeit auf Änderungen und Fehler. In diesem Kapitel werden die grundlegenden Konzepte von CI und CD ausführlich erläutert, um ein tiefgehendes Verständnis der Methoden und Techniken zu vermitteln, die für die Implementierung und den Betrieb von CI/CD-Systemen erforderlich sind. Zudem wird Java mit dem Build-System Gradle näher beschrieben, welches für die Bereitstellung der Anwendung notwendig ist.

2.1 Versionskontrollsysteme

Für die effektive Zusammenarbeit während der Projektentwicklung ist es unerlässlich, stets den aktuellen Entwicklungsstand zu nutzen und diesen effizient mit anderen Teammitgliedern zu teilen. Zur Lösung dieses Problems wurden Versionskontrollsysteme (Version Control System (VCS)) oder Quellcodemanagementsysteme (Source Code Management (SCM)) eingeführt [7]. Diese Systeme speichern Änderungen in einzelnen Revisionen, wodurch es möglich ist, den Verlauf der Änderungen nachzuvollziehen und bei Bedarf zu früheren Revisionen zurückzukehren. Darüber hinaus ermöglichen sie es, parallel an verschiedenen Entwicklungsständen (*Branches*) zu arbeiten und diese später zusammenzuführen. [8]

Es existieren zwei Hauptansätze für Versionskontrollsysteme: zentralisierte Versionskontrollsysteme (Centralized VCS, CVCS) und dezentralisierte Versionskontrollsysteme (Distributed VCS, DVCS). Bei zentralisierten Systemen muss eine Verbindung zu einem zentralen Server bestehen, auf dem das Repository (Quellcodeprojekt) gespeichert ist. Dieser Ansatz wird beispielsweise in *Subversion* oder dem *Perforce Revision Control System* verwendet. Im Gegensatz dazu legen dezentrale Systeme bei jedem Nutzer eine vollständige Kopie des Repositories lokal ab, was es ermöglicht, auch ohne Serververbindung zu arbeiten. Eine Verbindung zum Server ist lediglich notwendig, um den eigenen Stand mit dem des Servers zu synchronisieren und somit den aktuellen Stand für alle Teammitglieder verfügbar zu machen. Beispiele für dezentrale Systeme sind *Mercurial*, *BitKeeper* und *Git*. Insbesondere bei Open-Source-Projekten wird häufig der dezentrale Ansatz bevorzugt, da jede lokale Kopie des Repositories das Risiko eines Datenverlusts bei einem Fehler oder Systemausfall verringert [7].

Die Nutzung von Versionskontrollsystemen bietet mehrere Vorteile. Einerseits ermöglicht die detaillierte Nachverfolgung einzelner Änderungen (*Commits*) die frühzeitige Erkennung und Korrektur unerwünschter Änderungen bereits bei der Zusammenführung (*Merge*) mit dem aktuellen Entwicklungsstand. Andererseits unterstützt die Verzweigungsfunktion (*Branching*) die parallele Entwicklung, indem verschiedene Entwickler gleichzeitig Änderungen vornehmen können, ohne sich gegenseitig zu behindern. Sobald ein Entwickler seine Aufgabe abgeschlossen hat, kann er seinen Zweig (*Branch*) mit dem Hauptentwicklungsstand zusammenführen (integrieren), sodass die Änderungen auch für andere Entwickler lokal verfügbar sind. Eine Darstellung dieses Prozesses ist in Abbildung 2.1 zu sehen [8].

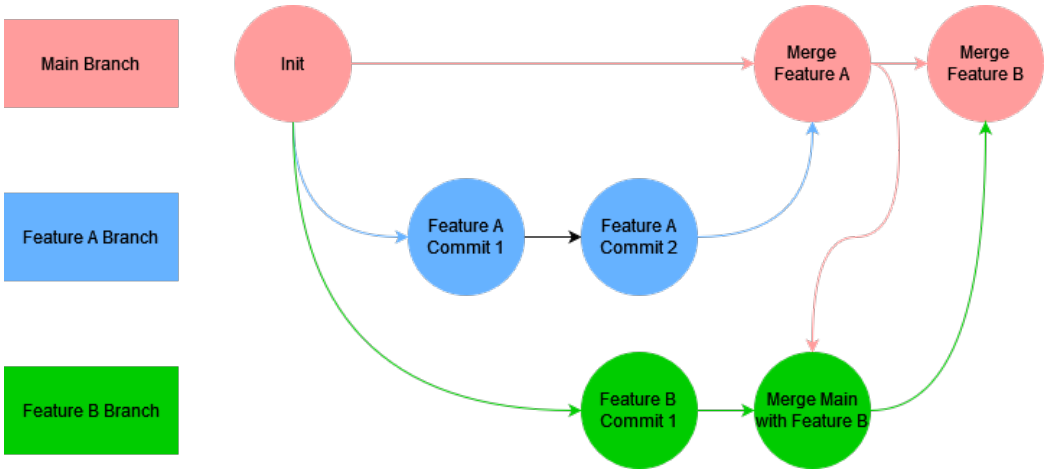

Abbildung 2.1: Git Ablauf

Entwickler verwenden häufig zusätzlich zu einem Versionskontrollsystem eine Plattform, die als zentraler Dienst Repositories bereitstellt. Neben der Nutzung von *Git* kommen dabei oft Plattformen wie *GitHub*, *GitLab* oder *BitBucket* zum Einsatz. Diese Plattformen bieten eine Reihe von Diensten, die die Arbeit mit *Git* erleichtern und die Verwaltung der Repositories verbessern. Dazu gehören Authentifizierungsmechanismen, Rechteverwaltung, benutzerfreundliche Oberflächen sowie weitere Funktionen, die eine effiziente Zusammenarbeit und Verwaltung der Daten ermöglichen.
Solche Plattformen ermöglichen es Entwicklern, ihre Repositories online zu speichern und zu verwalten, wodurch der Zugriff auf den Quellcode erleichtert und die Zusammenarbeit im Team gefördert wird. Sie bieten zudem Tools für Code-Reviews, Issue-Tracking, *Continuous Integration*/*Continuous Deployment* (CI/CD) und viele weitere Funktionen.

2.2 CI/CD

CI/CD dient der Optimierung und Automatisierung der Integration neuer Quellcodeänderungen sowie deren Bereitstellung. Dabei umfasst CI die Integration der neuen Codeänderungen in das Quellcode-Repository, während CD die Bereitstellung dieser Änderungen durch automatische Update-Releases in die Produktionsumgebung beinhaltet. CD kann sich dabei auf *Continuous Delivery* beschränken oder *Continuous Deployment* einschließen, da die Konzepte ähnlich sind und die Begriffe häufig synonym verwendet werden. *Continuous Delivery* beschreibt die Bereitstellung der Anwendung für die Produktionsumgebung, sodass diese für den Release vorbereitet ist, während *Continuous Deployment* die Anwendung tatsächlich auf dem Produktivsystem bereitgestellt. Dieser Ablauf ist in Abbildung 2.2 dargestellt. [6]

Eine CI/CD-Pipeline ist ein definierter Ablauf dieser einzelnen Prozesse, wobei ein Schritt nur ausgeführt wird, wenn der vorhergehende erfolgreich abgeschlossen wurde. CI kann auch ohne CD implementiert sein, jedoch kann CD nicht ohne CI verwendet werden, da CD auf CI aufbaut [9]. Bevor neue Quellcodeänderungen bereitgestellt werden, müssen diese zunächst ordnungsgemäß integriert und idealerweise getestet werden. Andernfalls können unvorhergesehene Fehler oder eine fehlgeschlagene Bereitstellung auftreten, was zu einem erhöhten anstatt reduzierten zeitlichen Aufwand führt.

Abbildung 2.2: CI/CD Ablauf

2.2.1 Continuous Integration

Um die Qualität der Software zu gewährleisten, werden Quellcodeänderungen häufig einem umfassenden Validierungsprozess unterzogen. Ein wichtiger Bestandteil dieses Prozesses ist die statische Code-Analyse, welche die Qualität des Quellcodes bewertet und auf logische Fehler überprüft. Diese Analyse kann wiederkehrende Fehler identifizieren und verhindern, dass solche Fehler in die Produktionsumgebung gelangen. Darüber hinaus wird der Quellcode auf bekannte Schwachstellen getestet, um diese bereits vor der Bereitstellung zu identifizieren und zu beheben. [10]

Nach der statischen Überprüfung des Quellcodes werden verschiedene Testverfahren angewendet, um die Logik und Funktionalität des Quellcodes sicherzustellen. Ein zentrales Element hierbei sind Unit-Tests, bei denen einzelne Komponenten des Quellcodes isoliert getestet werden, um die Korrektheit ihrer Rückgabewerte zu validieren [11]. Auf diesen Tests aufbauend können Regressionstests durchgeführt werden, die sicherstellen, dass behobene Fehler nicht erneut auftreten. Zusätzlich werden Integrationstests eingesetzt, um die Interoperabilität der einzelnen Komponenten zu überprüfen und sicherzustellen, dass sie zusammen ordnungsgemäß funktionieren. Sollten alle Tests erfolgreich verlaufen und die Quellcodeänderungen sich als funktional erweisen, werden diese Änderungen mit dem bestehenden Quellcode zusammengeführt. [9]

2.2.2 Continuous Deployment

Nach dem Bearbeiten des Quellcodes muss die Anwendung bereitgestellt werden, um die neue Version dieser Anwendung zu verteilen. Um neue Versionen schneller zu verteilen, sodass die Ausfallzeit möglichst gering ist, soll *Continuous Deployment* automatisch und ohne menschliche Interaktion die neue Version bereitstellen. Dies führt auch dazu, dass neue Versionen häufiger verteilt werden können, sodass beispielsweise Sicherheitslücken schneller geschlossen werden.

2.2.3 Sicherheit von CI/CD-Systemen

Die Sicherheit von *Continuous Integration*- und *Continuous Deployment*- (CI/CD) Systemen ist von entscheidender Bedeutung für die Integrität und Verlässlichkeit moderner Softwareentwicklungsprozesse. Das *Open Worldwide Application Security Project (OWASP)* hat aufgezeigt, dass Angriffe auf die CI/CD-Infrastruktur erhebliche Risiken für Organisationen mit sich bringen [12]. Ein prominentes Beispiel ist der Angriff auf den Build-Prozess der Software *Orion*, bei dem es Angreifern gelang, den Build zu manipulieren und die Anwendung mit einer gültigen Signatur auszuliefern. Dies führte dazu, dass ein Trojaner unbemerkt an über 18.000 Kunden verteilt wurde [13].

Die Konsequenzen solcher Sicherheitsvorfälle sind weitreichend und können nicht nur den Ruf eines Unternehmens erheblich schädigen, sondern auch zu finanziellen Verlusten und rechtlichen Konsequenzen führen. Um diesen Gefahren entgegenzuwirken, hat OWASP in Zusammenarbeit mit Experten aus der Industrie die zehn wichtigsten Sicherheitsprobleme im Bereich CI/CD identifiziert und analysiert. Diese Sicherheitsprobleme und die entsprechenden Lösungsansätze werden in den folgenden Abschnitten detailliert beschrieben.

2.2.3.1 Unzureichende Ablaufkontrollregelung

Unzureichende Ablaufkontrollmechanismen innerhalb von CI/CD-Prozessen stellen ein erhebliches Sicherheitsrisiko dar. Ein Angreifer mit Berechtigungen im Quellcodemanagementsystem kann bösartigen Code in die Pipeline einbringen, der ohne zusätzliche Genehmigungen oder Überprüfungen durch andere verarbeitet wird. Dies liegt in der Natur der Automatisierung, die das zentrale Ziel eines CI/CD-Systems darstellt. Der Prozess sieht vor, dass Quellcode, sobald er hochgeladen wird, automatisch und ohne Nutzerinteraktion verarbeitet wird. Infolgedessen können Quellcodeänderungen, die in das Quellcodemanagementsystem eingeführt wurden, in den bestehenden Quellcode integriert und schließlich in das Produktionssystem übernommen werden.

Um diese Sicherheitslücke zu schließen, sollten neue Quellcodeänderungen niemals direkt in das Produktionssystem übernommen werden, ohne vorher einer gründlichen Überprüfung durch andere Entwickler unterzogen worden zu sein. Automatisierte Tests können hingegen bedenkenlos durchgeführt werden, da sie in einer isolierten Umgebung ablaufen sollten. Weiterhin sollten Quellcodeänderungen nur dann in das Produktionssystem übernommen werden, wenn sie auf vorher definierten Branches, üblicherweise dem *master*- oder *main*-Branch, durchgeführt wurden. Der *master*- oder *main*-Branch sollte mit einer Regel ausgestattet sein, die verhindert, dass Änderungen manuell hinzugefügt werden können. Stattdessen sollten Änderungen nur über Pull-Requests von anderen Branches eingebracht werden, nachdem diese von anderen Entwicklern überprüft wurden. [14]

2.2.3.2 Unzureichendes Identitäts- und Zugriffsmanagement

Ein effektives Identitäts- und Zugriffsmanagement ist von zentraler Bedeutung für die Sicherheit von CI/CD-Systemen. Ein wesentlicher Grundsatz hierbei ist das Prinzip der minimalen Rechtevergabe (*Least Privilege Principle*). Dieses Prinzip besagt, dass ein Nutzer nur die minimalen Rechte erhalten sollte, die für die Erfüllung seiner Aufgaben notwendig sind. In der Praxis erhalten Nutzer jedoch häufig mehr Rechte als erforderlich. Dies kann auf eine mangelnde Granularität in der Rechteverwaltung zurückzuführen sein oder darauf, dass sich die Aufgaben eines Nutzers im Laufe der Zeit verändert haben, ohne dass seine Zugriffsrechte entsprechend angepasst wurden. [15]

Regelmäßige Überprüfungen der Nutzerkonten auf ihre Aktualität sind ebenfalls unerlässlich. Nach einem Mitarbeiterwechsel können verwaiste Konten mit aktiven Rechten bestehen bleiben. Ein Angreifer könnte solche Konten übernehmen und die gewährten Rechte ausnutzen, um Angriffe auf die Infrastruktur oder die Integrität des Quellcodes durchzuführen. Insbesondere Rechte, die den Zugriff auf sensible Daten innerhalb der CI/CD-Infrastruktur erlauben, sind kritisch. Angreifer könnten geheime Schlüssel auslesen, die zur Signatur von Softwareanwendungen verwendet werden. Dies könnte dazu führen, dass vermeintlich offizielle Software verbreitet wird, welche tatsächlich Malware enthält. [15]

Die Implementierung eines strikten Identitäts- und Zugriffsmanagements ist daher unverzichtbar. Durch regelmäßige Überprüfungen und Anpassungen der Zugriffsrechte sowie die Anwendung des Prinzips der minimalen Rechtevergabe können Sicherheitsrisiken erheblich reduziert werden. Dies trägt entscheidend zur Sicherung der CI/CD-Infrastruktur und zur Wahrung der Integrität des Softwareentwicklungsprozesses bei.

2.2.3.3 Abhängigkeitsverwaltung

Moderne Softwareprojekte sind häufig auf eine Vielzahl von Drittanbieter-Softwarepaketen angewiesen. Diese externen Abhängigkeiten bringen jedoch auch neue Sicherheitsrisiken mit sich. Insbesondere können Angreifer versuchen, durch die Manipulation von Abhängigkeiten schädlichen Code in den Build-Prozess einzuschleusen. Ein bekanntes Beispiel für eine solche Angriffsmethode ist die sogenannte *Dependency Confusion*. Dabei wird ein Paket mit demselben Namen wie ein internes Paket, das in einer privaten Paketverwaltung gespeichert ist, in eine öffentliche Paketverwaltung hochgeladen. Abhängig von der Konfiguration des Build-Systems kann es passieren, dass statt des internen Pakets das schädliche, öffentlich verfügbare Paket heruntergeladen wird. Dieses Paket kann dann bei der Installation Schadcode ausführen, insbesondere wenn es mit einem Installationsskript versehen ist. [16]

Ein weiterer Angriffspfad ist das *Typosquatting*. Hierbei wird ein bösartiges Paket mit einem ähnlichen Namen wie das Zielpaket erstellt und hochgeladen, um von typischen Schreibfehlern bei der Paketsuche zu profitieren [16].

Um solchen Angriffen vorzubeugen, empfiehlt es sich, statt öffentlicher Paketverwaltungen eine eigene, interne Paketverwaltung zu nutzen, die nur die benötigten und geprüften Pakete bereitstellt. Dies verhindert, dass Pakete nach ihrer erstmaligen Bereitstellung unbemerkt verändert werden können. Zusätzlich sollte ein Signaturvergleich der Checksummen dieser Pakete mit den bereitgestellten

Checksummen durchgeführt werden, um sicherzustellen, dass die Pakete unverändert und authentisch sind. Da viele Pakete Installationsskripte verwenden, ist es ratsam, diese Skripte in einem isolierten Kontext auszuführen. Auf diese Weise wird verhindert, dass vertrauliche Informationen wie geheime Schlüssel, die dem System bereitgestellt werden, ungewollt preisgegeben werden. [16]

Die sorgfältige Verwaltung von Abhängigkeiten ist somit ein wesentlicher Bestandteil der Sicherheits-Strategie in CI/CD-Systemen. Durch den Einsatz interner Paketverwaltungen und die Implementierung strenger Prüfmechanismen kann das Risiko, dass schädliche Pakete in den Build-Prozess gelangen, erheblich reduziert werden. Dies trägt maßgeblich zur Sicherstellung der Integrität und Sicherheit des gesamten Softwareentwicklungsprozesses bei.

2.2.3.4 Veränderung der Pipeline-Ausführung

Ein Angriff auf die CI/CD-Pipeline durch Veränderung des Quellcodes ohne direkten Zugriff auf die Build-Umgebung wird als Poisoned Pipeline Execution (PPE) bezeichnet. In diesem Szenario manipulieren Angreifer Quellcodedateien, wie etwa die Konfigurationsdateien von *GitHub Actions*, innerhalb des Quellcodemanagementsystems. Diese Änderungen können dazu führen, dass während des Build-Vorgangs schädliche Befehle ausgeführt oder gefährliche Parameter gesetzt werden, was letztlich die Integrität des gesamten Build-Prozesses gefährdet, wodurch die Pipeline „vergiftet“ wird. Sobald ein Angreifer die Kontrolle über die Pipeline erlangt, eröffnen sich vielfältige Möglichkeiten für böswillige Aktivitäten. Diese umfassen das Verändern von Quellcode und Konfigurationen, das Einfügen von schädlichen Dateien oder Code und das Stehlen von Geheimnissen wie Zugriffstoken. Die Auswirkungen solcher Angriffe können gravierend sein und die gesamte CI/CD-Infrastruktur sowie die resultierende Software kompromittieren. [17]

Zum Schutz vor solchen Angriffen empfiehlt es sich, keine Geheimnisse in automatisierten Build-Vorgängen zu verwenden, die unüberprüften Quellcode verarbeiten. Darüber hinaus sollte die Konfigurationsdatei für die Build-Umgebung in abgesicherten Branches oder separaten Repositories aufbewahrt werden. Durch diese Trennung wird es Angreifern mit Zugriff auf das Quellcodemanagementsystem erheblich erschwert, Änderungen am Build-Prozess vorzunehmen. Diese Maßnahmen tragen dazu bei, die Sicherheit der CI/CD-Pipeline zu erhöhen und die Integrität der erstellten Software zu gewährleisten. [17]

2.2.3.5 Unzureichende pipeline-basierte Zugriffskontrollen (PBAC)

Die Ausführung einer CI/CD-Pipeline involviert diverse sensible Operationen, einschließlich der Softwarekompilierung unter Verwendung von vertraulichen Informationen wie Umgebungsvariablen und Zugriffstoken. Diese Prozesse erfordern den temporären Zugriff der Pipeline auf die entsprechenden Daten. Pipeline-basierte Zugriffskontrollen (PBAC) definieren die Restriktion des Zugriffs durch die Pipeline in jedem Ausführungsschritt auf die für den jeweiligen Prozess essenziellen Ressourcen. Bei einer Infiltration der Pipeline durch bösartige Akteure besteht das Risiko eines unbefugten Zugriffs auf die der Pipeline zugänglichen vertraulichen Informationen. Dies kann potenziell zur Exfiltration sensibler Daten oder zur Einschleusung schädlicher Artefakte in den Quellcode führen. Die Auswirkungen einer solchen Kompromittierung korrelieren direkt mit der Granularität der implementierten PBAC: Je präziser die Zugriffskontrollen, desto geringer das potenzielle Schadensausmaß durch eine unerlaubte Nutzung der im spezifischen Kontext erforderlichen Ressourcen. [18]

Zur Prävention unbefugter Datenzugriffe wird empfohlen, individuelle Pipeline-Prozesse nach dem Prinzip der minimalen Rechtevergabe zu konfigurieren. Darüber hinaus sollten diese Prozesse keinen uneingeschränkten Internetzugang besitzen; stattdessen ist eine aufgabenspezifische Netzwerksegmentierung zu implementieren. Für den Fall, dass der Zugriff auf externe Internetressourcen erforderlich ist, können diese Operationen in separate Aufgaben ausgelagert werden, sodass der entsprechende Ausführungskontext keinen Zugriff auf vertrauliche Informationen erhält. Abschließend wird empfohlen, die Ausführungsknoten nach der Nutzung in ihren Ursprungszustand zurückzuversetzen, um permanente Modifikationen zu verhindern. [18]

2.2.3.6 Unzureichende Hygiene von Anmeldeinformationen

CI/CD-Systeme bestehen in der Regel aus einer Vielzahl von miteinander verknüpften Systemen, die durch Authentifizierung gesichert sind. Diese Systeme umfassen Quellcode-Repositories, Build-Server, Testumgebungen und Produktionssysteme. Um die Integrität und Sicherheit des gesamten Entwicklungsprozesses zu gewährleisten, sind Anmeldeinformationen wie Passwörter und Token im gesamten CI/CD-Prozess im Einsatz. Diese weit verbreitete Nutzung von Anmeldeinformationen birgt erhebliche Sicherheitsrisiken, insbesondere wenn diese nicht sorgfältig verwaltet und geschützt werden. [19]

Ein großes Risiko besteht darin, dass Anmeldeinformationen in Quellcode-Repositories eingecheckt werden können. Selbst wenn solche sensiblen Informationen später wieder entfernt werden, bleiben sie in den Commit-Historien gespeichert und sind somit weiterhin zugänglich. Dieses Szenario bietet Angreifern die Möglichkeit, auf diese Anmeldeinformationen zuzugreifen und damit auf die gesicherten Systeme zu gelangen. Um solche Risiken zu minimieren, sollten Passwörter und Token regelmäßig rotiert werden. Eine regelmäßige Rotation verhindert, dass gestohlene oder kompromittierte Anmeldeinformationen dauerhaft verwendet werden können. [19] Ein weiteres Sicherheitsproblem entsteht durch die Verwendung von Anmeldeinformationen in Kommandozeilenprogrammen. Befehle, die Anmeldeinformationen als Parameter enthalten, können in der Befehlsausgabe oder in Logdateien sichtbar sein. Dadurch wird es möglich, dass Anmeldeinformationen von unbefugten Personen abgefangen und missbraucht werden. Daher ist es entscheidend, dass Anmeldeinformationen niemals in der Befehlszeile oder in Logdateien erscheinen. [19]

Ein besonders effektiver Schutzmechanismus ist die Verwendung von Einmalpasswörtern (One-Time Passwords, OTPs), die nur für die Dauer einer bestimmten Ausführung gültig sind. Einmalpasswörter verhindern, dass abgegriffene Anmeldeinformationen für zukünftige Angriffe verwendet werden können. Diese Praxis erhöht die Sicherheit der CI/CD-Infrastruktur erheblich, da selbst kompromittierte Anmeldeinformationen nur für einen sehr kurzen Zeitraum nutzbar sind. [19]

Durch die Implementierung dieser Maßnahmen kann die Sicherheit von CI/CD-Systemen signifikant verbessert und das Risiko eines erfolgreichen Angriffs auf die Infrastruktur reduziert werden. Es ist essenziell, dass Organisationen diese Aspekte in ihre Sicherheitsstrategien integrieren, um die Integrität und Vertraulichkeit ihrer Entwicklungs- und Produktionssysteme zu gewährleisten.

2.2.3.7 Unsichere Systemkonfiguration

Jede Komponente eines CI/CD-Systems erfordert spezifische Konfigurationen und hat eigene Best Practices, um sicher betrieben zu werden. Besonders bei selbst gehosteten Systemen ist es essenziell, sicherzustellen, dass immer die neueste Version der Software genutzt wird. Dies minimiert das Risiko, dass bekannte Sicherheitslücken ausgenutzt werden können. Ein kritischer Aspekt der Systemsicherheit ist die richtige Einschränkung des Netzwerkzugriffs. Es muss gewährleistet sein, dass nur die unbedingt notwendigen Ressourcen und Systeme erreichbar sind. Dies reduziert die Angriffsfläche und schützt vor unbefugtem Zugriff auf sensitive Bereiche der CI/CD-Infrastruktur [20].

Selbst gehostete Systeme sollten zudem mit minimalen Rechten auf dem zugrunde liegenden Betriebssystem betrieben werden. Dies bedeutet, dass im Falle eines Angriffs der potenzielle Schaden begrenzt bleibt, da der Angreifer nicht auf andere kritische Teile des Systems zugreifen kann. Die Implementierung von Prinzipien wie dem Prinzip der minimalen Rechtevergabe kann dabei helfen, die Auswirkungen eines erfolgreichen Angriffs zu minimieren. [20]

Ein oft übersehener, aber entscheidender Punkt ist die Deaktivierung von Standard-Anmeldeinformationen. Standard-Benutzerkonten und -Passwörter stellen ein erhebliches Sicherheitsrisiko dar, da sie Angreifern einen einfachen Zugang ermöglichen. Zudem sollten neue Nutzer verifiziert werden, bevor sie Zugriff auf das System erhalten. Ein prominentes Beispiel für die Folgen unsicherer Konfigurationen ist der Vorfall bei Mercedes-Benz, bei dem Quellcode exfiltriert wurde. Der Quellcode wurde auf einem selbst gehosteten GitLab-Server gespeichert, der nicht ordnungsgemäß konfiguriert war. Dadurch konnten sich neue Nutzer selbstständig registrieren und auf die Quellcoderepositories zugreifen. [21]

Durch die konsequente Anwendung dieser Sicherheitsmaßnahmen und Best Practices können die Risiken, die mit unsicheren Systemkonfigurationen einhergehen, erheblich reduziert werden. Organisationen sollten kontinuierlich ihre Systeme überwachen und sicherstellen, dass ihre CI/CD-Infrastrukturen stets den neuesten Sicherheitsstandards entsprechen.

2.2.3.8 Unkontrollierte Nutzung von Diensten Dritter

Die Integration von Diensten Dritter in CI/CD-Systeme ist weit verbreitet und erfolgt oft mühelos. Bei Plattformen wie *GitHub* können Drittanbieterdienste durch *GitHub*-Anwendungen, OAuth-Token für die Authentifizierung, Bereitstellung über SSH-Schlüssel sowie die Konfiguration von Webhook-Ereignissen eingebunden werden. Diese Drittanbieterdienste erhalten damit weitreichende Berechtigungen, die in einigen Fällen die Verwaltung des gesamten Quellcodemanagementsystems umfassen können. Zudem können bei *GitHub Actions* externe Aktionen aus dem Marketplace importiert werden, was zusätzliche Risiken birgt. [22]

Die Integration dieser Dienste führt jedoch zu einem erheblichen Kontrollverlust. Es wird zunehmend schwieriger, den Überblick darüber zu behalten, welcher Drittanbieter welche Berechtigungen besitzt und welche Rechte er hat. Dies vergrößert die Angriffsfläche eines Unternehmens erheblich, da die Kompromittierung eines einzigen Drittanbieters weitreichende Konsequenzen haben kann. Beispielsweise könnte ein kompromittierter Dienst Malware über die CI/CD-Pipeline einschleusen und so das gesamte System gefährden. [22]

Um diesen Risiken entgegenzuwirken, sollten Drittanbieterdienste nach dem Prinzip der geringsten Rechtevergabe integriert werden. Das bedeutet, dass ein Drittanbieter nur die minimal notwendigen Berechtigungen erhalten sollte, die zur Erfüllung seiner Aufgaben erforderlich sind. Darüber hinaus ist eine umfassende Dokumentation aller Berechtigungen, die Drittanbietern gewährt wurden, unerlässlich. Dies ermöglicht eine bessere Nachverfolgbarkeit und Kontrolle über die eingesetzten Dienste. [22]

Es ist ebenfalls wichtig, sicherzustellen, dass Drittanbieter keinen Zugriff auf geheime Schlüssel oder andere sensible Informationen haben. Wenn ein Dienst diese Informationen nicht zwingend benötigt, sollte der Zugriff verweigert werden. Zudem sollten nicht benötigte Drittanbieterdienste konsequent entfernt werden, um die Angriffsfläche zu minimieren. [22]

Durch die sorgfältige Verwaltung und Überwachung der Nutzung von Diensten Dritter können die Sicherheitsrisiken erheblich reduziert und die CI/CD-Infrastruktur besser geschützt werden. Es ist entscheidend, regelmäßig zu überprüfen, welche Drittanbieter integriert sind, welche Berechtigungen sie haben und ob diese Berechtigungen noch notwendig sind.

2.2.3.9 Unzureichende Validierung der Integrität

Bei der Erstellung von Softwarepaketen im Rahmen von CI/CD-Prozessen werden häufig verschiedene Artefakte kombiniert. Diese stammen sowohl von Drittanbietern als auch aus eigener Entwicklung und werden von mehreren Beteiligten bereitgestellt. Diese Vielfalt an Quellen und Beteiligten schafft zahlreiche Einstiegspunkte, über die Angreifer versuchen können, Ressourcen innerhalb des CI/CD-Prozesses zu manipulieren. Ein solcher Angriff kann schwerwiegende Folgen haben, insbesondere wenn eine manipulierte Ressource in das Produktionssystem übernommen wird, da sie potenziell bösartigen Code mitbringen kann. [23]

Um diese Risiken zu minimieren, ist es entscheidend, die Integrität aller Artefakte, die in den CI/CD-Prozess einfließen, konsequent zu überprüfen. Eine effektive Methode zur Sicherstellung der Integrität ist das *Code Signing*. Dabei werden die Artefakte digital signiert, wodurch ihre Authentizität und Unverändertheit bestätigt wird. Zusätzlich zur Signierung der finalen Pakete können auch einzelne Commits signiert werden. Dies verhindert, dass der CI/CD-Prozess durch externe Eingriffe manipuliert wird. [23]

Ein weiterer wichtiger Mechanismus zur Integritätsprüfung ist der Vergleich von Hash-Werten der Ressourcen. Hierbei wird für jedes Artefakt ein Hash-Wert berechnet, der eindeutig dessen Inhalt repräsentiert. Diese Hash-Werte können dann mit bekannten, sicheren Werten verglichen werden, um sicherzustellen, dass keine unautorisierten Änderungen vorgenommen wurden. [23]

Durch die Implementierung dieser Maßnahmen kann die Integrität der verwendeten Ressourcen gewährleistet und die Sicherheit des gesamten CI/CD-Prozesses erheblich erhöht werden. Es ist unerlässlich, dass Unternehmen regelmäßig die Integritätsprüfung ihrer Artefakte durchführen und dabei modernste Techniken und Best Practices anwenden, um den steigenden Sicherheitsanforderungen gerecht zu werden.

2.2.3.10 Unzureichende Protokollierung

Zur Vorbereitung auf sicherheitsrelevante Vorfälle ist die Erkennung von Angriffen essenziell. Hierfür ist eine umfassende Transparenz der Systeme notwendig, sodass jeder Vorgang, wie etwa das Pushen von Code, die Ausführung von Builds oder das Hochladen von Artefakten, nachvollzogen werden kann. Fehlende Protokollierung ermöglicht es Angreifern, ungesehen zu agieren, und verhindert eine adäquate Reaktion seitens des Unternehmens auf potenzielle Sicherheitsbedrohungen. [24]

Um diesem Risiko vorzubeugen, sollten alle relevanten Systeme erfasst und deren Protokollierungsfunktionen aktiviert sowie aggregiert werden. Eine zentrale Sammlung und Analyse dieser Protokolle ist notwendig, um ein umfassendes Bild der Systemaktivitäten zu erhalten. Zudem sollten Warnungen bei Anomalien aktiviert werden, sodass bei ungewöhnlichen Aktivitäten sofortige Maßnahmen ergriffen werden können. [24]

Je mehr Protokolle vorliegen und je genauer diese überprüft werden, desto präziser ist die Kenntnis über die Abläufe im System. Dies verbessert die Fähigkeit, Anomalien zu erkennen und angemessen darauf zu reagieren. Durch die Implementierung umfassender Protokollierungs- und Überwachungsstrategien kann die Sicherheit der CI/CD-Systeme erheblich gesteigert werden, da potenzielle Bedrohungen frühzeitig erkannt und neutralisiert werden können.

2.3 Java

Die Programmiersprache Java wurde im Jahr 1995 von *Sun Microsystems* veröffentlicht [25] und ist eine weit verbreitete, objektorientierte Sprache, deren Syntax an die von C und C++ angelehnt ist. Java wurde ursprünglich entwickelt, um eine Vielzahl von Host-Architekturen zu unterstützen, sodass Software auf verschiedenen Systemen mit unterschiedlichen Hardware- und Softwarekonfigurationen ausgeführt werden kann. Das Hauptziel bestand darin, den Quellcode einmal zu schreiben und dann überall ausführen zu können (im Original: „write once, run anywhere“) [26]. Der zentrale Baustein der Java-Plattform ist die Java Virtual Machine (JVM). Diese Komponente gewährleistet die Hardware- und Betriebssystemunabhängigkeit, da sie einen eigenen Befehlssatz besitzt und während der Laufzeit verschiedene Speicherbereiche verwaltet. Die JVM interpretiert dabei Binärdateien im Klassendateiformat (*class*-Dateiformat), welches Bytecode oder auch Befehlssätze für die JVM sowie andere Zusatzinformationen enthält [27]. Neben Java nutzen auch andere Programmiersprachen die JVM, darunter Kotlin, Clojure, Jython und JRuby [28].

Um eine Java-Anwendung zu erstellen, wird ein Java Development Kit (JDK) benötigt, das verschiedene Entwicklerwerkzeuge, wie den Java-Compiler, bereitstellt. Der in Java geschriebene Quellcode wird durch den Compiler in Bytecode übersetzt, den die JVM interpretieren kann. Die JVM ist Teil des Java Runtime Environment (JRE), einer Softwareschicht, die Klassenbibliotheken und andere Ressourcen bereitstellt, die für die Ausführung einer Java-Anwendung erforderlich sind. Die JRE kombiniert den Java-Bytecode mit den notwendigen Bibliotheken, um die Ausführung in der JVM zu ermöglichen [26].

2.3.1 Java Virtual Machine

Die JVM interpretiert den kompilierten Java-Code, der in einem speziellen Binärformat vorliegt. Dieser Code wird typischerweise im Klassendateiformat gespeichert und repräsentiert eine Klasse oder Schnittstelle. Das JDK enthält einen Compiler, der den Quellcode der Programmiersprache Java in den Befehlssatz der JVM übersetzt. Es ist wichtig zu betonen, dass die Typüberprüfungen bereits vom Compiler durchgeführt werden und nicht erst in der JVM erfolgen. [27]

Die Verwendung des Klassendateiformats und die Durchführung von Typüberprüfungen während des Kompilierungsprozesses tragen zur Effizienz und Sicherheit der Java-Plattform bei. Der übersetzte Bytecode ermöglicht es der JVM, Programme plattformunabhängig auszuführen, was eine der größten Stärken von Java darstellt. Die JVM fungiert somit als ein zentraler Bestandteil der Java-Plattform, der die Ausführung von Java-Anwendungen ermöglicht, indem sie den kompilierten Bytecode interpretiert und ausführt. [26]

2.3.2 Java Werkzeuge

Das Werkzeug *jpackage* wurde mit JDK 14 eingeführt und basiert auf dem früheren JavaFX-Werkzeug *javapackager*, das mit JDK 11 entfernt wurde. *jpackage* dient der Kompilierung und Vorbereitung von Java-Anwendungen für die Auslieferung. Es unterstützt dabei native Verpackungsformate für verschiedene Betriebssysteme: *msi* und *exe* für Windows, *dmg* und *pkg* für macOS sowie *deb* und *rpm* für Linux. Ziel dieses Werkzeugs ist es, installierbare Anwendungen zu erstellen, die für den Nutzer in der jeweils plattformüblichen Variante installierbar, nutzbar und deinstallierbar sind. [29]

Eine Laufzeitumgebung, die für die Ausführung der Anwendung verantwortlich ist, enthält sowohl die Anwendung selbst als auch ein JDK-Abbild. Zur Erstellung dieser Laufzeitumgebung wird das Werkzeug *jlink* durch *jpackage* standardmäßig genutzt. *jlink* dient der Verknüpfung von Modulen und deren transitiven Abhängigkeiten. Darüber hinaus führt *jlink* Optimierungen durch, die während der Kompilierung zu komplex oder zur Laufzeit rechenintensiv wären. Zu diesen Optimierungen gehören unter anderem Berechnungen, deren Eingabewerte bereits bekannt sind, sowie das Entfernen von nicht aufgerufenem Quellcode. Das Ergebnis dieses Prozesses ist eine optimierte Laufzeitumgebung oder eine ausführbare Datei. [30]

Der Aufbau dieser Laufzeitumgebungen wird exemplarisch für jede Plattform in Abbildung 2.3 dargestellt.

2.4 Gradle

Gradle ist das führende Build-System für die JVM und dient als Standardsystem für Android- und Kotlin-Multiplattform-Projekte. Mit einem reichhaltigen Community-Plugin-Ökosystem kann Gradle eine Vielzahl von Software-Build-Szenarien automatisieren, sei es durch integrierte Funktionen, Plugins von Drittanbietern oder benutzerdefinierte Build-Logik. Ein wesentliches Merkmal von Gradle ist seine hochentwickelte, deklarative und ausdrucksstarke Build-Sprache, die das Lesen und Schreiben von Build-Logik erleichtert. Optimierungen wie inkrementelle Builds, Build-Caching und parallele Ausführung tragen dazu bei, die Build-Zeiten zu verkürzen und die Effizienz zu steigern. Durch seine

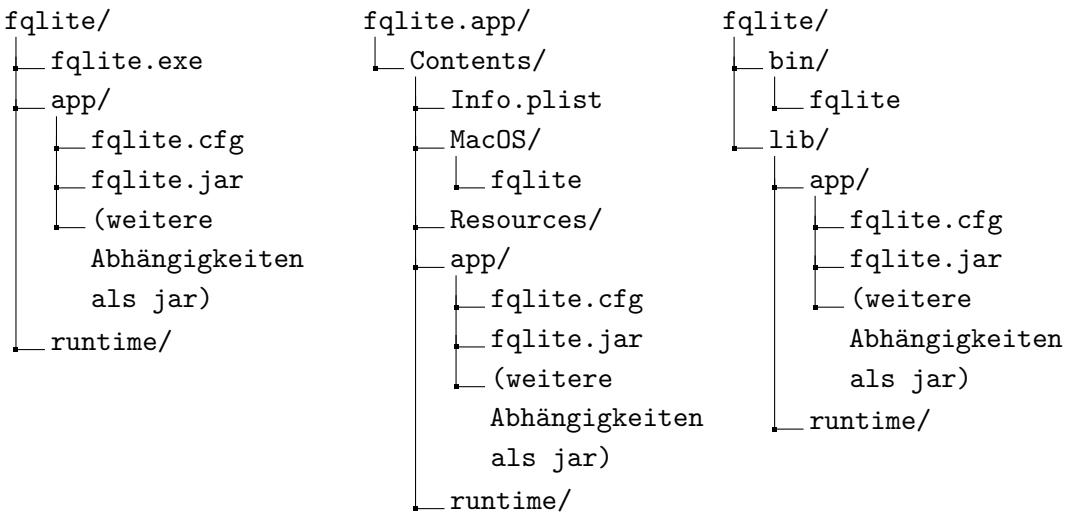

Abbildung 2.3: Exemplarischer Aufbau der Ordnerstruktur der Laufzeitumgebungen für Windows (links), macOS (mitte) und Linux (rechts)

Zuverlässigkeit und Leistungsfähigkeit hat sich Gradle als bevorzugtes Tool in der modernen Softwareentwicklung etabliert. In den folgenden Unterkapiteln werden die grundlegenden Konzepte, die Architektur sowie die wichtigsten Funktionen und Vorteile von Gradle detailliert erläutert. [31]

2.4.1 Projektaufbau mit Gradle

Ein Gradle-Projekt kann sowohl ein einzelnes Projekt als auch seine Unterprojekte bauen. Der empfohlene Weg für die Nutzung von Gradle verläuft über den *Gradle Wrapper*. Der *Gradle Wrapper* ist ein Skript, welches eine definierte Version von Gradle, sofern nicht vorhanden, installiert und ausführt. Dieser *Gradle Wrapper* wird als *gradlew* bezeichnet. [32]

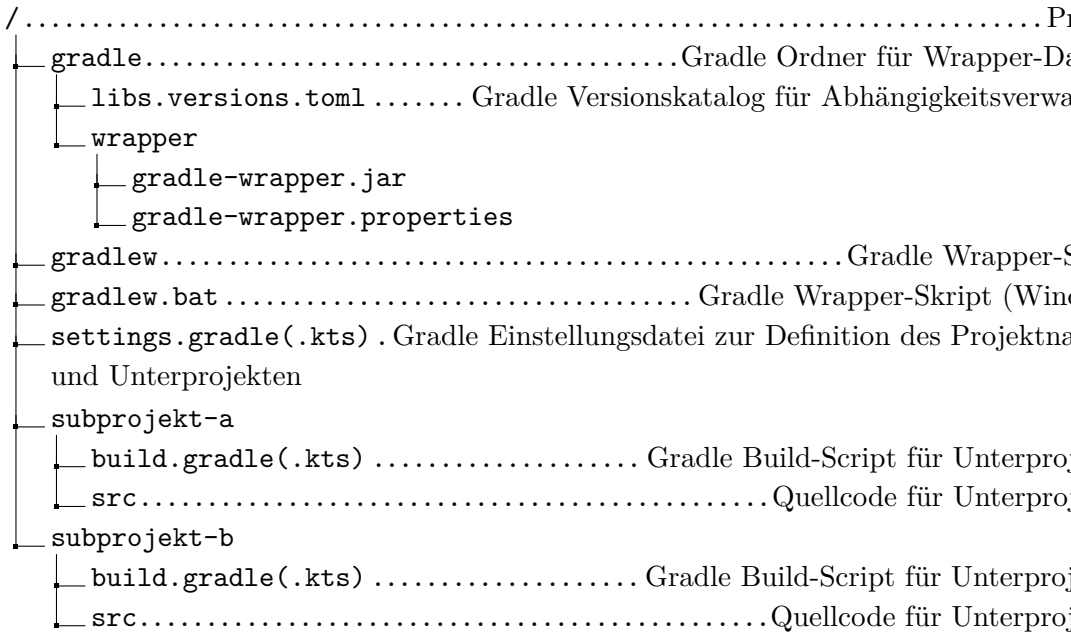

Abbildung 2.4: Exemplarischer Aufbau der Ordnerstruktur eines Gradle-Projekts mit Unterprojekten [33]

Ein exemplarischer Projektaufbau kann aus Abbildung 2.4 entnommen werden. Die zentralen Konfigurationsdateien für Gradle sind *settings.gradle(.kts)* und *build.gradle(.kts)*. Gradle-Dateien verwenden die Dateiendung *.gradle*, wenn die Groovy Domain Specific Language (DSL) genutzt wird, und *.gradle.kts*, wenn die Kotlin DSL verwendet wird. Groovy ist eine dynamische Skriptsprache für die JVM mit statischer Kompilierung für die Java-Plattform [34]. Die Kotlin DSL hingegen nutzt Kotlin als zugrundeliegende Programmiersprache zur Erstellung von Konfigurationen nach Kotlin-Syntax und wurde, analog zum Groovy-basierten Pendant, auf der Java-API von Gradle implementiert [35].

Die Hauptaufgabe der Datei *settings.gradle(.kts)* besteht darin, Unterprojekte zu einem Gradle-Projekt hinzuzufügen. Diese Datei ist optional für einzelne Projekte, jedoch erforderlich, wenn Unterprojekte definiert werden sollen. [36]

In der Datei *build.gradle(.kts)* werden die Build-Konfigurationen festgelegt. Jeder Gradle-Build besteht aus mindestens einem Build-Skript, das zwei Arten von Abhängigkeiten definiert: Erstens Abhängigkeiten wie Bibliotheken und Plugins, von denen Gradle selbst abhängt, und zweitens Bibliotheken, von denen der Quellcode des Projekts abhängt. Plugins erweitern die Funktionen von Gradle und können innerhalb des Build-Skripts konfiguriert werden. [37]

2.4.2 Build-Ablauf

Aufgaben beschreiben unabhängige Arbeitseinheiten, wie beispielsweise das Kompilieren von Klassen [38]. Beim Erstellen eines Build-Scriptw werden Aufgaben definiert, welche in einer bestimmten Abfolge durch Gradle ausgeführt werden. Vor jeder Ausführung wird durch Gradle ein gerichteter Graph erzeugt, welcher die Abfolge der Aufgaben darstellt. Ein exemplarischer Graph für eine Java-Anwendung wird in Abbildung 2.5 dargestellt. [39]

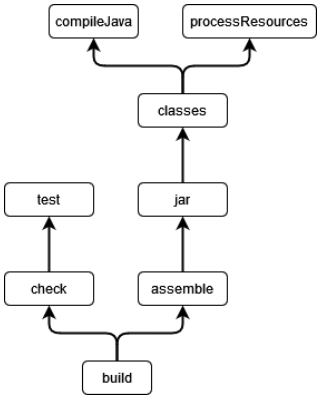

Abbildung 2.5: Gradle Task Graph für eine Java Anwendung [39]

Gradle durchläuft während der Ausführung drei Phasen: Initialisierung, Konfiguration und Ausführung.

Initialisierung Während der Initialisierung evaluiert Gradle die *settings.gradle(.kts)* Datei und erstellt intern eine *Settings*-Instanz mit den nötigen Konfigurationen, sowie eine *Project*-Instanz für jedes mit diesem Build verbundene Projekt.

Konfiguration Gradle verarbeitet die *build.gradle(.kts)*-Dateien jedes Projektes dieses Builds und erstellt einen Graphen für die angeforderten Aufgaben.

Ausführung Die Abhängigkeiten zwischen den Aufgaben bestimmen die Reihenfolge der Ausführung der einzelnen Aufgaben. So können einige Aufgaben parallel ausgeführt werden, während andere das Ergebnis einer zuvor ausgeführten Aufgabe benötigen.

2.5 Paketmanager

Im Bereich der Softwareentwicklung dient ein Paketmanager zur Automatisierung der Installation, Aktualisierung, Konfiguration und Entfernung von Softwarepaketen auf konsistente Weise. Dies reduziert den manuellen Aufwand für den Endanwender. Diese Werkzeuge interpretieren in der Regel auch Paket-Metadaten wie Versionsinformationen oder Paketabhängigkeiten, um Konflikte zu vermeiden und sicherzustellen, dass alle erforderlichen Voraussetzungen erfüllt sind. [40]

Alternative Installationsmethoden beziehen sich auf Verfahren, die von den herkömmlichen Installationsprogrammen für die jeweiligen Betriebssysteme abweichen. Hierzu zählt die Installation unter Nutzung eines Paketmanagers mit beispielsweise *Chocolatey* unter Windows sowie *Homebrew* unter macOS. Diese Methoden ermöglichen eine Installation, ohne dass das Installationsprogramm manuell heruntergeladen werden muss.

2.5.1 Chocolatey

Chocolatey ermöglicht die Installation von Anwendungen für Windows über die Kommandozeile, ohne dass das Installationsprogramm manuell heruntergeladen werden muss. Dies vereinfacht den Installations- und Deinstallationsprozess, da diese über *Chocolatey* behandelt und durch den Autor des Pakets spezifiziert werden können. Ein wesentlicher Vorteil von *Chocolatey* ist die vereinfachte Aktualisierung von Paketen, da Updates zentral verwaltet und bereitgestellt werden können. [41]

NuGet, der Paketmanager, auf dem *Chocolatey* basiert, wurde von Microsoft entwickelt und ist ein zentraler Bestandteil des .NET-Ökosystems. NuGet wurde speziell für die Bereitstellung und Verwaltung von Paketen in Microsofts .NET (dotnet)-Umgebung konzipiert. Ein NuGet-Paket ist im Wesentlichen eine komprimierte ZIP-Datei mit der Dateierweiterung *.nupkg*. Diese Datei enthält neben den eigentlichen Anwendungskomponenten ein Manifest, das wichtige Metadaten wie die Paketversion, die Autoreninformationen, sowie Abhängigkeiten zu anderen Paketen auflistet. [42]

Die Verteilung dieser Pakete erfolgt über öffentliche oder private Pakethoster. Öffentliche Pakethoster, wie das offizielle NuGet-Repository, ermöglichen den freien Zugang zu einer Vielzahl von Paketen, während private Pakethoster genutzt werden können, um unternehmensinterne oder proprietäre Pakete zu verteilen. Die Struktur von NuGet ermöglicht es auch, dass einzelne Pakete Abhängigkeiten zu anderen Paketen haben können, was eine modulare und flexible Entwicklung ermöglicht. Diese Abhängigkeiten werden automatisch verwaltet, indem NuGet sicherstellt, dass alle benötigten Bibliotheken und Pakete korrekt installiert und aktualisiert werden. [42]

NuGet ist nicht nur für .NET-Projekte von Bedeutung, sondern hat auch in anderen Bereichen der Softwareentwicklung an Relevanz gewonnen, da es eine effiziente und skalierbare Methode der Paketverwaltung bietet. Zudem ist NuGet Open Source, weshalb es relativ einfach ist, diesen zu

erweitern oder zu nutzen, weshalb auch *Chocolatey* darauf aufbaut. Dies ermöglicht zudem die Installation von Paketen aus verschiedenen Quellen wie Chocolatey.org, NuGet.org, MyGet.org sowie privaten Repositories. [40]

Die meisten Pakete sind über das öffentliche Community-Repository von *Chocolatey* verfügbar, wodurch keine weitere Konfiguration erforderlich ist. Zusätzlich können weitere Repositories leicht hinzugefügt werden, um auf eine größere Vielfalt von Quellen zugreifen zu können. [41]

2.5.2 Homebrew

macOS wird standardmäßig mit dem integrierten Paketmanager „App Store“ ausgeliefert, der ausschließlich von Apple genehmigte Anwendungen anbietet. Jedoch existiert eine Vielzahl weiterer Anwendungen, die unter macOS installiert werden können. Um die Notwendigkeit zu umgehen, jedes dieser Programme manuell aus dem Quellcode zu kompilieren, erweist sich die Nutzung alternativer Paketmanager als effizient. Ein solcher Paketmanager ist *Homebrew*, der häufig als der „fehlende Paketmanager für macOS“ bezeichnet wird. [43]

Homebrew basiert auf den Technologien *Git* und *Ruby*. Ein Paket in *Homebrew* wird durch eine sogenannte *Formula* definiert, die sowohl Metadaten des Pakets als auch detaillierte Anweisungen für die Installation enthält. Eine *Formula* ist ein *Ruby*-Skript, das von *Homebrew* interpretiert wird [44]. Der Zugriff auf diese *Formulas* erfolgt über *Git*. Beispielsweise existiert ein zentrales *Git* Repository (https://github.com/Homebrew/homebrew-core), das von *Homebrew* verifizierte Pakete enthält. Zusätzlich besteht die Möglichkeit, weitere *Git* Repositories lokal hinzuzufügen, die zusätzliche *Formulas* enthalten, welche nicht im öffentlichen Repository *homebrew-core* verfügbar sind. Diese *Third-Party Repositories* (TAP) sind *Repositories* welche *Formula*, *Casks* oder andere *Homebrew*-Befehle enthalten [45]. Dabei werden standardmäßig *GitHub* Repositories bezogen, *Homebrew* ist aber nicht auf *GitHub* limitiert. Neben *Formulas* gibt es mit den sogenannten *Casks* eine weitere Paketdefinition, die speziell für macOS-native Anwendungen, häufig mit grafischer Benutzeroberfläche, genutzt wird. [46]

3 Methodik

Für das automatisierte Bauen und Bereitstellen der Anwendung *FQLite* soll eine CI/CD-Pipeline mittels *GitHub Actions* implementiert werden. Im Zentrum dieser Arbeit steht die Sicherheit der Pipeline und der bereitgestellten Software, um unautorisierte Änderungen durch Dritte auszuschließen. Dies erfordert die Anwendung bewährter Sicherheitspraktiken und die Implementierung robuster Mechanismen zur Überprüfung und Validierung des Quellcodes sowie der erzeugten Artefakte.

Ein wesentlicher Aspekt der Sicherheit in CI/CD-Systemen ist die strikte Kontrolle über den Zugriff und die Berechtigungen innerhalb der Pipeline. Best Practices umfassen hier die Verwendung von Prinzipien der minimalen Rechtevergabe, regelmäßige Überprüfung und Rotation von Zugangsdaten sowie die Implementierung von Mechanismen zur Überwachung und Protokollierung von Aktivitäten innerhalb der Pipeline. Diese Maßnahmen sollen sicherstellen, dass nur autorisierte Änderungen in die Produktionsumgebung gelangen und potenzielle Angriffe frühzeitig erkannt und abgewehrt werden können.

GitHub Actions bietet eine flexible und leistungsfähige Plattform zur Automatisierung von Softwareentwicklungsprozessen. Durch die Definition von Workflows können komplexe Build- und Bereitstellungsprozesse effizient und sicher ausgeführt werden. Die Verwendung von *GitHub Actions* ermöglicht es, die verschiedenen Schritte des CI/CD-Prozesses, von der Quellcodeüberprüfung bis zur Bereitstellung der Anwendung, nahtlos zu integrieren und zu automatisieren.

Ein weiterer zentraler Bestandteil dieser Arbeit ist die Nutzung von Gradle zur Verwaltung und Durchführung des Build-Prozesses. Gradle bietet umfangreiche Möglichkeiten zur Definition und Automatisierung von Build-Tasks. Dies umfasst die Verwaltung von Abhängigkeiten, das Kompilieren des Quellcodes und die Erstellung plattformspezifischer Installationspakete. Durch die Integration von Gradle in die *GitHub Actions* Workflows kann sichergestellt werden, dass der Build-Prozess konsistent und reproduzierbar durchgeführt wird.

In diesem Kapitel werden die Methoden und Techniken zur Implementierung der CI/CD-Pipeline für *FQLite* im Detail erläutert. Dabei wird auf die spezifischen Konfigurationen und Sicherheitsmaßnahmen eingegangen, die erforderlich sind, um eine sichere und effiziente Pipeline zu gewährleisten.

3.1 CI/CD Sicherheit

Die Sicherheit der CI/CD-Pipeline kann durch die Implementierung der empfohlenen Maßnahmen der *OWASP* Top 10 erhöht werden. Nachfolgend werden die durchgeführten Anpassungen erläutert, welche im Rahmen der Umsetzung dieses Projektes durchgeführt wurden.

3.1.1 Unzureichende Ablaufkontrollregelung

Um sicherzustellen, dass neue Änderungen nicht unkontrolliert in ein Produktivsystem gelangen, ist es essenziell, strikte Überprüfungsprozesse in den Entwicklungsablauf zu integrieren. Mit der Einführung eines CI/CD-Systems können Änderungen, die in einen Feature-Branch eingebracht

und anschließend in den *master*- oder *main*-Branch überführt werden, automatisch als Release-Kandidaten gekennzeichnet werden. Dies würde bedeuten, dass die entsprechenden Änderungen in die Anwendung aufgenommen und zur weiteren Prüfung freigegeben werden. Um das Risiko unautorisierter Änderungen zu minimieren, bietet *GitHub* die Möglichkeit, durch *Branch Protection Rules* spezifische Kontrollmechanismen für Branches zu implementieren.

Diese Schutzregeln verhindern, dass Änderungen direkt auf einen geschützten Branch, wie etwa den *master*- oder *main*-Branch, geschrieben werden können. Stattdessen müssen alle Änderungen über Pull Requests eingereicht werden, was eine zusätzliche Sicherheitsebene einführt. Es kann zudem festgelegt werden, wer zur Überprüfung und Genehmigung der Pull Requests berechtigt ist und wie viele Genehmigungen erforderlich sind, bevor die Änderungen in den *master*- oder *main*-Branch integriert werden dürfen. Diese Maßnahmen tragen erheblich zur Sicherstellung der Qualität und Sicherheit des Quellcodes bei, da mindestens ein weiterer Entwickler die Änderungen überprüft, bevor sie in den Hauptentwicklungszweig übernommen werden. Dies erschwert es Angreifern oder unbefugten Personen, Änderungen ohne die Zustimmung des ursprünglichen Autors in den *master*- oder *main*-Branch und somit auch in den *Release* einzubringen.

Darüber hinaus bieten die *Branch Protection Rules* weitere Möglichkeiten, den Branch zu schützen, wie beispielsweise die standardmäßige Verhinderung der Löschung eines geschützten Branches. Die Implementierung solcher Kontrollmechanismen ist entscheidend, um die Integrität und Sicherheit des CI/CD-Prozesses zu gewährleisten. Durch die Einführung von verbindlichen Überprüfungsprozessen und strengen Regeln für Quellcodeänderungen kann das Risiko unautorisierter Modifikationen und potenzieller Sicherheitsvorfälle erheblich reduziert werden.

3.1.2 Unzureichendes Identitäts- und Zugriffsmanagement

Die Rechteverwaltung einzelner Nutzerkonten kann bei diesem Projekt vernachlässigt werden, da die Arbeiten nur von einer Person durchgeführt werden und somit keine weiteren Nutzerkonten mit Rechten ausgestattet sind.

3.1.3 Abhängigkeitsverwaltung

Die Verwaltung der Abhängigkeiten für dieses Projekt erfolgt durch Gradle, wobei die benötigten Pakete hauptsächlich von *MavenCentral*, einer weit verbreiteten Plattform für die Bereitstellung und Verwaltung von Softwarepaketen, bezogen werden. *MavenCentral* bietet nicht nur eine umfassende Sammlung von Paketen, sondern auch eine benutzerfreundliche Suchfunktion, Zugriff auf die Webseiten der einzelnen Pakete sowie die Möglichkeit, unterschiedliche Versionen eines Pakets abzurufen. Darüber hinaus stellt die Plattform vorgefertigte Referenzen für gängige Build-Systeme wie Gradle und Maven bereit. Diese Funktion minimiert das Risiko des *Typosquatting*, da Schreibfehler beim Kopieren der Referenzen vermieden werden, solange die richtige Referenz verwendet wird.

Da in diesem Projekt ausschließlich öffentliche Pakete eingesetzt werden, kann das Risiko einer *Dependency Confusion* vernachlässigt werden. Die wenigen Ausnahmen, bei denen Pakete nicht von *MavenCentral* bezogen werden, betreffen solche, die ausschließlich als *GitHub-Releases* veröffentlicht wurden. Diese Pakete werden explizit mit einer Pfadangabe auf dem jeweiligen System referenziert. Um zusätzliche Sicherheit zu gewährleisten, sind alle durch Gradle bezogenen Pakete

explizit mit einer festgelegten Versionsnummer versehen. Dies garantiert, dass nur die gewünschte und bereits veröffentlichte Version eines Pakets verwendet wird. Diese Vorgehensweise, wie im Kapitel 2.2.3.3 zur Abhängigkeitsverwaltung beschrieben, stellt sicher, dass die Anwendung nicht unbeabsichtigt mit anderen, potenziell schädlichen Paketen gebaut wird.

3.1.4 Veränderung der Pipeline-Ausführung

Um unerwünschte Veränderungen an der Pipeline zu verhindern, werden Workflows ausschließlich auf spezifischen Branches ausgeführt, die zuvor vom Autor überprüft und genehmigt wurden. Eine Ausnahme bildet ein Build-Prozess, der lediglich zur Überprüfung des Prozesses dient. Dieser wird ausgeführt, bevor Änderungen in den *master*- oder *main*-Branch übernommen werden, um sicherzustellen, dass die Anpassungen einen erfolgreichen Build erzeugen. Diese Maßnahme gewährleistet, dass nur geprüfte und autorisierte Änderungen in den Release-Prozess einfließen, wodurch das Risiko von Sicherheitslücken und Fehlern erheblich reduziert wird.

Darüber hinaus werden sicherheitsrelevante Informationen, wie zum Beispiel FTP-Passwörter, ausschließlich in geschützten Branches verwendet. Diese Geheimnisse werden in der Ausgabe der Pipeline maskiert, wodurch ein Abfluss dieser sensiblen Daten durch unbefugte Dritte effektiv verhindert wird.

3.1.5 Unzureichende pipeline-basierte Zugriffskontrollen (PBAC)

In *GitHub* können Zugriffskontrollen innerhalb der Pipeline so konfiguriert werden, dass Geheimnisse (Secrets) ausschließlich in geschützten Branches zur Verfügung stehen. Diese Maßnahme verhindert, dass von Dritten in die Pipeline eingefügte Programme Zugriff auf diese sensiblen Informationen erhalten und diese auslesen können. Zusätzlich wird für jeden Job ein neuer Runner verwendet, wodurch keine Änderungen an den Systemen dauerhaft gespeichert werden. Diese Architektur bedingt jedoch, dass während der Ausführung eines vollständigen Builds mit *Release* das *Git*-Projekt mehrfach heruntergeladen werden muss. Die einzigen Informationen, die zwischen den Jobs persistiert werden, sind spezifische Ausgaben wie der generierte Paketname oder die Projektversion.

3.1.6 Unzureichende Hygiene von Anmeldeinformationen

Die Anmeldeinformationen für die Bereitstellung auf dem FTP-Server werden sicher über *GitHub Secrets* verwaltet. Während der Ausführung der Pipeline werden diese Informationen in den Protokollen durch maskierende Sterne geschützt, um zu verhindern, dass sie im Klartext angezeigt und ausgelesen werden können. Diese Sicherheitsvorkehrungen sollen das Risiko minimieren, das mit unsachgemäßem Umgang von Anmeldeinformationen verbunden ist, wie in Kapitel 2.2.3.6 dargelegt.

Die Authentifizierungsinformationen für *GitHub* werden nicht direkt durch den Autor bereitgestellt, sondern durch abstrahierte *GitHub Tokens*. Diese Tokens werden mit spezifischen, im Projekt definierten Rechten ausgestattet und von *GitHub* in die Pipeline integriert. Der Hauptvorteil dieser Methode liegt darin, dass der Token bei jedem Start der Pipeline automatisch von *GitHub* generiert wird und nach Abschluss der Pipeline seine Gültigkeit verliert. Dadurch werden einmalige Tokens für die Authentifizierung gegenüber *GitHub*-Diensten verwendet, die selbst im Falle eines Datenlecks keinen weiteren Schaden verursachen können. [47]

3.1.7 Unsichere Systemkonfiguration

Die Konfiguration der Runner wird vollständig von *GitHub* verwaltet, wodurch sichergestellt ist, dass immer die aktuellste Version mit den neuesten Sicherheitsfunktionen zum Einsatz kommt. Dies entlastet den Entwickler von der Notwendigkeit, selbst Konfigurationen vorzunehmen, und stellt sicher, dass Sicherheitsstandards konsequent eingehalten werden.

Um die Sicherheit weiter zu erhöhen, wird bei jeder Pipeline-Ausführung ein neuer Runner eingesetzt. Diese Maßnahme gewährleistet, dass die Ausführungsumgebung stets frisch und frei von potenziellen Rückständen aus vorherigen Prozessen ist. Dadurch werden Probleme vermieden, die durch verbleibende Zustände aus früheren Läufen verursacht werden könnten. Zudem schützt diese Vorgehensweise vor der Gefahr, dass bösartige Änderungen, die in einer früheren Ausführung vorgenommen wurden, Einfluss auf die aktuelle Ausführung nehmen.

3.1.8 Unkontrollierte Nutzung von Diensten Dritter

In *GitHub* können *Actions* projektübergreifend genutzt werden, was die Entwicklung beschleunigt und die Wiederverwendung bewährter Lösungen ermöglicht. Zu diesem Zweck bietet *GitHub* einen *Marketplace* an, in dem *GitHub Actions* und andere Erweiterungen von Nutzern und Organisationen bereitgestellt werden. Diese *Actions* können in Workflows integriert werden, wodurch Entwickler auf bereits implementierte Lösungen zurückgreifen und Aufgaben effizienter abschließen können.

In diesem Projekt wurden verschiedene von *GitHub* entwickelte *Actions* verwendet:

- *actions/checkout@v4*: Klont den Quellcode in den Workflow.
- *actions/setup-java@v4*: Richtet das JDK auf den Runnern ein.
- *actions/attest-build-provenance@v1*: Erstellt Checksummen zur Validierung der Artefakte.
- *actions/upload-artifact@v4*: Lädt temporäre Artefakte in der Pipeline auf *GitHub* hoch.
- *actions/download-artifact@v4*: Lädt temporäre Artefakte aus der Pipeline zur Veröffentlichung herunter.

Zusätzlich zu den von *GitHub* bereitgestellten *Actions* kamen auch Drittanbieter-*Actions* zum Einsatz, um den Workflow weiter zu optimieren:

- *crazy-max/ghaction-chocolatey@v3*: Installiert *Chocolatey* auf Windows-Runnern.
- *gradle/gradle-build-action@v3*: Richtet Gradle auf den Runnern ein.
- *softprops/action-gh-release@v2*: Erstellt ein *Release* auf *GitHub* und stellt die Artefakte zum Download bereit.
- *Dylan700/sftp-upload-action@latest*: Lädt die Artefakte auf den FTP-Server hoch.

Zusätzlich zur Implementierungsbeschleunigung durch Drittanbieter-*Actions* ist es entscheidend, diese sorgfältig auszuwählen und kontinuierlich zu überwachen, um Sicherheitsrisiken und potenzielle Abhängigkeiten von externen Quellen zu minimieren. In *GitHub* können diese *Actions* öffentlich eingesehen und bewertet werden, was eine Einschätzung ihres Sicherheitsrisikos ermöglicht.

Ein Beispiel ist die *Action* für das Erstellen eines *GitHub Releases*, die den *GitHub Token* verwendet. Dieser Token ist erforderlich, um Änderungen im Repository des Nutzers durchzuführen, und wird dabei so konfiguriert, dass er nur die nötigen Rechte besitzt. Die *Action* zum Hochladen von Artefak-

ten auf einen FTP-Server benötigt hingegen die FTP-Anmeldeinformationen, welche in der Ausgabe maskiert werden, da ohne diese keine Verbindung zum Server hergestellt werden kann. Hierdurch wird sichergestellt, dass die notwendige Authentifizierung erfolgt, während gleichzeitig das Risiko minimiert wird, dass sensible Informationen unbeabsichtigt offengelegt werden.

3.1.9 Unzureichende Validierung der Integrität

Die Sicherstellung der Integrität von Artefakten ist ein wesentlicher Bestandteil des Sicherheitsprozesses, da sie garantiert, dass die bereitgestellte Software unverändert und authentisch ist. Wie in Kapitel 2.2.3.9 beschrieben, erhöht die Signierung und Validierung der Artefakte das Vertrauen in die Software erheblich.

In diesem Projekt erfolgt die Integritätsprüfung der erzeugten Artefakte in der Pipeline durch die Nutzung von *GitHub Attestations*. Hierbei werden die Artefakte signiert und die entsprechenden Signaturinformationen sowie die Herkunft des Artefakts werden auf einem öffentlichen Server zur Verfügung gestellt. Nutzer, die diese Artefakte herunterladen, können somit die Integrität durch Angabe des *GitHub* Repositories und der entsprechenden Datei einfach überprüfen. Weitere Details zum Ablauf und zur Durchführung dieser Überprüfung sind in Kapitel 4.1.3 erläutert.

3.1.10 Unzureichende Protokollierung

In *GitHub* werden alle Pipeline-Ausführungen detailliert protokolliert und übersichtlich aufgelistet. Diese Protokollierung umfasst sowohl die ausgeführten Anweisungen als auch die dabei entstehenden Ausgaben. Besonders hervorzuheben ist, dass *GitHub* die Geheimnisse, die über *GitHub Secrets* hinterlegt wurden, automatisch maskiert, um ein unbefugtes Auslesen dieser sensiblen Informationen zu verhindern. Die umfassende Protokollierung durch *GitHub* gewährleistet eine vollständige Nachvollziehbarkeit jedes einzelnen Prozessschritts und ermöglicht es, den Ablauf der Pipeline präzise zu überwachen und zu analysieren.

3.2 Gradle

Für dieses Projekt wurde Gradle, das führende Build-System für die JVM, gewählt. Mit Gradle soll die Software für die gängigen Betriebssysteme Windows, macOS und Linux Binaries erstellen. Um dieses Ziel zu erreichen, soll der Build-Prozess automatisiert werden. Dazu ist zunächst jedoch eine Umstellung der Anwendung erforderlich. Die Konfigurationen für Gradle werden in der *build.gradle* Datei vorgenommen, welche die Konfiguration als Skript für Gradle enthält. Die notwendigen Schritte werden in diesem Abschnitt erläutert.

3.2.1 Abhängigkeiten

In der ursprünglichen Java-Anwendung wurden Abhängigkeiten in einem spezifischen Verzeichnis abgelegt, welches die benötigten Pakete im JAR-Format enthielt. Dieser Ansatz erforderte, dass bei der Bereitstellung einer neuen Version eines Pakets das bestehende Paket manuell durch die neue Version ersetzt werden musste. Um diesen Prozess zu optimieren und zu automatisieren, wurde Gradle als Build-Tool eingeführt.

Gradle ermöglicht es, Abhängigkeiten automatisiert aus einem oder mehreren *Package-Repositories* in den gewünschten Versionen herunterzuladen und in das Projekt zu integrieren. Dies wird durch die Definition der Abhängigkeiten im Gradle Build-Skript *build.gradle* realisiert, welches dann die erforderlichen Pakete auflöst und in das Projekt einbindet. Dies ist in Abbildung 3.1 exemplarisch abgebildet.

```
1 // FQLite Abhängigkeiten in Gradle
2 dependencies {
3     implementation group: 'org.antlr', name: 'antlr4', version: '4.8'
4     implementation group: 'org.apache.avro', name: 'avro', version: '1.11.1'
5     implementation group: 'nl.pvanassen', name: 'bplist', version: '1.0.0'
6     implementation group: 'org.mongodb', name: 'bson', version: '3.6.0'
7     ...
8 }
```

Quelltext 3.1: Ausschnitt des *dependencies*-Block aus build.gradle

Die meisten Abhängigkeiten konnten aus dem *MavenCentral* Repository bezogen werden, einem weit verbreiteten Repository für Java-Bibliotheken. Ausnahmen bildeten jedoch *FxTextEditor*, *SerializationDumper* sowie *JavaFX-swt*:

- *FxTextEditor*: Diese Bibliothek ist auf *GitHub* verfügbar (`https://github.com/andy-goryachev/FxTextEditor/`), wurde jedoch vom Autor nur als Quellcode veröffentlicht. Daher musste sie lokal gebaut werden, um als JAR-Datei im Projekt verwendet zu werden.
- *SerializationDumper*: Diese Bibliothek ist ebenfalls auf GitHub verfügbar (`https://github.com/NickstaDB/SerializationDumper`), jedoch nicht in einem öffentlichen Repository. Der Quellcode musste lokal heruntergeladen und gebaut werden.
- *JavaFX-swt*: Diese Bibliothek erweitert JavaFX, um *Standard Widget Toolkit* (SWT) Komponenten unter JavaFX zu nutzen. SWT ist ein GUI-Toolkit, das ursprünglich von IBM entwickelt wurde und häufig in der Eclipse IDE verwendet wird.

Um auch diese Abhängigkeiten zu integrieren, ohne dass diese in einem öffentlichen Repository zu finden sind, werden diese Abhängigkeiten als Dateien an Gradle übergeben. In Quelltext 3.2 ist dargestellt, wie diese drei Dateien ebenfalls in den *dependencies*-Block übergeben werden.

```
1 dependencies {
2     ...
3     // Lokal gespeicherte Abhängigkeiten
4     implementation files('lib/FxTextEditor.jar')
5     implementation files('lib/SerializationDumper-v1.13.jar')
6     implementation files('lib/javafx-swt.jar')
7 }
```

Quelltext 3.2: Ausschnitt des *dependencies*-Block mit Dateien als Abhängigkeiten

Im Rahmen dieser Umstellung wurde unter anderem das Plugin *org.openjfx.javafxplugin* verwendet. Gradle-Plugins erweitern die Funktionalität des Build-Tools und ermöglichen spezifische Aufgaben. Das JavaFX-Plugin ermöglicht den Download und die Integration der JavaFX-Abhängigkeiten in der gewünschten Version für die benötigte Plattform. JavaFX ist eine Softwareplattform zur Erstellung und Bereitstellung von Desktop-Anwendungen mit einer umfangreichen grafischen Benutzeroberfläche.

Neben dem JavaFX-Plugin wurde auch das *Badass Runtime*-Plugin (*org.beryx.runtime*) hinzugefügt. Dieses ermöglicht das Erstellen von benutzerdefinierten Laufzeitumgebungen (*runtime images*) mittels der *jpackage*-Anwendung [48]. Ein Ausschnitt dieses Plugin-Blocks ist in Quelltext 3.3 dargestellt.

Zusätzlich wurde mit *de.undercouch.download* ein Plugin eingeführt, um Dateien automatisiert herunterzuladen. Dies ist nötig, um den für die Anwendung nötigen Protobuf-Compiler als Binärdatei für die jeweiligen Systeme herunterzuladen und in einem Verzeichnis abzulegen, was in Kapitel 3.2.3 beschrieben wird. Diese werden durch die Anwendung benötigt und sind nicht über MavenCentral zu beziehen.

```
1     // Genutzte Plugins für Gradle
2     plugins {
3         id 'java'
4         id 'java-library'
5         id 'application'
6         id 'org.openjfx.javafxplugin' version '0.1.0'
7         id 'org.beryx.runtime' version '1.13.1'
8         id 'de.undercouch.download' version '5.6.0'
9 }
```

Quelltext 3.3: Ausschnitt des *plugins*-Blocks aus *build.gradle*

Um mittels *jpackage* eine Version für Windows bereitzustellen, benötigt es zudem *WiX 3.0* oder neuer, wie es in dem offiziellen User Guide beschrieben ist [49]. Das *WiX Toolset* besteht aus Werkzeugen, um Windows Installationspakete zu erstellen. Zu diesem Zweck wurden die benötigten Dateien in dem Projektverzeichnis in einem eigenen Ordner bereitgestellt, sodass diese während der Erzeugung der Pakete in der CI/CD-Pipeline zur Verfügung stehen.

Mit diesen Änderungen wurden die meisten Abhängigkeiten der Anwendung durch Gradle abstrahiert und sind durch einfache Änderungen in der *build.gradle* Datei änderbar.

3.2.2 Plattformspezifische Abbilder

In der Gradle-Konfigurationsdatei *build.gradle* lässt sich *jlink* und *jpackage* konfigurieren. Dafür bringt das *Badass Runtime*-Plugin die notwendigen Erweiterungen mit. In dem dafür vorgesehenen *runtime*-Block lassen sich Parameter für *jlink* festlegen. Mit diesen Optionen lässt sich die Ausgabe von *jlink* verändern. Hier ist das Ziel, dass die Ausgabe eine möglichst kleine Größe hat und nicht notwendige Dateien nicht erstellt werden. Die genutzten Parameter sind in Quelltext 3.4 dargestellt. [50]

```
1     runtime {
2         options = ['--strip-debug', '--compress', '2', '--no-header-files', '--no-man-pages']
3         ...
```

Quelltext 3.4: Ausschnitt des *runtime*-Block mit jlink Optionen aus der *build.gradle*

Die Optionen haben folgende Bedeutungen:

- *strip-debug*: Diese Anweisung veranlasst *jlink*, Debug-Informationen aus der Ausgabe zu entfernen, um die Dateigröße zu reduzieren.

- *compress*: Diese Option ermöglicht die Kompression von Ressourcen. Der Wert „0“ steht dabei für keine Kompression, während der Wert „2“ die Ausgabe in ein Zip-Archiv komprimiert.
- *no-header-files*: Durch das Entfernen der Header-Dateien wird die Größe der Ausgabe weiter reduziert.
- *no-man-pages*: Diese Option entfernt Dokumentationsdateien aus der Ausgabe, um die Dateigröße zu verringern.

Um plattformspezifische Abbilder zu erstellen, ist es notwendig, für jede Plattform die entsprechenden JDKs zu verwenden. Das Werkzeug *jlink* bietet hierzu im *targetPlatform*-Block die Möglichkeit, für jede Plattform ein spezifisches JDK anzugeben. Bei jedem Build wird über die angegebene URL das passende JDK-Archiv heruntergeladen und als *jdkHome* festgelegt, da in einem neuen *Runner* kein JDK vorinstalliert ist. Um eine neue Version des JDK zu nutzen, genügt es daher, die URL zur passenden Version und Plattform auszutauschen. [48]

In der Gradle-Konfigurationsdatei kann eine neue Variable „platformOverride“ eingeführt werden, um das Erstellen eines Abbildes für eine spezifische Plattform als Parameter zu übergeben. Diese Variable ermöglicht es, den entsprechenden *targetPlatform*-Block aufzurufen, um das benötigte JDK bereitzustellen. Die Bereitstellung des *jdkHome* ist in Abbildung 3.5 dargestellt. Die Funktion *targetPlatform* erhält dabei einen identifizierbaren Namen sowie die auszuführenden Aktionen als Parameter.

```
1     // Einstellungen abhängig von Zielplattform
2     if (platformOverride == "lin") {
3         targetPlatform("lin") {
4             jdkHome = jdkDownload("https://download.java.net/java/GA/jdk20.0.2/6
                  ↪ e380f22cbe7469fa75fb448bd903d8e/9/GPL/openjdk-20.0.2_linux-x64_bin.tar.gz")
5         }
6     }
7     else if (platformOverride == "mac") {
8         targetPlatform("mac") {
9             jdkHome = jdkDownload("https://download.java.net/java/GA/jdk20.0.2/6
                  ↪ e380f22cbe7469fa75fb448bd903d8e/9/GPL/openjdk-20.0.2_macos-x64_bin.tar.gz")
10        }
11    } else if (platformOverride == "win") {
12        targetPlatform("win") {
13            jdkHome = jdkDownload("https://download.java.net/java/GA/jdk20.0.2/6
                  ↪ e380f22cbe7469fa75fb448bd903d8e/9/GPL/openjdk-20.0.2_windows-x64_bin.zip")
14        }
15    }
```

Quelltext 3.5: Festlegen der targetPlatform mittels „platformOverride“

Die Erstellung plattformspezifischer, installierbarer Pakete kann durch den *jpackage*-Block gezielt beeinflusst werden. In Kombination mit der *targetPlatform*-Methode von *jlink* muss im *jpackage*-Block die Zielplattform durch die Variable *targetPlatformName* angegeben werden. Dies ist erforderlich, da *jpackage*, im Gegensatz zu *jlink*, keine Installationsprogramme für andere Plattformen als die ausführende erstellen kann und somit nur gefiltert wird, welche Ausgabe von *jlink* verwendet werden soll. [48]

Die Lösung besteht darin, unterschiedliche Betriebssysteme auf den Runnern zu verwenden, sodass die Plattformen Windows, Linux und macOS jeweils zum Erstellen der plattformspezifischen Abbilder genutzt werden können. Um dies zu erreichen, muss *targetPlatform* nicht explizit gesetzt werden,

sondern kann durch das ausführende System bestimmt werden. Voraussetzung hierfür ist die Bereitstellung von JDKs auf den jeweiligen Systemen. Diese Vorbereitung kann im *GitHub Actions* Workflow erfolgen und wird detailliert in Kapitel 3.3.3 beschrieben. [48]

Zur effizienten Verwaltung und Ausführung der Build-Prozesse auf verschiedenen Betriebssystemen werden im Workflow Matrizen verwendet. Diese Matrizen erlauben es, mehrere Auftragsausführungen parallel zu definieren, die sich nur durch ihre Variablen unterscheiden. Dadurch wird es möglich, den Workflow einmal für jedes definierte Betriebssystem auszuführen, ohne dass separate Workflow-Skripte notwendig sind. Die *strategy.matrix.os*-Option in *GitHub Actions* ermöglicht die dynamische Zuweisung des Betriebssystems für jeden *GitHub* Runner, wodurch die plattformspezifischen Builds parallelisiert und somit zeitlich optimiert werden. Durch die von *GitHub* bereitgestellten Runner kann die Anwendung nativ für die einzelnen Betriebssysteme erstellt werden, wodurch keine Virtualisierung notwendig ist.

Zur weiteren Konfiguration stellt das *Badass Runtime*-Plugin den *launcher*-Block zur Verfügung. Dieser Block ermöglicht das Setzen der Variable „noConsole“, die ausschließlich unter Windows verwendet wird. Diese Einstellung bestimmt, ob die Anwendung ein zugeordnetes Konsolenfenster startet. Dabei wird festgelegt, ob das Kommando „javaw“ anstelle von „java“ genutzt wird. In dem spezifischen Anwendungsfall wurde dieser Parameter auf „true“ gesetzt, wodurch die Anwendung ohne Konsolenfenster gestartet wird. [48]

jpackage bietet weitere Optionen an, um dem Installationsprogramm Parameter zu übergeben, darunter der Paketname, die Option ein Installationsverzeichnis zu wählen, oder ein Icon zu setzen. Dies kann mittels des *Badass Runtime*-Plugin durch die Option „installerOptions“ festgelegt werden. Die genutzten Parameter können je nach Plattform variieren, die genutzten Optionen sind in Quelltext 3.6 dargestellt.

```
1   jpackage {
2       // Aktuelles Betriebssystem
3       def currentOs = org.gradle.internal.os.OperatingSystem.current()
4       installerOptions += ['--resource-dir', "resources"]
5
6       // Einstellungen abhängig von Zielplattform
7       if (currentOs.windows) {
8           installerOptions += ['--win-per-user-install', '--win-dir-chooser', '--win-menu',
                ↪ '--win-shortcut']
9       } else if (currentOs.linux) {
10          installerOptions += ['--linux-package-name', 'fqlite', '--linux-shortcut']
11      } else if (currentOs.macOsX) {
12          installerOptions += ['--mac-package-name', 'fqlite']
13      }
14  }
```

Quelltext 3.6: Festlegen der „installerOptions“

3.2.3 Bereitstellen des Protobuf-Compilers

Protocol Buffers (Protobuf) ist ein von Google entwickeltes Framework zur plattformunabhängigen Serialisierung von Daten. Dabei wird eine definierte Datenstruktur verwendet, um Daten effizient zu serialisieren und zu deserialisieren. In der vorliegenden Anwendung wird der Protobuf-Compiler *protoc* im Verzeichnis */protoc* innerhalb des Projekts erwartet. Da der Compiler je nach Betriebssystem

variiert, muss dieser spezifisch für das Zielsystem heruntergeladen werden. Um diesen Prozess zu automatisieren, wird das Gradle-Plugin *de.undercouch.download* verwendet. Dieses Plugin ermöglicht es, eine Gradle-Task vom Typ *Download* zu erstellen, die den Download durchführt. Hierbei werden zwei wesentliche Parameter benötigt: *src*, die URL der Datei, und *dest*, der Speicherort auf dem lokalen System, an dem die Datei abgelegt werden soll. [51]

Die verfügbaren Dateien für den Protobuf-Compiler folgen einer festgelegten Namenskonvention, die es ermöglicht, die Bezeichnung (*src*) der benötigten Datei programmatisch zu erzeugen und den entsprechenden Download zu starten. Die Namenskonvention lautet: *protoc-$VERSION-$OS.zip*. Da Gradle in der Lage ist, alle notwendigen Informationen über das ausführende System zu beziehen, kann der Dateiname direkt innerhalb des Gradle-Build-Skripts, wie in Quelltext 3.7 dargestellt, dynamisch erstellt werden.

```
1    // Herunterladen des Protoc-Compiler
2    task downloadProtoc(type: Download) {
3        def currentOs = org.gradle.internal.os.OperatingSystem.current()
4        def protocOsName = ""
5        if (currentOs.windows) {
6            protocOsName = "win64"
7        } else if (currentOs.linux) {
8            protocOsName = "linux-x86_64"
9        } else if (currentOs.macOsX) {
10           protocOsName = "osx-universal_binary"
11       }
12
13       // String für die Protoc-Binary wird zusammengestellt
14       var fileName = "protoc-${protocversion}-${protocOsName}.zip"
15       var protocUrl = "https://github.com/protocolbuffers/protobuf/releases/download/v${
             ↪ protocversion}/" + fileName
16       src protocUrl
17       dest new File(temporaryDir, fileName)
18   }
19
20   // Heruntergeladene Dateien aus dem zip-Archiv entpacken
21   task downloadAndUnzipProtoc(dependsOn: downloadProtoc, type: Copy) {
22       from zipTree(downloadProtoc.dest)
23       into layout.buildDirectory.dir('protoc')
24   }
25
26   // Die Aufgabe in der Ausführungsreihenfolge vor den jpackage Befehl einbringen
27   tasks.named('jpackage') {
28       dependsOn downloadAndUnzipProtoc
29   }
```

Quelltext 3.7: Herunterladen des Protobuf-Compilers

Zur Implementierung wurde in Gradle eine neue Aufgabe namens *downloadAndUnzipProtoc* angelegt, die vor der Ausführung der *jpackage*-Task durchgeführt wird. Dies wurde erreicht, indem *jpackage* von der erfolgreichen Ausführung der *downloadAndUnzipProtoc*-Aufgabe abhängig gemacht wurde. Auf diese Weise wird sichergestellt, dass der Download des Protobuf-Compilers stets vor der Paketierung der Anwendung erfolgt und alle notwendigen Dateien rechtzeitig zur Verfügung stehen.

```
1   tasks.jpackage.doFirst {
2       // Pfad zum jpackage-Verzeichnis im Build-Ordner
3       def fqliteJpackageDirectory = "build/jpackage/"
4       // Plattformspezifische Unterverzeichnisse
5       if (currentOs.windows) {
6           fqliteJpackageDirectory += "fqlite"
7       } else if (currentOs.linux) {
8           fqliteJpackageDirectory += "fqlite/lib"
9       } else if (currentOs.macOsX) {
10          fqliteJpackageDirectory += "fqlite.app/Contents/MacOS"
11      }
12      // Kopieren des Protoc-Verzeichnisses in das jpackage-Verzeichnis
13      copy {
14          from layout.buildDirectory.dir('protoc')
15          into "$fqliteJpackageDirectory/protoc"
16      }
17      // Kopieren der plattformspezifischen Dateien zum Aufruf von Protoc
18      copy {
19          from layout.projectDirectory
20          include 'proto.run', 'proto.sh', 'protoc.bat'
21          into "$fqliteJpackageDirectory"
22      }
23  }
```

Quelltext 3.8: Bereitstellen der Protobuf-Dateien in der Ausgabedatei

Um die notwendigen Dateien aus dem Build-Verzeichnis in die Ausgabe von *jpackage* zu integrieren, müssen diese in das plattformspezifische Verzeichnis innerhalb des *jpackage*-Ausgabeordners übertragen werden. Hierfür wurde im Gradle-Skript eine Anweisung mithilfe von *doFirst* in die *jpackage*-Task eingebunden, die vor der eigentlichen Ausführung von *jpackage* durchgeführt wird. Diese Anweisung bestimmt das plattformspezifische Zielverzeichnis innerhalb des *jpackage*-Ordners im Build-Verzeichnis und kopiert die in der *downloadAndUnzipProtoc*-Aufgabe heruntergeladenen Protobuf-Dateien in einen Unterordner */protoc*. Zusätzlich werden die plattformspezifischen Dateien (*proto.run*, *proto.sh*, *protoc.bat*) in das entsprechende Verzeichnis kopiert, um den Protobuf-Compiler mit den richtigen Parametern aufrufen zu können. Der Gradle-Konfigurationscode für diese Implementierung ist in Quelltext 3.8 zu finden.

3.3 GitHub

GitHub ist eine Plattform für Entwickler, um *Git* Repositories bereitzustellen. Zusätzlich dazu bietet *GitHub* Funktionen wie unter anderem Diskussionen, das Anlegen von Projektzielen und Tickets, das Hosten von statischen Webseiten (*GitHub* Pages) sowie Automatisierung mittels *GitHub Actions*. [52]

3.3.1 GitHub Actions

Die Automatisierung in *GitHub* wird durch sogenannte *GitHub Actions* ermöglicht. Eine einzelne *Action* wird als Auftrag (*job*) bezeichnet, und mehrere dieser Aufträge werden in *Workflows* zusammengefasst. Ein *Workflow* kann dabei so konfiguriert werden, dass die enthaltenen Aufträge entweder sequenziell oder parallel ausgeführt werden. Es ist möglich, für ein Repository mehrere

Workflows zu erstellen, die unterschiedlichen Zwecken dienen. Diese *Workflows* sind stets im Verzeichnis *.github/workflows* im Stammverzeichnis des Projekts abgelegt. Die Wiederverwendbarkeit von *Workflows* ist eine zentrale Funktion, die es ermöglicht, Verweise auf bestehende *Workflows* zu erstellen und diese in anderen Projekten zu nutzen. [53]

Die Ausführung eines *Workflows* wird durch ein Ereignis (*event*) initiiert. Zu den möglichen Ereignissen zählen unter anderem das Pushen von neuem Code auf einen Branch, das Erstellen von Pull-Requests oder das Zusammenführen von Branches. GitHub stellt eine umfassende Übersicht über alle verfügbaren Ereignisse bereit. [54]

GitHub Actions bieten eine flexible Möglichkeit zur Automatisierung von Softwareentwicklungsprozessen. Durch die Integration von *Workflows* in den Entwicklungszyklus können wiederkehrende Aufgaben automatisiert und somit die Effizienz des Entwicklungsprozesses gesteigert werden. Die Möglichkeit zur Wiederverwendung und Verknüpfung von *Workflows* trägt zudem zur Modularität und Wartbarkeit der Automatisierungsprozesse bei.

3.3.2 GitHub Runner

Ein Runner ist eine virtuelle Maschine, die dazu dient, Aufträge aus *Workflows* auszuführen. GitHub stellt hierfür eigene virtuelle Maschinen bereit, die kostenfrei in öffentlichen Repositories verwendet werden können. In privaten Repositories steht eine begrenzte Anzahl von Minuten für die Nutzung dieser Runner kostenfrei zur Verfügung. Überschreitet man dieses Limit, fallen zusätzliche Kosten an. Alternativ besteht die Möglichkeit, eigene Runner bereitzustellen. Dies kann beispielsweise mittels Docker realisiert werden. [55]

Ein wesentlicher Vorteil der von *GitHub* bereitgestellten Runner liegt in ihrer hohen Verfügbarkeit und der minimalen Wartung, die sie für Entwickler erfordern. Die gesamte Hardware-Wartung sowie die Bereitstellung von Sicherheitsupdates und neuen Softwareversionen werden vollständig von *GitHub* übernommen. Dies reduziert den administrativen Aufwand für Entwickler erheblich, da sie sich nicht um die Infrastrukturpflege kümmern müssen. Jeder *GitHub*-Runner entspricht einer neuen virtuellen Maschine, auf der die Runner-Anwendung sowie eine Vielzahl von gängigen Entwicklungswerkzeugen und Bibliotheken bereits vorinstalliert sind.

Diese Runner können auf verschiedenen Betriebssystemen betrieben werden, einschließlich Ubuntu (Linux), Windows und macOS, wobei für jedes Betriebssystem unterschiedliche Versionen zur Verfügung stehen. Besonders unter macOS bietet die Möglichkeit, zwischen verschiedenen Prozessorarchitekturen zu wählen – wie Intel (x86_64) und Apple M1 (arm64) – erhebliche Vorteile, insbesondere im Hinblick auf die Optimierung und Kompatibilität der Builds. Durch diese Flexibilität können Entwickler sicherstellen, dass ihre Anwendungen für eine breite Palette von Systemen und Architekturen kompiliert werden - ohne die Notwendigkeit, eine eigene Entwicklungsumgebung für jedes Zielsystem zu pflegen. [55]

Die Flexibilität und die breite Unterstützung unterschiedlicher Betriebssysteme und Architekturen in der CI/CD-Pipeline ermöglichen, dass Anwendungen unter verschiedenen Bedingungen getestet und bereitgestellt werden, ohne die sonst dazu nötige Hardware beschaffen zu müssen.

Zusätzlich zur Verwendung der von *GitHub* bereitgestellten Runner haben Entwickler die Möglichkeit, eigene Runner in ihre Projekte zu integrieren. Diese eigenen Runner können auf nahezu jeder Hardware laufen und bieten so die Möglichkeit, speziell angepasste Umgebungen bereitzustellen, die den spezifischen Anforderungen eines Projekts entsprechen. *GitHub* stellt in seiner Dokumentation klare Mindestanforderungen für diese selbstverwalteten Runner bereit. So muss beispielsweise unter Windows mindestens Windows 10 verwendet werden, unter Linux wird mindestens Debian 10 benötigt, und für macOS ist mindestens die Version 11.0 (Big Sur) erforderlich. [55]

3.3.3 Github Actions Workflow

Um die Anwendung nach Quellcodeänderungen neu zu erstellen, lässt sich mittels *GitHub Actions* ein Workflow definieren, welcher die Anwendung automatisiert baut und die gebaute Anwendung zu einem Ziel hochläd. Da die Anwendung auf dem *master*- oder *main*-Branch immer einen funktionierenden Stand haben sollte, lässt sich ein Trigger nutzen, welcher bei einem Pull-Request ausgelöst wird. Eine Liste mit Triggern für einen Workflow lässt sich in in einem Workflow mit dem Schlüsselwort „on“ festlegen. Sofern ein Event ausgelöst wird, welches als Trigger definiert wird, werden die Jobs des Triggers ausgeführt. [56]

Um die Anwendung nach Quellcodeänderungen automatisch neu zu erstellen, kann ein Workflow mittels *GitHub Actions* definiert werden, der die Anwendung baut und die resultierende ausführbare Datei an ein definiertes Ziel hochlädt. Da der *master*- oder *main*-Branch stets einen funktionsfähigen Zustand der Anwendung darstellen sollte, empfiehlt sich die Verwendung eines Triggers, der bei einem Pull-Request aktiviert wird. Ein Pull-Request dient dabei als Mechanismus, um neue Quellcodeänderungen in das Repository einzuführen. Dabei kann sichergestellt werden, dass nur geprüfte und funktionsfähige Codeänderungen in den *master*- oder *main*-Branch integriert werden. Sobald ein Ereignis eintritt, das als Trigger definiert wurde, werden die entsprechenden Jobs des Workflows ausgeführt. [56]

3.3.3.1 Multi-Plattform Build

Die Bereitstellung der Anwendung für unterschiedliche Betriebssysteme erfordert die Erstellung plattformspezifischer Abbilder. Wie bereits in Kapitel 3.2.2 erläutert, wird dies durch die Verarbeitung der Anwendung auf dem jeweiligen System mittels *jpackage* ermöglicht. *GitHub Actions* bietet die Möglichkeit, durch den Einsatz von Matrizen automatisch mehrere Auftragsausführungen zu erstellen, die sich nur durch ihre Variablen unterscheiden. Für die Erstellung plattformspezifischer Abbilder kann jede Ausführung auf einem anderen Betriebssystem durchgeführt werden. Dies erlaubt auch die Verwendung verschiedener Betriebssystemversionen oder Architekturen, wobei maximal 256 Aufträge pro Workflowausführung unterstützt werden. [57]

Um Anwendungen für die Plattformen Windows, macOS und Linux bereitzustellen, kann das Betriebssystem des *GitHub* Runners über die *strategy.matrix.os*-Option definiert werden. Mithilfe dieser Matrix kann die *runs-on*-Option gesetzt werden, welche das Betriebssystem des Runners bestimmt. Dadurch lässt sich der Workflow für jedes definierte Betriebssystem einmal ausführen. Die genutzte Konfiguration des Workflows kann in Quelltext 3.9 eingesehen werden. Genutzt werden aktuelle

Abbilder für Windows und Ubuntu Linux, sowie macOS in der Version 13, welche die Intel-Architektur *x86_64* nutzt. Zusätzlich wird macOS in der Version 14 verwendet, welches mit der M1-Architektur *arm64* arbeitet. [57]

```
1    # Job zum bauen der Anwendung
2    build:
3        name: Gradle Build ${{ matrix.os }}
4        # Wird auf jedem Betriebssystem welches in matrix definiert ist ausgeführt
5        runs-on: ${{ matrix.os }}
6        strategy:
7            matrix:
8                os: [ ubuntu-latest, macos-13, macos-14, windows-latest ]
9            fail-fast: false
10   ...
```

Quelltext 3.9: *GitHub Actions* Workflow

Um die Anwendung für das jeweilige Betriebssystem zu bauen, ist es notwendig, je nach Betriebssystem unterschiedliche Befehle auszuführen oder Variablen zu setzen. Daher wird in den abweichenden Schritten überprüft, welches Betriebssystem aktuell ausgeführt wird. Abhängig vom Betriebssystem entstehen nach der Ausführung von *jpackage* unterschiedliche Dateitypen. Während bei macOS ein *.dmg*-Dateityp erzeugt wird, resultiert die Ausführung unter Windows in einer *.exe*-Datei. Um diese Unterschiede beim Auffinden der Datei zum Hochladen zu berücksichtigen, werden in den anwendungsspezifischen Bereitstellungsprozessen Umgebungsvariablen gesetzt, welche den aktuellen Namen der Datei inklusive Dateityperweiterung enthalten. Beispielhaft wird dies in Quelltext 3.10 dargestellt.

```
1    ...
2    - name: Create MacOS file name
3    if: startsWith(matrix.os, 'mac')
4    run: |
5      # Speichern des Dateinamens als Umgebungsvariable
6      echo "OUTPUT_FILE_NAME=fqlite-${{ needs.get-version.outputs.version }}-macOS-$(uname -m)
         ↪ .dmg" >> $GITHUB_ENV
7    - name: Create MacOS installer
8    if: startsWith(matrix.os, 'mac')
9    run: |
10     # Bauen der Anwendung unter macOS
11     ./gradlew --info --stacktrace jpackage
12     find build/jpackage -name "*.dmg" -exec sh -c 'x="{}"; mv "$x" "${{ env.
         ↪ OUTPUT_FILE_NAME }}"' \;
13     # Speichern des Dateipfads als Umgebungvariable
14     echo "OUTPUT_FILE=$(realpath ${{ env.OUTPUT_FILE_NAME }})" >> $GITHUB_ENV
15   ...
```

Quelltext 3.10: *GitHub Action* zum Erstellen der MacOS Anwendung

Diese Schritte werden nur auf einem Runner mit dem Betriebssystem macOS ausgeführt, wobei es für die anderen Betriebssysteme ähnliche Schritte gibt. Bei der Ausführung wird zunächst der Dateiname des entstehenden Artefakts als Umgebungsvariable angelegt und persistiert. Der Dateiname setzt sich aus dem Anwendungsnamen „fqlite“, gefolgt von der Anwendungsversion, welche zuvor aus der Gradle-Konfigurationsdatei entnommen wurde, der Plattform, für welche die Datei erstellt wurde, und der Architektur des Prozessors zusammen.

Daraufhin wird das Artefakt mittels *jpackage* erstellt. Anschließend wird der Pfad zu diesem Artefakt mittels „realpath“ als Umgebungsvariable erstellt und im Speicher der Umgebungsvariablen persistiert, um die Datei später präzise zu finden.

3.3.4 Integritätsprüfung

Die Integrität der Artefakte sollte unabhängig überprüfbar sein, um Dritten zu ermöglichen, sicherzustellen, dass die heruntergeladene Anwendung tatsächlich die vom Autor bereitgestellte ist. Um die Quelle zurückzuverfolgen, braucht es überprüfbare Informationen über das Artefakt, wo, wann und wie etwas produziert wurde [58]. Bei Supply Chain Levels for Software Artifacts (SLSA) handelt es sich um eine Sammlung von Standards und Kontrollen zur Verhinderung von Manipulation, Verbesserung der Integrität und Sicherung von Paketen. [59]

GitHub bietet hierzu mit der Action *attest-build-provenance* einen Verarbeitungsschritt an, der für ein Artefakt eine überprüfbare, kryptografische Signatur anhand seines Hashwerts erstellt und diese in einer öffentlichen Instanz speichert [60]. Mit dem *GitHub* Command-Line Interface (CLI) *gh* lässt sich die Integrität überprüfen, indem der Befehl *gh attestation verify <file> [–repo | –owner]* verwendet wird. Neben der Datei muss dabei der Parameter *–repo <repository>* oder *–owner <owner>* angegeben werden. Dies ist exemplarisch in Quelltext 3.11 dargestellt. Die Ausgabe des *GitHub* CLI-Tools zeigt, dass die Signatur aus der öffentlichen Instanz heruntergeladen und überprüft wurde. Es wird genau angegeben, während welcher Workflow-Ausführung diese Signatur entstanden ist. Im Beispiel wurde der Workflow *build-action* während des Pull-Requests mit der ID 7 ausgeführt. Dadurch wird sichergestellt, dass die heruntergeladene Datei diesem Repository entspricht und nicht manipuliert wurde. [61]

```
1  $> gh attestation verify fqlite-1.0.0.exe --repo bocian67/fqlite
2  Loaded digest sha256:eb378a921dcb0da3d54ada21b4673f155468a0fe8cba59dbdfc0012f4e9ef02a for
       ↪ file://fqlite-1.0.0.exe
3  Loaded 1 attestation from GitHub API
4  Verification succeeded!
5
6  sha256:eb378a921dcb0da3d54ada21b4673f155468a0fe8cba59dbdfc0012f4e9ef02a was attested by:
7  REPO PREDICATE_TYPE WORKFLOW
8  bocian67/fqlite https://slsa.dev/provenance/v1 .github/workflows/build-action.yml@refs/
       ↪ pull/7/merge
```

Quelltext 3.11: Verifizieren der Integrität der heruntergeladenen Anwendung

3.3.5 Bereitstellung einer neuen Version

Bei der Erstellung einer neuen Version sollen alle Artefakte gesammelt veröffentlicht werden, sodass diese heruntergeladen werden können. Dazu bietet *Git Tags* und *GitHub Releases* an.

Git Tags sind Verweise oder Markierungen auf einen bestimmten Commit im Repository. Diese *Tags* werden als Objekte in der *Git*-Datenbank gespeichert und häufig verwendet, um Versionen der Software eindeutig festzulegen. Anders als ein *Branch* ist ein *Tag* unveränderlich, da dieser immer auf einen spezifischen Commit zeigt. Ein *Tag* kann allerdings gelöscht und neu erstellt werden, wodurch dieser auf einen anderen Commit zeigen kann. [62]

GitHub erweitert die Funktionalität von *Git Tags* durch die Bereitstellung von *Releases*. Ein *Release* basiert auf einem *Git Tag* und ermöglicht das Hinzufügen von Beschreibungen sowie die Bereitstellung von zusätzlichen Ressourcen. *GitHub* generiert automatisch Links zum Herunterladen einer ZIP-Datei (*.zip*) und eines Tarballs (*.tar.gz*), welche den Inhalt des Repositories zum Zeitpunkt des *Tags* enthalten. Darüber hinaus können weitere Artefakte hinzugefügt werden, beispielsweise die kompilierten Versionen der Software für verschiedene Plattformen [63].

Um einen aktuellen *Release* automatisch zu generieren, sollte dieser immer dann erstellt werden, wenn ein Release-Branch mit dem aktuell veröffentlichten Stand zusammengeführt wird. Dazu können innerhalb des *GitHub* Workflows verschiedene Auslöser definiert werden. In diesem Prozess werden die Auslöser *closed*, *synchronize* und *opened* genutzt. Dabei wird die Pipeline immer ausgeführt, wenn ein Pull-Request erstellt, aktualisiert oder geschlossen wird. Dies hat zur Folge, dass bei jeder Änderung die Anwendung als Artefakt erstellt wird und zu den *GitHub* Artefakten, einem Zwischenspeicher, hochgeladen wird. Damit kann die Anwendung getestet werden, bevor dafür ein *Release* erstellt wird. Ein Beispiel dieser Konfiguration ist in Quelltext 3.12 dargestellt.

```
1  on:
2  pull_request:
3    branches:
4      - 'master'
5    types:
6      - closed
7      - synchronize
8      - opened
```

Quelltext 3.12: Bedingung zur Ausführung des Workflows bei Aktualisierung eines Pull-Requests

Um die Version zu ermitteln, welche für den *Tag* und *Release* sowie den Dateinamen der entstehenden Artefakte genutzt wird, wird aus der Gradle-Konfigurationsdatei der Wert der Variable „version“ mittels *awk* ausgelesen und als *GitHub* Ausgabevariable abgelegt. Damit kann diese Variable von anderen Workflow-Schritten, unabhängig des Betriebssystems referenziert werden. Dieser Schritt, welcher in Quelltext 3.13 dargestellt ist, wird am Anfang jeder Workflowausführung durchgeführt.

```
1  get-version:
2    name: Get project version
3    outputs:
4      version: ${{ steps.gradle-version.outputs.version }}
5    runs-on: ubuntu-latest
6    steps:
7      - name: Checkout sources
8        uses: actions/checkout@v4
9      - name: Get project version
10       id: gradle-version
11       run: |
12         # Speichern der Anwendungsversion als Ausgabevariable des Build-Schritts
13         echo "version=$(gradle properties -q | awk '/^version:/ {print $2}')" >> "
               ↪ $GITHUB_OUTPUT"
```

Quelltext 3.13: Beziehen der Versionsnummer aus dem Namen des Branches

Um bei einem Merge automatisch einen *Release* mit den erstellten Artefakten zu generieren, muss die Bedingung des Merges explizit im *if*-Block des Jobs definiert werden. Dies ist notwendig, da ein Pull-Request auch ohne Merge geschlossen werden kann. Um sicherzustellen, dass der Workflow

nur dann ausgeführt wird, wenn ein Pull-Request tatsächlich gemerged wurde, muss der Trigger *pull_request* durch die Aktion *closed* eingeschränkt werden. Im *if*-Block wird anschließend geprüft, ob der Merge tatsächlich stattgefunden hat. Diese Konfiguration gewährleistet, dass ein neuer *Release* nur dann erzeugt wird, wenn alle relevanten Bedingungen erfüllt sind. Die detaillierte Umsetzung dieses Schritts ist in Quelltext 3.14 dargestellt.

```
1     publish:
2       # Führe den Schritt nur aus, wenn Pull-Request gemerged wurde
3       if: github.event.pull_request.merged == true
4       name: Publish release artefacts
5       runs-on: ubuntu-latest
6       # Jobs, welche zuvor ausgeführt werden müssen
7       needs:
8         - build
9         - get-version
```

Quelltext 3.14: Bedingung zur Ausführung des Jobs nach Merge eines Branches

Das Veröffentlichen eines *Release* erfolgt in mehreren Schritten: Zunächst wird das Projekt auf einem neuen Runner ausgecheckt. Anschließend werden alle in diesem Durchlauf erstellten Artefakte in einem Ordner namens „artifacts/“ gesammelt. Mit diesen Dateien wird ein *GitHub Release* erstellt, wobei die aktuelle Version als Name des *Release* dient. Der Inhalt der Datei „Changelog.md“, einer im Projekt enthaltenen Markdown-Datei, wird als Ergänzungstext genutzt. Zusätzlich werden sämtliche Dateien aus dem Ordner „artifacts/“ in den *Release* hochgeladen. Abschließend erfolgt das Hochladen der Dateien auf einen FTP-Server mittels einer weiteren *GitHub Action*, sodass die Dateien auch über diesen Weg bereitgestellt werden können.

Um die Anmeldeinformationen für den FTP-Server zu sichern, werden diese in *GitHub* als *Action Secrets* hinterlegt. Diese *Action Secrets* sind spezielle Variablen, die den Runnern während der Ausführung des Workflows zur Verfügung stehen und durch ihre entsprechenden Werte ersetzt werden. Es ist möglich, für unterschiedliche Umgebungen verschiedene Werte für denselben Variablennamen zu definieren. Dadurch kann beispielsweise sowohl eine Entwicklungs- als auch eine Produktionsversion einer Anwendung erstellt werden, ohne den zugrunde liegenden Prozess anzupassen. Außerdem werden *Action Secrets* im Log der Pipeline durch drei Sterne (***) maskiert, um sicherzustellen, dass sensible Informationen nicht sichtbar sind. Ein Beispiel für die Konfigurationsseite der *GitHub Action Secrets* ist in Abbildung 3.1 dargestellt. Dort zu sehen sind vier Variablen, welche zum Verbindungsaufbau zum FTP-Server benötigt werden: *FTP_SERVER* als Adresse, *FTP_USERNAME* als Nutzerkennung, *FTP_PASSWORD* als Passwort und *FTP_UPLOAD_PATH* als Zielpfad für die hochzuladenen Dateien.

3.4 Pakete für Paketmanager

Für die Anwendung *FQLite* sollen Pakete für Paketmanager, darunter *Chocolatey* für Windows und *Homebrew* für macOS, erstellt werden. Die Implementierung dieser alternativen Installationsmethoden wird im Folgenden detailliert beschrieben, einschließlich der Konfigurationsschritte und der notwendigen Anpassungen an der CI/CD-Pipeline.

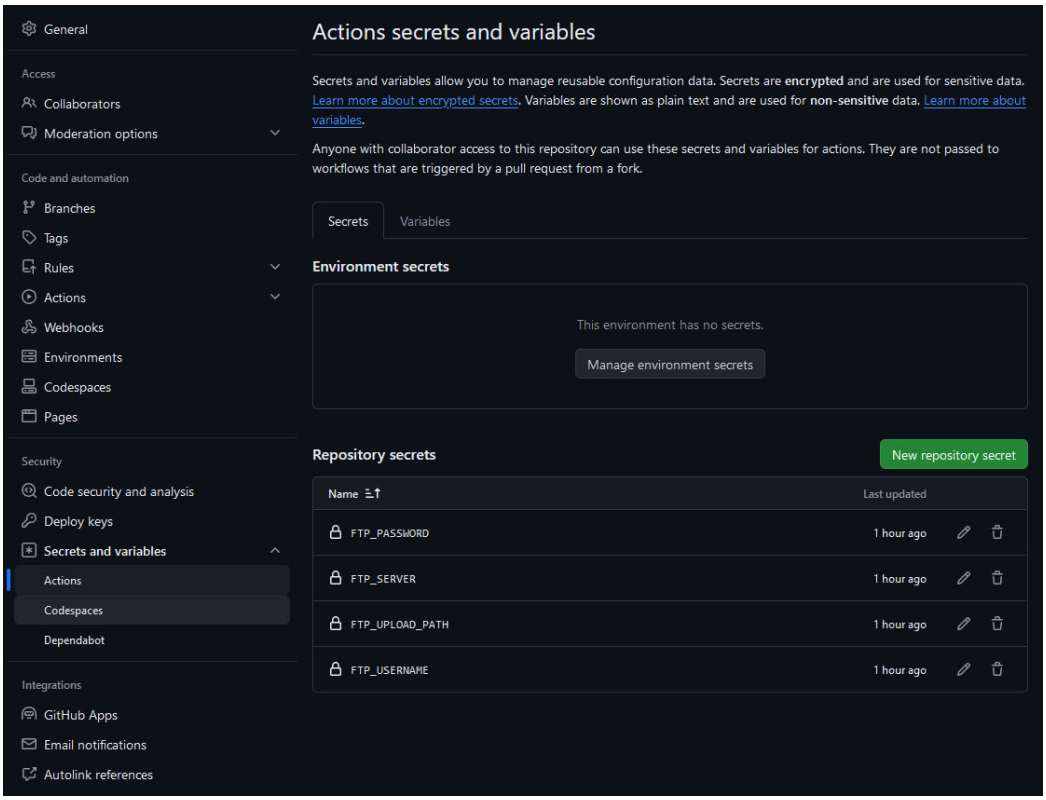

Abbildung 3.1: GitHub Action Secrets

3.4.1 Chocolatey

Ein *Chocolatey*-Paket stellt eine erweiterte Version eines NuGet-Pakets dar, das zusätzliche, *Chocolatey*-spezifische Metadaten enthält [64]. Wesentlicher Bestandteil ist das Paket-Manifest, die *.nuspec*-Datei, welche sämtliche Metadaten eines NuGet-Pakets umfasst. Diese Metadaten beinhalten unter anderem den Speicherort des Inhalts, die auszuführenden Aktionen zur Installation oder Deinstallation sowie die Versionsinformationen. Das finale NuGet-Paket von FQLite wird in Abbildung 3.2 mittels der Anwendung *NugetPackageExplorer* dargestellt, einer frei verfügbaren Anwendung zur Visualisierung von NuGet-Paketen. Der Inhalt eines NuGet-Pakets kann auch ohne Drittanbietersoftware untersucht werden, da das Paket lediglich ein Zip-Archiv darstellt und somit einfach entpackt werden kann, um Zugriff auf die enthaltenen Dateien zu erhalten.

Neben der *.nuspec*-Datei enthält das *Chocolatey*-Paket auch im Verzeichnis *Tools/* die Datei *LICENSE.txt*, die die aus *GitHub* übernommene Softwarelizenz beinhaltet. Darüber hinaus umfasst es das Installationsskript *chocolateyinstall.ps1*. Dieses Skript enthält einen Verweis auf die Download-URL des zuvor bereitgestellten Windows-Releases, den Dateityp sowie Parameter zur Installation. Für die Deinstallation ist keine zusätzliche Datei erforderlich, da der *auto uninstaller* von *Chocolatey* das Paket ohne weitere Konfiguration deinstallieren kann. Sollten weitere Schritte für die Deinstallation erforderlich sein, können diese in dem Skript *chocolateyuninstall.ps1* eingearbeitet werden, welches ähnlich zu dem Powershell-Installationsskript funktioniert.

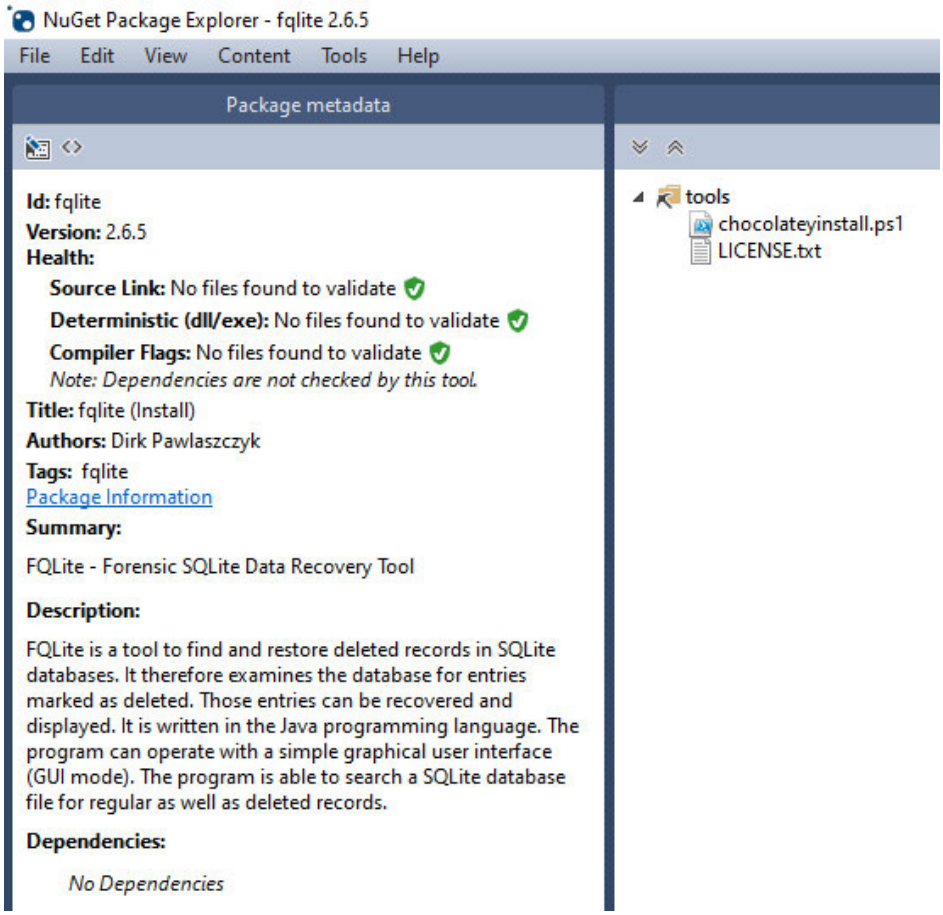

Abbildung 3.2: FQLite *Chocolatey* Paket im *NugetPackageExplorer*

Zur Bereitstellung des *Chocolatey*-Pakets wird dieses mithilfe des Befehls *choco pack fqlite.nuspec* gepackt. Dadurch entsteht das NuGet-Paket *fqlite.nupkg*. Anschließend muss dieses Paket lediglich bereitgestellt werden, sodass Nutzer es über *Chocolatey* installieren können. Beispielhaft sieht der Installationsbefehl dann wie in Quelltext 3.15 aus:

```
1    choco install fqlite --source fqlite.nupkg
```

Quelltext 3.15: Befehl zum installieren des FQLite *Chocolatey*-Pakets

3.4.2 Homebrew

Für *FQLite*, einer Anwendung mit grafischer Benutzeroberfläche, soll eine *Cask* erstellt werden, welche es den Benutzern ermöglicht, FQLite über *Homebrew* zu beziehen. In einer *Homebrew Cask* finden sich neben den Build-Schritten auch die Metadaten der Software sowie die Möglichkeit, die Software auch ohne lokalen Kompilierung zu beziehen. Zu den Metadaten zählt unter anderem die Projektseite, der Name sowie eine kurze Beschreibung. Da die Software bereits in gebauter Form durch den *GitHub Workflow* vorliegt, wird davon abgesehen erneut in *Homebrew* die Software zu bauen. Stattdessen wird mit dem Feld *url* ein Verweis auf den *Release* angegeben, durch welchen die Artefakte bereits zum Download angeboten werden. Zur Integritätssicherung wird der Hashwert des *Release* für die jeweilige Plattform festgelegt, sodass nur Anwendungen mit dem passenden Hashwert übernommen werden. Exemplarisch kann dies in Quelltext 3.16 eingesehen werden.

```
cask "fqlite" do
version "2.6.5"
arch arm: "arm64", intel: "x86_64"
sha256 arm: "4d9209b135f44fbac03f41ecbe7297403f38501d05986eb01ec15d20b21db58b",
       intel: "6838a7691fdfcb724b0d468f748b759d42cafb272e0c5172aa5a98fd1fa19349"
url "https://github.com/bocian67/fqlite/releases/download/#{version}/fqlite-#{version}-
    ↪ macOS-#{arch}.dmg"
name "fqlite"
desc "FQLite is a tool to find and restore deleted records in SQlite databases. It
    ↪ therefore examines the database for entries marked as deleted. Those entries can
    ↪ be recovered and displayed. It is written with the Java programming language. The
    ↪ program can operates with a simple graphical user interface (GUI mode). The
    ↪ program is able to search a SQLite database file for regular as well as deleted
    ↪ records."
homepage "https://www.staff.hs-mittweida.de/~pawlaszc/fqlite/"
app "fqlite.app"

def caveats
    <<~EOS
    Fqlite wird installiert. Um die Anwendung aus der Apple Quarantäne zu entfernen, führe
        ↪ folgendes aus:
       xattr -dr com.apple.quarantine /Applications/fqlite.app
    EOS
    end
end
```

Quelltext 3.16: *Homebrew Cask* für FQLite

Mit dem *arch*-Feld kann für die jeweilige Architektur ein Alias definiert werden, der der Architekturbezeichnung von *FQLite* entspricht. Dieser Wert kann, ebenso wie das Feld *version*, im Feld *url* verwendet werden, um die korrekte Bereitstellungs-URL zu generieren. Sofern keine weiteren spezifischen Installationsanweisungen angegeben sind, wird das Artefakt aus der definierten Quelle bezogen und durch *Homebrew* auf dem System installiert. [46]

Um diese *Cask* verwenden zu können, muss zunächst lokal der Verweis auf das entsprechende *Git Repository* hinterlegt werden, das diese *Cask* enthält, da sie nicht im Homebrew/homebrew-core *Repository* enthalten ist. Dies kann mit dem Befehl *brew tap* erfolgen. Nachdem das *Repository* eingebunden wurde, kann der Inhalt mittels *brew install* installiert werden. Alternativ besteht die Möglichkeit, die *Cask* direkt mit einem Verweis auf das *Repository* zu installieren, indem dieser vor den Anwendungsnamen gesetzt wird. Eine detaillierte Installationsanleitung ist in Quelltext 3.17 dargestellt.

```
# Direkte Installation von FQLite aus dem Repository
brew install --cask bocian67/fqlite/fqlite
ODER
# Auschecken des Repositories
brew tap bocian67/fqlite
# Installation der Cask aus dem lokal ausgechecktem Repository
brew install --cask fqlite
```

Quelltext 3.17: Hinzufügen der *Cask* mittels *Tap*

4 Ergebnisse

Im Folgenden werden die wichtigsten Ergebnisse der entwickelten CI/CD-Pipeline vorgestellt. Außerdem werden die Maßnahmen beschrieben, die zur Sicherstellung der Integrität der Anwendung *FQLite* ergriffen wurden.

4.1 Automatisierte Erstellung und Bereitstellung

Die Implementierung der CI/CD-Pipeline für die Anwendung *FQLite* hat signifikante Fortschritte in der Automatisierung und Effizienz der Erstellung und Bereitstellung der Software erzielt. Im Folgenden werden die zentralen Ergebnisse dieser Implementierung detailliert beschrieben.

4.1.1 Automatische Release-Erstellung durch Pull-Requests

Durch die Konfiguration der CI/CD-Pipeline ist es nun möglich, *Release* automatisch zu erstellen, sobald ein Branch mittels Pull-Request in den *master*- oder *main*-Branch gemerged wird. Diese Automatisierung reduziert den manuellen Aufwand und die Fehleranfälligkeit bei der Release-Erstellung erheblich. Der Trigger für diesen Prozess ist so eingerichtet, dass er bei einem geschlossenen Pull Request aktiviert wird, wobei sichergestellt wird, dass der Pull-Request tatsächlich gemerged wurde. Diese Bedingung wird im Workflow-Skript mittels *if*-Block überprüft, um die Integrität des Prozesses zu gewährleisten. Sofern dieser noch nicht gemerged wurde, werden Artefakte erstellt und abgelegt, sodass der Entwickler diese testen kann.

Zudem wird die Version der Software aus der Gradle-Konfigurationsdatei bezogen, um die Artfakte zu versionieren. Dies stellt sicher, dass die korrekte Anwendungsversion in den verschiedenen Schritten des Workflows verwendet wird und ermöglicht eine konsistente Versionierung über die verschiedenen Artefakte hinweg. Die Verwendung von *Git Tags* und *GitHub Releases* spielt dabei eine zentrale Rolle. *Tags* werden verwendet, um spezifische Punkte in der Historie des Repositories zu markieren, die als stabile Versionen freigegeben werden können. *GitHub Releases* bauen auf diesen *Tags* auf und ermöglichen es, zusätzliche Informationen wie Versionshinweise und Download-Links für die erstellten Artefakte bereitzustellen.

Neben der Veröffentlichung auf *GitHub* werden die Artefakte auf einem FTP-Server bereitgestellt und können zudem über *Chocolatey* unter Windows und *Homebrew* unter macOS bezogen werden.

Die Bereitstellung der einzelnen Dateien erfolgt direkt auf *GitHub*, was die Distribution und Nutzung der Anwendung für Endanwender vereinfacht. Dies bedeutet, dass Nutzer direkt auf der *GitHub*-Seite die jeweils aktuelle Version der Anwendung als Download vorfinden, was den Zugang zur Software erleichtert und die Verbreitung neuer Versionen beschleunigt. Die Version 2.6.5 der Anwendung, die in Abbildung 4.1 dargestellt ist, umfasst die für die verschiedenen Plattformen benötigten Artefakte. Für macOS stehen die Installationsdateien im .dmg-Format zur Verfügung, unterteilt nach den Prozessorarchitekturen *arm64* und *amd64 / x86_64*. Der zugehörige *Cask* für *Homebrew* kann aus dem Repository *homebrew-fqlite* (https://github.com/bocian67/homebrew-fqlite/) bezogen werden.

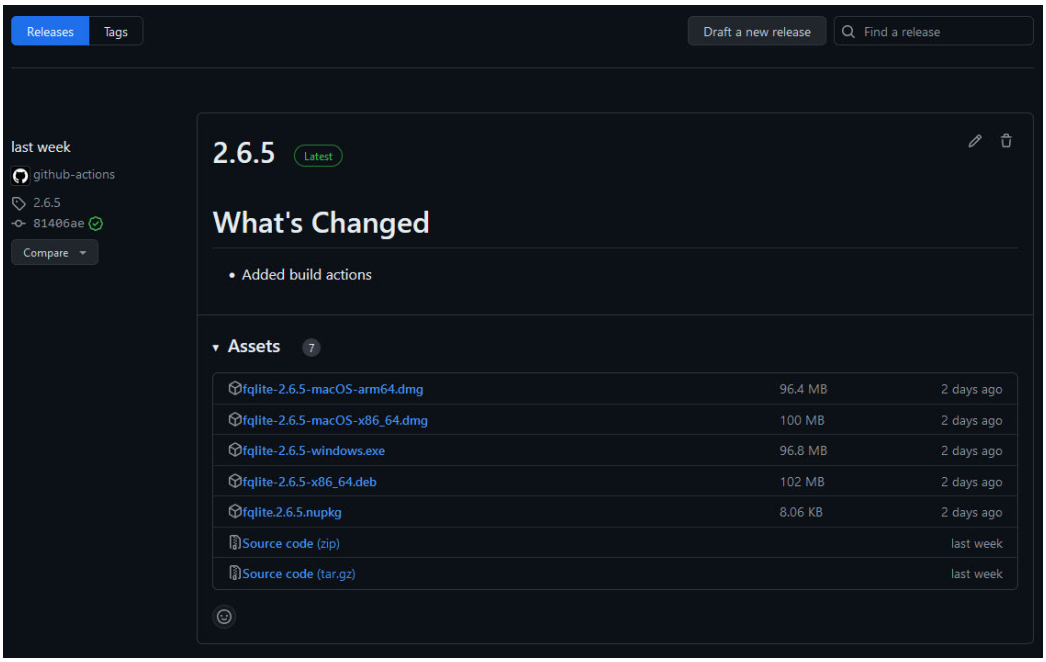

Abbildung 4.1: GitHub Release 2.6.5

Für Windows ist sowohl eine Installationsdatei im .exe-Format als auch ein .nupkg-Paket zur Installation mittels *Chocolatey* enthalten. Das Installationspaket für Linux wird im Debian-Format .deb bereitgestellt.

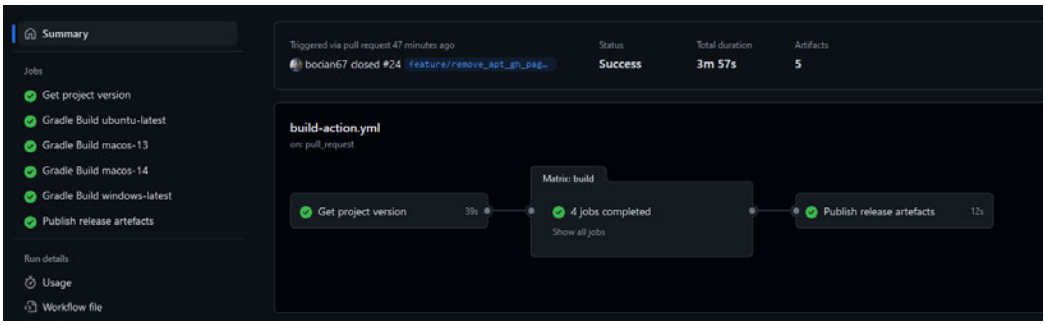

Abbildung 4.2: GitHub Workflow Ansicht

Auf *GitHub* kann die ausgeführte Aktion im Reiter *Actions* eingesehen werden, wie in Abbildung 4.2 dargestellt. Auf der linken Seite sind alle ausgeführten Jobs aufgeführt, während ihr Ablauf, basierend auf der Konfiguration in der Datei *build-action.yml*, zentral angezeigt wird. Es ist ersichtlich, dass der Release-Prozess durch die Zusammenführung des Branches *release/2.6.5* in den *master*- oder *main*-Branch ausgelöst wird. Im Anschluss erfolgt die Erstellung und Bereitstellung der Artefakte. Die grünen Haken neben den Aufgaben deuten auf eine erfolgreiche Ausführung hin.

In derselben Ansicht werden weiter unten zusätzliche Ausgaben angezeigt, einschließlich der erfolgreichen Signierung der Artefakte, wie in Abbildung 4.3 zu sehen ist. Nach der Erstellung des Artefakts von *FQLite* für macOS-14 mit der *arm64*-Architektur wird der zugehörige Hashwert der Anwendung dargestellt. Außerdem wird durch die Build-Action *gradle/gradle-build-action* das Ergebnis tabellarisch festgehalten. Diese Tabelle enthält Informationen zur Gradle-Task, dem Projekt, in welchem sie ausgeführt wurde, der Gradle-Version sowie einem Indikator, ob die Erstellung erfolgreich war.

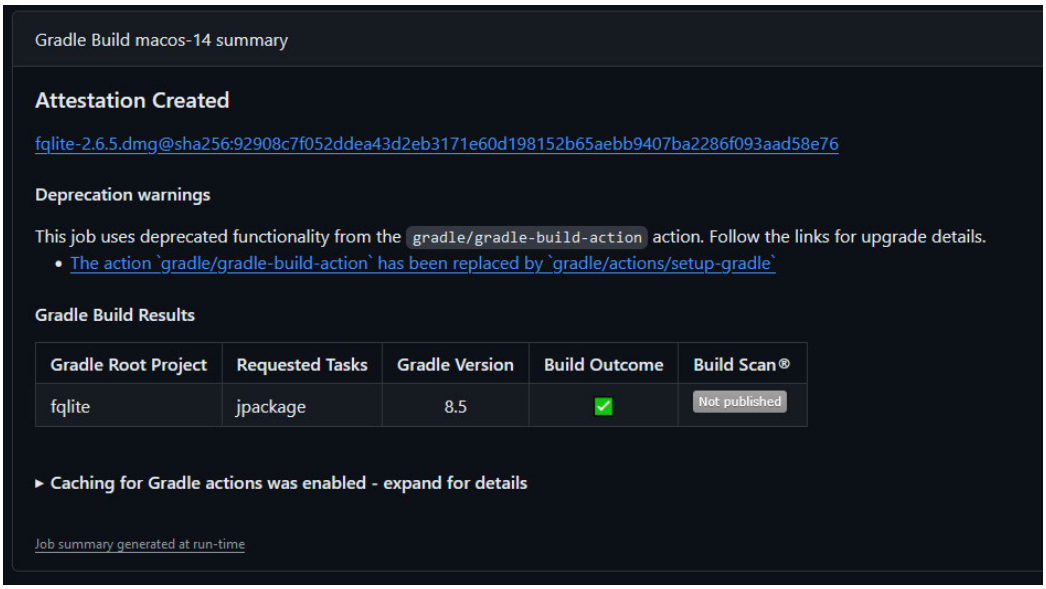

Abbildung 4.3: GitHub Attestation Ausgabe

4.1.2 Plattformspezifische Abbild-Erstellung

Durch die Verwendung von Gradle und dem *Badass Runtime*-Plugin wird die Konfiguration und Verwaltung von Abhängigkeiten und deren Versionen erheblich vereinfacht. Das Plugin ermöglicht eine flexible Anpassung der Build-Konfigurationen für verschiedene Betriebssysteme und Architekturen. Dies stellt sicher, dass die jeweils passenden Abhängigkeiten und deren Versionen für das spezifische Zielsystem verwendet werden, wodurch Kompatibilitätsprobleme minimiert werden.

Damit wird die automatische Erstellung plattformspezifischer Abbilder für Windows, Ubuntu Linux sowie macOS realisiert. Hierbei wird sowohl die Intel-Architektur (*x86_64*) als auch die ARM-Architektur (*arm64*) unterstützt. Diese plattformspezifischen Abbilder werden mittels *jpackage* generiert und stellen sicher, dass die Anwendung auf den wichtigsten Betriebssystemen lauffähig ist. Die automatische Anpassung der Konfiguration für unterschiedliche Betriebssysteme und Architekturen gewährleistet zudem eine hohe Flexibilität des Build-Prozesses.

4.1.3 Überprüfung und Validierung

Zur Sicherstellung der Integrität und Authentizität der erstellten Artefakte wird im Rahmen der CI/CD-Pipeline die Erstellung kryptografischer Signaturen gemäß den Best Practices der SLSA implementiert. Diese kryptografischen Signaturen gewährleisten, dass die generierten Artefakte während des Build-Prozesses nicht verändert wurden und vom ursprünglichen Autor stammen.

Die Validierung dieser Signaturen kann durch jeden Nutzer der Artefakte mithilfe des *GitHub* CLI durchgeführt werden. Hierzu bietet *GitHub* die Action *attest-build-provenance*, die für jedes Artefakt einen Hashwert erstellt und diesen kryptografisch signiert. Diese Signatur wird in einer öffentlichen Instanz gespeichert und kann über das *GitHub* CLI Tool überprüft werden. Der Befehl *gh attestation verify <file> [–repo | –owner]* ermöglicht es, die Signatur herunterzuladen und zu validieren, wodurch die Integrität der Datei sichergestellt wird.

Durch diese Maßnahmen wird sichergestellt, dass die heruntergeladenen Artefakte unverändert und authentisch sind. Dies erhöht das Vertrauen in die bereitgestellte Software und ermöglicht es den Endnutzern, die Integrität der Artefakte unabhängig zu überprüfen, auch wenn die Artefakte durch andere Quellen bezogen worden sind.

5 Diskussion

In diesem Kapitel werden Probleme thematisiert, welche sich während der Umsetzung gezeigt haben.

5.1 Bereitstellung auf macOS

Während sich die Anwendung auf den Plattformen Windows und Ubuntu Linux problemlos öffnen lässt, wird dies unter macOS eine andere Erfahrung für die Nutzer darstellen. Beim Versuch, die Anwendung zu starten, erhalten Nutzer unter macOS eine Sicherheitswarnung des Betriebssystems (getestet unter macOS 14) (siehe Abbildung 5.1).

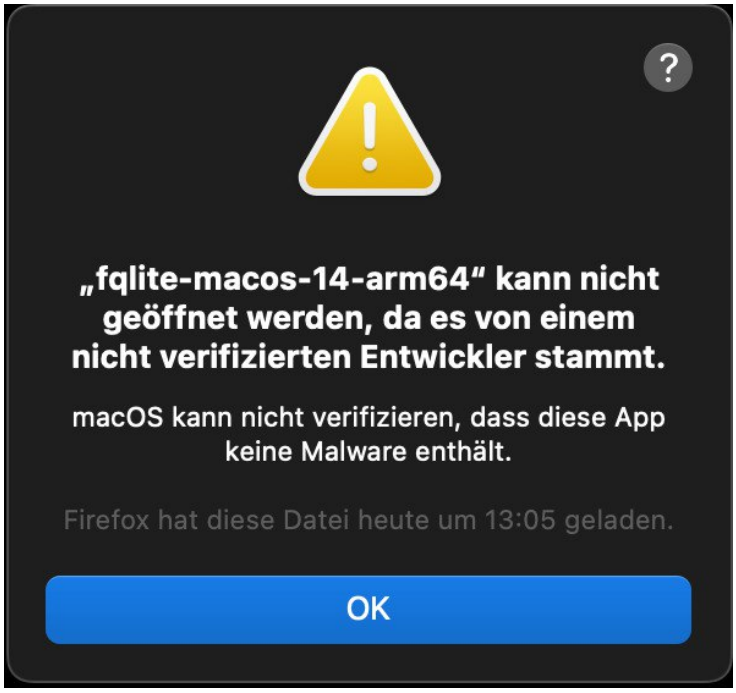

Abbildung 5.1: macOS: Warnung bei Ausführung von *FQLite*

Der Grund hierfür ist der sogenannte *Gatekeeper* von Apple, welcher sicherstellt, dass nur verifizierte Software ausgeführt wird. Dieser Mechanismus überprüft, ob eine Anwendung aus dem Apple Store oder dem Internet stammt. Im Falle einer Anwendung aus dem Internet wird die *Developer ID*, eine von Apple vergebene Entwicklerkennung, kontrolliert. Mit dieser *Developer ID* können Entwickler ihre Anwendungen signieren, um die Integrität der Anwendung und die Authentizität des Entwicklers zu gewährleisten. Diese Maßnahme dient dem Schutz der Nutzer vor bösartiger Software, da Apple die signierten Anwendungen auf schädliche Inhalte prüft [65]

Für die Signierung von Software ist jedoch ein Entwicklerkonto bei Apple erforderlich, welches eine Mitgliedschaft im Apple-Entwicklerprogramm voraussetzt. Diese Mitgliedschaft ist mit einer jährlichen Gebühr von 99 Dollar verbunden, weshalb im Rahmen dieser Arbeit darauf verzichtet wurde.

Trotzdem kann die Anwendung unter macOS geöffnet werden, indem in den Systemeinstellungen im Reiter „Datenschutz & Sicherheit“ im Abschnitt „Sicherheit“ die Option „Dennoch öffnen“ ausgewählt wird (siehe Abbildung 5.2).

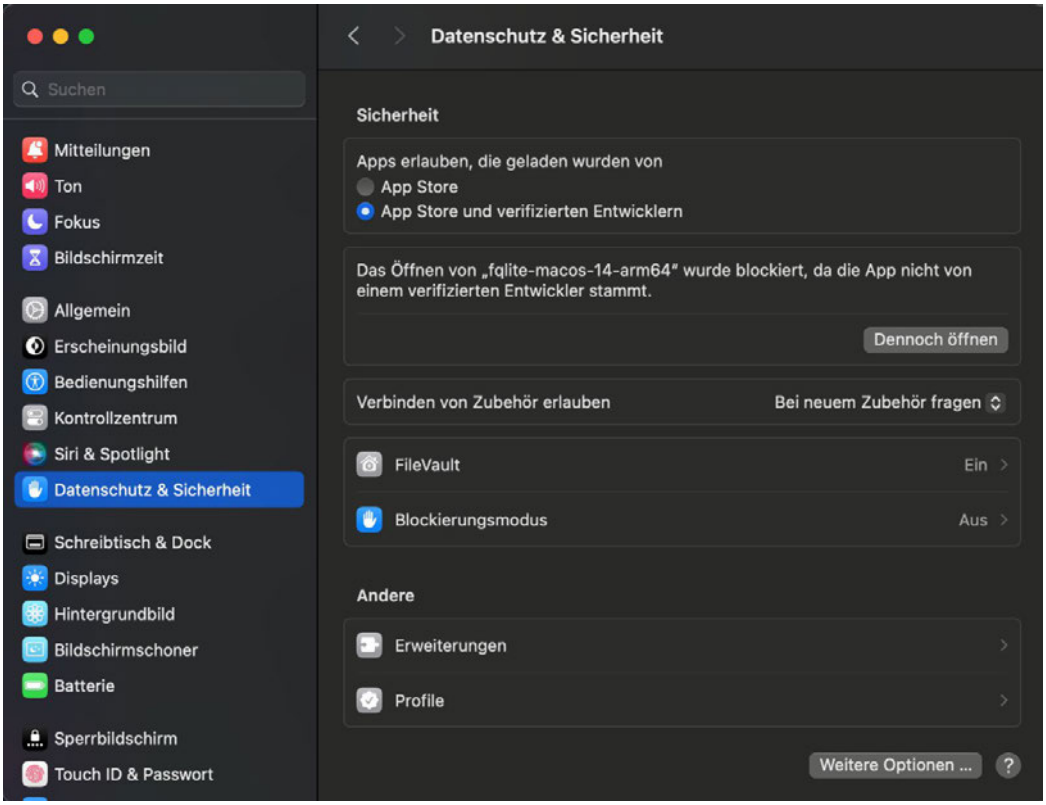

Abbildung 5.2: Anwendung trotz Quarantäne öffnen

Alternativ kann diese Einstellung auch über das Terminal vorgenommen werden, indem das Attribut *com.apple.quarantine*, das vom Gatekeeper verwendet wird, entfernt wird. Der entsprechende Befehl ist in Quelltext 5.1 dargestellt. Dabei muss individuell der Pfad zur Datei angepasst werden, in diesem Beispiel unter *fqlite*.

```
1    xattr -d com.apple.quarantine fqlite
```

Quelltext 5.1: Anwendung aus der Quarantäne entfernen

5.2 Nutzung von Paketmanagern

Die Installation der Anwendung kann mittels Installationsassistent auf den jeweiligen Betriebssystem durchgeführt werden. Zudem ist durch Nutzung von Paketmanagern wie *Chocolatey* unter Windows, *Homebrew* unter macOS und *apt* unter Linux die Installation der Anwendung möglich. Allerdings ist die Installation eingeschränkt durch den jeweiligen Verweis auf die zu installierende Konfiguration, darunter der .nupkg Datei für *Chocolatey* oder dem Repository des *Cask* unter *Homebrew*. Abhilfe könnte geschaffen werden, indem das Paket in die öffentlichen Repositories der jeweiligen Paketmanager eingebracht wird.

Homebrew stellt Anforderungen an die Aufnahme in das offizielle Repository auf ihrer Webseite bereit. Dazu zählen unter anderem eine Signierung der Anwendung mit Apples Entwicklerkonto, von welcher im Rahmen dieser Arbeit abgesehen worden ist. Mit einer offiziellen Signatur könnte eine Aufnahme in das offizielle Repository angestrebt werden, um die neueste Version von *FQLite* auch ohne Verweis auf den eigenen TAP bereitzustellen, wodurch die Installation auch ohne manuellen Download der Konfiguration möglich wäre.

Chocolatey stellt ebenfalls Anforderungen an die Aufnahme in das offizielle Community-Repository. Unter Berücksichtigung dieser wurde die Anwendung erfolgreich zur Prüfung eingereicht. Während dieser Prüfung wird zunächst die Chocolatey-Konfiguration validiert, daraufhin eine automatisierte Testinstallation auf einem Testsystem ausgeführt und anschließend ein Scan der Dateien auf potenziell schädliche Dateien durchgeführt wird. Nach diesen Prüfungen erfolgt eine Prüfung durch einen Moderator, welche bis zum Zeitpunkt der Abgabe dieser Arbeit aussteht. Sollte auch diese Prüfung erfolgreich verlaufen, kann das Paket über *Chocolatey* ohne Download der *.nupkg* Konfiguration bezogen werden.

5.3 Auswahl des Build-Systems

Es gibt verschiedene Build-Systeme für unterschiedliche Programmiersprachen. In dieser Arbeit wurde Gradle für Java verwendet, wobei auch *Apache Maven* eine alternative Option gewesen wäre. Gradle und Maven gehören zu den am weitesten verbreiteten Build-Systemen für Java-Projekte. Beide bieten ähnliche Funktionalitäten, insbesondere im Hinblick auf die Verwaltung von Abhängigkeiten und die Nutzung von Caching zur Steigerung der Performance. [66]

Maven basiert auf XML-Dateien, die als Projektobjektmodell (POM) bekannt sind, während Gradle eine Konfigurationsdatei in Form eines Build-Skripts (build.gradle) nutzt. Beide Systeme können durch Erweiterungen in ihrer Funktionalität ergänzt werden. Ein wesentlicher Unterschied liegt jedoch in der Flexibilität und Größe der Konfigurationsdateien: Maven setzt auf standardisierte, aber oft umfangreiche XML-Dateien, die zwar eine leichtere Verständlichkeit für erfahrene Maven-Nutzer bieten, jedoch weniger Spielraum für individuelle Anpassungen lassen. Gradle adressiert dieses Problem durch den Einsatz einer DSL (Domain Specific Language) auf Basis von Kotlin oder Groovy, wodurch kompaktere und flexiblere Konfigurationsdateien entstehen. [67]

Darüber hinaus hat sich gezeigt, dass Gradle in der Praxis eine höhere Performance bietet, indem Builds im Durchschnitt doppelt so schnell abgeschlossen werden wie bei Maven [66]. Die Verwendung der Groovy- oder Kotlin-basierten DSL macht Gradle zu einem leistungsfähigen und anpassungsfähigen Tool, das speziell auf die Anforderungen moderner Build-Prozesse abgestimmt ist.

Neben Gradle und Maven gibt es auch andere, oft jüngere Build-Systeme, die teilweise spezifische Anforderungen erfüllen. Ein Beispiel hierfür ist Bazel, ein von Google veröffentlichtes Build-System, das für verschiedene Programmiersprachen und Plattformen konzipiert ist und sich in gewisser Weise an den Prinzipien von Maven und Gradle orientiert [68]. Ein weiteres vielversprechendes Tool ist Buck2, entwickelt von Meta und seit 2023 als Open-Source-Projekt verfügbar [69]. Beide Build-Systeme bieten interessante Ansätze und sind in ihrer plattform- und sprachenunabhängigen Konzeption zukunftsweisend.

Jedoch sind Bazel und Buck2 aufgrund ihres noch jungen Alters weniger weit verbreitet und haben somit eine kleinere Nutzerbasis. Dies kann die Verfügbarkeit von Unterstützung und die Anzahl von Problemlösungen in der Community einschränken. Zudem sind diese Systeme noch nicht in dem Maße getestet und dokumentiert wie die etablierten Build-Systeme Gradle und Maven, die

durch ihre breite Anwendung in verschiedenen Szenarien umfassend erprobt sind. Die langjährige Nutzung etablierter Systeme hat dazu geführt, dass für viele spezifische Anforderungen detaillierte Dokumentationen und bewährte Lösungen zur Verfügung stehen.

6 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde die Anwendung erfolgreich mit Gradle umstrukturiert, sodass nun ein vollständig automatisierter Build-Prozess implementiert ist. Dieser Prozess stellt sicher, dass die Anwendung kontinuierlich und zuverlässig gebaut wird, was die Effizienz und Konsistenz der Softwareentwicklung erheblich verbessert. Durch den Einsatz von *GitHub* Runnern konnte die Anwendung plattformübergreifend für mehrere Betriebssysteme und Architekturen erstellt werden, was bedeutet, dass sie nun auf allen gängigen Systemen verfügbar ist.

Zusätzlich wurden spezifische Pakete für die Paketmanager *Chocolatey* (für Windows) und *Homebrew* (für macOS) erstellt. Diese Pakete ermöglichen es den Nutzern, die Anwendung unkompliziert zu installieren, ohne die Dateien manuell von *GitHub* herunterladen zu müssen. Dies verbessert nicht nur die Benutzerfreundlichkeit, sondern stellt auch sicher, dass die Installation standardisiert und fehlerfrei erfolgt.

Ein weiterer Schwerpunkt dieser Arbeit lag auf der Umsetzung der CI/CD-Pipeline unter strikter Berücksichtigung der OWASP Best Practices. Dadurch konnte eine möglichst sichere Ausführung und Erstellung der Anwendung gewährleistet werden. Die Einhaltung dieser Sicherheitsstandards minimiert das Risiko von Sicherheitslücken und schützt die Integrität des Entwicklungsprozesses.

Abschließend wurde die Integrität der Anwendungsdateien durch die Signierung mittels *GitHub Attestation* gesichert. Dies ermöglicht es den Benutzern, die Authentizität und Unveränderheit der heruntergeladenen Artefakte zu überprüfen, was das Vertrauen in die bereitgestellte Software weiter stärkt. Insgesamt wurden durch diese Maßnahmen wesentliche Fortschritte in der Automatisierung, Sicherheit und Verfügbarkeit der Anwendung erzielt.

Obwohl im Rahmen dieser Arbeit bedeutende Fortschritte bei der Automatisierung und plattformübergreifenden Bereitstellung der Anwendung erzielt wurden, gibt es noch einige Bereiche, die in zukünftigen Arbeiten weiterentwickelt werden könnten.

Für die Integration der Anwendung in das *Homebrew*-Community-Repository wäre eine Signierung der Anwendung mittels eines Apple Developer Accounts erforderlich. Durch diese Signierung würde die Anwendung den Sicherheitsanforderungen von macOS entsprechen und könnte somit ohne initiale Fehler auf diesen Systemen ausgeführt werden. Die Einbringung in das *Homebrew*-Repository würde die Verfügbarkeit und einfache Installation der Anwendung für Mac-Nutzer weiter verbessern und gleichzeitig das Vertrauen der Benutzer in die Sicherheit und Integrität der Software erhöhen.

Auf der Linux-Seite bietet sich die Möglichkeit, die Anwendung im .deb-Format unter dem Paketverwaltungssystem apt zu veröffentlichen. Diese Erweiterung würde es Linux-Benutzern ermöglichen, die Anwendung direkt über ihre bevorzugte Paketverwaltung zu installieren und zu aktualisieren. Aufgrund begrenzter Kapazitäten konnte diese Integration im Rahmen dieser Arbeit jedoch nicht umgesetzt werden, stellt aber ein wertvolles Ziel für zukünftige Arbeiten dar.

Ein weiterer Aspekt, der verbessert werden könnte, ist die Verwaltung der fehlenden Abhängigkeiten, die derzeit nicht über Gradle und MavenCentral bereitgestellt werden können. Eine Lösung bestünde darin, diese Abhängigkeiten in eine private Paketverwaltung zu integrieren. Gradle könnte dann unter Verwendung von Anmeldeinformationen diese Pakete automatisch beziehen. Dies hätte den Vorteil, dass alle Abhängigkeiten einheitlich über Gradle verwaltet werden könnten, anstatt sie aus einem lokalen Ordner im Projekt zu laden. Diese Vereinheitlichung würde nicht nur die Wartung des Projekts vereinfachen, sondern auch die Sicherheit und Nachvollziehbarkeit der verwendeten Bibliotheken erhöhen.

Literaturverzeichnis

[1] *Durchschnittlicher Stundensatz von IT-Freelancern in Deutschland in den Jahren 2013 bis 2017*, https://de.statista.com/statistik/daten/studie/957922/umfrage/durchschnittlicher-stundensatz-von-it-freelancern-in-deutschland/, [Online, abgerufen am 22.04.2024].

[2] *FQLite GitHub*, https://github.com/pawlaszczyk/fqlite, [Online, abgerufen am 02.05.2024].

[3] *Forensic SQLite Datenrettungs-Tool*, https://www.staff.hs-mittweida.de/~pawlaszc/fqlite/, [Online, abgerufen am 02.05.2024].

[4] D. Pawlaszczyk und C. Hummert, „Making the Invisible Visible – Techniques for Recovering Deleted SQLite Data Records“, *International Journal of Cyber Forensics and Advanced Threat Investigations*, Jg. 1, Nr. 1–3, S. 27–41, Feb. 2021, ISSN: 2753-9997. DOI: https://doi.org/10.46386/ijcfati.v1i1-3.17.

[5] P. P. Dingare, „Understanding CI/CD“, in *CI/CD Pipeline Using Jenkins Unleashed: Solutions While Setting Up CI/CD Processes*. Berkeley, CA: Apress, 2022, ISBN: 978-1-4842-7508-5. DOI: 10.1007/978-1-4842-7508-5_1. Adresse: https://doi.org/10.1007/978-1-4842-7508-5_1.

[6] *Was ist CI/CD?*, https://www.redhat.com/de/topics/devops/what-is-ci-cd, [Online, abgerufen am 23.04.2024].

[7] N. N. Zolkifli, A. Ngah und A. Deraman, „Version Control System: A Review“, *Procedia Computer Science*, Jg. 135, S. 408–415, 2018, The 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.08.191. Adresse: https://www.sciencedirect.com/science/article/pii/S1877050918314819.

[8] *What is version control?*, https://about.gitlab.com/topics/version-control/, [Online, abgerufen am 29.04.2024].

[9] *What is CI/CD?*, https://about.gitlab.com/topics/ci-cd/, [Online, abgerufen am 23.04.2024].

[10] P. Louridas, „Static code analysis“, *IEEE Software*, Jg. 23, Nr. 4, S. 58–61, 2006. DOI: 10.1109/MS.2006.114.

[11] *Unit testing*, https://en.wikipedia.org/wiki/Unit_testing, [Online, abgerufen am 24.04.2024].

[12] *OWASP Top 10 CI/CD Security Risks*, https://owasp.org/www-project-top-10-ci-cd-security-risks, [Online, abgerufen am 06.05.2024].

[13] *Angriff auf die Supply Chain - SolarWinds*, https://www.all-about-security.de/angriff-auf-die-supply-chain-solarwinds/, [Online, abgerufen am 06.05.2024].

[14] *CICD-SEC-1: Insufficient Flow Control Mechanisms*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-01-Insufficient-Flow-Control-Mechanisms, [Online, abgerufen am 06.05.2024].

[15] *CICD-SEC-2: Inadequate Identity and Access Management*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-02-Inadequate-Identity-And-Access-Management, [Online, abgerufen am 06.05.2024].

[16] *CICD-SEC-3: Dependency Chain Abuse*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-03-Dependency-Chain-Abuse, [Online, abgerufen am 06.05.2024].

[17] *CICD-SEC-4: Poisoned Pipeline Execution (PPE)*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-04-Poisoned-Pipeline-Execution, [Online, abgerufen am 12.06.2024].

[18] *CICD-SEC-5: Insufficient PBAC*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-05-Insufficient-PBAC, [Online, abgerufen am 05.07.2024].

[19] *CICD-SEC-6: Insufficient Credentials-Hygiene*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-06-Insufficient-Credential-Hygiene, [Online, abgerufen am 05.07.2024].

[20] *CICD-SEC-7: Insecure System Configuration*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-07-Insecure-System-Configuration, [Online, abgerufen am 05.07.2024].

[21] *Mercedes-Benz onboard logic unit (OLU) source code leaks online*, https://www.zdnet.com/article/mercedes-benz-onboard-logic-unit-olu-source-code-leaks-online/, [Online, abgerufen am 03.07.2024].

[22] *CICD-SEC-8: Ungoverned Usage of 3rd Party Services*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-08-Ungoverned-Usage-of-3rd-Party-Services, [Online, abgerufen am 07.07.2024].

[23] *CICD-SEC-9: Improper Artifact Integrity Validation*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-09-Improper-Artifact-Integrity-Validation, [Online, abgerufen am 07.07.2024].

[24] *CICD-SEC-10: Insufficient Logging and Visibility*, https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-10-Insufficient-Logging-And-Visibility, [Online, abgerufen am 07.07.2024].

[25] *Was ist Java*, https://www.java.com/en/download/help/whatis_java.html, [Online, abgerufen am 03.07.2024].

[26] *What is Java?*, https://www.ibm.com/de-de/topics/java, [Online, abgerufen am 03.07.2024].

[27] *The Java Virtual Maschine Specification, Java SE 20 Edition*, https://docs.oracle.com/javase/specs/jvms/se20/jvms20.pdf, [Online, abgerufen am 03.07.2024].

[28] *An Overview of the JVM Languages*, https://www.baeldung.com/jvm-languages, [Online, abgerufen am 03.07.2024].

[29] *JEP 392: Packaging Tool*, https://openjdk.org/jeps/392, [Online, abgerufen am 09.07.2024].

[30] *JEP 282: jlink: The Java Linker*, https://openjdk.org/jeps/282, [Online, abgerufen am 09.07.2024].

[31] *Gradle User Manual*, https://docs.gradle.org/current/userguide/userguide.html, [Online, abgerufen am 28.06.2024].

[32] *Gradle Wrapper Basics*, https://docs.gradle.org/current/userguide/gradle_wrapper_basics.html, [Online, abgerufen am 28.06.2024].

[33] *Gradle Basics*, https://docs.gradle.org/current/userguide/gradle_basics.html, [Online, abgerufen am 28.06.2024].

[34] *Groovy Language*, https://groovy-lang.org/, [Online, abgerufen am 28.06.2024].

[35] *Gradle Kotlin DSL*, https://docs.gradle.org/current/userguide/kotlin_dsl.html, [Online, abgerufen am 28.06.2024].

[36] *Gradle Settings Basics*, https://docs.gradle.org/current/userguide/settings_file_basics.html, [Online, abgerufen am 28.06.2024].

[37] *Gradle Buildfile Basics*, https://docs.gradle.org/current/userguide/build_file_basics.html, [Online, abgerufen am 28.06.2024].

[38] *Gradle Tasks*, https://docs.gradle.org/current/userguide/tutorial_using_tasks.html, [Online, abgerufen am 28.06.2024].

[39] *Gradle Build Lifecycle*, https://docs.gradle.org/current/userguide/build_lifecycle.html, [Online, abgerufen am 28.06.2024].

[40] M. Balliauw, *Pro Nuget*, 2nd ed., X. Decoster, Hrsg. Berkeley, CA: Apress L. P., 2013, 1372 S., Description based on publisher supplied metadata and other sources., ISBN: 978-1430260011.

[41] *Why Chocolatey?*, https://docs.chocolatey.org/en-us/why/, [Online, abgerufen am 06.08.2024].

[42] *An introduction to NuGet*, https://learn.microsoft.com/en-us/nuget/what-is-nuget, [Online, abgerufen am 30.08.2024].

[43] D. Platt, *Tweak Your Mac Terminal, Command Line macOS* (Springer eBook Collection). New York: Apress, 2021, 1535 S., ISBN: 978-1-4842-6171-2.

[44] *Homebrew*, https://brew.sh/, [Online, abgerufen am 12.08.2024].

[45] *Homebrew Taps*, https://docs.brew.sh/Taps, [Online, abgerufen am 12.08.2024].

[46] *Homebrew Formula Cookbook*, https://docs.brew.sh/Formula-Cookbook, [Online, abgerufen am 12.08.2024].

[47] *Automatische Tokenauthentifizierung*, https://docs.github.com/de/actions/security-for-github-actions/security-guides/automatic-token-authentication, [Online, abgerufen am 08.09.2024].

[48] *The Badass Runtime Plugin*, https://badass-runtime-plugin.beryx.org/releases/latest/, [Online, abgerufen am 09.07.2024].

[49] *JPackage User Guide*, https://docs.oracle.com/en/java/javase/17/jpackage/packaging-tool-user-guide.pdf, [Online, abgerufen am 29.07.2024].

[50] *jlink tool reference*, https://docs.oracle.com/javase/9/tools/jlink.htm, [Online, abgerufen am 11.07.2024].

[51] *Protocol Buffers*, https://protobuf.dev/, [Online, abgerufen am 08.09.2024].

[52] *GitHub Features*, https://github.com/features, [Online, abgerufen am 02.05.2024].

[53] *GitHub Features*, https://docs.github.com/de/actions/learn-github-actions/understanding-github-actions, [Online, abgerufen am 02.05.2024].

[54] *GitHub Features*, https://docs.github.com/de/actions/using-workflows/events-that-trigger-workflows, [Online, abgerufen am 02.05.2024].

[55] *GitHub Features*, https://docs.github.com/de/actions/using-github-hosted-runners/about-github-hosted-runners/about-github-hosted-runners, [Online, abgerufen am 02.05.2024].

[56] *Workflowsyntax für GitHub Actions*, https://docs.github.com/de/actions/using-workflows/workflow-syntax-for-github-actions, [Online, abgerufen am 15.07.2024].

[57] *Verwenden von Matrizen für deine Aufträge*, https://docs.github.com/de/actions/using-jobs/using-a-matrix-for-your-jobs, [Online, abgerufen am 15.07.2024].

[58] *SLSA Provenance*, https://slsa.dev/spec/v1.0/provenance, [Online, abgerufen am 22.07.2024].

[59] *About SLSA*, https://slsa.dev/spec/v1.0/about, [Online, abgerufen am 19.07.2024].

[60] *GitHub Action attest-build-provenance*, https://github.com/actions/attest-build-provenance, [Online, abgerufen am 19.07.2024].

[61] *GitHub gh attestation verify*, https://cli.github.com/manual/gh_attestation_verify, [Online, abgerufen am 19.07.2024].

[62] *Git Basics - Tagging*, https://cli.github.com/manual/gh_attestation_verify, [Online, abgerufen am 23.07.2024].

[63] *Veröffentlichungen in einem Repository verwalten*, https://docs.github.com/de/repositories/releasing-projects-on-github/managing-releases-in-a-repository, [Online, abgerufen am 23.07.2024].

[64] *Chocolatey - Getting Started*, https://docs.chocolatey.org/en-us/getting-started/, [Online, abgerufen am 06.08.2024].

[65] *Safely open apps on your Mac*, https://support.apple.com/en-is/102445, [Online, abgerufen am 29.07.2024].

[66] *Gradle vs Maven Comparison*, https://gradle.org/maven-vs-gradle/, [Online, abgerufen am 29.08.2024].

[67] *Ant vs Maven vs Gradle*, https://www.baeldung.com/ant-maven-gradle, [Online, abgerufen am 29.08.2024].

[68] *Why Bazel?*, https://bazel.build/about/why, [Online, abgerufen am 29.08.2024].

[69] *Why Buck2*, https://buck2.build/docs/about/why, [Online, abgerufen am 29.08.2024].

Eidesstattliche Erklärung

Hiermit versichere ich – Jan Schneider – an Eides statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt oder anderweitig veröffentlicht.

Mittweida, 27. September 2024 Jan Schneider, B.Sc.

Ort, Datum