Bug Detection in der Software-Wartung und -Pflege mit Large Language Models

Raphael Wudy
Fakultät für Informatik
Technische Hochschule Rosenheim
Rosenheim, Deutschland
raphael.wudy@stud.th-rosenheim.de

***Zusammenfassung*—In der Software-Wartung und -Pflege ist Debuggen eine wiederkehrende Aufgabe, die sehr viel Zeit in Anspruch nimmt. Dabei haben sich über die Zeit hinweg verschiedene Methoden zur Bug Detection entwickelt. Einige davon starten bereits im Entwicklungsprozess der Software (statische Code Analysen und Logging der Software) und andere dienen mehr dem Verständnis der zu wartenden Software. Dazu zählen dynamische Code Analysen. Durch diese ist bei eintreffendem Bug Report bereits ein Überblick des Systems vorhanden. Die Reproduzierbarkeit ist ein weiterer wichtiger Bestandteil. Nur so können angemessene Patches veröffentlicht werden. Durch das Aufstreben der Large Language Models stellt sich die Frage, welche neuen Möglichkeiten sich eröffnen und ob diese in bestehende Ansätze integriert werden können. Die in der jüngsten Vergangenheit durchgeführten Studien zeigen, dass Large Language Models tatsächlich hilfreich für Bug Detection sein können. Damit diese zuverlässig sind und eine gute Performance erreichen benötigen Large Language Models ein gezieltes Training und verschiedene Ausprägungen von Prompting. Prompting der Aufgabenstellung ohne weitere Anweisungen erzielt meist eine Erfolgsquote von 50%. In einer Binärklassifizierung würde Raten dabei eine ähnliche Erfolgsquote erzielen.**

***Index Terms*—bug detection, Software-Wartung, large language models, Methoden**

I. Einleitung

Ein ständige Herausforderung in der Software-Wartung und -Pflege ist die Bug Detection. Dabei geht es darum, die bereits ausgelieferte Software auf Bugs zu untersuchen und an deren Behebung mitzuwirken. Diese Aufgabe kann sich je nach Art des Bugs als komplex gestalten. Weil nicht jeder Bug auf Anhieb gefunden werden kann und nur unter bestimmten Umständen auftritt, ist Bug Detection eine ständige Aufgabe der Software-Entwicklung und -Wartung [1]. Daher stellt Debuggen eine zeitintensive Aufgabe dar [2]. Damit sich diese Zeiten reduzieren, haben sich über die Zeit hinweg verschiedene Methoden etabliert, die den Prozess der Bug Detection unterstützen sollen [3]–[6].

Large Language Models (LLMs) zeigen bereits Einfluss und Veränderung in der Branche der IT in ihren jungen Jahren [7]. Deswegen muss untersucht werden, welches Potential in diesen steckt und ob diese sich als neues Werkzeug im Toolkit der Bug Detection integrieren lassen. Auf dem Markt finden sich verschiedene Modelle wie Llama-2 [8], CodeBERT [9] (Open Source) oder ChatGPT [10] (Closed Source). Dabei erlauben Benchmarks den Vergleich der Performance von verschiedenen Modellen. Damit diese Vergleichbarkeit bei der Bug Detection entstehen kann, ist es wichtig für diese Aufgabe Benchmarks zu finden und konsequent zu verwenden [11].

Um einen vollen Einblick in den dahinterstehenden Prozess der Bug Detection von LLMs zu erhalten, sollten neben den wissenschaftlichen Untersuchungen die LLMs selbst gefragt werden. Daher folgt ein Auszug aus einem Chatverlauf mit ChatGPT 3.5, in der das LLM die Frage beantwortet warum ChatGPT in der Lage ist Bug Detection zu betreiben und weiterführend wie zuverlässig ChatGPT dabei ist. "[...] Wenn ich Code analysiere, zerlege ich ihn in kleinere Teile, um das Verständnis zu erleichtern. Ich überprüfe dann die Syntax, um sicherzustellen, dass sie den Regeln der Programmiersprache entspricht. Danach betrachte ich die Logik des Codes, um zu sehen, ob die Anweisungen sinnvoll sind und ob potenzielle Fehler, wie z.B. logische Widersprüche oder undefinierte Variablen, vorhanden sind. Mein Verständnis von Programmierkonventionen hilft mir auch, gängige Fehlermuster zu erkennen."[1]

"Meine Fehlererkennungsrate hängt von verschiedenen Faktoren ab, wie der Komplexität des Codes, der Klarheit der Fehlermeldung und meinem aktuellen Wissensstand. Generell bin ich recht gut darin, offensichtliche Fehler zu erkennen, aber bei komplexeren oder subtileren Problemen kann meine Genauigkeit variieren. Es ist wichtig zu beachten, dass ich zwar helfen kann, Fehler zu finden, aber nicht perfekt bin und menschliche Überprüfung oft ratsam ist, insbesondere bei kritischen Anwendungen."[2]

II. Hintergrund

A. Definition von Bugs

In der Software-Entwicklung gibt es für einen Bug unterschiedliche Beschreibungen. Für ein einheitliches Verständnis liefert IEEE ein Standard Glossar für Software Engineering Terminologien [12] und definiert dabei einen Bug als

1) Eine Differenz zwischen einem berechneten, beobachten oder gemessenen Wert oder Zustand und dem wahren spezifizierten oder theoretisch korrekten Wert

[1]Auszug Chatverlauf mit ChatGPT Version 3.5 - Selbsteinschätzung: Warum ChatGPT in der Lage ist Bug Detection zu betreiben

[2]Auszug Chatverlauf mit ChatGPT Version 3.5 - Selbsteinschätzung: Wie zuverlässig ChatGPT bei der Bug Detection ist

2) Eine falsche Schritt-, Prozess-, oder Datendefinition
3) Ein falsches Ergebnis
4) Eine menschliche Handlung, die ein falsches Ergebnis produziert

Diese Bugs können näher definiert werden. Einer dieser definierten Bugs kann dynamischer Natur sein und ist von einer Eingabe abhängig. Im Kontrast dazu sind statische Bugs unabhängig von der Eingabe. Diese beiden näheren Beschreibungen sind nur eine Auswahl der im IEEE Glossar [12] beschriebenen Definitionen. Daraus lässt sich schlussfolgern, dass ein nicht spezifiziertes Ergebnis oder Verhalten als Bug zu kategorisieren ist. Somit sind nicht nur logische Fehler wie zum Beispiel ein falscher Operand in einer If-Bedingung als Bug einzustufen, sondern auch fachliche Fehler wie das Subtrahieren aller Einzelpositionen in einem Bankkonto zum Anzeigen des Saldos.

Bei der Betrachtung verschiedener Arten, sind Regression Bugs ebenfalls zu berücksichtigen. Dieser entsteht durch Software Updates, die der Verbesserung dienen. Diese Bugs beschreiben einen vor dem Update funktionierenden Programmteil, der nach dem Update nicht mehr gemäß Spezifikation funktioniert [13]. Diesen zu identifizieren, einen Patch vorzubereiten und das ungewollte Ausrollen des Regression Bugs sind Bestandteil der Software-Wartung.

B. *Bug Detection ohne Large Language Models*

Damit im späteren Verlauf ein Vergleich zwischen der Bug Detection mit LLMs möglich ist, werden im folgenden bisherige Methoden zur Bug Detection erläutert. Eine Datenerhebung und Untersuchung von Studien wie im Abschnitt III-A erfolgt allerdings nicht. Für einen direkten Vergleich der Performance ist diese Arbeit nicht vorgesehen. Dennoch ist der Einfluss bisheriger Methoden auf die Bug Detection nicht zu vernachlässigen. Einige der aufgeführten Methoden können bereits in der Entwicklungsphase integriert und implementiert werden, wie zum Beispiel statische Code Analysen und Logging des Verhaltens der Software.

*a) **Statische Code Analyse**:* Die Software wird ohne tatsächliche Ausführung untersucht, um Bugs oder potentielle Schwachstellen in der frühen Entwicklungsphase der Software zu erkennen. Statische Code Analysen (SCA) sind heutzutage weit verbreitet und verbessern bei der Nutzung die Qualität der Software [6]. Nicht nur in der Entwicklungsphase der Software ist der Einsatz möglich, sondern auch im späteren Verlauf der Software-Wartung und -Pflege. Sofern noch keine SCA durchgeführt wurde, zeigt das statische Analyse Tool unter anderem Bugs und Schwachstellen der Software an. Aspekte der Software-Wartung sind zum Beispiel Wartbarkeit, Zuverlässigkeit und Sicherheit der Software. Diese analysieren SCA Tools wie beispielsweise SonarQube [14].

*b) **Dynamische Code Analysen**:* Diese basieren auf der Untersuchung der Eigenschaften eines Systems, die während der Laufzeit gewonnen werden. Dabei steigt das Verständnis während der Software-Wartung und -Pflege des Systems, weil dort zielgerichtet die Teile der Anwendung im Fokus stehen, die von Interesse sind. Dynamische Code Analysen dienen weniger der Bug Detection, sondern sind ein Werkzeug im Software Engineering für das Verständnis der zu wartenden Software [15].

*c) **Reproduzierbarkeit**:* Da Software Systeme komplexer werden und nicht nur in der ersten Veröffentlichung Bugs enthalten können, ist die Reproduzierbarkeit von Bugs ein wichtiger Bestandteil in der Software-Wartung und -Pflege. Die bereits ausgelieferten Bugs entstehen nicht mehr in der Testumgebung der Softwarehersteller, sondern manifestieren sich auf der Anwenderseite. Da die Berichte (sofern vorhanden) von Anwendern über den Bug unterschiedlich ausfallen können, ist das Verifizieren der Aussage notwendig. Durch die Reproduzierbarkeit ist dies möglich. Ein weiterer Aspekt für die Reproduzierbarkeit von Bugs ist, dass in diesen Fällen passende Patches entwickelt und veröffentlicht werden können [16].

*d) **Bug Reports & Logdateien**:* In den Bug Reports werden Schritte zur Reproduzierbarkeit genannt oder das erwartete und beobachtete Verhalten beschrieben. Nur wenige Bug Reports enthalten Stack Traces, Code Beispiele oder Testfälle. Meistens haben die Bug Reporter Schwierigkeiten diese bereitzustellen [17]. Neben den Bug Reports sind relevante Informationen zu dem Bug in den Logdateien der Software vorhanden. Diese beschreiben das Systemverhalten zum Zeitpunkt des Bugs, die Datei welche zum Fehler geführt hat oder sogar den Stack Trace. Des Weiteren kann in der Entwicklungsphase bereits mit Logdateien gearbeitet werden. Dabei kann die Software das Systemverhalten explizit mitschreiben oder andere Werte dokumentieren, die es im späteren Verlauf der Analyse leichter machen den Fehler zu finden. Dies muss bereits in der Entwicklungsphase implementiert werden, damit ein positiver Nebeneffekt für die Software-Wartung und -Pflege entstehen kann. Diese Logdateien können Hinweise auf die Funktionsweise des Systems und ähnliches mitteilen [18].

III. Aktueller Stand der Forschung

A. *Ansätze zur Bug Detection*

*1) **Statische Code Analyse mit LLM basiertem Ansatz**:* Dabei wird zuerst analysiert und bei der sogenannten Symbolic Execution (einem weiteren Teil der SCA) treten Laufzeit- oder Speicherprobleme auf. In dieser Phase ist es möglich Fehlalarme zu identifizieren und aus der potentiellen Bug-Liste zu entfernen oder weitere Bugs zu entdecken. Da diese Phase nicht abgeschlossen wird, bleibt die Fehlerliste in der initialen Größe vorhanden. Eine manuelle Analyse dieser ist nicht wünschenswert und unproduktiv. Daher kann in einem LLM integrierten Ansatz das LLM mit dieser übergebenen Liste arbeiten. Anhand der Liste kann das LLM eine gezielte Evaluation der Software durchführen. In komplexen System können Statische Code Analyse Frameworks an ihre Grenzen kommen und zeigen diverse Laufzeit- oder Speicherprobleme an. Es sind also weitere Methoden zur Verbesserung der Performance und Lösungsansätze mit LLMs notwendig. Einfaches Prompting ist dabei nicht ausreichend. Wie Tabelle V zeigt, wird hier nur ein niedriger F1-Score erzielt. Durch

Tabelle I
FINETUNING-ANSATZ UND IN-CONTEXT-LEARNING-ANSATZ MIT BINÄRKLASSIFIKATION IM VERGLEICH. SOWIE ZAHLEN ZU IN-CONTEXT-LEARNING MIT DER CODE PAIR KLASSIFIKATION BASIEREND AUF DEM PYPIBUGS DATENSATZ [19]

Ansatz	Aufgabe	LLM	Präzision	F1-Score
Supervised learning (initiales finetuning)	Binärklassifikation	CodeBERT	51.96	36.41
		CodeT5	50.00	49.67
Supervised learning	Binärklassifikation	CodeBERT	61.13	60.26
		CodeT5	60.48	59.68
In-context learning	Binärklassifikation	GPT-3.5	54.15	60.67
		CodeLlama	50.44	32.24
In-context learning	Code-pair Klassifikation	GPT-3.5	72.93	84.34
		CodeLlama	69.87	82.26

diesen kann die Effektivität des zugrundeliegenden LLMs gemessen werden [20]. Dennoch kann durch diesen Ansatz das LLM die potentielle Fehlerliste evaluieren und verifizieren. Die SCA kann dadurch für komplexe Systeme erfolgreich abgeschlossen werden [21]. Im Grunde analysiert das LLM die Warnungen der SCA und interpretiert diese. Dabei spielt zum Beispiel die Begrenzung der Token für Prompting eine Rolle [3].

*2) **In-Context Learning**:* LLMs sind in der Lage neue Aufgaben zu erlernen, wenn ihnen innerhalb einer Konversation die zugrundeliegenden Logik erläutert wird. Durch diese Gedankenkette (Chain of Thoughts) kann das LLM die präsentierte Logik adaptieren und auf die neue Aufgabe transferieren. Dadurch entsteht ein sogenanntes in context learning (ICL) und das LLM muss dabei keinen neuen Finetuningprozess durchlaufen, um diese Aufgabe zu erlernen. Diese Methode zeigt neues Potential in der Verwendung von LLMs in verschiedenen Aufgabenbereichen. Ein nicht auf Bug Detection trainiertes LLM, kann dies durch ICL erlernen und durchführen. Die Performance und Erfolgsquote der Ergebnisse hängt dennoch von anderen Faktoren, wie zum Beispiel dem Finetuningprozess ab [19], [22], [23].

*3) **Code Pair Klassifizierung**:* Bei diesem Ansatz wird implizit der ICL-Ansatz verwendet, damit das zugrundeliegende LLM nicht neu trainiert werden muss. Für die Durchführung einer Code Pair Klassifizierung wird dem ausgewählten LLM, bspw. GPT-3.5 oder CodeLlaMa, fehlerhafter Code und dessen korrigierte Version präsentiert. Die Aufgabe des LLMs ist es dann, den fehlerhaften Code zu identifizieren. Diese Aufgabe ist für das Model leichter zu bearbeiten als die Binärklassifizierung eines fehlerhaften Codeausschnitts ohne korrigierte Version [19].

*4) **Finetuning**:* Um ein pre-trained LLM weiter zu verfeinern gibt es den Prozess des Finetunings. In diesem Training wird das pre-trained Language Model weiter mit einem spezifischeren Datensatz trainiert. Der Datensatz kann dabei sehr spezifisch sein und nur aus Code Beispielen bestehen oder breiter gestreut sein. Wenn solche Daten etwa Beispiele samt Bezeichnungen enthalten [24] kann das LLM neue Fähigkeiten erlernen, wie das Verstehen und Interpretieren von Code oder Bug Detection betreiben. Dadurch ist es dem Trainer des LLMs möglich, dieses in bestimmte Bahnen zu leiten. Ein auf Code trainiertes LLM ist CodeLlama basierend auf Llama 2 mit einem weiteren Finetuningprozess mit codespezifischen Datensätzen [25].

*5) **Prompt Engineering**:* LLMs generieren Text als Antwort auf eine ihnen gestellte Aufgabe. So wie die Antwort eines Menschen zu unterschiedlichen Zeiten auf ein und dieselbe Aufgabe variieren kann, variieren die Antworten von LLMs auf dieselben Aufgaben. Die Kunst im Umgang von LLMs besteht demnach darin, zuverlässige Antworten von LLMs auf ein und dieselbe Aufgabe zu erhalten. Versuche mit LLMs haben gezeigt, dass diese in der Textgenerierung falsche Aussagen tätigen können [26]. Um diese Unzulänglichkeiten unter Kontrolle zu bringen, hat sich Prompt Engineering etabliert. Hier werden Prompts spezifisch für die Aufgabe entworfen, um bessere Ergebnisse zu erzielen. Die Prompts verlängern sich und enthalten dabei mehr Information für das LLM [27], [28]. Eine weitere Möglichkeit sind konfigurierbare Parameter wie bspw. der Temperature-Parameter bei ChatGPT. Je geringer dieser angesetzt ist, desto logischer und in der Semantik ähnlicher sind die Aussagen. Ein höherer Wert lässt mehr ”Kreativität” des LLMs zu [29].

B. Daten zur Bug Detection

Bei der Binärklassifizierung von fehlerhaftem Code mit dem Finetuning-Ansatz und dem ICL-Ansatz zeigt ein Experiment, dass beide Ansätze im Bereich der 50%-Marke angesiedelt sind und somit nicht besser als ”raten” sind. Die Tabelle I aus dem Experiment zeigt, dass bei einem Finetuning-Ansatz die Präzision der ausgewählten Modelle CodeBERT und CodeT5 um 10% steigt. Der verfolgte Ansatz der Code Pair Klassifizierung (CPK) gegenüber der Binärklassifizierung zeigt einen Anstieg von 19% in der Präzision. Der ICL-Ansatz mit CPK bei den ausgewählten LLMs GPT-3.5 und CodeLlaMa steigt von 54.15% auf 72.93% und von 50.44% auf 69.87%. Der Unterschied zwischen GPT-3.5 und CodeLlaMa liegt in erster Linie im Tainingsprozess. Die Datensätze für CodeLlaMa beziehen sich eher auf Source Code [25] und GPT-3.5 ist auf ein breites Spektrum und keine Nische trainiert [10]. Eine CPK im Finetuning-Ansatz wurde nicht durchgeführt. Die Zahlen zeigen eine Steigerung der Performance gegenüber der Binärklassifizierung im ICL-Ansatz. Auch der verbesserte F1-Score von GPT-3.5 mit 84.34% und von CodeLlaMa mit 82.26% weist auf einen Leistungssteigerung durch CPK hin [19].

Eine Studie zum Vergleich der Bug Detection zwischen LLMs und Studierenden verwendet die LLMs GPT-3 und

Tabelle II
PERFORMANCE VON GPT-3 UND GPT-4 BEI DER BUG DETECTION. KORREKT STEHT FÜR RICHTIG ERKANNTE BUGS (TRUE POSITIVE/TRUE NEGATIVE) UND INKORREKT STEHT FÜR FALSCH ERKANNTE BUGS (FALSE POSITIVE/FALSE NEGATIVE) [30]

Modell		Code Beispiel 1				Code Beispiel 2				Code Beispiel 3			
		Bug 1	Bug 2	Bug 3	Korrekt	Bug 1	Bug 2	Bug 3	Korrekt	Bug 1	Bug 2	Bug 3	Korrekt
GPT-3	Korrekt	30	23	8	30	29	27	24	29	29	30	24	12
	Inkorrekt	0	4	22	0	0	0	1	1	0	0	6	17
	Rate	1	0.852	0.267	1	1	1	0.96	0.967	1	1	0.800	0.414
GPT-4	Korrekt	29	30	28	19	30	30	28	0	30	30	28	0
	Inkorrekt	0	0	1	11	0	0	1	30	0	0	1	30
	Rate	1	1	0.966	0.633	1	1	0.966	0	1	1	0.966	0

GPT-4. Die präsentierten Bugs sind in C programmiert. Dabei handelt es sich um einen Operator Fehler (1), einen Expression Fehler (2) innerhalb einer if-Abfrage und einen out-of-bounds Fehler (3). Zwar stehen in dieser Studie die Leistungen der Studierenden gegen die Leistungen der LLMs im Vordergrund, dennoch sind die Daten zur Bug Detection der LLMs für dieses Dokument interessant und fließen Analyse mit ein. Des Weiteren wird in der Studie Prompt Engineering verwendet, um die Qualität der Antworten zu beeinflussen. Damit die Semantik der Antworten konvergieren verwenden die Autoren der Studie einen niedrigen Temperatur-Parameter [29]. Wie Tabelle II zeigt, schneiden beide Modelle bei allen drei Code Beispielen und vor allem bei den Bugs 1 und 2 sehr gut ab. GPT-4 zeigt bei der Bug Detection über das gesamte Feld eine überdurchschnittliche Performance. Allerdings zeigen die Zahlen auch, dass GPT-4 mit fehlerfreiem Code Probleme hat. Hier stuft das LLM den Code als fehlerhaft ein. Das liegt zum Teil daran, dass GPT-4 Inkonsistenten bei der Namenskonvention oder ein fehlendes Error handling als Fehler ansieht [30].

Eine neuere Studie [3] zur automatischen Analyse von statischen Bug Warnungen mit LLMs zeigt unter der Verwendung von ChatGPT (Version gpt-3.5-turbo-16k-0613) und LLama-2 (Version LLama-2-70b), dass LLMs in der Lage sind Bugs zu entdecken. Bemerkenswert ist, dass sowohl bei ChatGPT und LLama-2 mit der Juliet Test Suite (C/C++) im Bereich Use-after-free eine Präzision und ein Rückruf von 100.00% erreichen. Die Studie verwendet drei verschiedene statische Analysetools (Cppcheck, CSA und Infer), deren Ergebnisse den LLMs zur Bug Detection übergeben werden. ChatGPT zeigt eine hohe Präzision bei den Bugs Nullpointer (NP), Use-after-free (UAF), Divided by Zero (DBZ) und Uninitialized Variable (UVA). Die Präzision sinkt deutlich bei Memory Leaks (ML) und Buffer Overflow (BOF) Bugs mit Schwankungen zwischen 17% und 30% bei den statischen Warnungen aus dem Tool Cppcheck. Bei den Warnungen von CSA ist der schlechteste Wert bei ChatGPT im Bereich UVA Bugs [3]. Gleiches gilt für die Warnungen von Infer. Hier ist wiederum die Präzision bei BOF Bugs höher. Im Vergleich 87.00% Präzision bei Infer und 70.94% bei Cppcheck [3]. LLama-2 zeigt im Vergleich zu ChatGPT schlechtere Ergebnisse auf derselben Test Suite. Die Präzision von ML und BOF steigt geringfügig bei Cppcheck Warnungen, sinkt allerdings deutlich bei Infer Warnungen. Im Vergleich der allgemeinen Performance aus den Tabellen III und IV ist erkennbar, dass ChatGPT bessere Ergebnisse liefert, als Llama-2. Dabei ist die schlechteste Präzision von LLama-2 bei CSA Warnungen mit 66.03% und die beste Präzision von ChatGPT bei Infer Warnungen mit 85.25%. Dabei gilt: True Positive (TP) ist ein korrekt erkanntes positives Ergebnis, True Negative (TN) ein korrekt erkanntes negatives Ergebnis, False Positive (FP) ein fälschlich als positiv erkanntes negatives Ergebnis und False Negative (FN) ein fälschlich als negativ erkanntes positives Ergebnis. Des Weiteren zeigen die Spalten TP+TN und FP+FN bei den CSA Warnungen in der Tabelle III, dass entweder die Warnungen aus diesem Tool für LLMs nicht ausführlich genug beschrieben sind oder das LLM in diesem Bereich noch Schwächen aufweist und deshalb nur ein F1-Score von 70.71% erreicht. Verglichen mit den Ergebnissen von ChatGPT in Tabelle IV liegt die Vermutung nahe, dass das Llama-2 in diesem Bereich noch Schwächen aufweist, da ChatGPT bei den CSA Warnungen einen F1-Score von 85.25% erreicht.

Tabelle III
ALLGEMEINE PERFORMANCE VON LLAMA-2 ÜBER STATISCHE BUG WARNUNGEN, ERWEITERT UM DEN F1-SCORE [3]

Tool	**TP+TN**	**FP+FN**	**Präzision**	**Rückruf**	**F1-Score**
Cppcheck	1437	491	75.16%	90.66%	82.19%
CSA	894	460	66.03%	76.11%	70.71%
Infer	2516	1106	69.46%	74.80%	72.03%

Tabelle IV
ALLGEMEINE PERFORMANCE VON CHATGPT ÜBER STATISCHE BUG WARNUNGEN, ERWEITERT UM DEN F1-SCORE [3]

Tool	**TP+TN**	**FP+FN**	**Präzision**	**Rückruf**	**F1-Score**
Cppcheck	1507	421	78.16%	88.22%	82.89%
CSA	1009	345	74.52%	99.60%	85.25%
Infer	2976	646	85.25%	93.41%	89.14%

Tabelle V
AUSWIRKUNGEN VERSCHIEDENER PROMPTING METHODEN ZUR VERBESSERUNG DES F1-SCORE. MODELL: GPT-4 [21]

Kombination	**F1 Score**
Simple Prompt	0.13
PGA	0.31
PGA+PP	0.29
PGA+PP+SV	0.48
PGA+PP+TD	0.50
PGA+PP+TD+SV	0.93

Eine andere Studie [21] in Bezug auf die Verwendung von GPT-4 (gpt-4-0613) im Bereich von statischen Code Analysen zeigt einen verbesserten F1-Score durch schrittweise zunehmende Aufbereitung der Prompts. Zuerst werden einfache Prompts an das LLM gesendet. In jedem weiteren Versuch wird eine Stufe der geführten Prompts ergänzt:

- **Post-Constraint Guided Path Analysis (PGA)**: Durch ICL erlernt das LLM die Post-Constraints-Erkennung, also Bedingungen, die nach Ausführen der Funktion erfüllt sein müssten.
- **Progressive Prompt (PP)**: Durch ICL die Fähigkeit erlernen, nach fehlenden Funktionsdefinitionen zu fragen, wenn notwendig.
- **Self-Validation (SV)**: Das LLM soll seine eigene Antwort überprüfen.
- **Task Decomposition (TD)**: Zerlegung der Aufgaben für das LLM in mehrere Konversationen. Bspw. soll das LLM zuerst eine Antwort auf die Aufgabe in gewünschter Sprache formulieren und danach eine Zusammenfassung als JSON zur Verfügung stellen, damit die eine Automatisierung der Aufgaben stattfinden kann.

Durch die verschiedenen Methoden und steigenden Informationen bei der Ausführung der Aufgaben steigt der F1-Score von GPT-4 von 0.13 auf 0.93 (vgl. Tabelle V).

C. *Herausforderungen der Bug Detection*

Bei der Bug Detection stehen LLMs vor verschiedenen Herausforderungen. Bisher weist jede untersuchte Studie darauf hin, dass die Performance bei realen Daten schlechter ist als bei künstlich erzeugten Daten [19], [31]. Dies liegt unter anderem daran, dass die Trainingsdaten der LLMs häufig künstlich erzeugt wurden und aus Permutationen des fehlerfreien Codes entstanden sind [32]. Des Weiteren weisen reale Trainingsdaten meist eine geringe Größe auf oder sind nicht ausreichend vorhanden [33].

Eine weitere Herausforderung ist, dass die Konfiguration der LLMs Auswirkungen auf die Semantik der Aussagen des Modells hat. ChatGPT verwendet hierfür zum Beispiel einen Temperature-Parameter. Je höher diese Zahl, desto "kreativer" generiert ChatGPT in diesem Fall [29], [34]. Hier kommt es darauf an, ob bei der Verwendung überhaupt Einfluss auf diesen Parameter genommen werden kann. Auch wenn dieser Parameter steuerbar ist, kann es in Bereichen in denen wenig Trainingsdaten vorliegen, zu unterschiedlichen Aussagen der LLMs auf dieselbe Frage kommen [26], [30], [35].

Die untersuchten Studien [3], [19], [21], [30] haben gezeigt, dass wenigstens eine Art von Prompt Engineering implementiert werden sollte. Andernfalls fällt der F1-Score oder die Zahl der falsch positiven Ergebnisse zu hoch aus. Je höher der Anteil der falsch Positiven, desto eher erreicht das LLM eine Performance von 50% bei der Erkennung und kommt somit dem Zufallsprinzip gleich.

IV. Methode

Im Umfeld der Bug Detection mit LLMs wurden bereits mehrere empirische Studien wie im Abschnitt III-B beschrieben durchgeführt. Daher setzt diese Arbeit den Fokus auf die Dokumenten- und Literaturrecherche und deren Evaluation.

A. *Fragestellung*

Dabei stellt sich zunächst die Frage, ob Large Language Models in der Lage sind Bugs zuverlässig zu erkennen. Da Zuverlässigkeit eher subjektive bewertet wird, werden verschiedene Benchmarks der Studien untersucht, die die Performance der verwendeten LLMs bewerten und vergleichbar machen. Eine 100%-ige Erkennungsrate wäre nicht realistisch. Alles unterhalb der 50% ist schlechter als Raten und zwischen 50% - 80% besteht Verbesserungspotential. Insofern sollten die Benchmarks hierbei die 50%-Marke überschreiten und im besten Fall über 80% liegen.

B. *Evaluation der Recherche*

Die Evaluation der Dokumenten- und Literaturrecherche zeigt, dass die Aussagekraft der LLMs in der Bug Detection von mehreren Faktoren abhängig ist. Basierend auf den Ansätzen des Prompt Engineerings und den darunterliegenden Pretrained Language Model sowie dem weiteren Finetuning des Modells, können die Erfolgsquoten der LLMs variieren. Da LLMs unterschiedliches pre-training und finetuning durchlaufen, muss eine Vergleichbarkeit über Benchmarks hergestellt werden. Metriken, die in den Studien verwendet wurden, waren:

- F1-Score
- True Positives / False Negatives
- Datensatzqualität
- Vielfalt der Bugs im Trainingsset

Zur Berechnung des F1-Scores muss zunächst die Rate der richtig Positiven unter allen Positiven definiert sein (vgl. Gleichung 1). Mit den realen Positiven kann der Rückruf durch Gleichung 2 errechnet werden [36].

$$\text{Reale Positive} = \text{True Positives} + \text{False Negatives} \quad (1)$$

$$\text{Rückruf} = \frac{\text{True Positives}}{\text{Reale Positive}} \quad (2)$$

Die Präzision definiert durch Gleichung 3 errechnet dabei die Rate der positiven unter den ausgegebenen richtig und falsch positiven Meldungen.

$$\text{Präzision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \quad (3)$$

Der F1-Score definiert durch Gleichung 4 setzt die Präzision und den Rückruf in Relation. Dabei ist zu beachten, je höher der F1-Score, desto effektiver ist das zugrundeliegende LLM in Bezug auf die gestellte Aufgabe mit den verwendeten Ansätzen [20].

$$F1 = 2 \cdot \frac{\text{Präzision} \cdot \text{Rückruf}}{\text{Präzision} + \text{Rückruf}} \quad (4)$$

Neben dem F1-Score können die zur Berechnung verwendeten True Positives und False Positives verwendet werden, um eine erste Analyse zur Performance des LLMs durchzuführen.

Des Weiteren hat die Datensatzqualität der verwendeten Trainingsdaten Einfluss auf die Präzision des LLMs. Dabei lässt sich die Qualität der Daten nur schwer messen. Daher sollte die Vielfalt der enthaltenen Bugs im Trainingsset eine weitere Metrik zur Datensatzqualität sein. Je mehr Bugs ein LLM in der Trainingsphase im Finetuning evaluiert hat, desto effektiver wird die Performance. Die in Abschnitt III-B genannten Herausforderungen zeigen direkten Einfluss der Trainingsdaten auf die Ergebnisse. Dabei zeigt Bug Detection in realen Softwareprodukten eine schlechtere Performance der LLMs [33].

Programmiersprachen dienen verschiedenen Zwecken, daher gilt es bei der Vergleichbarkeit der Performance dies zu berücksichtigen. Die am häufigsten angetroffene Programmiersprache war C gefolgt von Python. Für einen aussagekräftigen Vergleich der Performance ist es wichtig, dass die verwendete Programmiersprache dieselbe ist oder ähnliche Funktionsweisen abdeckt.

Neben der Programmiersprache sind die Anzahl an Parameter eines Modells und der verwendete Trainingsprozess ein Aspekt für die Performance der Bug Detection. Allerdings werfen Anzahl an Parameter und Trainingsprozess des LLMs weitere Fragen auf. Zum Beispiel können LLMs mit einer geringeren Anzahl an Parametern (CodeLlaMa [25]), dafür mit einem speziellen Finetuning auf Bug Detection, besser abschneiden als LLMs wie ChatGPT [10] mit einer größeren Anzahl an Parametern und einem allgemeineren Finetuningprozess.

Unter der Verwendung der Benchmarks zum Vergleich der Ansätze zeigen die verschiedenen Ansätze eine effektive Performance in den Bereichen der Bug Detection bei fehlerhaftem Code (siehe Tabelle II). Bei korrektem Code sinkt die Performance, da ChatGPT diesen teilweise als fehlerhaft einstuft. Des Weiteren zeigt der F1-Score der jeweiligen Studien, dass die Verwendung von verschiedenen kombinierten Ansätzen bessere Ergebnisse liefert. Die in der Fragestellung definierte 80%-Marke wird mit dem verwendeten LLM ChatGPT erreicht (siehe Tabellen I, II, IV und V). Llama-2 liegt mit dem Cppcheck Tool oberhalb der 80%, ansonsten unterhalb dieser Marke (siehe Tabelle III). Allerdings wurden mit Llama-2 weniger Methoden zur Bug Detection getestet.

V. Diskussion

Die in den Studien betrachteten LLMs bezogen sich auf eine Version von ChatGPT (verwendet in allen vier untersuchten Studien), nur zwei der vier untersuchten Studien haben Open Source LLMs mit einbezogen. Die Vergleichbarkeit der Modelle ist dadurch erschwert ebenso das Ausfindigmachen der Stellschrauben für eine verbesserte Performance. Allerdings lässt sich durch etablierte Software wie ChatGPT mit API ein schnelleres und nachstellbares Ergebnis erzeugen. Hingegen ist bei Open Source Modellen, die eigenverantwortliche Administration benötigen, das Erzeugen solcher Ergebnisse schwieriger und nimmt mehr Zeit in Anspruch. Zudem sind hohe Anforderungen an die Hardware gegeben.

Ein weiterer Aspekt bei Closed und Open Source ist der Einfluss auf das Finetuning. Die Zahlen aus Tabelle II zeigen, dass zwischen den Modellen GPT-3.5 und GPT-4 die Erkennung von fehlerfreiem Code als fehlerhaftem Code ansteigen. Dies lässt vermuten, dass bei einem Versionswechsel von Closed Source LLMs die Vorgehensweise zur Bug Detection angepasst werden muss. Dadurch können Junior Entwickler neue Skills und Konventionen erwerben [30], allerdings kann im Senior oder Legacy Code Umfeld eine Nichteinhaltung der Namenskonvention als zweitrangig angesehen und nicht als Bug eingestuft werden. Abhängig von der Intention der jeweiligen Person.

Wie in den Herausforderungen erwähnt, ist ein Trainingsset mit realen Daten in der für ein LLM notwendigen Größe schwer zu finden. Die Studien verwenden Test Suites oder Bibliotheken wie PyPIBugs. Diese bestehen nicht nur aus realen Bugs, sondern aus einer Kombination aus künstlich erzeugten und realen Bugs. Des Weiteren erhielten LLMs in den Studien bereits von Statischen Code Analysetools potentielle Bugs. Dadurch kann eine gewisse Voreingenommenheit des Modells bei der Analyse entstehen. Je nach Beschreibung des potentiellen Bugs durch das Analysetool und den übergebenen Kontext.

Ebenso zeigt der Ansatz der Code Pair Klassifizierung gegenüber der Binärklassifizierung zwar einen Anstieg in der Performance, dieser kann dennoch durch die einfachere Aufgabe im Ansatz der Code Pair Klassifizierung liegen. Dem LLM wird bereits bei der Aufgabe eine 50/50 Chance zur Auswahl gegeben. Ein Vergleich der Performance von Code Pair Klassifizierung mit Binärklassifizierung scheint im gegebenen Kontext der Bug Detection als unfair. In der Binärklassifizierung ist zu entscheiden, ob der vorliegende Code fehlerhaft ist. In der Code Pair Klassifizierung ist zu entscheiden, welches der beiden Codestücke fehlerhaft ist. Somit ist bei korrekter Auswahl bereits die Lösung vorhanden und das LLM kann dementsprechend argumentieren. Dennoch gibt es Anwendungsbereiche, in denen dieses Verhalten sogar gewünscht sein kann. In Abschnitt II-A sind die verschiedenen Arten von Bugs näher erläutert, unter anderem Regression Bugs. In diesem Fall gibt es fehlerhaften Code in der neuen Version der Software und die bereits korrigierte Version in der alten Version der Software.

Im Vergleich zu den bisherigen Methoden der Bug Detection sind die LLM gestützten Ansätze eher proaktiv als reaktiv. Dadurch besteht Potential zu einer Neuerung und Verbesserung der Software-Wartung und -Pflege.

Die Zahlen deuten ebenfalls auf einen Einsatz von LLMs zur Bug Detection hin. Auch wenn die vorhin genannten Schwächen existieren, zeigen LLMs bemerkenswerte Zahlen in einer breiten Masse an Bugs. Des Weiteren gilt zu erwähnen, dass weder ChatGPT noch LLama-2 speziell darauf trainiert sind Bug Detection zu beschreiben. Dies belegt wieder die Effektivität von in-context-learning Methoden und Prompt Engineering. Diese beiden Ansätze wurden auch in unterschiedlichsten Formen in den untersuchten Studien durchgeführt. Von einem Lean-Ansatz durch unterschiedliche Formu-

lierungen der Prompts zu aufeinander aufbauenden Prompting Strategie mit Selbstvalidierung durch das LLM. Der erweiterte Ansatz des Prompt Engineerings belegt seine Effektivität in den Zahlen. Die durchgeführten Tests mit aufbauenden Prompts steigert bei jeder hinzugefügten Methode den F1-Score des LLMs.

Ein weiterer relevanter Aspekt der Bug Detection ist, inwiefern es möglich ist das Verhalten der Anwender durch LLMs in Bezug auf die Software nachahmen zu können. Dadurch könnten explorative Userinterface Tests mit menschenähnlichem Verhalten durchgeführt werden. Diese Untersuchung ist kein Bestandteil der Arbeit.

Ebenso nicht Bestandteil der Arbeit, aber erwähnenswert, ist die Performance von ChatGPT in der Studie zur Bug Detection zwischen Novizen in der Software Entwicklung und LLMs. Die Studie kommt zu dem Schluss, dass Novizen in der Software Entwicklung korrekten Code besser erkennen als ChatGPT. Allerdings ist in der Fragestellung, die an die Studierenden gerichtet wird eine natürliche Voreingenommenheit verankert. Die Studierenden erhalten als Aufgabe ein Stück Code mit der Aufgabenstellung zu prüfen, ob der Code fehlerhaft ist und wenn ja wie dieser zu beheben ist. Die Unwissenheit der Person kann dadurch ein richtiges Ergebnis erzielen, da eine Option für "Proband weiß die Antwort nicht" nicht existiert. Somit schneidet ChatGPT schlechter ab als dargestellt. Eine Kontrollfrage bei positivem Ergebnis wie zum Beispiel welchen Output liefert der Code könnte dieses Problem beheben. Nichtsdestotrotz schneidet ChatGPT in dieser Studie bei korrektem Code schlecht ab. Diese Erkenntnisse sollten in Prompts für ChatGPT-4 mit einfließen und das LLM dahingehend sensibilisieren. Korrekturen sind wünschenswert, aber der Code ist nicht als fehlerhaft einzustufen.

VI. Fazit

Der Einsatz von LLMs in der Software-Wartung und -Pflege ist durchaus denkbar mit den vorliegenden Zahlen der Studien. Diese weisen auf eine Verwendung von LLMs gepaart mit verschiedenen Methoden hin (in context learning, Prompt Engineering oder finetuning). Die Modelle sind nicht speziell darauf trainiert worden Bug Detection zu betreiben, daher zeigen die Zahlen bei Prompting ohne Verwendung dieser Methoden schlechtere Ergebnisse.

Für eine mögliche Vorgehensweise zur Integration eines LLMs in die Software-Wartung und -Pflege könnte sein, dass das LLM Zugang zum Quellcode über eine Art Code/ Repository Scanner erhält. Dadurch kann das LLM die Zusammenhänge der Software verstehen und in die Bearbeitung der Aufgabe einfließen lassen. Ähnlich wie bei Statischen Code Analyse Tools. Dieser Scanner ist direkt mit dem LLM verbunden und kann über eine API kommunizieren. Dabei muss der Scanner konfigurierbar sein für die Erweiterung oder Verbesserung der eingesetzten Methoden wie zum Beispiel in context learning oder Prompt Engineering. Durch diese Konfigurationsmöglichkeiten kann das Prompting an das LLM jederzeit angepasst werden und damit direkten Einfluss auf das Ergebnis der Analyse nehmen.

Zur Verwendung des zugrundeliegenden LLMs ist es ratsam, dass diese auf die Aufgabe Bug Detection, die verwendeten Programmiersprachen und bestehenden Code trainiert werden. Ist dies nicht möglich, da ein Anbieter verwendet wird, sollte darauf mit in context learning Verfahren eingegangen werden. Ein Vorteil des Trainings des LLMs auf den bestehenden Code kann sein, dass das LLM Muster der Entwickler erkennt und Bugs ableiten könnte. Diese Annahme muss noch überprüft werden. Bei einem LLM von einem Anbieter sollte dennoch in Betracht gezogen werden, welche Kosten für die übermittelten Token per Prompt anfallen. Diese können je nach Größe der Softwareprojekte und zu übermittelnde Information an das LLM unterschiedlich ausfallen.

Abgesehen von der Trainingsmethode sollte das LLM potentielle Bugs mehrfach vorgelegt bekommen und Selbstvalidierung betreiben, damit die Zahl der falsch positiven verringert werden kann.

Eine weitere Überlegung ist, das LLM frühzeitig in den Entwicklungsprozess zu integrieren und bereits dort bei der Bug Detection aushelfen zu lassen. Weniger Bugs im Quellcode bedeuten weniger Bugs in der ausgelieferten Software, die in den Software-Wartungsprozess übergeht. Damit verringert sich als positiver Nebeneffekt der Einsatz der LLMs in diesem Bereich. Eine komplette Vernachlässigung und Vermeidung von LLMs in der Software-Wartung und -Pflege sollte nicht das Ziel sein. Trotzdem steigt durch das frühe Einbinden der LLMs die Qualität der ausgelieferten Software.

Zusammenfassend sind LLMs für die Bug Detection in der Software-Wartung und -Pflege geeignet und können zuverlässige Ergebnisse liefern, wenn die unterstützende Methoden für die Verbesserung der Ergebnisse in Betracht gezogen werden. Ein Ansatz ähnlich der statischen Code Analyse kann eine vielversprechende Möglichkeit zur Verbesserung im Bereich der Bug Detection bieten.

Literatur

[1] A. Alaboudi and T. LaToza, "What constitutes debugging? an exploratory study of debugging episodes," *Empir Software Eng*, vol. 28, 2023, angenommen: 30.05.2023, Veröffentlicht: 11.09.2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10352-5

[2] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, 2010, pp. 185–194.

[3] C. Wen, Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and C. Tian, "Automatically inspecting thousands of static bug warnings with large language model: How far are we?" *ACM Trans. Knowl. Discov. Data*, mar 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3653718

[4] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2021.

[5] N. Rutar, C. Almazan, and J. Foster, "A comparison of bug finding tools for java," in *15th International Symposium on Software Reliability Engineering*, 2004, pp. 245–256.

[6] P. Louridas, "Static code analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, 2006.

[7] S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer, "An empirical study on usage and perceptions of llms in a software engineering project," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.16186

[8] Meta, "Llama 2," 2023, zugriff am: 23.05.2024. [Online]. Available: https://llama.meta.com/llama2/

[9] Microsoft, "Codebert license," 2023, zugriff am: 23.05.2024. [Online]. Available: https://github.com/microsoft/CodeBERT/blob/master/LICENSE

[10] OpenAI, "Openai platform documentation," 2023, zugriff am: 23.05.2024. [Online]. Available: https://platform.openai.com/docs/introduction

[11] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021.

[12] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990, autor unbekannt.

[13] G. Xiao, Z. Zheng, B. Jiang, and Y. Sui, "An empirical study of regression bug chains in linux," *IEEE Transactions on Reliability*, vol. 69, no. 2, pp. 558–570, 2020.

[14] SonarSource SA, *SonarQube 10.5 Documentation*, 2024, zugriff am: 29.05.2024. [Online]. Available: https://docs.sonarsource.com/sonarqube/latest/

[15] M. Kamran, M. Ali, and A. Ahmed, "Generating suggestions for initial program investigation using dynamic analysis," in *2017 International Conference on Communication, Computing and Digital Systems (C-CODE)*, 2017, pp. 233–237.

[16] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "Jcharming: A bug reproduction approach using crash traces and directed model checking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 101–110.

[17] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[18] A. R. Chen, T.-H. Chen, and S. Wang, "Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2905–2919, 2022.

[19] K. Alrashedy and A. Binjahlan, "Language models are better bug detector through code-pair classification," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2311.07957

[20] D. M. W. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2020.

[21] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, apr 2024. [Online]. Available: https://doi.org/10.1145/3649828

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2201.11903

[23] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, L. Li, and Z. Sui, "A survey on in-context learning," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2301.00234

[24] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," 2018.

[25] Meta AI, "Introducing code llama, a state-of-the-art large language model for coding," 2023, zugriff am: 29.05.2024. [Online]. Available: https://ai.meta.com/blog/code-llama-large-language-model-coding/

[26] C. Si, Z. Gan, Z. Yang, S. Wang, J. Wang, J. Boyd-Graber, and L. Wang, "Prompting gpt-3 to be reliable," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2210.09150

[27] S. Arora, A. Narayan, M. F. Chen, L. Orr, N. Guha, K. Bhatia, I. Chami, and C. Re, "Ask me anything: A simple strategy for prompting language models," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=bhUPJnS2g0X

[28] OpenAI, "Openai prompt engineering guide," 2023, zugriff am: 23.05.2024. [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering

[29] ——, "How should i set the temperature parameter? - openai api documentation," 2024, zugriff am: 21.05.2024. [Online]. Available: https://platform.openai.com/docs/guides/text-generation/how-should-i-set-the-temperature-parameter

[30] S. Macneil, P. Denny, A. Tran, J. Leinonen, S. Bernstein, A. Hellas, S. Sarsa, and J. Kim, "Decoding logic errors: A comparative study on bug detection by students and large language models," in *Proceedings of the 26th Australasian Computing Education Conference*. New York, NY, USA: Association for Computing Machinery, 2024, p. 11–18. [Online]. Available: https://doi.org/10.1145/3636243.3636245

[31] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2022.

[32] C. Richter and H. Wehrheim, "How to train your neural bug detector: Artificial vs real bugs," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1036–1048.

[33] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 27 865–27 876. [Online]. Available: https://openreview.net/forum?id=zOngaSKrElL

[34] OpenAI, "Create transcription - openai api documentation," 2024, zugriff am: 21.05.2024. [Online]. Available: https://platform.openai.com/docs/api-reference/audio/createTranscription

[35] L. Gao, Z. Dai, P. Pasupat, A. Chen, A. T. Chaganty, Y. Fan, V. Y. Zhao, N. Lao, H. Lee, D.-C. Juan, and K. Guu, "Rarr: Researching and revising what language models say, using language models," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2210.08726

[36] R. Yacouby and D. Axman, "Probabilistic extension of precision, recall, and f1 score for more thorough evaluation of classification models," in *Proceedings of the First Workshop on Evaluation and Comparison of NLP Systems*, S. Eger, Y. Gao, M. Peyrard, W. Zhao, and E. Hovy, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 79–91. [Online]. Available: https://aclanthology.org/2020.eval4nlp-1.9