Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Lan Mong Luu Van

Leistungsfähigkeit von Large Language Models bei der Fehlererkennung und Fehlerbehebung im Softwaretest

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department of Computer Science

Lan Mong Luu Van

Leistungsfähigkeit von Large Language Models bei der Fehlererkennung und Fehlerbehebung im Softwaretest

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Erstprüfer/in: Prof. Dr. Ulrike Steffens
Zweitprüfer/in: Prof. Dr. Bettina Buth

Abgabedatum: 17.06.2025

Zusammenfassung

Name des Studierenden

Lan Mong Luu Van

Thema der Bachelorthesis

Leistungsfähigkeit von Large Language Models bei der Fehlererkennung und Fehlerbehebung im Softwaretest

Stichworte

Large Language Models, Softwaretest, Fehlererkennung, Fehlerbehebung, Künstliche Intelligenz

Kurzzusammenfassung

Softwaretests sind ein zentraler Bestandteil der Qualitätssicherung, der jedoch oft mit erhöhten Kosten und Zeit verbunden ist. Large Language Models zeigen vielversprechendes Potenzial zur Unterstützung im Testprozess. Ziel dieser Arbeit ist die systematische Untersuchung der Leistungsfähigkeit von Large Language Models in den Testaktivitäten Fehlererkennung und Fehlerbehebung.

Hierzu wurden das generische Modell ChatGPT in der Version GPT-4 und das auf Code spezialisierte Modell Qwen2.5-Coder-32B-Instruct miteinander verglichen. Die Modelle erhielten standardisierte Eingaben unter Verwendung von Zero-Shot- und Chain-of-Thought-Prompts, um zusätzlich den Einfluss von Prompting auf die Leistungsfähigkeit zu ermitteln. Als Testdatensatz diente die Java-Version von QuixBugs. Die Bewertung der Modelle erfolgte anhand quantitativer Evaluationskriterien und qualitativer Beobachtungen.

Die Ergebnisse zeigen, dass beide Modelle grundsätzlich in der Lage sind, diese Testaufgaben zu bewältigen. ChatGPT erreichte in der Fehlererkennung hohe Recall-Werte, erzeugte jedoch häufiger False Positives. Qwen2.5-Coder-32B-Instruct wies eine höhere Precision auf. Der Fehlertyp incorrect output stellte sich als besonders problematisch dar. In der Fehlerbehebung erzielte ChatGPT mit Zero-Shot-Prompting eine Erfolgsrate von 93%, deutlich über früheren Studien. Ein signifikanter Zusammenhang zwischen Codekomplexität und Reparaturerfolg konnte nicht festgestellt werden. Chain-of-Thought-Prompts führten zu deutlich längeren Bearbeitungszeiten.

Die Ergebnisse unterstreichen das Potenzial von Large Language Models im Softwaretest. Die Effektivität ist jedoch abhängig von Modelltyp und Prompting-Strategie. Des Weiteren sind die Ergebnisse aufgrund der methodischen Einschränkungen des synthetischen Datensatzes nur eingeschränkt auf reale Projekte übertragbar.

Name of Student

Lan Mong Luu Van

Title of the paper

Performance of large language models in error detection and error correction in software testing

Keywords

Large language models, software testing, error detection, error correction, artificial intelligence

Abstract

Software testing is a central component of quality assurance, but is often associated with increased costs and time. Large language models show promising potential for supporting the testing process. The aim of this thesis is to systematically analyse the performance of large language models in the test activities of error detection and error correction.

For this purpose, the generic model ChatGPT in the version GPT-4 and the model Qwen2.5-Coder-32B-Instruct, which specialises in code, were compared with each other. The models were given standardised input using zero-shot and chain-of-thought prompts in order to additionally determine the influence of prompting on performance. The Java version of QuixBugs was used as the test data set. The models were assessed using quantitative evaluation criteria and qualitative observations.

The results show that both models are principally able to cope with these test tasks. ChatGPT achieved high recall values in error detection, but generated false positives more frequently. Qwen2.5-Coder-32B-Instruct showed a higher precision. The incorrect output error type proved to be particularly problematic. In error correction, ChatGPT with zero-shot prompting achieved a success rate of 93%, well above previous studies. A significant correlation between code complexity and repair success could not be established. Chain-of-thought prompts led to significantly longer processing times.

The results emphasise the potential of large language models in software testing. However, the effectiveness depends on the model type and prompting strategy. Furthermore, due to the methodological limitations of the synthetic data set, the results can only be transferred to real projects to a limited extent.

Vorwort

Ich bedanke mich bei jeder Person, die mir bei der Anfertigung meiner Bachelorarbeit geholfen hat. Mein besonderer Dank geht an meine Betreuerinnen Prof. Ulrike Steffens und Prof. Bettina Buth für ihre wertvolle Begleitung und fachliche Hilfe.

Inhaltsverzeichnis

Abbildungsverzeichnis **viii**

Tabellenverzeichnis **ix**

Abkürzungsverzeichnis **x**

1 Einführung **1**
1.1 Problemstellung und Motivation 1
1.2 Zielsetzung und Abgrenzung 2
1.3 Aufbau 3

2 Theoretische Grundlagen **4**
2.1 Software Engineering 4
2.2 Softwaretest 4
2.2.1 Testfall 5
2.2.2 Softwarefehler 5
2.2.3 Softwarequalität 6
2.2.4 Testaktivitäten 6
2.3 LLMs 7
2.3.1 Transformer-Architektur 7
2.3.2 Tokenisierung 8
2.4 Prompting und Prompt-Engineering 8
2.4.1 Ziele 9
2.4.2 Prompting-Strategien 9

3 Aktueller Forschungsstand LLMs im Software Engineering und Softwaretest **11**
3.1 LLM4SE: LLMs for Software Engineering 11
3.2 Aktueller Forschungsstand LLMs im Software Engineering 12
3.3 Aktueller Forschungsstand LLMs im Softwaretest 14
3.3.1 Ausgewählte Studien und Anwendungsfelder 15
3.3.2 Herausforderungen und offene Fragen 18
3.4 Einordnung der eigenen Arbeit 19
3.4.1 Motivation Fehlererkennung und Fehlerbehebung 19
3.4.2 Motivation LLM-Tools 20
3.5 Fazit 21

4 Methodik und Vorgehensweise **23**
4.1 Forschungsdesign und Vorgehen 23
4.2 Datengrundlage: QuixBugs 24
4.3 Auswahl der LLM-Tools 24
4.4 Prompt-Design/-Engineering 25
4.5 Evaluationskriterien 26
4.5.1 Fehlererkennung 26
4.5.2 Fehlerbehebung 27
4.5.3 Effizienz 28
4.6 Durchführung und Datenerhebung 28
4.7 Datenanalyse & Auswertung 29
4.7.1 Fehlererkennung und Fehlerbehebung 29
4.7.2 Vertiefende Analysen 30
4.8 Gütekriterien 30
4.9 Einschränkungen 31

5 Ergebnisdarstellung **32**
5.1 Fehlererkennung 33
5.1.1 Übersicht über erkannte und nicht erkannte Fehler 33
5.1.2 Vergleich quantitativer Erkennungsmetriken 34
5.1.3 Verteilung der Erkennungsmetriken auf Programmebene . 34
5.1.4 Zuordnung nicht erkannter Fehler 36
5.1.5 Fehlersensitivität bei korrekt funktionierendem Code . . . 37
5.2 Fehlerbehebung 38
5.2.1 Erfolgsrate der Fehlerbehebung 38
5.2.2 Übersicht zur strukturellen Komplexität der Fehlerbehebung 38
5.2.3 Vertiefte Analyse der Reparaturleistung 39
5.2.4 Fehlerabhängigkeit der Fehlerbehebung 42
5.3 Effizienz 43

6 Diskussion **44**
6.1 Zusammenfassung und Interpretation der Ergebnisse 44
6.1.1 Fehlererkennung 44
6.1.2 Erfolgsraten der Fehlerbehebung 45
6.1.3 Effizienz 46
6.1.4 Analyse potenzieller Zusammenhänge 47
6.2 Schwächen 48
6.3 Empfehlungen für zukünftige Arbeiten 50

7 Fazit **53**
7.1 Zusammenfassung 53
7.2 Ausblick 54

Literaturverzeichnis **55**

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit **59**

Abbildungsverzeichnis

3.1 Anzahl wissenschaftlicher Veröffentlichungen zu LLMs im Software Engineering von 2020 - 2024 12
3.2 Thematische Verteilung von Forschungsarbeiten zu LLMs im Software Engineering 13
3.3 Anzahl wissenschaftlicher Veröffentlichungen zu LLMs im Softwaretest von 2020 - 2024 14
3.4 Verteilung der analysierten Aufgabenbereiche beim Einsatz von LLMs im Software Engineering. Die Zahlen in Klammern geben jeweils die Anzahl der zugehörigen Studien an. 15

4.1 Ausgabe von ChatGPT für Prompt 1 beim Programm GCD . . . 29

5.1 Precision-Verteilung 35
5.2 Recall-Verteilung 35
5.3 F1-Score-Verteilung 36
5.4 Häufigkeit von FNs nach Fehlersymptom 37
5.5 Anzahl fehlgeschlagener Reparaturen je Modell und Prompt . . . 40
5.6 LOC-Verteilung 41
5.7 CC-Verteilung 41
5.8 Häufigkeit fehlgeschlagener Reparaturen nach Fehlersymptom . . 42

Tabellenverzeichnis

4.1 Verteilung der Failure Symptoms im Datensatz 24

5.1 Anzahl der Anfragen pro Modell und Prompt-Variante 32
5.2 Fehlererkennung nach Modell und Prompt: TP, FP, FN und erkannte Schwachstellen 33
5.3 Precision, Recall und F1-Score nach Modell und Prompt 34
5.4 Vergleich der Fehlersensitivität von ChatGPT und QC 37
5.5 Vergleich der Fehlerbehebungsleistung nach Modell und Prompt . 38
5.6 Durchschnittliche LOC und CC der generierten Lösungen 39
5.7 Beispielprogramme mit unterschiedlicher Komplexität und Reparaturerfolg aus ChatGPT Prompt 1 39
5.8 Durchschnittliche Bearbeitungsdauer der Modelle pro Prompt . . 43

Abkürzungsverzeichnis

LLM Large Language Model

KI Künstliche Intelligenz

QC Qwen2.5-Coder-32B-Instruct

TP True Positive

FP False Positive

FN False Negative

CC Zyklomatische Komplexität

LOC Lines of Code

Kapitel 1

Einführung

1.1 Problemstellung und Motivation

Softwaretests sind ein integraler Bestandteil der Softwareentwicklung, indem sie einen entscheidenden Beitrag zur Qualität, Zuverlässigkeit und Funktionalität leisten. Allerdings sind diese Testprozesse oftmals kostenintensiv und erfordern einen beträchtlichen Ressourcenaufwand. Viele Projekte stehen unter dem zunehmenden Druck, Software schnellstmöglich bereitzustellen und vernachlässigen dadurch oft das Testen, was schwerwiegende Folgen haben kann (Hossain, 2018).

Ein signifikantes Beispiel für die Konsequenzen ungenügender Softwaretests ist der Fall des Therac-25, ein computergesteuertes Strahlentherapiegerät aus den 1980er Jahren. Aufgrund mangelhafter Tests kam es zu Fehlern in der Software, die zu tödlichen Überdosierungen von Patienten führten. Kritische Randbedingungen blieben unentdeckt und unklare Fehlermeldungen bei der Bedienung wurden ignoriert. Die mangelnde Sensibilität für umfassende Testabdeckung und automatisierte Fehlererkennung machte diesen Vorfall zu einer der schwersten Katastrophen durch fehlerhafte Software in der Medizintechnik (Wong et al., 2010).

Dieses Beispiel unterstreicht auch Jahrzehnte später die Relevanz eines umfassenden Testverfahrens, um Fehler frühzeitig zu identifizieren und zu beheben. Angesichts der fortschreitenden Digitalisierung nimmt die Komplexität der Softwaresysteme zu. Eine vollständige Testabdeckung wird zunehmend schwieriger, wodurch die gezielte Planung von Testaktivitäten an Bedeutung gewinnt. Innovative und strategische Maßnahmen sind notwendig, um der wachsenden Komplexität wirksam entgegenzukommen (Gurcan et al., 2022).

Mit dem Aufstieg von Large Language Models (LLMs) eröffnen sich neue Möglichkeiten, aktuelle Herausforderungen zu bewältigen und den Testprozess grundlegend zu verändern. In der Softwarebranche wächst das Interesse an Künstlicher Intelligenz, da sie das Potenzial bietet, verschiedene Testaktivitäten effizient zu unterstützen.

Eine Untersuchung von CapGemini zeigt, dass die Softwarebranche im Jahr

2024 rund 9,4 Millionen US-Dollar in Künstliche Intelligenz (KI) für Software Engineering investierte, mit einer prognostizierten Erhöhung auf 13 Millionen US-Dollar im Jahr 2025. Des Weiteren wird erwartet, dass bis 2026 rund 85% der Softwareentwickler generative KI-Tools in ihre Arbeitsabläufe einbinden werden. Dies unterstreicht nicht nur das wachsende Interesse an KI, sondern auch die Dringlichkeit, sich mit den Auswirkungen und Herausforderungen dieser Technologie auseinanderzusetzen (Capgemini Research Institute, 2024).

Trotz des wachsenden Interesses besteht weiterhin ein erheblicher Forschungsbedarf hinsichtlich der tatsächlichen Effektivität und Zuverlässigkeit von LLMs. Zusätzlich zeigt sich, dass die Qualität und Vollständigkeit der generierten Antworten stark von der Gestaltung der Eingabeaufforderungen und der Modellarchitektur abhängen. Robuste Evaluationsmethoden sind notwendig, um KI-gestützte Testverfahren miteinander zu vergleichen (Fan et al., 2023).

1.2 Zielsetzung und Abgrenzung

Das Ziel dieser Arbeit ist es, die Leistungsfähigkeit von LLMs im Softwaretest systematisch zu untersuchen. Im Fokus steht dabei der Vergleich zwischen dem generischen Modell ChatGPT in der Version GPT-4 und dem auf Programmieraufgaben optimierten Modell Qwen2.5-Coder-32B-Instruct (QC). Beide Modelle werden in den Testaktivitäten Fehlererkennung und Fehlerbehebung evaluiert.

Die Leistungsfähigkeit wird in einer experimentellen Untersuchung auf Grundlage des öffentlich verfügbaren QuixBugs-Datensatzes in der Java-Version gemessen. Dieser Datensatz enthält gezielt fehlerhafte Codeabschnitte und hat sich als Benchmark in der Softwaretest-Forschung etabliert (Lin et al., 2017). Die untersuchten Modelle erhalten identische standardisierte Eingaben, die mithilfe der Prompting-Strategien Zero-Shot und Chain-of-Thought gestaltet werden. Die generierten Antworten werden strukturiert erfasst und anhand definierter quantitativer Kriterien bewertet und durch qualitative Beobachtungen ergänzt.

Die Untersuchung beschränkt sich ausschließlich auf die Testaktivitäten der Fehlererkennung und -behebung. Andere Testbereiche werden nicht behandelt. Zudem erfolgt keine Evaluation in realen Softwareprojekten, sondern unter kontrollierten Bedingungen mit reproduzierbaren Daten. Darüber hinaus wird analysiert, ob sich signifikante Unterschiede in der Leistungsfähigkeit zwischen den beiden Modelltypen zeigen. Neben der Modellleistung werden die Auswirkungen unterschiedlicher Prompting-Strategien auf die Ergebnisqualität und Laufzeit untersucht.

Für diese vorliegende Arbeit ergeben sich folgende Forschungsfragen:

1. Wie leistungsfähig sind LLMs in der Testaktivität der Fehlererkennung?
2. Wie leistungsfähig sind LLMs in der Testaktivität der Fehlerbehebung?
3. Wie unterscheidet sich die Leistungsfähigkeit von ChatGPT und QC in diesen Aufgaben?
4. Welchen Einfluss haben unterschiedliche Prompting-Strategien auf die Leistung der LLMs bei Fehlererkennung und Fehlerbehebung?

Diese Arbeit soll zu einem besseren Verständnis der Einsatzmöglichkeiten generischer und auf Programmieraufgaben spezialisierter LLMs im Softwaretest beitragen. Im Mittelpunkt steht die Fähigkeit der Sprachmodelle zur automatisierten Fehlererkennung und -behebung sowie die Einflüsse verschiedener Prompting-Strategien unter standardisierten Evaluationsbedingungen.

1.3 Aufbau

Diese Arbeit gliedert sich in insgesamt sieben Kapitel. Kapitel 1 führt in die Thematik ein, beschreibt die Problemstellung sowie die Zielsetzung und formuliert die zentralen Forschungsfragen.

In Kapitel 2 werden die theoretischen Grundlagen behandelt. Es beginnt mit einem Überblick über zentrale Prinzipien und Methoden des Softwaretestens. Anschließend werden die Grundlagen der LLMs vorgestellt. Zudem wird das Konzept des Prompt-Engineerings einschließlich verschiedener Prompting-Strategien systematisch erläutert.

Kapitel 3 widmet sich dem aktuellen Forschungsstand zum Einsatz von LLMs im Software Engineering und im Softwaretest. Es werden existierende Arbeiten betrachtet und untersucht, inwiefern LLMs bereits zur Unterstützung verschiedener Testaktivitäten eingesetzt wurden, welche Herausforderungen sich dabei ergeben und welche Erkenntnisse diese liefern.

Die Methodik wird in Kapitel 4 dargelegt. Neben der Beschreibung des gewählten Forschungsdesigns werden die eingesetzten Werkzeuge, Modelle und Prompt-Strategien erläutert. Zudem werden die Evaluationskriterien definiert und der Ablauf der Datenerhebung sowie der Datenanalyse systematisch beschrieben.

Kapitel 5 enthält die Ergebnisdarstellung. Die aus der Untersuchung erhobenen Daten werden nach den zuvor definierten Kriterien ausgewertet, tabellarisch und grafisch aufbereitet sowie entlang der Forschungsfragen strukturiert präsentiert.

Kapitel 6 umfasst die Diskussion der Ergebnisse. Dabei wird die Leistungsfähigkeit der LLMs sowie mögliche Stärken und Schwächen der Modelle untersucht und in den Kontext bestehender Forschung eingeordnet.

Abschließend fasst Kapitel 7 die zentralen Erkenntnisse der Arbeit zusammen, beantwortet die Forschungsfragen auf Grundlage der Ergebnisse und leitet abschließend mögliche Perspektiven für weiterführende Forschung und Anwendung ab.

Kapitel 2

Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen für die vorliegende Arbeit gelegt. Es führt in die zentralen Begriffe und Konzepte ein, die für das Verständnis der Untersuchung der Leistungsfähigkeit von LLMs im Softwaretestkontext relevant sind. In diesem Zusammenhang werden die Grundlagen des Softwaretestens, der LLMs sowie das Konzept des Promptings beleuchtet.

2.1 Software Engineering

Software Engineering ist ein systematischer, ingenieurwissenschaftlicher Ansatz zur Planung, Entwicklung, Wartung und Qualitätssicherung von Software. Es umfasst den gesamten Lebenszyklus eines Softwareprodukts, von der Anforderungsanalyse über die Implementierung bis hin zur Validierung und Weiterentwicklung. Das Ziel besteht darin, Softwarelösungen zu entwickeln, die funktional korrekt, zuverlässig und langfristig wartbar sind.(Sommerville, 2018).

2.2 Softwaretest

Die folgenden Begriffsklärungen zum Thema Softwaretest im Abschnitt 2.2 orientieren sich ausschließlich an Witte (2019).

Ein zentraler Bestandteil im Software Engineering ist der Softwaretest. Softwaretests stellen einen essenziellen Bestandteil des Entwicklungsprozesses dar und dienen der Sicherstellung der Qualität und Zuverlässigkeit von Softwareprodukten. Sie ermöglichen die Überprüfung, ob ein System den funktionalen und nicht-funktionalen Anforderungen genügt und unter definierten Bedingungen erwartungsgemäß funktioniert. Dabei werden Sollwerte aus der Spezifikation mit den Istwerten verglichen, um Fehler zu identifizieren. Die im Anforderungskatalog definierten Spezifikationen werden mit den tatsächlich erreichten Ergebnissen des Systems abgeglichen, um potenzielle Abweichungen frühzeitig zu identifizieren und geeignete Korrekturmaßnahmen einzuleiten.

Wie Witte (2019) betont, ist das Testen ein zentraler Teilprozess der Qualitätskontrolle mit dem primären Ziel, Fehler und Fehlfunktionen bereits vor

dem produktiven Einsatz aufzudecken. Dabei ist es wichtig zu verstehen, dass ein bestandener Test nicht zwangsläufig die Fehlerfreiheit der Software belegt, sondern lediglich nachweist, dass in einem bestimmten Fall kein Fehler aufgetreten ist. Ein Test gilt vielmehr dann als erfolgreich, wenn er vorhandene Fehler aufdeckt.

Der Prozess des Softwaretests ist iterativ und kontinuierlich. Da moderne Softwaresysteme in mehreren Entwicklungszyklen entstehen und fortlaufend weiterentwickelt werden, sind auch die zugehörigen Testaktivitäten wiederkehrend durchzuführen. Auf diese Weise tragen Tests nicht nur zur kurzfristigen Fehlererkennung, sondern auch zur langfristigen Stabilität und Wartbarkeit von Softwareprodukten bei.

Neben der Prüfung funktionaler Anforderungen leisten Softwaretests zudem einen wichtigen Beitrag zur Evaluation nicht-funktionaler Qualitätsmerkmale. Zu den entscheidenden Kriterien für die Gesamtqualität eines Systems zählen insbesondere Performance, Sicherheit, Benutzerfreundlichkeit und Skalierbarkeit.

2.2.1 Testfall

Ein grundlegendes Element des Softwaretests ist der Testfall. Dieser beschreibt eine klar definierte Kombination aus Eingabedaten, Rahmenbedingungen und der erwarteten Systemausgabe. Mithilfe des Testfalls wird eine bestimmte Softwarefunktion oder ein definiertes Systemverhalten überprüft. Ziel ist es dabei zu evaluieren, ob das System das gewünschte Verhalten entsprechend der Spezifikation tatsächlich zeigt. Besonders bedeutsam ist die Überprüfung, ob der tatsächliche Ablauf des Programms mit dem erwarteten Ergebnis übereinstimmt, vor allem bei kritischen Anwendungsfällen. Ein Testfall sollte nachvollziehbar, eindeutig dokumentiert und wiederholbar sein. Um seine Wirksamkeit dauerhaft zu gewährleisten, müssen Testfälle regelmäßig gepflegt und an geänderte Anforderungen oder Systemupdates angepasst werden. Ohne eine kontinuierliche Evaluation können Testfälle veralten oder redundant werden. Dadurch wird die Testabdeckung und Aussagekraft der Testergebnisse erheblich eingeschränkt. Daher sollten Testende nicht nur der Testspezifikation folgen, sondern diese auch kritisch hinterfragen und bei Bedarf überarbeiten, um die langfristige Effektivität der Tests sicherzustellen.

2.2.2 Softwarefehler

Im Zentrum aller Testaktivitäten steht die Identifikation und Behebung von Softwarefehlern, die auch als Defekte oder Bugs bezeichnet werden. Sie stellen ein allgegenwärtiges Problem dar und gelten in modernen Softwaresystemen als eine der Hauptquellen für Qualitätseinbußen. Die Sicherstellung von Qualität, Sicherheit und Zuverlässigkeit ist daher ein fundamentales Ziel der Softwareentwicklung. Fehlererkennung und -behebung sind dabei von entscheidender Bedeutung, da sie unmittelbar beeinflussen, ob fehlerhafte Komponenten rechtzeitig erkannt und korrigiert werden können.

Softwarefehler können in unterschiedlichen Formen auftreten und vielfältige Ursachen haben. Zu den gängigsten zählen lexikalische, syntaktische, semantische

oder numerische Fehler, aber auch Spezifikationsfehler, die auf unklare oder widersprüchliche Anforderungen zurückzuführen sind. Die Auswirkungen dieser Fehler sind ebenso vielfältig wie ihre Ursachen. Entsprechend ist der Begriff des Softwarefehlers nicht nur quantitativ, sondern auch qualitativ zu verstehen. Die Relevanz eines Fehlers ergibt sich aus dem Zusammenspiel von Auftrittswahrscheinlichkeit und möglicher Schadenshöhe.

2.2.3 Softwarequalität

Eng verbunden mit dem Auftreten von Softwarefehlern ist der Begriff der Softwarequalität. Sie stellt eine mehrdimensionale Eigenschaft dar, die durch verschiedene Faktoren beeinflusst wird. Gemäß der DIN-ISO-Norm 9126 bezeichnet Softwarequalität die Gesamtheit aller Merkmale, die die Eignung eines Produkts zur Erfüllung spezifizierter Anforderungen bestimmen. Ein zentrales Qualitätsmerkmal ist dabei die Funktionalität. Funktionale Fehler entstehen häufig durch unzureichende oder fehlerhafte Implementierungen, die dazu führen, dass die Software nicht wie vorgesehen arbeitet. Die Mehrheit der in der Praxis beobachtbaren Defekte lässt sich auf klassische Implementierungsfehler zurückführen.

2.2.4 Testaktivitäten

Softwaretests bestehen aus einer Vielzahl unterschiedlicher Testaktivitäten, die jeweils spezifische Ziele im Rahmen der Qualitätssicherung verfolgen. Laut Witte (2019) sind Testaktivitäten konkrete Aufgaben innerhalb des Testprozesses, die zur Identifikation von Fehlern, zur Verifikation von Anforderungen und zur Sicherstellung der allgemeinen Softwarequalität beitragen. Je nach Teststufe können sich die Schwerpunkte der Aktivitäten deutlich unterscheiden. Während in frühen Phasen der Fokus häufig auf der Validierung einzelner Komponenten liegt, rücken in späteren Phasen Aspekte wie Performanz, Sicherheit oder Nutzerfreundlichkeit stärker in den Vordergrund.

Fehlererkennung

Eine zentrale Aktivität im Softwaretest ist die Fehlererkennung. Sie zielt darauf ab, Abweichungen zwischen dem erwarteten und dem tatsächlichen Verhalten des Systems aufzudecken. Neben dynamischen Tests tragen auch statische Analysemethoden und manuelle Reviews zur frühzeitigen Fehlererkennung bei.

Ein möglichst früher Einsatz von Tests erhöht die Effizienz, da Fehler in frühen Phasen mit deutlich geringerem Aufwand behoben werden können. In automatisierten Testsystemen erfolgt die Fehlererkennung in der Regel durch den Abgleich von erwarteter und tatsächlicher Ausgabe. Dabei können automatisierte Fehlermeldungen generiert werden. Die Wirksamkeit der Fehlererkennung lässt sich mithilfe quantitativer Kennzahlen wie der Fehlerentdeckungsrate oder der Fehlerfindungsproduktivität evaluieren.

Fehlerbehebung

Auf die Fehlererkennung folgt die Fehlerbehebung, die ebenfalls eine zentrale Rolle im Testprozess einnimmt. Unter Fehlerbehebung versteht man die Schritte

der Erfassung, Beschreibung, Analyse, Korrektur und Validierung eines identifizierten Fehlers. Ziel ist es, den Fehler vollständig zu eliminieren, und zwar mit möglichst geringem Aufwand an Zeit und Ressourcen. Insbesondere in späten Projektphasen kann die Korrektur eines Fehlers erhebliche Kosten verursachen, weshalb eine frühzeitige Entdeckung und Behebung angestrebt wird.

Ein wichtiger Bestandteil der Fehlerbehebung ist das Debugging, bei dem Fehler im Quellcode gezielt lokalisiert und analysiert werden. Dies wird typischerweise durch die Entwickler selbst durchgeführt. Ein strukturierter Prozess des Fehlermanagements ist dabei entscheidend, um erkannte Defekte systematisch und nachvollziehbar zu bearbeiten.

2.3 LLMs

Die folgenden Ausführungen in Abschnitt 2.3 zur Funktionsweise von LLMs, zur Transformer-Architektur sowie zur Tokenisierung basieren ausschließlich auf Campesato (2024)

LLMs sind eine Klasse tiefer neuronaler Netzwerke, die auf der Verarbeitung, Generierung und Interpretation natürlicher Sprache spezialisiert sind. Sie basieren typischerweise auf der Transformer-Architektur und wurden auf umfangreichen Mengen unstrukturierter Textdaten trainiert. Diese Modelle verfügen über Milliarden von Parametern und sind in der Lage, komplexe sprachliche Muster zu erkennen sowie menschenähnliche Ausgaben zu erzeugen.

Charakteristisch für LLMs ist ihr prompt-basiertes Arbeiten. Ihre Ausgaben hängen stark von der Formulierung der Eingabeaufforderung, dem Prompt, ab. Diese Prompts steuern, welche Teile des trainierten Wissens aktiviert und wie die Informationen verarbeitet werden. Das Lernen erfolgt über selbstüberwachtes Lernen, bei dem das Modell anhand unbeschrifteter Daten statistische Zusammenhänge erlernt, ohne auf manuelle Annotationen angewiesen zu sein.

Gleichzeitig bestehen jedoch Einschränkungen. LLMs sind nicht deterministisch, sondern probabilistisch. Ihre Antworten basieren auf Wahrscheinlichkeiten, die zu nicht reproduzierbaren oder fehlerhaften Ausgaben führen können. Hinzu kommt der Black-Box-Charakter vieler Modelle, der die Nachvollziehbarkeit ihrer Entscheidungen erschwert.

Durch ihre generalisierende Sprachverarbeitung kommen LLMs in vielen Anwendungsfeldern zum Einsatz, von der Textgenerierung über Übersetzungen bis hin zur Analyse und Generierung von Programmcode. Diese Fähigkeiten machen sie für den Einsatz in der Softwareentwicklung und im Softwaretest interessant.

2.3.1 Transformer-Architektur

Die Transformer-Architektur bildet das technologische Fundament moderner LLMs. Ihre Einführung hat die Verarbeitung natürlicher Sprache grundlegend verändert. Im Gegensatz zu rekurrenten neuronalen Netzwerken, die Sprache sequenziell verarbeiten, ermöglicht der Transformer die parallele Verarbeitung aller Eingabeelemente. Dies führt zu einer Effizienzsteigerung und erlaubt es, auch lange Textsequenzen mit komplexen Abhängigkeiten zu modellieren.

Zentraler Bestandteil dieser Architektur ist der Self-Attention-Mechanismus, der es dem Modell ermöglicht, die Beziehungen zwischen allen Token einer Eingabesequenz gleichzeitig zu analysieren. Dabei werden sogenannte Attention-Gewichte berechnet, die bestimmen, wie stark ein Token bei der Berechnung der Ausgabe berücksichtigt wird. Durch diesen Mechanismus kann das Modell kontextuelle Abhängigkeiten über weite Distanzen hinweg erfassen. Dies ist eine entscheidende Fähigkeit für die semantisch präzise Sprachverarbeitung.

Ein weiteres Merkmal ist die Multi-Head Attention, bei der mehrere Attention-Mechanismen parallel eingesetzt werden, um unterschiedliche Beziehungsmuster gleichzeitig zu erfassen. Ergänzt wird die Architektur durch Feedforward-Netzwerke, die innerhalb jeder Ebene zusätzliche Transformationen ermöglichen.

Durch diese strukturellen Eigenschaften hat sich der Transformer als hochgradig skalierbare und effektive Grundlage für leistungsfähige Sprachmodelle etabliert. Seine Einführung markierte einen Wendepunkt im Natural Language Processing und bildet bis heute den Kern nahezu aller hochperformanten LLMs.

2.3.2 Tokenisierung

Token stellen im Kontext von LLMs die grundlegenden Verarbeitungseinheiten dar, in die Text vor der Modellverarbeitung zerlegt wird. Dieser Prozess wird als Tokenisierung bezeichnet und ist ein zentraler Bestandteil der Vorverarbeitung und eng mit der Transformer-Architektur verknüpft. Je nach verwendetem Tokenizer können Token vollständige Wörter, Teilwörter oder einzelne Zeichen sein. Moderne LLMs nutzen häufig subword-basierte Ansätze, um seltene Begriffe in kleinere Einheiten aufzuteilen, die die Effizienz erhöhen und das Vokabular kompakt halten. Zusätzlich können spezielle Steuerzeichen wie [CLS] oder [SEP] eingefügt werden, um die Modelllogik zu unterstützen. Dabei dient [CLS] oft als Startzeichen und [SEP] als Trenner für unterschiedliche Textsegmente.

Neben der Tokenisierung werden die Token in numerische IDs umgewandelt und in Vektoren projiziert, die als sogenannte Token-Embeddings dem Modell als Eingabe dienen. Die Verarbeitung erfolgt sequenziell auf Token-Basis, sowohl bei der Eingabe als auch bei der Ausgabe. Die maximale Anzahl verarbeitbarer Token pro Durchlauf ist durch die Architektur begrenzt. Werden diese Grenzen überschritten, werden überzählige Token verworfen, ein Prozess, der als Trunkierung bezeichnet wird. Die Qualität der Tokenisierung hat einen erheblichen Einfluss auf die Ausgabequalität, insbesondere in Bezug auf die Struktur und Effizienz von Prompts.

2.4 Prompting und Prompt-Engineering

Die Inhalte in Abschnitt 2.4 zu Prompting und Prompt-Engineering orientieren sich ausschließlich an Campesato (2024).

Mit der zunehmenden Integration von LLMs in den Softwareentwicklungsprozess hat sich das sogenannte Prompt-Engineering als ein zentraler Erfolgsfaktor für den praktischen Einsatz dieser Modelle etabliert. Während LLMs in der Lage sind, eine Vielzahl komplexer Aufgaben auszuführen, hängt die Qualität ihrer Ausgaben maßgeblich von der Formulierung der Eingabeaufforderung ab.

Prompt-Engineering bezeichnet somit die gezielte Gestaltung und Optimierung von Prompts, um LLMs für spezifische Aufgaben zu steuern und qualitativ hochwertige, relevante Ausgaben zu erzielen.

2.4.1 Ziele

Ein Prompt stellt die textuelle Eingabe dar, mit der ein LLM instruiert wird, eine bestimmte Aufgabe auszuführen. Da LLMs keine eigenständige Zielsetzung besitzen, leitet ein präzise formulierter Prompt das Modell dazu an, die gewünschte Antwortstruktur, den inhaltlichen Fokus sowie die Qualität der Ausgabe systematisch zu beeinflussen.

Prompt-Engineering verfolgt im Wesentlichen drei zentrale Zielsetzungen, und zwar die Führung des Modells, die Optimierung der Ausgabequalität und die klare Spezifikation der jeweiligen Aufgabe. Ein sorgfältig formulierter Prompt ermöglicht zunächst die gezielte Steuerung des Modellverhaltens und definiert somit die inhaltliche Ausrichtung der Antwort. Er legt auch fest, ob das Modell beispielsweise analysierend, erklärend oder korrektiv agieren soll. Des Weiteren hat die Ausgestaltung des Prompts unmittelbaren Einfluss auf die Qualität der generierten Ausgabe. Durch präzisere, kontextreichere oder strukturiertere Prompts können LLMs relevantere, konsistentere und informativer aufbereitete Antworten erzeugen. Schließlich bestimmt der Prompt maßgeblich, welche konkrete Aufgabe das Modell überhaupt bearbeitet.

Prompt-Engineering wird damit zunehmend als sowohl wissenschaftliche als auch kreative Disziplin verstanden. Es wird durch experimentelles Verfeinern, iteratives Testen und kontextbezogene Formulierungen optimiert. Gerade in der automatisierten Softwareanalyse und Fehlerbehebung ist diese Technik von hoher Bedeutung, da hier präzise und kontextbezogene Modellantworten erforderlich sind.

2.4.2 Prompting-Strategien

Aufbauend auf den zuvor beschriebenen Zielen und Prinzipien des Prompt-Engineerings lassen sich verschiedene Strategien unterscheiden, mit denen sich Prompts systematisch gestalten lassen. Diese Strategien unterscheiden sich im Hinblick auf den Grad der expliziten Anleitung und Struktur, die dem Modell bei der Aufgabenbearbeitung gegeben wird. Zu den zentralen Ansätzen zählen das Zero-Shot Prompting und das Chain-of-Thought Prompting, die im Folgenden exemplarisch vorgestellt werden. In dieser Arbeit wurden bewusst Zero-Shot und Chain-of-Thought Prompting als Strategien ausgewählt, da sie sich in der Strukturierung und Zielsetzung unterscheiden. Diese Unterschiede ermöglichen eine differenzierte Untersuchung, wie unterschiedliche Formulierungen von Prompts die Leistungsfähigkeit der LLMs beeinflussen.

Zero-Shot Prompt

Beim Zero-Shot-Prompt besteht die Eingabe ausschließlich aus einer allgemeinen Aufgabenbeschreibung, ohne dass dem Modell explizite Beispiele, Zwischenschritte oder erläuternde Anweisungen bereitgestellt werden. Diese Methode

setzt auf die Fähigkeit des Modells, Aufgaben direkt aus der Aufgabenbeschreibung zu erschließen.

Im Kontext des Softwaretests kann dieser Prompt dazu genutzt werden, um ein Modell aufzufordern, einen fehlerhaften Code zu analysieren, potenzielle Fehler zu identifizieren und eine korrigierte Version vorzuschlagen, ohne dabei Hinweise zur konkreten Vorgehensweise zu liefern.

Chain-of-Thought Prompt

Im Gegensatz dazu fordert das Chain-of-Thought-Prompt das Modell explizit dazu auf, eine Aufgabe schrittweise zu bearbeiten und dabei einen mehrstufigen Denkprozess zu durchlaufen. Diese Strategie zielt darauf ab, das logische Schlussfolgern des Modells zu fördern und strukturiertere, nachvollziehbare Ausgaben zu erzeugen.

In der Analyse von Quellcode kann dies beispielsweise bedeuten, dass das Modell zunächst die Funktion des Codes beschreibt, anschließend einzelne Codezeilen analysiert, erkannte Fehler begründet und schließlich eine überarbeitete Version mit Erläuterungen vorschlägt. Dieser strukturierte Ansatz könnte sich bei komplexeren Aufgaben als vorteilhaft erweisen.

Kapitel 3

Aktueller Forschungsstand LLMs im Software Engineering und Softwaretest

In diesem Kapitel wird der aktuelle Forschungsstand zur Anwendung von LLMs im Bereich des Software Engineerings mit Schwerpunkt auf Softwaretests beschrieben. Zunächst beginnt das Kapitel mit dem Forschungsfeld LLM4SE, das die zielgerichtete Integration von LLMs in verschiedene Aufgaben entlang des Softwareentwicklungsprozesses umfasst.

Im Anschluss wird die Entwicklung dieses Forschungsfelds näher analysiert, insbesondere die Entwicklung der Forschungsaktivität der letzten Jahre sowie die zunehmende Rolle von LLMs in verschiedenen Phasen der Softwareentwicklung. Ein besonderer Schwerpunkt liegt dabei auf Studien, die den Einsatz von LLMs in unterschiedlichen Testaktivitäten untersuchen.

Abschließend wird die vorliegende Arbeit in den bestehenden wissenschaftlichen Kontext eingeordnet. Ziel ist es, die Relevanz der eigenen Untersuchung klar herauszuarbeiten und deren methodische Ausrichtung im Spannungsfeld zwischen etablierten Erkenntnissen und offenen Forschungsfragen zu positionieren.

3.1 LLM4SE: LLMs for Software Engineering

Mit dem Aufstieg leistungsfähiger LLMs hat sich das Forschungsfeld des Software-Engineerings in den letzten Jahren grundlegend gewandelt.

LLM4SE steht für LLMs for Software Engineering und bezeichnet ein Forschungsfeld, das sich mit der systematischen Integration von LLMs in zentrale Aufgabenbereiche des Software Engineerings befasst. Im Zentrum steht die Nutzung der generativen, analysierenden und verstehenden Fähigkeiten moderner

LLMs zur Unterstützung des gesamten Softwareentwicklungsprozesses (Hou et al., 2024).

Ziel dieses Forschungsfeldes ist es, LLMs zur Unterstützung in verschiedenen Phasen des Softwareentwicklungsprozesses einzusetzen, wie etwa im Requirements Engineering, bei der Codierung, im Test, in der Wartung, der Codeüberprüfung und im Schwachstellenmanagement. Mit der Einführung von LLMs, die in der Lage sind, Quellcode zu verstehen, zu analysieren und zu generieren, verändern sich bestehende Entwicklungs- und Wartungsprozesse grundlegend. Tätigkeiten, die bislang manuell, zeitaufwendig oder fehleranfällig waren, können zunehmend automatisiert, beschleunigt und qualitativ verbessert werden (Gao et al., 2025).

Vor diesem Hintergrund stellt sich die Frage, in welchem Umfang und mit welchen Schwerpunkten LLMs bereits in der Forschung zum Software Engineering berücksichtigt werden. Der folgende Abschnitt gibt daher einen systematischen Überblick über den aktuellen Stand wissenschaftlicher Arbeiten in diesem Bereich.

3.2 Aktueller Forschungsstand LLMs im Software Engineering

LLMs haben sich in den vergangenen Jahren zu einer zentralen Technologie in der Softwareentwicklung entwickelt. Durch ihre generativen Fähigkeiten bieten LLMs vielversprechende Ansätze zur Bewältigung zentraler Herausforderungen im Software Engineering.

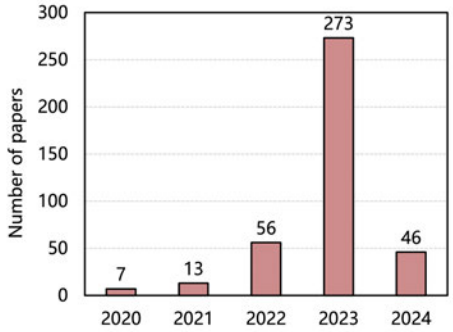

Abbildung 3.1: Anzahl wissenschaftlicher Veröffentlichungen zu LLMs im Software Engineering von 2020 - 2024
Quelle: (Hou et al., 2024)

Eine systematische Literaturübersicht von Hou et al. (2024) analysierte 395 wissenschaftliche Arbeiten zum Einsatz von LLMs im Software Engineering aus dem Zeitraum von 2017 bis Anfang 2024. Das Jahr 2017 als Beginn der Untersuchung wurde gewählt, da in diesem Jahr der erste Artikel zur Transformer-Architektur veröffentlicht wurde, die die Grundlage für die Entwicklung von LLMs bildet. Die Studie zeigt einen exponentiellen Anstieg von Publikationen.

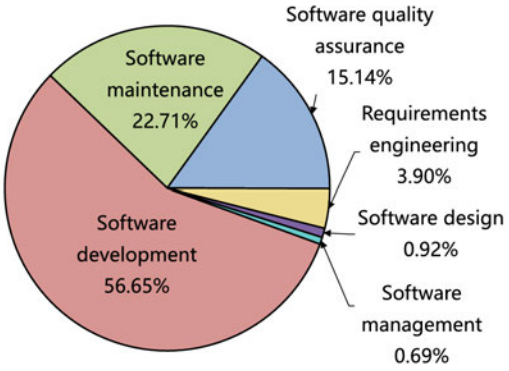

Abbildung 3.2: Thematische Verteilung von Forschungsarbeiten zu LLMs im Software Engineering
Quelle: (Hou et al., 2024)

Während im Jahr 2020 sieben Beiträge erschienen, stieg die Zahl bis 2023 auf 273 an und erreichte ihren bisherigen Höhepunkt (vgl. Abbildung 3.1). Bereits im Januar 2024 wurden 46 neue Arbeiten registriert. Diese Entwicklung unterstreicht die zunehmende Relevanz und Dynamik des Forschungsfelds.

Die Studien lassen sich inhaltlich in sechs Bereiche der Softwareentwicklung einordnen. Die thematische Verteilung der analysierten Studien ist in Abbildung 3.2 dargestellt. Diese sind Requirements Engineering, Software Design, Software Development, Software Maintenance, Software Quality Assurance und Software Management. Ein Schwerpunkt liegt auf den Phasen der Implementierung, insbesondere Software Development, Software Maintenance und Software Quality Assurance. Im Gegensatz dazu existieren vergleichsweise wenige Arbeiten zu den frühen Phasen, wie dem Requirements Engineering oder zu verwaltenden Aufgaben.

Die Relevanz von LLMs im Software Engineering zeigt sich zunehmend auch in der industriellen Praxis. In einer empirischen Studie von Jahić und Sami (2024) zur Nutzung von LLMs in der Softwareentwicklung wurden 15 Softwareunternehmen unterschiedlicher Größe befragt. Die Teilnehmenden verfügten über mehrjährige Berufserfahrung und repräsentierten verschiedene Rollen im Entwicklungsprozess. Die Ergebnisse zeigen, dass etwa zwei Drittel der befragten Unternehmen bereits KI-gestützte Werkzeuge einsetzen. Besonders häufig werden diese für Coding Assistance mit 90% und Code-Generierung mit 80% verwendet. Infolgedessen meldeten 80% der befragten Anwender eine gesteigerte Produktivität.

Innerhalb der identifizierten Anwendungsbereiche nimmt insbesondere der Softwaretest eine zunehmend prominente Rolle ein. Die automatisierte Unterstützung durch LLMs in dieser Phase des Entwicklungsprozesses bietet besonderes Potenzial zur Verbesserung der Effizienz und Qualitätssicherung. Der folgende Abschnitt widmet sich daher dem aktuellen Stand der Forschung zum Einsatz von LLMs im Softwaretest.

3.3 Aktueller Forschungsstand LLMs im Softwaretest

Aufbauend auf den allgemeinen Entwicklungen im LLM-gestützten Software-Engineering befasst sich der folgende Abschnitt mit dem Teilbereich des Softwaretests, in dem LLMs besonders intensiv erforscht werden. Im Fokus stehen dabei Testaktivitäten, wie die Testfallgenerierung und die automatische Fehlerbehebung.

Da Softwaretests entscheidend für die Sicherstellung von Qualität, Korrektheit und Zuverlässigkeit sind, steigt angesichts wachsender Systemkomplexität der Bedarf an skalierbaren und automatisierten Testmethoden (Broy & Kuhrmann, 2021). In diesem Kontext gewinnen LLMs zunehmend an Bedeutung, da sie neue Möglichkeiten zur Automatisierung klassischer Testaktivitäten eröffnen. Die durchdringenden Fortschritte bei LLMs haben verschiedene Domänen, einschließlich des Softwaretests, beeinflusst (Q. Wang et al., 2024).

Eine systematische Literaturübersicht von J. Wang et al. (2024) verdeutlicht auch hier eine dynamische Entwicklung dieses Forschungsbereichs. Während im Jahr 2020 lediglich eine relevante Publikation zum Einsatz von LLMs im Softwaretest erschien, stieg die Zahl relevanter Studien im Jahr 2022 auf 19 und erreichte 2023 mit 82 Arbeiten einen vorläufigen Höhepunkt. Dieser exponentielle Anstieg ist in Abbildung 3.3 erkennbar.

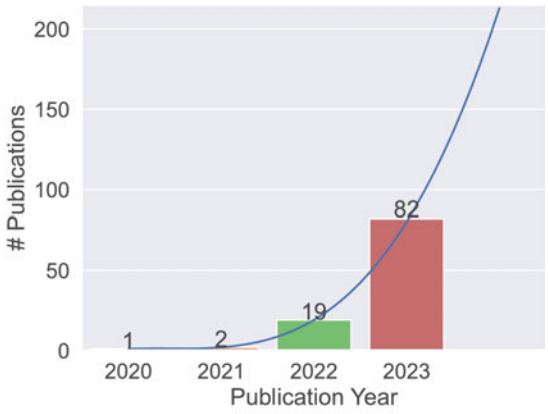

Abbildung 3.3: Anzahl wissenschaftlicher Veröffentlichungen zu LLMs im Softwaretest von 2020 - 2024
Quelle: (J. Wang et al., 2024)

Die Analyse von J. Wang et al. (2024) identifiziert eine Vielfalt möglicher Einsatzfelder, die in Abbildung 3.4 zu erkennen sind. Dazu zählen Testfallgenerierung, automatische Programmreparatur, Fehlersuche und Debugging, Mutanten- und Sicherheitsanalyse. Darin werden insbesondere die Einsatzgebiete Testfallgenerierung und automatische Programmreparatur als forschungsintensive Schwerpunkte herausgearbeitet.

Diese beiden Bereiche zählen zu den am häufigsten untersuchten Einsatzgebieten von LLMs im Softwaretesten.

Abbildung 3.4: Verteilung der analysierten Aufgabenbereiche beim Einsatz von LLMs im Software Engineering. Die Zahlen in Klammern geben jeweils die Anzahl der zugehörigen Studien an.
Quelle: (J. Wang et al., 2024)

Gleichzeitig weist die Studie auf bestehende Forschungslücken hin. Die frühen Phasen des Testprozesses, wie die Ableitung von Testanforderungen oder die Erstellung von Testplänen, wurden bislang nur unzureichend untersucht. Dies verdeutlicht die Notwendigkeit, die Einsatzmöglichkeiten von LLMs nicht nur in der eigentlichen Testdurchführung, sondern auch in vorgelagerten konzeptionellen Aktivitäten systematisch zu erforschen.

Die Relevanz dieser Technologien wird auch in der Industrie erkannt. Laut einer Studie von CapGemini ermöglichen generative KI-Modelle beim Debugging und Testen durchschnittliche Zeitersparnisse von 5%, in Einzelfällen sogar bis zu 20%. Bereits 29% der befragten Unternehmen setzen LLMs gezielt für Debugging-Aufgaben ein. Zudem erwarten Software-Fachkräfte, dass bis 2026 mehr als 25% der Tätigkeiten im Bereich Test und Qualitätssicherung durch generative KI unterstützt werden. Diese Zahlen verdeutlichen das wachsende Potenzial von LLMs, Effizienz und Qualität im Softwaretest nachhaltig zu steigern (Capgemini Research Institute, 2024).

Die Ergebnisse dieser Studien belegen ein zunehmendes Vertrauen in die Leistungsfähigkeit von LLM-basierten Methoden über verschiedene Phasen des Testprozesses hinweg, insbesondere bei der Generierung von Testfällen und der automatisierten Fehlerbehebung.

3.3.1 Ausgewählte Studien und Anwendungsfelder

In diesem Abschnitt werden relevante Studien zum Einsatz von LLMs in verschiedenen Bereichen des Softwaretests vorgestellt. Der Schwerpunkt liegt dabei auf den intensiv untersuchten Anwendungsgebieten wie der Testfallgenerierung, der automatisierten Programmreparatur sowie der Fehlersuche und -analyse.

Testfallgenerierung

Die Generierung von Testfällen stellt eines der am intensivsten untersuchten Anwendungsgebiete im Softwaretest dar. In mindestens 47 Studien wurde die Leistungsfähigkeit von LLMs bei der Erzeugung von Unit-Tests, Systemtest-Eingaben und Testorakeln evaluiert. LLMs profitieren dabei von ihrer Fähigkeit zum semantischen Codeverständnis und zur Generierung syntaktisch korrekter Strukturen (J. Wang et al., 2024).

Eine relevante Studie in diesem Bereich ist die Arbeit von T.-O. Li et al. (2023),

die die Technik des Differential Prompting einführt. Diese Methode nutzt gezielte Differenzierung zwischen mutierten und nicht mutierten Programmversionen, um fehlerverursachende Testfälle auf dem Quixbugs-Datensatz zu identifizieren. In der Evaluation zeigte sich, dass Differential Prompting auf dem QuixBugs-Datensatz eine Erfolgsrate von 75,0% beim Finden korrekter fehlerverursachender Testfälle erreichte. Ein einfacher Standard-Prompt, der lediglich eine generische Aufforderung zur Testfallerstellung enthielt, erreichte im Vergleich dazu mit ChatGPT lediglich eine Erfolgsrate von 28,8%. Das auf Python etablierte Testwerkzeug PYNGUIN erzielte in diesem Szenario lediglich 7,5%. Auch im Hinblick auf die Gesamtgenauigkeit bei der Analyse sowohl fehlerhafter als auch korrekter Programme zeigte Differential Prompting mit 94,6% eine deutlich höhere Präzision als die beste bisherige Baseline mit 14,3%. Die Methode ist jedoch auf kleinere Programme mit weniger als 100 Zeilen Code beschränkt.

Schäfer et al. (2024) untersuchen in ihrer Studie die automatisierte Generierung von Unit-Tests mithilfe von bestehenden LLMs ohne zusätzliches Training. In ihrer Studie stellen sie das LLM-Tool TESTPILOT zur Testfallgenerierung vor. Dieses basiert auf adaptivem Prompting und bezieht kontextuelle Informationen wie Quellcode, Dokumentation und Nutzungshinweise mit ein. Fehlgeschlagene Tests werden durch erneutes Prompting unter Einbeziehung der Fehlermeldung verbessert. Die Wahl des Prompt-Stils sowie die Relevanz der eingebundenen Code-Elemente wirken sich maßgeblich auf die Qualität der Tests aus. Im Vergleich zu Few-Shot-Ansätzen fokussiert sich das Modell TESTPILOT stärker auf die spezifische Ziel-Funktion. So erreicht TESTPILOT eine mediane Statement-Abdeckung von 70,2 % und eine Branch-Abdeckung von 52,8 % und übertrifft damit traditionelle Verfahren wie Nessie mit 51,3% bzw. 25,6% deutlich. Die Studie zeigt, dass LLMs effektive Unit-Tests erzeugen können, wobei insbesondere die Assertion-Generierung noch Verbesserungspotenzial bietet.

Automatisierte Programmreparatur

Die automatische Fehlerbehebung mittels LLMs ist ein weiteres relevantes Forschungsfeld mit mindestens 36 identifizierten Studien (J. Wang et al., 2024). Frühere Arbeiten haben bereits die Leistungsfähigkeit von LLMs umfassend untersucht.

So zeigte die Studie von Prenner et al. (2022), dass Codex auf QuixBugs in Java bis zu 14 von 40 Bugs erfolgreich beheben konnte. Im Vergleich dazu behoben andere Tools wie CoCoNut 13 und CURE 26 Bugs in Java. Dabei stellt Codex ein code-basiertes LLM dar, wohingegen es sich bei CURE explizit um ein vortrainiertes Sprachmodell für Java handelt und CoCoNut auf dem Paradigma der Neural Machine Translation basiert.

Sobania et al. (2023) evaluierten und analysierten die automatische Fehlerbehebungsleistung zwischen zwei Modellversionen von ChatGPT aus Dezember 2022 und Januar 2023 auf dem QuixBugs-Datensatz. Mit einer einfachen standardisierten Anfrage konnte ChatGPT 19 von 40 Bugs beheben. Damit lag das Modell auf Augenhöhe mit Codex und CoCoNut, übertraf jedoch herkömmliche APR-Methoden, die lediglich sieben Fehler korrigieren konnten. Durch den Einsatz von Follow-up-Anfragen mit konkreten Fehlerhinweisen steigerte sich die Erfolgsrate von ursprünglich 47,5% auf 77,5%.

Aus dieser und aktuelleren Studien wird auch erkennbar, dass ChatGPT mit geeignetem Prompting eine signifikante Leistungssteigerung erzielt. Eine vergleichbare Verbesserung konnte auch in der Studie von Wuisang et al. (2023) beobachtet werden. ChatGPT konnte hier zunächst 25 von 40 Fehlern korrekt beheben. Mithilfe zusätzlicher Nachfragen, die spezifische Kontextinformationen bereitstellten, erhöhte sich die Erfolgsrate auf 30 erfolgreiche Reparaturen.

Die Bedeutung von Kontextinformationen wird ebenfalls durch die Untersuchung von Qu et al. (2024) unterstrichen. Durch die Einbettung von Problembeschreibungen und Testfällen konnte die Erfolgsquote von 16,7% auf 83,3% gesteigert werden.

Ein signifikanter Fortschritt wurde von Hu et al. (2024) dokumentiert. Das dort eingesetzte Modell GPT-O1, eine speziell für reasoning-basierte Aufgaben optimierte Variante von ChatGPT mit integrierter Chain-of-Thought-Strategie, konnte sämtliche 40 Fehler im QuixBugs-Benchmark vollständig beheben und übertraf damit frühere Modellversionen deutlich.

Weitere Studien, unter anderem von Viet und Markov (2023), evaluierten den Einfluss verschiedener Prompting-Techniken und belegten deutliche Unterschiede in der Leistung zwischen Zero-Shot-Ansätzen und Few-Shot-Learning.

Die Studie von Yu et al. (2024) untersuchte umfassend die Selbstverifizierungsfähigkeit von LLMs wie ChatGPT bei Aufgaben der Programmreparatur. Dabei wurde das Modell aufgefordert, fehlerhaften Code zu reparieren und anschließend mittels verschiedener Prompting-Methoden zu beurteilen, ob der Fehler erfolgreich behoben wurde. Die Ergebnisse zeigten, dass ChatGPT bei direkter Nachfrage seine fehlgeschlagenen Reparaturversuche oft fälschlicherweise als erfolgreich bewertet. Während anleitende Fragen und Testreports die Erkennung weiterer fehlgeschlagener Reparaturen verbessern können, sind die Erklärungen hierfür meist unzutreffend und die Rate falsch-positiver Vorhersagen steigt. Zudem wurden widersprüchliche Halluzinationen beobachtet. Das Modell sieht zunächst eine Reparatur als erfolgreich an, widerruft dies jedoch bei der anschließenden Selbstverifizierung.

Diese Studien verdeutlichen, dass sich die Leistungsfähigkeit von LLMs im Bereich der automatisierten Programmreparatur zwischen 2022 und 2024 durch kontinuierliche Modellverbesserungen erheblich gesteigert hat. Zugleich weisen zahlreiche Arbeiten darauf hin, dass der Erfolg stark von der Gestaltung der Eingabeaufforderung abhängt. Insbesondere Strategien, die Kontextinformationen oder explizite Fehlerhinweise bereitstellen, führen in vielen Fällen zu signifikanten Leistungssteigerungen gegenüber generischen Prompts ohne zusätzlichen Input. Allerdings zeigt sich, dass die Fähigkeit zur verlässlichen Selbstverifizierung trotz verbesserter Reparaturleistung eingeschränkt ist und häufig zu Fehleinschätzungen oder widersprüchlichen Aussagen führt.

Fehlersuche, Debugging und statische Analyse

Auch in der Fehlersuche und statischen Analyse zeigen LLMs zunehmendes Potenzial.

Mohajer et al. (2024) führten eine Studie durch, um die Fähigkeiten von ChatGPT in statischen Analyseaufgaben zu untersuchen. Dies umfasste spezifisch

die statische Fehlererkennung und die Entfernung falsch positiver Warnungen. Für die Datensammlung nutzten sie das etablierte statische Analysetool Infer und verwendeten es auch als Baseline für Vergleiche. Das Ziel der Studie bestand darin, die Effektivität von ChatGPT bei der Durchführung dieser Analyseaufgaben zu evaluieren. Es wurde untersucht, ob ChatGPT die Leistung von statischen Fehlerdetektoren, wie Infer, verbessern kann. Die Studie zeigt, dass LLM-gestützte Ansätze bereits bestehende Baselines übertreffen oder verbessern können. Das Modell zeigte prinzipielle Anwendbarkeit, war aber auf Java und das spezifische Analysetool beschränkt.

Im Bereich der Sicherheitsanalyse in der Studie von Fu et al. (2023) zeigte sich allerdings auch, dass generische Modelle, wie GPT-4, gegenüber spezialisierten Lösungen wie GraphCodeBERT unterlegen sind. Getestet wurde auf den realen Datensätzen Big-Vul und CVEFixes, die über 190.000 C/C++-Funktionen enthalten. Beispielsweise erreichte GPT-4 bei der Vorhersage von Sicherheitslücken einen Recall von nur 27% gegenüber 83% bei GraphCodeBERT. Diese Ergebnisse betonen die Bedeutung domänenspezifischer Modellanpassungen durch Fine-Tuning.

Mutantenanalyse

Ein noch junges, aber wachsendes Forschungsfeld ist die Detektion äquivalenter Mutanten im Mutation Testing.

Eine umfassende Studie von Tian et al. (2024) zeigte, dass LLMs alle zehn Vergleichs-Baselines deutlich übertrafen. Dazu zählten unter anderem syntaktische und semantische Heuristiken, statische Analyseverfahren und klassische maschinelle Lernmodelle. Das mittels Feintuning angepasste Modell UniXCoder erzielte durchschnittliche F1-Scores von über 81% bei Inferenz-Zeiten unter 50 ms pro Mutantenpaar. Im Vergleich dazu schnitten prompt-basierte Ansätze im Vergleich deutlich schwächer ab.

Diese Ergebnisse zeigen, dass trainierte Code-LLMs nicht nur die Erkennungsgenauigkeit, sondern auch die Effizienz etablierter Mutanten-Analysetools übertreffen können.

3.3.2 Herausforderungen und offene Fragen

Trotz der vielversprechenden Ergebnisse von LLMs im Softwaretest bestehen weiterhin zentrale Herausforderungen beim Einsatz. In der Literaturübersicht von J. Wang et al. (2024) werden aktuelle Probleme durch den Einsatz von LLMs im Softwaretest beleuchtet.

Als grundlegendes Problem wird dort die Generierung vielfältiger und realistischer Testeingaben dargestellt. Während sich generative Modelle in der Codeerstellung auf die Produktion einzelner korrekter Fragmente konzentrieren, erfordert der Testprozess eine systematische Variation der Eingaben, um möglichst viele Programmzustände zu überprüfen.

Eine weitere zentrale Herausforderung umfasst das sogenannte Testorakelproblem, das die automatisierte Bewertung der Korrektheit von Testergebnissen betrifft. Gegenwärtige Ansätze beschränken sich häufig auf die Erkennung von

Laufzeitfehlern oder nutzen Differential Testing, bei dem mehrere Programmversionen miteinander verglichen werden. Diese Verfahren liefern jedoch lediglich begrenzte Aussagen über semantische Korrektheit und sind in ihrer Anwendbarkeit auf spezifische Testszenarien eingeschränkt. Die Integration alternativer Orakelmechanismen stellt bislang ein weitgehend unerforschtes Feld dar.

Zudem wird in der Studie die eingeschränkte Generalisierbarkeit der Ergebnisse erwähnt. Während LLMs auf synthetischen Benchmarks oftmals hohe Erfolgsraten erzielen, zeigt sich in realitätsnäheren Anwendungsszenarien ein deutlicher Leistungsabfall. Diese Abweichung wirft grundlegende Fragen hinsichtlich der tatsächlichen Praxistauglichkeit solcher Modelle auf.

Darüber hinaus gibt es einen Mangel an Studien, die den Einsatz von LLMs in frühen Phasen des Testprozesses systematisch untersuchen. Dieses Gebiet bleibt derzeit weitgehend unerforscht und wurde als eine Forschungsmöglichkeit identifiziert.

Als weitere Herausforderung gilt die eingeschränkte Selbstverifizierungsfähigkeit von LLMs, die in der Studie von Yu et al. (2024) als zentrales Problem hervorgehoben wird. Dies beeinträchtigt die Verlässlichkeit automatisierter Fehleranalysen und unterstreicht die Notwendigkeit robuster Evaluationsmethoden.

3.4 Einordnung der eigenen Arbeit

Die vorliegende Untersuchung knüpft an den aktuellen Forschungsstand an, indem sie zwei LLM-Modelle, ChatGPT mit der Modellversion GPT-4 und QC, hinsichtlich ihrer Fähigkeiten zur Fehlererkennung und -behebung evaluiert. Mithilfe der Java-Version des QuixBugs-Datensatzes und unter Verwendung unterschiedlicher Prompting-Strategien werden gezielt Unterschiede in der Leistungsfähigkeit, Effizienz und Robustheit der Modelle unter kontrollierten Bedingungen analysiert.

Der Fokus liegt auf der Frage, inwiefern der gewählte Prompt-Typ, der zugrunde liegende Modelltyp sowie der jeweilige Fehlertyp das Reparaturverhalten der Modelle beeinflussen. Ziel ist es, differenzierte Einblicke in die Effektivität und Grenzen aktueller LLMs bei zentralen Aufgaben der Fehlerdiagnose und -behebung zu gewinnen.

Aufbauend auf den im Forschungsstand dokumentierten Fortschritten und Ergebnissen früherer Studien zielt diese Arbeit darauf ab, die Leistungsfähigkeit von LLM-Modellen empirisch zu bewerten und die gewonnenen Erkenntnisse im Kontext des Einsatzes von LLMs im Softwaretest einzuordnen.

3.4.1 Motivation Fehlererkennung und Fehlerbehebung

Korrektheit, Sicherheit und Zuverlässigkeit zählen zu den Kernzielen der Softwareentwicklung. Eine zentrale Rolle spielt dabei der Softwaretest, insbesondere die Fehlererkennung und -behebung. Diese beiden Aufgabenbereiche tragen maßgeblich zur Identifikation und Korrektur fehlerhafter Komponenten bei. Ihre präzise Umsetzung ist entscheidend für die Auslieferung zuverlässiger und funktionsfähiger Softwaresysteme (Hoffmann, 2013).

Die Literaturübersicht von J. Wang et al. (2024) und die im vorherigen Abschnitt 3.3.1 behandelten Studien zeigen, dass im Bereich der automatisierten Programmreparatur sowie bei der Fehlersuche und dem Debugging vermehrt LLM-basierte Ansätze eingesetzt und untersucht werden. Diese empirischen Ergebnisse weisen auf vielversprechende Potenziale hin, etwa bei der Lokalisierung fehlerhafter Codestellen oder der automatisierten Generierung von Reparaturvorschlägen.

Die gezielte Untersuchung der Fehlererkennung und Fehlerbehebung erlaubt daher nicht nur eine vertiefte Analyse zweier zentraler Aufgaben im Softwaretest, sondern knüpft auch direkt an aktuelle Entwicklungen und offene Fragestellungen im Forschungsstand an. Zudem verdeutlichen die Studien in Abschnitt 3.3.1 den Bedarf an weiterer empirischer Forschung hinsichtlich der Effektivität und Zuverlässigkeit von LLMs.

Im Rahmen dieser Arbeit soll daher ein Beitrag zur fundierten Einschätzung des tatsächlichen Potenzials von LLMs in der Fehlerdiagnose und -korrektur geleistet werden. Durch den Vergleich unterschiedlicher Modelltypen und Prompting-Strategien wird angestrebt, systematische Erkenntnisse über Stärken, Schwächen und Einsatzgrenzen aktueller LLMs im Softwaretest zu gewinnen.

3.4.2 Motivation LLM-Tools

Mit dem Fortschritt von LLMs eröffnen sich neue Möglichkeiten für die Automatisierung zentraler Aufgaben im Softwaretest, wie in der Fehlerdiagnose und Programmreparatur. LLMs zeigen zunehmend Potenzial in verschiedenen Bereichen des Softwaretests.

So demonstrierten Sobania et al. (2023), dass ChatGPT mit geeigneten Prompts auf dem QuixBugs-Benchmark eine signifikant höhere Erfolgsrate bei der automatisierten Fehlerbehebung erzielen konnte als etablierte APR-Tools.

Diese empirischen Fortschritte unterstreichen die Relevanz einer differenzierten Analyse von LLMs im Softwaretest. Vor diesem Hintergrund ist eine systematische Untersuchung unterschiedlicher Modelltypen von zentraler Bedeutung, um deren jeweilige Leistungsfähigkeit, Anwendungsgrenzen und potenziellen Beitrag zur automatisierten Fehlerdiagnose und -behebung im Softwaretest fundiert zu evaluieren. Hierfür wird das generische Modell ChatGPT in der Version GPT-4 mit dem speziell für Programmcode entwickelten Modell QC verglichen.

Motivation ChatGPT

ChatGPT ist ein auf der GPT-Architektur basierendes LLM. GPT steht für Generative Pre-trained Transformer und gehört zur Klasse der Transformer-Modelle, die in Abschnitt 2.3.1 bereits erklärt wurde. ChatGPT wurde von OpenAI entwickelt.

Durch ein zweistufiges Training, das aus unbeaufsichtigtem Vortraining auf großen Textkorpora und anschließendem Feintuning besteht, wird die Fähigkeit zur Sprachverarbeitung und -generierung kontinuierlich verbessert.

Es wurde ursprünglich für generische Konversationsaufgaben entwickelt, ist jedoch durch seine Trainingsdaten auch für zahlreiche Aufgaben in der Software-

entwicklung einsetzbar (Yenduri et al., 2024).

Darüber hinaus belegen zahlreiche empirische Studien, beispielhaft die Untersuchung von Sobania et al. (2023), seine Einsatzfähigkeit und zunehmende Leistungsfähigkeit in verschiedenen Bereichen des Softwaretests, insbesondere bei der Testfallgenerierung, Fehlerlokalisierung sowie automatisierten Programmreparaturen. Frühere Studien belegen die Einsatzfähigkeit von ChatGPT in verschiedenen Bereichen des Softwaretests, wie die Testfallgenerierung. So zeigte beispielsweise die Untersuchung von T.-O. Li et al. (2023) im Unterabschnitt 3.3.1, dass ChatGPT eine hohe Leistungsfähigkeit beim Auffinden korrekter, fehlerverursachender Testfälle zu erzielen.

Zudem belegen Benchmarks die kontinuierliche Verbesserung der Modellvarianten. Im HumanEval-Benchmark erzielte GPT-4-1106-PREVIEW einen pass@1-Wert von 85,73%, während die Vorgängerversion GPT-3.5-Turbo-0301 72,19% erreichte. Diese Unterschiede spiegeln die Fortschritte in der Modellarchitektur und im Training wider und machen ChatGPT zu einem geeigneten Referenzmodell für generische LLMs in dieser Studie (D. Li & Murr, 2024).

Motivation QC

Die im folgenden Abschnitt aufgeführten Informationen basieren auf der Literatur von Hui et al. (2024). QC ist ein Sprachmodell, das auf die Bearbeitung von Code spezialisiert wurde und auf einem umfangreichen Datensatz von über 5,5 Billionen Tokens vortrainiert wurde. Dieser umfasst verschiedenste Datentypen, darunter Quellcode, mathematische Inhalte und synthetische Daten. Die Trainingspipeline kombiniert mehrere Schritte, darunter dateibasierte und repository-basierte Pretraining-Verfahren, Instruction Tuning sowie Methoden wie Fill-in-the-Middle und Direct Preference Optimization unter Nutzung von Ausführungsfeedback. Ziel war es, die Modellleistung bei anspruchsvollen Aufgaben der Codebearbeitung und Fehlerbehebung zu steigern.

In Benchmarks zeigt das Modell eine hohe Leistungsfähigkeit. Auf dem HumanEval-Datensatz erreichte QC einen pass@1-Wert von 92,7% und übertrifft damit das Modell GPT-4-1106-PREVIEW von 85,73% in dieser Metrik. Auch auf anderen Benchmarks, wie dem MBPP mit 90,2% und Aider Code Editing mit 60,9%, erzielt das Modell stabile Ergebnisse. Besonders hervorzuheben ist, dass bekannte Benchmarks explizit aus dem Trainingskorpus ausgeschlossen wurden.

Die Wahl von QC als Vergleichsmodell beruht auf seiner transparenten Modellarchitektur, der freien Verfügbarkeit und seiner hohen Leistungsfähigkeit bei codezentrierten Aufgaben. Die Gegenüberstellung mit einem proprietären, universell trainierten Modell wie ChatGPT erlaubt eine differenzierte Bewertung der Modelltypen im Hinblick auf ihre Eignung zur Unterstützung der Fehlerdiagnose und -korrektur im Softwaretest.

3.5 Fazit

Dieses Kapitel hat den aktuellen Forschungsstand im Bereich LLMs im Software Engineering und speziell im Softwaretest dargelegt. Es zeigte sich ein signifikantes Wachstum des Feldes, wobei insbesondere die Testfallgenerierung und

die automatische Programmreparatur als zentrale Anwendungsbereiche hervorstechen. Die Forschung belegt das zunehmende Potenzial von LLMs zur Unterstützung dieser Aufgaben, weist aber auch auf bestehende Herausforderungen und den maßgeblichen Einfluss von Prompting-Strategien auf die Modellleistung hin. Die Literatur zeigt, dass die Leistungsfähigkeit von LLMs in diesen Aufgaben stetig zunimmt, wobei neuere Modellversionen verbesserte Ergebnisse erzielen können und Prompting-Strategien einen maßgeblichen Einfluss auf den Erfolg haben.

Die Erkenntnisse dieses Kapitels begründen somit die methodische Ausrichtung der vorliegenden Arbeit. Sie motivieren insbesondere die Evaluation aktueller Modelle auf dem etablierten QuixBugs-Datensatz und die gezielte Untersuchung unterschiedlicher Prompting-Strategien, um deren aktuelle Fähigkeiten bei der Fehlererkennung und -behebung empirisch zu bewerten.

Kapitel 4

Methodik und Vorgehensweise

Dieses Kapitel beschreibt die Durchführung der Untersuchung, für die die Datengrundlagen, Methoden und Evaluationskriterien erläutert werden. Ziel ist es, den experimentellen Aufbau sowie die Erhebungs- und Auswertungsmethoden transparent und nachvollziehbar darzustellen, um die im weiteren Verlauf präsentierten Ergebnisse im Kontext einordnen zu können.

4.1 Forschungsdesign und Vorgehen

Diese Arbeit untersucht die folgenden zentralen Forschungsfragen:

1. Wie leistungsfähig sind LLMs in der Testaktivität der Fehlererkennung?
2. Wie leistungsfähig sind LLMs in der Testaktivität der Fehlerbehebung?
3. Wie unterscheidet sich die Leistungsfähigkeit von ChatGPT und QC in diesen Aufgaben?
4. Welchen Einfluss haben unterschiedliche Prompting-Strategien auf die Leistung der LLMs bei Fehlererkennung und Fehlerbehebung?

Ziel dieser Forschungsfragen ist ein systematischer Vergleich der Effektivität und Effizienz verschiedener LLMs. Zusätzlich soll untersucht werden, inwiefern unterschiedliche Arten der Prompt-Formulierung deren Leistungsfähigkeit beeinflussen. Auf dieser Grundlage lassen sich fundierte Aussagen über das Potenzial von LLMs zur Unterstützung im Softwareentwicklungsprozess ableiten. Diese Fragen knüpfen direkt an die in Kapitel 1 beschriebene Problemstellung und die in Kapitel 3 aufgezeigten offenen Fragen des Forschungsstandes an.

Zur Beantwortung dieser Fragen wurde ein experimentelles Forschungsdesign gewählt. Dieses ermöglicht eine gezielte Untersuchung der Leistungsfähigkeit von Sprachmodellen unter kontrollierten Bedingungen. Die Modelle ChatGPT in der Version GPT-4 und QC erhalten standardisierte Prompts zur Problemlösung,

sodass ein objektiver Vergleich möglich ist. Die Untersuchung basiert auf einem deduktiven Mixed-Methods-Ansatz, bei dem quantitative Metriken erhoben und durch qualitative Beobachtungen ergänzt werden (Hunziker & Blankenagel, 2024).

Für diese Untersuchung wurden drei verschiedene Prompts entwickelt. Die Datenerhebung für Prompt 1 und 2 erfolgte vom 24.02. bis 28.02.2025, und die Daten für Prompt 3 wurden vom 24.03. bis 25.03.2025 erfasst.

4.2 Datengrundlage: QuixBugs

Die Datengrundlage dieser Untersuchung ist der QuixBugs-Datensatz. Dabei handelt es sich um einen Testdatensatz, der aus 40 fehlerhaften Programmen besteht, die in Java und Python implementiert sind. Ursprünglich wurde der Datensatz im Rahmen der Plattform Quixey entwickelt. Die Programme wurden gezielt mit verschiedenen Fehlertypen versehen, um Debugging-Aufgaben zu simulieren (Lin et al., 2017). Da es sich um kompakte Einzelprogramme handelt, beinhalten die Fehler lokale Probleme innerhalb einer Methode oder Klasse. Integrationstests oder Auswirkungen auf größere Programmkontexte lassen sich mit diesem Datensatz nicht untersuchen. Aufgrund von Zeit- und Platzbeschränkungen wird in dieser Arbeit ausschließlich die Java-Version betrachtet.

Die Auswahl des QuixBugs-Datensatzes begründet sich, wie bereits im Forschungsstand in Abschnitt 3.3.1 dargestellt, durch seine Etablierung als Benchmark in relevanten Studien zur automatisierten Programmreparatur und Testfallgenerierung. Der Datensatz bietet hierfür eine geeignete Grundlage, da die Programme kurz, überschaubar und strukturiert sind. Zudem ermöglichen die standardisierten Testfälle eine objektive Erfolgsmessung (Lin et al., 2017).

Der Datensatz umfasst verschiedene Fehlersymptome, die basierend auf der Studie von Ye et al. (2021) klassifiziert wurden. Die nachfolgende Tabelle 4.1 zeigt die Verteilung dieser Fehlersymptome innerhalb des Datensatzes.

Failure Symptom	**Häufigkeit**
array index error	5
concurrent modification	1
incorrect output	26
null pointer	2
stack overflow	5
timeout/infinite loop	3

Tabelle 4.1: Verteilung der Failure Symptoms im Datensatz

4.3 Auswahl der LLM-Tools

Basierend auf den in Abschnitt 3.4.2 dargestellten Kriterien und Motivationen zur Untersuchung verschiedener Modelltypen wurden für diese Untersuchung die Modelle ChatGPT in der Version GPT-4 von OpenAI und QC von Alibaba ausgewählt.

ChatGPT ist ein kommerzielles Sprachmodell mit hoher Leistungsfähigkeit, das eine Vielzahl von Aufgaben in akademischen und industriellen Bereichen bewältigen kann (Bahrini et al., 2023). QC ist ein Open-Source-Modell, das für Programmieraufgaben optimiert wurde. Wie in Abschnitt 3.4.2 erläutert, erzielte es kompetitive Ergebnisse in Benchmarks wie HumanEval oder MMLU (Hui et al., 2024).

Für die Durchführung wurden Standardantworten der Modelle über ihre jeweilige Weboberfläche verwendet. Es erfolgten keine Modifikationen der Modellparameter, keine API-Nutzung und kein Fine-Tuning.

4.4 Prompt-Design/-Engineering

Ein zentraler Bestandteil der Untersuchung ist das gezielte Prompt-Engineering. Wie bereits in den theoretischen Grundlagen in Abschnitt 2.4 erläutert, bezeichnet dieser Begriff die gezielte Gestaltung von Eingabeaufforderungen zur Steuerung von LLMs. Zur Gewährleistung konsistenter und vergleichbarer Ausgaben wurden im Rahmen dieser Arbeit drei standardisierte Prompts entwickelt (Campesato, 2024).

Aus dem aktuellen Forschungsstand in Abschnitt 3.3.1 ergab sich, dass unterschiedliche Prompts signifikanten Einfluss auf die Leistung der Modelle haben. Dieser Einfluss wird in der vorliegenden Untersuchung weiterhin untersucht.

Prompt 1

Prompt 1 ist ein Zero-Shot-Prompt und prüft, ob das Modell ohne zusätzliche Hinweise in der Lage ist, Fehler zu erkennen und zu beheben. Hiermit sollte ein typischer Anwendungskontext mit minimaler Nutzereingabe simuliert werden.

> "Identifiziere bitte den Fehlertyp im Java-Codeabschnitt und korrigiere den Code entsprechend: [fehlerhafter Codeabschnitt]"

Prompt 2

Prompt 2 ist ein Chain-Of-Thought-Prompt und dient dazu, den Lösungsweg des Modells transparent darzustellen. Dadurch wird eine detailliertere Analyse der Modelllogik ermöglicht. Der gesamte Prompt wird dabei in einer einzigen Anfrage übermittelt.

> "Hier ist ein fehlerhafter Java-Codeabschnitt: [fehlerhafter Codeabschnitt]
> Bitte erkläre Schritt für Schritt, wie du bei der Fehlersuche und -behebung vorgehst, ohne den Code sofort zu korrigieren:
>
> 1. Beschreibe zunächst die vermutete Funktionalität.
> 2. Gehe Zeile für Zeile vor und erkläre, ob jede Zeile zur erwarteten Funktionalität beiträgt.
> 3. Identifiziere alle Fehler und erkläre deren Auswirkungen.

4. Präsentiere eine finale korrigierte Version des Codes und erläutere, warum diese korrekt ist."

Prompt 3

Prompt 3 ist ein Zero-Shot-Prompt, mit dem überprüft wird, ob die Modelle fälschlicherweise Fehler erkennen und identifizieren.

> "Analysiere den folgenden Code und bewerte, ob ein Fehler enthalten ist. Wenn ja, zeige ihn und klassifiziere ihn. Falls der Code korrekt ist, erkläre warum.[korrekter Codeabschnitt]"

4.5 Evaluationskriterien

Zur systematischen Bewertung der Leistungsfähigkeit der LLMs in den Aufgabenbereichen Fehlererkennung, Fehlerbehebung und Effizienz wurden verschiedene Evaluationskriterien definiert. Zusätzlich erfolgt eine Darstellung der erkannten und übersehenen Fehlerarten.

4.5.1 Fehlererkennung

Die Fähigkeit der Modelle, fehlerhafte Codeabschnitte zu identifizieren, wird anhand gängiger Metriken aus dem Bereich des Information Retrievals bewertet. Die Grundlagen für die Metriken der Fehlererkennung basieren auf der Literatur von Brito et al. (2020).

Die Analyse der Modellausgaben erfolgt durch eine Klassifizierung der fehlerhaften Codeabschnitte in die Kategorien True Positives (TPs), False Positives (FPs), False Negatives (FNs) und Schwachstellen. TPs sind korrekt identifizierte Fehlerstellen. FPs sind fehlerhaft markierte Stellen, die tatsächlich korrekt sind. FNs sind Fehler, die vom Modell nicht erkannt wurden.

Zusätzlich wurde die Kategorie Schwachstellen eingeführt, um potenziell problematische oder stilistisch verbesserungsfähige Stellen zu kennzeichnen. Diese Kategorie erlaubt eine Unterscheidung zwischen echten Falschmeldungen und konstruktiven Hinweisen. Als Schwachstellen gelten Hinweise auf camelCase-Verstöße, redundanten Code, Stellen, die als unnötig oder ineffizient bezeichnet werden, fehlende Typüberprüfungen oder Generics sowie zusätzliche Prüfungen auf Exceptions oder unplausible Zustände.

Auf Grundlage dieser Kategorisierung werden die folgenden Metriken für Prompt 1 und Prompt 2 berechnet.

Precision

Die Precision ist der Anteil der vom Modell als fehlerhaft markierten Stellen, die tatsächlich Fehler sind.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall

Der Recall ist der Anteil der tatsächlichen Fehlerstellen, die vom Modell erkannt wurden.

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1-Score

Der F1-Score ist das harmonische Mittel aus Precision und Recall.

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Anders als bei den Prompts 1 und 2, die sich auf fehlerhafte Programme konzentrieren, dient Prompt 3 der Untersuchung der Fehlersensitivität der Modelle bei funktionierendem Code. Dabei werden ausschließlich die Fälle erfasst, die von den Modellen als korrekt oder nicht korrekt klassifiziert werden, sowie die darauf basierenden FPs und identifizierten Schwachstellen.

4.5.2 Fehlerbehebung

Zur Bewertung der tatsächlichen Korrekturleistung der Modelle werden die Metriken Lines of Code (LOC), die Erfolgsrate der JUnit-Tests und die Zyklomatische Komplexität (CC) erfasst.

LOC

Dies beschreibt die Anzahl der Codezeilen in der generierten Lösung, wobei Kommentare nicht berücksichtigt wurden. Zeilen, die ausschließlich aus Klammern bestehen, wurden mitgezählt (Radjenović et al., 2013).

Erfolgsrate

In der Statistik stellt die relative Häufigkeit das Verhältnis der Anzahl von Vorkommen eines bestimmten Ereignisses zur Gesamtzahl der Beobachtungen dar. Dadurch können Verteilungen eines bestimmten Merkmals beschrieben und näher analysiert werden (Sibbertsen & Lehne, 2021).

Hierbei wird dieses statistische Prinzip im Kontext der Programmreparatur angewendet. Die Erfolgsrate ist das Verhältnis der bestandenen Unit-Tests zur Gesamtzahl der Testfälle, um die Leistung der Modelle zu evaluieren.

$$\text{Erfolgsrate} = \frac{\text{Bestandene Tests}}{\text{Gesamte Testfälle}}$$

CC

Die CC ist das Maß für die strukturelle Komplexität des Codes basierend auf der Anzahl der Entscheidungsstrukturen (Vytovtov & Markov, 2017).

$$M = E - N + 2P$$

Dabei bezeichnet die Variable M die resultierende CC. Die Variable E steht für die Anzahl der Kanten im Kontrollflussgraphen, die die möglichen Übergänge zwischen Anweisungen darstellen. N bezeichnet die Anzahl der Anweisungen oder Blöcke von Anweisungen innerhalb des Codes. P steht für die Anzahl der zusammenhängenden Komponenten im Kontrollflussgraphen (Vytovtov & Markov, 2017).

4.5.3 Effizienz

Ein weiteres Kriterium ist die Dauer der Antwortgenerierung, die in Sekunden gemessen wird. Diese Metrik liefert Informationen über die praktische Einsetzbarkeit der Modelle im Entwicklungsalltag. Die Antwortzeiten wurden manuell im Webbrowser unter den Entwicklertools ermittelt, indem jeweils die Dauer der entsprechenden Netzwerkanfragen abgelesen wurde. Dabei wurden Hintergrundanfragen, wie Authentifizierungs- oder Trackingprozesse, nicht berücksichtigt.

ChatGPT

Bei ChatGPT wurde die Zeitmessung anhand der `conversation`-Netzwerkanfrage durchgeführt, die den Prompt-Antwort-Zyklus repräsentiert.

QC

Bei QC wurde die Dauer der `completions`-Netzwerkanfrage verwendet, die den eigentlichen Antwortprozess abbildet.

4.6 Durchführung und Datenerhebung

Die drei entwickelten Prompts wurden jeweils an beide LLMs übermittelt. Für jedes Prompt wurden 40 Anfragen pro Modell gestellt, sodass alle 40 Programme aus dem QuixBugs-Datensatz abgedeckt wurden. Insgesamt hat ein Modell 120 Anfragen verarbeitet. Jede Anfrage wurde in einem neuen Chat gestartet, um Kontextbeeinflussung zu vermeiden. Die offiziellen Webinterfaces von ChatGPT (https://chatgpt.com/) und QC (https://chat.qwen.ai/) wurden verwendet, um die Antworten zu generieren.

Die Prompts wurden manuell über die Weboberflächen eingegeben. Dabei wurden die Antwortzeiten durch die Dauer der Netzwerkanfragen gemessen. Die generierten Antworten der LLMs folgten einem ähnlichen Muster. Zunächst erfolgte eine Analyse des Codes mit Erläuterungen zu potenziellen Fehlern. Anschließend wurde eine überarbeitete Version des Codes präsentiert, begleitet von Kommentaren oder Begründungen der Änderungen.

Ein exemplarischer Ausschnitt einer generierten Antwort ist in Abbildung 4.1 dargestellt. Dieses Beispiel basiert auf dem Prompt 1 zum Programm GCD und veranschaulicht die systematische Herangehensweise von ChatGPT an die gestellte Aufgabe.

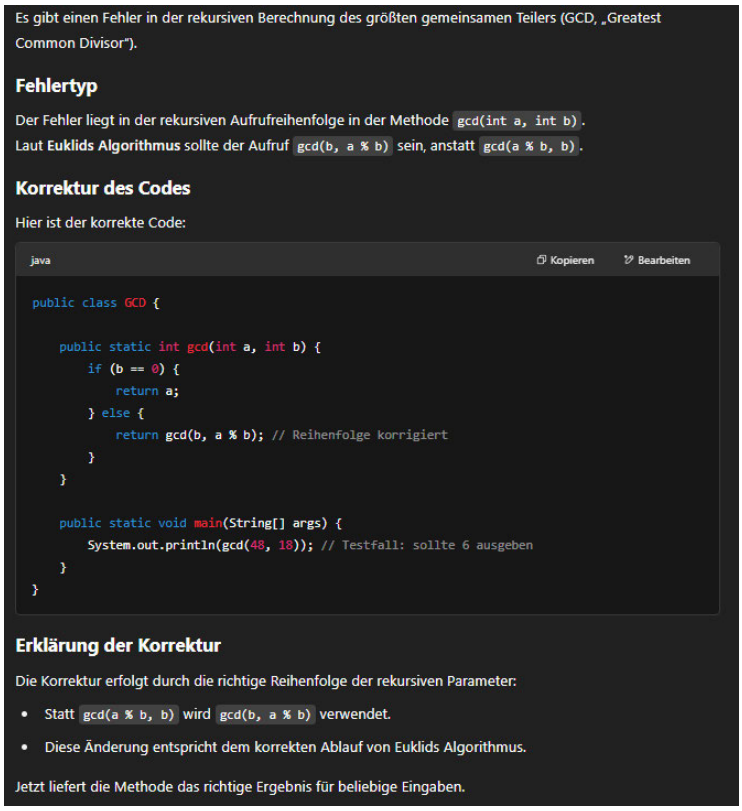

Abbildung 4.1: Ausgabe von ChatGPT für Prompt 1 beim Programm GCD
Quelle: Eigene Darstellung basierend auf einer Antwort von ChatGPT

Die Ergebnisse wurden anschließend in Excel dokumentiert. Für jede Anfrage wurden relevante Kennzahlen wie TPs, FPs, FNs sowie erkannte Schwachstellen festgehalten, um die Metriken der Fehlererkennung zu berechnen. Zur Validierung der generierten Lösungen wurden die bereitgestellten JUnit-Testfälle aus QuixBugs verwendet. Auf Basis dieser Tests wurden Erfolgsraten berechnet. Anschließend wurden Metriken, wie LOC-Differenz und CC, erhoben.

4.7 Datenanalyse & Auswertung

Um die Vergleichbarkeit zwischen den Modellen und Prompt-Typen zu gewährleisten, wurden alle erhobenen Werte tabellarisch aufbereitet. Diese strukturierte Erfassung bildet die Grundlage für die quantitative und qualitative Analyse der Modellleistung im Hinblick auf Fehlererkennung, Fehlerbehebung und Effizienz.

4.7.1 Fehlererkennung und Fehlerbehebung

Die zentralen Evaluationskriterien wurden getrennt nach Modell und Prompt-Strategie analysiert. Ziel war es, sowohl die Gesamtleistung der beiden LLMs als auch die Einflüsse der eingesetzten Prompting-Strategien zu erfassen. Neben den

aggregierten Metriken wurden auch deren Verteilungen auf Programmebene untersucht, um Rückschlüsse auf die Konsistenz und Stabilität der Modellleistung zu gewinnen.

Zusätzlich wurde jedes Programm den entsprechenden Fehlersymptomen aus dem Quixbugs-Datensatz von Ye et al. (2021) zugeordnet. Dadurch konnte untersucht werden, ob die Modelle bei der Verarbeitung von bestimmten Fehlertypen Schwächen aufzeigen. Diese Betrachtung sollte Aufschluss über potenzielle Grenzen in der Fehlerverarbeitung geben. Zusätzlich zur Untersuchung im Bereich der Fehlererkennung und -behebung erfolgte eine Analyse der Effizienz der Modelle. Hierzu wurde die durchschnittliche Bearbeitungsdauer pro Prompt gemessen und verglichen. Diese quantitative Auswertung wurde durch textuelle Beobachtungen ergänzt, um qualitative Unterschiede und auffällige Muster zu erkennen.

4.7.2 Vertiefende Analysen

Ergänzend wurden vertiefende Auswertungen durchgeführt, um ein umfassenderes Verständnis der Leistungsfähigkeit der Modelle zu ermöglichen.

Ein Schwerpunkt lag auf dem Zusammenhang zwischen den strukturellen Eigenschaften des Codes und der Erfolgsrate. Es wurde untersucht, inwiefern Merkmale, wie LOC und CC die Erfolgsrate beeinflussen. Diese Analyse zielte darauf ab, potenzielle Zusammenhänge zwischen Codekomplexität und Leistungsfähigkeit der Modelle bei der Reparatur zu identifizieren.

Darüber hinaus wurden die strukturellen Eigenschaften der generierten Lösungen untersucht. Hierzu wurden die Metriken LOC und CC der korrigierten Programme mit den entsprechenden Werten der Originalversionen verglichen. Somit sollte festgestellt werden, ob bestimmte Modelle oder Prompts zu komplexeren Lösungen führen.

4.8 Gütekriterien

Die im folgenden Abschnitt verwendeten Definitionen der zentralen Gütekriterien Objektivität, Reliabilität und Validität basieren auf dem Werk von Krebs und Menold (2019).

Objektivität

Objektivität steht für die Unabhängigkeit der Ergebnisse von der durchführenden Person. Sie liegt vor, wenn die Resultate nicht durch subjektive Einflüsse der Untersuchenden verzerrt werden. In der vorliegenden Studie wurde Objektivität durch ein standardisiertes Vorgehen gewährleistet, denn beide Modelle erhielten identische Prompts in jeweils neuen Chats, um kontextuelle Verzerrungen zu vermeiden. Die Auswertung erfolgte auf Basis klar definierter Evaluationskriterien, wodurch eine nachvollziehbare Bewertung sichergestellt wurde.

Validität

Die Validität der Untersuchung ist gegeben, da die eingesetzten Metriken etablierte Kennzahlen zur Bewertung der Reparaturleistung darstellen. Die interne Validität wird durch das kontrollierte experimentelle Design gewährleistet, bei dem gezielt verschiedene Modelltypen sowie die Prompting-Strategien eingesetzt wurden. Die externe Validität ist durch die Verwendung des etablierten QuixBugs-Benchmarks gegeben, der eine Vergleichbarkeit der Ergebnisse mit anderen Studien ermöglicht.

Reliabilität

Die Reliabilität bezeichnet die Zuverlässigkeit und Reproduzierbarkeit der Ergebnisse unter konstanten Bedingungen. Dies wurde durch die umfassende und transparente Dokumentation der Analyse- und Durchführungsschritte gewährleistet. Das Experiment ist unter gleichen Rahmenbedingungen prinzipiell reproduzierbar.

4.9 Einschränkungen

Trotz Einhaltung der Gütekriterien unterliegen die Ergebnisse dieser Untersuchung bestimmten Einschränkungen, die bei der Interpretation berücksichtigt werden sollten. Der verwendete Datensatz QuixBugs enthält synthetisch erzeugte Fehler in vergleichsweise kleinen Programmen und bildet somit nicht die Komplexität realer Softwareprojekte ab.

Zudem sind im Datensatz ausschließlich lokale Fehler in einzelnen Methoden oder Klassen enthalten. Das bedeutet, dass die Bewertung der Fehlerbehebung ausschließlich anhand von Unit-Tests erfolgt. Auswirkungen einer Korrektur auf andere Programmteile oder potenzielle neue Fehler außerhalb der unmittelbaren Fehlerstelle können nicht überprüft werden. In realen Projekten könnten fehlerhafte Fixes beispielsweise an anderer Stelle neue Fehler verursachen. Aufgrund dieser Einschränkungen ist die Aussagekraft der Ergebnisse in Bezug auf ihre Anwendbarkeit in realen Softwareentwicklungsprozessen begrenzt.

Zudem wurden die Sprachmodelle über öffentlich zugängliche Webinterfaces getestet. Dabei können externe Faktoren, wie Netzwerklatenzen oder temporäre Serverlasten, die Antwortzeiten beeinflussen. Auch die schnelle Weiterentwicklung großer Sprachmodelle stellt eine Einschränkung dar. Zukünftige Modellversionen könnten signifikant abweichende Ergebnisse erzielen, wodurch sich die langfristige Vergleichbarkeit der Resultate einschränkt.

Kapitel 5

Ergebnisdarstellung

In diesem Kapitel werden die zentralen Ergebnisse der experimentellen Untersuchung zur Leistungsfähigkeit von den LLMs in den Testaktivitäten Fehlererkennung und Fehlerbehebung präsentiert. Im Fokus stehen die beiden Modelle ChatGPT und QC, die in insgesamt 240 Anfragen mit drei verschiedenen Prompts auf ihre Leistungsfähigkeit getestet wurden. Die Verteilung der Anfragen ist in Tabelle 5.1 dargestellt.

	ChatGPT	QC	**Summe**
Prompt 1	40	40	80
Prompt 2	40	40	80
Prompt 3	40	40	80
Summe	**120**	**120**	**240**

Tabelle 5.1: Anzahl der Anfragen pro Modell und Prompt-Variante

Ziel ist es, die Antworten der Modelle systematisch entlang der in der Methodik definierten Evaluationskriterien auszuwerten, Unterschiede zwischen Modellen und Prompt-Varianten herauszuarbeiten und so die Grundlage für die anschließende Diskussion zu schaffen. Die Ergebnisdarstellung gliedert sich in die Bewertungsdimensionen Fehlererkennung, Fehlerbehebung und Effizienz.

Fehlererkennung

Bei der Fehlererkennung erfolgte die Analyse der identifizierten Fehlerstellen anhand von TPs, FPs und FNs, ergänzt durch Precision, Recall und F1-Score. Des Weiteren wurde untersucht, ob die Modelle in korrekten Programmen FPs und Schwachstellen erkennen oder diese zuverlässig als fehlerfrei identifizieren.

Fehlerbehebung

Im Bereich der Fehlerbehebung wurde der generierte Code nach funktionaler Korrektheit über die Erfolgsrate der JUnit-Tests sowie Betrachtung von LOC-Differenz und CC ausgewertet.

Effizienz

Die Effizienz der Modelle wurde anhand der durchschnittlichen Bearbeitungsdauer pro Anfrage bewertet. Diese Kennzahl erlaubt Rückschlüsse darauf, wie zeitintensiv bestimmte Prompt-Strategien im Vergleich zueinander sind.

5.1 Fehlererkennung

Zur Untersuchung der Fehlererkennungsleistung der LLMs wurden die Metriken Precision, Recall und F1-Score verwendet. Grundlage dieser Bewertung waren die erfassten TPs, FPs und FNs je Programm und Prompt. Zusätzlich wurde die Kategorie Schwachstellen eingeführt, um nicht-kritische, aber potenziell problematische Codeabschnitte zu erfassen, die von den Modellen als fehleranfällig erkannt wurden.

5.1.1 Übersicht über erkannte und nicht erkannte Fehler

	TP	FP	FN	Schwachstellen
ChatGPT Prompt 1	39	33	1	38
ChatGPT Prompt 2	40	37	0	65
QC Prompt 1	33	24	7	42
QC Prompt 2	34	26	6	51

Tabelle 5.2: Fehlererkennung nach Modell und Prompt: TP, FP, FN und erkannte Schwachstellen

Die Auswertung der Tabelle 5.2 zeigt differenzierte Werte zwischen den Modellen und Prompt-Varianten. ChatGPT erzielte in beiden Prompts eine höhere Anzahl an TPs als QC. Insbesondere mit Prompt 2 wurden alle 40 Fehler korrekt identifiziert, während bei Prompt 1 39 von 40 Fehlern erkannt wurden. QC hingegen identifizierte 33 Fehler in Prompt 1 und 34 Fehler in Prompt 2 korrekt, was gleichzeitig zu einer höheren Anzahl an FNs führte. Somit übersah QC mehr Fehler als ChatGPT, das in dieser Hinsicht eine nahezu vollständige Fehlererkennung zeigte.

Beide Modelle produzierten darüber hinaus FPs, wobei ChatGPT in beiden Prompts jeweils höhere Werte aufwies als QC. Ein ähnliches Bild zeigt sich bei der Anzahl der als potenziell problematisch klassifizierten Schwachstellen. ChatGPT markierte in Prompt 2 insgesamt 65 potenzielle Schwachstellen und markierte in Prompt 1 38 Stellen. Auch bei QC ist eine Zunahme bei Prompt 2 erkennbar.

Insgesamt lässt sich feststellen, dass beide Modelle bei Verwendung von Prompt 2 eine größere Anzahl an FPs sowie markierten Schwachstellen generierten. Dieser Effekt war bei ChatGPT besonders ausgeprägt. Die Ergebnisse deuten darauf hin, dass ChatGPT unabhängig von der Prompt-Variante eine umfassendere Fehlererkennung ermöglicht, dabei jedoch tendenziell auch eine höhere Anzahl an falsch-positiven Klassifikationen erzeugt.

Eine detaillierte Übersicht der Werte liefert Tabelle 5.2.

5.1.2 Vergleich quantitativer Erkennungsmetriken

	Precision	Recall	F1-Score
ChatGPT Prompt 1	54 %	98 %	70 %
ChatGPT Prompt 2	52 %	100 %	68 %
QC Prompt 1	58 %	83 %	68 %
QC Prompt 2	57 %	85 %	68 %

Tabelle 5.3: Precision, Recall und F1-Score nach Modell und Prompt

Die Auswertung der Performance-Metriken in der Tabelle 5.3 liefert weitere Erkenntnisse zur Leistungsfähigkeit der getesteten LLMs in der Fehlererkennung und inwiefern sich unterschiedliche Prompt-Strategien auf die Ergebnisse auswirken.

In beiden Prompts erzielte ChatGPT eine höhere Trefferquote in der Fehlererkennung, indem es einen Recall von 98 % in Prompt 1 und 100 % in Prompt 2 erreichte und konnte somit nahezu alle vorliegenden Fehler identifizieren. QCs lag hier deutlich darunter, mit einem Recall von 83 % in Prompt 1 und 85 % in Prompt 2.

Ein umgekehrtes Bild zeigt sich jedoch bei der Precision. Hier konnte QC leicht bessere Werte erzielen verglichen mit ChatGPT. Dies zeigt, dass QC tendenziell weniger Falschmeldungen produziert, dafür aber auch weniger Fehler insgesamt erkennt.

Der Vergleich der F1-Scores zeigt eine insgesamt ähnliche Leistungsfähigkeit beider Modelle. ChatGPT erreichte unter Prompt 1 mit 70 % den höchsten Wert, während der Rest mit 68 % knapp darunter lag.

Ein auffälliger Zusammenhang zeigt sich bei der Verwendung von Prompt 2. Beide Modelle steigerten dort leicht ihren Recall. Gleichzeitig nahm jedoch die Precision leicht ab, was auf eine Zunahme von FPs hinweist. Die Wahl der Prompting-Strategie beeinflusst dabei beide Modelle in vergleichbarer Weise, insbesondere im Hinblick auf das Verhältnis von Recall und Precision. Während ChatGPT eine umfassendere Fehlererkennung ermöglicht, weist QC eine höhere Genauigkeit bei der Identifikation tatsächlicher Fehler auf.

5.1.3 Verteilung der Erkennungsmetriken auf Programmebene

Neben den aggregierten Durchschnittswerten wurden die Verteilungen von Precision, Recall und F1-Score auf Programmebene in den Abbildungen 5.1 bis 5.3 erfasst. Die Diagramme zeigen deutliche Unterschiede in der Verteilung der Metriken zwischen den Modellen und Prompt-Strategien.

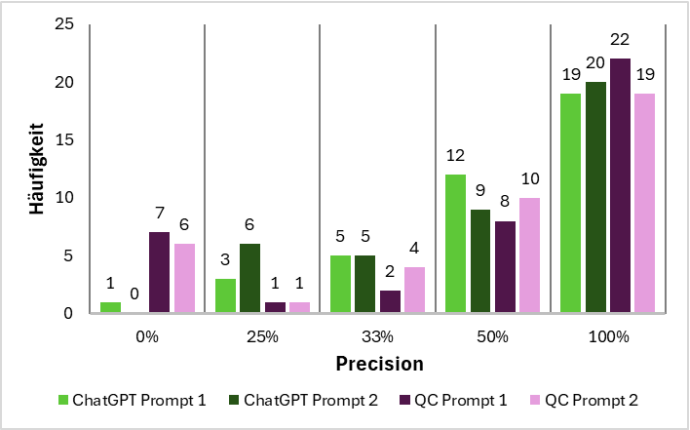

Abbildung 5.1: Precision-Verteilung
Quelle: Eigene Darstellung

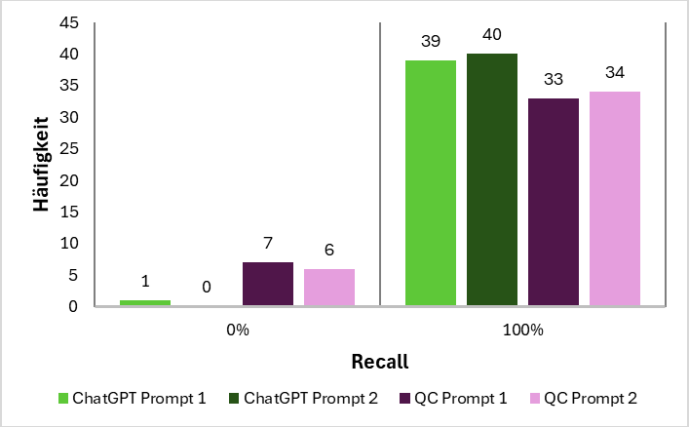

Abbildung 5.2: Recall-Verteilung
Quelle: Eigene Darstellung

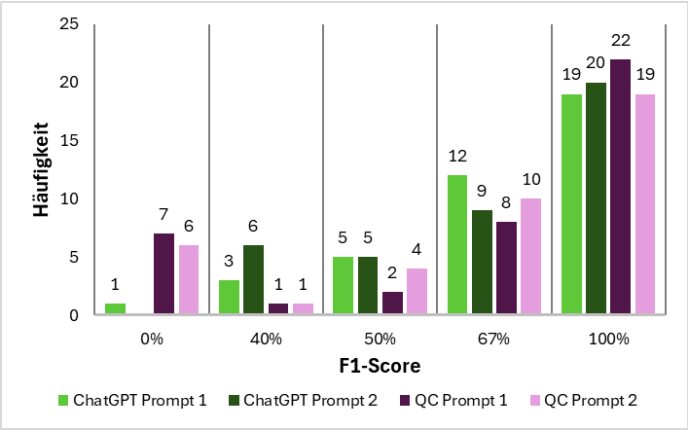

Abbildung 5.3: F1-Score-Verteilung
Quelle: Eigene Darstellung

Precision-Verteilung

Beide Modelle erreichten in der Precision den Maximalwert mit einer Häufigkeit zwischen 19 und 22 (vgl. Abbildung 5.1). Insbesondere ChatGPT zeigte in beiden Prompt-Varianten eine konsistente Precision, mit einem Großteil der Programme im oberen Leistungsbereich. QC erreichte ebenfalls mehrfach hohe Precision-Werte, zeigte jedoch insgesamt eine größere Streuung und weniger Programme mit konstant hoher Leistung.

Recall-Verteilung

Ein differenzierteres Bild zeigt sich beim Recall in Abbildung 5.2. Hier wurde deutlich, dass QC in beiden Prompts eine höhere Anzahl an Programmen mit einem Recall von 0% aufwies. Dies lässt sich auf die FNs zurückführen, wo das Modell die Fehler übersehen hatte. ChatGPT zeigte hingegen ein robusteres Verhalten. ChatGPT zeigte solche Ausfälle nur in einem Einzelfall im Prompt 1.

F1-Score-Verteilung

Auch beim F1-Score kehrte dieses Muster zurück (vgl. Abbildung 5.3). QC zeigte eine Häufung niedriger F1-Werte, was erneut auf Defizite bei der umfassenden Fehlererkennung hinweist. Solche Ausfälle traten bei ChatGPT nur in einem Einzelfall im Prompt 1 auf.

5.1.4 Zuordnung nicht erkannter Fehler

In der folgenden Abbildung 5.4 wurden die Programme mit FNs den Fehlersymptomen zugeordnet. Die Klassifizierung erfolgt nach der Studie Ye et al. (2021). Die Auswertung zeigt, dass sämtliche FN-Fälle ausschließlich die zwei Fehlertypen incorrect output sowie in einem Einzelfall stack overflow betreffen.

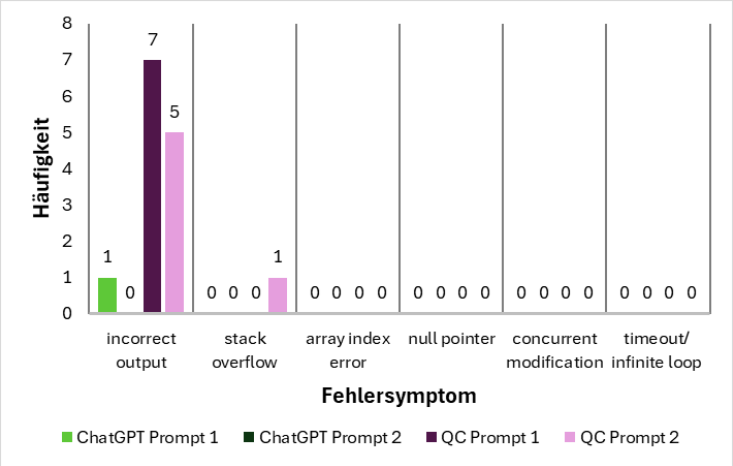

Abbildung 5.4: Häufigkeit von FNs nach Fehlersymptom
Quelle: Eigene Darstellung

Bei ChatGPT wurde unter Prompt 1 ein FN-Fall mit dem Symptom incorrect output festgestellt. Unter Prompt 2 wurde kein Fehler übersehen. QC zeigte dagegen eine deutlich höhere Fehleranfälligkeit. Unter Prompt 1 wurden sieben FN-Fälle mit dem Fehlersymptom incorrect output identifiziert. Auch unter Prompt 2 blieben Fehler unentdeckt, fünf davon waren incorrect output und zusätzlich ein Fall von stack overflow.

Die Ergebnisse zeigen, dass der Fehlertyp incorrect output für die Modelle schwer zu erkennen war. QC agierte in diesem Bereich systematisch weniger zuverlässig. ChatGPT hingegen erreicht eine nahezu vollständige Abdeckung und zeigt somit eine bessere Anpassung an die zugrunde liegenden Fehlermuster.

5.1.5 Fehlersensitivität bei korrekt funktionierendem Code

Die Analyse der Fehlersensitivität bei korrektem Code untersucht, inwieweit die Modelle zur Erkennung nicht existierender Fehler neigen. Die Tabelle 5.4 zeigt die Ergebnisse dieser Untersuchung.

Metrik	ChatGPT	QC
Als korrekt klassifiziert	7	16
Als nicht korrekt klassifiziert	33	24
Klassifizierte Schwachstellen	47	56
Klassifizierte FPs	36	24

Tabelle 5.4: Vergleich der Fehlersensitivität von ChatGPT und QC

ChatGPT klassifizierte in 33 von 40 Fällen funktionsfähigen Code fälschlicherweise als fehlerhaft und erzeugte dabei 36 FPs. QC war in dieser Hinsicht zurückhaltender und erzeugte lediglich 24 FPs, identifizierte jedoch gleichzeitig eine höhere Anzahl potenzieller Schwachstellen.

Beide Modelle zeigten eine gewisse Tendenz zur Übererkennung, jedoch mit unterschiedlichen Ausprägungen. ChatGPT klassifizierte häufiger FPs, während QC mehr potenzielle Schwächen markierte.

5.2 Fehlerbehebung

Zur Untersuchung der Fehlerbehebungsleistung wurde die Erfolgsrate und die strukturelle Komplexität des generierten Codes erfasst und analysiert. Die entsprechenden Ergebnisse finden sich in den nachfolgenden Abschnitten.

5.2.1 Erfolgsrate der Fehlerbehebung

Die Erfolgsrate der einzelnen Varianten ist in der Tabelle 5.5 dargestellt.

Variante	**Bestanden**	**Nicht bestanden**	**Erfolgsrate**
Quixbugs Originalcode	72	187	28 %
ChatGPT Prompt 1	240	19	93 %
ChatGPT Prompt 2	221	38	85 %
QC Prompt 1	205	54	79 %
QC Prompt 2	207	56	80 %

Tabelle 5.5: Vergleich der Fehlerbehebungsleistung nach Modell und Prompt

Die Ergebnisse zeigen erkennbare Leistungsunterschiede zwischen den Modellen sowie im Vergleich zum fehlerhaften Ausgangscode. Während dieser lediglich in 28% der Fälle erfolgreich ausführbar war, erzielten alle LLM-Konfigurationen höhere Erfolgsraten.

Die höchste Erfolgsrate lieferte ChatGPT mit Prompt 1, das in 93% der Fälle eine erfolgreiche Fehlerbehebung erreichte. Etwas darunter lag ChatGPT T mit Prompt 2 und einer Erfolgsrate von 85%. QC schnitt in beiden Varianten leicht schwächer ab als ChatGPT mit einer Erfolgsrate zwischen 79% und 80%, übertraf den Ausgangscode jedoch ebenfalls deutlich.

Aus den Ergebnissen der Erfolgsrate wird erkennbar, dass eine höhere Fehlererkennungsleistung unter Prompt 2 nicht zwangsläufig mit einer besseren Fehlerbehebung einhergeht. Die Fähigkeit zur Fehlerdiagnose und die Fähigkeit zur tatsächlichen Reparatur sind unterschiedlich stark ausgeprägt und jeweils durch die Prompt-Gestaltung unterschiedlich beeinflusst.

5.2.2 Übersicht zur strukturellen Komplexität der Fehlerbehebung

Zur Analyse der strukturellen Eigenschaften der von den Sprachmodellen erzeugten Korrekturen wurden die zwei zentralen Metriken LOC sowie die CC herangezogen. Als Referenz dienten die fehlerhaften Originalprogramme des QuixBugs-Datensatzes, deren Mittelwert 19,33 LOC betrug und die eine durchschnittliche Komplexität von 5,18 aufwiesen.

Modell	Durchschnitts-LOC	Durchschnitts-CC
QuixBugs Originalcode	19,33	5,18
ChatGPT Prompt 1	19,00	5,20
ChatGPT Prompt 2	19,63	5,78
QC Prompt 1	22,90	6,50
QC Prompt 2	21,10	5,90

Tabelle 5.6: Durchschnittliche LOC und CC der generierten Lösungen

Die Ergebnisse in der Tabelle 5.6 zeigen, dass die generierten Lösungen von ChatGPT in beiden Prompt-Varianten strukturell eng an den Referenzwerten orientiert sind. Die Korrekturen waren kompakt, und die Abweichungen in Codeumfang und Komplexität blieben gering.

QC hingegen erzeugte im Durchschnitt leicht umfangreichere und komplexere Lösungen, besonders bei Verwendung von Prompt 1. In dieser Herangehensweise erreichte der generierte Code mit durchschnittlich 22,9 LOC und einer Komplexität von 6,5 die höchsten Werte im Vergleich. Auch unter Prompt 2 bleiben die Werte über der Referenz, wenn auch leicht reduziert.

Des Weiteren zeigte sich kein klarer Zusammenhang zwischen der LOC oder CC und der Erfolgsrate bei der Fehlerbehebung. So wurden strukturell einfache Programme teilweise nicht erfolgreich behoben, wohingegen komplexere Programme von den LLMs erfolgreich korrigiert wurden. Die Tabelle 5.7 zeigt beispielhafte Programme, die den fehlenden Zusammenhang veranschaulichen.

Programm	LOC	CC	Erfolgsrate
SUBSEQUENCES	15	4	17 %
WRAP	14	3	20 %
HANOI	35	8	100 %

Tabelle 5.7: Beispielprogramme mit unterschiedlicher Komplexität und Reparaturerfolg aus ChatGPT Prompt 1

5.2.3 Vertiefte Analyse der Reparaturleistung

Zur differenzierten Bewertung der Modellleistung wurde neben den durchschnittlichen Werten auch die Verteilung der zentralen Metriken in der Fehlerbehebung analysiert. Im Fokus stehen dabei die Häufigkeit fehlgeschlagener Reparaturen sowie die Differenz zwischen der gemessenen LOC und der CC zum fehlerhaften Originalcode. Diese Verteilungsanalysen ermöglichen ein tieferes Verständnis darüber, in welchen Bereichen die Modelle zuverlässig korrigieren, welche strukturellen Änderungen typischerweise vorgenommen werden und unter welchen Bedingungen Reparaturen scheitern.

Verteilung fehlerhafter Programme

Eine ergänzende Abbildung 5.5 betrachtet die Anzahl der Programme, bei denen die Korrektur des LLMs zu mindestens einem fehlgeschlagenen Test führte.

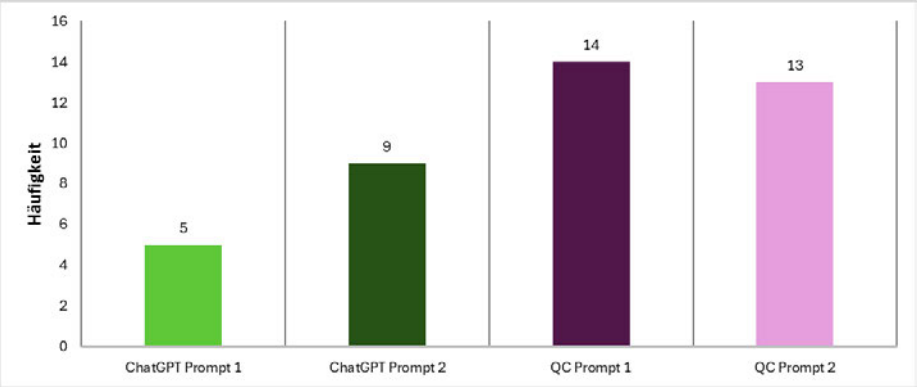

Abbildung 5.5: Anzahl fehlgeschlagener Reparaturen je Modell und Prompt
Quelle: Eigene Darstellung

Die Auswertung zeigt deutliche Unterschiede zwischen den Modellen. Während ChatGPT mit Prompt 1 nur fünf fehlerhafte Programme aufwies, erhöhte sich diese Zahl mit Prompt 2 auf neun. Die Nutzung eines Chain-of-Thought-Prompts führte bei ChatGPT zu weniger erfolgreichen Programmen.

QC zeigte insgesamt eine höhere Fehleranfälligkeit. Das Modell verzeichnete 14 fehlerhafte Programme bei Prompt 1 und 13 fehlerhafte Programme bei Prompt 2. Der Unterschied in der Effektivität zwischen den beiden Prompts war hierbei gering.

Verteilung LOC und CC

Zur vertiefenden Untersuchung der strukturellen Komplexität wurde die strukturelle Veränderung des generierten Codes im Vergleich zur ursprünglichen Quix-Bugs-Version analysiert. Hierbei wurden die Metriken LOC und die CC herangezogen. Für beide Metriken wurde die Differenz zwischen dem generierten Code und dem Originalcode berechnet. Ein positiver Differenzwert zeigt an, dass der von einem Modell generierte Code mehr Zeilen oder eine höhere strukturelle Komplexität aufweist als das Original. Ein negativer Wert weist darauf hin, dass der generierte Code kürzer oder strukturell einfacher ist. Diese Ergebnisse wurden in den folgenden Abbildungen visualisiert.

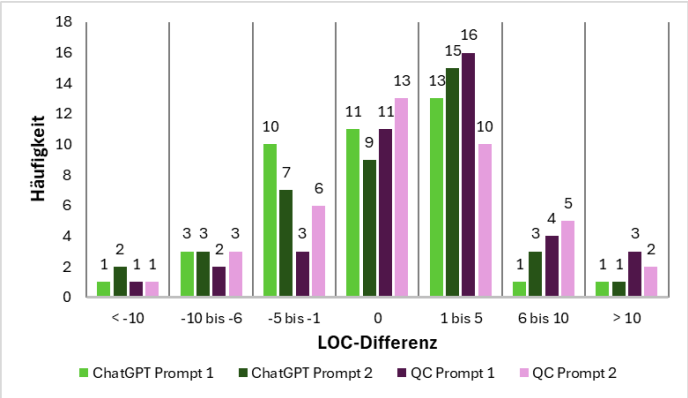

Abbildung 5.6: LOC-Verteilung
Quelle: Eigene Darstellung

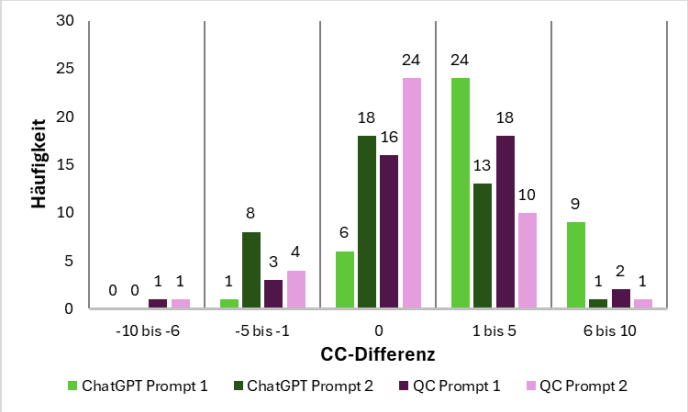

Abbildung 5.7: CC-Verteilung
Quelle: Eigene Darstellung

LOC Die Ergebnisse in der Abbildung 5.6 zeigen, dass die Mehrheit der von den Modellen erzeugten Codes mit geringen bis keinen LOC-Differenzen einherging. Ein Großteil der Lösungen wies eine LOC-Differenz im Bereich null oder von eins bis fünf Zeilen auf. Dies zeigt, dass viele Reparaturen durch kleine Modifikationen erreicht wurden.

Auffällig ist hierbei jedoch, dass QC in beiden Prompt-Varianten häufiger umfangreichere Änderungen vornahm als ChatGPT. Während bei ChatGPT nur vereinzelt größere Abweichungen mit mehr als zehn zusätzlichen Code-Zeilen auftraten, waren solche Fälle bei QC deutlich häufiger vertreten. Auch im Intervall zwischen sechs und zehn geänderten Code-Zeilen zeigte sich eine höhere Frequenz bei QC.

ChatGPT hingegen war ChatGPT gleichmäßiger über die unterschiedlichen LOC-Differenzbereiche verteilt.

Insgesamt lässt sich aus dieser Abbildung feststellen, dass QC unabhängig vom

Prompt zu tendenziell umfangreicheren Codekorrekturen neigt, während ChatGPT häufiger kürzere oder unveränderte Lösungen produziert.

CC Die Abbildung 5.7 zur CC zeigt die strukturellen Veränderungen durch die generierten Lösungen im Vergleich zum Originalcode. Insgesamt bewegt sich der Großteil der Differenzen im Bereich von null sowie von eins bis fünf.

Auffällig ist hierbei das Verhalten von ChatGPT unter Verwendung von Prompt 1, das signifikant häufiger zu einer Erhöhung der CC neigte. Sowohl bei moderaten Erhöhungen von ein bis fünf Einheiten als auch bei größeren Sprüngen von sechs bis zehn Einheiten zeigte diese Variante die höchsten Fallzahlen im Vergleich zu den anderen.

Im Gegensatz dazu führte Prompt 2 bei beiden Modellen häufiger zu strukturell stabileren Lösungen hinsichtlich der CC. Dies zeigte sich sowohl in einer höheren Anzahl von Fällen mit unveränderter Komplexität als auch in einer größeren Häufigkeit von Reduktionen der CC mit einer Differenz von minus eins bis minus fünf im Vergleich zu Prompt 1.

Die Ergebnisse unterstreichen somit den deutlichen Einfluss der Prompting-Strategie auf die strukturellen Eigenschaften des generierten Codes.

5.2.4 Fehlerabhängigkeit der Fehlerbehebung

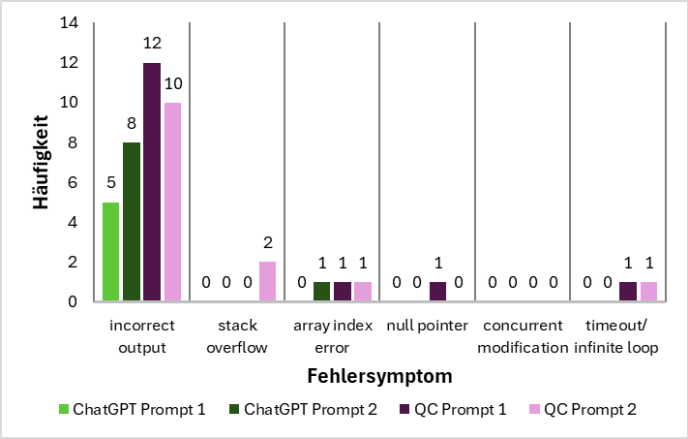

Abbildung 5.8: Häufigkeit fehlgeschlagener Reparaturen nach Fehlersymptom
Quelle: Eigene Darstellung

Die in Abbildung 5.8 dargestellte Übersicht ordnet jedes Programm mit gescheitertem Reparaturversuch einem spezifischen Fehlersymptom zu.

Die Ergebnisse der Analyse zeigen, dass der Großteil der gescheiterten Reparaturen auf den Fehlertyp incorrect output zurückzuführen war, der modell- und promptübergreifend auftrat. Dieser Fehler trat besonders häufig bei QC auf, wobei unter Prompt 1 insgesamt 12 und unter Prompt 2 10 Fälle registriert wurden. ChatGPT wies in dieser Kategorie geringere Werte auf.

QC zeigte eine größere Bandbreite an Fehlersymptomen, bei denen Reparaturen scheiterten. Das Modell wies bei anderen Fehlertypen, wie stack overflow, null pointer und timeout/infinite loop, vereinzelt Reparaturfehlschläge auf.

5.3 Effizienz

Modell und Prompt	**Dauer (s)**
ChatGPT Prompt 1	23,98
ChatGPT Prompt 2	51,65
ChatGPT Prompt 3	24,88
QC Prompt 1	13,79
QC Prompt 2	35,30
QC Prompt 3	15,90

Tabelle 5.8: Durchschnittliche Bearbeitungsdauer der Modelle pro Prompt

Die Ergebnisse der Zeitmessungen in Tabelle 5.8 zeigen, dass Prompt 2 bei beiden Modellen die höchste Bearbeitungszeit aufwies. Bei ChatGPT betrug die durchschnittliche Laufzeit für Prompt 2 über 51 Sekunden, während sie bei QC bei 35,3 Sekunden lag. Damit war Prompt 2 in beiden Fällen signifikant langsamer als Prompt 1.

Prompt 3, der zur Analyse von korrektem Code verwendet wurde, lag bei beiden Modellen im mittleren Bereich und war leicht über Prompt 1. Diese Ergebnisse zeigen, dass die Bearbeitungsdauer stark vom Prompt-Typ beeinflusst wird, wobei Prompt 2 die längste Ausführungszeit verursachte.

Kapitel 6

Diskussion

In diesem Kapitel werden die in der Ergebnisdarstellung präsentierten Befunde diskutiert und interpretiert. Ziel ist es, die Relevanz der Ergebnisse für bestehende Forschungsansätze einzuordnen sowie zentrale Zusammenhänge und Implikationen herauszuarbeiten. Darüber hinaus werden Schwächen und methodische Einschränkungen kritisch reflektiert, die sowohl die Interpretation der Ergebnisse beeinflussen als auch Ansatzpunkte für zukünftige Forschungen liefern. Abschließend werden Empfehlungen für weiterführende Studien formuliert und mögliche Entwicklungen im Themenfeld skizziert.

6.1 Zusammenfassung und Interpretation der Ergebnisse

Ziel dieser Arbeit war es, die Leistungsfähigkeit der zwei aktuellen LLMs ChatGPT und QCs in den Aufgabenbereichen Fehlererkennung, Fehlerbehebung und Effizienz zu untersuchen. Darüber hinaus sollten Zusammenhänge zwischen strukturellen Codeeigenschaften, verwendeten Prompt-Strategien und der Modellleistung identifiziert werden, um differenzierte Aussagen über Stärken, Schwächen und Einsatzpotenziale der Modelle treffen zu können.

6.1.1 Fehlererkennung

In der Fehlererkennung zeigt sich, dass ChatGPT, insbesondere mit Prompt 2, einen hohen Recall mit bis zu 100% erreichte. Jedoch ging dies mit einer erhöhten Anzahl an FPs und markierten Schwachstellen einher. Im Vergleich dazu zeigte QC eine höhere Precision und meldete seltener FPs. Dafür wurden hier insgesamt weniger tatsächliche Fehler erkannt, was sich in niedrigeren Recall-Werten von unter 90% widerspiegelte.

Eine mögliche Erklärung für die erhöhte Anzahl an FPs und Schwachstellen, die besonders in Prompt 2 auftraten, könnte in der Informationsdichte und Komplexität des Prompts liegen. Die umfassende Formulierung des Chain-of-Thought-Prompts enthielt zahlreiche Anforderungen, die das Modell dazu veranlassen

könnten, übervorsichtig zu agieren und mehr potenzielle Probleme zu markieren als tatsächlich vorhanden waren. Dieses Verhalten kann als eine Form von Halluzination interpretiert werden. Dabei generiert das Modell Warnungen oder Fehlerdiagnosen, die nicht durch den tatsächlichen Codezustand gedeckt sind. Dies legt nahe, dass eine übermäßige Prompt-Komplexität oder -Länge zu einer Überforderung des Modells führen kann.

Auffällig war zudem, dass beide Modelle empfindlich auf Fehler des Typs incorrect output reagierten. Dieses Fehlersymptom wurde sowohl bei der Fehlererkennung vergleichsweise häufig übersehen als auch bei der Fehlerbehebung mehrfach nicht korrekt behoben.

Die Tendenz zur Übererkennung äußerte sich auch in Prompt 3. Dort klassifizierten die Modelle zahlreiche syntaktisch korrekte Programme fälschlicherweise als fehlerhaft. Insbesondere ChatGPT neigte dort dazu, auch korrekten Code fälschlicherweise als fehlerhaft zu markieren.

Diese Beobachtungen zur Fehlererkennung decken sich mit den Ergebnissen der Studie von Yu et al. (2024), in der ChatGPT eine eingeschränkte Fähigkeit zur Selbstverifikation nachgewiesen wurde. Dort wurde festgestellt, dass das Modell häufig falsche Selbstbewertungen vornimmt und Halluzinationen produziert. Beispielsweise bewertete es die Reparatur von Code fälschlicherweise als erfolgreich, obwohl objektiv keine funktionale Verbesserung erzielt wurde. Darüber hinaus zeigten die Autoren, dass eine gezielte Modellsteuerung durch sogenannte Guiding Questions zwar zu einer Erhöhung der Recall-Werte führen kann, dies jedoch mit einer Abnahme der Precision einhergeht. Dieser Zielkonflikt spiegelt sich auch in der vorliegenden Arbeit wider.

In der Praxis könnte diese Übererkennung zu einem erhöhten Validierungsaufwand führen und somit den Nutzen automatisierter Fehlerdiagnose relativieren.

6.1.2 Erfolgsraten der Fehlerbehebung

Die Ergebnisse der Fehlerbehebung zeigen, dass beide untersuchten LLMs, ChatGPT und QC, grundsätzlich in der Lage sind, fehlerhaften Java-Code auf dem QuixBugs-Datensatz mit hoher Erfolgsquote zu reparieren.

ChatGPT erzielte in der Fehlerbehebung die beste Leistung und erreichte in Kombination mit Prompt 1 die höchste Erfolgsrate mit 93%. Auch der Prompt 2 führte mit einer Erfolgsquote von 85% zu einer insgesamt hohen Reparaturleistung. Allerdings zeigte sich, dass der positive Einfluss des Chain-of-Thought-Ansatzes auf die Code-Reparatur weniger stark ausgeprägt war als bei der Fehlererkennung.

QC konnte ebenfalls einen Großteil der Programme erfolgreich reparieren, erreichte jedoch geringere Erfolgsraten von 79 bis 80% und wies vermehrt Schwierigkeiten gegenüber mehr Fehlersymptomen als bei ChatGPT auf. Insgesamt liegen die in dieser Arbeit erzielten Erfolgsraten in der oberen Hälfte.

Im Vergleich zu früheren Studien lässt sich eine deutliche Leistungssteigerung feststellen. Prenner et al. (2022) berichteten beispielsweise von Erfolgsraten von 32,5% bei der Verwendung von CoCoNut für Java-Code und 65% mit dem

CURE-Modell. Auch Codex, ein auf Codeaufgaben spezialisiertes LLM, erreichte lediglich eine Erfolgsquote von 35%. Die in dieser Arbeit erzielten Erfolgsraten liegen damit deutlich über den Ergebnissen aus der Studie.

Sobania et al. (2023) untersuchten ebenfalls frühere Versionen von ChatGPT aus dem Zeitraum von Dezember 2022 bis Januar 2023. Mit einem Standard-Prompt, der dem Zero-Shot-Prompt gleicht, erreichten sie eine Erfolgsrate von 47,5%. Durch gezielte Folgeinteraktionen mit spezifischen Fehlerhinweisen konnte diese Quote auf 77,5% gesteigert werden. Die Ergebnisse der vorliegenden Untersuchung übertrafen auch dieses Ergebnis, da ChatGPT mit Prompt 1 bereits ohne zusätzliche Interaktionen eine Erfolgsrate von 93% erreichte.

Eine ähnliche Entwicklung zeigte sich in der Studie von Wuisang et al. (2023). Dort erzielte ChatGPT in der Version GPT-3.5 mit einem Standard-Prompt zunächst eine Erfolgsrate von 62,5%, die durch ergänzende Nachfragen auf 75% erhöht werden konnte. Die in dieser Arbeit erzielten Erfolgsraten belegen damit eine klare Leistungssteigerung durch neuere Modellversionen.

Unter den derzeit verfügbaren Studien sticht insbesondere die Arbeit von Hu et al. (2024) hervor, die mit dem neueren Modell GPT-O1 eine vollständige Fehlerbehebung auf dem QuixBugs-Benchmark erreichte. Dieses Ergebnis übertraf die in der vorliegenden Arbeit erzielten Werte deutlich und weist auf das Potenzial fortgeschrittener Prompting-Mechanismen, z.B. automatisierte Denkschritte, und spezialisierter Trainingsansätze hin. Zugleich deutet das Ergebnis darauf hin, dass die gegenwärtige GPT-4-Version trotz ihrer hohen Leistungsfähigkeit in bestimmten Aufgabenbereichen gegenüber gezielt weiterentwickelten Nachfolgemodellen wie GPT-O1 unterlegen sein kann.

Zusammenfassend unterstreichen die Ergebnisse dieser Arbeit den Fortschritt in der automatisierten Programmreparatur durch LLMs. Die Erfolgsraten in der vorliegenden Untersuchung waren höher als in früheren Studien aus den Jahren 2022 und 2023, das insbesondere auf den Einsatz moderner Modellversion GPT-4 zurückzuführen ist. Der Vergleich mit GPT-O1 zeigt jedoch, dass weiteres Optimierungspotenzial besteht.

Der Vergleich dieser Studien verdeutlicht damit die rasante Entwicklung der LLM-Technologie im Bereich der automatisierten Code-Reparatur innerhalb weniger Jahre. Gleichzeitig lässt sich daraus ableiten, dass durch spezialisierte Trainingsansätze und weitere Feinabstimmungen künftig weitere Leistungssteigerungen möglich sind.

Wie bereits im vorherigen Abschnitt 6.1.1 erwähnt, stellt das Fehlersymptom incorrect output in der vorliegenden Untersuchung die größte Herausforderung für die Modelle dar. Beim Modell QC traten zudem vereinzelt Schwierigkeiten im Umgang mit weiteren spezifischen Fehlertypen auf, darunter stack overflow, null pointer exceptions, timeouts bzw. infinite loops sowie array index errors.

6.1.3 Effizienz

Hinsichtlich der Effizienz zeigten sich signifikante Unterschiede in der Bearbeitungsdauer der Modelle in Abhängigkeit von der gewählten Prompting-Strategie. Prompt 2, der als Chain-of-Thought-Prompt konzipiert war, führte bei beiden untersuchten Modellen zu signifikant längeren Bearbeitungszeiten im Vergleich

zu Prompt 1. Die durchschnittliche Bearbeitungszeit für ChatGPT mit Prompt 2 lag bei über 51 Sekunden und für QC bei 35,3 Sekunden. Damit waren beide Modelle mit Prompt 2 deutlich langsamer als mit den anderen Prompts. Im Vergleich dazu waren die Modelle mit dem minimalistischeren Prompt 1 deutlich schneller, mit durchschnittlich 23,98 Sekunden für ChatGPT und 13,79 Sekunden für QC. Es zeigte sich auch ein Unterschied zwischen den Modellen selbst. QC war über alle Prompt-Varianten hinweg tendenziell schneller als ChatGPT.

Ein wesentlicher Grund für die längeren Bearbeitungszeiten bei Prompt 2 liegt in der höheren Eingabelänge und der geforderten Erklärstruktur. Der Prompt enthielt umfangreichere Anweisungen und forderte detaillierte Begründungen, was zu einer längeren Verarbeitungszeit führte.

Insgesamt verdeutlicht dieses Ergebnis den potenziellen Zielkonflikt zwischen der strukturellen Tiefe des Promptings und der Effizienz. Ausführliche Erklärungen zur Korrektur durch strukturierte Denkprozesse gehen zulasten der Bearbeitungsgeschwindigkeit.

6.1.4 Analyse potenzieller Zusammenhänge

Bezüglich der strukturellen Komplexität des Codes konnte kein eindeutiger Zusammenhang zwischen der Anzahl der LOC oder der CC und der Leistungsfähigkeit der Modelle festgestellt werden. Zwischen der strukturellen Komplexität des Codes und der Erfolgsrate der Reparatur ließ sich kein systematischer Zusammenhang feststellen. Diese Ergebnisse deuten darauf hin, dass diese beiden Metriken kein verlässlicher Prädiktor für den Erfolg von Fehlererkennung oder -behebung durch LLMs sind. Stattdessen scheint das Fehlersymptom einen stärkeren Einfluss auf die Modellleistung auszuüben.

Darüber hinaus zeigte sich, dass die gewählte Prompting-Strategie die Modellleistung bei der Fehlerbehebung beeinflusste. Prompt 2 führte in der Fehlererkennung zu einer erhöhten Recall-Rate, ging jedoch mit einer erhöhten Anzahl an FPs und längeren Bearbeitungszeiten einher. In der Fehlerbehebung hingegen konnte mit Prompt 2 keine konsistente Leistungsverbesserung gegenüber Prompt 1 erzielt werden. Bei ChatGPT sank die Erfolgsquote mit Prompt 2 sogar leicht von 93% auf 85%. Auch QC zeigte bei der Fehlerbehebung keine Vorteile durch Prompt 2.

Die Ergebnisse zeigen insgesamt, dass die Auswahl des Modells einen stärkeren Einfluss auf die Erfolgsraten hatte als die Wahl der Prompting-Strategie. Unabhängig von der gewählten Prompting-Strategie erzielte ChatGPT bei der Fehlerbehebung konsistentere Ergebnisse als QC. Dies zeigte sich in einer höheren Anzahl erfolgreich reparierter Programme und einer insgesamt geringeren strukturellen Komplexität des generierten Codes. Zwar hatte Prompt 2 vereinzelt Auswirkungen, insbesondere im Bereich der Fehlererkennung, jedoch ließ sich kein konsistenter oder signifikanter Leistungszuwachs durch die Verwendung dieser Strategie nachweisen. Auch zwischen der verwendeten Prompting-Strategie und der strukturellen Ausprägung des generierten Codes zeigte sich kein konsistentes Muster.

Diese Ergebnisse stehen teilweise im Kontrast zu mehreren früheren Studien. Sobania et al. (2023) konnten durch die Verwendung von Follow-Up-Anfragen

mit gezielten Fehlerhinweisen die Reparaturquote von ChatGPT von 47,5% auf 77,5% steigern. Wuisang et al. (2023) berichteten ebenfalls von einer signifikanten Verbesserung durch zusätzliche Interaktionen mit einer Steigerung von 62,5% auf 75%. Qu et al. (2024) erreichten durch die Einbettung von Kontextinformationen, wie Problembeschreibungen und Testfällen, eine Erfolgsquote von 83,3% beim Modell GPT-4.

Während in anderen Studien interaktive oder inkrementelle Kontextbereitstellungen genutzt wurden, wurde in der vorliegenden Untersuchung eine einmalige, lineare Aufforderung verwendet. Der Vergleich dieser Ergebnisse legt nahe, dass nicht jede Form der Eingabestruktur gleich wirksam ist. Die Effektivität von Prompting hängt nicht allein von der Länge oder dem Detailgrad der Eingabe ab, sondern vielmehr von deren funktionalem Design und Eignung zur jeweiligen Testaufgabe.

Insgesamt unterstreichen diese Erkenntnisse die zentrale Rolle von Prompt-Engineering in LLM-basierten Softwaretests. Sie bestätigen, dass die Gestaltung der Eingabeaufforderung maßgeblich über Effektivität, Precision und Effizienz entscheidet. Die Ergebnisse deuten darauf hin, dass ein direkter, klar formulierter Zero-Shot-Prompt in bestimmten Kontexten der komplexeren Chain-of-Thought-Variante überlegen sein kann. Des Weiteren weist der Vergleich mit früheren Studien darauf hin, dass die Effektivität des Prompts nicht nur von deren Länge oder Komplexität abhängt. Vielmehr verdeutlichen die Erkenntnisse die zentrale Rolle einer inhaltlich präzisen und kontextbezogenen Formulierung der Eingabe für eine erfolgreiche Programmreparatur.

6.2 Schwächen

Die vorliegenden Ergebnisse liefern fundierte Erkenntnisse über die Leistungsfähigkeit von LLMs in der Fehlererkennung und -behebung. Dennoch verbleiben zentrale Aspekte, die im Rahmen dieser Arbeit nicht abschließend beantwortet werden konnten oder durch methodische Grenzen nur eingeschränkt analysierbar waren.

So konnte zwar gezeigt werden, dass ChatGPT bei der Fehlererkennung und -behebung insgesamt überlegen ist, und dass Prompting-Strategien die Leistungsfähigkeit der LLMs beeinflussen. Jedoch blieben Fragen zur Ursache dieser Effekte offen. Besonders Fragen, inwiefern die Unterschiede in der Modellleistung auf Training, Architektur oder interne Repräsentationen zurückzuführen sind, ließen sich im Rahmen dieser Arbeit nicht analysieren.

Neben den beobachtbaren Leistungsunterschieden blieb auch die Frage offen, inwiefern strukturelle Eigenschaften des Codes LOC und CC mit der Erfolgsrate korrelieren. Die Erfolgsraten der Modelle variierten unabhängig von diesen strukturellen Metriken. Die verwendeten Evaluationskriterien zur strukturellen Analyse der Reparaturqualität waren geeignet, um allgemeine Unterschiede sichtbar zu machen. Dennoch zeigte sich, dass diese Metriken allein keine hinreichende Aussagekraft über die Qualität der Reparatur bieten, insbesondere in Bezug auf logische Korrektheit und potenzielle Nebeneffekte. Um die Qualität von Reparaturvorschlägen umfassender zu bewerten, sollten künftige Arbeiten neben strukturellen Metriken auch semantische Kriterien, wie Lesbarkeit oder

Änderungsumfang, berücksichtigen.

Die erzielten quantitativen Erfolgsraten, wie etwa der hohe Recall-Wert bei ChatGPT oder die hohe Erfolgsquote bei der Fehlerbehebung, spiegeln nur einen Teil des Gesamtbildes wider. Ihre praktische Relevanz und Aussagekraft werden durch qualitative Einschränkungen relativiert. Dazu zählen eine hohe Anzahl an FPs, eine erkennbare Tendenz zur Übererkennung sowie spezifische Schwächen bei der Erkennung bestimmter Fehlertypen. Diese stellen zwar keine Fehler im engeren Sinne dar, erfordern jedoch eine weitere manuelle Prüfung. Dieser zusätzliche Aufwand verringert die praktische Effizienz der Modelle im Anwendungskontext.

Methodisch war die Studie durch mehrere Einschränkungen geprägt. Obwohl Quixbugs als etablierter Benchmark anerkannt ist, bietet der Datensatz nur eine begrenzte Komplexität und Praxisnähe. Realitätsnahe Fehler, die sich über mehrere Klassen oder Module erstrecken, konnten in diesem Rahmen nicht abgebildet werden. Zudem war die Analyse ausschließlich auf Java beschränkt, sodass Aussagen über andere Programmiersprachen nur eingeschränkt übertragbar sind. Die Anzahl der untersuchten Programme war mit 40 begrenzt, was die statistische Generalisierbarkeit der Ergebnisse einschränkt.

Es ist wichtig zu beachten, dass der Datensatz QuixBugs eine ungleiche Verteilung der Fehlersymptome aufweist. Insbesondere trat der Fehlertyp incorrect output deutlich häufiger auf als andere Symptome, wie Laufzeitfehler oder Indexfehler. Eine neutralere Verteilung verschiedener Fehlertypen hätte möglicherweise zu differenzierteren Einsichten in die spezifischen Stärken und Schwächen der Modelle geführt. Das auffällige Vorkommen von incorrect output im Testdatensatz stellt somit eine potenzielle Verzerrung dar, die bei der Interpretation der Ergebnisse berücksichtigt werden muss.

Auch die Gestaltung der Prompts stellt eine Einschränkung dar. Prompt 2, der als monolithischer Chain-of-Thought-Prompt formuliert wurde, spiegelt nicht vollständig wider, wie Entwickler in realen Projekten mit Modellen interagieren würden. Eine inkrementelle Prompting-Strategie mit Zwischenfeedback und Nachfragen hätte hier möglicherweise zu realistischeren und differenzierteren Ergebnissen geführt. Ebenso wurden Interaktionsmöglichkeiten, etwa über Chat-Dialoge mit Fehlermeldungen oder konkreten Testszenarien, in dieser Arbeit nicht berücksichtigt, obwohl andere Studien zeigen, dass diese die Fehlerbehebung deutlich verbessern können. Insbesondere zeigte sich in diesem Prompt ein Konflikt. Ein höherer Recall durch Chain-of-Thought-Prompting ging mit mehr FPs und signifikant längeren Bearbeitungszeiten einher. Dies kann den anfänglich vermuteten Effizienzgewinn infrage stellen.

Schließlich waren nicht alle Methoden im Rahmen dieser Arbeit umsetzbar. So wurden kein Fine-Tuning, kein Few-Shot- oder Differential Prompting und keine Vergleichsmodelle aus anderen Architekturklassen getestet, obwohl Hinweise aus der Literatur darauf deuten, dass diese Ansätze die Leistung in spezifischen Anwendungsszenarien verbessern könnten. In der Studie von Fu et al. (2023) wurde festgestellt, dass Zero-Shot- und Few-Shot-Prompting in der Fehlerbehebung typischerweise eine geringere Leistung erzielten als Modelle, die durch Code-Einbettungen oder Fine-Tuning auf spezifische Aufgaben optimiert wurden.

Auch eine tiefergehende Fokussierung auf einen einzelnen Aspekt, z.B. nur auf die Fehlerbehebung oder nur auf die Fehlererkennung, hätte es ermöglicht, innerhalb des begrenzten Rahmens vertiefende methodische Vergleiche durchzuführen.

Dazu blieb die Frage offen, inwiefern die von den LLMs erzeugten Codekorrekturen langfristig tragfähig sind. Aspekte wie Wiederverwendbarkeit, Wartbarkeit oder Erweiterbarkeit konnten im Rahmen dieser Untersuchung nicht systematisch bewertet werden, da die Analyse auf einmalige Outputs und synthetisch begrenzte Aufgaben beschränkt war.

Zusammenfassend lässt sich festhalten, dass die Arbeit eine fundierte Basis für die Bewertung von LLMs im Softwaretestkontext geschaffen hat. Gleichzeitig bleiben wichtige Fragen zu Trainingsdaten, Modellsensitivität, Prompt-Design und strukturellen Einflüssen offen, die in zukünftiger Forschung gezielter adressiert werden sollten.

6.3 Empfehlungen für zukünftige Arbeiten

Die vorliegende Arbeit liefert eine fundierte Grundlage für die Bewertung von LLMs im Kontext der automatisierten Fehlererkennung und Fehlerbehebung in Programmcode. Die Ergebnisse zeigen, dass LLMs wie ChatGPT und QC bereits heute in der Lage sind, eine Vielzahl von Fehlern zu identifizieren und zu beheben. Gleichzeitig offenbaren sie jedoch Schwächen bei der logischen Analyse, bei bestimmten Fehlertypen sowie im Umgang mit korrektem Code. Daraus ergeben sich vielfältige Ansatzpunkte für weiterführende Forschung und Optimierung.

Ein zentraler Entwicklungspfad liegt in der gezielten Untersuchung des Einflusses von Fehlertypen. Die Ergebnisse dieser Arbeit verdeutlichen, dass insbesondere semantische Fehler, wie incorrect output, eine systematische Herausforderung für beide Modelle darstellen. Künftige Studien sollten daher eine strukturierte Typisierung und differenzierte Analyse von Fehlersymptomen vornehmen, um spezifische Stärken und Schwächen einzelner Modellarchitekturen oder Prompting-Strategien besser erfassen zu können.

Darüber hinaus sollte das Prompting-Verhalten der Modelle tiefergehend analysiert werden. Die vorliegenden Ergebnisse zeigen, dass Prompting-Strategien einen signifikanten Einfluss auf Recall, Precision und Bearbeitungsdauer haben. Auffällig war jedoch, dass komplexere Prompts, wie Prompt 2, zwar häufig zu einer höheren Fehlererkennung führten, gleichzeitig aber auch deutlich mehr FPs verursachten. Dies widerspricht der Erwartung, dass detailliertere Anweisungen automatisch zu besseren und präziseren Ergebnissen führen. Zukünftige Arbeiten sollten daher über Zero-Shot- und monolithisches Chain-of-Thought-Prompting hinaus auch interaktive, inkrementelle und dialogbasierte Prompting-Verfahren einbeziehen. Der Vergleich mit existierenden Studien, wie der von Sobania et al. (2023), legt nahe, dass solche interaktiven Ansätze, z.B. Follow-up-Fragen mit gezieltem Feedback, möglicherweise effektiver sind als komplexe Einzelprompts.

Neben der reinen Fehlerdiagnose und -behebung könnten auch qualitative Aspek-

te der Modellantworten in den Fokus rücken. Die Einbeziehung zusätzlicher Modellinformationen wie Modellkonfidenz oder Tokenanzahl könnte dabei helfen, die Qualität und Verlässlichkeit von Antworten besser einzuschätzen. Auch multimodale Modelle, die neben dem Quellcode auch zusätzliche Kontexte, wie Kommentare, Tests oder Anforderungen, verarbeiten, könnten zu einer realitätsnäheren Modellbewertung beitragen.

Ein weiterer Aspekt betrifft die Auswahl klassischer Softwaremetriken. In dieser Arbeit konnte kein signifikanter Zusammenhang zwischen strukturellen Eigenschaften des Codes, gemessen an LOC und CC, und der Erfolgsrate festgestellt werden. Zukünftige Studien sollten alternative Metriken einbeziehen, die beispielsweise die semantische Komplexität, den Änderungsumfang oder die Wartbarkeit abbilden, um die Reparaturqualität umfassender zu bewerten.

Des Weiteren sollte der Frage nachgegangen werden, inwiefern Übererkennung und Halluzinationen reduziert werden können. Die Ergebnisse deuten darauf hin, dass insbesondere ChatGPT in bestimmten Szenarien dazu neigt, auch korrektes Verhalten fälschlich als fehlerhaft zu markieren. Studien zur Selbstverifizierungsfähigkeit von LLMs, wie etwa von Yu et al. (2024), sollten weiter aufgegriffen werden, insbesondere in Hinblick auf die Rolle der geschätzten Antwortsicherheit des Modells und der Prompt-Struktur. Dies betrifft auch die Auswahl geeigneter Evaluationsmetriken. Neben Recall und Precision sollten künftig auch qualitative Aspekte, wie Erklärbarkeit, Nachvollziehbarkeit und Validierungsaufwand, stärker berücksichtigt werden.

Eine weitere offene Frage betrifft die Nachhaltigkeit der generierten Reparaturen. Langzeitstudien könnten untersuchen, ob von LLMs generierte Codeänderungen dauerhaft wiederverwendbar, wartbar und erweiterbar sind oder lediglich kurzfristige Lösungen bieten. Dabei wäre auch ein Vergleich mit menschlichen Fehlerbehebungsstrategien sinnvoll, um die tatsächliche Praxistauglichkeit automatisierter Korrekturvorschläge zu bewerten. Diese Aspekte wurden im Rahmen der vorliegenden Untersuchung nicht berücksichtigt, stellen jedoch eine naheliegende und relevante Erweiterung für zukünftige Forschungsarbeiten dar.

Ein weiterer sinnvoller Ansatz für zukünftige Forschung besteht darin, LLMs auf realen, produktionsnahen Softwareprojekten zu evaluieren. Der Einsatz von Quellcode aus industriellen Anwendungen würde nicht nur eine höhere Komplexität und Varianz abbilden, sondern auch praxisnähere Rückschlüsse auf die tatsächliche Anwendbarkeit und Leistungsfähigkeit der Modelle im realen Entwicklungsumfeld ermöglichen. Dies könnte dazu beitragen, die Aussagekraft und Übertragbarkeit der Ergebnisse über synthetische Benchmarks hinaus deutlich zu erhöhen. Hierbei ist jedoch der Datenschutz ein kritischer Aspekt, insbesondere bei der Verarbeitung interner oder sensibler Daten durch LLMs. Hou et al. (2024) berichten in ihrer Untersuchung von gravierenden Bedenken hinsichtlich der potenziellen Aufnahme von personenbezogenen Daten in den Eingaben. Dies könnten interne Geschäftsdaten, Benutzerprotokolle oder andere vertrauliche Informationen sein. Die sichere Handhabung und die Anonymisierung sensibler Daten sind somit eine notwendige Voraussetzung für Evaluationsstudien dieser Art.

Eine thematische Fokussierung auf einzelne Teilaspekte, wie beispielsweise ausschließlich Fehlererkennung oder Fehlerbehebung, erscheint schließlich empfeh-

lenswert. Durch eine gezielte Eingrenzung könnten präzisere Evaluationsmetriken, differenzierte Prompt-Varianten und umfangreichere Testsets eingesetzt werden, um die Analyse weiter zu vertiefen und kausale Zusammenhänge besser aufdecken zu können.

Insgesamt eröffnet die zunehmende Leistungsfähigkeit von LLMs vielfältige Möglichkeiten zur Unterstützung im Softwaretest. Um das volle Potenzial auszuschöpfen, bedarf es jedoch einer systematischen Weiterentwicklung der Methodik, einer gezielteren Nutzung von Kontextinformationen sowie einer kritisch reflektierten Integration der Modelle in reale Entwicklungsprozesse.

Kapitel 7

Fazit

Aufbauend auf den zuvor dargestellten und diskutierten Untersuchungsergebnissen werden im Folgenden die zentralen Erkenntnisse zusammengefasst, die Forschungsfragen systematisch beantwortet und die Ergebnisse in den übergeordneten wissenschaftlichen Kontext eingeordnet. Abschließend ein Ausblick auf potenzielle Perspektiven und Fragestellungen zukünftiger Forschungen.

7.1 Zusammenfassung

In der vorliegenden Arbeit wurde die Leistungsfähigkeit von LLMs in den Testaktivitäten Fehlererkennung und -behebung systematisch untersucht. Vor dem Hintergrund wachsender Softwarekomplexität und der zunehmenden Relevanz von LLMs im Software Engineering wurde zunächst der aktuelle Forschungsstand analysiert. Dieser zeigt ein starkes Wachstum von Publikationen in dieser Thematik, insbesondere in den Phasen Implementierung, Wartung und Qualitätssicherung. Zentrale Anwendungsfelder in der Qualitätssicherung sind dabei die Testfallgenerierung und die automatisierte Programmreparatur.

Zur empirischen Überprüfung wurde ein experimentelles Design gewählt, das das generische Modell ChatGPT und das auf Programmieraufgaben spezialisierte Modell QC auf dem QuixBugs-Datensatz in der Java-Version vergleicht. Beide Modelle wurden unter Verwendung unterschiedlicher Prompting-Strategien getestet.

Die Ergebnisse zeigten, dass ChatGPT in der Fehlererkennung tendenziell höhere Recall-Werte erreichte, während QC bessere Ergebnisse in der Precision erzielte. Allerdings generierten beide Modelle fehlerhafte Klassifizierungen. Dabei war ChatGPT unter Prompt 2 anfälliger für Übererkennung. Auffallend war zudem, dass das Fehlersymptom incorrect output für beide Modelle sowohl bei der Erkennung als auch in der Programmreparatur eine Herausforderung darstellte.

Bei der Fehlerbehebung zeigte ChatGPT mit Prompt 1 die höchste Erfolgsrate von 93%. Mit Prompt 2 wurde zwar die Fehlererkennung verbessert, jedoch führte dies nicht zu vergleichbaren Leistungssteigerungen in der Fehlerbehebung. QC lag in der Erfolgsrate leicht darunter und zeigte im Vergleich Schwierigkei-

ten mit einer größeren Varianz von Fehlersymptomen. ChatGPT lieferte in der Fehlerbehebung insgesamt bessere Ergebnisse durch höhere Erfolgsraten und strukturell einfachere Lösungsvorschläge im Vergleich zu QC.

Obwohl die Modelle den Großteil der Fehler erkennen und beheben konnten, lässt sich daraus keine generelle praktische Anwendung ableiten. Die Probleme im QuixBugs-Datensatz sind vergleichsweise klein und synthetisch. Die hohe Anzahl an Fehlklassifikationen und inkonsistenten Reparaturen unterstreicht, dass LLMs nicht blind im Qualitätssicherungsprozess eingesetzt werden sollten. Die generierten Ergebnisse sollten kritisch überprüft und hinsichtlich ihrer technischen Plausibilität evaluiert werden.

Auch zum Prompting konnten im Rahmen dieser Arbeit relevante Erkenntnisse gewonnen werden. Ein Zusammenhang zwischen der strukturellen Komplexität des generierten Codes und dem Reparaturerfolg konnte dabei jedoch nicht nachgewiesen werden. Während Prompt 2 bei der Fehlererkennung vereinzelt Leistungsverbesserungen erzielte, blieb eine konsistente Steigerung der Reparaturqualität aus. In einigen Fällen verschlechterten sich die Ergebnisse sogar in Bezug auf Laufzeit und Fehlerhäufigkeit.

Die Aussagekraft der Ergebnisse ist jedoch aufgrund methodischer Einschränkungen begrenzt. Dazu zählen der Einsatz des synthetischen QuixBugs-Datensatzes sowie die begrenzte Komplexität der untersuchten Programme. Dies schränkt die Generalisierbarkeit auf reale Projekte ein.

7.2 Ausblick

Aufbauend auf den gewonnenen Erkenntnissen ergeben sich verschiedene Ansatzpunkte für weiterführende Forschungen. Zukünftige Arbeiten sollten die modellabhängige Anfälligkeit gegenüber spezifischen Fehlertypen, wie incorrect output, detaillierter untersuchen. Die Weiterentwicklung von Prompting-Strategien, wie etwa durch interaktive oder dialogbasierte Ansätze, könnte zusätzliche Leistungssteigerungen ermöglichen.

Darüber hinaus ist eine Erweiterung der Evaluation auf realistischere, größere und diversifizierte Datensätze mit komplexeren Fehlerstrukturen und unterschiedlichen Programmiersprachen empfehlenswert, um die Übertragbarkeit und Generalisierbarkeit der Ergebnisse zu prüfen. Neben quantitativen Metriken sollten qualitative Evaluationskriterien, wie Wartbarkeit, Lesbarkeit oder Sicherheitsaspekte stärker berücksichtigt werden. Ergänzend erscheint eine Untersuchung der Nachhaltigkeit und Robustheit von Korrekturen der generierten Lösungen über einen längeren Zeitraum hinweg sinnvoll. Ebenso sollte die Analyse bislang vernachlässigter Phasen, wie beispielsweise der Testplanung, verstärkt in den Fokus genommen werden.

LLMs verfügen über ein hohes Potenzial zur Unterstützung verschiedener Aufgaben in der Softwareentwicklung. Die Ergebnisse dieser Arbeit verdeutlichen, dass sie bereits in der Lage sind, zentrale Testaufgaben wie die Fehlererkennung und -behebung effektiv zu unterstützen. Um dieses Potenzial jedoch weiter auszuschöpfen, ist weitere Forschung nötig, die sowohl die methodischen Grundlagen als auch die praktische Einbettung in den Entwicklungsalltag vertieft.

Literaturverzeichnis

Bahrini, A., Khamoshifar, M., Abbasimehr, H., Riggs, R. J., Esmaeili, M., Majdabadkohne, R. M., & Pasehvar, M. (2023). ChatGPT: Applications, Opportunities, and Threats. *2023 Systems and Information Engineering Design Symposium (SIEDS)*, 274–279. https://doi.org/10.1109/SIEDS58326.2023.10137850

Brito, C., Durelli, V. H. S., Durelli, R. S., Souza, S. R. S. d., Vincenzi, A. M. R., & Delamaro, M. E. (2020). A Preliminary Investigation into Using Machine Learning Algorithms to Identify Minimal and Equivalent Mutants. *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 304–313. https://doi.org/10.1109/ICSTW50294.2020.00056

Broy, M., & Kuhrmann, M. (2021). Softwareevolution. In *Einführung in die Softwaretechnik* (S. 535–574). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-50263-1_13

Campesato, O. (2024). *Large Language Models for Developers: A Prompt-based Exploration of LLMs*. Mercury Learning; Information. https://doi.org/10.1515/9781501520938

Capgemini Research Institute. (2024). *Turbocharging software with Gen AI: How organizations can realize the full potential of generative AI for software engineering* (Techn. Ber.). Capgemini. https://www.capgemini.com/wp-content/uploads/2024/03/Final-Web-Version-Report-Gen-AI-in-Software-Engineering.pdf

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large Language Models for Software Engineering: Survey and Open Problems. *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 31–53. https://doi.org/10.1109/ICSE-FoSE59343.2023.00008

Fu, M., Tantithamthavorn, C. K., Nguyen, V., & Le, T. (2023). ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, 632–636. https://doi.org/10.1109/APSEC60848.2023.00085

Gao, C., Hu, X., Gao, S., Xia, X., & Jin, Z. (2025). The Current Challenges of Software Engineering in the Era of Large Language Models [Place: New York, NY, USA Publisher: Association for Computing Machinery]. *ACM Trans. Softw. Eng. Methodol.* https://doi.org/10.1145/3712005 Just Accepted.

Gurcan, F., Dalveren, G. G. M., Cagiltay, N. E., Roman, D., & Soylu, A. (2022). Evolution of Software Testing Strategies and Trends: Semantic Content

Analysis of Software Research Corpus of the Last 40 Years. *IEEE Access*, *10*, 106093–106109. https://doi.org/10.1109/ACCESS.2022.3211949

Hoffmann, D. W. (2013). *Software-Qualität* (2. Aufl.). Springer Vieweg. https://doi.org/10.1007/978-3-642-35700-8

Hossain, M. (2018). Challenges of software quality assurance and testing. *International Journal of Software Engineering and Computer Systems*, *4*(1), 133–144.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology*, *33*(8), 1–79. https://doi.org/10.1145/3695988

Hu, H., Shang, Y., Xu, G., He, C., & Zhang, Q. (2024). Can GPT-O1 Kill All Bugs? An Evaluation of GPT-Family LLMs on QuixBugs [_eprint: 2409.10033]. https://arxiv.org/abs/2409.10033

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., Dang, K., Fan, Y., Zhang, Y., Yang, A., Men, R., Huang, F., Zheng, B., Miao, Y., Quan, S., … Lin, J. (2024). Qwen2.5-Coder Technical Report [_eprint: 2409.12186]. https://arxiv.org/abs/2409.12186

Hunziker, S., & Blankenagel, M. (2024). *Forschungsdesign im Bereich Betriebswirtschaft und Management: Ein praktischer Leitfaden für Studierende und Forschende*. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-44859-2

Jahić, J., & Sami, A. (2024). State of Practice: LLMs in Software Engineering and Software Architecture [ISSN: 2768-4288]. *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, 311–318. https://doi.org/10.1109/ICSA-C63560.2024.00059

Krebs, D., & Menold, N. (2019). Gütekriterien quantitativer Sozialforschung. In N. Baur & J. Blasius (Hrsg.), *Handbuch Methoden der empirischen Sozialforschung* (S. 489–504). Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-21308-4_34

Li, D., & Murr, L. (2024). HumanEval on Latest GPT Models – 2024 [_eprint: 2402.14852]. https://arxiv.org/abs/2402.14852

Li, T.-O., Zong, W., Wang, Y., Tian, H., Wang, Y., Cheung, S.-C., & Kramer, J. (2023). Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 14–26. https://doi.org/10.1109/ASE56229.2023.00089

Lin, D., Koppel, J., Chen, A., & Solar-Lezama, A. (2017). QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 55–56. https://doi.org/10.1145/3135932.3135941

Mohajer, M. M., Aleithan, R., Harzevili, N. S., Wei, M., Belle, A. B., Pham, H. V., & Wang, S. (2024). Effectiveness of ChatGPT for Static Analysis: How Far Are We? [event-place: Porto de Galinhas, Brazil]. *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 151–160. https://doi.org/10.1145/3664646.3664777

Prenner, J. A., Babii, H., & Robbes, R. (2022). Can OpenAI's codex fix bugs? an evaluation on QuixBugs [event-place: Pittsburgh, Pennsylvania]. *Proceedings of the Third International Workshop on Automated Program Repair*, 69–75. https://doi.org/10.1145/3524459.3527351

Qu, X., Zuo, F., Li, X., & Rhee, J. (2024). Context Matters: Investigating Its Impact on ChatGPT's Bug Fixing Performance. *2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA)*, 17–23. https://doi.org/10.1109/SERA61261.2024.10685568

Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, *55*(8), 1397–1418. https://doi.org/10.1016/j.infsof.2013.02.009

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, *50*(1), 85–105. https://doi.org/10.1109/TSE.2023.3334955

Sibbertsen, P., & Lehne, H. (2021). Eindimensionale empirische Verteilungen [Series Title: Springer-Lehrbuch]. In *Statistik* (S. 9–40). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-46235-5_2

Sobania, D., Briesch, M., Hanna, C., & Petke, J. (2023). An Analysis of the Automatic Bug Fixing Performance of ChatGPT. *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 23–30. https://doi.org/10.1109/APR59189.2023.00012

Sommerville, I. (2018). *Software Engineering*. Pearson Deutschland. https://elibrary.pearson.de/book/99.150005/9783863268350

Tian, Z., Shu, H., Wang, D., Cao, X., Kamei, Y., & Chen, J. (2024). Large Language Models for Equivalent Mutant Detection: How Far Are We? [event-place: Vienna, Austria]. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1733–1745. https://doi.org/10.1145/3650212.3680395

Viet, T. D., & Markov, K. (2023). Using Large Language Models for Bug Localization and Fixing. *2023 12th International Conference on Awareness Science and Technology (iCAST)*, 192–197. https://doi.org/10.1109/iCAST57874.2023.10359304

Vytovtov, P., & Markov, E. (2017). Source code quality classification based on software metrics. *2017 20th Conference of Open Innovations Association (FRUCT)*, 505–511. https://doi.org/10.23919/FRUCT.2017.8071355

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering*, *50*(4), 911–936. https://doi.org/10.1109/TSE.2024.3368208

Wang, Q., Wang, J., Li, M., Wang, Y., & Liu, Z. (2024). A Roadmap for Software Testing in Open-Collaborative and AI-Powered Era [Place: New York, NY, USA Publisher: Association for Computing Machinery]. *ACM Trans. Softw. Eng. Methodol.* https://doi.org/10.1145/3709355 Just Accepted.

Witte, F. (2019). *Testmanagement und Softwaretest: Theoretische Grundlagen und praktische Umsetzung*. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-25087-4

Wong, W. E., Debroy, V., Surampudi, A., Kim, H., & Siok, M. F. (2010). Recent Catastrophic Accidents: Investigating How Software was Responsible. *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, 14–22. https://doi.org/10.1109/SSIRI.2010.38

Wuisang, M. C., Kurniawan, M., Wira Santosa, K. A., Agung Santoso Gunawan, A., & Saputra, K. E. (2023). An Evaluation of the Effectiveness of OpenAI's ChatGPT for Automated Python Program Bug Fixing using QuixBugs. *2023 International Seminar on Application for Technology of Information and Communication (iSemantic)*, 295–300. https://doi.org/10.1109/iSemantic59612.2023.10295323

Ye, H., Martinez, M., Durieux, T., & Monperrus, M. (2021). A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software*, *171*, 110825. https://doi.org/10.1016/j.jss.2020.110825

Yenduri, G., Ramalingam, M., Selvi, G. C., Supriya, Y., Srivastava, G., Maddikunta, P. K. R., Raj, G. D., Jhaveri, R. H., Prabadevi, B., Wang, W., Vasilakos, A. V., & Gadekallu, T. R. (2024). GPT (Generative Pre-Trained Transformer)— A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions. *IEEE Access*, *12*, 54608–54649. https://doi.org/10.1109/ACCESS.2024.3389497

Yu, X., Liu, L., Hu, X., Keung, J. W., Liu, J., & Xia, X. (2024). Fight Fire With Fire: How Much Can We Trust ChatGPT on Source Code-Related Tasks? *IEEE Transactions on Software Engineering*, *50*(12), 3435–3453. https://doi.org/10.1109/TSE.2024.3492204

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original