Universität Potsdam
Mathematisch-Naturwissenschaftliche Fakultät
Informatik/Computational Science

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

Prototypisches Unterstützungssystem für die DLR-Guidelines

Erstgutachterin: Prof. Dr. Anna-Lena Lamprecht

Zweitgutachterin: M.Sc. Carina Haupt

vorgelegt von:
Oliver Zablocki
Matrikelnummer 808050
Potsdam, den 11. Oktober 2024

11. Dezember 2024

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Entwicklung eines Prototypischen Unterstützungssystems zur automatisierten Überprüfung von Softwareentwicklungsprojekten gemäß der Software-Engineering-Richtlinien des Deutschen Zentrums für Luft- und Raumfahrt (DLR). Ziel ist es, Forschende des DLR dabei zu unterstützen, die Qualität und Nachhaltigkeit ihrer Softwareprojekte zu verbessern. Der Prototyp ermöglicht eine Evaluierung der Softwareprojekte auf der DLR-internen GitLab-Instanz, wobei er sich auf die automatisierte Überprüfung von Dokumentationen, Programmierstandards, Testframeworks und Lizenzmanagement konzentriert. Durch die Anwendung dieser Methoden können Fehler minimiert und die Zuverlässigkeit der entwickelten Software, insbesondere in sicherheitskritischen Bereichen, erhöht werden. Diese Arbeit unterstreicht die Notwendigkeit der Einhaltung von Richtlinien und die Bedeutung automatisierter Prozesse für die langfristige Qualitätssicherung und Effizienzsteigerung in der Softwareentwicklung.

Inhaltsverzeichnis

Einleitung

In den folgenden Kapiteln werden die spezifischen Aufgaben des entwickelten prototypischen Unterstützungssystems (im Folgenden: *Prototyp*) sowie die zugrunde liegende Motivation für die Implementierung beschrieben.

1.1 Aufgabenbeschreibung

In dieser Arbeit wird ein Prototyp entwickelt, der Forschende des Deutschen Zentrums für Luft- und Raumfahrt (DLR) dabei unterstützen soll, ihre Softwareentwicklungsprojekte mithilfe der Software-Engineering-Richtlinie des DLR zu evaluieren. Die Software-Engineering-Richtlinie wurde erarbeitet, um Softwareentwicklungs- und Dokumentationsprozesse gemäß von Anwendungsklassen einzuschätzen und zu verbessern. Im DLR wird GitLab im Rahmen der Softwareentwicklung eingesetzt. Es bietet sich daher eine automatisierte Überprüfung der Entwicklung und Versionierung von Softwareprojekten auf der DLR-internen Instanz des Onlinedienstes GitLab an. Weiterhin verfügt GitLab über eine API, die es ermöglicht, viele der Richtlinien effizient zu prüfen. Im Rahmen dieser Arbeit wird der Prototyp konzeptioniert und implementiert, wobei sich der Fokus auf Empfehlungen konzentriert, deren Evaluierung sich automatisieren lässt. Zudem beschränkt sich der Prototyp auf Softwareprojekte, die in Python entwickelt wurden.

Folgende Aspekte des Softwareentwicklungs- und Dokumentationsprozesses werden automatisiert evaluiert:

- Die Dokumentation, einschließlich Glossaren und README-Dateien, wird auf ihre Existenz geprüft.
- Das Changelog, als Dokumentation der Versionshistorie, wird überprüft.
- Die Nutzung von Codeanalysewerkzeugen zur Einhaltung von Programmierregeln wird festgestellt und ausgewertet.
- Die Verfügbarkeit und Nutzung von Testframeworks sowie die Durchführung von Tests werden überprüft.
- Das Lizenzmanagement wird basierend auf der Existenz und Gültigkeit einer Lizenzdatei bewertet.
- Die Regelmäßigkeit und Eindeutigkeit von Releases wird analysiert.
- Die Verwendung automatisierter Build-Tools wird überprüft.

Zur Bewertung des Prototyps werden zunächst einzelne Projekte individuell auf die Erfüllung der Richtlinien überprüft, gefolgt von der Ausführung des Prototyps auf alle verfügbaren Projekte. Die Ergebnisse werden projektspezifisch gespeichert und zeigen, in welchem Umfang die Empfehlungen erfüllt sind und in welche Anwendungsklasse gemäß der DLR-Rahmenrichtlinie das Projekt einzuordnen ist.

1.2 Motivation

Die Software-Engineering Guidelines des DLR werden aufgrund der Notwendigkeit genutzt, die Qualität, Nachhaltigkeit und Reproduzierbarkeit der entwickelten Software zu verbessern. Viele WissenschaftlerInnen verfügen nicht über eine formale Ausbildung im Software-Engineering was zu Problemen wie unzureichender Dokumentation, unklaren Code-Strukturen und erhöhter Fehleranfälligkeit führt [3]. Insbesondere EntwicklerInnen, die aus wissenschaftlichen Disziplinen ohne Hintergrund in der Softwaretechnik kommen, benötigen zusätzliche Unterstützung und Schulungen, um gute Ergebnisse zu erzielen [vgl. 15]. Um diesen Herausforderungen zu begegnen, initiierte das DLR die Software Engineering Initiative, welche die Softwarequalität und -nachhaltigkeit steigern soll. Ein zentraler Bestandteil dieser Maßnahme sind die 2016 veröffentlichte *Software-Engineering Guideline*, die umfassende Ratschläge und bewährte Praktiken bieten, um sicherzustellen, dass entwickelte Software langfristig wartbar bleibt und dem Einsatzzweck angemessenen Qualitätsstandards entspricht [3].

Die Einhaltung dieser Richtlinie führt zu einer Erhöhung der Softwarequalität und fördert die Nachhaltigkeit der Entwicklung. Sie trägt zudem dazu bei, Wissen innerhalb des Projekts und der Organisation zu bewahren, insbesondere bei personellen Veränderungen oder langen Projektlaufzeiten [3]. Strukturierte Prozesse und bewährte Praktiken reduzieren das Risiko von Fehlentwicklungen und damit verbundenen finanziellen oder sicherheitsrelevanten Problemen, was in sicherheitskritischen Bereichen wie der Luft- und Raumfahrt von besonderer Relevanz ist [3]. Schließlich fördern die Richtlinien ein hohes Maß an Verantwortungsbewusstsein bei den Entwicklern, indem sie eine klare Orientierung bieten, wie Projekte effizient und qualitätsbewusst umgesetzt werden können [vgl. 14]. Die Initiative hat erfolgreich eine aktive Software-Engineering-Community im DLR aufgebaut und das Bewusstsein für die Bedeutung nachhaltiger und reproduzierbarer Software gestärkt [3].
Angesichts der zahlreichen Vorteile dieser Richtlinie ist die Entwicklung von Mechanismen zur automatischen Überprüfung ihrer Einhaltung besonders wertvoll. Eine automatische Prüfung stellt sicher, dass die Empfehlungen konsistent angewendet werden und hilft EntwicklerInnen Fehler oder Abweichungen frühzeitig zu erkennen. Dies steigert nicht nur die Softwarequalität, sondern verbessert auch die Effizienz und Zuverlässigkeit der Entwicklungsprozesse. Darüber hinaus reduziert ein automatisierter Ansatz den Aufwand für manuelle Überprüfungen und stellt sicher, dass alle Projekte gleichermaßen von den bewährten Praktiken profitieren.

1.3 Struktur der Arbeit

Einleitung und theoretische Grundlagen: Zunächst werden die Aufgabenstellung und die Motivation der Arbeit erläutert. Darüber hinaus wird eine Einführung in das Software-Engineering gegeben sowie ein Überblick über die Software-Engineering-Richtlinien des DLR präsentiert.

Konzept und Implementierung: Anschließend wird das Konzept und die Implementierung des Prototyps beschrieben, der zur automatisierten Überprüfung von Softwareprojekten entwickelt wurde. Wichtige Aspekte umfassen das Anforderungsmanagement, die Software-Architektur, das Design, das Änderungs- und Release-Management sowie die Automatisierung und das Abhängigkeitsmanagement.

Ergebnisse und Ausblick: Abschließend werden die Ergebnisse der Prototyp-Evaluierung präsentiert, wobei das Potenzial des Prototyps für zukünftige Entwicklungen und Erweiterungen betont wird. Insbesondere wird hervorgehoben, wie der Prototyp zur Qualitätssicherung und Effizienzsteigerung der Softwareprojekte im DLR beiträgt.

Software Engineering - Aktueller Stand

Nachfolgend werden die grundlegenden theoretischen Konzepte des Software Engineerings und die aktuellen Richtlinien des DLR für die Softwareentwicklung detailliert beschrieben.

2.1 Theoretische Grundlagen

Software Engineering ist ein systematischer und methodischer Ansatz zur Entwicklung, Bereitstellung und Instandhaltung von Software. Das Hauptziel des Software Engineerings besteht darin, Softwareprodukte von hoher Qualität effizient und kosteneffektiv zu erstellen, während gleichzeitig die Anforderungen der Benutzer und Stakeholder erfüllt werden. Dieser Prozess umfasst bewährte Methoden und Techniken, die den gesamten Lebenszyklus der Software abdecken, beginnend mit der Anforderungsanalyse über die Implementierung bis hin zur Wartung und kontinuierlichen Weiterentwicklung.

Im Folgenden werden die verschiedenen Kategorien der Softwareentwicklung beschrieben und ein kurzer Bezug zur Literatur über Software Engineering hergestellt. Dabei wird auf auf Ian Sommerville [16] zurückgegriffen, dessen Werk *Software Engineering* als Grundlage für die Definition grundlegender Begrifflichkeiten dient. Die Kategorien orientieren sich an den Vorgaben des DLR [vgl. 2].

Anforderungsmanagement Karl Wiegers betont: "Requirements encompass both the user's view of the external system behavior and the developer's view of some internal characteristics. They include both the behavior of the system under specific conditions and those properties that make the system suitable ... for use by its intended operators."[17, S. 24]. Anforderungsmanagement ist der strukturierte Prozess der systematischen Erfassung, Analyse, Spezifikation und Verwaltung von Anforderungen über den gesamten Lebenszyklus einer Software hinweg. Das Hauptziel besteht darin, die Bedürfnisse und Erwartungen der Stakeholder präzise zu erfassen und in nachvollziehbare und formale Anforderungen zu übersetzen, die als Grundlage für das Design, die Implementierung und die Wartung der Software dienen.

Software Architektur "Softwarearchitektur beschreibt die Gliederung der einzelnen Softwarekomponenten in Module"[1, S.50]. Sie dient dazu, das Verständnis der Entwickler für das Projekt zu vertiefen und einen klaren Plan für das Endprodukt bereitzustellen. Die vollständige Realisierung der Architektur stellt eine anspruchsvolle Herausforderung dar, die entscheidend ist für die Einhaltung des Entwicklungsplans der Software. wie etwa das Model-View-Controller (MVC)-Muster und die Layered Architecture, die zur Vereinfachung der Softwareentwicklung und -wartung beitragen.

Design und Entwicklung Design und Entwicklung sind essenzielle Komponenten der Softwareentwicklung. Design Thinking ist ein zentraler Ansatz, der kreative, nutzerzentrierte Problemlösungen in fünf Phasen ermöglicht: Verstehen, Definieren, Ideen finden, Prototyp entwickeln und Evaluieren [1, S. 264]. Sommerville betont die Wichtigkeit eines strukturierten, objektorientierten Designs sowie die Verwendung von UML-Diagrammen zur

Modellierung und Spezifikation von Systemen.

Änderungsmanagment "Das Änderungsmanagement für Anforderungen umfasst die Verfahren zur Änderung der festgeschriebenen Anforderungen"[1, S.216]. In Anbetracht der fortlaufenden Möglichkeit von Verbesserungen, Anforderungsänderungen oder allgemeinen Optimierungen während eines laufenden Projekts, ist es entscheidend, dass diese Änderungen nachvollziehbar und übersichtlich dokumentiert sind. Dieser Prozess stellt die Integrität des Projekts sicher. Sommerville beschreibt es als einen detaillierten Prozess, bei dem Änderungen identifiziert, analysiert, bewertet und entsprechend ihrer Priorität umgesetzt werden müssen.

Software-Test "Das Testen der Software auf funktionaler Ebene gemäß den Anforderungen ist ein wesentlicher Bestandteil bei der Entwicklung und Wartung von Anwendungen. In der Praxis wird die Korrektheit von Implementierungen weitgehend und bevorzugt durch Testen überprüft."[1, S.484]. Bei Sommerville sind es verschiedene Testebenen, wie Entwicklungstests, Release-Tests und Benutzertests. Er betont besonders den Einsatz von automatisierten Tests und kontinuierlicher Integration, um den Testprozess zu optimieren und frühzeitig Fehler zu erkennen.

Release-Managment Das Release-Management in der modernen Softwareentwicklung stellt sicher, dass stabile und nutzbare Versionen der Software effizient bereitgestellt werden [vgl. 1, S.450]. Sommerville unterscheidet zwischen Major Releases, die neue Funktionalitäten beinhalten, und Minor Releases, die Fehlerbehebungen umfassen. Er betont, dass jedes Release sorgfältig dokumentiert werden muss, um die exakte Reproduzierbarkeit der Version zu gewährleisten.

Automatisierung und Abhängigkeitsmanagment Die Automatisierung des Build-Prozesses und das effektive Management von Abhängigkeiten sind wichtige Elemente der modernen Softwareentwicklung. Sie ermöglichen die automatische Integration von Codeänderungen in ein zentrales Repository, wo sie automatisch kompiliert, getestet und bereitgestellt werden. Moderne Ansätze in der Softwareentwicklung, wie die Nutzung von GitLab setzen auf Continuous Integration/Continuous Deployment [4], wie das Bereitstellen von Codeänderungen, gefolgt von automatisierten Builds und Tests, um die Codequalität zu sichern und kontinuierliches Bereitstellen [9], indem getestete Änderungen automatisch in die Produktionsumgebung übertragen werden. Diese Automatisierung reduziert manuelle Aufgaben und minimiert Fehler, was die Effizienz der Softwareentwicklung erhöht.

2.2 Software-Engineering-Richtlinie des Deutsches Zentrum für Luft- und Raumfahrt

Das DLR hat Empfehlungen zusammengestellt, die WissenschaftlerInnen dabei helfen sollen, ihre Softwareentwicklung und -dokumentation besser einschätzen und verbessern zu können. Diese Empfehlungen sind in Anwendungsklassen unterteilt, die aufeinander aufbauen.
Die Kategorisierung bietet einen ersten Anhaltspunkt dafür, welche Empfehlungen nützlich sein

könnten. Diese Liste sollte jedoch bei Bedarf individuell angepasst werden, um den Projektspezifika gerecht zu werden. Das Ziel ist es, durch die einfache Ermittlung der angestrebten Anwendungsklasse einen schnellen und effizienten Einstieg in die Thematik zu ermöglichen.

2.2.1 Anwendungsklassen im Überblick

In dieser Sektion werden die Anwendungsklassen im Überblick dargestellt, um die verschiedenen Anforderungen und Zielgruppen zu verdeutlichen. Eine zusammenfassende Übersicht der Anwendungsklassen, einschließlich ihrer spezifischen Merkmale und relevanten Zielgruppen, ist in Tabelle 1 zu finden.

Anwendungsklasse 0: Diese Klasse ist für Software vorgesehen, die für den persönlichen Gebrauch entwickelt wird und einen begrenzten Funktionsumfang bietet, wie z.B. administrative Skripte und Software zum Testen von Ideen.

Anwendungsklasse 1: Software in dieser Klasse sollte so entwickelt werden, dass sie von Dritten genutzt und weiterentwickelt werden kann. Ziel ist, dass die Software nachvollziehbar und reproduzierbar ist. Ein breites Funktionsspektrum ist nicht erforderlich. Typischerweise umfasst diese Klasse Software, die im Rahmen von Studienarbeiten, Dissertationen oder ähnlichen Forschungsarbeiten entsteht.

Anwendungsklasse 2: Diese Klasse ist für Software konzipiert, die langfristig weiterentwickelt und gewartet wird. Die Software hat den Status eines Produkts und muss spezifische Anforderungen erfüllen. Diese Kategorie richtet sich an umfangreiche Forschungs-Frameworks und Software, die eine langfristige Wartbarkeit erfordert.

Anwendungsklasse 3: In der höchsten Klasse wird Software entwickelt, bei der Fehler nicht toleriert werden dürfen. Dies umfasst Risikomanagement und den Nachweis der Softwarekorrektheit. Diese Klasse ist für missionskritische Software und Anwendungen vorgesehen, die höchste Sicherheitsstandards erfüllen müssen.

Anwendungsklasse	Zielgruppe	Merkmale	Beispiele
0	Einzelpersonen, die Software für den Eigengebrauch schreiben.	Einfacher Funktionsumfang, wenig formale Dokumentation.	Administrative Skripte, Testsoftware.
1	Studierende, Forschende, die Software für akademische Arbeiten erstellen.	Nachvollziehbarkeit und Reproduzierbarkeit, begrenzter Funktionsumfang.	Studienarbeiten, Dissertationen.
2	Entwicklerteams, die Softwareprodukte erstellen.	Breites Funktionsspektrum, langfristige Wartbarkeit und Weiterentwicklung.	Umfangreiche Forschungs-Frameworks, Softwareprodukte mit langer Lebensdauer.
3	Entwickler von sicherheitskritischen Systemen.	Fehlerfreiheit, Risikomanagement, Korrektheitsnachweis.	Missionskritische Software, sicherheitskritische Anwendungen.

Tabelle 1: Übersicht zu den Anwendungsklassen. Diese Darstellung bietet eine umfassende Übersicht der Anwendungsklassen in der Softwareentwicklung. Sie beschreibt, welche Anwendungsklasse für welche Zielgruppe relevant ist, und erläutert die spezifischen Merkmale der jeweiligen Gruppen.

2.2.2 Empfehlungen für den Softwareentwicklungsprozess vom DLR

Die DLR-Richtlinien für Software-Engineering bieten umfassende Empfehlungen zur Verbesserung der Qualität und Effizienz in der Softwareentwicklung. Hier ist eine grobe Zusammenfassung der wichtigsten Bereiche und deren Inhalte:

Qualifizierung
Ein wesentlicher Punkt der Richtlinien ist die Sicherstellung der Qualifizierung der beteiligten Personen. Es wird empfohlen, dass alle EntwicklerInnen und Projektbeteiligten die erforderlichen Kenntnisse und Fähigkeiten besitzen, um ihre Aufgaben effizient und korrekt auszuführen. Dazu gehört nicht nur das technische Wissen, sondern auch ein tiefes Verständnis für die spezifischen Anforderungen und Herausforderungen des Projekts. Schulungen und regelmäßige Weiterbildungen sind essenziell, um die Nutzung der eingesetzten Werkzeuge und Methoden zu optimieren. Der Austausch zwischen Kollegen und die Kommunikation über den individuellen Qualifikationsbedarf mit Vorgesetzten fördern ein kontinuierliches Lernen und die Verbesserung der Fähigkeiten innerhalb des Teams.

Anforderungsmanagment
Das Anforderungsmanagement wird in den DLR-Richtlinien als entscheidender Faktor für den Erfolg eines Projekts hervorgehoben. Es wird empfohlen, dass die Aufgabenstellung klar und präzise mit allen Beteiligten abgestimmt und dokumentiert wird. Dies minimiert das Risiko von Missverständnissen und Fehlentwicklungen während des Entwicklungsprozesses. Besonders betont wird die Notwendigkeit, funktionale Anforderungen systematisch zu erfassen, zu priorisieren

und regelmäßig zu überprüfen. Durch diese sorgfältige Erfassung und Verwaltung der Anforderungen wird sichergestellt, dass die entwickelte Software die Erwartungen der Stakeholder erfüllt.

Software Architektur
Ein weiterer zentraler Aspekt ist die Software-Architektur, die als Fundament eines jeden Softwareprojekts betrachtet wird. Die Richtlinien empfehlen eine regelmäßige Überprüfung und Anpassung der Software-Architektur, um sicherzustellen, dass sie den aktuellen und zukünftigen Anforderungen des Projekts gerecht wird. Systematische Reviews helfen dabei, Abweichungen von der geplanten Architektur frühzeitig zu erkennen und zu korrigieren. Dadurch wird die langfristige Wartbarkeit und Weiterentwicklung der Software gesichert.

Änderungsmangment
Das Änderungsmanagement wird in der Richtlinie als ein strukturierter und transparenter Prozess beschrieben, der es ermöglicht, notwendige Anpassungen und Verbesserungen an der Software effektiv durchzuführen. Die Verwendung von Ticketsystemen zur zentralen Erfassung und Nachverfolgung von Änderungswünschen wird empfohlen, um die Nachvollziehbarkeit und Konsistenz der Änderungen zu gewährleisten. Dies trägt dazu bei, dass Änderungen kontrolliert und ohne negative Auswirkungen auf das Gesamtsystem umgesetzt werden können.

Design und Implementierung
Im Bereich Design und Implementierung wird die Anwendung bewährter Konstruktionsansätze und einheitlicher Programmierstandards betont. Die Richtlinie ermutigt Entwickelnde dazu, einen konsistenten Programmierstil zu verwenden und kontinuierliches Refactoring als festen Bestandteil des Entwicklungsprozesses zu etablieren. Dies soll die Struktur und Lesbarkeit des Codes verbessern, was wiederum die langfristige Qualität und Wartbarkeit der Software sicherstellt. Umfassende Tests auf Modulebene sind ebenfalls von großer Bedeutung, um Fehler frühzeitig zu erkennen und zu beheben.

Software-Test
Die Qualitätssicherung durch regelmäßige und systematische Tests ist ein weiterer kritischer Punkt in den DLR-Richtlinien. Es wird empfohlen, verschiedene Testebenen wie Modul-, Integrations- und Systemtests durchzuführen, um die Funktionalität und Zuverlässigkeit der Software zu gewährleisten. Ein strukturierter Testprozess hilft, potenzielle Fehlerquellen zu identifizieren und sicherzustellen, dass die Software den festgelegten Anforderungen entspricht.

Release Managment
Im Release-Management betont die Richtlinie die Wichtigkeit eines strukturierten und gut organisierten Prozesses zur Erstellung, Prüfung und Freigabe von Software-Versionen. Der Einsatz von Versionskontrollsystemen wird als unerlässlich angesehen, um eine konsistente und stabile Entwicklung sicherzustellen. Automatisierte Build-Prozesse tragen dazu bei, den Aufwand zu reduzieren und die Qualität der veröffentlichten Softwareversionen zu steigern.

Automatisierung und Abhängigkeitsmanagement
Schließlich wird in den DLR-Richtlinien die Automatisierung von Build-, Test- und Deployment-Prozessen empfohlen, um die Effizienz des Entwicklungsprozesses zu erhöhen und Fehler zu minimieren. Ein systematisches Management von Abhängigkeiten ist ebenfalls entscheidend, um Kompatibilitätsprobleme zu vermeiden und die Wartung der Software zu erleichtern. Dies ermöglicht es, komplexe Projekte erfolgreich zu verwalten und sicherzustellen, dass alle Komponenten reibungslos zusammenarbeiten.

2.3 GitLab: Eine Platform für Qualitätssicherung im Software-Engineering

GitLab stellt eine umfassende Online-Platform für das Software-Engineering bereit, die EntwicklerInnen eine effiziente Verwaltung des gesamten Lebenszyklus ihrer Software ermöglicht. Mit der in der Plattform integrierten Funktionen für Continuous Integration (CI) und Continuous Deployment (CD) sowie automatisierte Tests und effektives Release-Management sorgt GitLab dafür, dass die Softwarequalität über den gesamten Entwicklungsprozess hinweg aufrechterhalten wird. Die Plattform unterstützt Teams nicht nur bei der Planung und Verwaltung von Software-Updates, sondern bietet auch eine optimierte Verwaltung von Entwicklungsumgebungen und die nahtlose Integration externer Ressourcen. GitLab vereinfacht die Überwachung und Verbesserung der Softwarequalität, indem es eine konsistente Umgebung für das Testen, Entwickeln und Bereitstellen von Code bereitstellt. Die umfassenden Funktionen von GitLab gewährleisten, dass die Einhaltung von bewährten Methoden und Qualitätsstandards durch die Platform selbst unterstützt wird.

Die GitLab-API bietet einen detaillierten und konsistenten Zugriff auf die verschiedenen Funktionen und Daten der GitLab-Plattform [vgl. 12]. Sie ermöglicht es EntwicklerInnen, programmgesteuert auf nahezu alle Aspekte der Platform zuzugreifen, einschließlich Projekte, Repositories, Benutzer, Merge Requests, Issues und CI/CD-Pipelines. Durch spezifische Endpunkte und strukturierte Datenabfragen können tiefgreifende Informationen zu Build-Prozessen, Testberichten und Deployment-Status abgefragt werden. Dies ermöglicht eine präzise und umfassende Interaktion mit GitLab-Projekten, sei es zur Datenabfrage, Verwaltung oder Automatisierung von Prozessen. Neben der direkten Schnittstelle gibt es auch die Python Bibliothek *Python-Gitlab-API*. Diese Bibliothek stellt eine Schnittstelle bereit, um GitLab-Funktionen direkt aus Python-Skripten heraus zu nutzen. Sie ermöglicht die Automatisierung einer Vielzahl von Aufgaben.

Konzept und Implementierung

Zunächst liegt der Fokus auf den Empfehlungen, die der Prototyp zur automatisierten Überprüfung von Softwareprojekten umsetzt und die die Grundlage für die Evaluierung bilden. Anschließend werden die Randbedingungen skizziert, die die technische Machbarkeit beeinflussen. Darauf aufbauend wird der funktionale Ansatz erläutert, der den Programmablauf von der Datenerfassung bis zur Ergebnisbewertung umfasst.

3.1 Fokus der Empfehlungen

Der Prototyp bildet die Basis für die softwaretechnische Umsetzung einer automatisierten Überprüfung der Empfehlungen der DLR Software Engineering Rahmenrichtlinie hinsichtlich ihrer Anwendungsklassen (siehe 2.2.1). Er dient als Ausgangspunkt für die Etablierung einer effizienten und zuverlässigen Methode zur Überwachung und Verbesserung der Softwarequalität. Da es keine Empfehlungen für Anwendungsklasse 0 gibt, wird diese nicht vom Prototyp betrachtet und auch keine Einordnung in diese Klasse vorgenommen. Aufgrund des prototypischen Charakters der Implementierung werden nicht alle Empfehlungen im Detail behandelt, sondern ausschließlich diejenigen, die automatisiert überprüft werden können. Im Folgenden wird erläutert, welche Empfehlungen innerhalb der einzelnen Themenbereiche durch den Prototyp automatisiert überprüft werden können und welche aufgrund technischer oder konzeptioneller Einschränkungen nicht abgedeckt werden können.

3.1.1 Qualifizierung

In diesem Themenschwerpunkt der Guidelines liegt der Fokus auf der Überprüfung und Sicherstellung des Wissens der EntwicklerInnen [vgl. 2, S.12]. Diese Aspekte werden jedoch im Prototyp nicht schwerpunktmäßig behandelt, da eine umfassende Bewertung der individuellen Entwicklerqualifikationen anhand eines GitLab-Repositories nicht möglich ist.

3.1.2 Anforderungsmanagement

Die Empfehlungen zielen darauf ab, Anforderungen möglichst umfassend zu erfassen und zu erfüllen. Der aktuelle Prototyp kann jedoch die vollständige Überprüfung der Anforderungen noch nicht leisten. Der Fokus liegt derzeit auf den folgenden Aspekten des Anforderungsmanagements:

Erstellung eines Glossars: GitLab sieht für ein Glossar keine spezielle, standardisierte Glossardatei vor. Trotzdem sollte ein Glossar als integraler Bestandteil der Projekt- oder Entwicklungsdokumentation zentral und eindeutig im GitLab-Repository angelegt werden. Der Prototyp sucht daher gezielt nach der Existenz eines Glossars in der Root-Ebene des Projekts, da diese als zentrale Anlaufstelle für wichtige Dokumentationsdateien gilt. Dabei wird auf Dateinamen geachtet, die typischerweise für ein Glossar verwendet werden. Eine

inhaltliche Analyse der Datei wird nicht durchgeführt, da dies komplex und fehleranfällig wäre, insbesondere wenn das Glossar unkonventionell strukturiert ist. Zudem gibt es keine exakten Vorgaben in den Empfehlungen, wie ein Glossar aufgebaut sein muss. Daher konzentriert sich der Prototyp auf die einfache und zuverlässige Erkennung der Datei selbst.

Eindeutige Anforderungserfassung: In vielen Projekten werden Ticket-Systeme genutzt, um Anforderungen, Änderungsanträge und Fehler zu verfolgen. GitLab bietet ein komfortables Ticket-System, das eng mit den Repositories, Merge Requests und anderen Funktionen integriert ist. Der Prototyp überprüft zunächst, ob im GitLab-Projekt Issues und Epics [vgl. 10] vorhanden sind, da deren Vorhandensein auf ein strukturiertes Anforderungsmanagement hinweist. Issues dienen zur Dokumentation einzelner Aufgaben, Anforderungen oder Fehler, während Epics größere Aufgaben oder Projekte in kleinere, handhabbare Einheiten unterteilen. Anschließend prüft der Prototyp die zugeordneten Labels. Labels sind in der Regel als separate Attribute innerhalb der jeweiligen Issues und Epics gespeichert. Der Prototyp untersucht, ob diese Labels Informationen zum Anforderungsmanagement enthalten, indem er deren Inhalte und Bezeichnungen auf gängige Hinweise wie Requirements, Features, Specifications oder User Stories überprüft.

3.1.3 Software-Architektur

Im Rahmen dieser Arbeit wird bewusst auf eine detaillierte Überprüfung der Software-Architektur durch den Prototyp verzichtet. Diese Entscheidung basiert darauf, dass die automatisierte Prüfung der Umsetzung eines geplanten Architekturmodells oft mit erheblichen Herausforderungen verbunden ist. Es wird daher, ebenso wie bei der Qualifizierung, auch die Architektur nicht berücksichtigt. Die Automatisierung in diesem Bereich könnte keine hinreichend aussagekräftigen Ergebnisse liefern.

3.1.4 Design und Implementierung

Bezüglich dieser Phase des Entwicklungsprozesses konzentriert sich der Prototyp auf die gezielte Evaluierung automatisierter Codeanalyse-Tools. Dies ist unmittelbar und messbar und trägt direkt zur Qualitätssicherung bei. Die folgenden zentralen Prüfungsaspekte werden betrachtet:

Einhaltung der Programmiersprachregeln und deren automatisierte Prüfung: GitLab bietet die Möglichkeit, Style-Checker wie *Black*, *Flake8* und *Pylin*t über seine integrierten CI/CD-Pipelines [vgl. 9] unkompliziert in den Entwicklungsprozess zu integrieren. Diese Tools ermöglichen es, den Code automatisch auf Einhaltung von Programmierstandards und Stilrichtlinien zu überprüfen. Der Prototyp nutzt diese Funktionalität, indem er die Pipeline-Konfigurationsdateien der jeweiligen GitLab-Repositories abruft und analysiert ob solche Tools aufgerufen werden. Dadurch kann er feststellen, ob mittels solcher Tools die kontinuierliche Einhaltung der Programmiersprachregeln sicherstellt wird.

Existenz von Modultests: Python ermöglicht die Implementierung von Modultests mithilfe gängiger Testframeworks wie `pytest` und `unittest`, die eine strukturierte und automatisierte Testdurchführung unterstützen. Der Prototyp nutzt diese Funktionalität, indem er direkt im Quellcode der Python-Module nach Testklassen sucht. Dabei wird analysiert, ob diese Klassen Unit-Tests enthalten, die mit den genannten Frameworks umgesetzt sind. Dieser Ansatz stellt sicher, dass die Module über eine geeignete Testinfrastruktur verfügen und aktiv getestet werden.

3.1.5 Änderungsmanagement

Dieser Abschnitt formuliert Empfehlungen zur optimalen Nutzung des Versionierungssystems, um eine effektive Nachverfolgbarkeit (Traceability) sicherzustellen. Dabei werden insbesondere Empfehlungen fokussiert, die sich in konkrete Prüfungen umsetzen lassen. Zentrale Fragen hierbei sind: Werden die Funktionen von GitLab tatsächlich genutzt? Sind alle erforderlichen Dokumente zur Nachverfolgbarkeit vorhanden? Sind diese Dokumente korrekt und einheitlich benannt?:

Existenz einer Readme- oder Contributing-Datei: Die Konventionen, welche von Gitlab unterstützt wird, ermöglicht die Speicherung von README-Dateien im Markdown-Format unter dem Namen `README.md` sowie von Contributing-Dateien, die häufig als `CONTRIBUTING.md` abgelegt werden. Diese Dateien enthalten Anleitungen und Richtlinien zur Mitarbeit. Während Markdown [8] bevorzugt wird, sind auch andere Formate wie einfache Textdateien möglich. Der Prototyp nutzt diese Konventionen, indem er überprüft, ob die entsprechenden Dateien vorhanden sind. Wenn die Dateien im Standardformat vorliegen, kann der Prototyp zusätzlich den Inhalt auf bestimmte Schlüsselworte prüfen.

Existenz einer Roadmap: GitLab bietet die Möglichkeit, Roadmaps [13] visuell darzustellen, indem Milestones und Epics in einer übersichtlichen Roadmap-Ansicht angeordnet werden. Milestones repräsentieren wichtige Projektmeilensteine, während Epics größere, zusammenhängende Aufgabenbereiche darstellen. Die Roadmap zeigt dabei an, welche Milestones erreicht wurden, welche noch ausstehen und wie die Epics miteinander verknüpft sind. Um diese Funktionalität effektiv zu nutzen, muss der Prototyp überprüfen, ob Milestones und Epics im GitLab-Projekt angelegt sind.

Führen eines Changelogs: GitLab bietet die Möglichkeit, ein Changelog [6] zu erstellen, das sämtliche Änderungen und Verbesserungen in den verschiedenen Versionen eines Projekts dokumentiert. Diese unterstützenden Funktionen ermöglichen eine strukturierte und übersichtliche Darstellung der Änderungen. Der Prototyp nutzt diese Funktionen, indem er überprüft, ob ein entsprechendes Changelog im Projekt vorhanden ist.

Prüfung der Branch-Namen von Gitlab: Bewährte Praktiken umfassen konsistente Namenskonventionen und die eindeutige Kennzeichnung des Hauptzweigs, der die finale Version enthält. GitLab unterstützt Standards für die Namenskonventionen von Hauptzweigen, wobei die empfohlenen Namen `main`, `master`, und `develop` [2] sind. Der Prototyp nutzt

diese Standards, indem er über die GitLab-API die Namen der vorhandenen Branches im Projekt abruft und überprüft, ob der Hauptzweig entsprechend benannt ist.

3.1.6 Software-Test

Im Bereich des Software-Tests gibt es zahlreiche Empfehlungen, die zur Sicherstellung der Qualität und Zuverlässigkeit einer Software beitragen. Die Bewertung der Testqualität ist jedoch oft komplex und hängt stark von der spezifischen Software sowie einem tiefen Verständnis ihrer Funktionalität ab. Daher liegt der Fokus bei der automatisierten Überprüfung zunächst auf einem technisch gut umsetzbaren Aspekt:

Systematische Durchführung von Funktionstests: Um sicherzustellen, dass Test-Frameworks wie `pytest` oder `unittest` ordnungsgemäß verwendet werden, ist es erforderlich, die GitLab CI/CD-Konfiguration zu überprüfen. Die Integration dieser Frameworks in die CI/CD-Pipelines zeigt an, dass eine systematische und konsistente Durchführung von Funktionstests im automatisierten Entwicklungsprozess durchgeführt wird. Dazu werden die relevanten Konfigurationsdateien aus dem Repository abgerufen. Anschließend werden diese Dateien systematisch durchsucht, um spezifische Job-Skripte zu identifizieren, die Kommandos zum Start der Test-Frameworks während des Build- und Testprozesses enthalten.

3.1.7 Release-Management

Im Rahmen der Überprüfung der Empfehlungen des Release-Managements liegt der Schwerpunkt auf dem finalen Releasepaket, um sicherzustellen, dass es den festgelegten Anforderungen und Qualitätsstandards entspricht. In GitLab ermöglichen automatisierte Systeme, insbesondere CI/CD-Pipelines [vgl. 9], den gezielten Zugriff auf und die Verarbeitung der Endversion des Pakets. Diese Vorgehensweise unterstützt eine präzise Evaluierung der Konformität und Qualität des Endprodukts, da das finale Release die tatsächliche Version darstellt, die an die Benutzer bereitgestellt wird.

Der Fokus liegt dabei auf den folgenden Aspekten:

Eindeutiger Release-Name: Die Gewährleistung der Eindeutigkeit und Unterscheidbarkeit des Release-Namens ist wichtig, damit Dritte die korrekte Softwareversion identifizieren können. GitLab ermöglicht die Erstellung von Releases, bei denen jede Version mit einem klaren Namen und detaillierten Informationen versehen wird. Dies umfasst den Release-Namen, eine Beschreibung und zusätzliche Metadaten, die sicherstellen, dass jede Version eindeutig identifizierbar ist.

Auflistung aller Bestandteile des Release-Pakets: Alle Komponenten und Dateien, die im Release enthalten sind, müssen eindeutig angegeben sein. Das ist insbesondere für Compliance- und Audit-Zwecke wichtig. GitLab unterstützt die Erstellung von Software-Bill-of-Materials-Berichten (SBOM) über ein Dependency-Scanning [vgl. 7], die eine detaillierte Auflistung aller in einem Projekt verwendeten Komponenten und deren Lizenzen bieten [5].

Regelmäßige Veröffentlichung von Releases: Die Überwachung der Intervalle, in denen Releases veröffentlicht werden, ist wesentlich für eine sinnvolle und konsistente Aktualisierung der Software. GitLab speichert eine detaillierte Historie aller Commits, Merge-Requests und Releases, wodurch es einfach ist, die Zeitabstände zwischen den verschiedenen Releases nachzuvollziehen.

3.1.8 Automatisierung und Abhängigkeitsmanagement

Die Auswahl der zu überprüfenden Empfehlungen in diesem Bereich orientiert sich an ihrer Bedeutung für den gesamten Entwicklungsprozess und ihrer Eignung für eine automatisierte Überprüfung in GitLab-Projekten. Die Fokussierung auf diese Punkte ermöglicht eine präzise und effiziente Validierung, da sie sich gut in die vorhandenen CI/CD-Pipelines [vgl. 9] integrieren lassen und eine umfassende Überprüfung der Qualität und Integrität der Software gewährleisten:

Überprüfung der Nutzung automatisierter Builds: Verschiedene Schritte des Build-Prozesses (Build, Test, Deployment, Release) können in die GitLab CI/CD-Pipelines integriert werden. In Python-Projekten erfolgt die Automatisierung häufig auch über spezifische Konfigurationsdateien, die Abhängigkeiten und Build-Skripte definieren. Diese Dateien ermöglichen es, die Build- und Test-Umgebung systematisch und reproduzierbar zu konfigurieren. Die CI/CD-Pipelines in GitLab können so eingerichtet werden, dass sie diese Konfigurationsdateien nutzen, um automatische Tests, Builds und Deployments durchzuführen.

Überprüfung von Lizenzabhängigkeiten: GitLab bietet eine integrierte Lizenzmanagement-Funktion mit 'Dependency Scanning' [vgl. 7], die automatisch Verstöße, Lücken und Abhängigkeiten in Bezug auf Lizenzen identifiziert. Diese Lizenzüberprüfung ist in den kontinuierlichen Integrationsprozess von GitLab eingebunden. Zusätzlich kann eine Lizenzdatei, typischerweise `LICENSE` oder `LICENSE.txt`, direkt im Root-Verzeichnis eines GitLab-Repositories abgelegt werden. Diese Datei definiert die Lizenzbedingungen, unter denen der im Repository enthaltene Code veröffentlicht wird.

3.2 Randbedingung

Fokus auf Python-Projekte: Bei der Entwicklung des Prototypen liegt der Fokus zunächst speziell auf Python-Projekten. Python ist eine der am weitesten verbreiteten Programmiersprachen, insbesondere im Bereich der wissenschaftlichen Softwareentwicklung. Sie wird für eine Vielzahl von Anwendungen eingesetzt, von Webentwicklung über Datenanalyse bis hin zu künstlicher Intelligenz. Durch die Bewertung von Python-Projekten wird eine breite Palette von Anwendungsgebieten abgedeckt. Außerdem verfügt Python über ein reichhaltiges Ökosystem von Bibliotheken, Frameworks und Tools. Der Prototyp verwendet die Python-Bibliothek `python-gitlab`, um auf diese API zuzugreifen. Durch die Integration von `python-gitlab` kann der Prototyp gezielt automatisierte Überprüfungen und Interaktionen mit GitLab-Projekten durchführen.

Grenzen automatisierter Bewertungen: Einige Aspekte der Projektqualität, wie beispielsweise die Benutzerfreundlichkeit der Schnittstellen, können nur schwer automatisiert werden und erfordern menschliches Urteilsvermögen. Es ist außerdem wichtig, dass automatisierte Bewertungen als Ergänzung zu manuellen Überprüfungen betrachtet werden, um ein umfassendes Bild der Projektqualität zu erhalten. Automatisierte Bewertungen können zu falschen Schlussfolgerungen führen, wenn sie nicht sorgfältig konfiguriert sind oder wenn bestimmte Aspekte der Projekte nicht angemessen berücksichtigt werden. Daher ist es notwendig, die Ergebnisse automatisierter Bewertungen kritisch zu hinterfragen und zu validieren. Einige Empfehlungen wurden aufgrund der beschriebenen Einschränkungen von vornherein ausgeschlossen, wie zum Beispiel die folgenden Empfehlungen:

- "EQA.1: Der Software-Verantwortliche kennt die verschiedenen Anwendungsklassen und weiß welche für seine Software anzustreben ist"[2, S.12]

Diese Empfehlung erfordert, dass zunächst Interviews mit den Verantwortlichen durchgeführt werden. Da der Prototyp jedoch ausschließlich auf die Überprüfung der GitLab-Umgebung und nicht auf menschliche Faktoren fokussiert ist, gestaltet sich die Überprüfung dieser Art von Empfehlungen als problematisch.

- "EDI.4: Die Implementierung spiegelt die Software-Architektur wider"[2, S.27]

Diese Empfehlung würde eine vorherige Analyse der Architekturdokumentation sowie eine anschließende Analyse der Implementierung und den Vergleich beider erfordern. Ohne den Einsatz von Machine-Learning-Tools wäre eine solche Analyse schwierig durchzuführen. Andere Empfehlungen sind derzeit nur eingeschränkt durch automatisierte Verfahren überprüfbar. In der detaillierten Beschreibung der Umsetzung spezifischer Anforderungen werden die Herausforderungen und Hindernisse bei der Automatisierung ausführlicher dargelegt.
Flexibilität: Der entwickelte Prototyp stellt den Anfang eines iterativen Prozesses dar, der auf kontinuierliche Weiterentwicklung und Verbesserung ausgerichtet ist. Die Architektur des Prototyps ist modular gestaltet, um zukünftige Erweiterungen zu unterstützen und flexibel auf neue Anforderungen reagieren zu können. Während der Fokus zunächst auf Python-Projekten liegt, ist der Prototyp so konzipiert, dass er später auch für andere Programmiersprachen und Projektarten erweitert werden kann. Durch die flexible Struktur lassen sich neue Prüfkriterien und Anpassungen konfigurativ integrieren. Auch die grafische Ergebnisaufbereitung ist als eigenständige Komponente ausgelegt, sodass sie unabhängig von der Prüflogik erweitert werden kann. In Kapitel 4 wird auf Umsetzung genauer eingegangen.

3.3 Funktionaler Ansatz

In diesem Kapitel wird der funktionale Ansatz des Prototyps detailliert erläutert, der den gesamten Ablauf der automatisierten Evaluierung der Projekte sowie die Struktur der Ergebnisaufbereitung umfasst.

3.3.1 Genereller Programmablauf

Der Prototyp folgt einem klar definierten Ablauf, um eine systematische und reproduzierbare Vorgehensweise zu gewährleisten. Die einzelnen Schritte umfassen:

Authentifizierung bei GitLab: Diese Phase beinhaltet die Einrichtung einer sicheren Authentifizierung des Skripts beim GitLab-Server mittels eines privaten Tokens. Dadurch wird der Zugriff auf die Python-Bibliothek für die GitLab-API `python-gitlab` ermöglicht, um relevante Projektdaten abzurufen.

Laden der Projektdaten : Das Laden der Projektdaten kann direkt aus GitLab-Repositories oder aus einer zuvor erstellten JSON-Datei erfolgen, die relevante und zugängliche Projekte enthält. Die Nutzung der JSON-Datei ermöglicht eine schnellere Überprüfung, da nur spezifische Projekte geladen werden, ohne alle Repositories durchsuchen zu müssen. Diese Methode ist vorteilhaft, da die Datei nur bei Änderungen, wie dem Hinzufügen oder Entfernen von Projekten, aktualisiert werden muss, was den Überprüfungsprozess effizienter und zeitsparender gestaltet.

Anforderungsüberprüfung : Für jedes identifizierte Projekt werden spezifizierte Anforderungen anhand definierter Überprüfungsfunktionen durchgeführt. Diese verschiedenen Arten der Anforderungsüberprüfung hat der Prototyp:

- **Überprüfung der Dateiexistenz**: Es wird überprüft, ob eine spezifische Datei mit einem vorgegebenen Namen im GitLab-Repository vorhanden ist.
- **Nutzung von GitLab-Features**: Zur Erfüllung bestimmter Empfehlungen wird geprüft, ob in GitLab-Projekten spezifische Features verwendet werden. Dabei konzentriert sich die Überprüfung darauf, ob die Funktionalitäten überhaupt genutzt werden, ohne die Qualität oder Effizienz ihrer Anwendung zu bewerten. Mittels der Python-Bibliothek `python-gitlab` können API-Anfragen an das GitLab-Projekt gesendet werden, um Daten zu den verwendeten Features abzurufen
- **Code-Analyse**: Der Quellcode wird analysiert, um das Vorhandensein und die Implementierung bestimmter Funktionen oder Code-Elemente zu identifizieren und zu bewerten.
- **Analyse der GitLab CI/CD-Konfiguration**: Die Konfigurationsdatei für die kontinuierliche Integration und Bereitstellung wird auf spezifische Inhalte, wie bestimmte Job- oder Stage-Definitionen, überprüft. Dabei erfolgt die Suche zunächst nach der Standarddatei, die üblicherweise als `.gitlab-ci.yml` bezeichnet wird. Zusätzlich werden alle inkludierten Dateien berücksichtigt, um eine vollständige Analyse der Konfiguration zu gewährleisten. Die CI/CD-Jobs, die die zentralen Aufgaben innerhalb der Pipeline ausführen, werden daraufhin analysiert. Diese Jobs sind in Stages organisiert, die eine logische Abfolge von Aufgaben darstellen und den gesamten Integrations- und Bereitstellungsprozess strukturieren.
- **Verwendung von Lookup-Tabellen**: Für verschiedene Überprüfungen im GitLab-Projekt kommen Lookup-Tabellen zum Einsatz, um Schlüsselbegriffe und spezifische

Suchkriterien zu verwalten. Diese Tabellen erleichtern die Suche nach bestimmten Dateien, Funktionen und Konfigurationselementen und können flexibel angepasst und erweitert werden.

Ausgabe : Die Ergebnisse der Überprüfungen werden projektweise in einem strukturierten Datenformat abgelegt und anschließend in eine übersichtliche JSON-Datei konvertiert, die als Input für die weitere Ergebnisaufbereitung, wie Mapping oder grafische Darstellung, dient. Dieses Zwischenformat ermöglicht zudem eine archivierende Funktion, da es den späteren Vergleich der Projekte über verschiedene Entwicklungsstände hinweg erlaubt und so Rückschlüsse auf die Anwendung der Guidelines im Zeitverlauf bietet.

Mapping auf Anwendungsklassen: Das Ergebnis der Anforderungsüberprüfung wird anhand eines Mappings den entsprechenden Anwendungsklassen zugewiesen. Dieses Mapping basiert auf empfehlungsnahen Kriterien, die den einzelnen Anwendungsklassen zugeordnet sind. Gleichzeitig fungieren diese Kriterien als '*Schalter*', um gezielte Anforderungsüberprüfungen durchführen zu können. Als Ergebnis können so, durch das Erfüllen der jeweiligen Kriterien, die Projekte den entsprechenden Anwendungsklassen zugeordnet werden.

Erstellen von Kreisdiagrammen: Zusätzlich zu den Ergebnisdateien wird für jede Empfehlung ein Kreisdiagramm erstellt, das veranschaulicht, welcher Prozentsatz der Projekte die jeweilige Empfehlung erfüllt. Diese Diagramme bieten eine visuelle Übersicht über den Erfüllungsgrad der Empfehlungen und ermöglichen einen schnellen Vergleich zwischen den verschiedenen Anwendungsklassen.

Der Programmablauf wird nachfolgend vereinfacht in einem Aktivitätsdiagramm in Abbildung 1 dargestellt.

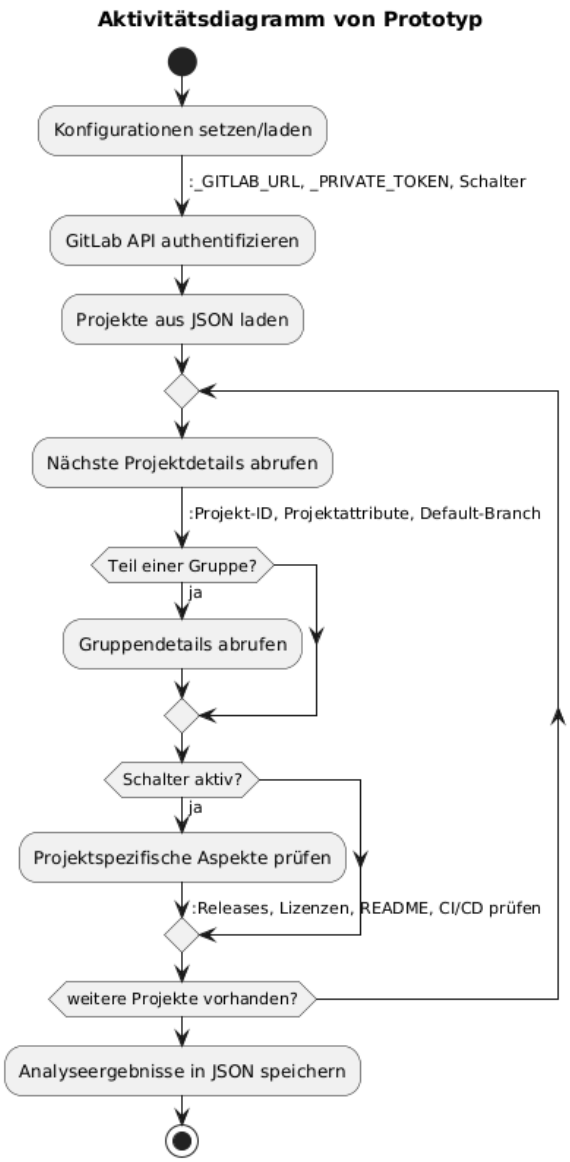


Abbildung 1: Aktivitätsdiagramm zur Darstellung des Ablaufs im Prototypen. Das Diagramm veranschaulicht den gesamten Prozess der automatisierten Überprüfung, beginnend bei der Authentifizierung über die GitLab API, gefolgt vom Abrufen und Prüfen projektspezifischer Aspekte, bis hin zur Speicherung der Analyseergebnisse.

3.3.2 Ansatz zur Evaluation

Alle analysierten Projekte sind Python-basierte Softwareentwicklungsprojekte, wodurch eine konsistente Grundlage für die Bewertung der Erfüllung der Empfehlungen gegeben ist. Zunächst erfolgt die Überprüfung spezifischer Projekte, um die Zuverlässigkeit und Funktionalität des entwickelten Prototyps zu testen. Dieser erste Schritt ermöglicht es, potenzielle Schwächen oder Verbesserungsmöglichkeiten in der Prototyp-Entwicklung zu identifizieren. Nachdem die Funktionalität des Prototyps validiert wurde, folgt eine umfassende Analyse aller verfügbaren Projekte. Dieser Schritt dient dazu, den allgemeinen Entwicklungsstand der Projekte zu erfassen und ein umfassenderes Bild der Erfüllung der Empfehlungen zu erhalten. Durch diese zweistufige Vorgehensweise wird sichergestellt, dass sowohl die Effizienz des Prototyps als auch die

Klassifizierungsgenauigkeit der gesamten Projektlandschaft bewertet werden können.

Ersteinschätzung des Prototyps anhand ausgewählter Projekte: Durch die gezielte Auswahl spezifischer Projekte wird manuell überprüft, ob die automatische Klassifizierung mit den tatsächlichen Eigenschaften der Projekte übereinstimmt. Diese Überprüfung liefert schnelles Feedback zur Leistungsfähigkeit des Prototyps und bewertet die Zuverlässigkeit der Klassifizierungen. Zur Effektivität des Prototyps werden Projekte mit erwarteter hoher und niedriger Klassifizierung ausgewählt. Diese Auswahl ermöglicht die Bewertung der Klassifizierungsfähigkeit für Projekte, die unterschiedlich viele Empfehlungen erfüllen.

- **Umfangreiches, aktives Projekt:** Das ausgewählte Python-Projekt gehört zu den größten, auf die Zugriff besteht, und wird aktiv weiterentwickelt. Es sollte deshalb viele Empfehlungen erfüllen. Bekannt ist, dass das Entwicklungsteam versucht, die Empfehlungen in ihre Entwicklungsprozesse zu integrieren.
- **Kleineres, aktives Projekt:** Dieses kleinere Python-Projekt könnte viele Empfehlungen erfüllen, jedoch möglicherweise nicht die der höchsten Anwendungsklasse. Diese Annahme beruht darauf, dass Projekte, die nicht für kritische Anwendungen konzipiert sind, oft nicht die höchste Klassifizierung anstreben.
- **Älteres Projekt:** Dieses Python-Projekt besteht aus älterem Quellcode, der offenbar einer anderen Projektstruktur entstammt und keine GitLab-Anbindung aufweist. Daher eignet es sich, um aufzuzeigen, dass möglicherweise nur sehr wenige Empfehlungen erfüllt werden, was zu einer niedrigen Klassifizierung des Projekts führt

Klassifizierung aller Projekte nach Anwendungsklassen: Die Überprüfung aller verfügbaren Projekte ermöglicht die Validierung des Prototyps und bietet gleichzeitig einen Überblick über die gesamte Projektlandschaft. Dies führt zur Erfassung statistischer Daten, die nicht nur die allgemeine Qualität der Projekte widerspiegeln, sondern auch Erkenntnisse für die Optimierung des Prototyps liefern. Zusätzlich werden Korrelationen zwischen verschiedenen Kriterien untersucht, um potenzielle Zusammenhänge und Einflussfaktoren zu identifizieren. Zur anschaulichen Darstellung der Ergebnisse werden automatisch Diagramme erstellt, die einen klaren Überblick über die gewonnenen Resultate bieten. Diese finden sich auch im Ergebnisteil [5] wieder.

Implementierung der Empfehlungsüberprüfung im Prototyp

In diesem Kapitel wird die Implementierung der im Prototyp umgesetzten Empfehlungen beschrieben. Jede Empfehlung wird zunächst anhand der Formulierung aus den Richtlinien [2] eingeführt. Anschließend folgt der jeweilige Lösungsansatz sowie die technische Umsetzung im Prototyp. Falls erforderlich, werden abschließend mögliche Verbesserungen für die automatisierte Überprüfung aufgezeigt, um die aktuellen Grenzen des Prototyps darzustellen. Der vollständige Quellcode des Projekts ist im GitLab-Repository verfügbar und kann über den folgenden Verweis [18] eingesehen werden. Aus Datenschutzgründen wurde ausschließlich die anonymisierte Übersicht der untersuchten Python-Projekte mit den zugehörigen Analyseergebnissen veröffentlicht.

4.1 EAM.8: Erstellung eines Glossars

Empfehlung	EAM.8: Es existiert ein Glossar, das die wesentlichen Begrifflichkeiten und Definitionen beschreibt.
Anwendungsklasse	2
Lösungsansatz	• Überprüfung der Existenz eines Glossars

Umsetzung des Prototyps

Namensgebung: Es gibt keine festgelegten Namenskonventionen für ein Glossar (siehe 3.1.2). Dies macht es schwierig, eine automatisierte Überprüfung zu implementieren, die alle möglichen Variationen korrekt identifiziert und bewertet. Eine Test-Suchfunktion, die über sämtliche Dateien des Projekt-Repositories iteriert, sucht nach gängigen Dateinamen, wie `'GLOSSARY.md'` oder `'GLOSSARY.txt'`. Die Implementierung im Prototyp auf Basis eines vergleichsweise kleinen Datensatzes erwies sich als nicht erfolgreich. Dennoch diente das grundlegende Konzept dieser Dateisuche als Ausgangspunkt für andere Implementierungen im Rahmen anderer Empfehlungen.

Vorschlag für Adaption

Da GitLab keine integrierte Funktion für die Erstellung eines Glossars bietet, wird empfohlen, eine Datei mit dem Namen `GLOSSARY.md` im Root-Verzeichnisdes Repository anzulegen. Diese Datei könnte in Markdown formatiert sein, da es eine flexible Formatierung ermöglicht und die Lesbarkeit erleichtert. Es ist sinnvoll, ein einfaches Template für das Glossar zu erstellen, das die Struktur vorgibt und eine automatische Überprüfung erleichtert.

4.2 EÄM.2: Existenz einer Readme- oder Contributing-Datei

Empfehlung	EÄM.2: Die wichtigsten Informationen, um zur Entwicklung beitragen zu können, sind an einer zentralen Stelle abgelegt.
Anwendungsklasse	1
Lösungsansatz	• Überprüfung der Readme-Datei • Überprüfung der Existenz einer Contributing-Datei

Umsetzung des Prototyps in Gitlab

- **Überprüfung der Dateiexistenz (siehe 3.3.1):** Die Prüfung der Existenz einer Datei erfolgt durch eine Suche durch die Repositories. Dafür wird eine Lookup-Tabelle verwendet, eine Liste mit gängigen Dateinamen, wie `'README'`, `'README.md'`, `'README.txt'`, `'README.rst'`, `'CONTRIBUTING'`, `'CONTRIBUTING.md'`.
  Da "typischerweise legt man diese in der Datei `"README"` oder `"CONTRIBUTING"` ab."[2]
- **Prüfung auf Plausibilität:** Die automatisierte Überprüfung der Inhalte von Readme- und Contributing-Dateien durch den Prototyp ist schwierig, da diese stark variieren können und nicht standardisiert sind. Die Dateien können unterschiedliche Informationen wie Projektbeschreibungen, Anleitungen und Richtlinien enthalten. Eine automatische Prüfung müsste diese Vielfalt interpretieren und bewerten, was technisch komplex ist. Daher konzentriert sich die automatisierte Überprüfung in diesem Fall nur auf spezifische Aspekte der `README.md`-Datei. Markdown-Dateien bieten den Vorteil, dass sie strukturierte Informationen mit Formatierungen enthalten, die bei der Analyse durch Python-Skripte effizient verarbeitet werden können. Hier wird prototypisch überprüft, ob einige Schlüsselwörter vorhanden sind, was einen einfacheren und direkteren Ansatz darstellt. Dies ermöglicht eine grundlegende Überprüfung ohne die Herausforderungen einer detaillierten Inhaltsbewertung.
- **Schlüsselwörter:** Project Name, Description, Overview, Summary, Installation, How to Install, Setup, Getting Started, Usage, How to Use, Examples, Quick Start, Contributing, How to Contribute, Development, Contribution Guidelines, Authors, Contributors, Credits, Acknowledgements, License, MIT License, GNU General Public License, Apache License, Contact, Support, Email, Get in Touch, Changelog, Release Notes, Version History, Dependencies, Requirements, Prerequisites, FAQ, Help, Troubleshooting, Common Issues

Vorschlag für Adaption

Für die Projekt-Dokumentation wird die Verwendung der Datei `README.md` und für Dokumentation der Mitwirkungsrichtlinien wird die Verwendung der Datei `CONTRIBUTING.md` im Hauptverzeichnis des Gitlab-Projektes empfohlen. Um die Dateien automatisch auf Richtigkeit zu überprüfen, ist es sinnvoll, bestimmte Schlüsselwörter und Strukturvorgaben fest zu definieren.

4.3 EÄM.6: Führen eines Changelogs

Empfehlung	EÄM.6: Es existiert eine detaillierte Änderungshistorie (Changelog), aus der hervorgeht, welche Funktionen und Fehlerbeseitigungen in welcher Software-Version enthalten sind.
Anwendungsklasse	2
Lösungsansatz	• Überprüfung der Existenz eines Changelogs

Umsetzung des Prototyps

Existenz einer Changelog-Datei: Der Prototyp nutzt eine Lookup-Tabelle, um sicherzustellen, dass relevante Dateinamen erkannt und validiert werden. Dateien mit dem Namen `"CHANGELOG.md"` oder `"CHANGELOG.rst"` werden gezielt im Repository gesucht, da dies die üblichen Bezeichnungen sind (siehe 3.1.5). Diese Datei dient als wesentlicher Indikator für die Dokumentation von Änderungen und wird zur Nachverfolgung von Versionshistorien verwendet.

Vorschlag für Adaption

Für die Dokumentation von Änderungen im GitLab-Projekt wird die Verwendung der Datei `CHANGELOG.md` im Hauptverzeichnis empfohlen. Um den Inhalt einer Changelog-Datei zu vereinheitlichen, empfiehlt es sich, eine klare und konsistente Struktur festzulegen, und den Inhalt in definierte Abschnitte zu unterteilen, wie `added`, `changed`, `removed` und `fixed`. Dies ermöglicht eine automatische Überprüfung der Einträge, um sicherzustellen, dass alle Änderungen korrekt dokumentiert sind.

4.4 EÄM.9: Prüfung der Branch-Namen

Empfehlung	EÄM.9: Falls mehrere gemeinsame Entwicklungszweige existieren, lässt sich deren Zweck einfach erschließen.
Anwendungsklasse	2
Lösungsansatz	• Überprüfung des Namens des Hauptbranches • Auflistung der Namen der weiteren Branches

Umsetzung des Prototyps

- **Hauptbranch:** Der Prototyp ruft den Namen des Hauptbranches eines Projektes ab und überprüft, ob er korrekt benannt wurde (siehe 3.1.5). Diese Überprüfung ist ausreichend, um die empfohlene Namenskonvention zu gewährleisten.
- **Weitere Entwicklungsbranches:** Die Namen der weiteren Entwicklungsbranches werden ebenfalls erfasst und aufgelistet. Diese Namen müssen manuell auf Sinnhaftigkeit bewertet werden. Eine automatisierte Überprüfung aller Branchnamen ist schwierig, wenn keine konsistenten Namenskonventionen vorliegen. Solche Überprüfungen könnten entweder sinnvolle Branchnamen fälschlicherweise als ungeeignet markieren (false negatives) oder unpassende Namen als akzeptabel werten (false positives).

Vorschlag für Adaption

Um eine konsistente und nachvollziehbare Struktur für Entwicklungsbranches in GitLab-Projekten zu gewährleisten, sollten spezifische Namenskonventionen eingeführt werden. Hier bietet GitLab eine automatisierte Namensgebung an, wenn ein Branch aus einem Issue entsteht.

4.5 EAM.2: Eindeutige Anforderungserfassung

Empfehlung	EAM.2: Funktionale Anforderungen sind zumindest mit eindeutiger Kennung, Beschreibung, Priorität, Ursprung und Ansprechpartner erfasst.
Anwendungsklasse	2
Lösungsansatz	• Überprüfung auf vorhandenen Issue-Tracker oder Epic • Überprüfung der verwendeten Labels

Umsetzung des Prototyps

- **Überprüfung von Issues und Epics:** Mithilfe der `python-gitlab`-Bibliothek (siehe 3.1.2) werden Daten über Issues und Epics aus dem GitLab-Projekt abgerufen. Diese Anfragen ermöglichen es, eine Liste der bestehenden Issues und Epics zu erhalten und deren Verwendung im Projekt grundsätzlich zu verifizieren.
- **Überprüfung der verwendeten Labels:** Der Prototyp extrahiert die Labels aus den Datenstrukturen der Issues und Epics, um deren Relevanz für Anforderungen oder Aufgaben zu verifizieren. Dafür wird mit einer Lookup-Tabelle verglichen, die Begriffen, die mit Anforderungen in Verbindung gebracht werden, enthält, wie: `'requirement'`, `'feature'`, `'specification'`, `'user-story'`, `'NFR'`, `'architecture'`, `'epic'`.

Vorschlag für Adaption

Die Nutzung der Issue-Tracker-Funktionalität in GitLab ermöglicht die effiziente Erfassung und Verwaltung funktionaler Anforderungen, die sich nahtlos in den Entwicklungsprozess integrieren lassen. Für eine spätere Implementierung der automatisierten Anforderungserfassung können die Issues, die funktionalen Anforderungen entsprechen, mit spezifischen Labels oder Tags wie `'Requirement'`, `'Feature'`, `'User Story'` gekennzeichnet sein. Zusätzlich können auch die Issue-Typen oder Kategorien verwendet werden, um funktionale Anforderungen von anderen Issue-Arten zu unterscheiden. Durch klare Regelungen und eine konsistente Kennzeichnung der Issues wird sichergestellt, dass der Automatisierungsprozess die relevanten Issues korrekt erkennt und ihren Bearbeitungsstatus akkurat verfolgt, was letztlich zur Nachvollziehbarkeit des Projektfortschritts beiträgt.

4.6 EÄM.4: Existenz einer Roadmap

Empfehlung	EÄM.4: Es existiert eine Planungsübersicht (Roadmap), die beschreibt, welche Software Versionen mit welchen Ergebnissen wann erreicht werden sollen.
Anwendungsklasse	2
Lösungsansatz	• Überprüfung des Roadmap-Features in GitLab • Überprüfung der Existenz einer Roadmap-Datei

Umsetzung des Prototyps

- **Roadmap-Feature:** Das Roadmap-Feature kann nicht direkt überprüft werden. Vielmehr prüft der Prototyp, ob die notwendigen Faktoren vorhanden sind (siehe 3.1.5). Das Vorhandensein von Meilensteinen und Epics innerhalb eines Projekts oder einer Gruppe dient als Indikator für die Nutzung der Roadmap-Funktionalität und signalisiert eine aktive Planung und Visualisierung der Entwicklungsaktivitäten.
- **Existenz einer Roadmap-Datei:** Es wird auch nach einer Roadmap-Datei gesucht, die möglicherweise unabhängig von GitLab erstellt wurde. Die automatisierte Suche nach einer Roadmap-Datei im Projekt kann als sinnvolle Ergänzung zur Verwendung des Roadmap-Features in GitLab betrachtet werden.

Vorschlag für Adaption

Um das Roadmap-Feature in GitLab effektiv zu nutzen und zeitgemäß zu agieren, sollten Projekte Epics und Issues verwenden sowie Meilensteine zuweisen, um eine strukturierte Planung zu ermöglichen. Durch diese Maßnahmen kann eine visuelle Darstellung und leicht verständliche Präsentation der geplanten Entwicklungsarbeiten über einen definierten Zeitraum hinweg generiert werden. Dadurch entsteht auch eine einheitliche Übersicht über mehrere Projekte.

4.7 EDI.1 und EDI.7: Einhaltung der Programmiersprachregeln und automatisierte Prüfung

Empfehlung	EDI.1: Es werden die üblichen Konstrukte und Lösungsansätze der gewählten Programmiersprache eingesetzt sowie ein Regelsatz hinsichtlich des Programmierstils konsequent angewendet.Der Regelsatz bezieht sich zumindest auf die Formatierung und Kommentierung
Anwendungsklasse	1
Empfehlung	EDI.7: Die Einhaltung einfacher Regeln des Programmierstils wird automatisiert geprüft bzw. sichergestellt.
Anwendungsklasse	2
Lösungsansatz	• Suche nach Jobs in der CI/CD-Konfiguration • Analyse der Script-Anweisungen

Umsetzung des Prototyps

- **Suche nach Jobs in der CI/CD-Konfiguration:** Der Prozess zur automatisierten Prüfung der Einhaltung von Programmierregeln erfolgt über die CI/CD-Pipeline (siehe 3.3.1). Der Prototyp analysiert die CI/CD-Konfigurationsdateien, um die darin definierten Jobs zu identifizieren.
- **Analyse der Script-Anweisungen:** Nachdem die relevanten Jobs in den Konfigurationsdateien identifiziert wurden, erfolgt die detaillierte Analyse der Script-Anweisungen innerhalb dieser Jobs. Eine Lookup-Tabelle, die gängige Python-Style-Checker wie `pylint`, `flake8`, `mypy`, `pyflakes` und `prospector` enthält, wird verwandt, um die Script-Anweisungen auf das Vorhandensein dieser Style-Checker zu überprüfen. Der Prototyp vergleicht die Script-Anweisungen in den CI/CD-Jobs mit den Einträgen in der Tabelle, um sicherzustellen, dass notwendige Style-Checker in der Pipeline integriert sind.

Vorschlag für Adaption

Um die automatisierte Überprüfung von Programmierregeln durch den Prototyp zu optimieren, sollten die CI/CD-Konfigurationen standardisierte Templates für die Integration von Style-Checkern nutzen. Diese Templates gewährleisten eine einheitliche Strukturierung der Jobs und erleichtern die Erkennung von Style-Checker-Anweisungen. Durch die Verwendung dieser Vorlagen können außerdem Anpassungen und Aktualisierungen schnell implementiert werden.

4.8 EDI.3: Existenz von Modultests

Empfehlung	EDI.3: Zu jedem Modul gibt es möglichst durchgängig Modultests. Die Modultests zeigen deren typische Verwendung und Einschränkungen auf.
Anwendungsklasse	2
Lösungsansatz	• Erfassung aller Python-Dateien im Repository • Unittest-Tests identifizieren • Pytest-Tests identifizieren

Umsetzung des Prototyps

Erfassung aller Python-Dateien im Repository: Der Prototyp beginnt mit der systematischen Erfassung sämtlicher Python-Dateien im Repository. Jede Datei wird durchsucht, um die gesuchten Arten von Tests zuidentifizieren.
Unittest-Tests identifizieren (siehe 3.1.4): Der Inhalt jeder Python-Datei wird auf das Vorhandensein der Zeichenfolgen `class` und `unittest.TestCase` überprüft, um Unittest-Tests zu erfassen.
Pytest-Tests identifizieren (siehe 3.1.4:) Zusätzlich wird nach der Zeichenfolge `def test` gesucht, um Pytest-Tests zu identifizieren.

Vorschlag für Adaption

Für die behandelte Empfehlung EDI.3 wird keine weitere spezifische Empfehlung ausgesprochen, da die bestehende Methodik aufgrund der Eigenschaften von Python weitgehend ausreicht.

4.9 EST.2: Systematische Durchführung von Funktionstests

Empfehlung	EST.2: Funktionstests werden systematisch erstellt und ausgeführt.
Anwendungsklasse	2
Lösungsansatz	• Suche nach Jobs in der CI/CD-Konfiguration • Analyse der Script-Anweisungen nach Funktionstests

Umsetzung des Prototyps

Die Umsetzung folgt dem Ansatz der Style-Checker-Überprüfung (siehe 4.7). Dabei werden jedoch andere Lookup-Tabellen verwendet: Eine spezifische Tabelle, die gängige Python-Funktionstest-Tools wie `unittest`, `pytest` und `doctest` beinhaltet, wird genutzt, um die Script-Anweisungen auf das Vorhandensein dieser Testwerkzeuge zu überprüfen.

Vorschlag für Verbesserung

Für die Empfehlung EST.2 kann auf die bereits genannten Verbesserungsvorschläge zur Style-Checker-Überprüfung (siehe 4.7) verwiesen werden, da die zugrunde liegenden Tests und Ansätze ähnlich sind.

4.10 ERM.1: Eindeutiger Release-Name

Empfehlung	ERM.1: Jedes Release besitzt eine eindeutige Release-Nummer. Anhand der Release-Nummer lässt sich der zugrunde liegende Softwarestand im Repository ermitteln.
Anwendungsklasse	1
Lösungsansatz	• Prüfen der Release-Namen

Umsetzung des Prototyps

Um die Eindeutigkeit von Release-Namen in einem GitLab-Projekt zu überprüfen, ruft der Prototyp zunächst alle Releases eines Projekts ab. Dabei werden die Release-Namen gesammelt und in eine Liste aufgenommen. Anschließend vergleicht das Python-Skript die Release-Namen auf Duplikate. Diese Überprüfung stellt sicher, dass jeder Release-Namen innerhalb des Projekts einzigartig ist und keine doppelten Einträge existieren.

Vorschlag für Adaption

Für die behandelte Empfehlung ERM.1 wird keine weitere spezifische Empfehlung ausgesprochen, da in diesem Fall, diese nicht nötig ist.

4.11 ERM.3: Regelmäßige Veröffentlichung von Releases.

Empfehlung	ERM.3: Releases werden in regelmäßigen, kurzen Abständen veröffentlicht
Anwendungsklasse	2
Lösungsansatz	Gebe die Daten aller Releases aus

Umsetzung des Prototyps

Die Interpretation und Beurteilung der Intervalle wird der manuellen Sichtung überlassen, da es keine standardisierten Intervalle für Veröffentlichungen gibt und die Frequenz oft vom Projektkontext abhängt. Der Prototyp verwendet die GitLab-API, um alle Releases eines Projekts abzurufen und deren Veröffentlichungsdaten zu sammeln. Er erstellt eine Liste dieser Daten, die es ermöglicht, die zeitlichen Abstände zwischen den Releases zu analysieren. Das Skript liefert lediglich die gesammelten Daten.

Vorschlag für Adaption

Die Struktur der Releases sollte einer klaren und einheitlichen Versionierung folgen, wie beispielsweise `v1.0.0` oder `v1.1.0`, um die Überprüfung zu vereinfachen. Zusätzlich müssen zeitliche Vorgaben festgelegt werden, um die Intervalle für die Releases zu definieren. Der Prototyp kann dann die Zeitabstände zwischen den Releases analysieren und überprüfen, ob diese Vorgaben eingehalten werden.

4.12 EAA.2 und EAA.3: Überprüfung von Lizenzabhängigkeiten

Empfehlung	EAA.2: Die Abhängigkeiten zum Erstellen der Software sind zumindest mit dem Namen, der Versionsnummer, dem Zweck, den Lizenzbestimmungen und der Bezugsquelle beschrieben.
Anwendungsklasse	1
Empfehlung	EAA.3: Neue Abhängigkeiten werden auf Kompatibilität zur angestrebten Lizenz überprüft.
Anwendungsklasse	2
Lösungsansatz	• Überprüfung der Existenz einer Lizenz-Datei • Überprüfung in der CI/CD-Konfiguration

Umsetzung des Prototyps

- **Überprüfung der Existenz einer Lizenz-Datei:** Der Prototyp durchsucht das Root-Verzeichnis des GitLab-Repositories nach Lizenzinformationen. Dabei verwendet es eine Lookup-Tabelle mit typischen Namen wie `LICENSE.txt` oder `LICENSE`, um sicherzustellen, dass eine entsprechende Lizenzdatei vorhanden ist. Die inhaltliche Überprüfung der Lizenzdatei wird nicht durchgeführt, weil die Relevanz und Korrektheit des Lizenzinhalts stark vom spezifischen Kontext des Projekts abhängt. Unterschiedliche Projekte haben unterschiedliche rechtliche Anforderungen, und ein automatisiertes Skript könnte diese komplexen Zusammenhänge nicht angemessen bewerten.
- **Überprüfung in der CI/CD-Konfiguration (siehe 3.1.8):** Der Prototyp analysiert die CI/CD-Konfigurationsdateien, um das Vorhandensein spezifischer Jobs zu überprüfen. Es durchsucht die Konfigurationsdateien nach Jobs, die Begriffe wie `license_scanning`, `license_management` oder `dependency_scanning` enthalten. Diese Begriffe werden vom Prototyp als Indikatoren für Lizenzprüfungen gewertet.

Vorschlag für Adaption

Die Einführung standardisierter Templates und Namenskonventionen für CI/CD-Konfigurationsdateien könnte die Vielfalt verringern und die automatische Überprüfung erleichtern. Die softwaregestützte Überprüfung der CI/CD-Konfigurationsdateien zur Einhaltung von Lizenzbestimmungen hat jedoch Grenzen. Die Konfigurationsdateien können sehr unterschiedlich strukturiert sein, da sie stark von den spezifischen Anforderungen und Präferenzen des Projekts abhängen. Es gibt keine festen Standards für die Benennung von Jobs und Stages, was die automatische Identifikation bestimmter Aufgaben erschwert. Ab der GitLab-Version 16 kann das Lizenz-Scanning auch auf dem, durch das Dependency-Scanning generierten, CycloneDX-SBOM basieren. Dies ermöglicht es, in den CI/CD-Konfigurationsdateien direkt die Lizenzen aus der generierten SBOM-Datei zu parsen und zu identifizieren.[11]

4.13 EAA.8 und EAA.9: Überprüfung der Nutzung automatisierter Build

Empfehlung	EAA.8: Ein Integrations-Build ist eingerichtet.
Anwendungsklasse	2
Empfehlung	EAA.9:Ein Release-Build ist eingerichtet.
Anwendungsklasse	3
Lösungsansatz	• Überprüfung in der CI/CD-Konfiguration • Überprüfung spezifischer Konfigurationsdateien

Umsetzung des Prototyps

- **Überprüfung in der CI/CD-Konfiguration:** Der Prototyp verwendet eine Lookup-Tabelle, die eine Sammlung gängiger Job-Namen für Integration- und Release-Builds enthält. Das Skript durchsucht die CI/CD-Konfigurationsdateien (siehe 3.1.8) nach diesen vordefinierten Namen, indem es die Job-und Stage-Definitionen extrahiert und mit den Einträgen in der Tabelle abgleicht. Integration-Builds sind darauf ausgerichtet, den Code kontinuierlich zu integrieren und sicherzustellen, dass Änderungen korrekt zusammengeführt und getestet werden. Diese Builds lassen sich in den CI/CD-Konfigurationsdateien durch Job-Namen wie `build`, `integration` oder `ci` identifizieren. Im Gegensatz dazu zielen Release-Builds darauf ab, eine Version der Software zu erstellen, die für die Veröffentlichung vorgesehen ist. Diese Builds können durch spezifische Job-Namen wie `release`, `deploy` oder `publish` erkannt werden.
- **Überprüfung spezifischer Konfigurationsdateien:** Der Prototyp durchsucht das GitLab-Repository gezielt nach Python-spezifischen Konfigurationsdateien. Er prüft das Vorhandensein von Dateien wie `requirements.txt`, `pyproject.toml`, `setup.py` und `setup.cfg`, da diese häufig auf automatisierte Build-Prozesse, wie ein Integrationsbuild hinweisen.

Vorschlag für Adaption

Um die Überprüfung zu verbessern, ob Projekte in GitLab automatische Builds verwenden, können mehrere Maßnahmen ergriffen werden:

1. Standardisierte Namenskonventionen für Job- und Stage-Namen schaffen Klarheit und erleichtern die automatische Erkennung.
2. Die Nutzung von CI/CD-Templates und gemeinsamen Bibliotheken für die Konfigurationsdateien ermöglicht wiederverwendbare Konfigurationen für häufige Aufgaben und stellt eine einheitliche Konvention sicher.

4.14 Mapping auf die Anwendungsklassen

Das Mapping der Projekte auf die Anwendungsklassen (siehe 3.3.1) erfolgt durch codespezifische Kriterien, die auf den Empfehlungen basieren und den jeweiligen Anwendungsklassen zugewiesen sind. Diese Kriterien sind boolesche Variablen, die den Erfüllungsstatus einer Empfehlung durch die Werte `true` oder `false` darstellen. Die Ermittlung der empfohlenen Anwendungsklasse für ein Projekt erfolgt basierend auf den Werten der einzelnen Kriterien. Dabei wird festgelegt, wie viele der erforderlichen Kriterien erfüllt sein müssen, damit die Klasse als erfüllt betrachtet wird. Eine Gewichtung der Anwendungsklassen berücksichtigt deren Hierarchie, sodass eine höhere Klasse nur dann als erfüllt gilt, wenn auch die darunter liegenden Klassen die erforderlichen Kriterien erfüllen.

- **Anwendungsklasse 1**: Zu den Kriterien, die die grundlegenden Anforderungen an ein Projekt erfüllen, gehören die eindeutige Angabe einer Versionsnummer (_RELEASE_NUMBER_UNIQUE), das Vorhandensein eines Lizenzmanagements (_LICENSE_MANAGEMENT) sowie eine vollständige README-Datei (_README), die sowohl durch einen Namen (_README_NAME) als auch durch bestimmte Begriffe im Inhalt (_README_FOUND_TERMS) gekennzeichnet ist. Darüber hinaus wird geprüft, ob eine automatisierte Code-Analyse (_AUTOMATED_CODE_ANALYSIS) vorhanden ist.
- **Anwendungsklasse 2**: Kriterien, die weitergehende Anforderungen an Projekte abdecken, umfassen die Verfügbarkeit einer Projekt-Roadmap (_ROADMAP), die Einhaltung bestimmter Branch-Standards (_BRANCH), wie (_MAIN_BRANCH_CORRECT, _CURRENT_MAIN_BRANCH, _OTHER_BRANCHES) und die Implementierung automatisierter Funktionstests (_AUTOMATED_FUNCTION_TEST). Zusätzlich werden die Daten der verschiedenen Releases durch das Release-Datum (_RELEASE_DATE) angezeigt. Es wird außerdem nach einem Changelog (_CHANGELOG) und einem Glosar (_GLOSSARY) gesucht. Beim Testen wird zusätzlich nach die Existenz von Moduletests (_MODULE_TEST) Überprüft. Für das Vorhandensein von Anforderungen (_REQUIREMENTS) werden zwei Aspekte überprüft: die Existenz von Issues (_REQUIREMENTS_ONLY_ISSUE) und sowie die Existenz von Epics (_REQUIREMENTS_ONLY_EPIC). Für die automatisierten Builds (_AUTOMATED_BUILD_CONFIGURATION) wird nach einem Integration-Build (_INTEGRATION_BUILDS) gesucht und und das Vorhandensein einer Python-Build-Konfiguration (_PYTHON_BUILD_FILES) überprüft.
- **Anwendungsklasse 3**: Kriterien, die fortgeschrittene Anforderungen an Projekte adressieren, wie die Durchführung von automatisierten Release-Builds (_RELEASE_BUILDS) innerhalb der automatisierten Build-Umgebung (_AUTOMATED_BUILD_CONFIGURATION).

Ergebnisse

Im Rahmen der Ergebnisanalyse (siehe 3.3.2) wurden zunächst ausgewählte Projekte individuell untersucht, um die Zuverlässigkeit und Funktionalität des Prototyps schon während der Entwicklung zu testen. Diese initiale Überprüfung ermöglichte es, potenzielle Schwächen oder Verbesserungsmöglichkeiten im Prototyp zu identifizieren (siehe 6.2). Die detaillierte Bewertung der Projekte basierend auf dem finalen Prototypen wird in Abschnitt 5.1 erläutert. Anschließend folgte eine umfassende Auswertung aller verfügbaren Projekte, die den allgemeinen Entwicklungsstand erfasst. Diese zweistufige Vorgehensweise dient dazu, die Genauigkeit der automatischen Klassifizierung durch den Prototyp zu bewerten und gleichzeitig allgemeine Muster sowie statistische Zusammenhänge in der Erfüllung der Empfehlungen zu erkennen (siehe 5.2). Durch diesen Ansatz wird sichergestellt, dass sowohl die Effizienz des Prototyps als auch die Klassifizierungsgenauigkeit der gesamten Projektlandschaft evaluiert werden.

5.1 Ersteinschätzung: Prototyp-Analyse ausgewählter Projekte

Im Folgenden werden die Ergebnisse der Analyse zuvor ausgewählter Projekte vorgestellt (siehe 3.3.2). Dabei wird geprüft, inwieweit die automatische Klassifizierung des Prototyps mit den tatsächlichen Eigenschaften der Projekte übereinstimmt. Die ausgewählten Projekte geben einen ersten Eindruck von der Funktionalität des Prototyps und der Zuordnung zu den Anwendungsklassen.
Da kein Kontakt zu den Entwicklern dieser Projekte bestand, ist die Analyse auf die Ergebnisse des Prototyps und eigene Überprüfungen beschränkt. Ein Gespräch oder ein detaillierter Einblick in den Entwicklungsprozess war jedoch nicht vorgesehen, könnte aber eine umfassendere Analyse ermöglichen.
Eine vollständige Übersicht darüber, welche der überprüften Kriterien für die einzelnen Projekte jeweils erfüllt sind, ist in Tabelle 2 aufgeführt.

Kleineres, aktives Projekt A: Das Projekt wird mit 100 % in Anwendungsklasse 2 eingeordnet und erfüllt Empfehlungen wie eine GitLab-konforme Readme im Markdown-Format, Anforderungserfassung, Branch-Nutzung, automatisierte Buildumgebung sowie Funktions- und Modultests. Da jedoch automatisierte Release-Builds fehlen, ist es wahrscheinlich nicht für kritische Anwendungen konzipiert und erfüllt daher die Anforderungen einer höheren Anwendungsklasse nicht. Diese Beobachtung bestätigt die Annahme, dass das Projekt voraussichtlich nicht auf die höchste Klassifizierung ausgelegt ist.

Umfangreiches, aktives Projekt B: Das Projekt wird sicher in die höchste Anwendungsklasse 3 eingeordnet und erfüllt 75 % der überprüften Empfehlungen. Es verfügt über eine Readme, Changelog, definierte Anforderungen, automatisierte Codeanalysen sowie Tests. Besonders hervorzuheben ist, dass das Projekt sowohl automatisierte Release-Builds als auch Integration Builds bereitstellt, was auf eine strukturierte und gut organisierte Entwicklungsumgebung hinweist. Allerdings fehlen noch eine Roadmap und das Lizenzmanagement. Die hohen Erfüllungsgrade in Anwendungsklasse 3 bestätigen, dass das

Projekt die Empfehlungen in seine Prozesse integriert und eine hohe Klassifizierung anstrebt.

Besonderheit: Als Ergebnis der automatisierten Überprüfung wurde zwar eine Readme-Datei gefunden, jedoch nicht als vollständig erfüllt gewertet, da sie nicht im Markdown-Format vorlag. Der Prototyp ist darauf ausgelegt, Readme-Dateien im Markdown-Format zu analysieren, um die automatisierte Überprüfung in GitLab zu unterstützen. Dies könnte sowohl als Limitation des Prototyps als auch als fehlende klare Empfehlung zur Nutzung des standardisierten Formats interpretiert werden (siehe 4.2).

Älteres Projekt C: Das Projekt basiert auf älterem Quellcode und erfüllt die Erwartungen an die Empfehlungen nur geringfügig. Es konnte keiner Anwendungsklasse sicher zugeordnet werden. Der Erfüllungsgrad liegt bei 33,33 % für Anwendungsklasse 1 und 18,52 % für Anwendungsklasse 2, während Klasse 3 nicht erreicht wurde. Eine Readme-Datei fehlt, und die Projektstruktur zeigt nur den Hauptbranch, ohne weitere Branches. Modultests sind vorhanden, jedoch fehlen automatisierte Funktionstests und Codeanalysen. Diese Ergebnisse bestätigen die Erwartungen, dass das Projekt aufgrund seines Alters und der Inaktivität nur wenige Empfehlungen erfüllt.

Kriterium	Projekt A	Projekt B	Projekt C
Release number unique	✓	✓	×
Release date	✓	✓	×
Roadmap	×	×	×
License management	×	×	✓
README check	✓	×	×
README check_name	✓	✓	×
README check_terms	✓	×	×
Auto build config	×	✓	×
Auto build config_integration	✓	✓	×
Auto build config_release	×	✓	×
Auto build config_python	✓	✓	✓
Requirements (Has issues & epics)	✓	✓	×
Branches (Main correct, current, other)	✓	✓	✓
Changelog	×	✓	✓
Glossary	×	×	×
Auto function test	✓	✓	×
Auto code analysis	✓	✓	×
Module test	✓	✓	✓

Tabelle 2: Übersicht der Erfüllungsgrade für die Projekte A, B und C anhand der empfohlenen Kriterien in den Anwendungsklassen. Die Darstellung zeigt, welche Anforderungen erfüllt (✓), und welche nicht erfüllt (×) sind. Projekt A entspricht vollständig der Anwendungsklasse 2, Projekt B erreicht größtenteils die Kriterien der Klasse 3, und Projekt C zeigt nur geringe Erfüllung für Klasse 1 und 2.

5.2 Detaillierte Analyse der Zuordnung aller Projekten zu Anwendungsklassen

In dieser Arbeit wurden insgesamt 154 Python-Projekte in GitLab mithilfe des Prototyps anhand spezifischer Kriterien in drei Anwendungsklassen eingeordnet. Die Klassifikation basiert auf dem Mapping (siehe 3.3.1) dieser Kriterien zu den Anwendungsklassen. Projekte, die in keiner Anwendungsklasse mehr als 0 % Erfüllung aufwiesen, wurden von der Analyse ausgeschlossen, da sie überwiegend leere Projekte, neu angelegte Projekte oder Projekte ohne Code waren. Nach diesem Ausschluss verblieben 130 Projekte für die Auswertung.
Die Ergebnisse dieser Einordnung werden im Folgenden detailliert beschrieben. Die Abbildung 2 bietet eine grafische Darstellung, die die Verteilung der 130 analysierten Projekte auf die drei Anwendungsklassen veranschaulicht.

Anwendungsklasse 1

Nach der Einsortierung durch den Prototypen umfasst die erste Anwendungsklasse 102 Projekte (Abbildung 2). Diese Projekte erfüllen grundlegende Kriterien wie eine eindeutige Versionsnummer, Lizenzmanagement und eine vollständige README-Datei. Der durchschnittliche Erfüllungsgrad liegt bei 96,49 %, mit einer Spannbreite von 1 % bis 100 %.
Die Analyse zeigt jedoch, dass nur ein kleiner Teil dieser Projekte Funktionen wie Changelogs (3,92 %), automatisierte Build-Konfigurationen (5,88 %) oder automatisierte Funktionstests (7,84 %) implementiert hat.

Anwendungsklasse 2

Nach der Einsortierung umfasst die zweite Anwendungsklasse 14 Projekte (Abbildung 2). Diese Projekte erfüllen weitergehende Anforderungen wie die Verfügbarkeit einer Roadmap, die Einhaltung von Branch-Standards und die Implementierung von Funktionstests. Der durchschnittliche Erfüllungsgrad liegt bei 99,50 %, mit einer Spannbreite von 15 % bis 100 %.
Projekte dieser Klasse neigen eher zur Implementierung fortgeschrittener Funktionen. So verfügen 28,57 % der Projekte über einen Changelog, 35,71 % über eine automatisierte Build-Konfiguration und 42,86 % über automatisierte Funktionstests.

Anwendungsklasse 3

Nach der Einsortierung umfasst die dritte Anwendungsklasse 5 Projekte (Abbildung 2). Diese Projekte erfüllen fortgeschrittene Kriterien wie die Durchführung von Release-Builds und das Vorhandensein einer Python-Build-Konfiguration.
Der durchschnittliche Erfüllungsgrad liegt bei 98,10 %, mit einer Spannbreite von 15 % bis 100 %. In dieser Klasse ist der Anteil der Projekte mit Funktionen höher. So verfügen 20 % der Projekte über einen Changelog, 60 % über eine automatisierte Build-Konfiguration und 50 % über automatisierte Funktionstests.

Grafische Auswertung

Zur Auswertung des Prototyps wurden verschiedene Diagramme erstellt, die der Visualisierung der Ergebnisse dienen und eine fundierte Analyse ermöglichen sollen. Diese Diagramme bieten die Möglichkeit, sowohl Vergleiche zwischen den Projekten zu ziehen als auch den Erfolg der einzelnen Empfehlungen zu bewerten.

Im Anhang sind ergänzend die in Abbildung 6 dargestellten Kreisdiagramme zu finden, die eine umfassende Analyse der Verteilung der spezifischen Empfehlungen präsentieren. Es ist hervorzuheben, dass diese Kreisdiagramme Teil des automatischen Outputs des Prototyps sind, während die anderen Diagramme nicht automatisch generiert werden. Die Auswertung der Kreisdiagramme verdeutlicht sofort, dass viele Empfehlungen einen hohen Prozentsatz an Nichterfüllung aufweisen, insbesondere diejenigen, die für die Erfüllung der höheren Anwendungsklassen entscheidend sind. Besonders auffällig ist, dass sowohl das Glossar als auch die Roadmap in keinem der Projekte erfüllt sind. Im Gegensatz dazu zeigen die Diagramme, dass die Empfehlungen zu den Branch-Namen und die README-Dateien relativ gut beachtet werden.

Das Balkendiagramm in Abbildung 2 veranschaulicht die Zuordnung von insgesamt 130 analysierten Python-Projekten zu den Anwendungsklassen 1, 2 und 3. Es zeigt die Anzahl der Projekte in jeder Klasse und bietet somit eine anschauliche Darstellung der Verteilung. Die Visualisierung ermöglicht es, schnell zu erkennen, wie viele Projekte den verschiedenen Anwendungsklassen zugeordnet wurden und spiegelt die Ergebnisse der detaillierten Analyse wider, die im Kapitel 5.2 beschrieben wurde.

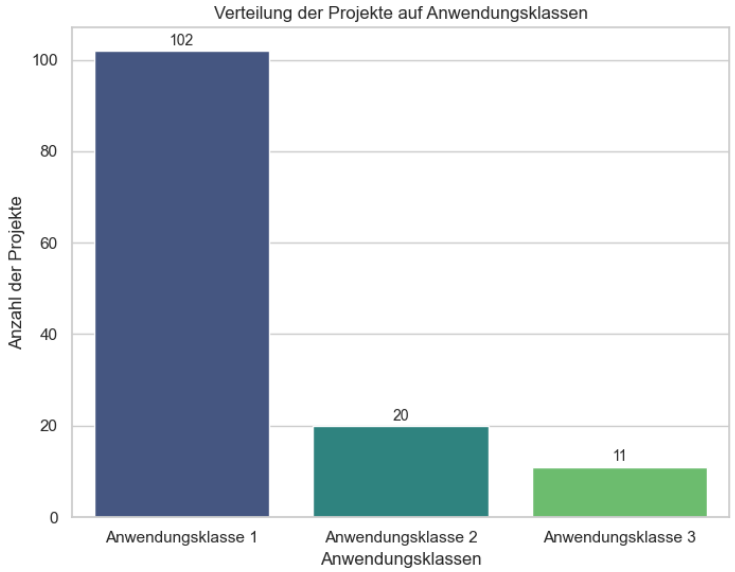


Abbildung 2: Balkendiagramm zur Darstellung der Zuordnung von 130 analysierten Python-Projekten in ihrer Gesamtheit zu den Anwendungsklassen 1, 2 und 3. Das Diagramm verdeutlicht die Anzahl der Projekte in jeder Klasse und bietet einen schnellen Überblick über deren Verteilung.

Die Abbildung 3 zeigt die Korrelationen zwischen den verschiedenen Kriterien in den Projekten. Auffällige starke bzw. positive Korrelationen, angezeigt durch dunklere Rottöne), deuten auf starke Zusammenhänge zwischen bestimmten Kriterien hin, während schwache bzw. negative Korrelationen, mit Blautönen dargestellt, auf invers korrelierte Kriterien hinweisen.

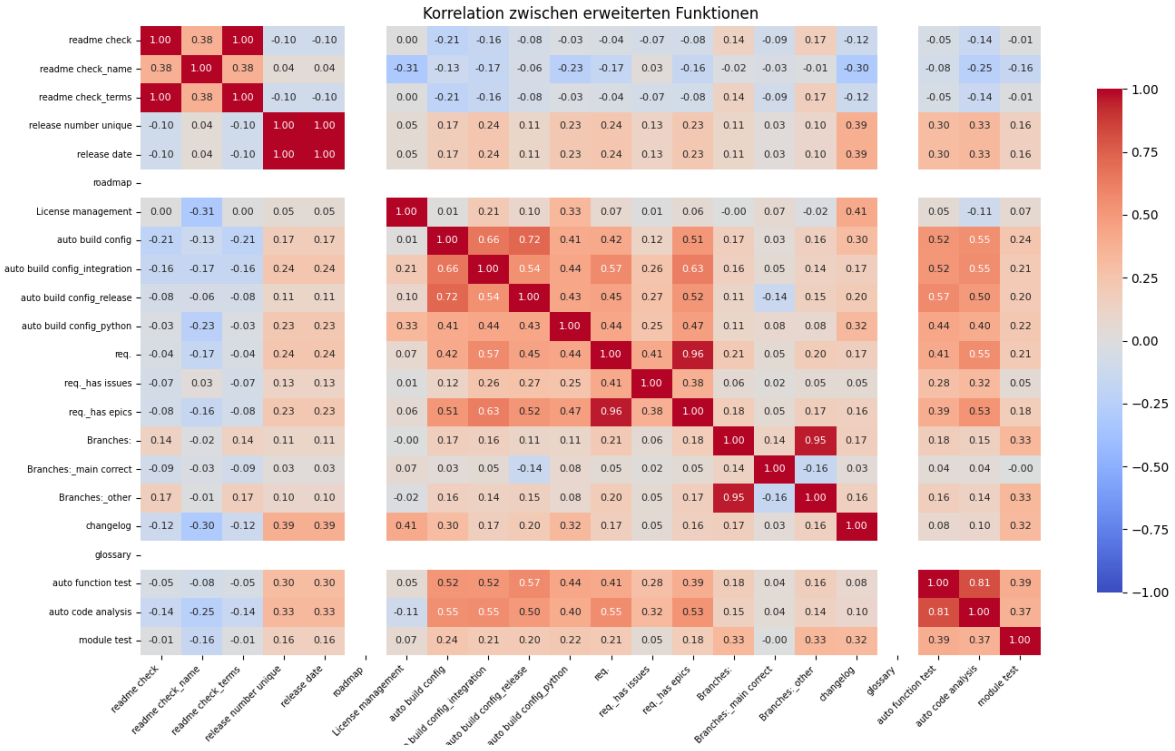


Abbildung 3: Heatmap zur Korrelation. Diese Darstellung zeigt die Korrelationen zwischen den Kriterien der analysierten Projekte. Dunklere Rottöne repräsentieren starke positive Korrelationen, während hellere Rottöne schwächere positive Zusammenhänge anzeigen. Blautöne deuten auf schwache oder negative Korrelationen hin..

Eine starke Korrelation zwischen bestimmten Kriterien könnte darauf hindeuten, dass Projekte, die in einem Bereich gut aufgestellt sind, tendenziell auch in anderen Aspekten hohe Standards erfüllen, um eine entsprechende Anwendungsklasse zu erreichen.
So zeigt sich, dass eine hohe Korrelation (*0.56*) zwischen einem eingerichteten automatischen Integrationsbuild (Anwendungsklasse 1) und automatischen Releasebuilds (Anwendungsklasse 2) besteht (siehe 4.13). Projekte, die ein automatisches Integrationsbuild nutzen, haben tendenziell auch ein automatisches Releasebuild eingerichtet. Obwohl dies nicht das einzige Kriterium ist, deutet die Korrelation darauf hin, dass Fortschritte in einer Anforderung häufig mit Fortschritten in einer anderen Anforderung einhergehen.
Auch das Vorhandensein eines Changelogs (siehe 4.3) und die Vergabe eindeutiger Releasenummern (siehe 4.10) zeigen eine hohe Korrelation (*0.39*). Dies legt nahe, dass Projekte, die eindeutige Releasenummern vergeben, oft auch ein Changelog implementiert haben. Ein solches strukturiertes Vorgehen bei der Versionierung von Software ist häufig mit einer systematischen Dokumentation von Änderungen und Neuerungen verbunden.

Eine niedrige Korrelation zwischen einigen Kriterien kann auf Verbesserungspotential hinwei-

sen. Diese Erkenntnisse lassen sich nutzen, um Empfehlungen zur Optimierung der Projekte zu formulieren und deren Anwendungsklasse zu erhöhen.
So zeigt sich, dass die Existenz einer Readme (siehe 4.2) mit Anwendungsklasse 1 häufig in sehr niedriger Korrelation zu vielen anderen Kriterien steht. Projekte, die sich ausschließlich auf eine Readme konzentrieren, zeigen oft Schwächen in Kriterien wie automatisierten Tests (siehe 4.9), die eine niedrige Korrelation von *0,04* aufweisen, oder auch in automatisierten Build-Prozessen (siehe 4.13), die eine negative Korrelation von *-0,11* zeigen. Diese Werte deutet darauf hin, dass solche Projekte wahrscheinlich für den internen Gebrauch entwickelt wurden oder auch hohes Verbesserungspotential besitzen. Es liegt allerdings auch nahe, dass viele Projekte, auch sowohl mit niedriger als auch höherer Anwendungsklasse, eine Readme-Datei besitzen, da das Hinzufügen einer Readme in der GitLab-Projektverwaltung ein unkomplizierter Schritt ist. Zudem wird sie oft als schnelles Mittel im intensiven Prototyping verwendet, um grundlegende Informationen festzuhalten.

Die **fehlenden Korrelationswerte** für eine Roadmap und ein Glossar sind besonders auffällig. Da in keinem Projekt eine Roadmap oder ein Glossar vorhanden ist und diese Einträge durchgehend den Erfüllungsstatus `false` (siehe 4.14) aufweisen, kann aufgrund der fehlenden Varianz keine Korrelation berechnet werden. Ein möglicher Grund für das Fehlen einer Roadmap (siehe 4.6) könnte die geringe Vertrautheit mit diesem GitLab-Feature sein. Insbesondere bei kleineren oder inaktiven Projekten, wo Zeitmangel und der Fokus auf die Implementierung eine Rolle spielen. Ein Glossar (siehe 4.1) wird häufig nicht priorisiert, da relevante Begriffe oft in anderen Dokumenten als separate Kapitel integriert sind.

Eine weitere wichtige Dimension ist der Erfüllungsgrad der Projekte, der nach Anwendungsklassen differenziert betrachtet wird. Die Abbildung 4 verdeutlicht die signifikanten Unterschiede der Erfüllungsgrade in den verschiedenen Anwendungsklassen und liefert Hinweise zur Konsistenz und Qualität der Umsetzung der Empfehlungen innerhalb dieser Klassen.

Die Verteilung der Erfüllungsgrade in Abbildung 4 verdeutlicht, dass die Projekte in der Anwendungsklasse 1 eine größere Streuung aufweisen. Diese Streuung deutet darauf hin, dass Projekte dieser Klasse unterschiedlich stark an die Empfehlungen angepasst sind, was auf eine heterogene Qualität der Umsetzung hindeutet. Im Gegensatz dazu sind die Erfüllungsgrade der Projekte in den Anwendungsklasse 2 und 3 um den Medianwert enger gruppiert, was darauf schließen lässt, dass Projekte in diesen höheren Klassen konsistenter die empfohlenen Standards erfüllen. Insbesondere die höchste Anwendungsklasse 3 zeigt eine geringere Variabilität und eine tendenziell stärkere Erfüllung der Empfehlungen, was auf eine gleichmäßigere Umsetzung der Qualitätskriterien hindeutet und somit auf eine insgesamt fortgeschrittenere und zuverlässigere Projektdokumentation und -struktur schließen lässt.

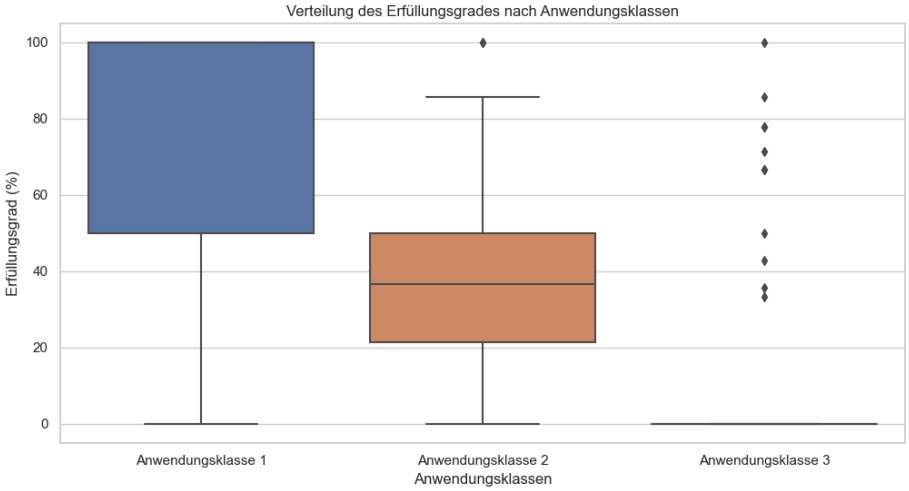


Abbildung 4: Boxplot der Erfüllungsgrade für die Anwendungsklassen 1, 2 und 3. Die Verteilung zeigt, dass Projekte der Klasse 1 eine breite Streuung und damit größere Variabilität in der Erfüllung der Kriterien aufweisen. Dagegen sind die Erfüllungsgrade in den Klassen 2 und 3 enger um den Median gruppiert, was auf eine konsistentere Umsetzung der Empfehlungen in diesen Klassen hindeutet.

Zusammenfassung der Projekteinordnung und ihre Einschränkungen

Im Rahmen dieser Überprüfung wurde ein zweistufiger Ansatz angewendet, um die Zuverlässigkeit und Funktionalität eines Prototyps zur automatischen Klassifizierung von Python-Projekten zu testen sowie den allgemeinen Entwicklungsstand der Projekte zu bewerten.

Zunächst erfolgte eine detaillierte Analyse ausgewählter Projekte, um potenzielle Schwächen und Verbesserungspotenziale zu identifizieren. Diese Überprüfung konzentrierte sich auf Aspekte, die die Effizienz und Klassifizierungsgenauigkeit des Prototyps betreffen.

Im zweiten Teil wurden alle analysierten Projekte in die drei Anwendungsklassen eingeordnet. Diese Einordnung zeigt eine deutliche Konzentration in der ersten Klasse, was auf eine hohe Erfüllung der grundlegenden Kriterien für die Klassifizierung hinweist. Die Analyse zusätzlicher Merkmale wie Changelogs, automatisierte Build-Konfigurationen und Funktionstests lieferte weitere Einblicke in die Entwicklungsqualität der einzelnen Projekte. In den analysierten Projekten wurde festgestellt, dass weder ein Glossar noch eine Roadmap vorhanden ist, was die Qualität der Projektdokumentation beeinträchtigen kann.

Aktive Projekte zeigen eine signifikant höhere Übereinstimmung mit den Empfehlungen des Prototyps, was darauf hinweist, dass Entwickler bei der Implementierung neuer Projekte verstärkt auf diese achten. Gleichzeitig ist die Anzahl der Projekte in den Anwendungsklassen 2 und 3 gering. Die Korrelationen zwischen verschiedenen Kriterien in den Projekten könnten ebenfalls näher untersucht werden. Starke Korrelationen könnten auf Synergien hinweisen, während schwache Korrelationen Verbesserungspotenziale aufzeigen. Es ist zu beachten, dass die begrenzte Anzahl analysierter Projekte die Generalisierbarkeit der Ergebnisse beeinträchtigen kann. Da nicht alle Empfehlungen automatisiert überprüft werden, können auch Inkonsistenzen in der Klassifizierung auftreten.

Insgesamt liefern die Ergebnisse Hinweise auf den Entwicklungsstand und die Reife der analysierten Python-Projekte und verdeutlichen die Erfüllung der festgelegten Empfehlungen.

Diskussion und Ausblick

Der entwickelte Prototyp zur Klassifizierung von Projekten auf Grundlage definierter Kriterien zeigt vielversprechende Ansätze, insbesondere bei der Einordnung in grundlegende Anwendungsklassen. Die Analyse der Projekte offenbarte jedoch auch Herausforderungen vor allem bei der korrekten Zuordnung fortgeschrittener Funktionen und der Handhabung älterer Projekte. Der Prototyp liefert bereits nützliche Erkenntnisse, erfordert jedoch weitere Verfeinerungen und eine präzisere Formulierung der Bewertungskriterien, um eine breitere Anwendbarkeit und höhere Automatisierung zu ermöglichen. Zukünftige Entwicklungen sollten sich auf die Verbesserung der Skalierbarkeit, Benutzerfreundlichkeit und inhaltlichen Analyse der Dokumente konzentrieren, um die Qualität der Klassifikation zu optimieren.

6.1 Nutzen und Potenzial des Prototyps

Der entwickelte Prototyp bildet die Basis für weiterführende Forschungen und Optimierungen. Er zeigt sowohl Herausforderungen als auch Potenziale auf, die im Folgenden detailliert betrachtet werden:

Qualität und Relevanz des Datensatzes:

- Der vorhandene Datensatz umfasst Projekte, die möglicherweise nur teilweise oder gar keine der festgelegten Anforderungen erfüllen. Eine gezielte Auswahl oder Erweiterung des Datensatzes könnte die Relevanz und Aussagekraft der Analyse verbessern und genauere Ergebnisse liefern.
- Es ist unklar, ob die Entwickler der überprüften Projekte die relevanten Richtlinien kannten. Diese Unklarheit könnte die Ergebnisse beeinflussen, da die Bewertung möglicherweise nicht die tatsächliche Einhaltung der Richtlinien widerspiegelt. Daher muss diese Unsicherheit bei der Analyse berücksichtigt werden, um Verzerrungen zu vermeiden.

Bewertungskriterien und Automatisierungsunterstützung:

- Bei der objektiven Bewertung von Dokumenten wie README-Dateien, Contributing-Dateien oder Glossaren fehlen oft spezifischere Kriterien. Klare Bewertungsmaßstäbe würden die Überprüfbarkeit erhöhen und die Nützlichkeit des Prototyps verbessern.
- Zur Reduzierung der Abhängigkeit von zusätzlicher manueller Überprüfung könnten detailliertere Empfehlungen, inklusive spezifischer Beispiele und klarer Richtlinien, implementiert werden, um die Automatisierung zu unterstützen.

Ergebnisse der Klassifikation:

- Die Projekteinordnung zeigt eine deutliche Konzentration in der ersten Anwendungsklasse. Dies liegt vermutlich daran, dass überwiegend kleinere, öffentlich zugängliche Projekte analysiert wurden, die nur einige Empfehlungen erfüllen müssen. Somit war die Erfüllung weiterführender Empfehlungen für die höheren Anwendungsklassen weniger häufig.

- Die Abwesenheit von Glossaren und Roadmaps in den analysierten Projekten legt nahe, dass diese Dokumentationselemente in der Praxis weniger verbreitet sind oder vom Prototyp nicht zuverlässig erkannt wurden. Um festzustellen, ob diese Dokumentationen allgemein fehlen oder ob ihre Erkennung verbessert werden muss, könnte eine genauere Analyse sinnvoll sein.
- Es zeigte sich, dass aktive neuere Projekte die Empfehlungen häufiger erfüllen, was darauf hinweist, dass solche Entwicklungen tendenziell besser auf die aktuellen Standards und Best Practices abgestimmt sind. Eine Fokussierung auf solche Projekte könnte die Klassifikation und Bewertung weiter optimieren.
- Die Analyse der Korrelationen zeigt, dass starke Zusammenhänge zwischen bestimmten Kriterien auf eine integrierte Herangehensweise bei der Softwareentwicklung hinweisen. Projekte, die in einem Bereich gut abschneiden, neigen dazu, auch in anderen Bereichen hohe Standards zu erfüllen. Dies spricht für Synergien im Entwicklungsprozess, die gezielt gefördert werden sollten.
- Die automatische Klassifizierung der Projekte könnte in einigen Fällen ungenau sein. Die Komplexität von Softwareprojekten und die Vielfalt der Anforderungen machen eine manuelle Validierung notwendig, um sicherzustellen, dass Projekte den jeweiligen Anwendungsklassen korrekt zugeordnet werden. Eine solche manuelle Überprüfung, zusammen mit präziseren Vorgaben, kann die Klassifikationsgenauigkeit verbessern und Fehlklassifikationen vermeiden.

Optimierung des Prototyps:

- Der Prototyp zeigt vielversprechende Ansätze, erfordert jedoch zusätzliche Verfeinerungen, um eine genauere Klassifikation von Projekten zu ermöglichen. Momentan bietet er nur allgemeine Tendenzen und keine präzise Zuordnung zu spezifischen Anwendungsklassen.
- Der aktuelle Prototyp hat noch Potenzial zur Weiterentwicklung, insbesondere hinsichtlich der Benutzerfreundlichkeit und des Interface-Designs, um eine intuitivere und effizientere Nutzung zu ermöglichen.
- Der Prototyp stößt bei der Verarbeitung großer Datenmengen und bei der Handhabung zahlreicher Projekte, insbesondere beim Zugriff auf Repositories, auf erste Grenzen. Dies unterstreicht die Notwendigkeit, die Skalierbarkeit zu verbessern, um eine breitere Anwendbarkeit zu gewährleisten.

Abschließend lässt sich festhalten, dass der entwickelte Prototyp eine Basis für die automatisierte Klassifizierung von Projekten gemäß den DLR-Richtlinien schafft. In zukünftigen Entwicklungsschritten sollten gezielte Optimierungen und Erweiterungen in den Bereichen Skalierbarkeit, Benutzerfreundlichkeit und präziser Kriterienformulierung vorgenommen werden.

6.2 Zukünftige Empfehlungen und Entwicklungen

Um den Prototypen weiter zu verbessern und eine breitere Anwendbarkeit zu ermöglichen, sollten folgende Aspekte berücksichtigt werden:

Optimierung der DLR-Guidelines:

- Die DLR-Guidelines müssen detaillierter formuliert werden, um eine automatisierte Überprüfung zu ermöglichen. Dazu zählen klare Vorgaben für Benennungen, wie beispielsweise einheitliche Dateinamenskonventionen, präzise zeitliche Anforderungen, sowie eine eindeutige Festlegung der zulässigen Tools, die für bestimmte Aufgaben verwendet werden dürfen.
- Es muss untersucht werden, inwieweit die detaillierte Kenntnis der Entwickler über die DLR-Guidelines zur Verbesserung der Projektqualität beiträgt.
- Konkrete Beispiele in den Empfehlungen könnten den Entwicklern dabei helfen, die Anforderungen und Vorgaben der DLR-Guidelines klarer nachzuvollziehen und sie gezielter in ihren Projekten umzusetzen.

Weiterentwicklung des Prototyps:

- Eine effektive Methode zur inhaltlichen Analyse von Dokumenten muss noch speziell für den Prototyp entwickelt werden, um dessen Fähigkeit zur Überprüfung der Übereinstimmung mit den Anforderungen zu gewährleisten. Ein erster konkreter Schritt könnte darin bestehen, im Prototyp ein regelbasiertes System zu implementieren, das gezielt Schlüsselwörter und Phrasen erkennt, um grundlegende Übereinstimmungen und Abweichungen zu identifizieren. Darüber hinaus könnte der Prototyp durch den Einsatz von Natural Language Processing (NLP)-Technologien erweitert werden, die für die spezifischen Dokumenttypen trainiert werden müssten, um eine automatische inhaltliche Analyse zu ermöglichen.
- Der Prototyp könnte durch die Implementierung von Feedback-Schleifen kontinuierliche Verbesserungen im Entwicklungsprozess unterstützen. Dies ließe sich durch regelmäßige, automatisierte Reviews realisieren, bei denen der Prototyp den Entwicklungsfortschritt analysiert und basierend auf definierten Kriterien gezielte Rückmeldungen liefert.

Zusammenfassend ist festzustellen, dass die Erweiterung des Prototyps durch gezielte Anpassungen und die Implementierung neuer Technologien das Potenzial hat, die Einhaltung der DLR-Guidelines präziser zu gestalten. Die Einführung konkreter Standards, die Nutzung von Natural Language Processing zur inhaltlichen Analyse sowie regelmäßige Feedback-Schleifen könnten die Anwendbarkeit des Prototyps verbessern und den Entwicklungsprozess insgesamt stärken.

Literatur

[1] Manfred Broy und Marco Kuhrmann. Einführung in die Softwaretechnik. Springer Vieweg, 2021. ISBN: 978-3-662-50262-4.

[2] DLR. Software-Engineering-Empfehlungen des DLR. DLR, 2018.

[3] DLR. The Software Engineering Community at DLR. 2017. URL: https://elib.dlr.de/114050/2/20170822_track1the_software_engineering_community_at_dlr.pdf (besucht am 01. 09. 2024).

[4] GitLab Inc. Use CI/CD to build your application. n.d. URL: https://docs.gitlab.com/ee/ci/quick_start/ (besucht am 01. 09. 2024).

[5] Sandra Gittlen. The ultimate guide to SBOMs. 2024. URL: https://about.gitlab.com/blog/2022/10/25/the-ultimate-guide-to-sboms/ (besucht am 01. 09. 2024).

[6] GitLab Inc. Changelog entries. n.d. URL: https://docs.gitlab.com/ee/development/changelog.html (besucht am 01. 09. 2024).

[7] GitLab Inc. Dependency list. n.d. URL: https://docs.gitlab.com/ee/user/application_security/dependency_list/ (besucht am 01. 09. 2024).

[8] GitLab Inc. GitLab Flavored Markdown (GLFM). n.d. URL: https://docs.gitlab.com/ee/user/markdown.html (besucht am 01. 09. 2024).

[9] GitLab Inc. Introduction to GitLab CI/CD. n.d. URL: https://docs.gitlab.com/ee/ci/introduction/index.html (besucht am 01. 09. 2024).

[10] GitLab Inc. Issues. n.d. URL: https://docs.gitlab.com/ee/user/project/issues/ (besucht am 01. 09. 2024).

[11] GitLab Inc. Python GitLab. n.d. URL: https://docs.gitlab.com/ee/user/compliance/license_scanning_of_cyclonedx_files/ (besucht am 01. 09. 2024).

[12] GitLab Inc. REST API. n.d. URL: https://docs.gitlab.com/ee/api/rest/ (besucht am 01. 09. 2024).

[13] GitLab Inc. Roadmap. n.d. URL: https://docs.gitlab.com/ee/user/group/roadmap/ (besucht am 01. 09. 2024).

[14] Tobias Schlauch. Software Engineering Initiative of DLR. 2017. URL: https://elib.dlr.de/117717/1/Software%20Engineering%20Initiative%20of%20DLR-ESA-PA-Workshop-2017.pdf (besucht am 01. 09. 2024).

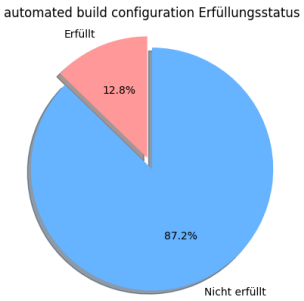
[15] Tobias Schlauch und Carina Haupt. Helping a friend out - Guidelines for better software. 2017. URL: https://elib.dlr.de/114049/2/Helping%20a%20friend%20out%20%E2%80%93%20Guidelines%20for%20better%20software_4_3.pdf (besucht am 01. 09. 2024).

[16] Ian Sommerville. Software Engineering. 9. Aufl. Addison-Wesley, 2010. ISBN: 978-0-13-703515-1.

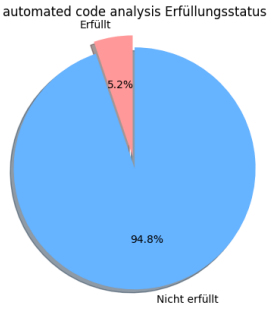
[17] Karl E. Wiegers und Joy Beatty. Software Requirements. Microsoft Press, 2013. ISBN: 978-0735679665.

[18] Olvier Zablocki. Der Prototyp. Zugriff am 12. Oktober 2024. 2024. URL: https://gitup.uni-potsdam.de/zablocki1/BA_geteilt.

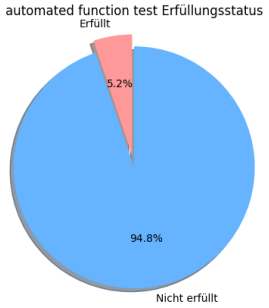
Anhang



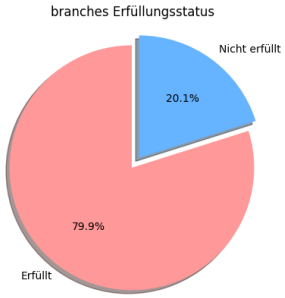
EAA.8 und EAA.9: Überprüfung der Nutzung automatisierter Build



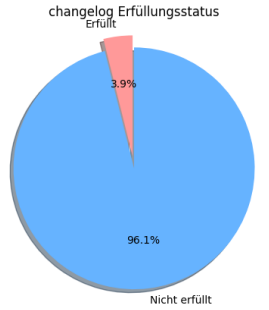
EDI.1 und EDI.7: Einhaltung der Programmiersprachregeln und automatisierte Prüfung



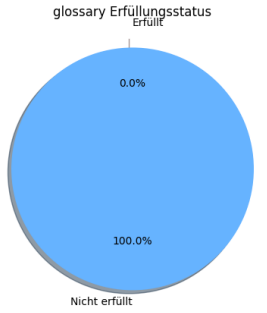
EST.2: Systematische Durchführung von Funktionstests



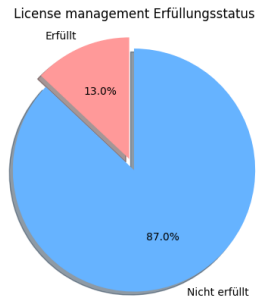
EÄM.9: Prüfung der Branch-Namen



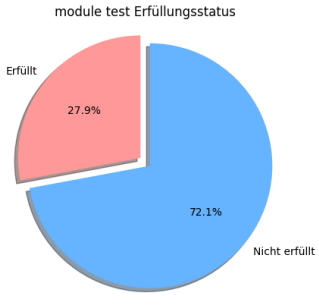
EÄM.6: Führen eines Changelogs



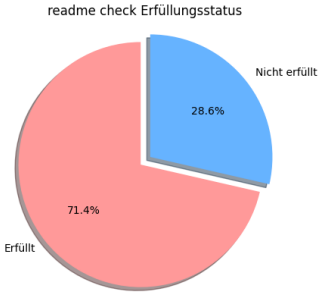
EAM.8: Erstellung eines Glossars



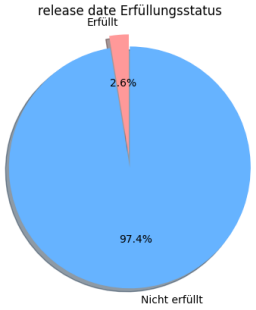
EAA.2 und EAA.3: Überprüfung von Lizenzabhängigkeiten



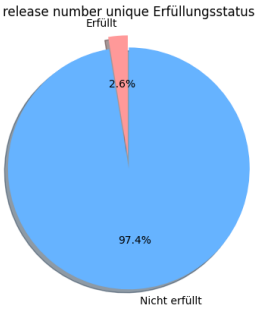
EDI.3: Existenz von Modultests



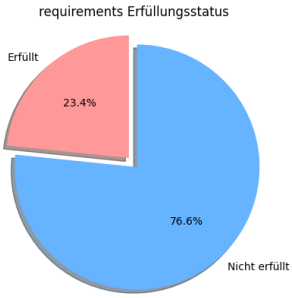
Existenz einer Readme- oder Contributing-Datei



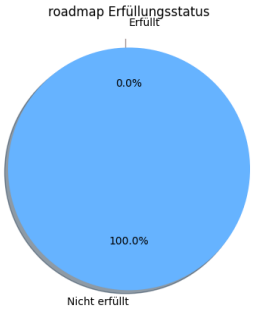
ERM.3: Regelmäßige Veröffentlichung von Releases



ERM.1: Eindeutiger Release-Name



EAM.2: Eindeutige Anforderungserfassung



EÄM.4: Existenz einer Roadmap

Abbildung 6: Kreisdiagramme zur Darstellung der Verteilung spezifischer Empfehlungen unter den 130 analysierten Python-Projekten. Jedes Diagramm visualisiert den Anteil der Projekte, die die jeweiligen Empfehlungen erfüllen, und bietet einen Überblick über die Implementierungshäufigkeit dieser Empfehlungen im gesamten Datensatz

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Oliver Zablocki