

TECHNISCHE HOCHSCHULE KÖLN

PRAXISPROJEKT

Anwendungspotenzial von LLMs in der ABAP-Entwicklung am Beispiel von ChatGPT

Stephan Wallraven 11147184

Tim Köhne 11147022

Betreut durch
Prof. Dr. Hartmut Westenberger

Zusammenfassung

Diese Ausarbeitung zeigt dass ChatGPT nicht für die Codegenerierung in ABAP geeignet ist, es konnte nur 2.45% der 16400 Aufgaben lösen. Die Ergebnisse der Codegenerierung lassen sich durch Anpassungen der Parameter kaum verbessern. Wir vermuten das dies der Fall ist, da es zu wenig Trainingsdaten von ABAP-Code gibt. Die Version GPT-4 erzielte noch schlechtere Ergebnisse als die ältere Version GPT-3.5.

ChatGPT eignet sich jedoch gut für Codeerklärungen und zum Verständnis von Code. Hier wurde der ABAP-Code in nahezu allen Testfällen korrekt erklärt.

Weitere umfangreichere Tests sind erforderlich, um mehr Parameterkombinationen und deren Einfluss auf die Programmentwicklung zu untersuchen.

5. November 2023

Inhaltsverzeichnis

1	Einleitung und Überblick	3
1.1	Einleitung	3
1.2	Zielsetzung	4
1.3	Vorgehensweise	4
1.4	Begriffsklärung	4
1.4.1	SAP	4
1.4.2	ABAP	5
1.4.3	LLMs	5
1.4.4	ChatGPT	7
1.5	Kategorisierung von Softwareentwicklungsaufgaben und Abschätzung ihrer Tauglichkeit zur Unterstützung durch LLMs	7
1.5.1	Software Development Life Cycle Phasen	7
1.5.2	Rollenbezug	9
1.6	Wissenschaftliche Literatur zur KI-Unterstützten Softwareentwicklung	9
1.6.1	Programmieren lernen mit LLMs	9
1.6.2	LLM Benchmarking	13
1.7	Datensicherheit und die Nutzung von LLMs	15
2	Vorbereitung des Benchmarks	16
2.1	Wie sind LLM Software-Generierungsbenchmarks aufgebaut?	16
2.2	Welche Anforderungen haben wir an unseren Benchmark?	17
2.2.1	Welche Anforderungen entstehen durch ABAP?	17
2.2.2	Welche Anforderungen entstehen durch ChatGPT?	18
2.3	Relevante Daten und ihre Messbarkeit	18
2.3.1	Parameter	18
2.3.2	Kennzahlen	19
2.4	Automatisierte Tests	20
2.4.1	Versuchsaufbau	20
2.4.2	Testvorbereitung	21
2.4.3	Testplanung: Parameter finden	23
2.4.4	Testplanung: Funktionsbausteine generieren	24
2.4.5	Testplanung: Unterschiede deutsche und englische Prompts	24
2.4.6	Testplanung: GPT-4	25
2.5	Manuelle Tests	25
3	Durchführung des Benchmarks	26
3.1	Automatische Tests	26
3.1.1	Testergebnis: Parameter finden	26
3.1.2	Testergebnis: Funktionsbausteine generieren	28
3.1.3	Testergebnis: Unterschiede deutsche und englische Prompts	30
3.1.4	Testergebnis: GPT-4	30
3.1.5	Zusammenfassung: Automatische Tests	31
3.2	Manuelle Tests	31
3.2.1	Testergebnis: Syntaxfehler finden	32
3.2.2	Testergebnis: Codeerklärung	33
3.2.3	Testergebnis: Korrekte Codeerkennung	34
3.2.4	Zusammenfassung: Manuelle Tests	35
3.3	Hypothese(n) zur Begründung der Ergebnisse	35
4	Fazit und Ausblick	37
4.1	Fazit	37
4.2	Ausblick	38
6	Anhang	43

Abbildungsverzeichnis

1	Autoregressives Pretraining: Kontext wirkt nur in eine Richtung [DJ23]	6
2	Maskiertes Pretraining: Kontext wirkt in beide Richtungen [DJ23]	6
3	Software Development Life Cycle [RTK22]	8
4	Rollen und die Bereiche ihrer Verantwortungsbereiche [RTK22]	10
5	Beispiel für eine schlechte Aufgabe[Tli+23]	11
6	Beispiel für eine gelungene Aufgabe[Bec+23]	12
7	Pass@K vs K, Temperature von Codex in Python [Che+21]	14
8	Versuchsaufbau: Antwortgenerierung durch ChatGPT	21
9	Versuchsaufbau: Ausführung der Antworten in ABAP	22
10	Beispielhafte Darstellung vom Prompt 0 des HumanEval Datensatzes	23
11	Aufbau unseres Benchmarks	26
12	Anzahl erfolgreicher Testdurchläufe je Parameterkombination	27
13	Summe der Ergebnisse je Parameter	27
14	Ergebnis der Funktionsaufrufe von allen Prompts mit 100 Wiederholungen	28
15	Vergleich unserer Ergebnisse mit Codex in Python [Che+21] und ChatGPT in Python [Liu+23]	29
16	Liste der aufgetreten Exceptions. Es handelt sich überall um ABAP RunTimeError außer wenn explizit ApplicationError oder ExternalRuntimeError davor steht.	30
17	Anteil der Antworten die ein korrektes Interface-Kommentar im ABAP-Funktionsbaustein enthalten	31
18	Ergebnisse der Fehlererklärung des Fehlerfindungstests	32
19	Ergebnisse der Fehlerbehebung des Fehlerfindungstests	33
20	Ergebnisse des Codeerklärungstests	34
21	Ergebnis des korrekten Codeerkennungstests	35
22	Vergleich des weltweiten Suchinteresse von Python und ABAP von 2004 bis September 2023 von Google Trends	36
23	Vergleich des durchschnittlichen weltweiten Suchinteresse von Python und ABAP seit 2004 bis September 2023 von Google Trends	36
24	Musterlösung zu Prompt 11 der Manuellen Tests	43
25	Musterlösung zu Prompt 15 der Manuellen Tests	43
26	Musterlösung zu Prompt 157 der Manuellen Tests	44

1 Einleitung und Überblick

1.1 Einleitung

Künstliche Intelligenz ist momentan ein sehr relevantes und umstrittenes Thema, ob es um die Analyse von Daten zu medizinischen oder ingenieurstechnischen Zwecken, die Bildgeneration von Systemen wie DALL-E 2 oder Midjourney, oder um Deepfakes mit Video und Audio geht. Aber das aktuell am meisten diskutierte Produkt in dem Bereich ist ChatGPT, nur 5 Tage nach Veröffentlichung hatte es eine Millionen Nutzer, nach 2 Monaten über 100 Millionen. ChatGPT ist für die meisten Menschen der erste direkte Kontakt mit künstlicher Intelligenz und viele Leute sind begeistert von dessen Fähigkeiten. Es wird für alle möglichen Aufgaben genutzt: von persönlichen Rezeptempfehlungen oder Geschenkvorschlägen, über das Verfassen von Emails und Briefen für persönliche oder professionelle Zwecke aber auch zur Datenanalyse von Unternehmensdaten. Insgesamt sollen über 10% der Arbeitnehmer weltweit ChatGPT schon einmal an ihrem Arbeitsplatz genutzt haben. [Col23].

Gleichzeitig gibt es auch weit verbreitete Bedenken um die Nutzung von Künstlicher Intelligenz. Taxi- und Lastkraftwagenfahrer fühlen sich schon länger durch selbstfahrende Systeme wie Tesla Autopilot bedroht, und ChatGPT hat diese Angst noch weiter verbreitet weil es so vielfältig ist. Goldman Sachs vermutet generative künstliche Intelligenz wie ChatGPT könnte bis zu 300 Millionen Arbeitskräfte ersetzen [23n], was für viel Diskussion gesorgt hat, nachdem internationale Nachrichten darüber berichtet haben

Ein weiterer Aspekt, welcher die Gesellschaft gerade im Bereich der Lehre in Bezug auf generative Künstliche Intelligenzen wie ChatGPT beschäftigt, ist die Frage, wie die moderne Lehre in der Zukunft aussehen wird. Grund dafür ist, dass man nun mithilfe von einer generativen KI jegliche Art von Aufgaben wie zum Beispiel Coding Aufgaben ohne viel Hintergrundwissen erfolgreich bearbeiten kann, was einerseits ein großes Risiko aber auch große Chancen mit sich bringt. Dabei ergibt sich das Problem, dass einerseits nicht mehr wirklich bestimmt werden kann, wie viel wirklich durch Schüler selbst erledigt worden ist und wie viel die generative KI übernommen hat. Durch diese Aspekte ergibt sich die Frage, wie nun das Lernen mit diesen generative KIs gestaltet wird, ob man Tools entwickelt, welche erkennen ob KI genutzt wurden sind, und man somit gegen die Technik arbeitet oder ob man diese in das Lernen und Lehren integriert, die Aufgaben anpasst und man zum Beispiel Prompt Engineering lehrt. Ein Positiver Aspekt zu ChatGPT als Lernassistent wäre, dass man am Arbeitsplatz auch nicht alle Informationen auswendig kennen muss, stattdessen werden diese bei Bedarf nachgeschaut um korrekte und aktuelle Informationen zu erhalten.

Im Bezug dazu könnte es Sinn machen während der Lehre schon ein Tool zu nutzen, womit man gezielt durch Prompts auf eine Frage eine meist gute Antwort bekommt.

Ein weiteres Problem, welches neben dem erledigen von Aufgaben existiert, ist die Frage, wie es mit dem Urheberrecht von KI erstellten Werken aussieht. Erstellt eine generative KI einen Text greift sie dort auf die Trainingsdaten zu einem bestimmten Thema zu. Diese Trainingsdaten bestehen aber teilweise auch aus urheberrechtlich geschützten Werken, weshalb es dort zu Schwierigkeiten kommen könnte. Weitere Risiken im Umgang mit KI könnte auch der Datenschutz sein, denn wenn man zum Beispiel ChatGPT etwas fragt wobei private Daten wie Geburtsdatum, Adresse etc. mit angegeben wird, werden diese von OpenAI also dem Unternehmen, welches ChatGPT zur Verfügung stellt mit abgespeichert.

Anhand dieser direkt erkennbaren Risiken haben bereits manche Schulen in den USA ChatGPT verboten, da man dort zu viel Potenzial sieht, um bei generellen Aufgaben oder auch Tests zu betrügen. Letztlich bleibt abzuwarten ob dieses Verbot bestand hat, denn wie bereits erwähnt kann es auch positive Aspekte mit sich bringen ChatGPT ins Lernen zu integrieren und andererseits wird es schwierig sein, aktuell zu erkennen, ob die Werke von einer KI geschrieben worden sind. Ebenfalls zu beachten ist, dass die Antworten einer generative KI nur so gut sind wie die Trainingsdaten die ihr zur Verfügung standen. Das bedeutet, dass gerade bei spezifischen Fragen aus bestimmten Themengebieten, die Trainingsdaten nicht wirklich ausreichend sein könnten und gegebenenfalls dadurch falsche Antworten und somit falsches Wissen vermittelt werden könnte.

In der Softwareentwicklung wird ChatGPT vergleichsweise viel eingesetzt, es wird zur Generierung, Analyse und Erklärung von Code genutzt. Außerdem hilft es Programmiersprachen, Frameworks oder Libraries zu lernen und Implementierungsbeispiele zu geben oder Alternativen

vorzuschlagen. Entwickler nutzen schon lange Werkzeuge wie IDEs mit Syntax-Highlighting, integrierter Dokumentationsansicht, Autovervollständigung und automatischer Generierung von Hilfsfunktionen und Vorlagen um ihre Arbeit zu erleichtern. In 2021 wurde Github Copilot [Cit23a] als erste generative künstliche Intelligenz veröffentlicht um bei der Codegenerierung direkt in der IDE zu helfen. ChatGPT ist zwar nur über die Webseite (oder die API) erreichbar, wird aber trotzdem von vielen Entwicklern für die gleichen Aufgaben genutzt und scheint gute Ergebnisse zu liefern.

1.2 Zielsetzung

Mit diesem Projekt verfolgen wir mehrere Ziele. Unser Hauptziel ist die Entwicklung eines Benchmarks, um die Unterstützungsfähigkeiten von LLMs bei der ABAP Softwareentwicklung objektiv messen zu können. Wichtig ist hier, dass sich diese Benchmarks in einer einheitlichen und somit gleichbleibenden Umgebung abspielen, damit die resultierenden Testergebnisse nicht durch äußere Einflüsse verfälscht werden. Auf diese Testergebnisse bezieht sich das nächste Ziel, denn anhand der Testergebnisse wollen wir die Effektivität von ChatGPT bei der ABAP-Programmentwicklung bewerten und messen.

Zudem wollen wir untersuchen, inwiefern diese Testergebnisse durch verschiedene Einflussfaktoren variieren. Diese Einflussfaktoren sind zum Beispiel die Sprache, in der die Prompts gestellt werden oder auch Unterschiede in den Versionen von LLMs, in welcher die Prompts getestet werden. Des weiteren ist unser Ziel, am Ende der Auswertungen ein eindeutiges Fazit ziehen zu können, wie sinnvoll nun ChatGPT aktuell bei der Unterstützung von Softwareentwicklung in ABAP ist.

Abschließend wollen wir neben dem Fazit eine Prognose treffen können, inwiefern Systeme wie z.B. ChatGPT bei der generellen Softwareentwicklung, unabhängig von der Programmiersprache in der Zukunft eingesetzt werden können und welche Verbesserungen es gegebenenfalls geben könnte.

1.3 Vorgehensweise

Da Large Language Models noch relativ neu sind, werden wir zunächst mit Begriffsklärungen und einer Übersicht über ihre Funktionsweise anfangen. Dann werden wir die Bestandsliteratur durchsuchen, um gegebenenfalls bereits bestehende Erkenntnisse mit einbeziehen zu können und eine Redundanz zu verhindern. Als zweiten Schritt werden wir die verschiedenen Aufgaben in der Softwareentwicklung kategorisieren bzw. einordnen. Die Einordnung erfolgt hier, indem wir uns anschauen, in welchen Bereichen der Softwareentwicklung die Unterstützung durch LLMs überhaupt sinnvoll ist bzw. wo sie sich eher nicht eignet. Beispielhaft für die genannten Aufgaben sind: Anforderungs-ermittlungen, Softwaretests, Problemlösung oder Softwaredokumentation, wobei diese wahrscheinlich in sehr unterschiedlichen Wegen von LLMs unterstützt werden können. Daraufhin werden wir unsere Benchmarks vorbereiten, indem wir untersuchen, wie Benchmarks zur Softwareentwicklung normalerweise aufgebaut sind, unsere Anforderungen an unsere Benchmarks definieren und dann untersuchen, wie wir von ChatGPT repräsentative Antworten erhalten können. Daraufhin werden wir den Benchmark in ABAP erstellen, damit wir eine einheitliche und standardisierte Analyse durchführen können. Diesen Benchmark entwickeln wir, um daraus quantitative Daten ableiten und somit ChatGPT im Bezug zur ABAP-Entwicklung messbar machen zu können. Hierfür werden wir automatische Tests zur Programmentwicklung und manuelle Tests zur Programmierklärung, korrekte Syntaxerkennung und zur Fehlerfindung entwickeln. Aus den Messdaten werden wir schließlich verschiedene Hypothesen aufstellen, die das Verhalten von ChatGPT versuchen zu erklären. Ebenfalls werden wir testen ob die Version oder die Eingabesprache, mit der wir die Benchmarks an ChatGPT kommunizieren, Einfluss auf die Messergebnisse hat, wie beispielsweise die Qualität der Ergebnisse bzw. die Anzahl an enthaltenen Fehlern. Zudem werden wir ein Fazit formulieren, welches darstellen soll, wie es mit der aktuellen und zukünftigen Nutzbarkeit von LLMs durch Schnittstellen wie ChatGPT zur Unterstützung von ABAP-Entwicklern aussieht. Abschließend werden wir aufgrund unserer erarbeiteten Erkenntnisse, einen Ausblick auf das Potential von zukünftigen LLMs und neueren Versionen von ChatGPT treffen.

1.4 Begriffsklärung

1.4.1 SAP

Zunächst einmal stellt sich die Frage, was ist eigentlich SAP, welches das Unternehmen selbst mit dem folgenden Satz erklärt: “SAP ist einer der weltweit führenden Anbieter von Software für die Steuerung von Geschäftsprozessen und entwickelt Lösungen, die die effektive Datenverarbeitung

und den Informationsfluss in Unternehmen erleichtern.“ Der Name des Unternehmens SAP steht hierbei als Abkürzung für “Systemanalyse Programmentwicklung”. Weltweit hat das Unternehmen nach eigenen Angaben mehr als 110000 Mitarbeiter, 22000 Partnerunternehmen und 245 Millionen Cloud-Abonnenten. [SAP23e] Das Unternehmen wurde in Deutschland im Jahr 1972 von Dietmar Hopp, Hasso Plattner, Klaus Tschira, Claus Wellenreuther und Hans-Werner Hector gegründet. SAP ist nach aktuellstem Stand, das wertvollste börsennotierte Unternehmen Deutschlands, mit einem Marktwert von 151 Milliarden Euro [Fin23]. Eine zentrale Rolle bildete das Unternehmen weltweit bei der Einführung von seiner ERP-Software (Enterprise Resource Planning), SAP R/2 und SAP/3 mit denen es einen Standard setzte [SAP23g]. Diese Software dient dazu Hauptprozesse eines Unternehmens, wie das Personalwesen oder die Logistik, gebündelt verwalten zu können [SAP23f]. Die neueste SAP ERP-Software ist SAP S/4HANA, welche es unter anderem ermöglicht, weitaus größere Datenmengen als früher effizient verwalten zu können. Des weiteren ermöglicht SAP S/4HANA es, neuste Innovationen wie beispielsweise Künstliche Intelligenz mit in das System zu integrieren. [SAP23d]

1.4.2 ABAP

ABAP (Advanced Business Application Programming) ist eine proprietäre Programmiersprache von SAP die erstmals 1983 veröffentlicht wurde und zur Programmierung kommerzieller Anwendungen im SAP-Umfeld dient. Programme liegen als Sourcecode in der SAP Datenbank vor, diese können zu einem Bytecode kompiliert werden der auch innerhalb der SAP Datenbank abgelegt wird. Dieser Bytecode kann dann von der ABAP-Laufzeitumgebung innerhalb der SAP GUI interpretiert werden.

Der Programmierer greift über die SAP GUI auf die ABAP Workbench, wo der ABAP Editor (SE38) zu finden ist zu. Die Programmiersprache bietet typische Variablen, Datentypen, Operationen und Kontrollstrukturen wie If-Then-Else, Do-While und Case-Anweisungen. Außerdem gibt es Funktionsbausteine und Standardfunktionsbausteine die häufige Aufgaben der Datenverarbeitung optimieren. Seit 1999 unterstützt ABAP auch objektorientierte Entwicklung durch hinzufügen von Klassen, Objekten und Vererbung. Eine der wichtigsten Funktionen von ABAP ist die Datenbankinteraktion, hier bietet ABAP eine Abstraktionsebene um sämtliche Datenbankinteraktionen unabhängig von der Datenbankplattform zu machen und um Datenabfragen, -manipulationen und -speicherungen zu optimieren.

1.4.3 LLMs

Large Language Models (LLMs) sind große künstliche neuronale Netze mit vielen Parametern die, anhand großer Datenmengen von Text und typischerweise durch die Lernverfahren unsupervised learning oder self-supervised learning, lernen die natürliche Sprache algorithmisch zu verarbeiten. Es hat sich gezeigt, dass LLMs für viele verschiedenen Aufgaben die mit Sprache zutun haben eingesetzt werden können, sie werden unter anderem zur Textgenerierung, Textkorrektur, Textübersetzung, als Frage-Antwort-System, als Chatbot, zur Textanalyse und Textzusammenfassung genutzt. Ihre Fähigkeit scheint direkt abhängig von der Menge der Trainingsdaten, ihrer Größe (Anzahl der Parameter) und des Rechenaufwandes (FLOPs) zu sein [Bow23; Kap+20; Wei+22].

LLMs verwenden einen sogenannten Tokenizer um Text in repräsentative Zahlen (Tokens) umzuwandeln, mit den daraus entstehenden Tokens arbeitet das neuronale Netz dann weiter. Wie der Tokenizer den Text in einzelne Tokens aufteilt kommt auf die spezielle Implementierung an, es werden aber üblicherweise statische Modelle verwendet um häufige Buchstabenkombinationen oder ganze Wörter in ein Token zusammenzufassen. Derzeit werden üblicherweise ca. 4 Zeichen (Buchstaben, Zahlen, Sonderzeichen) pro Token kodiert. Dies wird gemacht da LLMs nur begrenzt viele Tokens verarbeiten können um die Performanz zu gewährleisten. Wird das Tokenlimit überschritten kann das LLM keine weiteren Informationen aufnehmen und das Ende der Eingabe wird ignoriert.

Ein weiterer wichtiger Begriff für LLMs ist die Temperatur, dies ist eine Einstellung die bei der Generierung gemacht wird um die Ausgabe zu beeinflussen. Bei einer Temperatur von 0 ist das LLM nahezu deterministisch, durch erhöhen der Temperatur wird die “Kreativität” erhöht. Die maximale Temperatur ist abhängig vom speziellen LLM üblich sind 1, 2 und 5. OpenAI verwendet für ChatGPT mit GPT-3.5 und GPT-4 jeweils 2 als Maximalwert.

LLMs arbeiten heutzutage hauptsächlich nach dem Deep-Learning Modell der Transformer-Architektur. Die Eingabesequenz wird durch Encoder-Blöcke in eine Zwischensequenz überführt, die dann im Self-Attention Layer durch Attention-Heads anhand ihrer Beziehung zueinander gewichtet werden. Anschließend wird die Zwischensequenz durch Decoder-Blöcke in eine Ausgabesequenz überführt [Vas+17]. Durch die Gewichtung wird der Kontext innerhalb der Eingabesequenz berücksichtigt, ohne jedes Wort einzeln bearbeiten zu müssen. Dies hat sich besonders für die natürliche Sprachverarbeitung als sehr hilfreich erwiesen, da die gesamte Eingabesequenz auf einmal verarbeitet werden kann und durch Parallelisierung die Trainingszeit reduziert werden kann.

Üblicherweise werden LLMs trainiert indem sie große Mengen an unklassifizierten Trainingsdaten im Pretraining erhalten. Hierfür gibt es aktuell zwei Ansätze:

- Autoregressives Pretraining (autoregressive pretraining, GPT-Style) ist ein Trainingsstil, bei dem das Modell den Kontext nur Unidirektional betrachtet. Eingaben werden von links nach rechts bearbeitet und nur der bereits bearbeitete Teil dient als Kontext, sodass der Anfang keinen Kontext hat und das Ende den gesamten Text als Kontext ansieht. Diese Kontextbetrachtung ist in Abbildung 1 dargestellt.

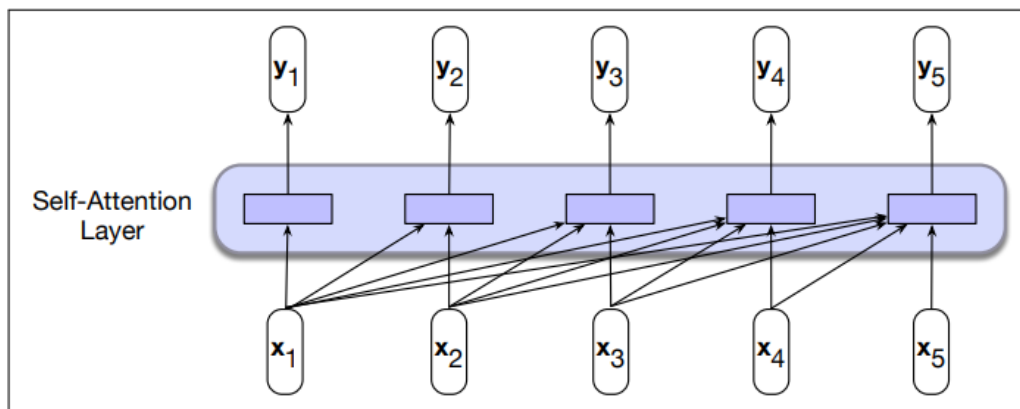


Abbildung 1: Autoregressives Pretraining: Kontext wirkt nur in eine Richtung [DJ23]

- Maskiertes Pretraining (masked pretraining, BERT-Style) ist ein Trainingsstil, bei dem das Modell den Kontext Bidirektional betrachtet. Hier werden zufällig Eingabetoken maskiert mit dem Ziel, dass die maskierten Token durch den gesamten Kontext berechnet werden sollen. Somit wird die gesamte Eingabe überall als Kontext benutzt was einige Aufgaben wie die Stimmungsanalyse (sentiment analysis) von Texten verbessert. Diese Kontextbetrachtung ist in Abbildung 2 dargestellt, wobei beliebige Eingabeparameter x im Training maskiert werden können.

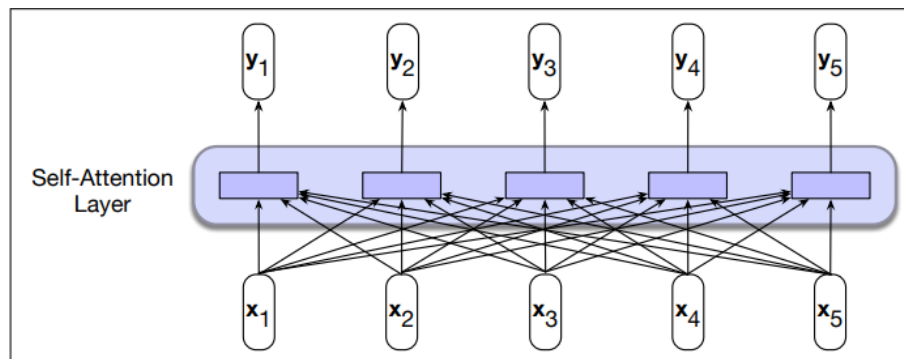


Abbildung 2: Maskiertes Pretraining: Kontext wirkt in beide Richtungen [DJ23]

Nachdem das Pretraining abgeschlossen ist werden LLMs häufig durch fine-tuning auf eine bestimmte Aufgabe spezialisiert. Dies wird erreicht indem die Gewichte des künstlichen neuronalen Netzes auf neuen, bereits klassifizierten Daten trainiert werden, teilweise werden dabei Layer

eingefroren, um Änderungen an bestimmten Stellen zu vermeiden oder neu hinzugefügt um die Aufgabe zu verfeinern.

1.4.4 ChatGPT

ChatGPT ist ein Chatbot, welcher von dem Unternehmen OpenAI entwickelt und veröffentlicht wurde. Er basiert auf dem LLM GPT-3.5 und wurde durch die Unterstützung von Microsoft Azure und deren Supercomputern trainiert [Ope23]. Ebenfalls hat Microsoft 10 Milliarden Dollar in OpenAI investiert, wodurch sie die Möglichkeit haben, ChatGPT in ihre Produktpalette, wie beispielsweise Bing zu integrieren [23f].

Chatbot ist in der Lage, menschenähnliche Kommunikationen abzuhalten, verschiedene Dinge verständlich zu erklären oder auch Texte zu verschiedenen Themen zu verfassen [23d; Eck23]. Ebenfalls kann der Chatbot aus einer Konversation lernen und sich somit für nachfolgende Fragen verbessern, was sich jedoch nur auf die aktuelle und nicht auf neue bzw nachfolgende Konversationen bezieht. Erwähnenswert zu ChatGPT im Bezug auf die menschenähnliche Kommunikation ist ein Beispiel, welches darstellt, wie effizient der Chatbot ist. Dieses Beispiel ereignete sich in Österreich, wo ein Politiker eine von ChatGPT geschriebene Rede im Landtag abhielt und niemandem auffiel, dass diese von einer KI bzw. von ChatGPT geschrieben wurde, bis dieser es selbst aufdeckte [23j]. Man unterscheidet verschiedene Versionen von dem Chatbot, aktuell gibt es GPT-3.5 und GPT-4, welche unterschiedliche Trainingsdaten und Parameter besitzen und dementsprechend variieren auch ihre Fähigkeiten [Hug00]. Diese Antworten der LLMs können trotz der gewaltigen Datenmengen, die zum Training genutzt worden sind falsch sein und Filter erlauben zudem keine Antworten auf "gefährliche" und "unangemessene" Fragen. Diese falschen Antworten kann ChatGPT teilweise selbstständig durch Folgefragen korrigieren und somit ein richtiges Ergebnis liefern. Trotzdem muss man beachten, dass ChatGPT auch wenn es sich sicher ist, dass die Antworten korrekt sind, diese trotzdem falsch sein können und man diese doppelt prüfen sollte. Im weiteren Verlauf betrachten wir die Basis Version von ChatGPT basierend auf GPT-3.5 (aus dem Jahr 2022) und die neueste Version, welche auf GPT-4 basiert (aus dem Jahr 2023).

Hierbei sollte noch berücksichtigt werden, dass ChatGPT nur mit Daten bis September 2021 trainiert worden ist, und somit keine aktuellen Ereignisse wissen kann, sofern man sie ihm im Konversationsverlauf nicht mitteilt. [23g]

1.5 Kategorisierung von Softwareentwicklungsaufgaben und Abschätzung ihrer Tauglichkeit zur Unterstützung durch LLMs

In diesem Kapitel werden wir die Aufgaben der Softwareentwicklung in die sechs Phasen des Software Development Life Cycles (SDLC) einordnen und abzuschätzen inwiefern LLMs in den Entwicklungsphasen helfen können.

Der Software Development Life Cycle ist ein Modell aus der Softwareentwicklung um bei Entwurf, Entwicklung und Test von guten Softwareprodukten zu helfen und Rahmenbedingungen wie Projektkosten, Abgabetermine und Kundenzufriedenheit einzuhalten. Er dient als Grundlage für viele der beliebtesten Softwareentwicklungsmodelle wie dem Wasserfallmodell, dem Spiralmodell, der Agilen Softwareentwicklung, dem inkrementellen Vorgehensmodell und dem V-Modell. [RTK22; Reb]

1.5.1 Software Development Life Cycle Phasen

1.5.1.1 Planung

Der erste Schritt im Software Development Life Cycle ist die Planung, hier werden mit dem Kunden zusammen Projektziele festgelegt und ihre Realisierbarkeit geprüft, allgemeine Rahmenbedingungen wie Entwicklungszeit und -kosten geplant und benötigte Ressourcen ermittelt. Außerdem werden Anforderungen mit Domänenexperten ermittelt, Qualitätssicherungsanforderungen geplant und eine Risikobetrachtung durchgeführt um das Projekt im weiteren Verlauf erfolgreich durchführen zu können. [RTK22; Reb]

In der Planungsphase ist keine großartige Unterstützung durch LLMs zu erwarten, da die Arbeit hier hauptsächlich in der Kommunikation mit dem Kunden und Experten liegt und ein LLM keinen Zugriff auf diese Kommunikation hat. Zwar ist es denkbar dass im Nachhinein Mitschriften oder Audioaufnahmen an ein LLM übergeben werden, aber selbst dann kommt es zu Kommunikationsschwierigkeiten wenn Rückfragen gestellt werden müssen. Die einzige sinnvolle Unterstützung

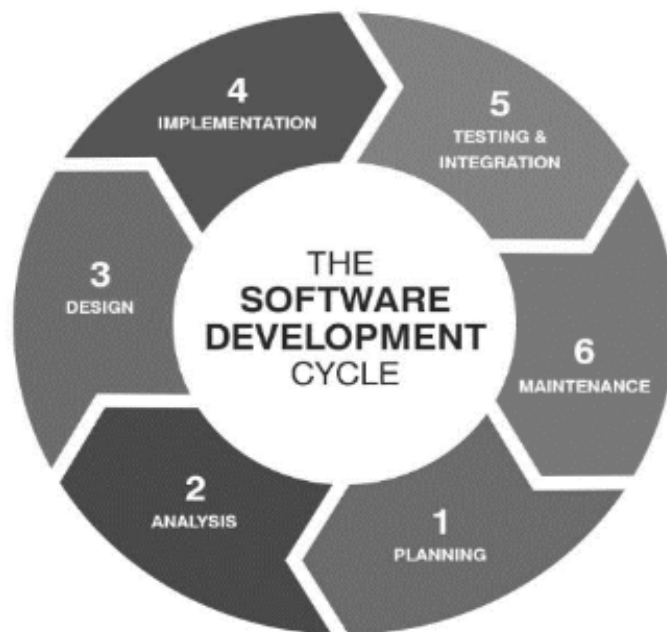


Abbildung 3: Software Development Life Cycle [RTK22]

bieten LLMs hier in der Vor- und Nachbereitung von Planungsgesprächen von einzelnen Teilnehmern um das Planungsgespräch zu verbessern.

1.5.1.2 Anforderungsanalyse

Die Anforderungsanalyse beschäftigt sich damit die Anforderungen, die in der Planungsphase ermittelt wurden, zu definieren, dokumentieren, analysieren und letztlich die Kundenzusage zu erhalten. Es sprechen sich Experten der Anforderungsanalyse, der Softwareentwicklung und der Qualitätssicherheit (Software Testing) miteinander ab und diskutieren die vorgeschlagenen Anforderungen, deren Dokumentation und die geplante Implementierung [RTK22; Reb].

In der Anforderungsanalyse ist auch keine großartige Unterstützung durch LLMs zu erwarten, es handelt sich auch hier hauptsächlich um kommunikative Arbeit, die im direkten Gespräch stattfindet und nicht direkt von LLMs unterstützt werden kann. Wie in der Planungsphase könnten LLMs von den Gesprächsteilnehmern zur Vor- und Nachbereitung genutzt werden.

1.5.1.3 Design / Entwurf

Bei der Entwurfsphase, werden verschiedene Aspekte des Projekts wie das geplante Budget oder die zeitliche Beschränkung mit den Stakeholdern abgesprochen und der beste Gestaltungsansatz wird für das Produkt ausgewählt. Ebenfalls werden in der Entwurfsphase zum Beispiel die Programmiersprache und die Hardware Spezifikationen, welche für das Projekt benötigt werden abgesprochen und auf dessen Basis das Projekt entwickelt wird. [RTK22; Reb]

Da diese Phase ziemlich dynamisch ist, sich dort also noch vieles ändern kann, vermuten wir, dass die Unterstützung durch LLMs hier eher eine Belastung wäre. Ein weiterer Aspekt, der darstellt das die Unterstützung durch LLMs hier nicht sinnvoll ist, wäre, dass aufgrund der vielen Absprachen und Rücksprachen wie zum Beispiel zwischen Design Team und den Stakeholdern die Einbeziehung von LLMs zur Unterstützung dieses Prozesses schwierig werden könnte.

1.5.1.4 Umsetzung

Die Umsetzungsphase dient dazu das Softwareprodukt anhand der vorher spezifizierten Anforderungen und des erstellten Entwurfs zu erstellen. Hierfür werden die vorher festgelegten Richtlinien und Regeln betrachtet und vereinbarte Programmiersprachen, Formatierungsregeln und Hilfsmittel wie Compiler, Interpreter und Debugger benutzt um ein einheitliches und wartbares Softwareprodukt zu erschaffen [RTK22; Reb].

In der Umsetzungsphase ist die mögliche Unterstützung durch LLMs vermutlich am größten. Die Programmierung wird zwar von einem gesamten Team durchgeführt aber der Großteil der Arbeit findet alleine am Computer statt, was die Kommunikation mit einem LLM ermöglicht. Erste Tests

haben gezeigt dass LLMs in der Lage sind Programmcode zu generieren, dokumentieren und auf Fehler zu analysieren. Kleine, unabhängige, Programmausschnitte lassen sich ohne Probleme von Hand an ein LLM geben was heute schon viele Programmierer über ChatGPT machen. Größere Ausschnitte könnten durch die Integrierte Entwicklungsumgebung (IDE) automatisch als Kontext an ein LLM gegeben werden, auch dieser Ansatz wird bereits verfolgt [Git23a].

1.5.1.5 Test und Integration

Die Test und Integrationsphase zeichnet sich dadurch aus, dass sie eigentlich eine Untergruppe von allen anderen Phasen ist, da generell bei SDLC Modellen in jeder Phase getestet wird. Das spezifische an der Test und Integrationsphase ist, dass sie sich speziell nur auf das Testen des Produkts wie zum Beispiel Tests zur erfolgreichen Integration fokussiert und Mängel bzw Fehler dokumentiert werden. Diese dokumentierten Fehler werden dann korrigiert und immer wieder getestet, bis die Qualität den gewünschten Anforderungen entspricht [RTK22; Reb].

Wir gehen davon aus, dass sich bei dieser Phase LLMs als Unterstützung gut eignen könnte, das es beispielsweise komplexe Zusammenhänge gut verstehen und erklären kann und somit vermutlich hilfreich bei der Analyse von Testergebnissen sein könnte.

1.5.1.6 Wartung

Vor der initialen Veröffentlichung kann es Feedback z.B. von der Geschäftsleitung geben und das Produkt wird entweder an das Feedback angepasst und Änderungen werden miteinbezogen, oder es wird ohne weitere Änderungen im aktuellen Zustand veröffentlicht. Der Fokus der Wartungsphase ist, dass nachdem das Produkt schließlich veröffentlicht wurde, es für die Kundschaft weiterhin unterstützt / geupdatet wird. Diese Updates umfassen zum Beispiel das beheben von Bugs oder von Fehlern, welche nach der Veröffentlichung gefunden wurden. Desweiteren werden in der Wartungsphase Änderungen am Produkt vorgenommen diese können Umfassen: Verbesserungen, neue Features oder auch das verhindern von Zukünftig auftretende Probleme, in Form von Updates. [RTK22; Reb]

Hier sind LLMs geeignet als Unterstützung, da man auf konkrete Anfragen wie zum Beispiel Updates oder Tickets reagiert und diese direkt an ein LLM weitergegeben werden könnten. Ebenfalls können LLMs Verbesserungen beim existierenden Produkt vorschlagen und so Ideen für kommende Updates liefern. Bei der Fehlerbehebung und dem Hinzufügen von neuen Features können LLMs eingesetzt werden, wie in der Umsetzungs- und Testphase beschrieben.

1.5.2 Rollenbezug

In einem Unternehmen gibt es verschiedene Bereiche welche unterschiedlichen Aufgaben erledigen. Anhand eines Projekts kann man sehr gut die verschiedenen Rollen bzw. Bereiche der Mitarbeiter darstellen. Beispiele hierfür sind unter anderem Salesmanager, Business Analyst oder auch Developer. Diese verschiedenen Rollen haben unterschiedliche Schwerpunkte im Bezug auf den Software Development Life Cycle. Anhand der Abbildung 4 erkennt man, dass zum Beispiel der Salesmanager sehr spezifisch auf nur eine Phase des SDLC einwirkt, wobei der Business Analyst auf ganze 6 Phasen Einfluss hat. Interessant für uns ist hierbei die Betrachtung, welche Phasen einerseits (unserer Meinung nach) gut durch LLMs unterstützt werden können und andererseits, welche Mitarbeiter wie viele dieser geeigneten Phasen begleiten.

Anhand dieser Betrachtung können wir Aussagen darüber treffen, welche Rolle bzw. welche Mitarbeiter am meisten durch LLMs profitieren würden. Ein Beispiel dafür wäre, dass der Developer, wie man an der Abbildung 4 erkennen kann, die Phasen Design, Implementation, Testing, Integration, Support und Dekommissioning begleitet. Viele dieser Phasen, vor allem Implementation und Testing eignen sich, durch LLMs unterstützt zu werden, wodurch man sagen kann, dass die Rolle des Developer durch LLM Unterstützung stark profitieren könnte.

1.6 Wissenschaftliche Literatur zur KI-Unterstützten Softwareentwicklung

1.6.1 Programmieren lernen mit LLMs

In der aktuell vorhandene Bestandsliteratur findet man bereits viele relevante Informationen, welche für unser Projekt hilfreich sein können. Beispielhaft hierfür ist das Thema, wie LLMs, über Schnittstellen wie ChatGPT in der Bildung eingesetzt werden können bzw. welche Möglichkeiten

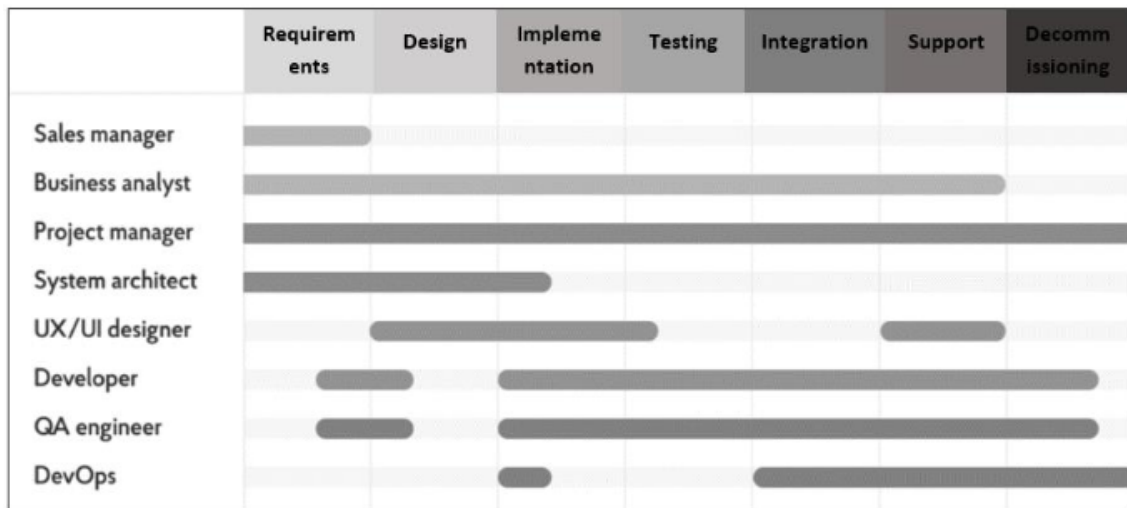


Abbildung 4: Rollen und die Bereiche ihrer Verantwortungsbereiche [RTK22]

und Herausforderungen im Umgang mit diesen KI-Tools gerade im Bereich der Softwareentwicklung existieren.

Zunächst einmal wird angesprochen, dass in jüngster Zeit äußerst viele KI-Modelle veröffentlicht wurden, welche in verschiedenen Bereichen wie der angesprochenen Codegenerierung hilfreich sein können [Bec+23; Tli+23]. Es wird davon gesprochen, dass diese Tools zur Codegenerierung bereits von Informatikstudenten genutzt werden, um beispielsweise Programmieraufgaben der Lehrveranstaltungen schneller bzw. einfacher lösen zu können.

Des Weiteren wird beschrieben, dass die Anforderungen an Informatikstudenten gerade am Anfang im Bereich der Programmierung schon länger diskutiert werden und das programmieren an sich von vielen als schwer angesehen wird. Im Hinblick auf die Schwierigkeiten beim programmieren wird davon gesprochen, dass es also nicht wirklich verwunderlich ist, dass viele Studenten auf die KI-Codegenerierung zurückgreifen.

Es wird die Frage aufgeworfen, dass wenn von nun an durch die KI-Codegenerierung viele Programmieraufgaben mit Leichtigkeit gelöst werden können, wie dann die zukünftigen Informatik Lehrveranstaltungen und Lehrpläne aussehen werden. Infolgedessen wird davon ausgegangen, dass sich in näherer Zukunft im Bezug zu “wie wird programmieren gelehrt” vermutlich einiges ändern wird [Bec+23].

Außerdem wird erwähnt, dass bereits einige Forscher die Vorteile und Nachteile von LLMs in der Bildung untersucht, und dementsprechend potenzielle Richtlinien für einen Umgang mit diesen verfasst haben. Die potenziellen Nachteile von KI-Tools im Bildungsbereich wurden jedoch bis jetzt nicht genau untersucht.

In der Literatur wird erläutert, dass manche Schulen ChatGPT bereits aus ihren Netzwerken gesperrt haben aufgrund von Bedenken im Hinblick auf Schummeln. Vor diesem Hintergrund wird erläutert, dass es wichtig ist, die Bedenken hinsichtlich KI-Tools wie ChatGPT im Bereich der Bildung näher zu untersuchen, um eine zweifelsfreie Nutzung sicher zu stellen. [Tli+23]

Eine Quelle hat verschiedene Ansätze zur Untersuchung von Meinungen zu ChatGPT in der Bildung erstellt und analysiert. Für uns interessant sind hierbei hauptsächlich die Analyse von Interviews und die Untersuchung von Nutzererfahrungen.

Bei der Analyse von Interviews betrachten wir verschiedene relevante Bereiche wie: Wandel in der Lehre, Antwortqualität und Ethik. Angefangen bei dem Wandel in der Lehre, wo über die Änderungen welche ChatGPT in der Bildung bringen könnte, gesprochen wird. Es wird genannt, dass die Antworten der meisten Teilnehmer erkennen lassen, dass ChatGPT die Chance auf einen Bildungserfolg steigert. Zudem wird von den Teilnehmern genannt, dass ChatGPT komplexe Themen effizient und verständlich erklären kann, woraus hergeleitet wird, dass ChatGPT zu einer Umstrukturierung des Unterrichts führen wird. Die andere Seite der Teilnehmer, nannte jedoch Bedenken im Umgang mit ChatGPT in der Bildung, denn wenn die eigene Motivation von Studierenden fehlt, können schnell und einfach Lösungen generiert werden, ohne dass man sich selbst mit den Aufgaben auseinandersetzt. Worunter dann die eigenen Fähigkeiten wie zum Beispiel das

Erarbeiten von Problemlösungen leiden.

Der zweite Bereich also die Antwortqualität setzt sich mit der Qualität der von ChatGPT gegebenen Antworten auseinander. Es wird erwähnt, dass die Qualität und die Genauigkeit der gegebenen Antworten grundsätzlich gut sind, teilweise aber irreführende Informationen in diesen Antworten enthalten sind. Außerdem wird ein Beispiel genannt, dass ein von ChatGPT erzeugter Programmcode in einer Entwicklungsumgebung nicht funktionierte. Ein Interviewter Programmierer sagt, dass die Qualität der Antworten von den Fragen abhängt, dass wenn beispielsweise die Fragen zu aktuell sind, die Antworten nicht so gut sein werden, da der Kontext fehlt oder das Fragen nicht spezifisch genug gestellt werden [Tli+23].

In dem letzten für uns relevanten Bereich, Ethik, wird sich mit den ethischen Bedenken, welche durch die Nutzung von ChatGPT im Bildungsbereich entstehen können, auseinandergesetzt. Beispiele hierfür sind unter anderem die Förderung von Plagiaten, Betrug (zum Beispiel bei Tests), die Übernahme von voreingenommenen Informationen, ohne sich selbst damit zu befassen und die daraus resultierende Erzeugung von Faulheit. Ein weiteres Beispiel wären gegebenenfalls Lizenzierungsprobleme, denn die Daten, mit denen die Modelle trainiert wurden, enthalten auch lizenzierten Code auf Basis dessen dann teilweise die Codegeneration erfolgt [Bec+23; Tli+23]. Zudem sehen die Teilnehmer bedenken bei relevanten Themen, wo teilweise durch ChatGPT nur grobe und ungenaue Antworten gegeben wurden. Des Weiteren wird beschrieben, dass das kritische Denken und das kritische Auseinandersetzen mit den gegebenen Antworten teilweise fehlt und sich einfach auf die von ChatGPT gegebenen Antworten verlassen wird [Tli+23]. Ein weiterer Aspekt der ethischen Bedenken im Bezug zu KI-Modellen wäre die Nachhaltigkeit. Das Trainieren von diesen Modellen wie ChatGPT verbraucht große Mengen von Energie auf Kosten der Umwelt. [Bec+23]

Weiter geht es mit der genannten Untersuchung von Nutzererfahrungen bzw. die Möglichkeiten welche durch die KI-Tools wie ChatGPT in der Bildung mit Fokus auf die Programmierung entstehen. Der Fokus liegt hier bei der Erstellung von Lerninhalten und geht es darum, dass KI-Tools fast unlimitiert qualitative Aufgaben generieren können, welche normalerweise einen hohen Wissensgrad in dem gegebenen Bereich voraussetzen [Bec+23]. Beachtet werden muss jedoch, dass sich die verschiedenen erstellten Aufgaben, auch wenn sie zum gleichen Thema sind, stark unterscheiden können. Diese Unterscheidungen werden in Form von Quiz Aufgaben dargestellt. Es gibt unter anderem unterschiedliche Schwierigkeitsstufen der generierten Aufgaben oder auch Aufgaben, welche zu einfach im Kontext gestellt sind und dadurch die gesuchte falsche Antwort ohne Hintergrundwissen beantwortet werden kann [Tli+23]. In Abbildung 5 ist ein Beispiel hierzu dargestellt, die letzte Antwortmöglichkeit hat nicht einmal Ansatzweise etwas mit der Fragestellung zu tun, sodass man die Frage auch ohne jegliches Fachwissen beantworten kann. In Abbildung 6 ist ein zu einer besser gestellten Aufgabe zu sehen, hier sind alle Antwortmöglichkeiten im gleichen Kontext wie die Frage.

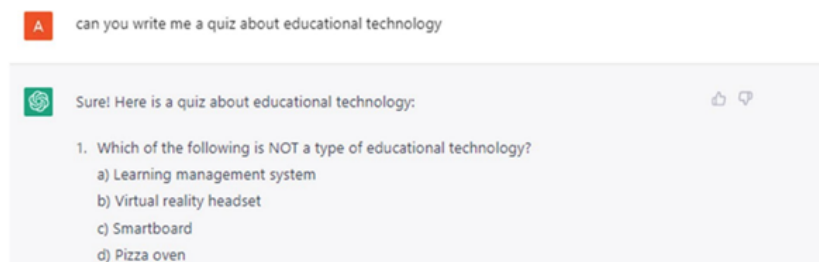


Abbildung 5: Beispiel für eine schlechte Aufgabe[Tli+23]

Neben diesen gelegentlichen Qualitätsabweichungen der Aufgaben lassen sich mit KI-Tools jedoch wie bereits erwähnt gute Übungen mit passenden Musterlösungen generieren. Es wurde festgestellt, dass KI-Modelle neue und sinnvolle Programmierübungen erstellen und diese an bestimmte Themenbereiche und Kontexte anpassen können. Des weiteren können diese Modelle komplexen Code einfach und verständlich erklären, was ebenfalls zum lernen und verstehen der Studierenden beiträgt. Bei einer Untersuchung wurden zudem festgestellt, dass diese Erklärungen meistens zutreffend sind. Neben den Programmieraufgaben und Codeerklärungen ist es ebenfalls möglich durch KI-Tools verschiedenen Lösungswege und Beispiele zu generieren welche den Weg zur Problemlösung verständlicher machen können [Bec+23].

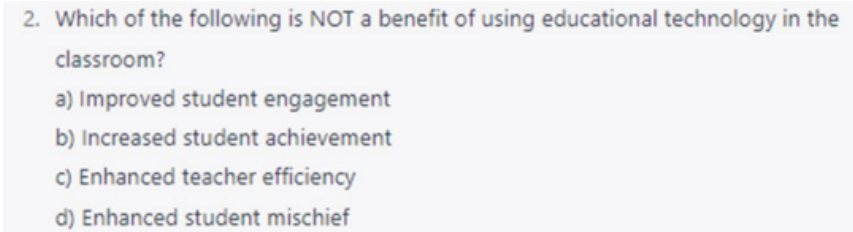
- 
2. Which of the following is NOT a benefit of using educational technology in the classroom?
- a) Improved student engagement
 - b) Increased student achievement
 - c) Enhanced teacher efficiency
 - d) Enhanced student mischief

Abbildung 6: Beispiel für eine gelungene Aufgabe[Bec+23]

Aus diesen zu beachtenden Punkten und Möglichkeiten werden verschiedene neue Ideen zu Bildungsmethoden der Zukunft aufgezeigt. Die ersten Methoden sollen darauf abzielen, das selbstständige Lernen zu erhalten und zu verhindern, dass man Aufgaben wie Hausarbeiten nur mithilfe von zum Beispiel ChatGPT und ohne eigenes lernen schreibt. Geschehen soll dies, indem die Beurteilung der Studenten mit Fokus auf mündliche Ausarbeitungen wie Diskussionen oder Präsentationen verschoben wird. Eine weitere Methode ist, dass das kritische Denken der Schüler gestärkt werden sollte, damit diese die Qualität der von den KI-Tools gelieferten Informationen richtig einordnen können und sich nicht nur blind darauf verlassen. Zudem sollte durch die schnelle Entwicklung im KI-Bereich mehr erforscht werden, wie man Chatbots wie ChatGPT nutzen kann um die Studenten wie auch die Dozenten in der Bildung zu unterstützen und “kollaborative Intelligenz” also das effektive zusammenarbeiten von KI und Mensch in der Lehre zu ermöglichen [Tli+23].

Eine weitere Quelle verfolgt die Ansätze, dass einerseits die Grundprinzipien vom Programmieren und Syntax von der KI übernommen werden sollte, damit sich Studierende vermehrt auf Algorithmen und Problemlösungen fokussieren können. Des Weiteren soll den Studenten “Prompt Engineering” näher gebracht werden, also die Lehre wie man algorithmische Probleme klar kommuniziert, um somit effiziente Eingaben für die KI schreiben zu können, sodass qualitative Antworten hervorgebracht werden. Außerdem sollen einerseits Hindernisse beim Programmieren genommen werden, indem das KI-Tool den Studenten schon Code gibt, auf Basis dessen diese weiterarbeiten können, damit diesen ein klarer Anfang aufgezeigt wird. Hinsichtlich dessen könnte man überlegen, ob sich der Fokus auf das umschreiben, debuggen oder erweitern von Code verlagert, was oftmals der Realität näher ist, als den kompletten Code neu zu schreiben. Ein weiteres Hindernis ist die schlechte Lesbarkeit von Compilerfehlern beim Programmieren, diese können umgangen werden indem KI-Tools sie (wie es bereits durch Untersuchungen aufgezeigt wurde) verständlich erklären und sogar passende Lösungen für die Fehler je nach Kontext bereitstellen [Bec+23].

1.6.2 LLM Benchmarking

Um die Fähigkeiten von LLMs objektiv darzustellen werden Benchmarks entwickelt welche die Qualität der Ergebnisse von LLMs messen sollen. Dafür werden viele verschiedene Tests eingesetzt, die aktuell bekanntesten Ergebnisse sind die, die OpenAI in ihrem technischen Report zu GPT-4 veröffentlicht hat [Ope23]. Hier wurden einige Abschlussarbeiten und Zulassungsprüfungen an GPT-4 getestet. Es hat in berühmten US-amerikanischen Tests wie unter anderem dem Bar-Exam (Jura), SAT und GRE (Lesen, Schreiben, Mathe) sowie in fortgeschrittenen Kursen zu Kunstgeschichte, Biologie, Chemie, Umweltwissenschaften, Psychologie sehr gut abgeschnitten[Ope23].

Um genauere und allgemeinere Betrachtungen machen zu können gibt es Benchmarks und Benchmarking-Frameworks die sich üblicherweise auf ein Fachgebiet fokussieren und dieses genauer untersuchen. So gibt es beispielsweise GLUE [Wan+18] zur Analyse von Grammatik, Textähnlichkeit, Umschreibungen und Inferenz, SuperGLUE [Wan+19] zur Analyse vom Verständnis natürlicher Sprache, Argumentation, Kausalität und dem Dialog mit Menschen. Außerdem gibt es den BLEU Algorithmus [Pap+01] sowie das komplexere Framework COMET [Rei+20] zur Analyse der Qualität von maschineller Übersetzung natürlicher Sprachen. Für uns relevant sind vor allem Benchmarks die sich auf die Analyse von synthetisiertem Programmcode durch LLMs beziehen. Die am weitesten verbreiteten Benchmarks hierfür sind HumanEval[Che+21] und MBPP[Aus+21].

HumanEval ist ein Benchmarking Framework das zusammen mit Codex (dem LLM hinter Github Copilot [Git23a]), veröffentlicht wurde um die funktionale Korrektheit von Programmen, die vom Docstring generiert wurden, zu testen. Es enthält insgesamt 164 manuell erstellte Aufforderungen (Prompts) mit Erklärung in Docstring und eine Funktionsdefinition, die das LLM anweisen, alleinstehende Pythonfunktionen zu generieren. Außerdem stehen für jede Aufgabe eine empfohlene Lösung und einige (im Durchschnitt 7.7) Unittests, zur Überprüfung der generierten Lösung zur Verfügung. Hier ist es sehr wichtig dass die Prompts selbst erstellt sind, da LLMs anhand von riesigen Datenmengen trainiert worden sind und Internetseiten wie Github, Gitlab und viele Lehrbücher vermutlich Teil der Trainingsdaten sind.

Im dazugehörigen Paper “Evaluating Large Language Models Trained on Code” [Che+21] wird erklärt wie der iterative Entwicklungsprozess und Problemlösungsprozess, welcher in der professionellen Softwareentwicklung typisch ist, auch auf die Ergebnisse von LLMs angewendet werden kann. Hierfür wurde das LLM Codex als Grundlage genommen und ersteinmal auf alleinstehende, korrekt implementierte Pythonfunktionen feinabgestimmt (fine tuning), das resultierende Modell wurde Codex-S genannt. Anhand von Codex-S wurde dann gezeigt, dass es im ersten Versuch in der Lage war 37.7% der Problemstellungen zu lösen und mit 100 Versuchen viel beeindruckendere 77.5% der Probleme erfolgreich lösen konnte. Da die automatisierte Testdurchführung in der professionelle Softwareentwicklung bereits Standard ist, kann sie mit etwas Anpassung auf die Lösungsvorschläge von einem LLM angewendet werden, bis dieses eine erfolgreiche Lösung findet. Es wird die geschätzte Kennzahl

$$pass@k := \mathbb{E}_{Problems} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

eingeführt, welche die Wahrscheinlichkeit angibt nach k Durchläufen mindestens eine korrekte Lösung zu finden. Hierbei ist c die Anzahl der korrekten Lösungen und n die gesamte Anzahl an Versuchen. Es gilt $n \geq k \geq c$ und für $k \rightarrow \infty$ steigt die Genauigkeit der Schätzung. Außerdem wurde ein Zusammenhang zwischen Pass@K, der Versuchsanzahl (k) und der eingestellten Temperatur des LLMs festgestellt der in Abbildung 7 zusehen ist. Bei wenigen Versuchen ist eine geringe Temperatur von Vorteil um ein sehr berechenbares Ergebnis zu erhalten, aber mit steigender Versuchsanzahl ist eine höhere Temperatur besser, da so mehr Variation in die geprüften Antworten kommt. So wurden die besten Ergebnisse erreicht indem Pass@1 bei einer Temperatur von 0.2 und Pass@100 bei einer Temperatur von 0.8 ausgeführt wurden.

Ein Problem das die Autoren identifiziert haben, ist die Notwendigkeit der Ausführung von unbekannten, eventuell unsicherem Code. Programmcode der von einem LLM generiert wurde ist aus vielen Gründen ein großes Sicherheitsrisiko, das Lernmaterial ist extrem umfangreich und kann nicht komplett von Hand geprüft werden, außerdem ist der Lernprozess selbst von Experten nicht komplett vorhersagbar. Dem wurde vorgebeugt indem der Programmcode nur innerhalb einer isolierten Sandbox-Umgebung ausgeführt wurde wo der unbekannte Programmcode keinen dauerhaften Schaden anrichten kann. Ein weiteres Problem war Code der nicht terminiert oder

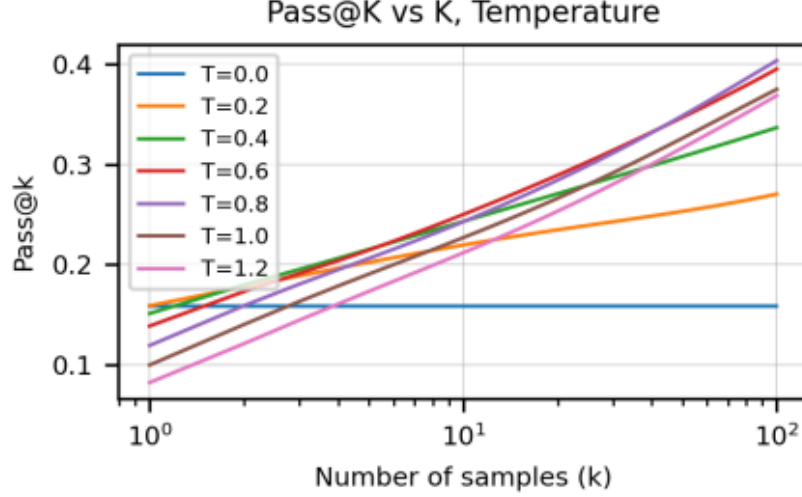


Abbildung 7: Pass@K vs K, Temperature von Codex in Python [Che+21]

sehr ineffizient ist. Hierfür wurde ein Timeout festgelegt, der die maximale Laufzeit begrenzt, die Ausführung wird danach automatisch abgebrochen und der Versuch als fehlgeschlagen angesehen.

In “Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT” [Yet+23] wurde HumanEval genutzt um die Code-Generierungsfähigkeiten von Github Copilot, Amazon CodeWhisper und ChatGPT in den Bereichen Validität, Korrektheit, Sicherheit, Zuverlässigkeit und Wartbarkeit zu vergleichen. In diesem Vergleich wurden alle LLMs manuell bedient, weshalb auf die Wiederholung der Tests verzichtet, und nur die erste Antwort des LLMs betrachtet wurde.

Copilot und CodeWhisperer wurden über ihre IDE Integration angesprochen, ChatGPT hingegen wurde über das Chat-Fenster auf der OpenAI Webseite. Das Gespräch wurde mit dem Prompt “Generate code using the prompts I will provide” gestartet, dann wurden die Prompts nacheinander in gleichen Chat eingegeben. [23b]

Es wurden drei Szenarien geprüft: zuerst wurde den LLMs ein Prompt mit Erklärung und Funktionsdefinition gegeben, dann ein Prompt das nur eine Funktionsdefinition enthält und schließlich ein Prompt, dass nur eine Dummy-Funktionsdefinition mit unbedeutendem Namen hat. Die Antworten der LLMs wurden abgespeichert und dann automatisch durch den Python Interpreter und die HumanEval Unittests auf Validität und Korrektheit geprüft. Für jedes Prompt wurde die Korrektheit CCS (Code Correctness Score) als prozentualer Anteil der bestandenen Unittests berechnet. Für jedes LLM wurden daraus dann folgende Messwerte erzeugt:

$$\text{Code Correctness} = \frac{\sum_{i=0}^{163} CCS_i [CCS_i = 1]}{164}$$

$$\text{Average Code Correctness} = \frac{\sum_{i=0}^{163} CCS_i}{164}$$

Daraufhin wurde SonarQube [23c] genutzt um die Sicherheit, Wartbarkeit und Zuverlässigkeit der generierten Lösungen zu bewerten. Für die Sicherheit wurde die Anzahl der Schwachstellen in Code analysiert, für die Zuverlässigkeit wurde betrachtet wieviele Bugs SonarQube findet. Die Wartbarkeit wurde anhand von Code-Smells betrachtet, Probleme die im Code vorhanden sind, die zwar syntaktisch und semantisch korrekt sind, aber auf einen schlechten Softwareentwurf hinweisen oder die weiterführende Entwicklung schwieriger machen (schlechte Variablennamen, unbenutzte Funktionen u.ä.).

Die Ausarbeitung kam zu dem Schluss, dass

1. Wenn das Prompt eine Erklärung und Funktionsdefinition enthält schneidet ChatGPT am besten ab, 93.3% der Prompts haben validen Code produziert, 65.2% waren auch korrekt.

Copilot hat für 91.5% der Prompts validen Code generiert, es waren 46.3% korrekt. und CodeWhisperer hat bei 90.2% der Prompts validen Code generiert, es waren aber nur 31.1% korrekt.

2. Wenn das Prompt nur die Funktionsdefinition enthält liefern Copilot und CodeWhisperer mit 78.0% am häufigsten validen Code, gegenüber 76.8% von ChatGPT. Jedoch hat ChatGPT mit 22.0% am häufigsten korrekten Code produziert, in Gegensatz zu 20.1% von Copilot und 14.6% von CodeWhisperer.
3. Wenn das Prompt nur eine Dummy-Funktionsdefinition enthält generiert Copilot mit 93.9% am häufigsten validen Code, gegenüber 89.6% bei CodeWhisperer und 92.7% bei ChatGPT. Aber die Korrektheit ist bei ChatGPT mit 61.6% am höchsten, Copilot erreichte 42.1% und CodeWhisperer nur 27.4%.

Bei der Betrachtung der Sicherheit durch SonarQube hat dieses die Antworten von allen LLMs als maximal Sicher eingestuft, das ist vermutlich auf die Prompts von HumanEval zurückzuführen, diese wurden explizit für kurze, in sich geschlossene Lösungen entwickelt. Die Betrachtung der Zuverlässigkeit hat in einer Lösung von CodeWhisperer, in zwei Lösungen von ChatGPT und in drei Lösungen von Copilot kleine Bugs gefunden. Um diese zu beheben schätzt SonarQube eine manuelle Bearbeitungszeiten von 5 Minuten für CodeWhisperers Lösung, 12.5 Minuten für ChatGPTs und 15 Minuten für die von Copilot. Die Betrachtung der Wartbarkeit ist deutlich schlechter ausgefallen, insgesamt wurden in CodeWhisperers Antworten 24 Code-Smells gefunden, die zusammen eine geschätzte Bearbeitungszeit von 117 Minuten haben. ChatGPT produzierte 15 Code-Smells mit einer Bearbeitungszeit von 134 Minuten und Copilots Lösungen enthielten 19 Code-Smells, für deren Bearbeitung 172 Minuten zu erwarten sind.

Insgesamt wurden bereits solide Grundlagen im Bereich des LLM Benchmarkings gelegt und es hat sich gezeigt, dass einige Vorgehensweisen hierfür gut geeignet sind und diese werden wir in unsere Benchmarkentwicklung einfließen lassen.

Die Wiederverwendung von Prompts aus bereits bestehenden Benchmarks kann, wenn der Versuchsaufbau es erlaubt, einen großen Vorteil darstellen. Dadurch können die erzielten Ergebnisse direkt mit denen von anderen Forschungsgruppen, aber auch von anderen LLMs oder in anderen Programmiersprachen verglichen werden.

Außerdem wird häufig der Ansatz verfolgt, erst kleine Tests durchzuführen um die richtigen Parametereinstellungen zu finden und daraufhin mit den richtigen Parametern einen großen Testdurchlauf zu machen. Dies hat sich als Hilfreich erwiesen weil durch viele Wiederholungen aussagekräftigere Beobachtungen gemacht werden können. Da der Zugriff auf LLMs jedoch mit Kosten (und Zeit) verbunden ist können nicht unbegrenzt viele Durchläufe stattfinden. Die Kombination aus kleinen Parameterfindungstests gefolgt von einem großen Test mit den besten Parametern scheint einen guten Kompromiss darzustellen.

1.7 Datensicherheit und die Nutzung von LLMs

Datensicherheit ist für Unternehmen ein extrem wichtiges Thema und die Veröffentlichung von LLMs haben diese stark gefährdet. Samsung hat die Nutzung von LLMs verboten nachdem herausgekommen ist, dass einige Ingenieure ChatGPT für ihre Arbeit genutzt haben und dabei sensible Daten preisgegeben haben [Gur23]. Zahlreiche Banken wie die Bank of America Corp., die Deutsche Bank AG und die Goldman Sachs Group Inc. haben den Zugriff zu ChatGPT von ihren System blockiert und verbieten ihren Mitarbeitern darauf zuzugreifen [MSL23]. Microsoft und Amazon haben die Nutzung zwar nicht verboten, aber haben Warnungen an alle Mitarbeiter veröffentlicht, keine sensiblen Daten anzugeben [Siu23; Kim23].

Laut der Datensicherheitsfirma Cyberhaven haben bis Juni 2023 10.8% der Arbeitnehmer weltweit ChatGPT auf der Arbeit genutzt und 4.7% haben sensible Daten in ChatGPT eingefügt. Außerdem geben sie an, dass die Nutzung von ChatGPT am Arbeitsplatz weiterhin stark wächst trotz der Verbote vieler Firmen. Am meisten weitergegeben werden (Vorfälle je 100000 Mitarbeitern): Interne Dokumente (319), Source-Code (278), Kundendaten (260), Kundenlisten (171) und Patientendaten (146) [Col23].

Die Weitergabe dieser wichtigen Daten führt zu vielen schwerwiegenden Problemen. Die uneingewilligte Weitergabe von Kunden- und vor allem Patientendaten kann eine Straftat darstellen.

Firmeninterne Dokumente und Source-Code können von Konkurrenten genutzt werden um Marktvorteile zu erlangen. Der Betreiber des LLMs hat vollen Zugriff auf diese Daten, OpenAI speichert und nutzt ChatGPT Konversationen über die Webseite (nicht die API) [23m] zur Verbesserung von ChatGPT. Außerdem können diese Daten durch Leaks an die Öffentlichkeit gelangen sowie beim Leak am 20. März von OpenAI [23e] wo Nutzer Zugriff auf Benutzerdaten und Konversationen von anderen Nutzern hatten.

Diese Aspekte können einen gravierenden negativen Einfluss auf die Zukunft von LLMs in der ABAP Entwicklung haben. ABAP-Systeme arbeiten üblicherweise mit kritischen Daten die unter keinen Umständen an eine (unberechtigte) externe Firma oder an die Öffentlichkeit kommen dürfen. Es ist zu erwarten dass auch hier von Firmen Zugriffsverbote oder Zugriffssperren eingesetzt werden um diese Daten sicher zu halten. Mögliche Lösungsansätze hierfür werden wir in Kapitel 4.2 betrachten.

2 Vorbereitung des Benchmarks

2.1 Wie sind LLM Software-Generierungsbenchmarks aufgebaut?

Benchmarks für LLMs die Unterstützung bei der Softwareentwicklung bieten, bestehen üblicherweise aus Unittest und einem Framework um diese automatisiert auszuführen und die Ergebnisse zu analysieren.

Die Tests umfassen viele einzelne Aufgaben, die dazu dienen einen möglichst großen oder repräsentativen Bereich der zu testenden Programmiersprache abzudecken. Jeder Test wird mehrfach ausgeführt um zuverlässige Ergebnisse zu erhalten.

Ein Test besteht dabei aus:

- Einem Prompt mit der Aufgabenstellung die an das LLM gegeben wird. Hier können unterschiedliche Ansätze verfolgt werden, je nachdem was analysiert werden soll. In “Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT” [Yet+23] wurde festgestellt, dass Prompts am effektivsten sind wenn sie sowohl die erwartete Funktionssignatur als auch einen Docstring zur Erklärung der erwarteten Funktionsweise enthalten. Die Funktionssignatur sollte hierfür einen sinnvoll gewählten Funktionsnamen und die erwarteten Input-Parameter mit Datentyp sowie den Rückgabentyp der Funktion enthalten. Der Docstring enthält eine textuelle Beschreibung der Funktion und einige Beispielaufrufe mit erwartetem Ergebnis.
- Eine Musterlösung für den Funktionsaufruf wird üblicherweise zur Vollständigkeit mit angegeben. Diese kann in der Analyse helfen, die Komplexität der Aufgabe zu verstehen und Hinweise darauf geben, welche Programmelemente das LLM besonders gut oder schlecht “versteht”. Außerdem dient sie zur Überprüfung der Unittests.
- Unittests, die eingesetzt werden um die generierten Lösungen automatisiert zu testen. Bei der Programmgenerierung kann kein direkter Vergleich der generierten Lösung mit einer Musterlösung durchgeführt werden, da es in der Softwareentwicklung üblicherweise beliebig viele mögliche Lösungen für ein Problem gibt. Stattdessen werden Unittests eingesetzt um die funktionale Äquivalenz der generierten Lösung in Vergleich zur Musterlösung zu prüfen. Sie können zwar keine absolute funktionale Äquivalenz garantieren (da es nur begrenzt viele Testfälle gibt), aber Unittests haben sich in der (menschlichen) Softwareentwicklung bewährt und werden deshalb auch hier eingesetzt.

Das Framework ist ein Softwareprojekt, welches die Durchführung der Tests und die Analyse der Ergebnisse soweit wie möglich automatisiert. Es umfasst üblicherweise eine Reihe von Programmen oder Funktionen die unter anderem die Prompts aus den Testdaten lesen, die Prompts an das LLM senden, Antworten vom LLM entgegennehmen, Unittests durchführen und Ergebnisse analysieren. Das Framework sollte der Software dabei Zugriff auf alle benötigten Ressourcen ermöglichen, die wichtigsten sind hier vor allem eine API für den Zugriff auf das LLM und Zugriff auf den benötigten Compiler um die Unittests durchzuführen.

Um gute Ergebnisse zu erzielen werden in LLM-Benchmarks häufig erst kleinere Testdurchläufe mit 1% – 10% der Testdaten gemacht um gute Testparameter (Promptaufbau, Temperatur, u.ä.) zu finden bevor ein großer Testdurchlauf mit allen Testdaten und hoher Wiederholungszahl stattfindet.

2.2 Welche Anforderungen haben wir an unseren Benchmark?

An unseren Benchmark haben wir verschiedenste Anforderungen um aussagekräftige und zuverlässige Testergebnisse durch ChatGPT in Bezug zur ABAP-Programmentwicklung zu erzielen. Unter diese Anforderungen fallen unter anderem, die Reproduzierbarkeit von Benchmarks, welche bedeutet, dass die Benchmark Ergebnisse reproduzierbar und unabhängig von Zeit und Ort ein ähnliches Ergebnis hervorbringen sollen. Eine weitere Anforderung durch die Reproduzierbarkeit ist, dass die Tests bzw. der Benchmark mit gleichen Bedingungen mehrfach Durchlaufen werden können. Dieser Punkt ist wichtig, um einen statistischen Zusammenhang zu erhalten, und nachzuweisen, dass die Ergebnisse nicht auf den Zufall zurückzuführen sind.

Die Abdeckung des Testbereichs ist eine weitere Anforderung an unseren Benchmark. Durch diese stellen wir sicher, dass der Testbereich vollständig abgedeckt wird und beispielsweise verschiedene Programme erstellt werden. Der Fokus des Benchmarks wird sichergestellt, indem individuelle Tests erstellt werden, welche das spezifische Themengebiet, in unserem Fall die ABAP-Programmentwicklung testen.

Des Weiteren steht als Anforderung bereit, dass relevante Lösungen hervorgebracht werden. Diese Anforderung steht in Bezug zu der vorherigen Anforderung, der Wiederholbarkeit, da zur Erreichung von Aussagekräftigen und relevanten Lösungen eine große Anzahl an Wiederholungen der Tests notwendig ist. Zudem werden eventuell Einflussfaktoren der Tests verändert und dann wieder mehrfach Durchlaufen, um weitere Rückschlüsse und Vergleiche ziehen zu können. Ein Beispiel dafür wäre die Veränderung der Temperatur der im Benchmark gegebenen Prompts. Durch diese Tests kann dann geschaut werden, welche Ausgabe ggf. relevanter oder welche Einstellung von Parametern am effektivsten ist.

Eine weitere Anforderung ist die Genauigkeit, welche sich in unserem Fall hauptsächlich auf den Teil des Prompts im Benchmark fokussiert. Das Prompt muss präzise und genau sein und darf nicht zu komplex sein, um klare und schnelle Lösungen zu erzielen, welche im besten Fall genau die Lösung auf die gegebene Frage ist.

Eine weitere relevante Anforderung ist die Skalierbarkeit. Die Benchmarks sollten auf verschiedenen Versionen von bestimmten LLMs problemlos übertragbar sein. In unserem Beispiel wäre dies einmal die Anwendung auf der ChatGPT Version "GPT-3.5" und "GPT-4".

Die Anforderung der Sicherheit sollte ebenfalls nicht außer Acht gelassen werden, da der Code von einem LLM generiert und automatisch getestet wird muss darauf geachtet werden, dass die Benchmarks so gestaltet werden das am besten kein unsicherer Code generiert wird. Da man dies aber vermutlich nicht wirklich beeinflussen kann, sollten die Tests in einer isolierten Umgebung ausgeführt werden, wo sie im Problemfall keinen großen Schaden anrichten.

Die nächste Anforderung ist das Zeitlimit, welches wir beachten sollten, damit ineffizienter Programmcode nicht zu lange bzw. sogar unendlich läuft. Dafür wird ein Timeout in Bezug auf die Laufzeit angegeben, nach welchem das Programm abbricht und das Ergebnis als Fehler deklariert wird.

Die vorletzte Anforderung ist die der realistischen Nutzungsszenarien diese sollten die Realität abbilden, um den Einsatz in der tatsächlichen Umgebung widerzuspiegeln, testen und bewerten zu können. Als letzte Anforderung steht die Dokumentation, die Benchmark Ergebnisse sollten verschiedene relevante Daten enthalten und einheitlich dokumentiert werden. Beispiele für diese Daten wären unter anderem: Erfolg oder Fehlschlag eines Tests, Grund für Fehlschlägen, Durchlaufzeit, das generierte Programm, LLM-Version.

Alle diese Anforderung sollen dazu beitragen, relevante Ergebnisse zur erhalten, auf deren Basis man eine Aussage über die Nützlichkeit von LLMs zur Unterstützung der Programmentwicklung in ABAP treffen kann.

2.2.1 Welche Anforderungen entstehen durch ABAP?

Da ABAP eine proprietäre Programmiersprache ist, die nur innerhalb der SAP Entwicklungsumgebung läuft, entstehen Schwierigkeiten bei der Umsetzung des Frameworks zur automatisierten Ausführung der Tests. Wir benötigen weitgreifende Zugriffsrechte auf einen SAP-Server um beliebigen Code auszuführen, außerdem benötigen wir die Berechtigungen und Möglichkeiten diese Zugriffe (im besten Fall: von außerhalb der SAP Entwicklungsumgebung) zu automatisieren. Hierfür haben wir zwei Methoden gefunden, die beide besondere Zugriffsrechte vom Server voraussetzen: SAP GUI Scripting, was die Bedienung der SAP GUI automatisiert, und SAP NetWeaver Remote Function Call (SAP NW RFC), was Zugriff auf SAP Funktionsbausteine über eine externe SDK gibt.

Wir benötigen zusätzlich eine Möglichkeit unseren generierten Code, in sich geschlossen, auf dem Server auszuführen. Hierfür haben wir wieder verschiedene Ansätze gefunden: mit dem ABAP-Befehl “GENERATE SUBROUTINE POOL” kann beliebiger Text als ABAP-Code ausgeführt werden. Kombiniert mit ABAPs Objektorientierung kann dann eine Klasse mit einer Methode implementiert werden, die unabhängig aufgerufen werden kann und Eingabe- sowie Rückgabeparameter akzeptiert. Das gleiche lässt sich auch umsetzen, indem der integrierte ABAP-Funktionsbaustein “RS.FUNCTIONMODULE.INSERT” aufgerufen wird. Dieser erstellt aus Text einen Funktionsbaustein, der dann beliebig aufgerufen werden kann. Mit “Z.FUNCTION.DELETE” kann man ihn später wieder löschen.

2.2.2 Welche Anforderungen entstehen durch ChatGPT?

Um die Unterstützung von ChatGPT bei der ABAP-Programmentwicklung zu evaluieren, ist es wichtig verschiedene Anforderungen, welche an dieses LLM gestellt werden zu betrachten. Die Relevanz zur Betrachtung dieser spezifischen Anforderungen entsteht dadurch, dass bei den Benchmarks, eine konsistente und einheitliche Testumgebung geschaffen werden muss. Diese einheitliche Umgebung ist von höchster Wichtigkeit, da wenn sie nicht eingehalten wird, Ergebnisse und somit die Bewertung verzerrt werden könnte.

Anforderungen, welche in Folge unserer Tests zur Unterstützung von ChatGPT bei der ABAP-Programmentwicklung entstehen sind beispielsweise, dass wir bei jedem Tests eine neue Konversation starten. Diese Anforderung entsteht durch den Fakt, dass sich ChatGPT auf vorherige sich im Chat befindende Informationen bezieht, und diese die nächste Ausgabe beeinflussen.

Dies würde bedeuten, dass das LLM mehrere Iterationen zur Verfügung hätte und auf mehr Informationen zugreifen könnte, und somit die nächste Ausgabe (je nachdem ob die vorherigen Informationen gut oder schlecht waren) verbessert oder auch verschlechtert werden würde, was letztlich das Resultat verzerren würde.

Um das einheitliche Testen zu garantieren wollen wir somit immer nur genau einen Testdurchlauf und damit eine Ausgabe in einer Konversation erzeugen. Eine zweite wichtige Anforderung im Umgang mit ChatGPT ist die Qualität von Prompts, welche zum Beispiel die Bereiche betrifft, wie das stellen von präzisen und nicht zu komplexen Prompts.

Diese Anforderung bewirkt einerseits, dass durch Präzision genau das gemacht wird, was von dem LLM verlangt wird und es keine Abweichungen von der eigentlichen Fragestellung gibt. Die Komplexität des gegebenen Prompts sollte ebenfalls beachtet werden, da ChatGPT eine bestimmte Menge an Informationen bzw. Trainingsdaten zur Verfügung hat, und simple Prompts gegebenenfalls dem LLM mehr Freiraum bei der Einbeziehung von Daten und bei der Generierung von Antworten geben kann.

Eine weitere logische Anforderung ist das Verwenden von gleichen Prompts für die gleichen Tests, testet man beispielsweise das Erkennen von Syntaxfehlern, ist es wichtig, dass das Prompt bei verschiedenen Testdurchläufen immer gleich bleibt, um nicht unterschiedliche Ergebnisse durch unterschiedliche Prompts zu erzeugen. Dies stellt sicher, dass unterschiedliche oder auch gleiche Ausgaben nicht durch einen Eingriff in die Testumgebung, sondern allein durch das LLM entstanden sind.

2.3 Relevante Daten und ihre Messbarkeit

2.3.1 Parameter

Als Parameter bezeichnen wir alle, für den Benchmark relevanten Daten, die wir selbst festlegen. Darunter fallen Einstellungen am LLM sowie Entscheidungen über die Menge der Testfälle sowie die Anzahl der Durchläufe pro Testfall.

- Ein einfacher, aber dennoch relevanter Parameter wäre beispielsweise die Anzahl an Testfällen, welche die Basis für viele relevante Kennzahlen bildet.
- Ein ebenfalls wichtiger Parameter, welcher in Bezug zur Anzahl der Testfälle steht, ist die Anzahl an Wiederholungen pro Testfall. Diesen werden wir manuell festlegen und er soll aufzeigen, dass die Ergebnisse nicht auf den Zufall zurückzuführen, sondern statistisch nachzuweisen sind.
- Als weiteren Parameter haben wir die Temperatur identifiziert. Wir benutzen diese Option, um effizientere Ausgaben zu erreichen. Die Temperatur gibt hierbei an, wie deterministisch

die Ausgabe ist. Wobei eine höhere Temperatureinstellung eine kreativere Ausgabe hervorbringt und die Ausgabe bei einer niedrigen Temperatureinstellung vorhersehbarer ist.

- Das Systemprompt ist ein, für das LLM, besonderes Prompt das dazu dient die folgende Konversation zu beeinflussen. Es wird üblicherweise genutzt um die Antworten des LLMs in eine bestimmte Richtung zu lenken, einen Sprachstil vorzugeben, die Aufgabe einzuschränken oder genauer auszuführen. Wir werden in der Testvorbereitung verschiedene Systemprompts entwickeln um im Parameterfindungstest dann das beste Systemprompt auszuwählen.
- Als Promptoption bezeichnen wir den grundlegenden Aufbau der Prompts die wir an das LLM geben. Hierfür werden wir in der Testvorbereitung unterschiedliche Promptoptionen entwickeln und im Parameterfindungstest dann prüfen welche Promptoption die besten Antworten liefert.
- Ein weiterer Aspekt den wir betrachten ist die Sprache. Diesen Parameter berücksichtigen wir, da LLMs mit verschiedenen Daten von unterschiedlichen Quellen trainiert werden und diese sich in der Sprache unterscheiden können. Wir testen damit, ob der Umfang an Daten in verschiedenen Sprachen einen Einfluss auf die Ausgabe hat.
- Das LLM ist ein weiterer Parameter den wir festlegen. Wir werden größtenteils mit dem ChatGPT-LLM: GPT-3.5 arbeiten, aber auch einen Test durchführen um die Unterschiede in GPT-4 darzustellen. Diese Unterscheidung treffen wir da unterschiedliche LLMs mit anderen oder mit mehr Daten trainiert worden sein können, was vermutlich ebenfalls die Ausgabe beeinflussen würde.
- Zudem betrachten wir als Parameter die Softwareversion des LLMs. Diese kann Einfluss auf die Ausgabe haben, da bei unterschiedlichen Versionen gegebenenfalls Anpassungen bei den internen Parametern des LLM vorgenommen wurden. Mit der Nutzung einer festgelegten Version garantieren wir, dass keine Updates durch interne Parameteränderungen oder ähnliches die Ausgabe des LLMs beeinflussen.

2.3.2 Kennzahlen

Es existieren viele Kennzahlen welche wir benötigen um die Unterstützung durch LLMs in Bezug auf die ABAP-Programmentwicklung bewerten zu können. Hierbei handelt es sich größtenteils um allgemeine Kennzahlen die bei Testauswertungen in jedem Fachbereich betrachtet werden können. Diese umfassen Absolutkennzahlen um die genauen Werte zu messen und Relativkennzahlen um prozentuale Anteile zu betrachten.

- Ebenfalls wichtige Daten sind jeweils die Anzahl an bestandenen aber auch die Anzahl an fehlgeschlagenen Tests, welche man zum Beispiel in Bezug zur Gesamtanzahl an Testfällen oder der Gesamtanzahl setzen kann.
- Eine ähnliche relevante Kennzahl wäre die Anzahl an erfolgreichen bzw. an nicht erfolgreichen Durchläufen.
- Diese kann man in Beziehung setzen zu Anzahl an Wiederholungen pro Test, woraus sich die Prozentuale Anzahl an erfolgreichen bzw. an fehlgeschlagenen Durchläufen pro individuellen Testfall ergibt. Des Weiteren könnte man die Anzahl an Fehlgeschlagenen und erfolgreichen Durchläufe in Bezug zu der Gesamtmenge an Durchläufen betrachten.
- Man könnte auch die durchschnittliche Anzahl an Bestandenem bzw. nicht Bestandenem Testdurchläufen berechnen und mit der Anzahl an Wiederholungen pro Testfall vergleichen. Demnach wäre zum Beispiel bei einer Anzahl von 10 Wiederholungen pro Testfall und einer durchschnittlichen Bestehensquote von 60% die Anzahl an erfolgreichen Tests höher als die der fehlgeschlagenen wäre und man könnte damit z.B. sagen, dass sich das LLM bei diesem Testfall als Unterstützung eignet.
- Ebenfalls wichtig ist die pass@k Kennzahl [Che+21] die angibt, wie die Wahrscheinlichkeit ist, nach k Durchläufen einen erfolgreichen Testfall gefunden zu haben.
- Außerdem interessant könnte die Betrachtung der bestandenen Unittests je Testfall sein, hierdurch könnte man sehen welche Aufgabentypen besser oder schlechter geeignet sind und ob das LLM einen Teil der Aufgabe erfolgreich bearbeitet hat.

- Des Weiteren kann man bei den fehlgeschlagenen Durchläufen der Testfälle die Anzahl der enthaltenen Semantik bzw. der Syntax Fehler betrachten, wodurch man schlussfolgern könnte, in welchen Bereichen in Bezug auf ABAP-Programmierung vermehrt Fehler gemacht werden und falls man die Unterstützung der LLMs bei der Programmentwicklung in Betracht zieht, sich diese Bereiche nochmal genauer anschauen sollte.
- Eine sehr ähnliche Kennzahl wäre die Anzahl an gleichen Compilerfehlern, wodurch man erkennen könnte, wie oft welcher Fehler vorkommt, und somit auch wieder, in welchen Bereichen die LLM ggf. nicht effektiv ist.
- Eine weitere Kennzahl wäre die Anzahl an Iterationen der Testfälle, bis die erste korrekte Lösung des Testfalls erfolgt.
- Weitere relevante Daten wäre unter anderem die Erkennungsrate von korrektem bzw. auch von fehlerhaftem Code. Mit diesen Daten könnte man absehen, ob das LLM einfachen Code verstehen kann und zum Beispiel bei der Fehlersuche hilfreich sein kann.
- Als letzte Kennzahl wäre die Anzahl von korrekten Codeerklärungen zu nennen, welche ebenfalls erkennen lässt, ob das LLM in diesem Bereich als Unterstützung dienen kann. Diese lässt sich ohne weitere Modifikation jedoch nur manuell ermitteln.

2.4 Automatisierte Tests

2.4.1 Versuchsaufbau

Um eine zuverlässige Durchführung einer großen Anzahl von Tests zu gewährleisten haben wir einen lokalen ABAP-Server aufgesetzt auf dem die Antworten vom LLM ausgeführt werden können. Er läuft innerhalb einer Virtuellen Maschine (VM) von Oracle VM VirtualBox [23i] die Zugriff auf 4 virtuelle Prozessorkerne, 8GB Arbeitsspeicher und 100GB SSD-Festplattenspeicher hat. Die Einrichtung basiert weitgehend auf der Installationsanleitung [SAP23c] von SAP wobei an mehreren Stellen Abweichungen stattfinden mussten um unter anderem neuere Softwareversionen nutzen zu können oder die SAP-Lizenzdatei durch eine neuere zu ersetzen. Darauf aufbauend haben wir openSUSE Leap 15.5 [Cyn23] als Betriebssystem, und das SAP Softwarepaket “SAP NetWeaver AS ABAP Developer Edition 7.52 SP04” [SAP23a] mit einer kostenlosen Testlizenz installiert.

Das Softwarepaket enthält den relationalen Datenbankserver “SAP Adaptive Server Enterprise (ASE) 16.0”, einen darauf basierenden ABAP-Server der mit Fiori-Launchpad, SAP Cloud Connector, der SAP Java Virtual Machine, Backend und Frontend Verbindungsschnittstellen, Nutzerrollen, Beispielanwendungen sowie ABAP Standardinfrastruktur wie Transaktionsmanagement, Datenbankoperationen, Änderungs- und Transportsysteme und den SAP Gateway.

Für unseren Aufbau relevant sind vor allem die grundlegenden Funktionalitäten des Datenbank- und ABAP-Servers und die SAP Netweaver Remote Function Call (SAP NW RFC) Schnittstelle. SAP NW RFC ist eine externe Schnittstelle des Servers die dazu dient SAP Systeme zu verbinden, wir nutzen sie um ABAP Funktionsbausteine von außerhalb des SAP-Systems aufzurufen und so die Ausführung von beliebigem Code zu automatisieren. Wir nutzen das lokal installierte Softwareentwicklungspaket “SAP NetWeaver RFC SDK” [23l] um eine C/C++ Schnittstelle aufzubauen und das Python Paket “PyRFC” [23k] um über diese Schnittstelle mit dem ABAP-Server zu kommunizieren und Funktionsbausteine aus unserem Python-Programm aufzurufen.

Unser Programm ist grundlegend in zwei Teilaufgaben aufgeteilt:

1. Im ersten Schritt werden alle Versuchsvarianten, Testfälle und Wiederholungen durchlaufen und dementsprechende Anfragen an ChatGPT gesendet. Die Antworten werden gespeichert um im zweiten Teil weiter verwendet. Dieser Ablauf ist in Abbildung 8 dargestellt.

Unsere Kommunikation mit ChatGPT erfolgt über die OpenAI API mit dem Python Paket “openai” [23h]. Diese ermöglicht es uns automatisiert Anfragen zu stellen und Einstellungen über Temperatur, Systemnachrichten und den Kontext eines bestehenden Chatverlauf zu treffen. Die richtigen Einstellungen für Temperatur und eine Systemnachricht werden wir im weiteren Verlauf durch kleine Testdurchläufe evaluieren und dann für weitere Tests nutzen. Wir haben uns dazu entschieden, keinen bestehenden Chatverlauf als Kontext zu nutzen und stattdessen für jeden Testdurchlauf einen komplett neuen und unabhängigen Chat zu nutzen, um möglichst objektive Ergebnisse zu erzielen.

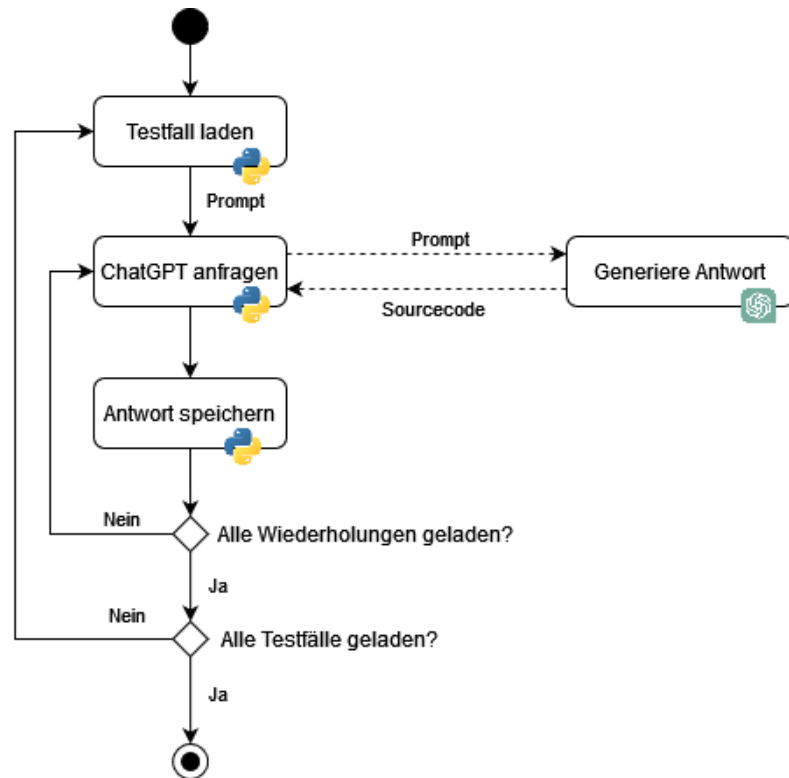


Abbildung 8: Versuchsaufbau: Antwortgenerierung durch ChatGPT

2. Im zweiten Schritt durchlaufen wir die zuvor generierten Antworten des LLMs und führen sie auf dem ABAP-Server aus. Die Ausführung auf dem ABAP-Server läuft nach dem folgenden Prinzip ab und ist in Abbildung 9 dargestellt.
 - (a) Aufruf des ABAP Funktionsbausteins “RS.FUNCTIONMODULE_INSERT” um aus ABAP-Sourcecode einen eigenen Test-Funktionsbaustein zu erstellen. Dabei müssen einige Eigenschaften des Test-Funktionsbausteins wie der Name, die Import- und Exportparameter sowie ihre Typen als separate Argumente vom eigentlichen Sourcecode übergeben werden. Da der Sourcecode von einem LLM generiert wurde und reiner Text ist, werden die benötigten Informationen hier in Python über String-manipulation extrahiert.
 - (b) Aufruf des neu erstellten Test-Funktionsbausteins. Um den Test-Funktionsbaustein vollautomatisiert zu testen, werden Unittests eingesetzt. Hierbei wird der Test-Funktionsbaustein, je nach Testfall, mehrmals mit verschiedenen Import-Parametern aufgerufen und der Export-Parameter mit der erwarteten Antwort verglichen. Hier haben wir in den RFC-Funktionsaufruf zusätzlich einen Timeout von 10 Sekunden eingebaut, dieser verhindert Endlosschleifen und zu langlaufende Programme.
 - (c) Aufruf des selbst erstellten Funktionsbaustein “Z_FUNCTION_DELETE”, der den Test-Funktionsbaustein wieder löscht. Der Test-Funktionsbaustein muss nach dem Testdurchlauf wieder gelöscht werden, damit weitere Testdurchläufe wieder in einer “sauberen” Umgebung laufen und keine Fehler entstehen können, wie beispielsweise durch die Wiederverwendung des gleichen Funktionsnamens. Unser Funktionsbaustein “Z_FUNCTION_DELETE” greift direkt auf den Standard ABAP-Funktionsbaustein “FUNCTION_DELETE” zu, der Umweg über den selbstgeschriebenen Funktionsbaustein ist hier nötig, da “FUNCTION_DELETE” nicht Remote-Enabled ist und somit nicht über NW RFC aufgerufen werden kann.

2.4.2 Testvorbereitung

Wir haben uns dafür entschieden unseren Benchmark aufbauend auf dem HumanEval Datensatz [Che+21] zu gestalten da dies ein beliebter Ansatz in der wissenschaftlichen Literatur ist und einen

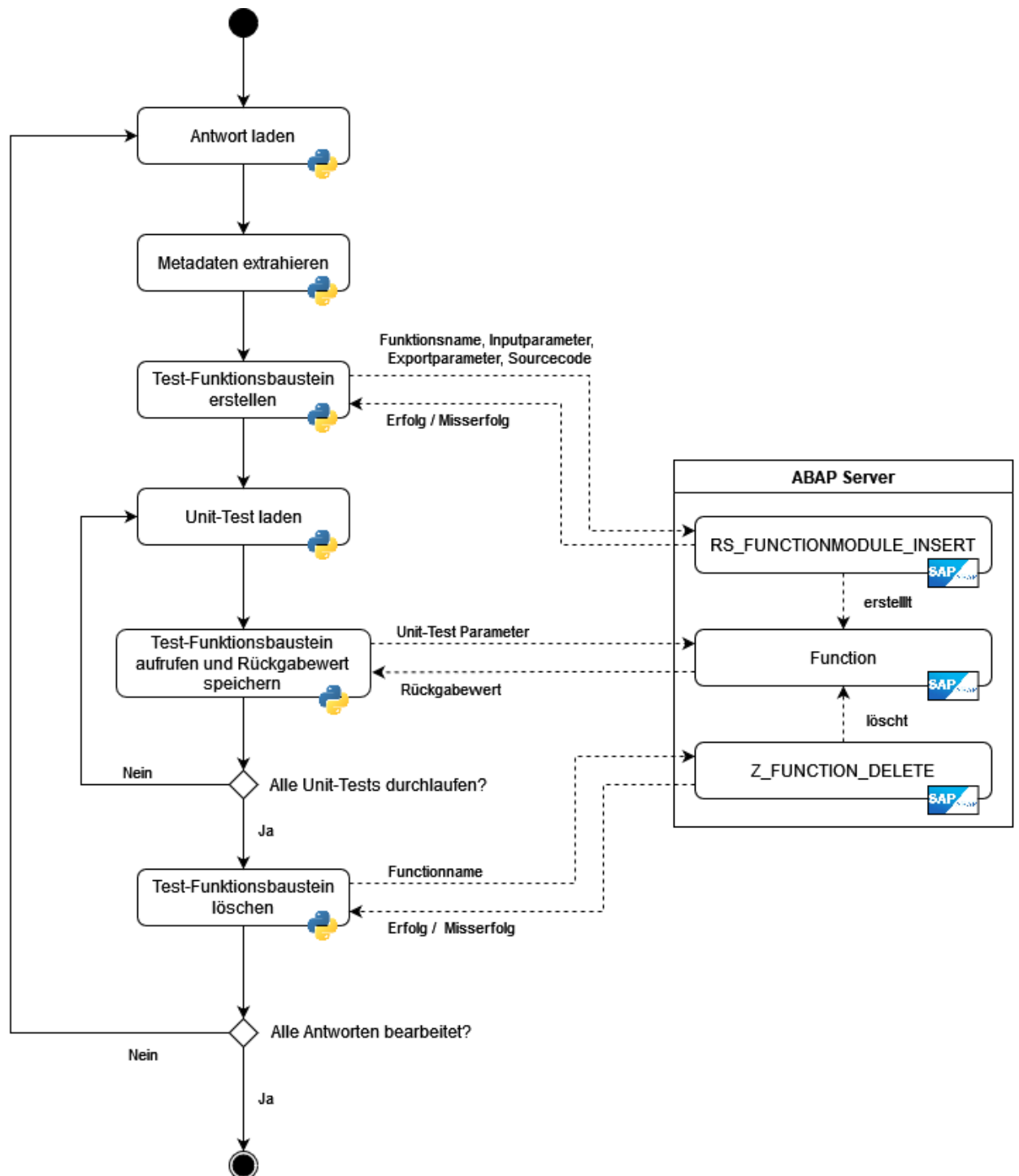


Abbildung 9: Versuchsaufbau: Ausführung der Antworten in ABAP

Vergleich der Ergebnisse zu bereits bestehenden Auswertungen von anderen Programmiersprachen ermöglicht. Beispielhaft ist Prompt 0 aus dem HumanEval Datensatz in Abbildung 10 dargestellt.

Damit haben wir eine Grundlage von 164 Prompts mit Python-Funktionssignatur, Funktionsparameter und Rückgabeparameter inklusive ihrer Pythontypen sowie einem Docstring mit einer Erklärung und Beispielaufrufen mit erwarteten Rückgabewerten. Außerdem haben diese je Testfall eine Python-Testfunktion, welche durch Unittests die Korrektheit der Funktion sicherstellt.

Zur Durchführung der Tests mussten wir einige Änderungen an den Testfunktionen durchführen:

- Jedes “assert” in den Testfunktionen wird in einem Try-Catch-Block ausgeführt, so können

HumanEval Prompt 0
<pre>Prompt def has_close_elements(numbers: List[float], threshold: float) -> bool: """ Check if in given list of numbers, are any two numbers closer to each other than giventhreshold. >>> has_close_elements([1.0, 2.0, 3.0], 0.5) False >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3) True """</pre>
<pre>Solution for idx, elem in enumerate(numbers): for idx2, elem2 in enumerate(numbers): if idx != idx2: distance = abs(elem - elem2) if distance < threshold: return True return False</pre>
<pre>Test METADATA={ 'author': 'jt', 'dataset': 'test' } def check(candidate): assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False</pre>

Abbildung 10: Beispielhafte Darstellung vom Prompt 0 des HumanEval Datensatzes

wir alle Testfälle automatisch hintereinander ausführen und Erfolge sowie Misserfolge zählen.

- Einige der HumanEval Testfunktionen enthielten “assert True” Aussagen welche zum Debugging genutzt wurden. Diese haben zwar keinen Einfluss auf das endgültige Bestehen des gesamten Testfalls, aber bei der nachträglichen Analyse könnte man durch sie fälschlicherweise auf eine Teillösung schließen, weshalb wir uns dafür entschieden haben sie aus den Tests zu entfernen.
- Die Prompts 32, 38 und 50 enthielten jeweils eine bereits geschriebene Python-Hilfsfunktion, die das LLM nutzen sollte um seine eigentliche Lösung zu implementieren. Da diese Hilfsfunktionen auch in den Testfällen benötigt werden, mussten wir sie in die jeweiligen Tests hinzufügen.
- Die Testfunktion zu Prompt 33 vergleicht die Ergebnisse direkt mit denen der beiliegenden Musterlösung, statt mit vordefinierten Antworten. Hier mussten wir die Musterlösung als aufrufbare Funktion im Tests bereitstellen.

Die Entwicklung der automatischen Tests auf <https://github.com/timkoehne/Praxisprojekt-ABAP-und-ChatGPT> zu finden.

2.4.3 Testplanung: Parameter finden

Um möglichst aussagekräftige Ergebnisse zu erreichen werden wir unsere Tests mit einem kleineren Testdurchlauf beginnen, der dazu dient die korrekten Test-Parameter zu finden. Sie werden in diesem Testdurchlauf bestimmt und dann für alle weiteren Tests genutzt.

Wir haben drei Testparameter festgelegt die wir in dieser Untersuchung betrachten wollen:

Das Systemprompt ist eine Anweisung an das LLM wie es sich im weiteren Verlauf der Konversation verhalten soll. Hiermit kann die “Persönlichkeit” des LLMs verändert werden oder spezielle

Antwortmustern definiert werden. Wird kein Systemprompt gesetzt, oder über die Webseite auf ChatGPT zugegriffen wird vermutlich [23h] das Systemprompt “You are a helpful assistant.” am Anfang jeder Konversation gesetzt.

Wir nutzen das Systemprompt um dem ChatGPT seine grundlegende Aufgabe mitzuteilen, nämlich einen ABAP Funktionsbaustein zu generieren (“generate a abap function module”). Da die Antworten so jedoch häufig textuelle Erklärungen und Beispiele enthalten nutzen wir das Systemprompt außerdem um dem LLM mitzuteilen das diese unerwünscht sind (“no explanations or examples.”). Unser Versuchsaufbau erfordert den Sourcecode von Funktionsbausteinen inklusive dem Interface Kommentar mit “IMPORTING” und “EXPORTING” für die Parameter, durch diese Erklärung wird das Systemprompt länger und komplexer. Aus unserer anekdotischen Erfahrung heraus funktionieren kurze, einfache Prompts jedoch oft besser als lange, genauere Prompts. Um dies auf die Probe zu stellen werden wir die folgenden drei Systemprompts unterschiedlicher Länge testen.

- “generate a abap function module. no explanations or examples.”
- “generate a abap function module including the interface comment. no explanations or examples.”
- “generate a abap function module including the interface comment with IMPORTING and EXPORTING. no explanations or examples.”

Als zweiten Parameter wollten wir die LLM Temperatur analysieren. Die Temperatur hat einen hohen Einfluss auf die Ergebnisse von LLM-Antworten und diese zu optimieren kann viele Testdurchläufe und große Datenmengen erfordern. Da wir hier erst einmal eine grobe Einschätzung der Fähigkeiten von ChatGPT für die ABAP-Entwicklung machen wollen, werden wir uns nur auf drei verschiedene Temperaturen beschränken: 0, 0.8 und 2. Laut [Che+21] ist 0.8 die beste Temperatur zur Python Codegenerierung bei vielen Wiederholungen vom LLM Codex (das auch auf der OpenAI GPT Architektur basiert). Somit ist 0.8 unser Startpunkt und wir nutzen die beiden Extrema 0 und 2 um letztendlich Schlüsse ziehen zu können, in welche Richtung die Temperatur in weiteren Tests angepasst werden sollte.

Unsere Betrachtung wird außerdem den Promptaufbau als dritten Parameter enthalten. Die Prompts von HumanEval [Che+21] enthalten Python-Funktionssignaturen, Python-Typen und Beispielauf-rufe der Funktion. Da unsere Analyse jedoch in ABAP stattfindet wollen wir drei unterschiedliche Promptoptionen betrachten um herauszufinden ob dieser Promptaufbau in ABAP auch sinnvoll ist oder das LLM durcheinander bringt. Hierfür werden wir zuerst die ursprünglichen, unveränderten HumanEval [Che+21] Prompts testen. Dann werden wir abgeänderte Prompts testen, die nur den Docstring enthalten und keine Python-spezifischen Code-Bauteile oder den Funktionsaufruf beinhalten. Darüber hinaus werden wir auch Prompts testen, die die Funktionssignatur und den Docstring enthalten, jedoch keine Beispielauf-rufe.

Wir kommen also auf 27 verschiedene Testfälle (3 Systemprompts * 3 Temperatureinstellungen * 3 Promptoptionen). Außerdem werden wir eine zufällige Stichprobe von 40 der 164 Prompts auswählen die repräsentativ in diesen (und einige spätere Tests) genutzt wird um die Testmenge für diesen Test gering zu halten. Für jedes Prompt werden drei Versuchsdurchläufe gemacht. Somit kommen wir hier auf $27 * 40 * 3 = 3240$ Funktionsbausteine die wir generieren lassen und ausführen.

2.4.4 Testplanung: Funktionsbausteine generieren

Unser umfangreichster Test ist die Funktionsbaustein-Generierung. Hier werden wir alle 164 Prompts mit je 100 Wiederholungen durchführen für insgesamt 16400 Testdurchläufe. Hierfür werden wir die Parameter einsetzen, die wir vorher in der Parameterbestimmung herausgefunden haben.

2.4.5 Testplanung: Unterschiede deutsche und englische Prompts

OpenAI hat für ChatGPT keine Statistiken zur Sprachherkunft der Trainingsdaten veröffentlicht, jedoch wissen wir, dass ChatGPT auf GPT-3 basiert, wofür es diese Statistiken gibt. Demnach sind 92.65% der Wörter in den Trainingsdaten englisch, und nur 1.47% der Wörter deutsch [Git23b]. Aus diesem Grund haben wir uns dafür entschieden, den Großteil der Tests auf Englisch durchzuführen.

Trotzdem wollen wir auch einen Test auf deutsch durchführen um ChatGPTs Fähigkeiten in der deutschen Sprache zu testen. Da ABAP ursprünglich von einem deutschen Unternehmen entwickelt wurde, gibt es zu ABAP vielleicht verhältnismäßig mehr Trainingsdaten auf Deutsch was dann eventuell zu besseren Antworten auf deutsch führen könnte.

Der deutschsprachige Test wird aufgebaut sein aus der selben Stichprobe mit 40 Prompts aus der Parameterbestimmung und auch hier werden wir 3 Wiederholung pro Prompt durchführen. Die Prompts werden wir vorher manuell auf deutsch Übersetzen und die Parameter einsetzen, die sich in der Parameterbestimmung als am besten erwiesen haben.

2.4.6 Testplanung: GPT-4

Außerdem wollen wir die Ergebnisse von ChatGPT noch mit der neueren Version GPT-4 vergleichen, auf welche wir über die OpenAI API auch Zugriff haben. GPT-4 wird im Internet generell als besser in nahezu jeder Disziplin beschrieben, es gibt jedoch noch nicht viele ausführliche Tests die dies wissenschaftlich darstellen. Für die Python Programmierung hat die Auswertung von OpenAI in “GPT-4 - Technical Report” [Ope23] gezeigt, dass GPT-4 64% der Aufgaben des HumanEval Benchmarks lösen konnte, im Gegensatz zu 48.1% von GPT-3.5.

Für unseren Test werden wir wieder die selbe Stichprobe von 40 Prompts nutzen, 3 Wiederholungen pro Prompt durchführen und die Ergebnisse dann mit dem Durchschnittsergebnis von ChatGPT vergleichen.

2.5 Manuelle Tests

Neben den automatisierten Tests erstellen wir zudem manuelle Tests, welche nicht automatisch ausgewertet werden können. Diese Tests führen wir durch, da durch sie weitere wichtige Aussagen über die Unterstützung von LLMs (in unserem Fall über ChatGPT) bei der ABAP-Programmentwicklung getroffen werden können.

Um eine repräsentative Auswahl von Aufgaben für die manuellen Tests sicherzustellen, werden zufällig fünf HumanEval Musterlösungen aus dem Lösungspool ausgewählt. Diese Musterlösungen übersetzen wir dann manuell nach ABAP, um die Korrektheit dieser zu garantieren.

Jede dieser Musterlösungen wird anschließend fünfmal im spezifischen Test wiederholt, um eine bessere Vergleichbarkeit der Ergebnisse zu gewährleisten und potenziell zufällige Ergebnisse auszuschließen.

Somit kommen wir auf eine Anzahl von 25 Ergebnissen pro Testfall und durch die drei verschiedenen Tests auf eine Gesamtanzahl von 75 Ergebnissen.

Uns ist bewusst, dass dabei das Ausmaß der Tests und somit die letztliche Aussagekraft nicht an die Anzahl der automatisierten Tests herankommen kann, aber wie bereits erwähnt, dass auch wenn es sich um eine kleine Stichprobe handelt, diese Tests und ihre Ergebnisse trotz dessen eine hohe Relevanz besitzen da es auf diese Weise nutzen. Ein größerer Umfang ist im Kontext unseres Projekts jedoch nicht möglich, da diese Tests nicht automatisieren lassen.

Der Ablauf dieser Tests gestaltet sich immer gleich, damit einerseits eine Konsistenz entsteht und andererseits keine Abweichungen der Testumgebung stattfindet.

Dies wird erreicht, indem ein neues Chatfenster von ChatGPT bei jedem individuellen Testdurchlauf geöffnet wird, um eine Bezugnahme auf vorherige Ausgaben zu vermeiden. Zusätzlich wird die gesamte Ausgabe gespeichert, um spätere Abgleiche zu ermöglichen.

Die Sprache, in der die Prompts für die manuellen Tests definiert werden, ist Englisch. Dies wird aufgrund der Annahme gewählt, dass eine größere Datenmenge in dieser Sprache verfügbar ist, was potenziell zu besseren Lösungen führen könnte.

Diese spezifischen Tests teilen sich in drei Bereiche auf.

Der erste Test befasst sich damit, wie gut ChatGPT bestimmten ABAP-Code erklären kann. Wir lassen ChatGPT bei diesem Test jede Code Musterlösung fünfmal mit dem gleichbleibenden Prompt “Explain this ABAP code: <Code>” erklären. Diese Erklärung wird dann wie bereits genannt, für jeden Durchlauf separat abgespeichert.

Beim zweiten Tests bauen wir in jede der fünf Musterlösungen einen syntaktischen Fehler ein. Das LLM soll dann diesen eingebauten Fehler finden. Hierbei benutzen wir für jeden Durchlauf den Prompt: “What is the mistake in this code <Code>”.

Der letzte Test prüft, ob das LLM überhaupt syntaktisch korrekten Code erkennen kann. Hierfür geben wir ihm einfach die von uns erstellten Musterlösungen und gucken, ob ein nicht existierender Fehler gefunden wird, oder ob die LLM die Korrektheit des Codes erkennt. Ebenfalls verwenden wir für diesen Test einen gleichbleibenden Prompt, “Are there any syntax errors in the given code?”

<Code>” lautet.

Anzumerken ist, dass die Tests in keiner festen Reihenfolge erfolgen müssen und die Ergebnisse dieser individuell und nicht in Relation zueinander stehen.

Der Aufbau unseres gesamten Benchmarks ist in Abbildung 11 dargestellt.

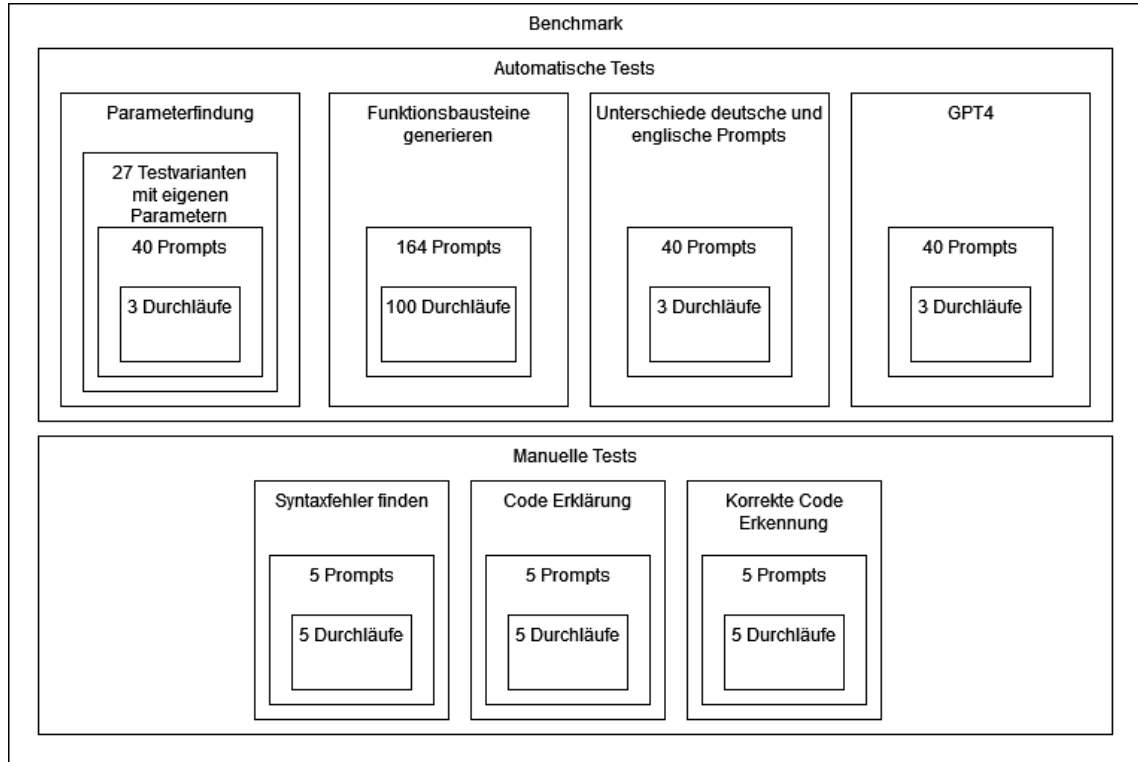


Abbildung 11: Aufbau unseres Benchmarks

3 Durchführung des Benchmarks

Alle Anfragen an ChatGPT wurden im Zeitraum 20. Juli 2023 bis 10. August 2023 gemacht, indem nach OpenAI zwei Versionsänderungen von ChatGPT stattfanden, diese sollten jedoch keinen Einfluss auf das LLM haben sondern nur auf die Webseite und die App [23a].

3.1 Automatische Tests

3.1.1 Testergebnis: Parameter finden

Bei der Durchführung der Parameterfindung hat sich herausgestellt, dass insgesamt nur wenig erfolgreiche Durchläufe gab. Von 3240 Testdurchläufen wurden 2138 Funktionsbausteine erfolgreich erstellt aber nur 17 Tests erfolgreich abgeschlossen.

In Abbildung 12 ist die Verteilung der erfolgreichen Durchläufe je Parameterkombination dargestellt. Demnach haben die drei folgenden Parameterkombinationen mit jeweils drei erfolgreichen Durchläufen die höchste Bestehenszahl.

- Systemprompt 1: “generate a abap function module. no explanations or examples.”, Prompt-Option: Standard HumanEval Prompt, Temperatur: 0
- Systemprompt 2: “generate a abap function module. no explanations or examples.”, Prompt-Option: HumanEval Prompt ohne Beispiele, Temperatur: 0
- Systemprompt 3: “generate a abap function module including the interface comment. no explanations or examples.”, PromptOption: HumanEval Prompt ohne Funktionsaufruf und ohne Beispiele, Temperatur: 0

Successful Runs				
Systemprompt 1		temperature		
		0	0,8	2
promptOption	default	3	1	0
	only text	0	0	0
	without examples	3	1	0
Systemprompt 2		temperature		
		0	0,8	2
promptOption	default	0	0	0
	only text	3	1	0
	without examples	1	1	0
Systemprompt 3		temperature		
		0	0,8	2
promptOption	default	0	0	0
	only text	0	1	0
	without examples	2	0	0

Abbildung 12: Anzahl erfolgreicher Testdurchläufe je Parameterkombination

Da unsere Daten keine eindeutig beste Parameterkombination hergeben und vor allem weil insgesamt sehr wenig Testdurchläufe bestanden wurden, haben wir uns dafür entschieden die Ergebnisse je Parameter zu summieren und den besten Wert je Parameter unabhängig von den anderen Parametern zu betrachten. Hierdurch entsteht die Verteilung aus Abbildung 13, nach der die besten Parameter folgende sind:

- Systemprompt: “generate a abap function module. no explanations or examples.”
- Temperatur: 0
- PromptOption: HumanEval Prompt ohne Funktionsaufruf und ohne Beispiele

Mit diesen Parametern werden wir in den folgenden Tests weiterarbeiten.

systemprompt	success
1	8
2	6
3	3

temperature	success
0	12
0,8	5
2	0

promptOption	success
default	4
only text	5
without examples	8

Abbildung 13: Summe der Ergebnisse je Parameter

3.1.2 Testergebnis: Funktionsbausteine generieren

Beim Durchlauf des großen Tests mit allen 164 Prompts und jeweils 100 Wiederholungen pro Prompt kann man ganz klar erkennen dass ChatGPT mit diesem ABAP Aufgaben nicht gut klar kommt. In Abbildung 14 sind die Ergebnisse der Funktionsaufrufe dargestellt.

Hierbei wurden für jeden der 16400 Testdurchläufe alle Unittests ausgeführt, oder solange bis eine Exception geworfen wurde. Wir kommen zu einer Erfolgsquote von Grade einmal 2.45%, weit unter anderen Ergebnissen die sich mit Python beschäftigt haben (Vergleich Abbildung 15).

159 der 164 Prompts wurde nie gelöst, die erfolgreichen Durchläufe kommen daher, das die Prompts 13, 53, 55 und 97 jeweils 100 mal erfolgreich gelöst wurden. Außerdem wurde Prompt 79 einmal erfolgreich gelöst. Hier fällt das “Problem” auf, dass wir mit einer Temperatur von 0 getestet haben. Die Lösungen haben sehr wenig Variation und führen deshalb nahezu immer zum gleichen Ergebnis.

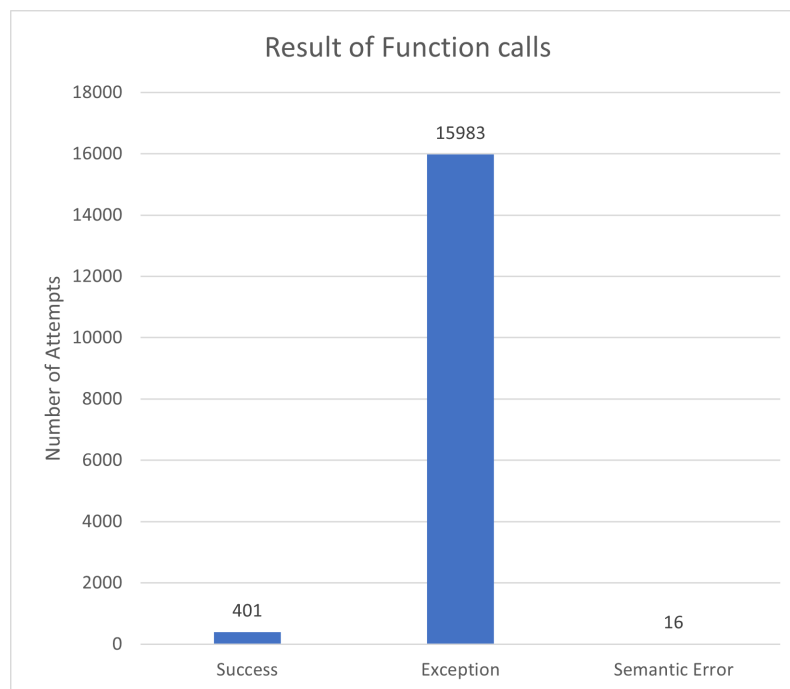


Abbildung 14: Ergebnis der Funktionsaufrufe von allen Prompts mit 100 Wiederholungen

In Abbildung 16 sind die aufgetretenen Fehlermeldungen dargestellt, die bei der Ausführung entstanden sind. Hier eine genauere Beschreibung mit Beispielen der aufgetretenen Fehlermeldungen:

- **SYNTAX_ERROR**
Der häufigste Fehler dieser umfasst alle Syntaxfehler die ABAP erkennt. Aufgrund unseres Versuchsaufbaus ist es uns hier leider nicht möglich eine genauere Kategorisierung der Syntaxfehler vorzunehmen, da über die Remote-Verbindung keine Details über die Fehlerherkunft zurückgegeben werden. Eine Lösung hierfür ist einer der wichtigsten Schritte zur Verbesserung unserer Testmöglichkeiten, vor allem weil der ABAP-Server genauere Details über die Fehlerherkunft hat und diese über die, manuelle, SAP-GUI auch ausgibt und Lösungsvorschläge anbietet.
- **ApplicationError FU_NOT_FOUND**
Dieser Fehler entsteht wenn ein Funktionsbaustein aufgerufen wird, der nicht existiert. Dies passiert hier vor allem dadurch dass die Erstellung des Funktionsbausteins fehlschlägt und dieser dann beim Aufruf nicht existiert.
- **ExternalRuntimeError(field 'x' not found)**
Dieser Fehler kann von der RFC-Schnittstelle geworfen werden, dies passiert hier manchmal wenn der Wert für einen Parameter gesetzt wird, obwohl der Funktionsbaustein keinen Parameter mit diesem Namen kennt.

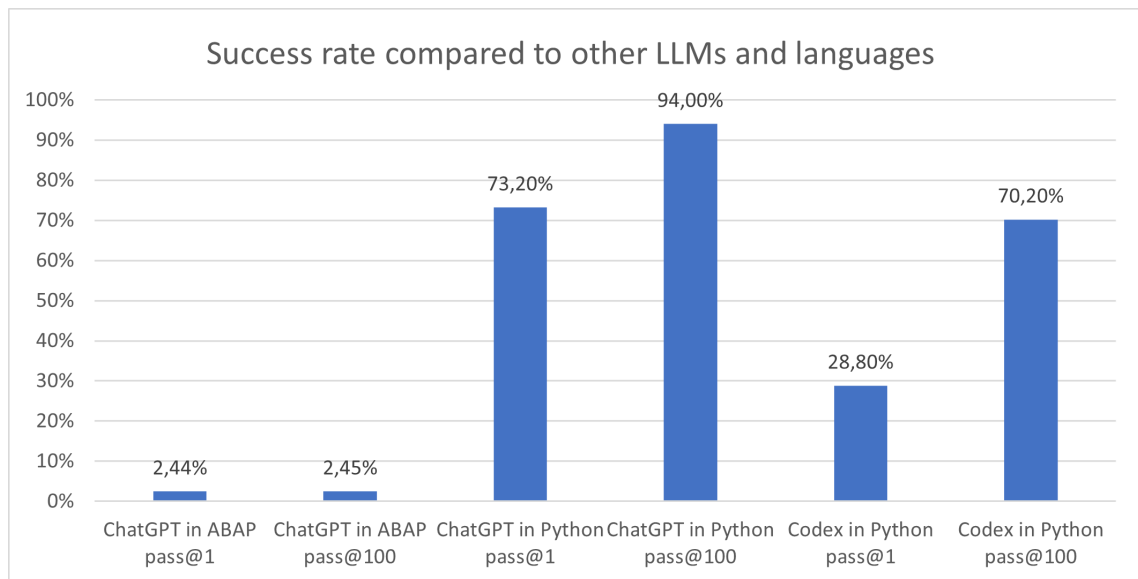


Abbildung 15: Vergleich unserer Ergebnisse mit Codex in Python [Che+21] und ChatGPT in Python [Liu+23]

- **ApplicationError NOT_FOUND**
Diese Fehlermeldung entsteht beim Aufruf einer Funktion wenn ein Import- oder Exportparameter einen ungültigen Typ hat, dies passiert hier meistens weil ChatGPT einen Datentyp als “table of i” o.ä. angibt. Dies ist zwar ein valider Datentyp in Data-Bereich im Funktionskörper, jedoch nicht im Interface des Funktionsbausteins. Hier muss stattdessen beispielsweise der vordefinierte Datentyp “int4_table” genutzt werden.
- **STRING_OFFSET_TOO_LARGE**
Dieser Fehler wird geworfen wenn mit String-Offset gearbeitet wird und der Offset die Länge des Strings überschreitet.
- **CALL_FUNCTION_NOT_FOUND** Diese Fehlermeldung tritt hier aus zwei unterschiedliche Gründen auf. Zum einen versucht ChatGPT teilweise in seiner Implementation andere Funktionsbausteine aufzurufen, die nicht existieren. Außerdem entsteht diese Exception dadurch, das ChatGPT versucht mehrere Funktionen direkt hintereinander zu implementieren. Dies ist aufgrund des Versuchsaufbaus nicht möglich da der zweite Funktionsbaustein dann innerhalb des ersten steht. Es ist aber auch manuell über den SE37 “Function Builder” in der SAP-GUI nicht möglich die beiden Funktionsbausteine in einem Text zusammen anzulegen, sondern man müsste sie einzeln erstellen.
- **COMPUTE_POW_RANGE**
Hier wurde die arithmetische Operation ** zur Potenzberechnung genutzt und das Ergebnis dieser Operation ist zu groß für den dafür vorgesehenen Datentyp.
- **COMPUTE_INT_TIMES_OVERFLOW**
Hier wurde die arithmetische Operation * zur Multiplikation genutzt und das Ergebnis dieser Operation ist zu groß für den dafür vorgesehenen Datentyp.
- **CALL_FUNCTION_PARM_UNKNOWN**
Dieser Fehler wird geworfen weil ChatGPT für die Lösung von Prompt 162 den ABAP Funktionsbaustein “MD5_CALCULATE_HASH_FOR_CHAR” zur Berechnung eines MD5-Hashes einsetzt. Es scheint jedoch Schnittstelle des Funktionsbausteins nicht zu kennen und versucht unbekannte Parameter an den Funktionsbaustein zu übergeben.
- **CONVT_NO_NUMBER**
Hier wird ein nicht-numerischer Typ wie eine Zahl behandelt, dies passiert bei uns meistens wenn der Unittest einen String als Parameter an den Funktionsaufruf gibt, der Funktionsbaustein jedoch fehlerhaft ist und einen Integer erwartet.

- **REPLACE_INFINITE_LOOP**

Zur Lösung von Prompt 2 wird das ABAP Statement “Replace all occurrences of” genutzt um Leerzeichen zu ersetzen. Das Replace-Statement ignoriert jedoch Leerzeichen wenn die Eingabe ein Datentyp variabler Länge ist, wie hier wo meistens Strings genutzt werden. Also wird versucht jedes vorkommen des leeren Strings zu ersetzen, was diese Fehlermeldung wirft, da dieser unendlich oft vorkommt.

- **CALL_FUNCTION_CONFLICT_TYPE**

Hier wird versucht einen bereits existierenden ABAP-Funktionsbaustein aufzurufen, jedoch ist der Import- oder Exportparameter von einem falschen Datentyp.

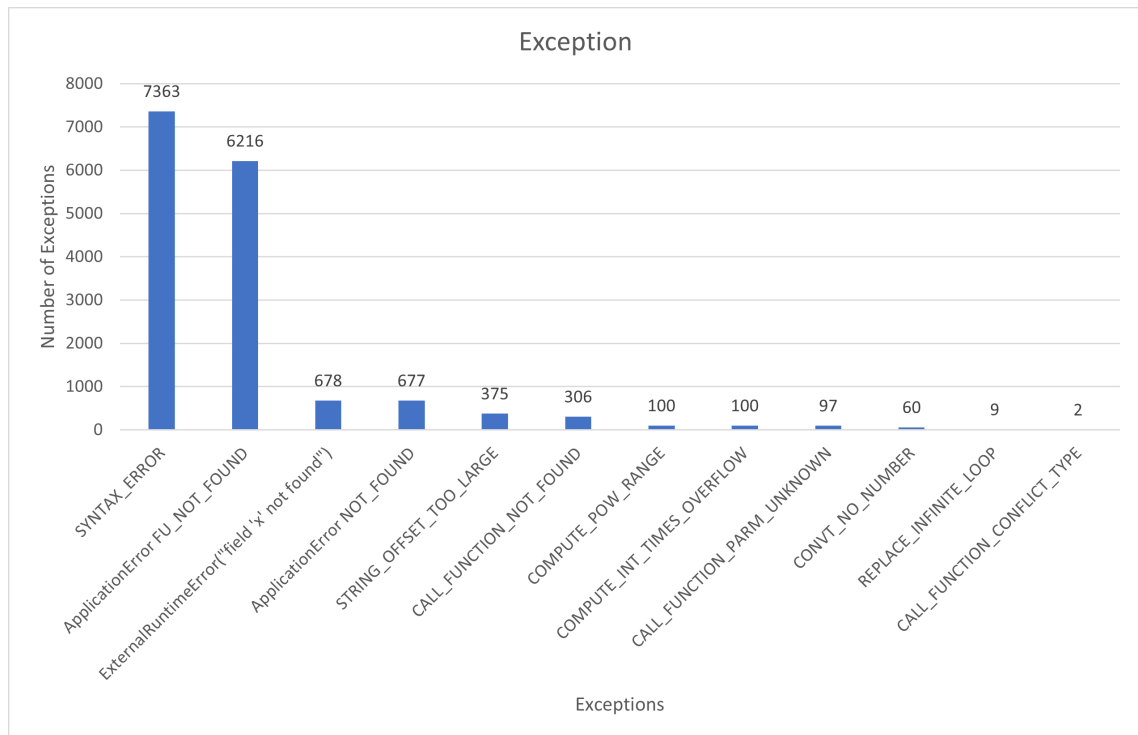


Abbildung 16: Liste der aufgetreten Exceptions. Es handelt sich überall um ABAP RunTimeError außer wenn explizit ApplicationError oder ExternalRuntimeError davor steht.

3.1.3 Testergebnis: Unterschiede deutsche und englische Prompts

ChatGPT zeigt, bei unserem Testumfang keinen relevanten Unterschied zwischen deutschen und englischen Aufgabenstellungen. Bei der Betrachtung von 40 Prompts mit je drei Wiederholungen hat ChatGPT drei von 120, also 2.5% der Aufgaben erfolgreich gelöst. Dies entspricht nahezu exakt dem Ergebnis des großen Testdurchlaufs in englisch bei dem insgesamt 2.45% gelöst wurden. Die kleine Abweichung lässt sich aufgrund des Testumfangs nicht genauer herleiten und wir müssten einen signifikant größeren Test durchführen um genauere Ergebnisse festzustellen.

3.1.4 Testergebnis: GPT-4

Im Testdurchlauf mit GPT-4 wurden 40 Prompts mit je drei Wiederholungen ausgeführt, es wurde kein Testdurchlauf erfolgreich beantwortet. Dies ist unerwartet, da GPT-4 generell als besser in nahezu jeder Disziplin angesehen wird als ChatGPT, und selbst nach OpenAI in der Python-Programmierung besser sein soll [Ope23].

Ein Grund für dieses unerwartete Ergebnis ist das Vorhandensein des Interface-Kommentars zu Beginn des ABAP-Funktionsbausteins. Wie in Abbildung 17 zu sehen, generiert GPT-4 nur in 14.17% der Antworten einen korrekten Interface-Kommentar, in Vergleich dazu wird dieser von ChatGPT in 69.93% der Antworten generiert. Häufig lässt GPT-4 hier den Kommentar komplett weg und versucht die Eingabe- und Ausgabeparameter stattdessen über eine einfach “DATA” Anweisung zu erstellen, teilweise generiert es auch die Interface-Kommentar-Schlüsselworte “IMPORTING”

und “EXPORTING” innerhalb des Funktionskörpers.

Außerdem haben wir für GPT-4 die gleiche Parameter genutzt, die wir für GPT-3.5 als am besten identifiziert haben. Da es sich um ein anderes LLM handelt, sind diese vermutlich nicht optimal.

3.1.5 Zusammenfassung: Automatische Tests

Insgesamt haben die automatischen Tests gezeigt, dass ChatGPT in ABAP schlechte Ergebnisse liefert und derzeit kein sinnvolles Werkzeug zur Programmgenerierung ist. Mit einer durchschnittlichen Erfolgchance von nur 2.45% müssten Entwickler viel mehr Zeit in die Generierung der Lösungsvorschläge und vor allem in die Überprüfung und das Testen investieren, als wenn sie die Aufgabe selbst Lösen. Außerdem wurden viele der Aufgaben von ChatGPT nie gelöst.

Unterschiedliche Parameter scheinen nur einen sehr geringen Einfluss auf das Ergebnis zu haben. Trotzdem halten wir eine größer angelegte Parameterfindung, zur Weiterführung unserer Analyse, für sinnvoll, da wir hier nur wenige Wiederholungen je Prompt machen konnten. Vor allem die Vorteile der unterschiedlichen Temperatur kamen in diesem Umfang vermutlich nicht zum Vorschein. Ob die Prompts in deutsch oder englisch an das ChatGPT gestellt werden scheint keinen Einfluss zu haben. Offensichtlich versteht es beide Sprachen gut genug, um die relevanten Daten zu extrahieren und die Probleme tauchen danach bei der ABAP-Entwicklung auf.

Interessanterweise scheint GPT-4 in der ABAP-Entwicklung schlechter zu sein als GPT-3.5, dies liegt unserer Vermutung nach daran, dass GPT-4 auf die Entwicklung in beliebteren Programmiersprachen spezialisiert wurde und sich dessen Fähigkeiten in ABAP dadurch verschlechtern haben.

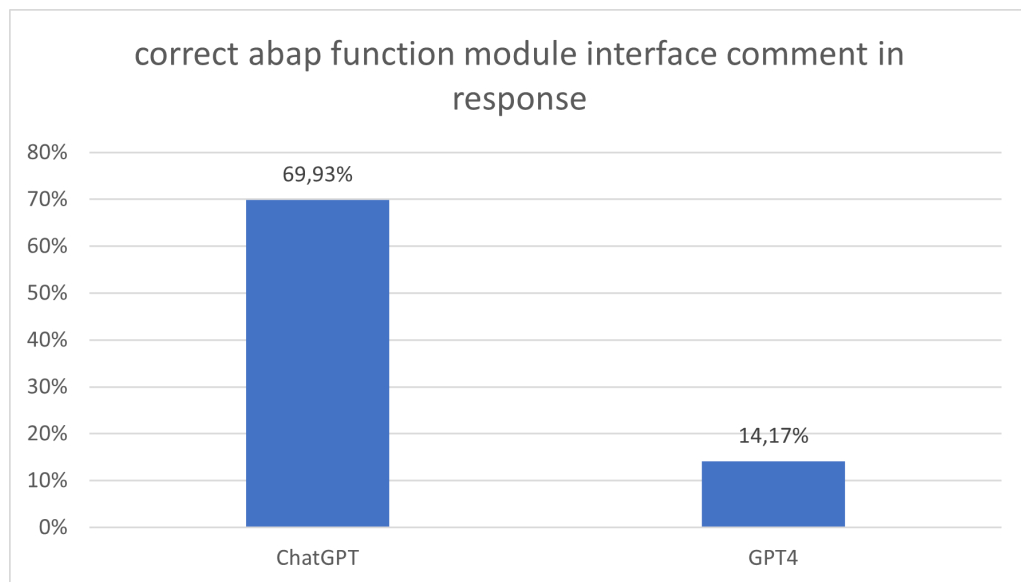


Abbildung 17: Anteil der Antworten die ein korrektes Interface-Kommentar im ABAP-Funktionsbaustein enthalten

3.2 Manuelle Tests

Die Auswertung der Tests erfolgt, um verschiedene Kennzahlen extrahieren zu können, welche wiederum dazu beitragen die Unterstützung von ChatGPT in verschiedenen Bereichen der Softwareentwicklung zu evaluieren.

Die Rohdaten und unsere Auswertung befindet sich im Github-Verzeichnis unter <https://github.com/timkoehne/Praxisprojekt-ABAP-und-ChatGPT/tree/main/results/manual%20tests/>.

Angefangen bei dem Benchmark, wo Syntaxfehler in die verschiedenen Tests eingebaut wurden, wo betrachtet wird, ob ChatGPT diese erkennen kann oder nicht.

Hier haben wir aufgrund der Anzahl von Kennzahlen zwei Diagramme erstellt, um die Ergebnisse übersichtlicher darzustellen.

Bei der Auswertung betrachten wir hier für jeden Test die aufaddierte Anzahl an spezifischen Punkten. Wenn das LLM etwas gemäß der einzelnen Punkte macht, wird dies mit einer eins befüllt, macht es dies nicht wird es mit einer null befüllt. Ein Beispiel hierfür wäre “liefert ChatGPT eine

korrekte Erklärung für den Fehler?“. Ist dies zutreffend, wird in dieses Feld eine eins geschrieben ist dies nicht zutreffend, wird eine null hineingeschrieben.
Nun können wir anhand der spezifischen Punkte für die einzelnen Tests ein Gesamtergebnis bilden, und dieses mit anderen Tests zusammen vergleichen.

3.2.1 Testergebnis: Syntaxfehler finden

Beginnend beim Fehlerfindungstest, mit den Daten zu korrekte Erklärung und falsche Erklärung, wie bereits erwähnt, zeigt das Ergebnis in diesen Feldern an, wie oft der spezielle Fehler korrekt und wie oft der Fehler falsch von dem LLM erklärt wurde.

Diese beiden Kennzahlen stehen direkt in Verknüpfung zueinander denn, wenn eine Erklärung richtig ist, somit eine eins hineingeschrieben wird, kann die Erklärung nicht falsch sein und eine 0 wird in dieses Feld geschrieben.

Sichtbar wird dies bei der Betrachtung von Abbildung 18.



Abbildung 18: Ergebnisse der Fehlererklärung des Fehlerfindungstests

Ebenfalls lässt sich anhand von Abbildung 18 erkennen, dass ChatGPT bei der Erklärung von den einzelnen Fehlern anscheinend nur selten korrekte Erklärungen gibt und die durchschnittliche Quote für eine korrekte Erklärung über alle Testdurchläufe hinweg bei 24% liegt. Die durchschnittliche Quote für eine falsche Erklärung ist hingegen 76%, woran man erkennt, dass sich das LLM in diesem Bereich wahrscheinlich nicht wirklich eignet.

Die weiteren Punkte, nach denen eine Auswertung erfolgt, sind behobene Fehler, hier wird betrachtet wie oft ChatGPT den eingebauten Syntaxfehler beheben konnte. Daneben werden neu hinzugefügte Syntax- und Semantikfehler betrachtet. Hier schauen wir einfach wie oft das LLM einen Fehler behoben, aber dafür einen neuen Fehler mit eingebaut hat. Möglich ist auch dass es einfach einen Fehler nicht behebt, und zusätzlich einen weiteren Fehler mit einbaut.

Anhand von Abbildung 19 kann man erkennen, dass ChatGPT sehr oft die eingebauten Syntaxfehler beheben kann, und dies durchschnittlich bei 64% der Durchläufe schafft, was für die Nutzung von LLMs in diesem Bereich spricht.

Zudem kann man erkennen, dass neu eingebaute Fehler ob Syntax- oder Semantikfehler relativ selten vorkommen. Es lässt sich aber in Hinblick auf die anderen Tests erkennen, dass gerade bei einem Testfall (Prompt 11) viele neue Fehler erzeugt wurden. Dies wird wahrscheinlich daran liegen, dass dieser Test komplexer in Bezug auf den zu testenden Bereich als die anderen war.

Vergleicht man die beiden Musterlösungen (Abbildung 24 und Abbildung 26) lässt sich relativ einfach feststellen, dass der Code von Prompt 157 kürzer und weniger komplex ist als der von Prompt 11. Aus dieser Feststellung resultierend, kann man davon ausgehen, dass das LLM in einem kurzen und simplen Code Fehler deutlich besser erkennen und beheben kann als bei einem langen und komplexen.

Ein langer und zu komplexer Code, wie der bei Prompt 11 scheint dem LLM hingegen Probleme

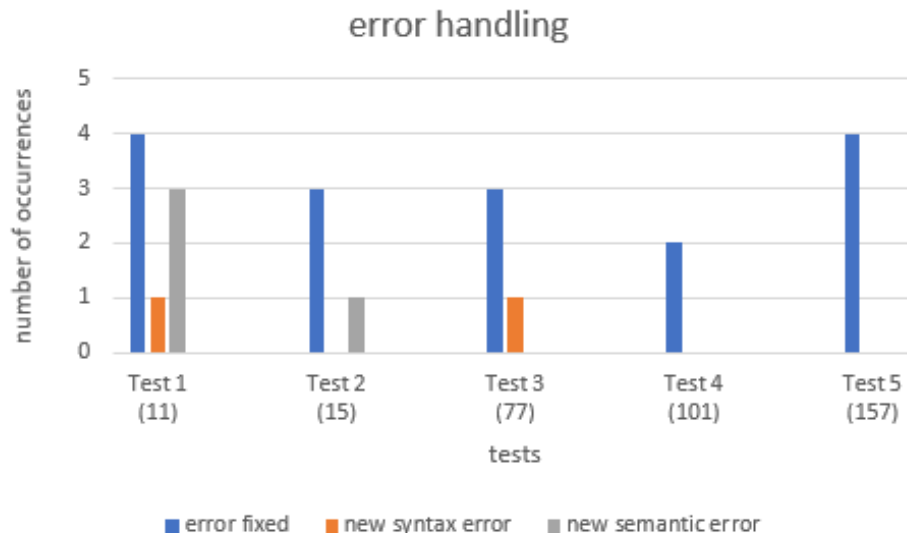


Abbildung 19: Ergebnisse der Fehlerbehebung des Fehlerfindungstests

zu machen, sodass einerseits zwar Fehler behoben, aber dafür neue Fehler eingebaut werden. Dies kann verschiedene Gründe haben wie zum Beispiel die Verwendung von IF-Else Verzweigungen in Schleifen oder auch die Verwendung von Befehlen wie Concatenate. Diese genannten Punkte tragen wie bereits erwähnt zur Komplexität des Codes und somit ggf. zu Schwierigkeiten für das LLM bei dem Umgang mit diesem bei.

Wichtig zu beachten ist hier jedoch, dass das LLM auch aus ganz anderen Gründen abseits von der Code Komplexität versagen kann. Ein Beispiel hierfür wäre, dass das LLM einen primitiven Fehler wie das Setzen von einem Index auf Eins und nicht auf Null bei jedem Durchlauf macht, und somit bei jedem Durchlauf ein neuer Fehler ausgegeben wird.

3.2.2 Testergebnis: Codeerklärung

Als zweites folgt die Auswertung des Tests, bei dem Code erklärt werden soll. Hier haben wir verschiedene erhobene Daten für jeden der fünf Testdurchläufe der spezifischen Tests, denen wir jeweils für die Bewertung eine unterschiedliche Wertigkeit zuweisen. Diese Werte werden dann wieder für jeden der fünf Tests aufaddiert.

Wir betrachten pro Durchlauf einmal die Erklärung also die Qualität der Erklärung an sich und ob diese Sinn macht sowie die Korrektheit vom Rückgabewert wo geguckt wird, ob die Rückgabe, welche durch ChatGPT erklärt und dargestellt wird korrekt ist.

Wir bewerten hierbei die Erklärung gut mit einer 1, den Rückgabewert korrekt mit einer 1, die Erklärung mit kleinen Fehlern 0.5 und größere Fehler in der Erklärung und im Rückgabewert als 0.

Wir können anhand von Abbildung 20 erkennen, dass wahrscheinlich eine Korrelation zwischen Erklärung und Rückgabewert besteht. Man kann zudem wie bei dem ersten Test erkennen, dass es wieder einen “Ausreißer” unter den Tests gegeben hat. Erklären lässt sich dies wahrscheinlich wieder damit, dass dieser spezifische Test einen Teil enthält, welcher von ChatGPT bei fast jedem Durchlauf falsch interpretiert worden ist. Wir können aber auch erkennen, dass die restlichen Tests bei jedem Durchlauf ohne Probleme erklärt und die korrekten Rückgabewerte erbracht haben.

Anhand der Ergebnisse lässt sich schlussfolgern, dass ChatGPT in diesem Bereich, also der Erklärung von Code geeignet für die Unterstützung ist.

Man kann als zusätzliche Absicherung neben der Erklärung zusätzlich auf den “Wert”, welcher durch Rückgabewert gegeben wird, gucken, da man anhand von ihm schnell erkennen kann, ob einerseits das Programm korrekt interpretiert und somit vermutlich auch die Erklärung korrekt ist. Ist der Rückgabewert einmal nicht zu einer Erklärung gegebenen kann man diesen von ChatGPT ohne weitere Probleme anfordern, da dieser sich auf die vorherige Erklärung beziehen wird (solange man sich in der gleichen Konversation befindet).

Insgesamt gesehen sind die Ergebnisse bei 80% der Tests einerseits bei der Erklärung und andererseits bei dem Rückgabewert korrekt.

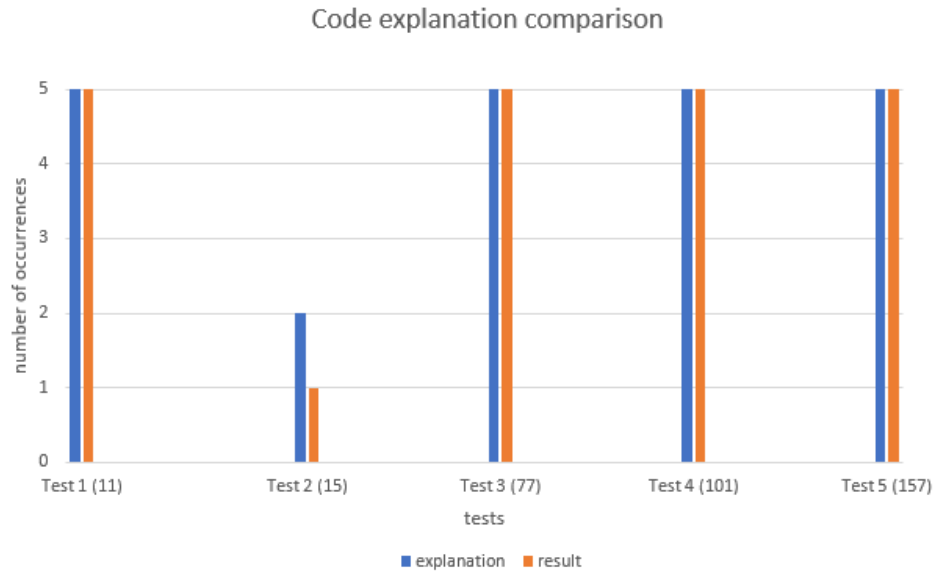


Abbildung 20: Ergebnisse des Codeerklärungstests

Zunächst lässt sich beim zweiten Test wieder davon ausgehen, dass er einfach Komplexer ist als der Rest und deshalb viele Fehlerhafte Erklärungen auftreten. Vergleicht man jedoch die Musterlösung vom zweiten Test (Abbildung 25) und somit von Prompt 15 mit der vom ersten Test (Abbildung 24) also Prompt 11 lassen sich verschiedene Dinge feststellen.

Einerseits wurde der zuvor als komplex beschriebene Code von Test Eins ohne Probleme bei jedem Durchlauf erklärt und des Weiteren ist der Code von Test Zwei sogar vergleichsweise weniger komplex als der von Test Eins. Daraus lässt sich ableiten, dass die Codekomplexität nicht immer ein Indikator für den Erfolg oder Misserfolg eines Tests ist, und dass der Code jeweils bei bestimmten Aufgabenstellungen für die LLM geeignet oder ungeeignet sein kann.

Wie bei dem vorherigen Testergebnis kann man vermutlich darauf schließen, dass das LLM jedes Mal an derselben Stelle scheitert und einen primitiven Fehler gemacht hat (Dieses Verhalten haben wir öfters bei den Testdurchläufen beobachtet). Dies könnte wie bereits erwähnt ein Fehler sein, wo z.B. das LLM einmal zu viel inkrementiert, was die Erklärung als falsch erscheinen lässt.

3.2.3 Testergebnis: Korrekte Codeerkennung

Der letzte der drei manuellen Tests befasst sich mit der Erkennung von korrektem Code. Hierbei soll gezeigt werden, ob das LLM korrekten Code auch wirklich als korrekt erkennt, oder ob es fälschlicherweise diesen als inkorrekt deklariert. Hierbei wurden wieder einmal für das zutreffen eine 1 in die Daten eingetragen und für das nicht zutreffen eine 0, um die Ergebnisse wieder aufaddieren, und somit als Gesamtbild auswerten zu können.

Hierbei werden die folgenden Daten betrachtet einmal “wurde ein Fehler gefunden?” und als zweites “wurde eine Empfehlung zur Code Verbesserung ausgegeben?”. Abbildung 21 stellt die kumulierten Ergebnisse welche aus den Rohdaten erstellt worden sind dar.

Anhand von Abbildung 21 lässt sich schnell erkennen, dass ChatGPT in allen Tests bei mindestens einem Durchlauf einen Fehler gefunden hat. Zudem erkennt man, dass er bei 60% aller Testdurchläufe Fehler in dem eigentlich korrekten Code vermutet. Allein aus diesem Grund kann man schon absehen dass sich ChatGPT zur Unterstützung in diesem Bereich, der Verifikation, ob Code korrekt ist, vermutlich nicht eignet.

Hinzu kommt, dass ChatGPT zudem in 80% der Testdurchläufe eine Verbesserung des Codes ausgibt. Diese Verbesserungen sind jedoch beim Großteil nicht von Relevanz und sie können sogar Fehler enthalten, welche das Programm negativ beeinflussen.

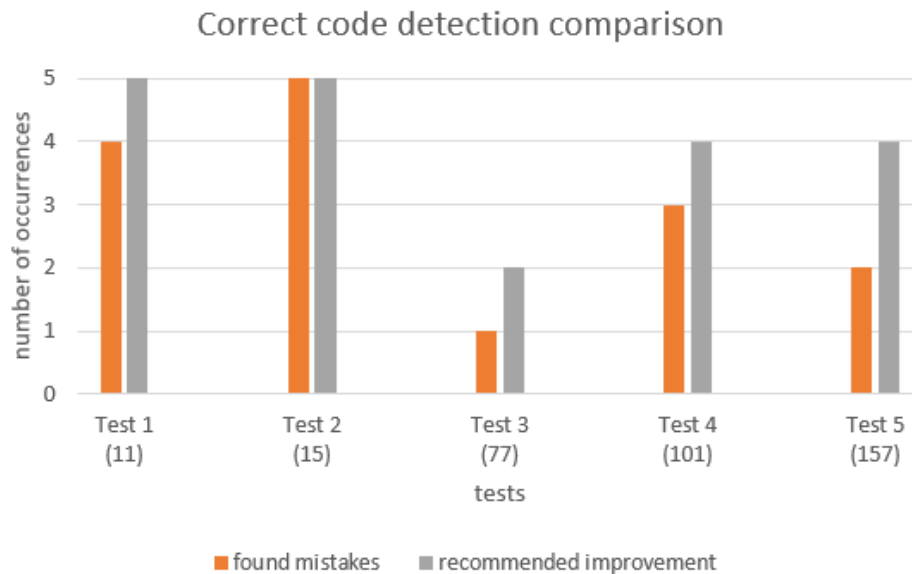


Abbildung 21: Ergebnis des korrekten Codeerkennungstests

3.2.4 Zusammenfassung: Manuelle Tests

Abschließend lässt sich zusammenfassen, dass durch die Auswertung der manuellen Tests festgestellt werden kann, dass sich ChatGPT in manchen Fällen wie beispielsweise bei den Codeerklärungen als Unterstützung eignen kann.

Andere Bereiche wie zum Beispiel die Erklärung von Fehlern im Code eignen sich nach den Auswertungen der manuellen Tests nicht wirklich, da die Erklärungen in vielen der Testfällen falsch sind.

Um eine endgültige Aussage über die Unterstützung von ChatGPT in verschiedenen Bereichen der Softwareentwicklung treffen zu können, sollte man einerseits den bestimmten Bereich, wo man die Unterstützung durch ein LLM beabsichtigt, intensiver testen, um auch sicher zu sein, dass sich das LLM auch wirklich eignen.

Denn wie bereits erwähnt, geben die manuellen Tests einen ersten Einblick in die Tauglichkeit von ChatGPT zur Unterstützung der verschiedenen Bereiche. Dennoch sind sie aber nicht vollends aussagekräftig, da die Anzahl an Tests und der Testdurchläufe und somit der gesamten Stichprobe zu gering ist um eine endgültige Aussage anhand dieser Treffen zu können.

3.3 Hypothese(n) zur Begründung der Ergebnisse

Unsere Ergebnisse zeigen eindeutig dass sich ChatGPT derzeit nicht als Hilfsmittel zur Codegenerierung eignet, jedoch vermutlich (aufgrund der viel kleineren Testmenge) sinnvoll als Lernhilfe und zur Erklärung von bestehendem Code genutzt werden kann. Genaue Begründungen für diese Ergebnisse können wir nicht wissen, wir wollen jedoch Vermutungen aufstellen, wie diese Ergebnisse zustande gekommen sind.

Für die Codegenerierung ist unserer Meinung nach der wichtigste Faktor die Menge der Trainingsdaten. ChatGPT soll recht gute Ergebnisse in der Codegenerierung in Python erreichen [Sid+23; Yet+23; Ope23; Liu+23]. Python ist jedoch auch eine sehr viel beliebtere Programmiersprache. Für den Suchbegriff “ABAP” zeigt Google aktuell 32.3 Millionen Suchergebnisse, im Vergleich dazu werden 1.37 Milliarden Suchergebnisse für “Python” angezeigt. Ähnliche Unterschiede erhalten wir von einer Google Trends Stichwortsuche: In Abbildung 22 ist zu erkennen, dass seit Beginn der Aufzeichnung in 2004 das Interesse an Python viel höher ist als an ABAP und der Unterschied ist seitdem stark gewachsen. In Abbildung 23 ist das durchschnittliche Interesse über den gesamten Zeitraum dargestellt. Außerdem hat die neuste Umfrage auf Stackoverflow zur Beliebtheit von Programmiersprachen in 2023 [Sta23] ergeben dass Python unter dem Top 3 der beliebtesten Programmiersprachen ist, ABAP hingegen hat es nicht einmal auf die Liste von 50 Programmiersprachen geschafft. IEEE Spectrum behauptet es gibt derzeit mehr als 123 mal mehr

Arbeitsplätze in Python als in ABAP [Cas23].

Diese Punkte zeigen zusammen ganz klar dass Python viel beliebter ist als ABAP. Die Menge an öffentlich verfügbaren Informationen wie Programmcode und Dokumentation sind ähnlich anzunehmen und spiegeln sich vermutlich in dem Trainingsdaten wieder, was zu den Unterschieden in der Codegenerierung von ChatGPT beiträgt. Da die Trainingsdaten von ChatGPT nicht öffentlich sind lässt sich diese Vermutung jedoch nicht beweisen.

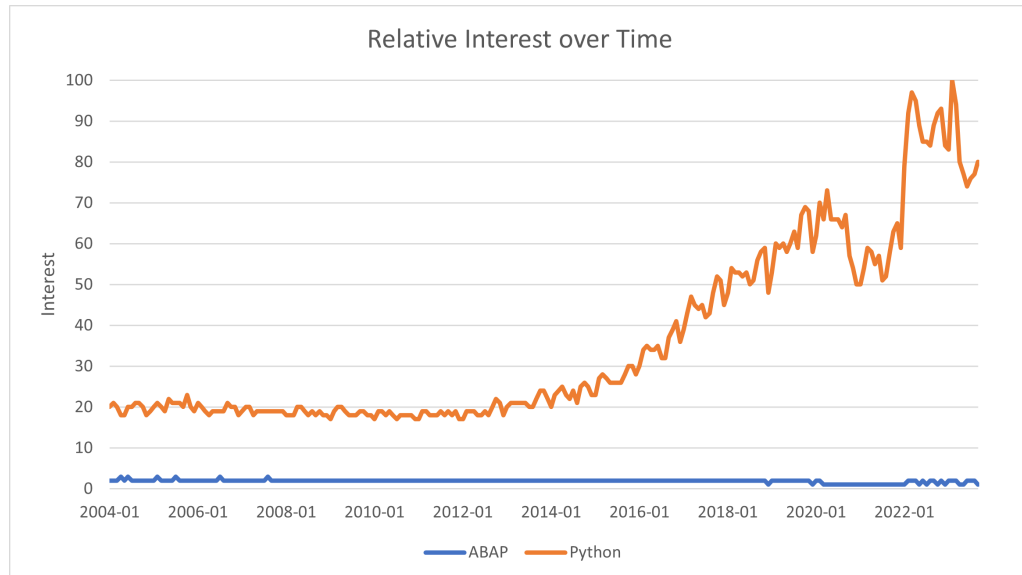


Abbildung 22: Vergleich des weltweiten Suchinteresse von Python und ABAP von 2004 bis September 2023 von Google Trends

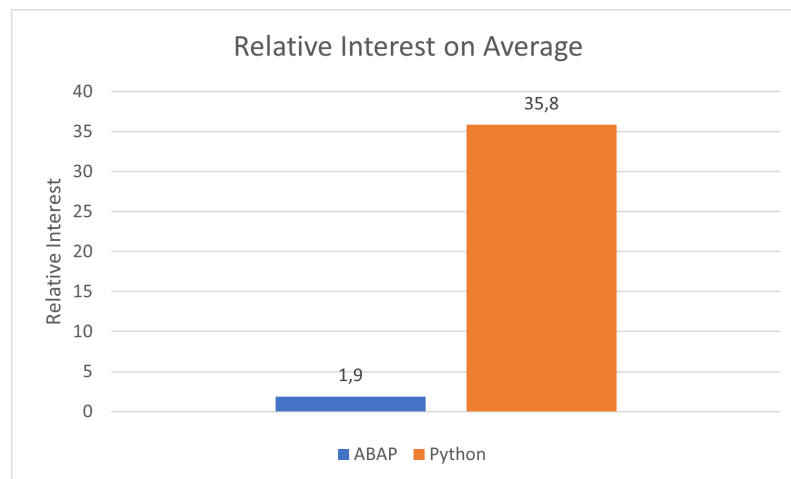


Abbildung 23: Vergleich des durchschnittlichen weltweiten Suchinteresse von Python und ABAP seit 2004 bis September 2023 von Google Trends

Darauf aufbauend haben wir anekdotisch festgestellt, dass ABAP Programmcode im Internet nur selten in dem für uns relevanten Format vorliegt: als vollständiger Funktionsbaustein mit dem Interface-Kommentar am Anfang. Der Großteil der Codeausschnitte ist als Bericht (Report) oder komplett ohne Kontext angegeben. Und wir nehmen an dass dies in den Trainingsdaten genauso der Fall ist. Dann vermuten wir dass der Mangel an Trainingsdaten zum Aufbau des Interface-Kommentars der Grund für viele der aufgetretenen Fehler ist. Die Definition der Import- und Exportparameter waren sehr oft ein Problem beim erstellen der Funktionsbausteine, aber auch für sonstige Syntax-Errors und unbekannte Variablen.

Ein Grund für das besonders schlechte Ergebnis von GPT-4 ist unserer Vermutung nach, dass die Programmierfähigkeiten von GPT-4 durch fine-tuning auf die beliebtesten Programmierspra-

chen verbessert wurde, dabei jedoch keine Rücksicht auf unbeliebtere Sprachen genommen wurde. Dies wäre ein logischer Schritt zur Verbesserung der Leistung für die größte Menge an Nutzern, und es kann offensichtlich keine Rücksicht auf jede Sprache genommen werden.

Bei einigen Prompts wurden von ChatGPT die Details der Aufgabenstellungen nicht verstanden. Der Großteil der Aufgabe wurde korrekt bearbeitet, aber kleine Details sorgen letztendlich für ein falsches Ergebnis. Ein gutes Beispiel hierfür ist unser manueller Test zur Code-Erklärung zu Prompt 15. Hier entstanden Fehler dadurch dass die Anweisung “inklusive” nicht beachtet wurde, und deshalb ein Schleifendurchlauf zu wenig ausgeführt wird. Auch wurde bei der String-Konkatenation nicht beachtet, dass hier Leerzeichen zwischen den Strings eingefügt werden sollen. Solche “einfachen” Fehler werden beim Ausführen schnell erkannt und von Menschen, oder auch durch ChatGPT bei erneuter Nachfrage, schnell behoben. Unsere Testmethodik erlaubt es uns jedoch nicht solche “einfachen” Fehler von Komplexeren automatisiert zu unterscheiden und dies im größeren Umfang zu analysieren.

Das gute Resultat unseres Codeerklärungstests kommt unserer Vermutung nach daher, dass zum Verständnis vom bestehenden Code nicht nur ABAP Trainingsdaten relevant sind, sondern auch die von anderen Programmiersprachen. Beispielsweise ist die Begründung um beim Datentyp zwischen Float und Integer zu unterscheiden in ABAP die gleiche wie in Java und C, es muss nur der gegebene Syntax richtig “verstanden” werden. Und da die ABAP Dokumentation vermutlich teil der Trainingsdaten ist wird gegebener Syntax größtenteils richtig verstanden. Außerdem kann bestehender Code oft einfach durch gutes Englisch-Verständnis verstanden werden, und darin scheint ChatGPT sehr gut zu sein. Beispielsweise ist eine Anweisung wie `“CONCATENATE 'a' 'b' 'c' INTO result SEPARATED BY space”` auch ohne ABAP Kenntnisse einfach zu verstehen, vor allem wenn Konzepte wie Variablen aus anderen Programmiersprachen bekannt sind.

Ein Aspekt den wir hier nicht untersuchen konnten ist die Änderung des Antwortverhaltens über verschiedene Versionen des gleichen LLMs. Im Internet gibt es zahlreiche Diskussionen darüber ob ChatGPT mit der Zeit schlechtere Ergebnisse liefert, in “How Is ChatGPT’s Behavior Changing over Time?” haben Forscher dieses Thema untersucht und kamen zu dem Schluss, dass die Ergebnisse zwischen Versionen sehr stark schwanken können [CZZ23].

4 Fazit und Ausblick

4.1 Fazit

Das Fazit unserer Arbeit umfasst viele verschiedene Erkenntnisse über die Unterstützung zur Programmentwicklung durch ChatGPT, welche wir durch unserer Tests erarbeitet haben. Die Haupte Erkenntnis ist hierbei, dass sich ChatGPT nicht für die Code-Generierung in ABAP eignet, da der erzeugte Code im Großteil der Ausgaben falsch war.

Ebenfalls zusammenhängend mit dieser Feststellung ist der Fakt, dass die Ausgabe der Codegenerierung trotz verschiedenster Anpassungen der Parameter nahezu keinen Einfluss auf die Code Ausgaben haben, diese nicht wirklich verbessern und sie somit falsch bleibt. Andere Quellen belegen, dass die Codegenerierung in Python mithilfe von ChatGPT jedoch gut funktioniert, was unsere Vermutung bestärkt, dass höchstwahrscheinlich einfach zu wenige Trainingsdaten zu ABAP-Code vorliegen. Diese Vermutung lässt sich aber leider nicht ohne weiteres bestätigen, da zu den Trainingsdaten keine spezifischen Informationen seitens der Entwickler vorliegen.

Es gibt jedoch auch viele gute Gründe warum ABAP in diesem Test schlechtere Ergebnisse liefert als Python.

Ebenfalls haben wir festgestellt, dass Codeerklärungen durch ChatGPT äußerst gut und in vielen der Tests korrekt waren und sich hier ChatGPT hier gut eignet, um beispielsweise Code zu verstehen. Zudem haben wir herausgefunden, dass auch wenn sich ChatGPT für Dinge wie die angesprochene Codeerklärung eignen würde, es dennoch Probleme im Umgang mit Datenschutz relevanten Daten, worunter oftmals auch Programmcode fällt, geben kann. Problem ist hier, dass diese Daten von OpenAI gespeichert und weiterverarbeitet werden können und somit gegebenenfalls Firmengeheimnisse in Umlauf gelangen.

Eine weitere Erkenntnis, die wir erlangt haben, ist, dass die Version GPT-4 welche eigentlich als besser gilt, in unserem Fall bei der Programmentwicklung in ABAP schlechter war als die ältere GPT-3.5 Version. Außerdem haben wir bei der Auswertung unserer Tests gemerkt, dass der Um-

fang der Tests, auch wenn er gerade bei den automatischen Tests schon groß war, noch deutlich größer sein muss, um mehr Parameterkombinationen zu testen. Unsere Temperatureinstellung, Systemprompt und Promptaufbau waren sicher nicht optimal und müssten mit weiteren groß angelegten Tests untersucht werden.

Letztendlich zeigt unsere Arbeit, dass sich ChatGPT in den meisten Fällen beim Thema der Programmentwicklung in ABAP als Unterstützung nicht eignet. Das Einzige was sich in diesem Bereich gegebenenfalls eignen würde, wenn man die genannten Datenschutzprobleme außer acht lässt, sind triviale Dinge wie die Codeerklärungen, welche man nutzen kann, wenn man zum Beispiel den Code von jemand anderem erweitern oder auch ändern soll.

4.2 Ausblick

Wir gehen trotz unserer Forschungsergebnisse davon aus, dass LLMs in der Zukunft mehr und mehr zur Unterstützung in der Softwareentwicklung genutzt werden. Sie haben ihre Nützlichkeit in beliebten Programmiersprachen bewiesen und werden einerseits von Endnutzern regelmäßig genutzt und Firmen investieren in öffentliche als auch in private LLMs und damit zusammenhängende Forschungsprojekte.

Auch in der ABAP-Entwicklung erwarten wir viel bessere Ergebnisse von LLMs in der Zukunft. Der Schritt von GPT-3 zu Codex [Che+21] hat gezeigt das durch gezielte Anpassungen starke Verbesserungen in der Programmierungsfähigkeit von LLMs erreicht werden können, und wir erwarten eine ähnliche Verbesserung beim Fokus auf ABAP und bei der Einbindung ins SAP-Ökosystem. SAP scheint die Fähigkeiten von Generativer KI verstanden zu haben und hat seit der Veröffentlichung von ChatGPT eine Roadmap zur Einbindung von Generativer KI [SAP23b] in das SAP Ökosystem veröffentlicht, wobei ihr Fokus eher auf Textgeneration, digitalen Assistenten und KPI-Vorschlägen liegt als auf Code-Generation.

Grade von großen Firmen erwarten wir dass sie sich von fremden LLMs distanzieren und stattdessen eigene Systeme aufbauen, die sie einerseits auf ihre Zwecke anpassen können, die aber auch dazu dienen ihre Daten sicher zu halten. Hierfür werden sie voraussichtlich auf Open-Source Modellen aufbauen und in diese investieren um die Weiterentwicklung zu garantieren, so wie es aktuell schon bei zahlreichen Open-Source Softwareprojekten der Fall ist.

5 Aufteilung des Projekts

- 1 Einleitung und Überblick
 - 1.1 Einleitung - Stephan & Tim
 - 1.2 Zielsetzung - Stephan
 - 1.3 Vorgehensweise - Stephan
 - 1.4 Begriffsklärung
 - 1.4.1 SAP - Stephan
 - 1.4.2 ABAP - Tim
 - 1.4.3 LLMs - Tim
 - 1.4.4 ChatGPT - Stephan
 - 1.5 Kategorisierung von Softwareentwicklungsaufgaben und Abschätzung ihrer Tauglichkeit zur Unterstützung durch LLMs - Tim
 - 1.5.1.1 Software Development Life Cycle Phasen: Planung - Tim
 - 1.5.1.2 Software Development Life Cycle Phasen: Anforderungsanalyse - Tim
 - 1.5.1.3 Software Development Life Cycle Phasen: Design / Entwurf - Stephan
 - 1.5.1.4 Software Development Life Cycle Phasen: Umsetzung - Tim
 - 1.5.1.4 Software Development Life Cycle Phasen: Test und Integration - Stephan
 - 1.5.1.4 Software Development Life Cycle Phasen: Wartung - Stephan
 - 1.5.2 Rollenbezug - Stephan
 - 1.6 Wissenschaftliche Literatur zur KI-Unterstützter Softwareentwicklung
 - 1.6.1 Programmieren lernen mit LLMs - Stephan
 - 1.6.2 LLM Benchmarking - Tim
 - 1.7 Datensicherheit und die Nutzung von LLMs - Tim
- 2 Vorbereitung des Benchmarks
 - 2.1 Wie sind LLM Software-Generierungsbenchmarks aufgebaut? - Tim
 - 2.2 Welche Anforderungen haben wir an unseren Benchmark?
 - 2.2.1 Welche Anforderungen entstehen durch ABAP? - Tim
 - 2.2.2 Welche Anforderungen entstehen durch ChatGPT? - Stephan
 - 2.3 Relevante Daten und ihre Messbarkeit - Stephan
 - 2.3.1 Parameter - Stephan
 - 2.3.2 Kennzahlen - Stephan
 - 2.4 Automatisierte Tests
 - 2.4.1 Versuchsaufbau - Tim
 - 2.4.2 Testvorbereitung - Tim
 - 2.4.3 Testplanung: Parameter finden - Tim
 - 2.4.4 Testplanung: Funktionsbausteine generieren - Tim
 - 2.4.5 Testplanung: Unterschiede deutsche und englische Prompts - Tim
 - 2.4.6 Testplanung: GPT-4 - Tim
 - 2.5 Manuelle Tests - Stephan
- 3 Durchführung des Benchmarkings
 - 3.1 Automatische Tests
 - 3.1.1 Testergebnis: Parameter finden - Tim
 - 3.1.2 Testergebnis: Funktionsbausteine generieren - Tim
 - 3.1.3 Testergebnis: Unterschiede deutsche und englische Prompts - Tim
 - 3.1.4 Testergebnis: GPT-4 - Tim
 - 3.1.5 Zusammenfassung: Automatische Tests - Tim
 - 3.2 Manuelle Tests
 - 3.2.1 Testergebnis: Syntaxfehler finden - Stephan
 - 3.2.2 Testergebnis: Codeerklärung - Stephan
 - 3.2.3 Testergebnis: Korrekte Codeerkennung - Stephan
 - 3.4.4 Zusammenfassung: Manuelle Tests - Stephan
 - 3.3 Hypothese(n) zur Begründung der Ergebnisse - Tim
- 4 Fazit und Ausblick
 - 4.1 Fazit - Stephan
 - 4.2 Ausblick - Tim

Literatur

- [23a] *ChatGPT — Release Notes — OpenAI Help Center*. 2023. URL: <https://help.openai.com/en/articles/6825453-chatgpt-release-notes> (besucht am 30. Okt. 2023).
- [23b] *ChatGPT: Get instant answers, find inspiration, learn something new*. 2023. URL: <https://chat.openai.com/> (besucht am 30. Okt. 2023).
- [23c] *Code Quality Tool & Secure Analysis with SonarQube*. 2023. URL: <https://www.sonarsource.com/products/sonarqube/> (besucht am 30. Okt. 2023).
- [23d] *Introducing ChatGPT*. 2023. URL: <https://openai.com/blog/chatgpt> (besucht am 30. Okt. 2023).
- [23e] *March 20 ChatGPT outage: Here's what happened*. 2023. URL: <https://openai.com/blog/march-20-chatgpt-outage> (besucht am 30. Okt. 2023).
- [23f] „Microsoft investiert 10 Milliarden in Open AI“. In: *Handelszeitung* (2023). URL: <https://www.handelszeitung.ch/tech/microsoft-investiert-10-milliarden-in-open-ai-566394> (besucht am 30. Okt. 2023).
- [23g] *OpenAI Platform*. 2023. URL: <https://platform.openai.com/docs/models> (besucht am 30. Okt. 2023).
- [23h] *OpenAI Platform*. 2023. URL: <https://platform.openai.com/docs/libraries/python-library> (besucht am 30. Okt. 2023).
- [23i] *Oracle VM VirtualBox*. 2023. URL: <https://www.virtualbox.org/> (besucht am 30. Okt. 2023).
- [23j] „Politik: ChatGPT-Rede im Landtag blieb unerkannt“. In: *ORF.at* (2023). URL: <https://steiermark.orf.at/stories/3194758/> (besucht am 30. Okt. 2023).
- [23k] *PyRFC - The Python RFC Connector — pyrfc 3.3 documentation*. 2023. URL: <https://sap.github.io/PyRFC/> (besucht am 30. Okt. 2023).
- [23l] *SAP NetWeaver Remote Function Call (RFC) Software Development Kit (SDK)*. 2023. URL: <https://support.sap.com/en/product/connectors/nwrfcsdk.html> (besucht am 30. Okt. 2023).
- [23m] *Terms of use*. 2023. URL: <https://openai.com/policies/terms-of-use> (besucht am 30. Okt. 2023).
- [23n] *The Potentially Large Effects of Artificial Intelligence on Economic Growth (Briggs/Kodnani)*. 2023. URL: <https://www.gspublishing.com/content/research/en/reports/2023/03/27/d64e052b-0f6e-45d7-967b-d7be35fabd16.html> (besucht am 30. Okt. 2023).
- [Aus+21] Jacob Austin u. a. *Program Synthesis with Large Language Models*. 16. Aug. 2021. URL: <http://arxiv.org/pdf/2108.07732v1>.
- [Bec+23] Brett A. Becker u. a. „Programming Is Hard - Or at Least It Used to Be“. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. Hrsg. von Maureen Doyle u. a. New York, NY, USA: ACM, 2023, S. 500–506. ISBN: 9781450394314. DOI: [10.1145/3545945.3569759](https://doi.org/10.1145/3545945.3569759).
- [Bow23] Samuel R. Bowman. *Eight Things to Know about Large Language Models*. 2. Apr. 2023. URL: <http://arxiv.org/pdf/2304.00612v1>.
- [Cas23] Stephen Cass. „The Top Programming Languages 2023“. In: *IEEE Spectrum* (2023). URL: <https://spectrum.ieee.org/the-top-programming-languages-2023> (besucht am 30. Okt. 2023).
- [Che+21] Mark Chen u. a. *Evaluating Large Language Models Trained on Code*. 7. Juli 2021. URL: <http://arxiv.org/pdf/2107.03374v2>.
- [Col23] Cameron Coles. *11% of data employees paste into ChatGPT is confidential - Cyberhaven*. 2023. URL: <https://www.cyberhaven.com/blog/4-2-of-workers-have-pasted-company-data-into-chatgpt/> (besucht am 30. Okt. 2023).
- [Cyn23] Cynthia Sanchez: front-end and UI, Zvezdana Marjanovic: graphic design. *OpenSuse*. 2023. URL: <https://www.opensuse.org/> (besucht am 30. Okt. 2023).

- [CZZ23] Lingjiao Chen, Matei Zaharia und James Zou. *How is ChatGPT's behavior changing over time?* 18. Juli 2023. URL: <http://arxiv.org/pdf/2307.09009v2>.
- [DJ23] Dan Jurafsky und James H. Martin. *Speech and Language Processing*. 2023. URL: <https://web.stanford.edu/~jurafsky/slp3/> (besucht am 30. Okt. 2023).
- [Eck23] Svea Eckert. „ChatGPT - wichtige Fragen und Antworten zur KI-App“. In: *NDR* (2023). URL: <https://www.ndr.de/ratgeber/ChatGPT-wichtige-Fragen-Antworten-zur-KI-App,chatgpt138.html> (besucht am 30. Okt. 2023).
- [Fin23] Finanzen100. *Top100: Deutsche Unternehmen an der Börse - Finanzen100*. 2023. URL: <https://www.finanzen100.de/top100/die-grossten-borsennotierten-unternehmen-deutschlands/> (besucht am 30. Okt. 2023).
- [Git23a] GitHub. *GitHub Copilot · Your AI pair programmer*. 2023. URL: <https://github.com/features/copilot> (besucht am 30. Okt. 2023).
- [Git23b] GitHub. *gpt-3/dataset_statistics/languages_by_word_count.csv at master · openai/gpt-3*. 2023. URL: https://github.com/openai/gpt-3/blob/master/dataset_statistics/languages_by_word_count.csv (besucht am 30. Okt. 2023).
- [Gur23] Mark Gurman. „Samsung Bans ChatGPT, Google Bard, Other Generative AI Use by Staff After Leak“. In: *Bloomberg* (2023). URL: <https://www.bloomberg.com/news/articles/2023-05-02/samsung-bans-chatgpt-and-other-generative-ai-use-by-staff-after-leak#xj4y7vzkg> (besucht am 30. Okt. 2023).
- [Hug00] Alex Hughes. „ChatGPT: Everything you need to know about OpenAI's GPT-4 tool“. In: (1695658398000). URL: <https://www.sciencefocus.com/future-technology/gpt-3> (besucht am 30. Okt. 2023).
- [Kap+20] Jared Kaplan u. a. *Scaling Laws for Neural Language Models*. 23. Jan. 2020. URL: <http://arxiv.org/pdf/2001.08361v1>.
- [Kim23] Eugene Kim. „Amazon warns staff not to share confidential information with ChatGPT“. In: *Insider* (2023). URL: <https://www.businessinsider.com/amazon-chatgpt-openai-warns-employees-not-share-confidential-information-microsoft-2023-1> (besucht am 30. Okt. 2023).
- [Liu+23] Jiawei Liu u. a. *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*. 2. Mai 2023. URL: <http://arxiv.org/pdf/2305.01210v2>.
- [MSL23] Gabriela Mello, William Shaw und Hannah Levitt. „Wall Street Banks Are Cracking Down on AI-Powered ChatGPT“. In: *Bloomberg* (2023). URL: <https://www.bloomberg.com/news/articles/2023-02-24/citigroup-goldman-sachs-join-chatgpt-crackdown-fn-reports> (besucht am 30. Okt. 2023).
- [Ope23] OpenAI. *GPT-4 Technical Report*. 15. März 2023. URL: <http://arxiv.org/pdf/2303.08774v3>.
- [Pap+01] Kishore Papineni u. a. „BLEU“. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*. Hrsg. von Pierre Isabelle. Morristown, NJ, USA: Association for Computational Linguistics, 2001, S. 311. DOI: [10.3115/1073083.1073135](https://doi.org/10.3115/1073083.1073135).
- [Reb] Rebecca Swarupa. „SDLC“. In: (). URL: <https://www.academia.edu/6328712/SDLC>.
- [Rei+20] Ricardo Rei u. a. „COMET: A Neural Framework for MT Evaluation“. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Hrsg. von Bonnie Webber u. a. Stroudsburg, PA, USA: Association for Computational Linguistics, 2020, S. 2685–2702. DOI: [10.18653/v1/2020.emnlp-main.213](https://doi.org/10.18653/v1/2020.emnlp-main.213).
- [RTK22] Tatyana Rozhnova, Valeriia Tomachynska und Denis Korsun. „Life cycle models, principles and methodologies of software development“. In: *InterConf* 28(137) (2022), S. 394–401. DOI: [10.51582/interconf.19-20.12.2022.040](https://doi.org/10.51582/interconf.19-20.12.2022.040).
- [SAP23a] SAP. *Developer Trials and Downloads — SAP Developer*. 2023. URL: <https://developers.sap.com/trials-downloads.html> (besucht am 30. Okt. 2023).
- [SAP23b] SAP. *Generative AI with SAP*. 2023. URL: <https://www.sap.com/products/artificial-intelligence/generative-ai.html> (besucht am 30. Okt. 2023).

- [SAP23c] SAP. *Installing AS ABAP 7.52 dev edition on Virtual Box and Linux*. 2023. URL: <https://www.sap.com/documents/2019/09/32638f18-687d-0010-87a3-c30de2ffd8ff.html> (besucht am 30. Okt. 2023).
- [SAP23d] SAP. *SAP S/4HANA Cloud, Public Edition*. 2023. URL: <https://www.sap.com/germany/products/erp/s4hana.html> (besucht am 30. Okt. 2023).
- [SAP23e] SAP. *Unternehmensinformationen — Informationen zur SAP SE*. 2023. URL: <https://www.sap.com/germany/about/company.html> (besucht am 30. Okt. 2023).
- [SAP23f] SAP. *Was ist ERP? — Definition von Enterprise Resource Planning — SAP*. 2023. URL: <https://www.sap.com/germany/products/erp/what-is-erp.html> (besucht am 30. Okt. 2023).
- [SAP23g] SAP. *Was ist SAP? — Definition & Bedeutung — SAP Abkürzung*. 2023. URL: <https://www.sap.com/germany/about/what-is-sap.html> (besucht am 30. Okt. 2023).
- [Sid+23] Mohammed Latif Siddiq u. a. *Exploring the Effectiveness of Large Language Models in Generating Unit Tests*. 30. Okt. 2023. URL: <http://arxiv.org/pdf/2305.00418v1>.
- [Siu23] Diamond Naga Siu. „Microsoft tells employees not to share ‘sensitive data’ with ChatGPT“. In: *Insider* (2023). URL: <https://www.businessinsider.com/microsoft-tells-employees-not-to-share-sensitive-data-with-chatgpt-2023-1?op=1> (besucht am 30. Okt. 2023).
- [Sta23] Stack Overflow. *Stack Overflow Developer Survey 2023*. 2023. URL: <https://survey.stackoverflow.co/2023/#learning-to-code-learn-code> (besucht am 30. Okt. 2023).
- [Tli+23] Ahmed Tlili u. a. „What if the devil is my guardian angel: ChatGPT as a case study of using chatbots in education“. In: *Smart Learning Environments* 10.1 (2023). DOI: [10.1186/s40561-023-00237-x](https://doi.org/10.1186/s40561-023-00237-x).
- [Vas+17] Ashish Vaswani u. a. *Attention Is All You Need*. 12. Juni 2017. URL: <http://arxiv.org/pdf/1706.03762v7>.
- [Wan+18] Alex Wang u. a. *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*. 20. Apr. 2018. URL: <http://arxiv.org/pdf/1804.07461v3>.
- [Wan+19] Alex Wang u. a. *SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems*. 2. Mai 2019. URL: <http://arxiv.org/pdf/1905.00537v3>.
- [Wei+22] Jason Wei u. a. *Emergent Abilities of Large Language Models*. 15. Juni 2022. URL: <http://arxiv.org/pdf/2206.07682v2>.
- [Yet+23] Burak Yetiştiren u. a. *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. 21. Apr. 2023. URL: <http://arxiv.org/pdf/2304.10778v2>.

6 Anhang

```
FUNCTION Z_STRING_XOR.
  """-----
  """Local Interface:
  """ IMPORTING
  """   VALUE(IV_A) TYPE  STRING
  """   VALUE(IV_B) TYPE  STRING
  """ EXPORTING
  """   VALUE(RV_RESULT) TYPE  STRING
  """-----

  DATA: lv_result TYPE string.

  DATA: lv_length_a TYPE i,
        lv_length_b TYPE i,
        lv_index type i.

  lv_length_a = strlen( iv_a ).
  lv_length_b = strlen( iv_b ).

  IF lv_length_a <> lv_length_b.
    MESSAGE 'Both input strings must have the same length.' TYPE 'E'.
    RETURN.
  ENDIF.

  lv_index = 0.

  DO lv_length_a TIMES.
    if iv_a+lv_index(1) = iv_b+lv_index(1).
      CONCATENATE RV_RESULT '0' INTO RV_RESULT.
    ELSE.
      CONCATENATE RV_RESULT '1' INTO RV_RESULT.
    ENDIF.
    lv_index = lv_index + 1.
  ENDDO.

ENDFUNCTION.
```

Abbildung 24: Musterlösung zu Prompt 11 der Manuellen Tests

```
FUNCTION z_string_sequence.
  """-----
  """Local Interface:
  """ IMPORTING
  """   VALUE(N) TYPE  I
  """ EXPORTING
  """   VALUE(RETURN_VALUE) TYPE  STRING
  """-----

  data lv_str type string.

  DO n + 1 TIMES.
    lv_str = sy-index - 1.
    CONCATENATE return_value lv_str INTO return_value.
  ENDDO.

ENDFUNCTION.
```

Abbildung 25: Musterlösung zu Prompt 15 der Manuellen Tests

```

FUNCTION z_right_angle_triangle.
*"-----
***Local Interface:
*" IMPORTING
*"     VALUE(A) TYPE I
*"     VALUE(B) TYPE I
*"     VALUE(C) TYPE I
*" EXPORTING
*"     VALUE(RETURN_VALUE) TYPE C
*"-----

DATA: square_a TYPE i,
      square_b TYPE i,
      square_c TYPE i.

square_a = a * a.
square_b = b * b.
square_c = c * c.

return_value = ' '.
if square_a = square_b + square_c or
   square_b = square_a + square_c or square_c = square_a + square_b.
   return_value = 'X'.
endif.

ENDFUNCTION.

```

Abbildung 26: Musterlösung zu Prompt 157 der Manuellen Tests