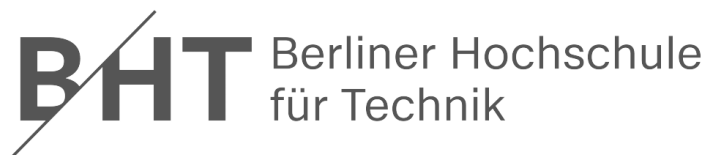


Evaluierung und Optimierung von Large Language Models für die Entwicklung von Webanwendungen

Ein Ansatz zur Verbesserung des Entwicklungsprozesses bei Softwareprojekten



Masterthesis

für den angestrebten akademischen Grad
Master of Science im Studiengang Medieninformatik

Eingereicht von: Wilfried Pahl
Matrikelnummer: 901932
Studiengang: Online Medieninformatik
Berliner Hochschule für Technik

Betreuer Prof. Dr. S. Edlich
Berliner Hochschule für Technik
Gutachter Prof. Dr. Alexander Löser
Berliner Hochschule für Technik

Temmen-Ringenwalde, der 2. März 2025

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel „Evaluierung und Optimierung von Large Language Models für die Entwicklung von Webanwendungen (*Ein Ansatz zur Verbesserung des Entwicklungsprozesses bei Softwareprojekten*)“ selbstständig und ohne unerlaubte Hilfe verfasst habe. Alle benutzten Quellen und Hilfsmittel sind vollständig angegeben und wurden entsprechend den wissenschaftlichen Standards zitiert.

Ich versichere, dass alle Passagen, die nicht von mir stammen, als Zitate gekennzeichnet wurden und dass alle Informationen, die ich aus fremden Quellen übernommen habe, eindeutig als solche kenntlich gemacht wurden. Insbesondere wurden alle Texte und Textpassagen anderer Autoren sowie die Ergebnisse von Sprachmodellen wie OpenAI's GPT-3 entsprechend den wissenschaftlichen Standards zitiert und referenziert.

Ich versichere weiterhin, dass ich keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und dass ich keine Teile dieser Arbeit in anderer Form für Prüfungszwecke vorgelegt habe.

Mir ist bewusst, dass eine falsche eidesstattliche Erklärung strafrechtliche Konsequenzen haben kann.

Temmen-Ringenwalde, 2. März 2025
Ort, Datum



Unterschrift

ABSTRACT

In erster Linie besteht die Zielsetzung dieser Thesis darin, die Fähigkeiten der generativen KI zu prüfen und verschiedenen Modelle zu vergleichen, inwieweit sich diese für die Webanwendungsentwicklung für Unternehmen und Entwicklern aus diesem Bereich, eignen. In dem Vergleich werden *Cloused-Source-Modelle* von kommerziellen Anbietern mit *Open-Source-Modellen* verglichen. Erstaunlicherweise schneiden die kommerziellen Modelle bei dem HumanEval-XL Benchmark nicht so überzeugend ab wie zuvor erwartet. Getestet werden die Programmiersprache PHP mit deutschsprachigen Proben aus dem genannten Benchmark. Trotz des angewendeten Benchmarks eignen sich diese nur bedingt, um den vollen Umfang von LLMs zu evaluieren. Dieses Problem entsteht da die generative KI nicht wie ein deterministisches System agiert, sondern es wird eine Wahrscheinlichkeit berechnet, welches die beste Lösung sein könnte. Somit zeigen die LLMs ein sehr dynamisches Verhalten, was mit statischen Benchmarks schwer abzudecken ist. Dennoch liefert der Benchmarks erste Ergebnisse und Modelle können verglichen werden. Nachdem die Evaluierung durchgeführt abgeschlossen ist, werden verschiedene Ansätze zur Optimierung der Eingabeaufforderung umgesetzt und ebenfalls evaluiert. Zum einen zeigt das Framework DSPy bei einigen Modellen eine signifikante Verbesserung der Ergebnisse, die so nicht erwartet wird. Zum Anderen gibt es Modelle, bei denen eine Verschlechterung der Ergebnisse eintritt und Modelle mit einer großen Anzahl an Parametern können aufgrund von Fehler nicht evaluiert werden.

Inhaltsverzeichnis

Abstract	i
Abbildungsverzeichnis	v
Tabellenverzeichnis	vi
Listings	vii
Abkürzungsverzeichnis	viii
1 Einleitung	1
1.1 Hintergrund und Kontext	1
1.2 Problemstellung	2
1.3 Stand der Forschung	2
1.4 Zielsetzung und Thesen	3
1.5 Aufbau der Arbeit	4
1.6 Abgrenzung	5
2 Grundlagen	7
2.1 Künstliche Intelligenz	8
2.1.1 Maschinelles Lernen	8
2.1.2 Neuronale Netze	9
2.1.3 Deep Learning	11
2.2 Natural Language Processing	12
2.3 Large Language Model	12
2.3.1 Grundlagen	12
2.3.2 Grenzen und Probleme bei LLMs	13
2.3.3 Verständnis für die LLMs	14
2.4 Prompt Engineering	15
2.4.1 Prompt-Techniken	15
2.4.2 Grenzen beim Prompt-Engineering für LLMs	16
2.4.3 Promptoptimierung mit DSPy	16
2.5 Grundlagen der Webentwicklung	18
2.5.1 Programmiersprachen	18
2.5.2 Entwicklung	18
2.6 Benchmark für LLM	19

2.6.1	pass@k Methode	20
3	Konzeption / Design	21
3.1	Definition der Evaluierungsziele	21
3.2	Auswahl der LLMs und deren Konfiguration	22
3.3	Design der Evaluierung	25
3.3.1	HumanEval-XL Benchmark	25
3.3.2	Durchführung der Evaluierung	28
3.3.3	Dokumentation der Daten	29
3.4	Konzeption der Optimierung	29
3.4.1	Optimierung durch Frameworkauswahl	31
3.4.2	Evaluierung der Optimierungen	31
3.5	Testumgebung	31
4	Implementierung	33
4.1	Lokale Modelle	33
4.1.1	Bereitstellen der Modelle	33
4.1.2	Ergebnisse generieren	34
4.2	Optimierung der Antworten durch Änderung des Frameworks	35
4.2.1	Ergebnisse generieren	37
4.3	Benchmark Codeevaluation	37
5	Evaluation	41
5.1	Modellbewertung	41
5.1.1	Evaluierung der Parametergröße	44
5.1.2	Evaluierung nach Sprache	46
5.2	Optimierung der Ergebnisse durch die Wahl des Frameworks	48
6	Lessons Learned	51
6.1	Evaluierungsaufbau und Vorbereitung	51
6.2	Evaluierung der großen Sprachmodelle	52
6.2.1	Lokale Ressourcen	52
6.2.2	Auswertung des Benchmarks	52
6.2.3	Nachteile der Evaluierung	53
6.3	Optimierung der Abfragen	57
7	Diskussion und Ausblick	61
7.1	Bewertung der Zielsetzung und Thesen	61
7.2	Grenzen und Einschränkungen	63
7.3	Impulse für zukünftige Forschungen	63
7.4	Praktische Anwendung	64
8	Fazit	67
	Literatur	69

Anhang	73
A Installationshinweise	73
A.1 Python	73
A.2 Installation und Konfiguration von Ollama	73
A.3 Open WebUI Installationshinweise	74
B Download der Hugging Face Modelle	75
C Abfragen lokaler Modelle	77
D Abfrage cloudbasierter Modelle	82
E Evaluation der Antworten von den Modellen	88
F Methodenerkennung unter PHP	94

Abbildungsverzeichnis

2.1	LLMs im Kontext der Forschungsbereiche von KI	7
2.2	Biologische Nervenzelle	10
2.3	Künstliche Nervenzelle	10
3.1	Aufbau des HumanEval-XL Benchmarks	26
3.2	Ablauf der Evaluierung mit dem HumanEval-XL Benchmarks	29
5.1	Ergebnisse der pass@k-Methode für die Modelle	43
5.2	Ergebnisse der pass@k-Methode für die Modelle	45
5.3	Ergebnisse der pass@k-Methode für Llama Modelle in gewählten Sprachen . . .	47
5.4	Ergebnisse der pass@k-Methode für unterschiedliche Frameworks	49
6.1	Auslastung des VRAMs während eines Modellwechsels	58
7.1	Anwendungsbeispiel für die Integration einer LLM	66
7.2	Anwendungsbeispiel für ein Eingabeformular für Prompts	66

Tabellenverzeichnis

3.1	Auswahl der LLMs für die Evaluierung	24
3.2	Einstellungen der Modellparameter	25
5.1	Ergebnisse der pass@1 und pass@5 Methode	42
5.2	Ergebnisse der pass@1 Methode mit Angabe der Parameter	46
5.3	Vergleich der englischen und deutschen Sprache für den pass@1	48
5.4	Ergebnisse der pass@1 und pass@5 Methode verschiedener Frameworks	49
6.1	Ergebnisse für das <i>Deepseek Coder V2</i> Modell mit und ohne Probe 5 Korrektur	55

Listings

3.1	PHP Beispiele für die erste Probe	27
3.2	PHP Beispiele für den Test der ersten Probe	28
3.3	PHP Beispiele für den Test der dritten Probe, One-Shot-Prompt	30
4.1	Interaktion in Python mit Ollamaserver	34
4.2	Interaktion in Python mit Ollamaserver	34
4.3	Konfiguration eines Models in DSPy	35
4.4	Interaktion in Python mit Ollama-Server	37
4.5	Codesnippet zur Extrahierung des Codes aus der LLM Antwort	37
4.6	Codesnippet zur Ausführung des PHP Interpreters	38
4.7	Berechnung der pass@k Metrik in Python	39
5.1	Wortlaut der Aufgabe 23 aus dem HumalEval-XL Benchmark	42
6.1	Aufgabenstellung der Probe php/5	54
6.2	Aufgabenstellung der Probe php/5	54
6.3	Übersetzte Aufgabenstellung der Probe php/5	55
6.4	Ergebnisse der Probe php/5 für das <i>Deepseek Coder V2</i> Modell	55
6.5	Fehler bei der Auswertung durch fehlerhafte Anführungszeichen	56
6.6	Generierte Funktion mit falschen Namen	56
8.1	Ollama Hostanpassng für Netzwerkbetrieb	73
8.2	Open WebUI installieren	74
8.3	Laden der Modelle von Hugging Face und lokal speichern	75
8.4	Abfragen der Ollama Modelle mit langchain	77
8.5	Abfragen der Ollama Modelle mit DSPy	79
8.6	Ausführen der Prompts für Gemini Modelle.	82
8.7	Ausführen der Prompts für OpenAI Modelle.	85
8.8	Evaluation der Modellantworten: Suche nach Codeausschnitten	88
8.9	PHP Skript welches verschiedene Funktionen implementiert	94
8.10	PHP Skript das Methoden zur Laufzeit erkennt	94
8.11	Erkannte PHP Methoden	95

ABKÜRZUNGSVERZEICHNIS

DL	Deep Learning.
IDE	Integrated Development Environment.
KNN	Künstliche Neuronale Netze.
LLM	Large Language Model.
ML	Maschine Learning.
NL2Code	Natural Language to Code.
NLP	Natural Language Processing.
RAG	Retrieval Augmented Generation.
VRAM	Video Random Access Memory.

1.1 Hintergrund und Kontext

Durch die zunehmende Globalisierung und Digitalisierung wird die Gesellschaft in der Gegenwart und Zukunft geprägt. Der Ausbau von Hochgeschwindigkeitsnetze und die globale Corona-Pandemie haben diese Entwicklung beschleunigt. Immer mehr Unternehmen erkennen die Potenziale der Digitalisierung und passen ihre Geschäftsprozesse an und nutzen die Möglichkeiten der digitalen Systeme. Ganze Wertschöpfungsketten werden auf cloudbasierte Umgebungen umgestellt. Angefangen bei der Kommunikation, über Beschaffung und Produktion bis zum Verkauf der Waren und Dienstleistungen, vergleiche mit [1, Seite 21 ff.] und [2]. In allen Stufen der Prozesse kommen webbasierte Anwendungen zum Einsatz, um die Kommunikation der Anwender mit den Systemen zu ermöglichen oder Schnittstellen für die Datenübertragung zwischen den verschiedenen Systemen zu gewährleisten. Durch die wachsende Anzahl von Web-Anwendungen wächst auch der Druck für die Entwicklungsfirmen, Anwendungen den oft schnell und wechselnden Kundenanforderungen anzupassen.

Durch diesen Prozess getrieben, müssen Entwicklungsfirmen in immer kürzeren Release-Zyklen Softwarekomponenten hinzufügen oder vorhandene erweitern. Gleichzeitig wachsen aber auch die Anforderungen an Stabilität und Sicherheit der cloudbasierten Anwendungen, sowie der Bedarf an kostengünstigeren IT-Abläufen. Ein weiteres Problem ist der wachsende Fachkräftemangel in der Wirtschaft und die damit verbundenen steigenden Gehälter der Entwickler.

Die Verwendung künstlicher Intelligenz bei der Programmierung gewinnt immer mehr an Bedeutung. Eine Technologie die im besonderen Maße an dieser Entwicklung beteiligt ist, sind die Large Language Models. Insbesondere mit der Veröffentlichung vom ChatGPT wurde hier ein regelrechter Hype um die LLMs ausgelöst. Diese Modelle erlauben eine Softwareentwicklung mit natürlicher Sprache. Dadurch sind viele Nutzer in der Lage Programmcode zu erzeugen und diesen in vorhandene Software zu integrieren. Oft haben die Nutzer aber keine oder wenig Erfahrungen in der Softwareentwicklung und die damit verbundenen Kenntnisse der

Programmierung.

1.2 Problemstellung

So groß der Hype um künstliche Intelligenz auch sein mag, zurzeit kann KI nicht alle Anforderungen selbstständig lösen. Dies sollte auch bei der Verwendung von KI generierten Inhalten und Programmcodes beachtet werden.

KI denkt nicht, KI trifft keine Entscheidungen. Eine KI antwortet auf eine Eingabe nicht mit der besten Antwort, sondern mit der Wahrscheinlichsten.

VATTENFALL ONLINE , KI für Unternehmen – Die Grenzen der KI

Der Nutzer muss die generierten Ergebnisse überprüfen, ehe erstellte Programmcodestücke in vorhandene Programme eingefügt und in Produktionsumgebungen implementiert werden. Den im Gegensatz zur natürlichen Sprache, ist bei Problemen der Codegenerierung, die Syntax der jeweiligen Programmiersprache einzuhalten. Andernfalls kann es zu Laufzeitfehlern kommen oder einem unerwarteten Verhalten der Software führen.

Viele Entwickler setzen auf Chatbots, wie ChatGPT oder Gemini zur Generierung von Code, wie eine Umfrage von *stackoverflow* vom Mai 2024 zeigt [3]. Wenn der generierte Code ohne Prüfung und Tests in bestehende Projekte implementiert wird, kann dies dazu führen, dass sich unter anderem technische Schulden anhäufen. Dadurch erhöhen sich langfristig die Wartungsaufwände und das Hinzufügen von Erweiterungen ist ebenfalls mit erhöhtem Aufwand verbunden.

Ein weiteres Problem der generierten Codes sind die vorhandenen Sicherheitslücken. Werden diese fehlerhaften Codes übernommen, ist es oft ein leichtes für Angreifer sensible Kundendaten zu stehlen. In der Arbeit [4] wird das Thema Schwachstellen in von ChatGPT generiertem PHP-Code evaluiert und ebenfalls vor der sorglosen Verwendung gewarnt. Inwieweit die erstellten Codes Auswirkungen auf echte Webseiten haben, ist zurzeit noch nicht hinreichend untersucht.

Viele Entwickler und Nichtentwickler sind sich dieser mangelhaften generierten Code nicht bewusst und verwenden diese ohne weitere Prüfung. Hinzu kommt das in den meisten Fällen nur ein Modell befragt wird, nicht aber ein zweites oder die Prüfung durch ein weiteres erfolgt.

1.3 Stand der Forschung

Gerade in den letzten Monaten sind viele Forschungsfelder zum Thema Sprachmodelle hinzugekommen. Diese befassen sich mit der Optimierung und effizienter Nutzung der Modelle bei der Generierung von Codes.

In [5] wird eine bis dato fehlende Literaturrecherche zum Thema „Codegenerierung durch große Sprachmodelle“ bemängelt, was in dieser Arbeit nachgeholt wird und im Juni 2024 wurde die Literatur zusammengetragen, welche sich mit Codegenerierung befasst.

Um die Prompts im Ingenieurwesen zu optimieren, wird in [6] die GPEI (Goal Prompt Evaluation Iteration) Methodik vorgeschlagen, welche aus vier Schritten besteht. Zuerst wird das Ziel definiert, dann ein Entwurf der Anforderung, im Anschluss die Bewertung gefolgt von der Iterationen.

Es gibt Bestrebungen kleinere Modelle, die auf Codegenerierung spezialisiert sind zu verwenden, um große teure Sprachmodelle zu ersetzen, so auch in [7]. Hier werden die Modelle als „Granite Code Models“-Familie zusammengefasst. Eine weitere Arbeit die sich mit kleinen Modellen befasst, ist die Arbeit [8] mit StarCoder 2. Dieses kleine Modell wurde speziell für die Generierung von Codes trainiert.

Der wissenschaftliche Artikel [9] befasst die sich ebenfalls mit der Web-Entwicklung mittel GPT-3. Hierbei wird die Verwendung von Generativ Adversarial Networks (GANs) vorgeschlagen, ein neuer Ansatz, mit der die Nachbearbeitung minimiert und die Codequalität optimiert wird.

Eine weitere Arbeit ist [10]. Diese befasst sich mit einer Umfrage zum Thema „Natural Language-to-Code“ und gibt eine Übersicht über 27 Modelle und geben einen Überblick über Benchmarks und Metriken. Hier wird auch der in dieser Arbeit angewandte Benchmark *HumanEval* vorgestellt.

1.4 Zielsetzung und Thesen

Das Ziel in der Softwareentwicklung war und ist die Optimierung des Entwicklungsprozesses, um Ressourcen und Kosten einzusparen und dadurch einen Wettbewerbsvorteil zu erlangen. Ein Bereich der Besonders stark von der Digitalisierung profitiert, ist der Bereich der Webentwicklung. Durch die steigende Nachfrage von Cloud-Anwendungen steigt auch der Optimierungsdruck in diesem Bereich besonders stark.

Vor diesem Hintergrund lässt sich die erste These bereits aus dem Titel „*Evaluierung und Optimierung von Large Language Models für die Entwicklung von Webanwendungen*“ dieser Arbeit herleiten. Es soll gezeigt werden das die Möglichkeit besteht aus natürlicher Sprache Code für die Webanwendungsentwicklung mithilfe von LLMs zu generieren und somit tragen LLMs zur Optimierung des Softwareentwicklungsprozesses mit bei. In dieser Arbeit wird, wie schon in der Arbeit [5, vgl. Seite 2] die Abkürzung NL2Code (Natural-Language-to-Code) verwendet.

Es sind in den letzten Jahren eine Vielzahl von Benchmark-Test für die Evaluierung von NL2Code Problemen entstanden. Unter anderem der HumanEval-XL, der eine Erweiterung des HumanEval ist. Dieser erweiterte Benchmark enthält verschieden Test für unterschiedliche Programmiersprachen, auch solche die für die Webanwendungsentwicklung wichtig sind. Darunter einen Benchmark für PHP und Javascript. Trotz ihrer vielen spezifischen Tests auf verschiedene Programmiersprachen ist die

zweite These, dass diese Benchmarks sich nur bedingt für die Evaluierung von LLMs eignen, die für Webanwendungsentwicklung Code generieren.

Für NL2Code muss eine bestimmte Syntax eingehalten und Feinheiten wie Sonderzeichen müssen exakt beachtet werden. Dafür ist es relevant, dass die Eingabeaufforderungen zu optimieren. Diese Optimierungen lassen für NL2Code nicht durch die Anpassung der LLMs erreichen, was beispielsweise durch die Anpassung der Gewichte erfolgen könnte. Dies führt zur dritten These, dass die Optimierung mit der Optimierung der Eingabeaufforderungen erfolgen kann.

Die Thesen dieser Arbeit lassen sich in den folgenden kurz formulierten Sätzen zusammenfassen,

- T1** Durch den Einsatz von LLMs wird auch die Effizienz im Entwicklungsprozess gesteigert und eine Verbesserung der Codequalität erzielt. Es ist jedoch entscheidend die Leistungsfähigkeit der LLMs zu evaluieren und deren Stärken und Schwächen zu identifizieren. Es ist essenziell wichtig für den Entwicklungsprozess geeignete Methoden zu finden, um diesen Prozess optimal zu gestalten und die Vorteile des Einsatzes von LLMs im vollen Umfang auszuschöpfen.
- T2** Benchmarks, wie der HumanEval-XL eignen sich nur bedingt zur Evaluierung von LLMs die für Webanwendungsentwicklung eingesetzt werden sollen. Bei heutigen Webanwendungen werden vielmehr die Webtechnologien verschiedener Sprachen kombiniert, um nutzerfreundliche und effiziente Systeme zu erschaffen.
- T3** Eine Optimierung der LLMs zur Codegenerierung für Webanwendungsentwicklung lässt sich auch ohne tiefer gehende Änderungen der Modelle erreichen. Mit Optimierungen der Eingabeaufforderungen lassen sich signifikante Verbesserungen der Ausgaben erreichen und somit eine Verbesserung der Evaluierungen.

1.5 Aufbau der Arbeit

Um ein grundlegendes Verständnis der Thematik zu gewährleisten, werden die theoretischen Grundlagen in Kapitel 2 ausführlich erläutert.

Kapitel 4 widmet sich der Implementierung, die für die Codegenerierung und die nachfolgende Evaluierung notwendig ist. Die daraus gewonnenen Ergebnisse werden anschließend in Kapitel 5 analysiert und diskutiert.

Die in dieser Arbeit gesammelten positiven und negativen Erfahrungen sowie die aufgetretenen Herausforderungen werden in Kapitel 6 thematisiert. Zudem werden mögliche Lösungsansätze vorgeschlagen.

Abschließend werden in Kapitel 7 die erzielten Ergebnisse eingehend diskutiert und mögliche Anregungen für zukünftige Arbeiten und den praktischen Einsatz vorgestellt, bevor in Kapitel 8 die Arbeit zusammengefasst und ein abschließendes Fazit gezogen wird.

1.6 Abgrenzung

In dieser Arbeit liegt der Fokus auf der Evaluierung und Optimierung von durch Large Language Models (LLMs) generiertem Code im Kontext der Webanwendungsentwicklung, insbesondere in Bezug auf die verwendete Programmiersprache PHP. Andere Anwendungsbereiche wie die Entwicklung von Desktop-Anwendungen werden nicht explizit untersucht, obwohl mögliche Parallelen und Erkenntnisse in diesen Bereichen nicht ausgeschlossen werden. Diese Arbeit bleibt bewusst auf den Webentwicklungsbereich beschränkt, um eine gezielte Analyse und Optimierung der Prompts zu ermöglichen.

Rechtliche und ethische Überlegungen im Umgang mit Künstlicher Intelligenz sind zweifellos wichtige Themen. Allerdings werden diese Aspekte in der vorliegenden Arbeit nicht behandelt. Es gibt bereits umfassende Literatur zu diesen Themen, die in der Arbeit zur Kenntnis genommen werden, jedoch erfolgt keine vertiefte Auseinandersetzung damit. Der Schwerpunkt liegt vielmehr auf der technischen Evaluierung und der Verbesserung der Prompt-Strategien, ohne dabei Änderungen an den LLMs selbst vorzunehmen, wie beispielsweise Bias-Anpassungen oder Modifikationen am Modell.

Die Arbeit beschränkt sich auf die Anwendung von LLMs im Bereich der Webanwendungsentwicklung. Andere Anwendungsfälle, wie die Generierung von Texten für kreative Inhalte oder wissenschaftliche Artikel, werden nicht in die Untersuchung einbezogen. Ziel ist es, die spezifischen Anforderungen und Herausforderungen der Webentwicklung in den Fokus zu rücken und gezielt Optimierungen für diesen Bereich zu erarbeiten.

Der Schwerpunkt der Optimierung liegt ausschließlich auf der Anpassung und Verfeinerung der Eingabeprompts. Es wird bewusst darauf verzichtet, Änderungen an den zugrundeliegenden Modellarchitekturen, den Trainingsdaten oder der internen Bias-Reduktion der LLMs vorzunehmen. Diese Arbeit konzentriert sich auf die Möglichkeiten, die sich durch die geschickte Gestaltung der Prompts eröffnen, um die Qualität des generierten Codes zu verbessern.

Der Fokus dieser Arbeit liegt auf der technischen Optimierung und Evaluierung der durch LLMs generierten Codes. Aspekte wie Benutzerfreundlichkeit, Design oder User Experience der resultierenden Webanwendungen werden in dieser Untersuchung nicht betrachtet. Ziel ist es, die technische Qualität und Funktionalität des Codes zu analysieren und zu verbessern.

Die Untersuchung konzentriert sich ausschließlich auf deutschsprachige Prompts und die daraus generierten Codes. Andere Sprachen oder Mehrsprachigkeit werden in dieser Arbeit nicht berücksichtigt. Diese Einschränkung ermöglicht eine präzisere Analyse und Vergleichbarkeit der Ergebnisse innerhalb des gewählten Sprachraums.

In diesem Kapitel werden die grundlegenden Konzepte vorgestellt, die für das Verständnis dieser Arbeit relevant sind und ein Überblick über die wichtigsten Teilgebiete gegeben. Die einzelnen Bereiche werden nur oberflächlich behandelt, um eine grundlegende Einführung zu bieten.

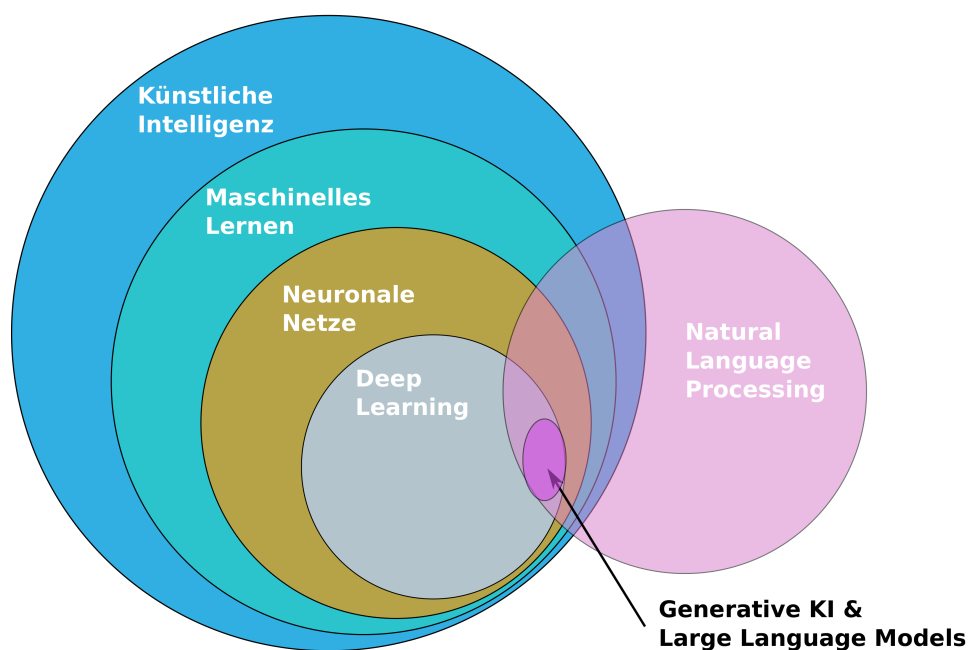


Abbildung 2.1: LLMs im Kontext der Forschungsbereiche von KI

Die Forschungsfelder der großen Sprachmodelle (Large Language Models, LLM) sind eng mit den Teilbereichen Deep Learning (DL) und der Verarbeitung natürlicher Sprache (Natural Language Processing, NLP) verbunden. Diese Teilgebiete sind wiederum Unterbereiche des maschinellen Lernens, das selbst ein Teil der umfassenderen Disziplin der künstlichen Intelligenz ist. Die Grafik 2.1 veranschaulicht die sprachliche und konzeptionelle Einordnung der Begriffe „künstliche Intelligenz“, „maschinelles Lernen“, „neuronale Netze“, „DL“ und „NLP“.

2.1 Künstliche Intelligenz

Künstliche Intelligenz hat bereits in viele Unternehmensprozesse Einzug gehalten und unterstützt die Nutzer und Systeme. Besonders die generative KI, mit ihren großen Sprachmodellen wird in den nächsten Jahren immer weiter in die Unternehmensbereiche vorstoßen und viele Aufgaben übernehmen. Entscheider und Führungspersonal versprechen sich von der Technologie nicht nur effizientere Prozesse, sondern auch Kosteneinsparungen im Personalbereich.

Eine explizite Definition für *künstliche Intelligenz* ist zurzeit noch nicht einheitlich erfolgt. Geschuldet ist diese Tatsache, dass der Begriff *Intelligenz* nicht eindeutig definiert ist. Somit finden sich viele Versuche eine Definition für künstliche Intelligenz herzuleiten. In dieser Arbeit wird für die künstliche Intelligenz, die Definition aus [11, 6 ff.] verwendet.

Systeme der künstlichen Intelligenz (KI-Systeme) sind vom Menschen entwickelte Softwaresysteme (und gegebenenfalls auch Hardwaresysteme), die in Bezug auf ein komplexes Ziel auf physischer oder digitaler Ebene handeln, indem sie ihre Umgebung durch Datenerfassung wahrnehmen, die gesammelten strukturierten oder unstrukturierten Daten interpretieren, Schlussfolgerungen daraus ziehen oder die aus diesen Daten abgeleiteten Informationen verarbeiten, und über das bestmögliche Handeln zur Erreichung des vorgegebenen Ziels entscheiden. KI-Systeme können entweder symbolische Regeln verwenden oder ein numerisches Modell erlernen, und sind auch in der Lage, die Auswirkungen ihrer früheren Handlungen auf die Umgebung zu analysieren und ihr Verhalten entsprechend anzupassen.

Bitkom e.V.

Aus den Forschungsgebieten der künstlichen Intelligenz sind für die großen Sprachmodelle der Bereich des DL besonders interessant. Hier findet die Überschneidung mit dem Bereich der NLP statt, welche massiv dazu beitrug, dass die großen Sprachmodelle diesen Erfolg erfahren. In den folgenden Kapiteln wird auf die Teilgebiete eingegangen, in denen DL angesiedelt ist.

2.1.1 Maschinelles Lernen

Als Teilgebiet der künstlichen Intelligenz befasst sich maschinelles Lernen mit dem Problem wie Maschinen Lernen und Denken können. Wobei hier nicht von selbstständigem Lernen und Denken gesprochen werden kann, sondern lediglich von Imitieren dieser Prozesse. Aber ML ist sehr wohl in der Lage aus großen Datenmengen komplexe Muster und Funktionen zu erkennen. Für das maschinelle Lernen gibt es mehrere Formen von Lernparadigmen.

Beim *überwachten Lernen* sind für die Eingaben der Trainingsdaten dazugehörige Ausgaben, die Labels definiert. Das Ziel ist es eine Funktion zu trainieren um künftige Eingaben korrekt klassifizieren oder

vorhersagen zu können. Dieses Lernparadigma wird häufig eingesetzt, wenn es sich um Regressionens- und Klassifizierungsprobleme handelt.

Die gelabelten Ausgaben sind beim *unüberwachten Lernen* nicht vorhanden. Hierbei wird beispielsweise durch Clustering oder Dimensionsreduktion versucht Muster und Strukturen zu erkennen. Des Weiteren soll die Methode helfen Anomalien in Daten zuerkennen oder Assoziationen zwischen Datenobjekten zu finden.

Das *selbst überwachte Lernen* ermöglicht es Modellen, sich selbst zu überwachen ohne gelabelte Daten. Hierbei lernen die Algorithmen einen Teil der Eingaben von anderen Teilen und generieren automatisch Labels. So werden unüberwachten Problemen in überwachte Probleme überführt. Diese Art des Lernens ist u.a. besonders nützlich bei NLP, da hier die Trainingsdaten in großer Anzahl vorliegen

Beim *verstärkten Lernen* (engl. Reinforcement Learning) werden die Systeme mit Belohnung und Strafe trainiert. Das System wird aufgrund seines Handelns bewertet, dadurch wird es ermutigt gute Praktiken weiterzuverfolgen und schlechte zu verwerfen. Das Lernen wird häufig bei der Videospielentwicklung und in der Robotik eingesetzt.

Das *semi-überwachte Lernen* stellt eine Hybridform aus unüberwachtem und überwachtem Lernen dar. Bei diesem Ansatz werden kleine Mengen gelabelter Datensätze genutzt, um eine große Anzahl ungelabelter Daten effizient zu steuern. Diese Methode ermöglicht es den verwendeten Technologien, neue Inhalte zu generieren und bildet die Grundlage moderner generativer KI-Systeme. Die Bandbreite der Technologien in diesem Bereich reicht von Generative Adversarial Networks (GANs) bis hin zu Diffusionsmodellen, welche wesentliche Fortschritte in der Erzeugung von Texten, Bildern und anderen Inhalten ermöglichen.

2.1.2 Neuronale Netze

Künstliche neuronale Netze (KNN) sind spezifische Algorithmen des maschinellen Lernens, die von der Struktur und der Funktionsweise des menschlichen Gehirns inspiriert sind. Die Abbildung 2.2 von [12] zeigt eine stark vereinfachte biologische Nervenzelle.

Biologische Nervenzellen reagieren auf elektrischen Reize, welche von Sinnesorganen oder anderen Nervenzellen stammen. Die Dendriten nehmen die elektrischen Signale auf und leiten diese an den Zellkern weiter. Dort erfolgt die Zusammenführung der eingehenden Signale und bildet das Aktionspotential. Übersteigt es das Schwellenpotential der Zelle, so wird das Signal über das Axon abgeleitet, die Nervenzelle „*feuert*“. Diese Beschreibung ist die grundlegende Ausführung über die Arbeitsweise der biologischen Nervenzellen und erläutert und beachtet keine tiefer gehenden Prozesse.

Die nachempfundene Nervenzelle stellt die kleinste Einheit in künstlichen neuronalen Netzen und wird als Neuron bezeichnet. Die Abbildung 2.3 zeigt die generelle Funktionsweise eines einfachen Neurons.

Neuronen in neuronalen Netzen erhalten als Eingabewerte typischerweise einen Vektor und liefern

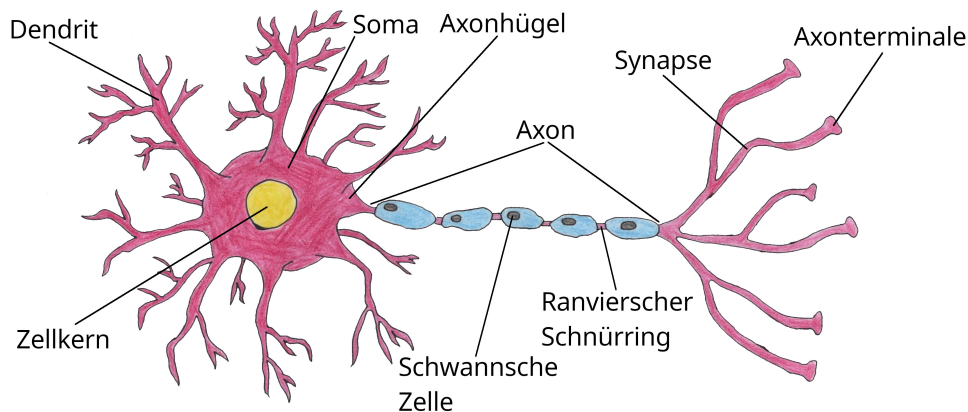


Abbildung 2.2: Biologische Nervenzelle

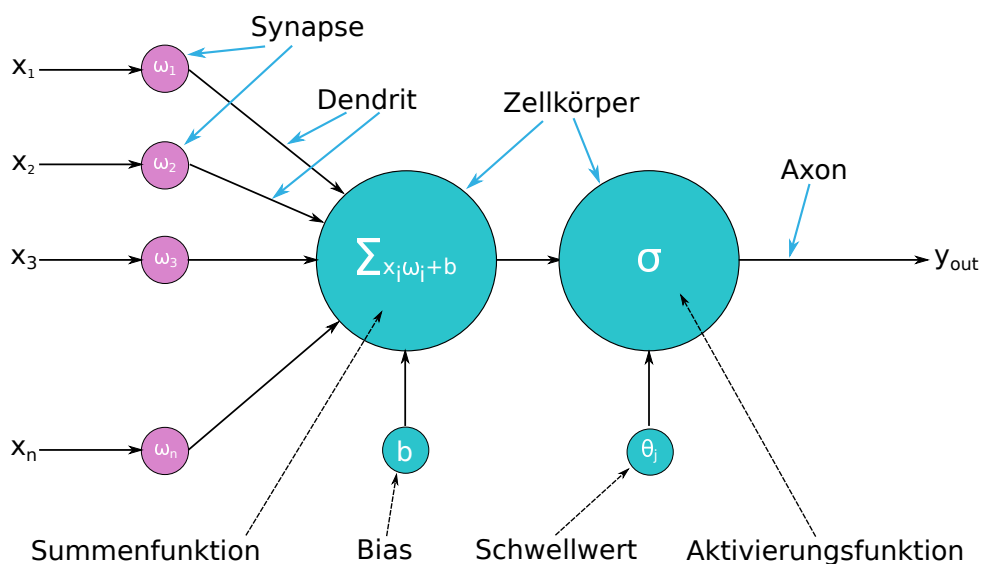


Abbildung 2.3: Künstliche Nervenzelle

ein skalares Ausgangssignal. Abgesehen von der Eingabeschicht, bei der die Eingangssignale direkt aus den Daten stammen, ist jedes Eingangssignal x_n in den verborgenen und Ausgabeschichten das Ausgangssignal y_{out} eines vorhergehenden Neurons. Die Gewichtungen der Eingangssignale modellieren dabei die synaptischen Verbindungen zwischen biologischen Neuronen, die entweder verstärkend oder hemmend wirken können. Alle gewichteten Eingangssignale werden durch die Summenfunktion aufaddiert, bevor ein Bias hinzugefügt wird. Die Gleichung 2.1 zeigt die Summenfunktion für n Eingangssignale unter Berücksichtigung des Bias-Wertes.

$$y_{sum} = x_1 + x_2 + \dots + x_n + b \quad (2.1)$$

Nach der Summenfunktion wird das Signal an die Aktivierungsfunktion übergeben. Diese Funktion

leitet ein Signal erst weiter, wenn ein festgelegter Schwellwert überschritten wird. Die Analogie zur biologischen Nervenzelle ist das Aktionspotential, welches durch die Reize anderer Nervenzellen aufgebaut wird und wie beim künstlichen Neuron führt das Überschreiten eines Schwellenwertes dazu, dass das Neuron „feuert“. Die Formel 2.2 zeigt das Verhalten einer „Binary Step“-Aktivierungsfunktion mit vorgegebenen Schwellenwert S .

$$\sigma(y_{sum}) = \begin{cases} 1 & : y_{sum} > S \\ 0 & : \text{sonst} \end{cases} \quad (2.2)$$

Neben dieser einfachen Aktivierungsfunktion wie die *Binary Step* gibt es viele weitere Aktivierungsfunktionen, beispielsweise die *Sigmoidfunktion* oder *ReLU (Rectified Linear Unit)* Funktion. Diese Aktivierungsfunktionen verwenden für die Berechnung immer das Ergebnis der Summenfunktion. Es gibt auch Aktivierungsfunktionen die alle Neuronen einer Schicht zur Berechnung verwenden. Zu diesen Funktionen zählen u.a. die Softmax- und die Maxout-Aktivierungsfunktion.

Das eben beschriebene Neuronen-Modell ist ein einfaches Modell, welches oft in Netzen wie *Feedforward Neural Netzwerke (FNN)*, *Rekurrente neuronale Netze (RNNs)* oder *Long Short-Term Memory Networks (LSTM)* Anwendung findet. Andere Neuronen-Modelle wie beispielsweise das *Leaky-Integrate-And-Fire* Modell, finde seine Anwendung in gepulsten Netzwerken. Mit diesen mathematischen Modellen wird versucht das biologische Nervensystem nachzubilden, mit all seinen Stärken und Schwächen. Die Forschung hat in den letzten Jahren bereits große Fortschritte gemacht, aber mit immer besser werdender Technik und verbessertem Verständnis der biologischen Abläufe ist das Potenzial der neuronalen Netze noch nicht erschöpft.

2.1.3 Deep Learning

Das Teilgebiet *Deep Learning* versucht möglichst präzise Vorhersagen und Entscheidungen aus komplexen Daten zutreffen. Hierfür werden tiefe neuronale Netze verwendet. Das sind Netze mit mehreren versteckten Schichten zwischen der Ein- und Ausgabeschicht. Jede Schicht verarbeitet Daten, extrahiert Merkmale und übergibt diese an die nächste Schicht. Diese Struktur und Arbeitsweise erlauben die Verarbeitung und Analyse komplexer Datenmuster in großen Datenmengen.

Durch den Einsatz von Deep Learning Methoden können relevante Merkmale aus Daten extrahiert werden ohne manuelles Feature-Engineering. Auch erreichen die Methoden eine hohe Genauigkeit und können an eine Vielzahl von Aufgaben angepasst werden.

Die großen Herausforderungen im Bereich Deep Learning liegen in den riesigen Datenmengen, die hohen Rechenleistungen welche das Training benötigt und die schwierige Nachvollziehbarkeit der durch Deep Learning getroffenen Entscheidungen.

Deep Learning hat den vergangenen Jahren viele Anwendungsbereiche erschlossen, darunter auch

natürliche Sprachverarbeitung, wie Übersetzung, Textveränderung oder Textgenerierung.

2.2 Natural Language Processing

Natural Language Processing ist ein Teilgebiet der Informatik und nutzt Deep Learning. NLP soll es digitalen Systemen in die Lage versetzen Texte und Sprachen zu erkennen, um diese zu verstehen und verarbeiten zu können. Dabei muss NLP die Bedeutung (Semantik) der Texte erkennen, die Grammatik und Beziehungen zwischen den Teilen der Sprache herstellen, Wortarten wie Verben, Adjektive und Nomen spezifizieren, sowie verschiedene Formen der Sprache beherrschen wie beispielsweise Prosa oder wissenschaftliches Schreiben.

NLP wird aber auch in anderen Bereichen eingesetzt. Mithilfe von NLP können Bilder generiert, Suchmaschinen abgefragt, Chatbots für den Kundenservice betrieben werden und Sprachassistenten wie Amazon Alexa, MS Cortana und Apple Siri nutzen ebenfalls die NLP Techniken.

Zunehmend findet NLP Einsatz im unternehmerischen Bereich. Hier werden vor allem Prozesse automatisiert um die Produktivität der Mitarbeiter zu steigern. Neben Aufgaben wie Kundensupport, Datenanalyse oder Dokumentenverwaltung kommt NLP auch in der Entwicklung von Software zum Einsatz. Hierbei werden fast alle Segmente der Entwicklung abgedeckt, von der Codegenerierung über Test und Qualitätsmanagement bis hin zur Bereitstellung.

Die ersten große Erfolge hatte NLP mit neuronalen Netzen, beispielsweise mit *Feedforward Neural Networks* und *Convolutional Neural Networks*, wie [13] zeigt. Mit der Einführung von ChatGPT und BERT, wurde auch hier die neuen Transformer Modellen eingesetzt. Die Forschungen im Bereich NLP haben die großen Sprachmodelle erst ermöglicht.

2.3 Large Language Model

Die Teilgebiete Deep Learning und Natural Language Processing haben es den großen Sprachmodellen LLM ermöglicht kommunikationsfähig zu werden. Sie verstehen Anfragen und können Antworten generieren. Die LLMs sind in der Lage Bilder und andere Medien wie Video oder Audio zu generieren.

Diese Modelle wurden mit sehr großen Datenmengen trainiert und sind daher in der Lage natürliche Sprache zu verstehen.

2.3.1 Grundlagen

Die großen Sprachmodelle können menschliche Sprache arbeiten. Sie sind speziell für die Lösung sprachbezogene Probleme geeignet, wie Textgenerierung, Klassifizierung und Übersetzung. Sie nehmen Anfragen sog. *Prompts* entgegen und errechnen daraus die wahrscheinlichste Antwort. Des Weiteren

können Prompts als Anweisung (instruction-tuning) oder in Dialogform (chat fine-tuning) gestellt werden. Die meisten der heutigen großen Sprachmodelle verwenden die Transformer Technik.

Die grundlegende Funktionsweise der Large Language Models kann in vier Teilschritte unterteilt werden,

1. Tokenisierung: zerlegen der Texte in einzelne Token.
2. Embedding: Vergleiche mit anderen Vektoren und Einordnung in einer Gesamtstruktur.
3. Vorhersage: Wahrscheinlichkeit des nächsten Tokens berechnen.
4. Decodierung: Auswahl der Ausgabestrategie.

Durch die mögliche hochgradige Parallelisierung kann in diesen Modellen die Trainingszeit erheblich verkürzt werden, was ihre Effizienz steigert. Des Weiteren sind diese Modelle sehr flexibel und für verschiedene Aufgaben eingesetzt werden.

Diese Technik hat die Leistungsfähigkeit der LLMs erheblich gesteigert und versetzt Modelle in der Lage den Kontext von Worten in Sätzen zu verstehen und Abhängigkeiten zu modellieren. Beispiele für Modelle die diese Technik nutzen, sind unter anderem GPT-3, BERT und T5.

2.3.2 Grenzen und Probleme bei LLMs

Auch wenn Künstliche Intelligenz mit ihren großen Sprachmodellen in vielen Bereichen der privaten Nutzung und in den Prozessen von Unternehmen immer präsenter wird, hat diese Technologie auch Grenzen. Im folgen werden kurz die wichtigsten Grenzen und Probleme erläutert.

Ressourcenverbrauch

Mit dem Aufkommen der großen Sprachmodelle ist auch der Verbrauch an Ressourcen enorm angestiegen. Dabei stehen diese nur in einem begrenzten Maß zur Verfügung. Kleine und mittlere Unternehmen kommen hier schnell an ihre Grenzen und nutzen daher die Modelle der Anbieter wie OpenAI, Google oder Microsoft. Auch hier gilt Ressourcenbegrenzung, sodass die Modelle nicht unendlich groß werden können. Die folgenden Ressourcen, die hier genannt werden, haben direkten Einfluss auf die Modelle und deren Betrieb,

- Speicher
- Rechenleistung
- Netzwerk
- Energie
- Finanzen

Im Lebenszyklus der großen Sprachmodelle werden Ressourcen in unterschiedlichen Mengen benötigen. So ist beispielsweise die Rechenleistung beim Training der Modelle enorm hoch.

Datenmengen

Ein weiteres Problem, vor dem die Entwickler der Modelle stehen, sind die riesigen Datenmengen, die große Sprachmodelle für das Training benötigen. Diese Daten müssen zuvor gesammelt, kuratiert und gelabelt werden. Diese Prozesse sind zeitaufwändig und kostspielig.

Interpretierbarkeit

Da die Ergebnisse stark von den Trainingsdaten abhängen, die nicht immer offen liegen und die Modelle hochkomplex sind, ist ein Ergebnis schwer nachvollziehbar. Dies erschwert auch das Aufspüren von Fehlern im Bias und den Modellen.

Probleme bei der Generierung von Texten

Vor allem bei der Generierung von Texten können Halluzinationen auftreten. Dann erzeugen die Modelle falsche oder unsinnige Informationen, die aber plausibel erscheinen. Ein weiteres Problem ist die Erzeugung diskriminierender Informationen. Diese entstehen beispielsweise durch nicht repräsentative Trainingsdaten, fehlerhafte Label in Trainingsdatensätzen und fehlendes Gesamtbild.

Die genannten Probleme und Grenzen lassen sich auch bei der Codegenerierung beobachten. Die generierten Codes können plausibel erscheinen, funktionieren aber nicht oder werfen Fehler. Fehlenden Informationen über geeignete Frameworks und Funktionen veranlassen die Modelle Klassen und Methoden aus anderen Kontexten abzuleiten oder vermischen Codesegmente.

2.3.3 Verständnis für die LLMs

Viele Nutzer (Privatnutzer aber auch Firmen) wissen nicht, was hinter den großen Sprachmodellen steckt oder wie diese funktionieren. Dieses Unwissen birgt die Gefahr, dass Nutzer nicht korrekte Eingabe in die LLMs übergibt und dann die Ergebnisse der LLMs falsch interpretieren oder die LLMs nicht korrekte Aussagen trifft. Werden aufgrund dieser falschen Ergebnisse Entscheidungen getroffen, können diese enorme finanzielle und personelle Einbußen nach sich ziehen. Zudem kann es weiterhin zu Desinformation, Diskriminierung, juristische Probleme und zum Vertrauensverlust in die Technologie führen.

Um diesen Problemen bei Entwicklern entgegenzuwirken, sind vor, während und nach der Einführung einer LLM zur Codeentwicklung, die Nutzer aufzuklären. Sie müssen sich im Klaren sein, dass LLMs Fehler produzieren können und es erforderlich ist, die Ergebnisse zu validieren. Nur so kann einem Vertrauensverlust entgegengewirkt werden und eine stetige Weiterentwicklung der Modelle erfolgen.

2.4 Prompt Engineering

Prompt Engineering optimiert die Antworten große Sprachmodelle, ohne Parameter, wie Bias und Gewichte des Models ändern zu müssen. Dieser Forschungsbereich hat in den letzten Jahren enorm an Bedeutung gewonnen und sich zu einer eigenen Disziplin im Bereich der Künstlichen Intelligenz entwickelt.

Ein Prompt oder Anweisung muss entweder als Anweisung oder als Frage gestellt werden. Dies kann, wie in [14] beschrieben, in Form von einer einfachen Anweisung bis hin zu detaillierten Beschreibungen oder spezifischen Aufgaben erfolgen.

2.4.1 Prompt-Techniken

Die folgenden Techniken dienen dazu die Abfragen zu optimieren und somit eine bessere Antwort von den Sprachmodellen zu erhalten. In dieser Arbeit werden hauptsächlich *Contextual Augmentation* und *Meta-Prompts* verwendet.

Contextual augmentation sind Prompts die neben der eigentlichen Eingabeaufforderung zusätzliche Informationen für die LLMs bereitstellen. Dazu gehören Kontextangaben, anzuwendende Programmiersprache, geforderte Bibliotheken, Angaben der Zielgruppe und Definition über die auszugebende Struktur wie Methoden oder Klassen.

Beispiel für einen kontextuellen Prompt:

Du bist ein Erfahrener PHP Programmierer, hier deine Aufgabe. Erstelle eine PHP-Funktion, die eine Liste von Ganzzahlen sortiert und zugibt.

Ausgabe der LLM:

```
1  function sortiere_und_gebe_aus($zahlen) {  
    // Die Liste wird nach aufsteigender Reihenfolge sortiert  
    sort($zahlen);  
  
    // Die sortierte Liste wird ausgegeben  
6  foreach ($zahlen as $zahl) {  
    echo $zahl . " ";  
    }  
}
```

Diese Art der Prompts können hilfreich sein, wenn Erweiterungen oder Änderungen in bestehende Programme erfolgen. Hier sind oft schon Standards oder Bibliotheken definiert.

Meta-Prompts sind Prompts, die eine LLMs anweisen, einen Prompt zu erstellen.

Meta-Prompt Beispiel:

Du bist eine Prompt-Engineer und hier ist deine Aufgabe. Schreibe einen Prompt, der eine PHP-Funktion generiert, die eine Methode erstellt, die eine Liste von Ganzzahlen sortiert.

Ausgabe der LLM:

Schreibe eine PHP-Funktion namens `sortiereZahlen`, die ein Array von Ganzzahlen als Parameter entgegennimmt. Die Funktion soll das Array in absteigender Reihenfolge sortieren und das sortierte Array zurückgeben. Nutze eine geeignete PHP-Funktion für die Sortierung.

Es gibt weitere Prompt-Typen, die nicht relevant für diese Arbeit sind. Eine Zusammenfassung von Methodiken und deren Beurteilung kann [15] entnommen werden.

2.4.2 Grenzen beim Prompt-Engineering für LLMs

Trotz der bemerkenswerten linguistischen Leistung, stoßen große Sprachmodelle an ihre Grenzen, unter anderem wie in [14] beschrieben. Oft ist die Komplexität der Sprache für die LLMs ein großes Hindernis, da diese oft mehrdeutig sein kann und zu unerwarteten Aussagen seitens der LLM führen kann. Ebenso haben LLMs Probleme mit Sarkasmus und Ironie, was ebenfalls zu nicht erwünschten Ausgaben führen kann.

Ein weiteres Kriterium sind die Modellbeschränkungen. Die Qualität der Trainingsdaten entscheidet oft über die Qualität der Antwort. Ebenfalls wirkt sich die Architektur auf die Art der zu bewältigenden Aufgaben aus. Nicht alle Modelle sind für alle Prompts geeignet. Modelle können zwar Muster in Daten erkennen, ihnen fehlt aber ein tiefes Verständnis für die Welt und ihnen fehlen Zusammenhänge zwischen Ereignissen.

Ein großes Problem stellen die praktischen Herausforderungen. Da ein guter Prompt ein iterativer Prozess ist, der Zeit und Geduld erfordert, welche oft nicht investiert wird. Hinzu kommen die Kosten kommerzieller Modelle, welche sich negativ auf die Gesamtkosten von Softwareprojekten auswirken kann.

2.4.3 Promptoptimierung mit DSPy

Das DSPy Framework wurde im Oktober 2023 und baute auf das im Dezember 2022 veröffentlichte DSP auf und wurde von einer Forschungsgruppe der Stanford University entwickelt, die sich mit Natural Language Processing befasst. Das Framework ist in Python entwickelt und erlaubt einen deklarativen Programmieransatz für Eingabeaufforderungen, dadurch entfällt deren aufwendige Entwicklung.

Der Entwickler muss nur die Aufgabe definieren und die Kennzahlen festlegen, DSPy optimiert die Eingabeaufforderungen anhand der Vorgaben. Durch den modularen Aufbau können verschiedene vorhandene und eigene Module kombiniert und leicht erweitert werden, sodass ganze Verarbeitungsketten entstehen. Es existiert bereits eine Reihe von Modulen, die das Framework mitbringt, wie

beispielsweise ChainOfThought oder ReAct. Somit können das Ketten an die Bedürfnisse der Aufgaben leicht angepasst werden. Des Weiteren bringt das Framework eine Reihe von Optimierer Techniken mit. Darunter sind z.B. BootstrapFewShot oder MIPROv2. Diese Techniken werden von dem Entwicklerteam kontinuierlich weiter entwickelt.

Das Framework bringt, durch die genannten Komponenten verschiedene Vorteile mit sich. Dadurch, dass das aufwendige und durch „Versuch und Irrtum“-Prompt-Engineering entfällt, reduzieren sich dadurch die Fehler. Die Prompts werden durch das Framework automatisiert optimiert, was zu einer einfacheren Entwicklung mit LLMs führt. Ein weiterer Vorteil ist die leichte Anpassbarkeit an veränderte oder erweiterte Aufgaben. Dazu muss nur die Aufgabenstellung, die Kennzahlen und die Module angepasst oder zusätzliche eingebunden werden.

Nachfolgen ein Beispiel für die Verwendung des Frameworks. Die Codesegmente für die Beispielaufgabe sind von Gemini 2.0 Flash erstellt wurden. Zuerst wird die Bibliothek dspy importiert und im Anschluss wird das Modell konfiguriert. In dem Fall ist es das GPT 3.5 von OpenAI.

```
1 import dspy

# Konfiguration des LLM (hier: OpenAI GPT-3.5-turbo)
llm = dspy.OpenAI(model='gpt-3.5-turbo')
dspy.settings.configure(lm=llm)
```

Ist das Model konfiguriert, wird die Signatur festgelegt. Mit ihr wird das Verhalten des Moduls bestimmt. Die Kommentare in der Klasse und in den Parameterdefinitionen werden vom Framework ebenfalls für die Optimierung verwendet.

```
# Definition der Aufgabe als Signatur
class FrageBeantworten(dspy.Signature):
    """Beantworte die gestellte Frage."""
    frage = dspy.InputField()
5    antwort = dspy.OutputField(desc='Die Antwort auf die Frage.')
```

Dann erfolgt die Definition des Moduls, indem die zuvor erstellte Signatur verwendet wird, welches im Konstrukt der Klasse EinfachesModul erfolgt. Durch die Methode vorwaerts wird im Predict-Modul die Abfrage bei der LLM ausgelöst und die Antwort zurückgegeben.

```
# Erstellung des Moduls
class EinfachesModul(dspy.Module):
    def __init__(self):
        super().__init__()
5        self.vorhersager = dspy.Predict(FrageBeantworten)

    def vorwaerts(self, frage):
        return self.vorhersager(frage=frage)
```

```
10 # Initialisierung des Moduls  
    modul = EinfachesModul()
```

2.5 Grundlagen der Webentwicklung

In diesem Unterkapitel soll kurz auf Anforderungen der Webentwicklung eingegangen werden.

2.5.1 Programmiersprachen

Grundsätzlich kann jede Programmiersprache verwendet werden. Es gibt jedoch Programmiersprachen, die explizit für Webanwendungen entwickelt wurden und einige Funktionen mitbringen, welche die Entwicklung vereinfachen. Eine der am weitesten genutzten Programmiersprachen für Webanwendungen ist PHP. Die meisten visuellen Anwendungen generieren HTML Code als Grundgerüst und generieren CSS Dateien für das Layout, die als Standardformatierungssprache gilt. Anwendungen die als RestAPI fungieren, liefern in den meisten Fällen ihre Ausgaben in Form von JSON aus. Neben JSON Format gibt es weitere beispielsweise XML oder YAML.

2.5.2 Entwicklung

Bei der Entwicklung von Webseiten werden längst schon dieselben Prozesse und Tools verwendet wie bei anderen Softwareprojekten. An deren Anfang steht auch hier der Entwickler, welcher durch KI Unterstützung erhalten kann.

Technische Anforderungen

Entwickler sollten Grundlagen in HTML, CCS und JavaScript aufweisen und zudem Programmierkenntnisse für die serverseitige Entwicklung in PHP, Python, Java oder Node.js besitzen. Neben den Kenntnissen in den erforderlichen Programmiersprachen müssen Webanwendungsentwickler Wissen im Bereich von Datenbanken, wie *MySQL* oder *PostgreSQL* aufweisen und Frameworks beherrschen, wie beispielsweise *React*, *Flask* oder *Symfony*, je nach Programmiersprache. Des Weiteren sind Fähigkeiten für die Bereiche Versionskontrolle nötig, da die meisten Entwicklungsfirmen oft GitLab oder GitHub hierfür einsetzen. Auch Wissen in den Bereichen Web-Performance und Web-Security sind wichtige Aspekte für Entwickler die immer mehr an in den Vordergrund rücken.

Design und UX

Design und UX ist ein weiteres Feld, welches Webanwendungsentwickler beherrschen müssen. Das Design entscheidet bei Webseiten beim Nutzer in sekundenschnelle über Erfolg oder Misserfolg einer Seite bzw. einer Anwendung. Darum ist es wichtig, als Entwickler ein grundlegendes Verständnis in *Usability* und *User Experience* zu entwickeln. Ebenso ist es wichtig, das Entwickler sich mit dem Thema Barrierefreiheit auseinandersetzen, um allen Nutzer einen unkomplizierten Zugang zu einer Anwendung ermöglichen, hier sei auf die WCAG¹ Richtlinien hingewiesen. Es ist essenziell, das Webentwickler, die genannten Design-Konzepte Kennen und Anwenden können.

Inhalte

Zu den Aufgaben können auch Themengebiete wie die Erstellung von hochwertigem Inhalten die informativ und ansprechend sind gehören. Eine weitere Aufgabe der Entwickler ist es Webseiten für Suchmaschinen zu optimieren (SEO) und auf die ständigen Neuerungen auf diesem Gebiet zu reagieren. Um die Anwendungen zu schützen, müssen die Systeme und verwendete Software ständig aktualisiert werden, dazu sind diese Komponenten regelmäßig zu überprüfen.

Einsatz von KI

Der Einsatz von Künstlicher Intelligenz kann in allen Entwicklungsphasen eingesetzt werden, angefangen von der Codegenerierung über die Bereitstellung mittels Pipeline bis zur Inhaltserstellung.

Der Einsatz von NL2Code steckt hier noch in den Anfängen, bietet aber sehr gute Ansätze viele Aufgaben zu automatisieren oder als unterstützenden Werkzeug, welches die Entwicklung effizienter gestalten kann.

Die Codegenerierung für Designelemente kann ebenso mittels NL2Code erfolgen, wie die Erstellung komplexer Backendfunktionalitäten. Die Konzeption für Webanwendungen kann ebenfalls durch eine LLM erfolgen oder kann diesen Prozess unterstützen.

2.6 Benchmark für LLM

Bei der Evaluierung großer Sprachmodell hinsichtlich des generierten Codes, gibt es einige Herausforderungen. Herkömmliche Methoden, wie BLUE-Score misst die Textähnlichkeiten nicht aber die funktionale Korrektheit des Codes und vernachlässigt auch den Kontext, in dem der Code erstellt wurde. Ein generierter Code kann in seiner Lösung stark von einer vorgegebenen Beispiellösung abweichen, trotzdem aber seine Funktionalität erfüllen. Menschliche Programmierer würden das mit verschiedenen Unit-Tests überprüfen, aus diesem Grund sollte der Code mit einer weiteren Methode geprüft werden.

¹Web Standard für barrierefreie Webseiten <https://www.w3.org/WAI/standards-guidelines/wcag>

Als Benchmark für die Bewertung der Sprachmodelle wird der HumanEval-XL verwendet, welche unter <https://github.com/FloatAI/humaneval-xl/tree/main> heruntergeladen werden können.

2.6.1 pass@k Methode

Für die Bewertung wird das Vorgehen gewählt, welches in [16] und [17] beschrieben ist. Die Tests werden exemplarisch, mit den für die Webentwicklung relevanten Sprachen PHP und JavaScript durchgeführt. Die Evaluierung der Modelle wird auf den Ebenen „einfache Fragen“ und „komplexe Aufgaben“ erfolgen. Die „einfachen Fragen“ werden bereits durch den zuvor genannten Benchmarks abgedeckt, sodass der entwickelte Fragenkatalog sich auf die Ebenen mit den „komplexen Aufgaben“ konzentriert.

Der pass@k ist eine Methode zur Bewertung der Leistungsfähigkeit von LLMs. Dabei wird gemessen, mit welcher Wahrscheinlichkeit eine korrekte Lösung unter den Top-k Lösungen vorhanden ist. Diese Methode wird häufig für die Bewertung von Codegenerierung eingesetzt und existiert in verschiedenen Variationen.

Ein Evaluierungsdatensatz sollte einen großen Aufgabenbereich der jeweiligen Programmiersprache abdecken, was mit dem HumanEval-XL umgesetzt wurde. Dieser Datensatz bringt 80 Aufgaben mit, mit denen Modelle geprüft werden können.

Die Methode berechnet den Erwartungswert der Modelle und bietet eine statistische Evaluation der Ergebnisse. Es werden für die Berechnung die Anzahl der Versuche pro Probe, die Anzahl der korrekt generierten Lösungen und die Gesamtanzahl der generierten Lösungen betrachtet, was die Formel 2.3 zeigt.

$$\text{pass@k} = \mathbb{E}_{\text{sample}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (2.3)$$

Dadurch dass alle korrekten Lösungen mit in die Bewertung einfließen, bietet die Methode eine umfassendere Analyse zur Bewertung von LLMs, welche bei der Codegenerierung eingesetzt wird. Nachdem alle Aufgaben einzeln betrachtet wurden, wird der Erfolgswahrscheinlichkeit des gesamten Modells berechnet, die Formel 2.4 zeigt die mathematische Berechnung.

$$\text{pass@k}_{\text{gesamt}} = \frac{1}{N} \sum_{i=0}^N \text{pass@k}_i \quad (2.4)$$

Mit dieser Methoden ist eine starke statistische Evaluation der Modelle möglich. Hierbei werden nicht nur die aggregierten Anteile der Lösungen betrachtet, sondern es fließen alle korrekten Lösungen mit in die Berechnung ein. Somit erhält man einen guten vergleichbaren Wert über die Wahrscheinlichkeit, dass das Modell in k Versuchen mindestens eine korrekte Lösung liefert.

In diesem Kapitel werden die Rahmen- und Randbedingungen für das methodische Vorgehen der Evaluation großer Sprachmodelle festgehalten, welche für die Codegenerierung eingesetzt werden. Dies umfasst die Festlegung der verwendeten LLMs, die geprüfte Programmiersprachen, die Frameworks zur Erstellung und Auswertung der Tests und die Systeme welche für die Bereitstellung und Verarbeitung der Ergebnisse zum Einsatz kommen. Die Evaluierung der Modelle erfolgt in deutscher Sprache, bei der Optimierung werden einige Modelle mit englischen Proben getestet.

3.1 Definition der Evaluierungsziele

Ausgehend, von den in Kapitel 1.4 aufgestellten Thesen dieser Arbeit, werden hier die Konzepte und Designs für die Evaluation und Optimierung besprochen. Es wird dargelegt wie Untersuchungen durchgeführt werden, um valide Aussagen zu den Thesen treffen zu können. Es wird darauf geachtet das die Ergebnisse nachvollziehbar und überprüfbar sind. Die Evaluation soll zeigen, inwieweit die LLMs korrekte Ergebnisse liefern und für die Webanwendungsentwicklung geeignet sind. Um die generierten Ergebnisse Vergleichen zu können, wird hierfür ebenfalls der HumanEval-XL Benchmark eingesetzt.

Für die Messung zur „Wahrscheinlichkeit das eine korrekte Antwort unter den TOP Antworten ist“, wird die `pass@k` Methode, für die Überprüfung der generierten Lösungen angewandt. Für jede Probe liefert der HumanEval-XL eigene Tests mit. Dieser Test zusammen mit dem generierten Code sollten einen ausführbaren und testbaren Code ergeben. Daraus kann dann mit der `pass@k` Methode die repräsentative Zuverlässigkeit des Modells für jedes Problem ermittelt und anschließend für das gesamte Modell berechnen.

Die Proben aus dem HumanEval-XL Benchmark beschreiben grundlegende Verständnisfragen für die Modelle und sind mit einer einzelnen Funktion zu beantworten. Somit kann das grundlegende Verständnis einer LLM in Hinblick auf eine, für die Webanwendungsentwicklung relevante Programmiersprache evaluiert werden.

Nach der Evaluierung wird versucht einige Modelle mit verschiedenen Ansätzen zu optimieren. Anschließend werden diese Ergebnisse mit den Evaluierungen ohne Optimierung verglichen. Hier soll

geprüft werden, ob sich die dritte These, aus Kapitel 1.4 beweisen lässt und eine Optimierung für die Webanwendungsentwicklung herbeizuführen ist, ohne die Modelle grundlegend zu ändern.

3.2 Auswahl der LLMs und deren Konfiguration

Für die Evaluation werden experimentell freie und kommerzielle Modelle ausgewählt und miteinander verglichen. Hauptsächlich wurden bei den freien Modellen, jene ausgewählt welche den Fokus auf die Codegenerierung legen und mit diesem Argument beworben werden. Als Referenzen sollen die kommerziellen Modelle *Gemini 1.5* und *ChatGPT* dienen, welches einer stetigen Verbesserung unterliegen und eine große Nutzeranzahl aufweisen.

Im Folgenden werden die ausgewählten LLMs kurz vorgestellt und warum diese gewählt wurden. Die Reihenfolge stellt an dieser Stelle keine Wertung der LLM oder über deren generierten Inhalte dar.

Das **Qwen2.5-Coder**-Modell zeichnet sich durch seine spezialisierte Architektur für die Codegenerierung aus. Trainiert, um sowohl syntaktisch korrekten als auch funktional hochwertigen Code zu produzieren, integriert es fortschrittliche Mechanismen zum Kontextverständnis und semantisch sinnvolle Ausgabe. Es findet Anwendung in verschiedenen Bereichen der Softwareentwicklung, insbesondere in der Web- und Anwendungsprogrammierung. Die Qwen2.5-Coder Modellbeschreibung ist [18] und [19] entnommen und wird in den Arbeiten vertieft.

Deepseek-Coder-V2 ist die zweite Generation der Deepseek-Coder-Reihe, von der gleichnamigen KI-Entwicklungsfirma DeepSeek und soll verbesserte Fähigkeiten zur Codegenerierung und -optimierung bieten. Das Modell nutzt fortschrittliche Suchalgorithmen, um präzisere und effizientere Codestücke zu erstellen. Es ist insbesondere für seine hohe Genauigkeit bei der Generierung komplexer Algorithmen und Datenstrukturen bekannt. Die Modellbeschreibung ist unter anderem aus [20] und [21] entnommen. Des Weiteren wird das Modell in beiden Arbeiten mit verschiedenen Open-Source und Close-Source Modellen verglichen.

Die jüngste Innovation der chinesischen KI-Entwicklungsfirma DeepSeek ist das **DeepSeek-R1** Modell. Mit seiner offiziellen Vorstellung im Januar 2025 erregte es bedeutende Aufmerksamkeit sowohl im Bereich künstlicher Intelligenz als auch an den Finanzmärkten. Laut Unternehmensaussagen gleichwertig zu closed-source -Systemen wie ChatGPT-4 oder Gemini 2.0, demonstriert das R1-Modell eine erhebliche Leistungskraft. Im Rahmen dieser Untersuchung wurde speziell die Version deepseek-r1 analysiert, die über 32 Milliarden Parameter verfügt.

Die Modelle **Llama 3.1-Claude** und **Llama 3.1** gehören mit 8 Milliarden Parametern zu den kleineren Modellen von Meta. Beide Modelle basieren auf dem Llama3.1 Modell, das Llama3.1-Claude ist aber mit anderen Systemaufforderungen erstellt wurden. Hierfür wurden die Systemaufforderungen vom Claude Sonnet 3.5 der Firma Anthropic's verwendet, nachzulesen unter [22]. Ein ähnliches Modell ist auf Hugging Face veröffentlicht [23]. Eine Modelcard mit weiteren Informationen zum Modell ist zu

finden unter [24].

Ein weiteres Modell von der KI-Entwicklungsfirma **Meta**, wurden speziell zur Codegenerierung entwickelt. Hierbei handelt es sich um das Modell *Codellama*, welches mit unterschiedlicher Parameteranzahl evaluiert wird. CodeLlama basiert auf Metas Llama2 Modell wurde im Januar 2024 veröffentlicht. Zuerst wird das Modell *Codellama:13b* evaluiert. Mit dieser Anzahl von Parametern gehört das Modell zu den kleineren Modellen. Dies wurde die Gewichte auf zwei Bit quantisiert, somit ist die Größe des Modells auf 7,4 GB reduziert wurden. Die zweite Modellkonfiguration ist das *Codellama:70b* und wurde durch eine Quantisierung Q2_K von 39 GB auf 25 GB verkleinert. Laut [25] unterstützt das Modell unter anderem Programmiersprachen wie PHP, welche für die Erstellung für Webanwendungen relevant sind.

Mistral ist ein modernes leistungsfähiges Sprachmodell, welches nicht speziell für die Codegenerierung und -analyse entwickelt wurde. Es verwendet fortschrittliche Transformer-Architekturen und ist für eine Vielzahl von Aufgaben einsetzbar. Darunter fallen beispielsweise natürliche Sprachverarbeitung, Textzusammenfassungen, maschinelle Übersetzung und Textklassifizierung. Dieses Modell ausgewählt, um ein Modell zu evaluieren, welches nicht speziell auf Codegenerierungsaufgaben trainiert wurde. In der Arbeit [26] wurde Mistral, mit verschiedenen Modelle zur Spielecodegenerierung verglichen. Während in [27] eine Evaluation für natürlichsprachlicher Erklärungen, Mistral mit anderen Modellen verglichen wurde.

Das Modell **ChatGPT 4** wurde von OpenAI entwickelt und ist ein vielseitig einsetzbares Close-Source Modell. Neben allgemeinen textuellen Einsatzgebieten, lässt es sich für die Codegenerierung nutzen. Seit November 2022 ist das Modell ChatGPT 3.5 für alle kostenlos nutzbar und wird von sehr vielen Nutzer eingesetzt. Mit diesen gesammelten Daten wurde das neue Modell ChaGPT 4 weiterentwickelt und trainiert. Dadurch werden die Modelle in ihren Vorhersagen kontinuierlich besser. Eine Übersicht der Modelle von OpenAI ist unter [28] einsehbar.

Mit **Gemini 1.5** präsentiert Google ein Modell zur Verarbeitung von natürlicher Sprache und stellt es für die freie Nutzung zur Verfügung. Wie in [29] beschrieben, setzen auch die Gemini Modelle, Transformer-Architektur ein, was sie dazu befähigt, komplizierte Sprachmuster zu erkennen und präzise Vorhersagen zutreffen. In [29] werden die Ergebnisse von Gemini 1.5, bei der Codegenerierung mit denen von ChatGPT und Copilot verglichen. Nach [30] kann sich das Gemini-Ultra-Modell beim MMLU-Benchmark sogar mit menschlichen Experten messen und bedient eine breite Palette von Anwendungsbereichen. Hier wurden die Fähigkeiten zur Codegenerierung an Sicherheitsfragen im E-Commerce Bereich getestet. Unter [31] sind weitere Informationen zu den Gemini Modelle von Google einsehbar.

Neben den genannten Quellverweisen sind die Seiten der jeweiligen KI-Entwicklungsfirma, gute Quellen für weiterführende Informationen zu deren Modellen. Die Tabelle 3.1 zeigt zusammenfassend die ausgewählten Modelle.

Modell	Param	Quantisierung	Größe	Sprache	offen	EXEC
Qwen2.5-coder	32b	q4_K_M	19 GB	DE	X	Ollama
DeepSeek-coder-V2	16b	lite-instruct-q5_K_S	11 GB	DE	X	Ollama
DeepSeek-R1	32b	q4_K_M	19 GB	DE	X	Ollama
Llama3.1-Claude	8b	q4_0	4,7 GB	DE	X	Ollama
Llama3.1	8b	q4_K_M	4,7 GB	DE/EN	X	Ollama
Llama3.2	3b	q4_K_M	2,0 GB	DE/EN	X	Ollama
Llama3.3	70b	instruct-q2_K	26 GB	DE/EN	X	Ollama
Codellama	13b	q4_0	7,4 GB	DE	X	Ollama
Codellama	70b	instruct-q2_K	25 GB	DE	X	Ollama
Mistral Small	22b	q4_0	12 GB	DE	X	Ollama
Gemini 1.5 Pro	k.A.	-	k.A.	DE	-	Online
Gemini 1.5 Flash	k.A.	-	k.A.	DE	-	Online
ChatGPT 4 Turbo	k.A.	-	k.A.	DE	-	Online

Tabelle 3.1: Auswahl der LLMs für die Evaluierung

Die Einstellung für das Abfragen der Proben, wurden bei allen Modellen, soweit es ging identisch gewählt. Ausnahmen waren die Modelle mit 70 Milliarden Parametern. Hier entfiel die Längenbeschränkung der Token.

Für die Abfragen wurde eine *temperature*-Wert von 0.2 gewählt. Ein niedriger Wert veranlasst die Modelle zu deterministischen und standardisierten Antworten. Somit wird Kreativität und Zufälligkeit im generierten Code der Modelle verhindert. Die Generierung von Programmcode soll konsistenten und präzisen Code liefern.

Ein hoher *top_p* Wert verlangt von den Modellen nur Antworten zu geben, die mit hoher Wahrscheinlichkeit korrekt sind. Für die Codegenerierung sollten nur die wahrscheinlichsten und syntaktisch korrekten Token angewandt werden. Für die Abfragen wird hier ein Wert von 0.95 angesetzt.

Die maximale Anzahl an Token sollte bei der Generierung von Programmcode zwischen 200 und 1000 Token eingestellt sein, je nach Umfang der Antworten. Da hier nur die Funktionsfähigkeit geprüft wird, werden Struktur und Coding-Standards vernachlässigt, sodass *max_token* auf 600 festgelegt wird. Eine Ausnahme wird es beim *Llama3.1* geben, um zu prüfen, ob eine höhere Tokenanzahl bessere Ergebnisse generiert.

In der Tabelle 3.2 sind die Werte in übersichtlicher kurzer Form noch einmal dargestellt.

Für die lokalen Modelle kommen weitere Parameter hinzu. Unter anderem wird der Parameter *do_sample* auf *False* gesetzt, was die Modelle veranlasst den wahrscheinlichsten folgenden Token zu wählen und ein deterministisches Verhalten fördert. Ein weiterer Parameter ist *return_full_text* der ebenfalls auf *False* gesetzt wird. Dadurch werden nur die neu generierten Tokens zurückgegeben, was die Relevanz der Antworten fördert.

Modell	Temp.	max. Token	Top-p
Qwen2.5-coder	0.2	600	0.95
Deepseek-Coder-v2	0.2	600	0.95
Deepseek-R1	0.2	offen	0.95
Llama3.1	0.2	600/1200	0.95
Llama3.1-Claude	0.2	600	0.95
Llama3.2	0.2	600	0.95
Llama3.3	0.2	600	0.95
Codellama:13b	0.2	600	0.95
Codellama:70b	0.2	offen	0.95
Mistral Small	0.2	600	0.95
Gemini 1.5 Pro	0.2	600	0.95
Gemini 1.5 ChatBot	k.A.	k.A.	k.A.
ChatGPT 4 Tubro	0.2	600	0.95

Tabelle 3.2: Einstellungen der Modellparameter

Alle Skripte zur Erstellung und Auswertung der Antworten werden mit Python realisiert und nutzen die jeweiligen APIs der Anbieter. Dies gilt für die offenen lokalen wie auch für die kommerziellen Modelle. Die Umsetzung erfolgt für die lokalen Modelle mit Python `langchain` und für die kommerziellen Modelle der Gemini-Reihe wird die Google eigene Bibliothek, `google.genai` verwendet. Dasselbe trifft für die Modelle von OpenAI zu, hierbei kommt die Python Bibliothek `openai` zum Einsatz.

3.3 Design der Evaluierung

In dem Kapitel wird das Design der Evaluierung besprochen. Der verwendete Benchmark wird vorgestellt und anschließend erläutert wie die Durchführung der Evaluierung erfolgt. Als letzter Punkt wird in diesem Kapitel die Art und Weise festgehalten, wie die Daten dokumentiert werden, um die Überprüfung und Nachvollziehbarkeit zu gewährleisten.

3.3.1 HumanEval-XL Benchmark

Das Experiment wird mit dem HumanEval-XL Benchmark durchgeführt. Dieser Benchmark besteht aus einer Reihe von 80 Proben in verschiedenen Programmier- und Landessprachen, die wie folgt aufgebaut sind,

1. **task_id**: Die Kennung der Datenprobe, als eine eindeutige ID. An ihr ist bereits die Programmiersprache erkennbar, welche verwendet wird. Ein Beispiel für die PHP-Proben ist `php/0` oder `php/1`, für JavaScript wäre das beispielsweise `javascript/0`.
2. **prompt**: Der Prompt ist die Anfrage für das Modell und besteht aus Funktionsheader und Docstring. Hier ist die eigentliche Aufgabe der Probe definiert. Dieser Teil des Benchmarks wird

später in diesem Kapitel, für die PHP Proben als *Opening Tag* und *Kommentar* ausführlicher beschrieben.

3. **entry_point**: Der Einstiegspunkt für die Probe ist der `entry_point`, in welcher der zu verwendende Methodenname explizit genannt wird. Dieser muss im Ergebnis und Test angewandt werden.
4. **test**: Die vordefinierte Testaufgaben, für die geforderte Funktion im generierten Code, wird ebenfalls später in diesem Kapitel ausführlicher beschrieben.
5. **description**: Diese enthält ausführliche Beschreibung der Aufgabe des Benchmarks, welche für den Nutzer bestimmt ist und nicht für das Modell.
6. **language**: Mit Kennung der Programmiersprache wird nochmals explizit auf diese hingewiesen, für die das Modell eine Lösung generieren soll.
7. **canonical_solution**: Diese Lösung für das Problem findet keine Verwendung, da sie im HumanEval-XL nicht verwendet wird und zudem keine Angaben enthält.
8. **natural_language**: Hier findet sich die Angabe zur Landersprache, in der die Proben erstellt wird.

Die Abbildung 3.1 zeigt den Aufbau und damit alle wichtigen Bereiche des Benchmarks. In dessen Struktur sind vier wesentliche Teilbereiche erkennbar. Dazu gehören das *Opening Tag* (optional), der *Kommentar*, das *Ergebnis* und der *Test*.

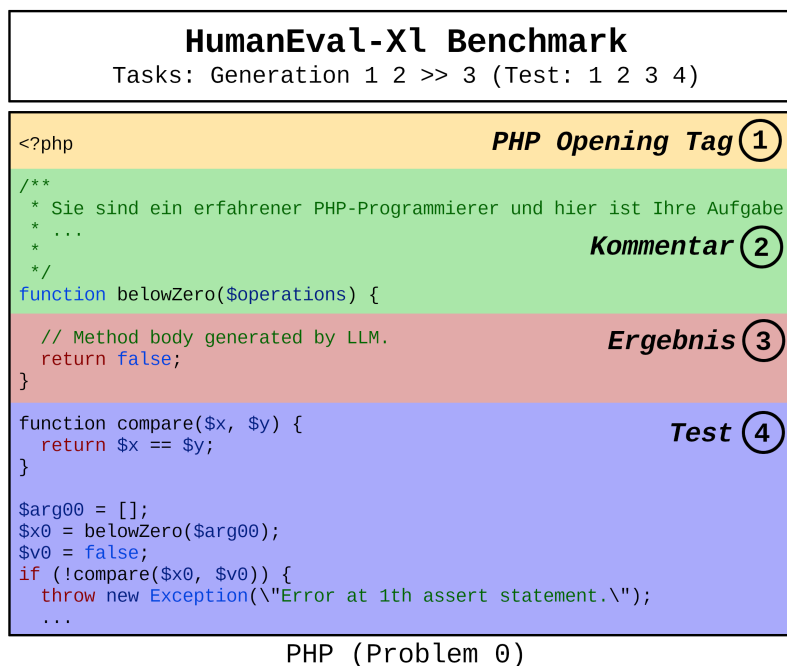


Abbildung 3.1: Aufbau des HumanEval-XL Benchmarks

Das *PHP Opening Tag*, in der Abbildung 3.1 mit 1 bezeichnet, ist eine optionale Angabe zur Programmiersprache und durch den Benchmark vorgegeben. Während bei den PHP-Proben das Opening Tag `<?php` lautet, ist beispielsweise bei den JavaScript-Proben kein Opening Tag angegeben. Hier beginnt die Probe direkt mit dem Kommentar. Der in der Abbildung 3.1 mit 2 bezeichnete Kommentar, enthält die eigentliche Aufgabe, die das Modell lösen soll. Auch der Kommentar ist vom Benchmark vorgegeben. Die letzte Zeile enthält den Methodennamen als Codevorgabe, der zwingend verwendet werden muss. Der Methodenname ist erforderlich, um nach dem Generieren der Antwort einen Test durchführen zu können. Die Listings 3.1 zeigt ein Beispiel für die beiden Bereiche der ersten Probe des Benchmarks. In dem Listing ist auch zu erkennen das es sich bei den Proben, um Few-Shot-Prompting handelt. Die Aufgaben werden mit mehreren Beispiellösungen angegeben.

```

1 <?php
2 /**
4  * Sie sind ein erfahrener PHP-Programmierer und hier ist Ihre Aufgabe.
  * Sie erhalten eine Liste von Einzahlungs- und Abhebungsvorgängen auf
    einem Bankkonto, das mit einem Nullsaldo beginnt. Ihre Aufgabe
    besteht darin, festzustellen, ob zu irgendeinem Zeitpunkt das
    Guthaben des Kontos unter Null fällt, und an diesem Punkt sollte die
    Funktion True zurückgeben. Andernfalls sollte sie False zurückgeben.
  * >>> below_zero([1, 2, 3])
  * False
  * >>> below_zero([1, 2, -4, 5])
9  * True
  *
  */
  function belowZero($operations){

```

Listing 3.1: PHP Beispiele für die erste Probe

Die beiden Teilbereiche *Opening Tag* und *Kommentar* werden an das Modell als Eingabeaufforderung übergeben. Es generiert den Programmcode und liefert diesen als *Ergebnis* zurück. Die Modelle sollten im Idealfall den vorgegebenen Methodennamen aufgreifen und diesen in ihrer Ausgabe verwendet. Somit findet sich der Methodenname auch im Teilbereich 3 wieder. Ebenfalls werden oft das Opening Tag und der Kommentar durch das Modell übernommen. In der Abbildung 3.1 ist das Ergebnis mit 3 ausgezeichnet. Der Bereich 4 bildet den letzten Teil des Benchmarks. Der Test ist ebenfalls vom HumanEval-XL Benchmark vorgegeben. Hier wird ein einfacher Vergleich durchgeführt der bei Abweichung von der Vorgabe eine Exception auslöst. Dazu werden an die vordefinierte Methode Parameter übergeben, aus denen der generierte Code eine Lösung errechnet. Diese wird mit einer, durch den Benchmark vordefinierten Lösung abgeglichen. Das Listing 3.2 zeigt den Aufbau des Tests, am Beispiel der ersten Probe des Benchmarks. Hier sind in Zeile fünf und elf die erwarteten Methodennamen und die zu übergebenen Parameter zusehen. Ebenfalls sind die Vergleichswerte in Zeile vier und zehn angegeben und die einfache Methode `compare` in Zeile eins bis drei, welche

```
function compare($x, $y) {  
    return $x == $y;  
3 }  
$arg00 = [];  
$x0 = belowZero($arg00);  
$v0 = false;  
8 if (!compare($x0, $v0)) {  
    throw new Exception("Error at 1th assert statement.");  
}  
$arg10 = [1, 2, -3, 1, 2, -3];  
$x1 = belowZero($arg10);  
$v1 = false;  
13 if (!compare($x1, $v1)) {  
    throw new Exception("Error at 2th assert statement.");  
}  
// more tests
```

Listing 3.2: PHP Beispiele für den Test der ersten Probe

einen statischen Vergleich durchführt. Der hier gezeigte Test der ersten Probe ist in einer gekürzten Fassung dargestellt. Der vollständige Code zur Probe kann unter https://github.com/willi-pahl/master-thesis/blob/main/data/problems/PHP_German.jsonl entnommen werden.

3.3.2 Durchführung der Evaluierung

Nach der Generierung des Codes durch das Modell werden die Bereiche 3 (Ergebnis) und 4 (Test) für die Evaluierung der Probe verwendet. Hierfür werden die Bereiche zusammengeführt und ergeben einen ausführbaren und testbaren Code. Die Ergebnisse werden dokumentiert und dienen für die Berechnung der Wahrscheinlichkeit, mit $\text{pass}@k$, dass ein Modell unter k -Proben eine korrekte Antwort liefert. Der Ablauf der Evaluation ist in Abbildung 3.2 dargestellt.

Um die Modelle zu evaluieren und ihre Fähigkeiten hinsichtlich Code für Webanwendungsentwicklung zu generieren, werden die Tests in der Programmiersprache PHP und in der deutschen Version vorgenommen. Dafür sollen die Modelle mehrfach einfache Funktionen generieren. Alle Modelle werden im ersten Schritt mit denselben Parametern getestet, die bereits in der Tabelle 3.2 dokumentiert wurden.

Der Benchmark liefert für jede Probe einen Test für die Evaluierung mit. Dafür werden in den Prompts die Namen der Methoden und die zu übergebenen Parameter angegeben, welche durch die Modelle zu erstellen sind. Der jeweilige Test verwendet dann diesen Namen und übergibt die geforderten Parameter. Von jeder Probe werden pro Modell fünf Varianten erstellt. Die Ergebnisse werden in einer Liste chronologisch gespeichert. Mithilfe der $\text{pass}@k$ Methode erfolgt die Evaluierung der Ergebnisse und anschließend die Bewertung der Modelle.

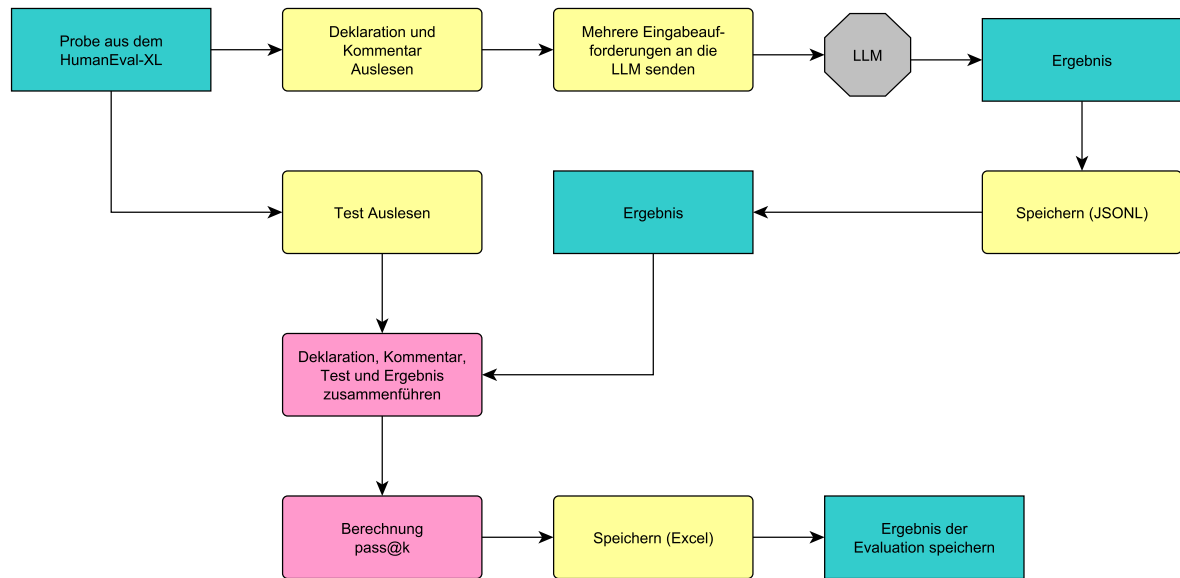


Abbildung 3.2: Ablauf der Evaluierung mit dem HumanEval-XL Benchmarks

3.3.3 Dokumentation der Daten

Die verwendeten Dateien des HumanEval-XL Benchmarks sind im *JSONL*-Format vorgegeben. Ebenso werden die von der LLM generierten Codes in diesem Format dokumentiert und persistent gespeichert. Für jede Probe wird eine separate Datei erstellt, die jeweils fünf Antworten von einer LLM enthält. Für den gesamten Benchmark sind, nach dem Test pro LLM, 80 *JSONL*-Dateien vorhanden.

Neben den Rohdaten werden die Ergebnisse in verschiedenen *Open Document Spreadsheet* (ODS) Dokumente übernommen und aufbereitet. Die Dateien lassen sich mit Tabellenkalkulationstools von OpenOffice oder LibreOffice öffnen. Diese dienen für die Zusammenfassungen der Ergebnisse und zusätzlich werden hieraus Grafiken generiert, welche auch in diesem Dokument Verwendung finden.

Als Letztes werden die programmierten Klassen und Methoden in Python-Dateien (*.py) abgelegt, mit denen die Ergebnisse erstellt und evaluiert werden. Somit lassen sich alle erstellten Ergebnisse nachvollziehen.

3.4 Konzeption der Optimierung

Die Prompts aus dem HunamEval-XL Benchmark sind, wie bereits erwähnt als *Few-Shot-Prompts* verfasst. Einige der Proben besitzen nur ein Beispiel und sind somit als *One-Shot-Prompts* definiert, was in Listing 3.3 beispielhaft dargestellt wird. Hier ist die dritte Probe des Benchmarks zu sehen. Vor den Beispielen ist die eigentliche Aufgabe, welche von der LLM gelöst werden muss. Diese ist in Zeile drei bis sechs zu sehen. In der Zeile sieben ist der beispielhafte Methodenaufwurf für die LLM dargestellt

und das erwartete Ergebnis folgt in Zeile acht. Die Zeile elf zeigt in diesem Beispiel den vordefinierten Methodennamen und die zu übergebenen Parameter für die Methode.

```
<?php
/**
4  * Sie sind ein erfahrener PHP-Programmierer und hier ist Ihre Aufgabe.
  * Die Eingabe sind zwei Zeichenketten a und b, die nur aus 1en und 0en
    bestehen.
  * Führen Sie eine binäre XOR-Operation auf diesen Eingaben aus und
    geben Sie das Ergebnis ebenfalls als Zeichenkette zurück.
  * >>> string_xor('010', '110')
  * '100'
9  *
  */
function stringXor($a, $b){
```

Listing 3.3: PHP Beispiele für den Test der dritten Probe, One-Shot-Prompt

Anders als bei *Zero-Shot-Prompts*, können LLMS durch die Angabe von Beispielen, die Regeln besser erlernen, Muster und Konzept der Aufgabe ableiten, helfen der LLM beim Erkennen des erwarteten Formates und reduzieren Halluzinationen bei den Antworten. Somit wird eine Verbesserung des generierten Programmcodes erreicht. Durch das Anwenden dieses Prompt-Design wurden die Eingabeaufforderungen bereits durch die Autoren des HumanEval-XL optimiert bereitgestellt.

Neben der Optimierung des Prompt Designs wurden weitere Optimierungen in den Eingabeaufforderungen des HumanEval-XL angewandt. So werden die Eingabeaufforderung bereits als Kommentare der jeweiligen Programmiersprache verfasst. Bei den Proben für die Programmiersprache PHP wird vor dem Kommentar, dass für PHP Programme erforderliche *Opening Tag* gesetzt. Somit wird ein weiterer Kontextparameter definiert und liefert für die LLM einen weiteren Anhaltspunkt für die zu generierende Programmiersprache.

Eine weitere Optimierung der Eingabeaufforderung ist die bereits angesprochene letzte Zeile. Hier wird der Name für die erwartete Funktion angegeben und es wird die Fähigkeit der Modelle ausgenutzt, Funktionen und Codes zu vervollständigen. Dies ist wichtig, da die Tests im Benchmark auf einen existierenden Funktionsnamen angewiesen sind.

Somit lässt sich feststellen, dass die Eingabeaufforderungen weitestgehend optimiert sind. Daher wird auf eine weitere manuelle Optimierung der Eingabeaufforderung verzichtet. Es gibt viele Arbeiten, die sich mit genau diesem Problem befassen und es sind durchaus weitere Möglichkeiten vorhanden, die Eingabeaufforderungen textuell zu optimieren.

3.4.1 Optimierung durch Frameworkauswahl

Eine andere Methode zur Optimierung der Eingabeaufforderungen ist die Verwendung verschiedener Frameworks. Hierbei werden die Eingabeaufforderungen durch das Framework angepasst, beispielsweise durch das Hinzufügen zusätzlichen Meta-Angaben. Es soll untersucht werden wie sich die Verwendung unterschiedlicher Frameworks auf die Ausgaben der Modelle auswirkt. In der Arbeit werden die Auswirkungen auf die Ergebnisse, der Python-Frameworks *langchain* und *DSPy*, für verschiedene Modelle verglichen. Für eine automatisierte Auswertung der Tests wird hier ebenfalls der HumanEval-XL angewandt.

Das Framework *langchain* wurde 2022 vorgestellt und ermöglicht Nutzern die Entwicklung komplexer Anwendungen für Interaktionen mit LLMs. Jedoch ist Fachwissen im Bereich Prompt-Engineering erforderlich, um optimale Eingabeaufforderungen an die Modelle zu stellen. Dieses Fachwissen ist beim *DSPy*-Framework nicht mehr notwendig. Hier übernimmt das Framework die Optimierung der Eingabeaufforderungen für die Modelle und macht somit das manuelle Optimieren der Prompts und Kenntnisse in diesen Techniken überflüssig. *DSPy* ist im Oktober von den Forschern der *Stanford NLP Group* veröffentlicht worden, die an Standards für *Natural Language Processing* arbeiten.

3.4.2 Evaluierung der Optimierungen

Wie auch schon bei der Evaluierung der Modelle, werden die Ergebnisse der Optimierung ebenfalls festgehalten, um diese nachvollziehen zu können. Die erhobenen Daten werden in Dateien, die im *JSONL*-Format vorliegen gespeichert. Für die Auswertung der Ergebnisse ist auch hier das *Open Document Spreadsheet (ODS)* vorgesehen. Hier erfolgt die Zusammenfassung der Ergebnisse und deren grafischen Darstellung.

Die erstellten Methoden und Klasse, für die Erzeugung der Antworten und Ausführen der Evaluierungen, sind in Python erstellt und in Dateien *.py abgelegt, um die Nachvollziehbarkeit der Ergebnisse zu gewährleisten.

3.5 Testumgebung

Die *Open-Source*-Modelle laufen auf einem System, welches mit 8 Kernen (16 Threads) und 32 GB RAM ausgestattet ist. Um zusätzlichen Speicher zu erhalten, wurde eine 100 GB Swap Partition genutzt. Zur weiteren Optimierung wurde eine RTX 3060 Grafikkarte der Firma Nvidia, mit 12 GB VRAM verwendet. Das System ist mit dem lokalen Netzwerk verbunden und stellt den Zugriff der Modelle über das Netzwerk bereit.

Auf dem System ist ein Debian 12 installiert und für die Bereitstellung kommt das freie Framework Ollama zum Einsatz. Zur Evaluierung wurde PHP in der Version 8.2.26 (Build-Stand: 25.11.2024)

angewandt.

Über die Hard- und Softwarekomponenten der kommerziellen *Cloused-Source*-Modelle sind keine Hardwarespezifikationen oder andere tiefer gehende Informationen vorhanden. Da die Performance für die Evaluierung in dieser Arbeit keine entscheidende Rolle spielt, wird nicht weiter recherchiert.

Alle wichtigen Codes und Codesequenzen, sowie die Ergebnisse sind im Dokument oder im Anhang zu finden. Der gesamte Code und die Ergebnisse sind dokumentiert, sodass die Möglichkeit besteht die Evaluierungen nachzuvollziehen. Alle Daten und Dateien werden unter <https://github.com/willipahl/master-thesis> bereitgestellt.

IMPLEMENTIERUNG

In diesem Kapitel wird auf die Umsetzung des Konzeptes und des Designs beschrieben, welches in Kapitel 3 eingeführt wurde. Es wird auf die Umsetzung der Bereitstellung eingegangen, wie die Implementierung erfolgt, um Antworten von dem Modelle zu erhalten und vorgestellt wie die Ergebnisse evaluiert werden.

4.1 Lokale Modelle

Um die lokalen Modelle testen zu können, ist es erforderlich diese auf geeignete Hardware bereitzustellen. Die zur Verfügung stehende Hardware erlaubt die Bereitstellung von Modellen bis etwa 70b, wobei hier eine Modellquantisierung angewandt werden muss, sodass die Speichergröße der Modelle, 25 bis 30 Gigabyte nicht überschreiten sollte.

Ein Tool zur Verwaltung und Ausführung von LLMs, ist Ollama. Ollama bieten eine Reihe von Modellen an, die den zuvor genannten Bedingungen entsprechen. Weitere Vorteile die Ollama bietet, ist eine API und ein optionales Webinterface. Diese werden verwendet, um mit den Modellen zu interagieren. Dies gilt für die Generierung des Codes und für einfache Tests, um die Funktionen von Ollama zu überprüfen.

4.1.1 Bereitstellen er Modelle

Durch die Anbindung an einer API, welche sich beispielsweise mittels Python abfragen lässt, lassen sich Modelle von der Ollama Modell Seite testen. Dazu wird Ollama auf dem Server installiert und konfiguriert, siehe Anhang A.2. Nach dem Download stehen die Modelle zur Verfügung und mittels der integrierten API können Interaktionen erfolgen. Das Listing 4.1 zeigt beispielhaft die erforderlichen und optionalen Parameter, die für eine einfache Interaktion mit einem Ollama-Server notwendig sind. Neben der Angabe des Modells sind nur noch die Server-IP und der Port erforderlich. Alle weiteren Parameter sind optionale Angaben für eventuelle Einschränkungen oder Anpassungen an das Verhalten der Modelle. Wie aus der ersten Zeile hervorgeht, wird der Ollama-Service in Python durch das

langchain-Framework unterstützt und bietet fertige Klassen und Methoden für dessen Verwendung. Ab der Zeile drei wird das Modell konfiguriert, welches auf dem Server angesprochen werden soll.

```
4 from langchain_ollama.llms import OllamaLLM

model = OllamaLLM(
    base_url="192.168.1.56:11434", # required
    model="deepseek-r1:32b", # required
    temperature=0.2, top_p=0.95, num_predict=2048, # optional
)
```

Listing 4.1: Interaktion in Python mit Ollamaserver

Wie bereits angesprochen, bietet Ollama die Möglichkeit ein grafisches Tool zum Testen zu installieren. Mit Open WebUI wird ein Browser basierendes Tool eingesetzt, dass auf dem Ollama-Server aufgesetzt wird. Nach der Installation ist das Tool einsatzbereit und im lokalen Netzwerk, unter `http://«server-ip»:«webui-port»` erreichbar. Die Installation wird im Anhang A.3 beschrieben.

4.1.2 Ergebnisse generieren

Nachdem die Modelle bereitstehen, erfolgt das Generieren der Ergebnisse für jedes einzelne Modell. Mit Prompts, die aus den Proben des HumanEval-XL Benchmark bestehen, werden die Modelle mehrmals hintereinander abgefragt. Die vollständig generierten Antworten der Modelle werden, für eine spätere Auswertung und Nachvollziehbarkeit im *JSONL*-Dateiformat gespeichert. Für jede Probe erfolgen fünf Abfragen an jedes Modell. Welche Modelle für die Generierung verwendet wurden, ist in Kapitel 3.2 nachzulesen. Alle Modelle, die in dieser Arbeit verwendet werden, wurde von Ollama bereitgestellt. Das Listing 4.2 zeigt wie eine einfache Eingabeaufforderung, an ein Modell gesandt wird. Nachdem in Zeile fünf, die Probe eingelesen wurde und das Model in Zeile 7 definiert ist, wird das Template für den Prompt ab Zeile neun erstellt und in Zeile vierzehn mit der Probe aus dem HumanEval-XL befüllt. Wie das Modell, aus Zeile sieben initialisiert wird, wurde in Listing 4.1 bereits gezeigt. Die eigentliche Abfrage des Modells erfolgt in Zeile siebzehn und achtzehn. Hierbei werden die Antworten in die `answers` Liste geschrieben und in Zeile zwanzig an die Methode `write_result` übergeben, in der die Daten, in *JSONL*-Dateien persistent gespeichert werden.

```
3 from langchain_ollama.llms import OllamaLLM
from langchain.prompts import PromptTemplate
from helper import read_sample_by_task, write_result

sample = read_sample_by_task(task_id=task_id)

model: OllamaLLM
```

```

8  template = PromptTemplate(
    input_variables=["prompt"],
    template="{prompt}"
)
13
prompt = template.invoke({"user_prompt": sample.get("prompt")})

answers: list = []
for index in range(0, 5):
18     answers.append(model.invoke(prompt))

write_result(sample=sample.get("task_id"), answers=answers)

```

Listing 4.2: Interaktion in Python mit Ollamaserver

Um die Aufgabeninformation `sample.get("prompt")`, in Zeile fünf, aus den Proben zu lesen, wird hier auf den Aufbau der Proben hingewiesen der in Kapitel 3.3.1 beschrieben wurde. Aus den gespeicherten Daten, lassen sich später nach der `pass@k`-Methode, mit `k={0, ..., 5}`, die Modelle evaluieren.

4.2 Optimierung der Antworten durch Änderung des Frameworks

Wie das `langchain` Framework, basiert `DSPy`¹ ebenfalls auf Python und eignet sich für die Abfrage lokale Ollama Modelle. Somit kann der vorhandene Ollama-Server weiterhin genutzt werden. Das Listing 4.3 zeigt die vorbereitenden Schritte, mit der `DSPy`-Bibliothek, welche erforderlich sind und eine generierte Antwort von einem Modell zu erhalten.

```

import dsp

class BasicProgramming(dsp.Signature):
    question = dsp.InputField(desc="Eine Frage zu PHP Programmierung")
5    answer = dsp.OutputField(desc="Generiere Programmcode")

model: dsp.LM = dsp.LM(
    api_base="http://192.168.1.56:11434", # required
    model="ollama_chat/deepseek-r1:32b", # required
10    api_key="", # required
    temperature=0.2, # optional
)

```

¹Informationen zur Installation der Python-Bibliothek `DSPy` sind unter <https://pypi.org/project/dspy> einsehbar. Weitere Informationen zum Framework sind auf der Internetseite <https://dspy.ai> zu finden.

```
    cache=False, # optional
    cache_in_memory=False, # optional
)
15
dspy.configure(lm=model)
model(messages=[
    {
        "role": "user",
20        "content": "Du bist erfahrener PHP Entwickler",
    }
])

chain = dspy.ChainOfThought(BasicProgramming)
```

Listing 4.3: Konfiguration eines Modells in DSPy

Der Code zeigt den Import, in Zeile eins den DSPy-Bibliothek. Anschließend wird in Zeile drei bis fünf eine Signatur für die Abfrage erstellt. Hier wird die Interaktion des Moduls definiert. Es wird die erwartete Eingabe (`question`) und Ausgabe (`answer`) festgelegt. Die `desc`-Parameter im `InputField` und `OutputField` dienen dem Framework für die Erstellung und Optimierung der Prompts, welche in die LLMs gesandt werden.

Im Anschluss wird das Lokale Modell mit erforderlichen und optionalen Parametern konfiguriert. Die Parameter `api_base`, `model` und `api_key` sind erforderlich und müssen angegeben werden. Im Beispiel wird das R1 Modell von Deepseek auf dem lokalen Ollama-Server angesprochen. Der Wert für `key` bleibt leer, solange der Ollama-Server keine Keys für die Anmeldung verwendet. Der Zusatz `ollama_chat` veranlasst das Modell-Objekt nach einem Ollama-Server unter der, in `api_base` angegebenen IP zu suchen. Mit einer `temperature`-Angabe von 0.2 wird das Modell zu einer deterministischen Antwort geleitet. Hier ist eine hohe Kreativität nicht gewünscht. Werden die optionalen Parameter `cache` und `cache_in_memory` nicht gesetzt, so wird ein Cache eingesetzt, was dazu führt, dass die Abfrage nur einmal in die LLM leitet und die Antwort in einem Cache abgelegt wird. Bei allen weiteren Anfragen würde diese Antwort wiederholt zurückliefert. Das würde bedeuten, dass immer nur der *pass@k* für $k = 1$ erstellt werden könnte. Um dies zu verhindern, müssen beide Parameter den Wert `False` erhalten.

Nachdem das Modell konfiguriert ist, wird in Zeile 16 DSPy mit dem Modell `model` konfiguriert und definiert somit deren Verwendung. Ab der Zeile 17 wird dann die Systemnachricht initialisiert. Diese Systemnachricht wird als erste Nachricht an die LLMs gesendet und legt dadurch den Kontext fest.

Die letzte Zeile des Listings 4.3 zeigt die Erstellung eines `ChainOfThought` Moduls, mit der zuvor erstellten Signatur. Das erst verwendete einfachere Modul `dspy.Predict` könnte nicht genutzt werden, da dies immer wieder zu Laufzeitfehlern führte und die Evaluierung nicht durchgeführt werden konnte.

4.2.1 Ergebnisse generieren

Mit dem fertig konfiguriertem `dspy.ChainOfThought` Modul, können anschließend Interaktionen mit einem Modell auf lokalen Ollama-Server erfolgen. Das Listing 4.4 zeigt das Vorgehen bei der Erstellung von generierten Antworten aus den Modellen.

```

1 import dspy
  from helper import read_sample_by_task, write_result

  sample = read_sample_by_task(task_id=task_id)

6 chain: dspy.ChainOfThought
  answers: list = []

  for index in range(0, 5):
      answers.append(chain(question=sample.get("prompt")).answer)
11 write_result(sample=sample.get("task_id"), answers=answers)

```

Listing 4.4: Interaktion in Python mit Ollama-Server

Nach dem Import der erforderlichen Bibliotheken und dem Laden der Testprobe in Zeile 4 wird das `chain`-Objekt aus Listing 4.3 verwendet. Anschließend werden die Testproben mehrfach an das LLM übermittelt. Es werden wie bei der Bewertung der Modelle jeweils fünf Antworten für jede Probe generiert und diese in JSONL-Dateien gespeichert.

4.3 Benchmark Codeevaluation

Die Analyse der generierten Antworten muss für jedes Modell individuell angepasst werden, da die erzeugten Codefragmente zwischen den Modellen variieren. Insbesondere unterscheiden sich die Formate, in denen die Codesnippets enthalten sind, beispielsweise durch das Setzen ohne `““php` oder mit `““php\n <?php Opening Tag`. Dementsprechend erfordert die Extraktion des relevanten Codes aus den Modellantworten eine flexible Methode, die an das jeweilige Ausgabeformat der Modelle angepasst wird. Ein exemplarischer Lösungsansatz zur Extraktion des generierten Codes ist in Listing 4.5 dargestellt.

```

def get_generatet_code(code=code):
    if len(code.split("““php")) > 1: # find start
3       code = code.split("““php")[1]
        code = code.split("““")[0]

    if code.startswith("<?php"): # find start
        code = code.split("<?php")[1]

```



```
8         code = code.split("?>")[0]

        if len(code.split(r"\n}\n")) > 0: # find end
            code = code.split(r"\n}\n")[0] + "\n}\n"

13    return code
```

Listing 4.5: Codesnippet zur Extrahierung des Codes aus der LLM Antwort

Um den generierten Code zu evaluieren, wird dieser zusammen mit dem Test, der jeweiligen Probe aus dem Benchmark zusammengeführt. Das Ergebnis ist ein ausführbarer Code, der die geforderte Methode enthält. Mittels Python wird der Code getestet, ob der dieser ohne Fehler ausführbar ist. Entsteht bei der Ausführung ein Laufzeitfehler erfolgt der Abbruch des Tests. Dieser kann ausgelöst werden durch eine nicht korrekte Syntax, einer Endlosschleife oder wenn die geforderte Methode nicht generiert wurde. All diese Ereignisse führen dazu, das die Probe als nicht bestanden gilt. Das Listing 4.6 zeigt den Ausschnitt im Code zur Ausführung des PHP Interpreters in Python.

```
import subprocess

2
def test_answer(task_id, repetition):
    # read generated code
    answer = read_answer(task_id=task_id, repetition=repetition)
    answer = get_generatet_code(code=answer)

7

    # read test in HumanEval-XL
    test = read_sample(task_id=task_id).get("test")

    try:
12        result = subprocess.run(
            ["php", "-r", f"{test}{answer}"],
            capture_output=True,
            text=True,
            check=False,
17            timeout=5,
        )
    except subprocess.TimeoutExpired:
        return False

22    if result.stderr.strip() == "":
        return True

    return False
```

Listing 4.6: Codesnippet zur Ausführung des PHP Interpreters

Nachdem eine von fünf Antworten des Modells und der Test aus dem Benchmark vorliegen, erfolgt die Prüfung des generierten Codes. Diese wird im Listing 4.6 ab Zeile zwölf gezeigt. Die Funktion liefert `True` zurück, wenn es keine Fehler im Test gab, sonst immer `False`. Das Exception-Handling ab Zeile 19 wird aufgerufen, wenn die PHP Ausführung in einer Schleife hängt. Hier wird ein Timeout abgefangen und somit gilt der Test als nicht bestanden. Aus den erhaltenen generierten Ergebnissen, der Proben eines Tasks, berechnet die `pass@k` Methode, „die Wahrscheinlichkeit das unter `k`-Proben, eine korrekte Probe dabei ist“. Anschließend wird der Durchschnitt für die Zuverlässigkeit des gesamten Modells errechnet.

Umsetzung der `pass@k` Methode

Nach Abschluss der Tests einer Probe werden die Ergebnisse mithilfe der `pass@k`-Methode analysiert. In Python steht hierfür, die Bibliothek `pass_at_k` zur Verfügung. Das Listing 4.7 zeigt die Implementierung der Methode gemäß Gleichung 2.3 in Kapitel 2.6.1, wie sie auch in der Python-Bibliothek verwendet wird.

5

```
def custom_pass_at_k(n: int, c: int, k: int) -> float:
    """
    :param n (int): numbers of total samples.
    :param c (int): number of current samples.
    :param k (int): number of consider samples.
    """
    if n - c < k:
        return 1.0
    return 1.0 - np.prod(1.0 - k / np.arange(n - c + 1, n + 1))
```

Listing 4.7: Berechnung der `pass@k` Metrik in Python

Die Methode erwartet drei Parameter. Der Parameter n bezeichnet die Gesamtanzahl der Abfragen pro Probe. Der Parameter c gibt die Anzahl der korrekten Abfragen innerhalb einer Probe an. Schließlich legt der Parameter k fest, wie viele der **TOP** Lösungen für die Bewertung berücksichtigt werden. Alle drei Parameter sind ganzzahlig (Integer), während das Ergebnis als Gleitkommazahl (Float) zurückgegeben wird.

In dieser Arbeit werden die Werte $k = 1$ und $k = 5$ für die Evaluierung herangezogen. Nachdem jede Probe des HumanEval-XL-Benchmarks einzeln bewertet wurde, wird eine aggregierte Bewertung für das gesamte Modell ermittelt. Diese Berechnung erfolgt gemäß der Gleichung 2.4 aus Kapitel 2.6.1.

In diesem Kapitel werden die Ergebnisse der Benchmark Proben ausgewertet und die Ergebnisse aus der `pass@1` und `pass@5` Methode diskutiert. Weiterhin werden die Resultate in Bezug auf die Auswahl der Landessprache und die Wahl der Parameteranzahl bei den Modellen erläutert. Abschließend wird darauf eingegangen, wie sich die Auswahl eines Frameworks für die Interaktion mit den Modellen auf die Ergebnisse auswirken.

5.1 Modellbewertung

Für die Bewertung wird das Vorgehen gewählt, welches in [16] und [17] beschrieben wird. Die Tests werden exemplarisch, mit der für die Webentwicklung relevanten Sprache PHP durchgeführt und in deutscher Landessprache.

Aus den Ergebnissen der Tests, wird mithilfe der `pass@k`-Methode, die Zuverlässigkeit der jeweiligen Modelle berechnet. Dieser Wert gibt an, mit welcher Wahrscheinlichkeit mindestens eine richtige Lösung unter k ausgewählten Vorschlägen vorhanden ist.

Nachdem die fünf Versuche pro Probe ausgewertet wurden, sind die Ergebnisse in Tabelle 5.1 aufgeführt. Hier wird auf den Werten für $k = 1$ und $k = 5$ eingegangen. In der Tabelle sind alle getesteten Modelle nach Namen sortiert und es wird dadurch noch keine Wertung abgegeben.

Die ersten Tests zur Generierung von Code wurden mit den Closed-Source-Modellen *ChatGPT 4* und *Gemini 1.5* durchgeführt. Hierbei wurden die Tests mit 600 Token bei beiden Modellen durchgeführt. Insbesondere *Gemini 1.5* hat mit dieser Tokenlänge Probleme. Die kurze Tokenlänge führte dazu, dass die Antworten mitten in der Beantwortung der Frage unterbrochen worden. Somit waren die generierten Codefragmente nicht nutzbar. Es wurde kein lauffähiger Code generiert. Das Listing 5.1 zeigt ein unvollständig generiertes Codefragment für die Probe *PHP/23*. Die Zeile sechzehn ist die letzte Zeile, welche vom Modell zurückgesandt wurde und ist zudem unvollständig.

Hier wurde durch *Gemini* oft zuerst die Frage wiederholt, bevor die neu generierten Teile kommen und somit die definierte Anzahl der Token nicht ausreicht, was zu einem Nachteil für dieses Modell wurde.

Model	pass@1	pass@5
ChatGPT 4 API (T600)	0,335	0,6125
Codellama:13b	0,005	0,025
Codellama:70b	0,3325	0,6375
DeepSeek-Coder-V2	0,555	0,65
DeepSeek-R1	0,7025	0,8375
Gemini Flash 1.5 Chatbot	0,2925	0,55
Gemini Flash 1.5 API (T600)	0,06	0,075
Llama3.1:8b (T600)	0,43	0,6375
Llama3.1:8b (T1200)	0,445	0,6375
Llama3.1-claude:8b	0,4625	0,65
Llama3.2:3b	0,2625	0,55
Llama3.3:70b	0,6275	0,725
Mistral-small 22b	0,22	0,4
Qwen 2.5-Coder 32b	0,1975	0,3

Tabelle 5.1: Ergebnisse der pass@1 und pass@5 Methode

```

1  /**
   * Sie sind ein erfahrener PHP-Programmierer und hier ist Ihre Aufgabe.
   * Gib den größten Primfaktor von n zurück. Gehe davon aus, dass n > 1
   *   und keine Primzahl ist.
   * >>> largest_prime_factor(13195)
   * 29
6  * >>> largest_prime_factor(2048)
   * 2
   *
   */
11 function largestPrimeFactor($n){
    $largestFactor = 1;

    // Teile n wiederholt durch 2, bis es ungerade ist
    while ($n % 2 == 0) {
16         $largestFactor = 2;
        $n

```

Listing 5.1: Wortlaut der Aufgabe 23 aus dem HumalEval-XL Benchmark

Die gleichen Fragen wurden anschließend an den ChatBot *Gemini 1.5* gestellt, was zu einem besseren Ergebnis führte. Während die Gemini API mit 600 Token für den pass@1 einen Wert von 6% erreichte waren es bei den Abfragen des Gemini ChatBots 29,3%.

Das Modell *ChatCPT 4* von OpenAI wurde über die API abgefragt. Hierbei lag der Wert für den pass@1 bei 33,5% und der Wert für den pass@5 bei 61,25%. Weitere Tests wurden aufgrund der Kosten mit diesem Modell nicht weitergeführt. Somit ist die Ermittlung der Werte für die pass@k Methode bei einer höheren Tokenlänge nicht mehr erfolgt.

Eines der getesteten Open-Source-Modelle ist das *code1lama* Modell mit 13 Milliarden Parametern. Dieses hat beim Generierung der PHP Lösungen aus dem Proben des Benchmarks am schlechtesten abgeschnitten. Der Grund hierfür war, dass die meisten generierten Codes für die Programmiersprache Python erstellt worden. Das ist der Grund warum die meisten Tests, welche durch den Benchmark vorgegeben waren, ein negatives Ergebnis auswiesen. Das zweite der Codellama-Familie ist das Modell mit 70 Milliarden Parametern. Dieses Modell erreichte beim *pass@1* ein Ergebnis von 33,25% und erreicht mit ChatGPT identische Werte. Während dieses Modell bei der Evaluation des *pass@5* den gleichen Wert 63.75%, wie das Llama3.1 erreicht und somit über dem Ergebnis von ChatGPT 4 liegt.

Des Weiteren wurden eine Reihe von Llama-Modellen getestet, darunter verschiedene Llama3.1 Modelle. Bei dem *Llama3.1 8b* wurden zwei verschiedene Versuche durchgeführt mit jeweils unterschiedlicher Tokenlängen. Anders als bei dem Modell *Gemini 1.5 (API)* können die kleineren Modelle schon mit geringerer Tokenlänge valide Antworten erzeugen. Die unterschiedliche Tokenlängen haben kaum einen Einfluss auf das Ergebnis. Hier wies das Llama 3.1 mit 1200 Token gegenüber dem Llama 3.1 mit 600 Token eine Verbesserung von 1,5% bei der *pass@1* Methode auf. Bei der Auswertung der *pass@5* Werte, war keine Veränderung der Ergebnisse zu beobachten, hier liegt der Wert konstant bei 63,75%.

Um einen Ansatz zur Optimierung der Prompts zu testen, wurde das Modell *Llama3.1-Claude 8b-q4* ebenfalls evaluiert. Bei diesem Modell handelt es sich um ein *Llama 3.1:8b* Model, welches mit den System Prompts des Modells *Claude Sonnet 3.5* von der Firma Anthropic erstellt wurde. Bei der Lösungsgenerierung wurden die Antworten beide Modelle mit 600 Token limitiert. Hier konnte eine Verbesserung gegenüber dem Basismodell um 3,25% für den *pass@1* und um 1,25% für den *pass@5* festgestellt werden. Die Ergebnisse sind in Tabelle 5.1 und in der Abbildung 5.1 zu sehen.

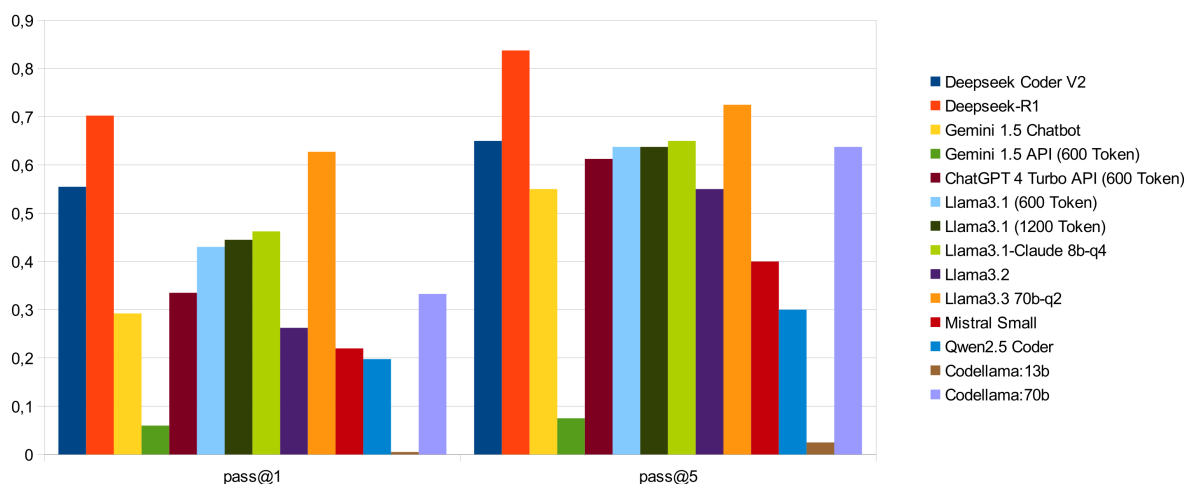


Abbildung 5.1: Ergebnisse der *pass@k*-Methode für die Modelle

Ein weiteres getestetes Modell aus dem Hause *Meta* ist das Modell *LLama3.2:3b*. Es ist mit 3 Milliarden Parameter kleiner als das getestete Vorgängermodell *Llama3.1:8b* und somit das kleinste welches in

dieser Arbeit evaluiert wird. Diesem Modell erzielte bei der Generierung von PHP Code, beim pass@1 eine Wahrscheinlichkeit von 26,25% und beim pass@5 eine Wahrscheinlichkeit von 55%.

Ein letztes Modell aus der Llama3 Reihe, das dem Benchmark unterzogen wurde, ist das *Llama3.3 70b-instruct-q2_K*. Mit einem pass@1 Ergebnis von 62,75% und 72,5% beim pass@5 liefert dieses Modell das drittbeste Ergebnis. Bei dem vorliegenden Modell handelt es sich um ein quantisiertes Modell, was die Größe des Modells auf rund 26 GB reduziert. Bei diesem Modell wurde einige Komponenten einer 2-Bit Quantisierung unterzogen und eine fast gleichbleibende Ausgabenqualität liefern. Weitere Informationen zu diesem Modell sind [32] zu entnehmen. Das Basismodell mit ca. 43 GB ließ sich nicht mit den verfügbaren lokalen Ressourcen testen.

Ebenfalls wurden zwei Open-Source-Modelle von der KI-Entwicklungsfirma Deepseek evaluiert. Das Modell *Deepseek Coder V2 16b* erreichte bei der Auswertung des pass@1 eine Wahrscheinlichkeit von 55,5% und beim pass@5 eine Wahrscheinlichkeit von 65%. Somit liegt es in der Bewertung des Benchmarks für den pass@1 mit 12,5% vor den *Llama3.1* und mit 29,25% vor dem *Llama3.2* Modell und ist somit erzielte das Modell deutlich bessere Werte. Beim pass@5 liegt das Modell mit dem *Llama3.1-Claude*, mit 65% gleich. Identische Werte lieferte auch die Evaluation mit der pass@5 Methode im Vergleich zu den *Llama3.x* Modellen. Das *Deepseek Coder V2* Modell lag von dem *Llama3.1*, mit 1,25% und vor dem *Llama3.2*, mit 10%. Ähnliches ergab die Auswertung der pass@5 Methode. Hier lag *Deepseek Coder V2* 7,5% hinter dem *LLama3.3* Modell.

Das zweite Modell von der KI-Entwicklungsfirma DeepSeek, welches evaluiert wurde, ist das Modell *DeepSeek-R1*. Dieses Modell hat die besten Ergebnisse erzielt, obwohl es mit 32 Milliarden Parameter nicht das größte von den getesteten Modellen ist, erreicht es bei der Evaluierung der pass@1 Methode 70,25% und beim pass@5 83,75%. Somit liegt es beim pass@1 7,5% vor dem *Llama 3.3* und bei der Auswertung des pass@5 sind es 11,5% vor dem *Llama3.3* Modell.

Zusätzlich wurden zwei weitere Modelle aus dem Open-Source-Bereich in die Evaluation einbezogen: *Mistral-Small:22b*, entwickelt von der KI-Entwicklungsfirma MistralAI, sowie *Qwen2.5 Coder:32b* von Qwen-AI. *Mistral Small* hat bei er Evaluation des pass@1 ein Ergebnis von 22% und beim pass@5 ein Ergebnis von 40%. Das Modell *Qwen2.5 Coder* erreicht beim pass@1 ein Ergebnis von 19,75% und beim pass@5 ein Ergebnis von 30%. Somit ist die Wahrscheinlichkeit eine korrekte Antwort unter den TOP Antworten zu generieren etwa nur halb so groß wie bei den getesteten *Llama3.1* Modellen.

5.1.1 Evaluierung der Parametergröße

Eine weitere Untersuchung soll zeigen, ob es einen Zusammenhang gibt, zwischen der Modellparameteranzahl und der *Wahrscheinlichkeit das sich eine korrekte Antwort unter k Probe befindet*. Somit soll die Evaluation aufzeigen, inwieweit die Anzahl der Parameter die Ausgaben beeinflusst und ob die Auswahl eher auf große Modelle fallen sollte. Die Abbildung 5.2 zeigt die getesteten Modelle, in Abhängigkeit ihre Parametergröße und ihren Ergebnissen bei der pass@1 Methode für die deutsche

Sprache. Die Größeneinheit für die Parameterangaben ist in Milliarden und die Angaben der pass@k Methode in Prozent. In dieser Arbeit werden nur die Open-Source-Modelle verglichen, da keine offiziellen Daten zu den Parametern der Closed-Source-Modelle bereitgestellt werden.

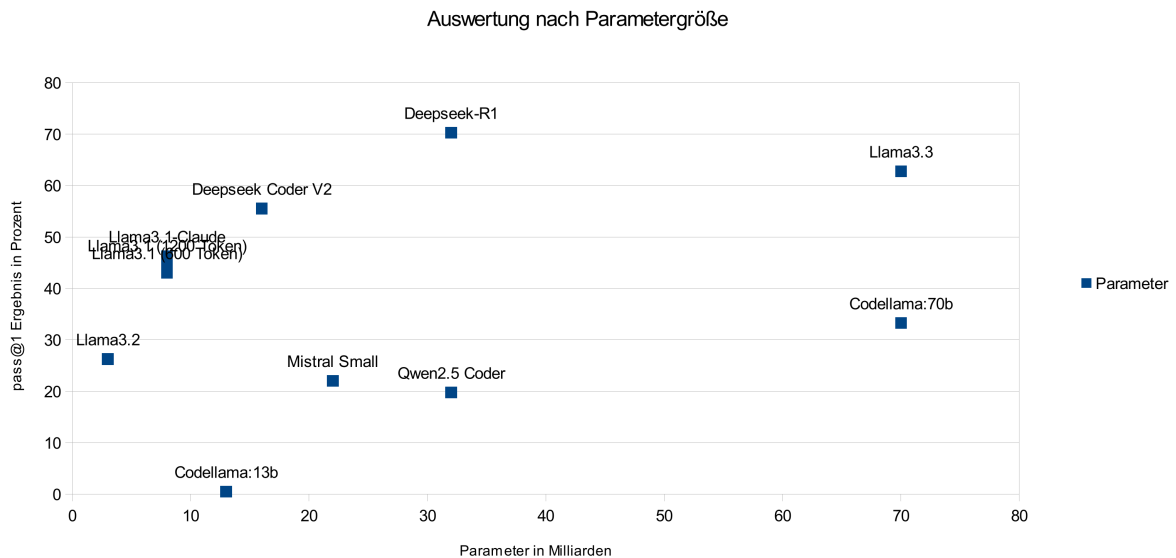


Abbildung 5.2: Ergebnisse der pass@k-Methode für die Modelle

Die hier getesteten Modelle sind eher kleine Modelle und lassen sich in drei Gruppen aufteilen. Die erste Gruppe sind Modelle mit weniger oder genau 10b Parametern, in der zweiten Gruppe werden alle Modelle die größer 10b sind und bis 32b reichen und schließlich die dritte Gruppe umfasst alle Modelle die größer als 32b sind.

Die kleineren *Llama3.1:8b* Modelle übertreffen in der Evaluierung die Modelle der zweiten Gruppe *Mistral Small:22b*, *Qwen2.5 Coder:32b* und *Codellama:13b*. Die *Llama3.1:8b* Modelle liegen in ihrer Gruppe deutlich über dem *Llama3.2:3b* Modell.

Bei den Modellen zwischen 11b und 32b haben die Modelle von Deepseek *Coder V2* und *R1* die besten Ergebnisse. Das *Deepseek-R1* übertrifft sogar die größten hier getesteten Modelle *Llama3.3:70b* und *Codellama:70b*. Während die Modelle *Mistral Small* und *Qwen2.5-Coder* deutlich unter den Ergebnissen der Deepseek-Modelle liegen.

Bei den Modellen mit 70 Milliarden Parameter liegt das *Llama3.3* vor dem *Codellama* Modell. Jedoch liegen beiden getesteten Modelle unter den Ergebnissen des *Deepseek-R1* Modell, welches nur 32 Milliarden Parameter aufweist. Das *Deepseek Coder V2* Modell mit 13 Milliarden Parametern und die *LLama3.1* Modelle mit 8 Milliarden Parametern, erzielten ebenfalls besser Ergebnisse als das *Codellama* Modell.

Somit lässt sich feststellen, dass bei den hier getesteten Modellen kein Zusammenhang zwischen der

Anzahl der Parameter und der Qualität des generierten Codes besteht und die Anzahl der Parameter nicht ausschlaggebend ist, um eine gute Codequalität zu gewährleisten.

Die Ergebnisse aus diesem Test sind in Tabelle 5.1.1 zusammengefasst und zeigen die genauen Werte, für die Ergebnisse des Benchmarks, welche in der Abbildung 5.2 grafisch dargestellt sind.

Model	Parameter	pass@1
Codellama:13b	13	0,005
Codellama:70b	13	0,3325
Qwen2.5 Coder	32	0,1975
Mistral Small	22	0,22
Llama3.2	3	0,2625
Llama3.1 (600 Token)	8	0,43
Llama3.1 (1200 Token)	8	0,445
Llama3.1-Claude	8	0,4625
Deepseek Coder V2	16	0,555
Llama3.3	70	0,6275
Deepseek-R1	32	0,7025

Tabelle 5.2: Ergebnisse der pass@1 Methode mit Angabe der Parameter

5.1.2 Evaluierung nach Sprache

Eine Überprüfung, inwieweit die englische Sprache besser Ergebnisse liefern würde, wurde exemplarisch mit den *Llama*-Modellen 3.1:8b, 3.2:3b und 3.3:70b durchgeführt. Alle drei Modelle repräsentieren unterschiedliche Anzahl der Parameter und somit verschiedene Modellgrößen.

Ein Grund für diese Annahme ist, die in Kapitel 6.2.3 gezeigten Fehler, die bei der Übersetzung der Proben festzustellen war. Es kann auch nicht ausgeschlossen werden, dass die Trainingsdaten für die Modelle Fehler in der Übersetzung enthalten. Diese könnten ebenfalls zu falsch generiertem Code, in einer nicht englischen Sprache führen. Aus den genannten Gründen, lag die Vermutung nahe, dass eine Evaluierung in der englischen Sprache, wesentlich bessere Ergebnisse liefern könnte. Diese Vermutung wurde auch durch die Auswertungen in [17, S. 11] (Tabelle 12), bei denen die Programmiersprache PHP, in der englischen Version im Durchschnitt besser abschnitt bestärkt.

Diese Vermutung konnte mit den durchgeführten Tests der ausgewählten Modelle nur bedingt bestätigt werden. Das *Llama3.3* zeigt beim pass@1, in der englische Version einen besseren Wert. Hier betrug der Unterschied 3,5%, bei denen die Ergebnisse in Deutsch einen Wert von 62,75% erreichten. Die Werte nähert sich mit steigendem k Wert an, sodass bei $k = 5$ die Ergebnisse in Englisch und Deutsch identisch sind und beide Sprachen einen Wert für die Wahrscheinlichkeit von 72,5% erreichten.

Das *Llama3.2* Modelle zeigte beim pass@ k für $k = 1, \dots, 5$ ein umgekehrtes Verhalten, als das zuvor genannte *Llama3.3* Modell. Mit einem steigendem k -Wert erzielte diese Modellvariante immer bessere

Resultate in der deutschen Sprache. Für den $\text{pass}@1$ erzielte das *Llama3.2* Modell für die deutschen Proben einen Wert von 26,25% und bei den englischen Proben einen Wert von 24,75%. Einen Wert von 45% konnte für das englische Modell beim $\text{pass}@5$ festgestellt werden und das deutsche Modelle erreichte einen Wert von 55%. Somit betrug der Unterschied bei den Ergebnissen des Benchmarks, beim $\text{pass}@1$ gerade einmal 1,5%, während es beim $\text{pass}@5$ bereits 10% waren.

Ein ausgeglichenes Resultat zeigt das *Llama3.1* Modell, bei dem Vergleich beider Sprachen. Hier sind die $\text{pass}@3$ Resultate in Deutsch und Englisch fast identisch und der Unterschied liegt hier bei 0,75%. Während bei abnehmenden k -Werten die englischen Resultate besser werden, ist bei zunehmenden k -Wert eine Verbesserung der deutschsprachigen Ergebnisse zu erkennen. Für den $\text{pass}@1$ ist das englische Ergebnis von 48,5% um 5,5% besser als die Resultate mit dem deutschen Proben des Benchmarks. Beim $\text{pass}@5$ lagen die Ergebnisse in Deutsch, mit 63,75% um 2,5% vor den englischen Ergebnissen.

In der Abbildung 5.3 werden die Ergebnisse der $\text{pass}@k$ -Methode mit $k = 1 \dots 5$ für die ausgewählten Llama-Modelle zusammengefasst. Die Proben wurden jeweils in Englisch und in Deutsch an die Modelle gestellt.

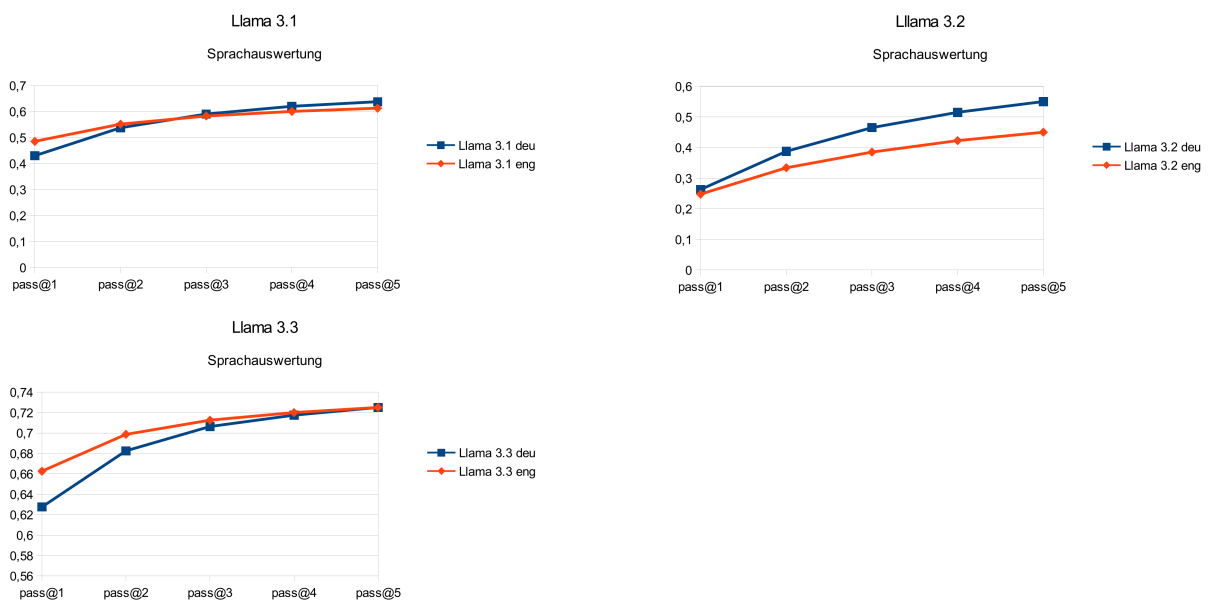


Abbildung 5.3: Ergebnisse der $\text{pass}@k$ -Methode für Llama Modelle in gewählten Sprachen

Die Tabelle 5.1.2 dokumentiert die Resultate des Benchmarks für die *Llama3.x* Modelle. Hier sind alle Werte für die $\text{pass}@k$, mit $k = 1, \dots, 5$ notiert. Die Differenz zwischen der deutschen und englischen Sprachen wurde nach der Formel $DE - ENG = diff$ berechnet. Somit stellt eine positive Differenz, ein besseres Ergebnis für die Proben in Deutsch dar und eine negative Differenz ein besseres Ergebnis für die Proben in Englisch dar.

k	Llama3.1			Llama3.2			Llama3.3		
	DE	ENG	diff	DE	ENG	diff	DE	ENG	diff
1	0,43	0,485	-0,055	0,2625	0,2475	0,015	0,6275	0,6625	-0,035
2	0,5375	0,5512	-0,0137	0,3875	0,3337	0,0538	0,6825	0,6987	-0,0162
3	0,59	0,5825	-0,0075	0,465	0,385	0,08	0,7063	0,7125	-0,0062
4	0,62	0,6	0,02	0,515	0,4225	0,0925	0,7175	0,72	-0,0025
5	0,6375	0,6125	0,025	0,55	0,45	0,1	0,725	0,725	0

Tabelle 5.3: Vergleich der englischen und deutschen Sprache für den pass@1

Dieser Test zeigt, dass eine deutschsprachige Verwendung der Modelle nicht zwangsläufig schlechteren Code generiert.

5.2 Optimierung der Ergebnisse durch die Wahl des Frameworks

Um zu prüfen, ob sich mit verschiedenen Frameworks Optimierungen erzielen lassen, wurden die Modelle *Codellama*, *Llama3.1*, *Mistral Small* und *Qwen 2.5 Coder* herangezogen. Von den vier getesteten Modellen zeigten drei eine signifikante Verbesserung in den Ergebnissen. Nur das im Test verwendete Llama3.1 Modell hat nach der Änderung des Frameworks eine Verschlechterung der Ergebnisse gezeigt. Bei diesem Modell sank das Ergebnis beim pass@1 um 0,095 von 0,43 auf 0,335 und beim pass@5 sank das Ergebnis des Tests von 0,6375 auf 0,6125%.

Bei den getesteten Modellen *Codellama* und *Mistral Small* wurde eine erhebliche Verbesserung festgestellt. Beim *Codellama* Modell erhöhte sich das Ergebnis des Benchmarks beim pass@1 um 0,1425 von 0,005 auf 0,1475 und beim pass@5 um 0,375 von 0,025 auf 0,4. Eine ähnliche Verbesserung zeigt das Modell *Mistral Small*. Hier steigt der Wert des Ergebnisses beim pass@1 um 0,3175 von 0,22 auf 0,5375 und beim pass@5 um 0,325 von 0,4 auf 0,725.

Die besten Werte, nach der Framework Änderung erzielte das Modell *Qwen 2.5 Coder*. Hier ist ein Anstieg der Ergebnisse beim pass@1 um 0,515 und beim pass@5 um 0,525 zu beobachten. Somit erreicht das Modell im Benchmark für den pass@1 ein Ergebnis von 0,7125 und für den pass@5 einen Wert von 0,825. Mit diesem Ergebnis liegt das *Qwen 2.5 Coder* Modell fast gleich auf mit dem neuesten Modell von Deepseek, dem *Deepseek-R1* Modell. Dieses hat im Benchmark, mit dem langchain Framework einen Wert für den pass@1 von 0,7025 und für den pass@5 von 0,8375 erreicht.

Die Ergebnisse des Vergleichs der beiden Frameworks langchain und DSPy werden in den Diagrammen der Abbildung 5.4 dargestellt. Jedes Diagramm zeigt die Ergebnisse eines Modells für den pass@1 und für den pass@5 mit beiden Frameworks.

Die Werte aus der Abbildung 5.4 sind in der Tabelle 5.2 detailliert aufgeführt. Zusätzlich ist der Tabelle

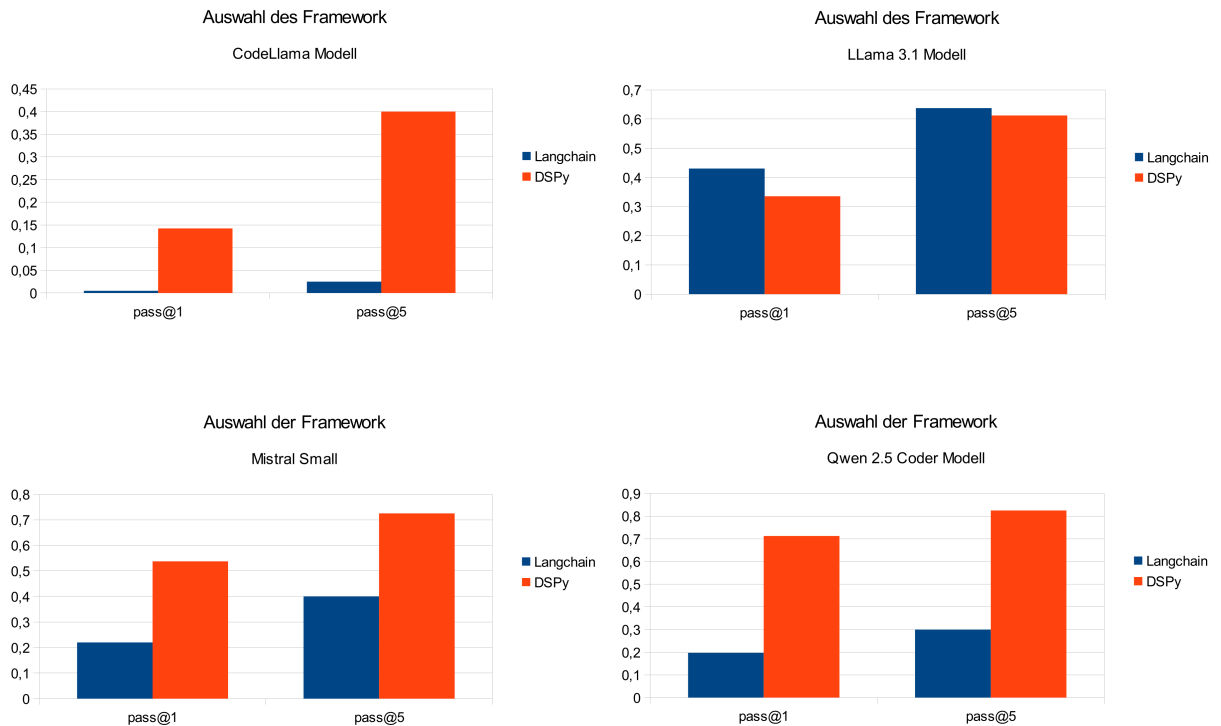


Abbildung 5.4: Ergebnisse der pass@k-Methode für unterschiedliche Frameworks

die Differenz angegeben, welche zwischen den Ergebnissen der Frameworks langchain und DSPy festgehalten wurden. Die Differenz wurde nach der Formel $DSPy - langchain = Diff$ berechnet. Somit zeigt eine negative Differenz an, dass das Framework langchain bessere Werte erzielte, während eine positive Differenz anzeigt, dass das DSPy-Framework die besseren Ergebnisse erzielt hat.

k	Codellama		Llama3.1		Qwen 2.5 Codier		Mistral Small	
	1	5	1	5	1	5	1	5
Langchain	0,005	0,025	0,43	0,6375	0,1975	0,3	0,22	0,4
DSPy	0,1425	0,4	0,335	0,6125	0,7125	0,825	0,5375	0,725
Diff	0,1375	0,375	-0,095	-0,025	0,515	0,525	0,3175	0,325

Tabelle 5.4: Ergebnisse der pass@1 und pass@5 Methode verschiedener Frameworks

Dieser Test zeigt, dass die Wahl des Framework für die Interaktion mit den Modellen, durchaus eine Verbesserung der Ergebnisse bewirken kann und durchaus in Erwägung gezogen werden sollte.

LESSONS LEARNED

In diesem Kapitel werden die Arbeitsprozesse der Evaluation und Optimierung reflektiert. Des Weiteren geht es darum, Vorschläge zu unterbreiten, die den Prozess unterstützen, dass optimale Sprachmodell auszuwählen.

Zudem werden die größten Hindernisse und Probleme besprochen, die während des gesamten Prozesses der Evaluation auftraten. Diese beinhalten die Bereitstellung der Modelle, das Erheben der Daten von den Proben des Benchmarks und deren Auswertung. Um in folgenden Arbeiten diese Fehler zu verhindern werden dazu Lösungsansätze und Vorschläge diskutiert.

6.1 Evaluierungsaufbau und Vorbereitung

Um die Evaluierungen an Open-Source-Modellen durchführen zu können, mussten diese lokal bereitgestellt werden. Die Wahl fiel auf das Ollama-Framework, da die Installation und Konfiguration sehr gut durch die Entwicklungsfirma und verschiedene Foren unterstützt wird. Neben der vorhandenen API lässt sich das Tool *Open-WebUI* einfach in das Ollama-Framework integrieren. Dieses Tool bietet eine gute UI die von allen Clients mit dem Browser im Netzwerk aufgerufen werden kann. Ebenfalls ein großer Vorteil sind die vielen Modelle welche für Ollama zum Download bereitstehen. Darunter sind Modelle von KI-Entwicklungsfirmen, wie Mistral, Llama, Deepseek und Qwen-Coder.

Eine weitere Möglichkeit ist, die Modelle lokal auszuführen ohne ein Framework einzusetzen. Hierbei können die Abfragen nur auf dem lokalen System erfolgen, wenn keine eigene API Schnittstelle erstellt wird. Des Weiteren fehlte auch eine Web-UI Lösung, sodass diese Möglichkeit nicht weiter in Betracht gezogen wurde.

Für die Evaluierung der kommerziellen Closed-Source-Modelle ist dieser Schritt nicht erforderlich. Hier wird lediglich ein Zugang, in Form eines API-Schlüssels benötigt.

6.2 Evaluierung der großen Sprachmodelle

Ein großes Problem stellte der Zugriff auf die Cloused-Source-Modelle dar. Durch die beschränkten Bezahlmethoden konnte ein permanenter Zugriff auf die Modelle nicht erfolgen. Aus diesem Grund wurden hauptsächlich Open-Source-Modelle lokal evaluiert und getestet. Bei der Optimierung sind ausschließlich Open-Source-Modelle zu Einsatz gekommen.

6.2.1 Lokale Ressourcen

Eines der größten Probleme für das lokale Betreiben von großen Sprachmodellen sind die Hardwareanforderungen. Hier spielen neben der Prozessoranzahl, der Speicherplatz auf der Festplatte, der RAM und der VRAM der Grafikkarte eine entscheidende Rolle. Während der Arbeit wurde eine SSD-Festplatte mit höherer Kapazität eingesetzt und die Grafikkarte, durch eine andere mit größeren VRAM ersetzt.

Die größere SSD-Festplatte wurde notwendig, während des Ladens und Speichern der Modelle mithilfe des Ollama Framework. Speziell für das Speichern der Modelle, war ein RAM notwendig, der doppelt so groß sein musste, wie das Modell selbst. Um diesen RAM bereit zustellen wurde eine SWAP Partition von 100 GB auf der SSD eingerichtet. Dies stellte eine kosten günstigere Variante dar, als der Erwerb von echtem RAM. Dieser emulierte RAM wurde auch für die Ausführung größerer Modelle benötigt, bei der Verwendung von der Grafikkarte mit 4 GB VRAM.

Eine weitere und signifikante Verbesserung brachte der Austausch der Grafikkarte. Hierbei wurde die vorhandene Nvidia GTX 1050 TI mit 4 GB VRAM und einer Bandbreite von 112 GB/s durch eine Nvidia RTX 3060 mit 12 GB VRAM und einer Bandbreite von 360 GB/s ersetzt. Durch den Austausch der Grafikkarte wurde die eingerichtete SWAP Partition nicht mehr benötigt und die RAM Auslastung ist auch bei den größeren Modellen stark gesunken.

Durch diese Anpassungen der Hardware bestand nun die Möglichkeit größere Modelle zuladen und bereit zustellen. Des Weiteren konnte eine wesentliche Verbesserung der Antwortzeit festgestellt werden. So wurde bei dem *Deekseek-Coder-V2* Modell eine Verbesserung der Berechnungszeit für den Benchmark von etwa 24 Stunden auf circa eine Stunde beobachtet. Da der Fokus dieser Arbeit nicht auf eine Optimierung der Hardware liegt, wird auf eine genaue Messung der Antwortzeiten verzichtet.

Dennoch war die Berechnungszeit für die Generierung der Codes bei einigen Modellen, die mehr als 12 GB groß waren sehr hoch. Sodass die Wartezeit, das Auswerten der Evaluierung verzögerte.

6.2.2 Auswertung des Benchmarks

Die Anwendung des vorgeschlagenen Parametersatzes zeigte bemerkenswerte unerwünschte Auswirkungen auf die Erzeugung der Antworten. Insbesondere bei einer Tokenlänge von 600 traten Störungen auf, bei denen einige Modelle unbrauchbaren Code generierten. Dieser Umstand, resultierte beim

DeepSeek-R1 aus einem '<think>'-'Abschnitt innerhalb der Antwortstruktur, in dem das Modell eine detaillierte Herleitung seines Denkprozesses anbietet, bevor es zur eigentlichen Lösung gelangte. Bei dem Modell *Gemini 1.5* führten ausführliche Erklärungen am Anfang und Ende der Antwort zum selben Effekt, sodass diese Ergebnisse ebenfalls nicht brauchbar waren. Aus diesem Grund wurde bei den Modellen *DeepSeek-R1* und *Gemini 1.5* auf eine Einschränkung der Tokenlänge verzichtet.

Die hohen Erwartungen bei den Resultaten der kommerziellen Closed-Source-Modelle wurden nicht erfüllt. Die Modelle, *Gemini 1.5* mit einem pass@1 Ergebnis von 29,25% und *ChatGPT 4* mit 33,5% liegen somit im unteren Bereich der Auswertung und werden von vielen Open-Source-Modellen übertroffen. Auf eine erneute Evaluierung des OpenAI Modells musste aus Kostengründen verzichtet werden. Zu einem ähnlichen Ergebnis kommt die Fallstudie, welche in [33] vorgestellt wird. Bewertet wurde hier der generierte Code von ChatGPT 4 und dessen Nutzen für die Gestaltung einer barrierefreien Webseite. Die Studie kommt zu dem Schluss, dass ChatGPT für den Entwickler ein nützliches Tool darstellt, aber das Urteilsvermögen und Fachwissen des Entwicklers nicht ersetzen kann [vgl. 33, S. 10].

Bei der Auswertung des Benchmarks traten einige Fehler auf, die weitestgehend beseitigt werden konnten. Ein Großteil waren kleinere Fehler, die bei der Auswertung aufgetreten, die in Kapitel 6.2.3 noch ausführlicher besprochen werden. Der gravierendste Fehler trat aber bei der Berechnung des pass@k für das gesamte Modell auf. Hier wurde eine falsche Python-Methode implementiert, sodass die Ergebnisse, um ein Vielfaches niedriger waren. Erst durch den Vergleich mit Ergebnissen anderer Arbeiten und den Herstellerangaben ist der Fehler aufgefallen. Nach intensiver Suche wurde dieser schlussendlich gefunden und beseitigt.

Bewehrt hat sich der Einsatz von Python und dessen Bibliotheken für Umsetzung der Evaluierungs- und Optimierungsaufgaben. Mittlerweile existieren für die meisten Probleme und Anforderungen, fertige Bibliotheken. Oft von den Herstellern der Modelle selbst. Basierend auf den vorhandenen Bibliotheken, konnte die Entwicklung der Evaluationsaufgaben in kurzer Zeit umgesetzt und implementiert werden.

6.2.3 Nachteile der Evaluierung

Trotz seines häufigen Einsatzes bei der Evaluierung von Modellen, in verschiedenen wissenschaftlichen Arbeiten, zeigt der Benchmark-Test für die deutschen PHP Proben Fehler. Aber auch bei der Erstellung der Auswertung mit Python traten Fehler auf. Dadurch entstehen Nachteile für die Modelle, bei der Bewertung mit dem HumanEval-XL Benchmark. Im Verlauf dieses Kapitels werden einige Nachteile diskutiert.

Bei der Probe *php/5* wurde eine **nicht eindeutige Übersetzung** festgestellt. So wurde die Probe übersetzt, nicht aber die Tests. Das Listing 6.1 zeigt die gültigen Werte welche als Eingabeparameter für die zu generierende Funktion erlaubt sind. Hier handelt es sich um deutsche Zahlworte von *null* bis *neun*. Im angegebenen Beispielttest sind Eingabe- und Ausgabeparameter in englischer Sprache. Es

wird auch nicht explizit angegeben, das eine Übersetzung erfolgen soll.

```
/**
 * Sie sind ein erfahrener PHP-Programmierer und hier ist Ihre Aufgabe.
 * Die Eingabe ist ein durch Leerzeichen getrennter String von Ziffern
   von 'null' bis 'neun'.
4  * Gültige Optionen sind 'null', 'eins', 'zwei', 'drei', 'vier', 'fünf',
   'sechs', 'sieben', 'acht' und 'neun'.
 * Gib den String mit den Zahlen sortiert von klein nach groß zurück.
 * >>> sort_numbers('three one five')
 * 'one three five'
 *
9  */
function sortNumbers($numbers){
```

Listing 6.1: Aufgabenstellung der Probe php/5

Die Probe *php/5*, welcher in Listing 6.2 zu sehen ist, stellt einen Test zur Verfügung der die Prüfung mit englischem Zahlwörtern von *zero* bis *nine* ausführt. Keines der getesteten Modelle hat diese Probe bestanden. Alle getesteten Modelle haben Codes generiert, die deutsche Zahlwörter als Parameter erwarten um diese zu sortieren. Die englischen Zahlwörter werden durch die Methoden nicht als Zahlwörter erkannt. Dadurch hat kein Modell diesen Test bestanden.

```
// more tests.

$arg40 = "six five four three two one zero";
$x4 = sortNumbers($arg40);
5 $v4 = "zero one two three four five six";
if (!compare($x4, $v4)) {
    throw new Exception("Error at 5th assert statement.");
}
// more tests
```

Listing 6.2: Aufgabenstellung der Probe php/5

Eine mögliche Lösung ist es, die Anpassung des Benchmarks vorzunehmen und alle Übersetzungen anzupassen oder zu korrigieren. Mit der originalen Benchmark in Deutsch und mit PHP, hat beispielsweise das Modell *Deepseek Coder V2* keine korrekte Lösung generieren können. Wird der Test im Benchmark angepasst und die Zahlwörter ebenfalls übersetzt, so wie der in Listing 6.3 gezeigte Codeabschnitt, wird die Probe von dem Modell mit drei korrekten Antworten bestanden. Derselbe Test in der englischen Sprache wurde beispielsweise vom Modell *Llama3.3* in allen fünf Antworten, mit bestanden bewertet. Da in der englischen Probe alle Zahlwörter in Englisch aufgeführt sind, ist anzunehmen das bei dieser Probe eine Übersetzung nicht das Ziel ist.

```

1 // more tests.

$arg40 = "sechs fünf vier drei zwei eins null";
$x4 = sortNumbers($arg40);
$v4 = "null eins zwei drei vier fünf sechs";
6 if (!compare($x4, $v4)) {
    throw new Exception("Error at 5th assert statement.");
}
// more tests

```

Listing 6.3: Übersetzte Aufgabenstellung der Probe php/5

Die Ergebnisse vor und nach der Änderung für die fünfte Probe sind im Listing 6.4 für das Modell *Deepseek Coder V2* dargestellt. Nach dem Namen *php/5* wird die Gesamtanzahl der Proben angegeben, gefolgt von den insgesamt bestandenen Durchläufe der Probe. Anschließend werden die Wahrscheinlichkeiten, das eine korrekte Antwort unter den TOP Antworten zu finden ist, angegeben, beginnend mit dem `pass@1`.

```

1 # origin sample no. 5
php/5;5;0;0.0000;0.0000;0.0000;0.0000;0.0000

# translated sample no. 5
php/5;5;3;0.6000;0.9000;1.0000;1.0000;1.0000

```

Listing 6.4: Ergebnisse der Probe php/5 für das *Deepseek Coder V2* Modell

Diese Probe zeigt, nach der Änderung wurden drei von fünf möglichen Antworten mit korrekt bewertet. Die Änderung für die Bewertung der `pass@k` Methode des gesamten Modells ist in Tabelle 6.2.3 dargestellt. Diese Änderung wurde nicht mehr auf alle Modelle ausgeweitet, sodass der Fehler in allen Bewertungen enthalten ist.

k	1	2	3	4	5
ohne Korrektur	0,555	0,6112	0,6312	0,6425	0,65
mit Korrektor	0,565	0,6225	0,6438	0,6550	0,6625

Tabelle 6.1: Ergebnisse für das *Deepseek Coder V2* Modell mit und ohne Probe 5 Korrektur

Mit diesem Problem wird gezeigt, dass bereits ein kleiner Fehler das Ergebnis um mehr als ein Prozent für ein Modell verbessern kann.

Eine andere Fehlerquelle ist das ***Zusammenführen der Antworten der Modelle und den Tests aus dem Benchmark***. Diese Fehlerquelle ist nicht der Benchmark selbst, sondern die Umsetzung der Ausführung und Bewertung. Oft sind die Probleme das Parsen der Sonderzeichen, wie beispielsweise die Verwendung der doppelten Anführungszeichen im Test und den erstellten Antworten. Eine manuelle

Prüfung der Aufgabe 2 hat das Fehlverhalten aufgezeigt. Durch das Implementieren einiger weiteren Codezeilen in Python, wie das Listing 6.5 zeigt, konnten die Bewertung einiger Modelle verbessert werden. So hat sich das Ergebnis für die *pass@1* Methode des Deepseek-Coder-V2 um 7% von 0,53 auf 0,6 verbessert. Ebenso wurde für das Llama3.1 Modell eine Verbesserung der Bewertung um 0,025, von 0,45 auf 0,475 ermittelt.

```
answer = answer.replace(r"\n", "\n")
+ answer = answer.replace(r'\\"', '\"')
+
+ test = test.replace(r'\\"', '\"')
```

Listing 6.5: Fehler bei der Auswertung durch fehlerhafte Anführungszeichen

Ein weiteres Problem, welches sich in den Tests gezeigt hat, sind die von einigen Modellen **geänderten Methodennamen**. Einige Modelle ändern diesen ab und passen ihn an die Funktionalität der Methode an. Dieses Verhalten ist beispielsweise beim Google Modell *Gemini 1.5* mehrfach aufgetreten. Ein Beispiel ist die Probe 26 aus dem Benchmark. Hierbei sollte das n -te Element der Fibonacci-Folge berechnet werden. Der geforderte Name der Methode war als *fibfib* definiert. Das Modell schlug in einem Durchlauf den Namen *fibfib_iterativ* vor, da über ein beliebig großes n iteriert wird. Somit ist der Test für diese Probe fehlgeschlagen, obwohl die Methode die Berechnung korrekt durchführte. Dies wurde durch händische Änderungen am PHP Code überprüft und alle generierten Codes dieser Probe wurden bestanden. Das Listing 6.6 zeigt diese durch *Gemini 1.5* generierte Funktion.

```
1 function fibfib_iterativ($n) {
    if ($n < 0) {
        throw new InvalidArgumentException("n muss nicht-negativ sein");
    }

    6 if ($n <= 2) {
        return $n - 1;
    }

    $a = 0;
    11 $b = 0;
    $c = 1;
    for ($i = 3; $i <= $n; $i++) {
        $temp = $a + $b + $c;
        $a = $b;
        16 $b = $c;
        $c = $temp;
    }
    return $c;
}
```

```
}
```

Listing 6.6: Generierte Funktion mit falschen Namen

Ein Entwickler würde den Methodennamen anpassen und die Methode einsetzen. Eine Möglichkeit dies zu umgehen, ist es, den Methodennamen zur Laufzeit zu ermitteln. Mit den Proben sollen die Fähigkeiten der LLMs geprüft werden, inwieweit diese, grundlegende Funktionen implementieren können. Hierbei ist der Name der Methode nicht entscheidend. Anders würde es sich verhalten, wenn es sich dabei um Erweiterungen bestehender Programme handeln würde, in denen die erstellten Funktionen integriert werden soll.

Die Tests aus dem Benchmark und Ergebnisse der Modelle wurde stichprobenartig manuell geprüft und weitere Fehler behoben. Dennoch ist nicht auszuschließen das die jetzige Evaluation weitere Fehlerquellen enthält.

Es gibt durchaus weitere Fehlerquellen, welche die Ergebnisse negativ beeinflussen können. Beispielsweise könnten bei der Formulierung der Tests, Randbedingungen nicht korrekt betrachtet wurden und die Modelle die Aufgabe falsch interpretieren und so die Aufgabe falsch lösen. Oder Tests enthalten gänzlich falsche Parameter und richtig generierter Code wird trotzdem als Falsch gewertet. Die eben genannten Fehler oder ähnliche, konnten im vorliegenden Benchmark nicht nachgewiesen werden, ganz auszuschließen sind sie aber nicht.

6.3 Optimierung der Abfragen

Die Tests mit den ausgewählten Modellvarianten des *LLama3.x* Modells, dieser Arbeit haben gezeigt, dass die Verwendung der englischen Sprache nicht bei allen Modellen eine Verbesserung der Ergebnisse liefert. Daraus wird der Schluss gezogen, das bei der Optimierung der Prompts, der gewählten Modelle, eine Übersetzung ins Englische nur bedingt zu einer Verbesserung führt. Dennoch kann es bei verschiedenen Modellen einen Vorteil bringen. Aus diesem Grund sollten Modelle mit verschiedenen Sprachen ausgiebig getestet werden. Beispielsweise durch die Verwendung verschiedener Benchmarks. So ein Test könnte bei mehrsprachiges Entwicklerteam ein Grund darstellen, welches dann auch die Eingabeaufforderungen in verschiedene Sprachen erstellen, aber nur eine LLM eingesetzt werden soll. Wenn Modelle die bei verschiedenen Sprachen gleich gut abschneiden, kann dies ein weiteres Kriterium darstellen, um die Auswahl der zu nutzenden LLMs einzuschränken oder um eine endgültige Entscheidung zu treffen.

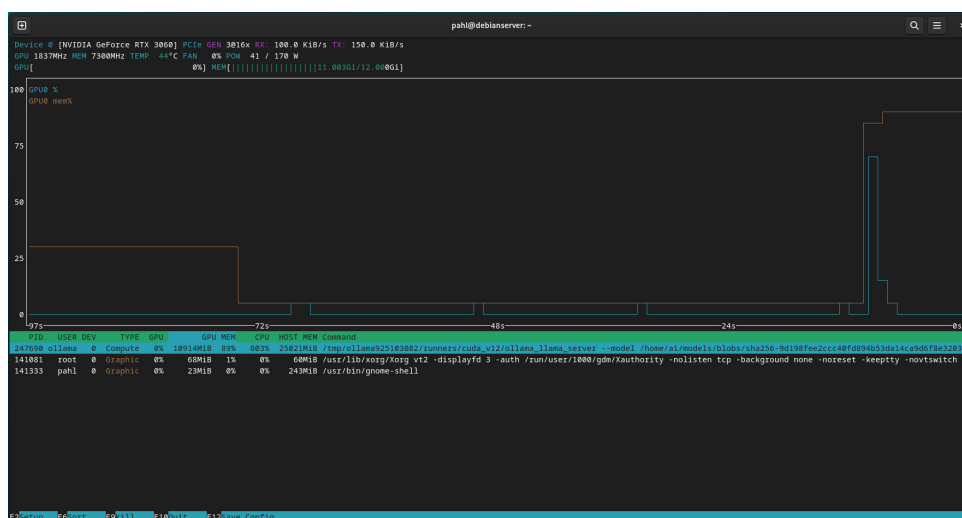
Wie bereits in Kapitel 3.4 besprochen, sind die Prompts im HumanEval-XL Benchmark bereits sehr gut optimiert. Die Optimierungstests mit der Auswahl eines anderen Framework hat jedoch zu einer erheblichen Steigerung der Ergebnisse geführt. Diese signifikante Verbesserung wurde so nicht erwartet. Dieses Ergebnis zeigt, dass die Prompts in des HumanEval-XL Benchmarks weiter optimiert werden

können. Zum anderen zeigt es, dass das DSPy Framework die Prompts selbstständig optimiert, ohne dass Entwickler tiefgehende Kenntnisse im Prompt Engineering haben müssen.

Dennoch zeigt der Test auch, dass diese Optimierung Grenzen hat. So konnte sich beim *Llama 3.1* keine Optimierung erzielt lassen. Hier sind die Ergebnisse schlechter ausgefallen, als mit dem *langchain* Framework. Für die Tests wurde nur eine von vielen Konfigurationen aus dem DSPy Framework getestet. Es bietet verschiedene Möglichkeiten für die Optimierung der Prompts. Hierbei könnten etliche Konfigurationen bestehend aus Signaturen und Modulen getestet werden, die das Framework mitbringt und somit an die eigenen Bedürfnisse anpassbar macht.

In den Auswertungen der Optimierung sind einige Modelle nicht aufgeführt, da das DSPy Framework bei den Anfragen der Proben an die Modelle, mit Fehlern abbricht. Hierbei handelt, ist sich, unter anderem um das Codelama-Modell *codellama:70b-instruct-q2_K*. Abfragen mit dem *langchain* Framework zeigen diese Fehler nicht. Das gleiche Verhalten ist bei weiteren größeren Modellen, wie das *llama3.3:70b* und *deepseek-r1:32b* zu beobachten. Es wurden verschiedene Module, darunter *Predict*, *ProgramOfThought* oder *ChainOfThought* und Signaturen getestet, zum Zeitpunkt der Arbeit wurde keine Lösung für dieses Problem gefunden.

Eine mögliche Ursache, kann darin liegen, dass DSPy einen deklarativen Ansatz implementiert und im Gegensatz zu *langchain*, Abfrageketten erstellt. Hierbei besteht die Möglichkeit, dass unvollständige Ergebnisse, die von den LLMs gesandt werden und nicht so effektiv vom Framework verarbeitet werden, wie es bei *langchain* der Fall ist. Hierbei könnte der Fehler dadurch entstehen, da der zur Verfügung stehen VRAM des Servers auf 12 GB beschränkt ist und somit besteht die Möglichkeit, dass größere Modelle unvollständige Antworten erstellen. Bei allen drei Modellen übersteigt die Modellgröße den verfügbaren VRAM.



Die Abbildung 6.1 zeigt den Modellwechsel vom *Llama3.2* zum *Codellama:70b* Modell. In der Abbildung werden die Grafikkartenparameter mit dem Tool `nvidia-smi` ausgelesen und angezeigt. Während die Auslastung des VRAMs beim *Llama3.2:3b* 3,64 GB beträgt, sind es beim *Codellama:70b* mehr als 11 GB. Das *Llama3.2:3b* hat eine tatsächliche Größe von 2 GB, während das *Codellama:70b* eine tatsächliche Größe von 25 GB hat. Im linken Teil der Abbildung ist das geladene *Llama3.2:3b* Modell zusehen, gefolgt von einem Leerlauf des Speichers zur Vorbereitung auf das neu zu ladende *Codellama:70b* Modell. Die Speicherauslastung dieses Modells ist auf der rechten Seite der Abbildung 6.1 zu erkennen. Ebenfalls wurde eine erhöhte Auslastung der CPUs beobachtet. Während die Auslastung bei der Verwendung des *LLama3.2* Modells einen Load average von 0.12 hervorrief, ist der Wert bei Ausführung des *Codellama* Modells bei 7.53 und mehr gestiegen. Das Problem konnte nicht abschließend aufgeklärt werden.

DISKUSSION UND AUSBLICK

Dieses Kapitel stellt eine Zusammenfassung der wichtigsten Ergebnisse dieser Arbeit auf, die zur Beantwortung der Thesen führen. Es werden zusammenfassend die Grenzen und Einschränkungen geklärt, zu denen die Evaluation durchgeführt wurde. Des Weiteren werden Impulse für weitere Forschungen diskutiert. Das Kapitel endet mit einer praktischen Anwendung, bei der die Ergebnisse dieser Arbeit angewandt werden können.

7.1 Bewertung der Zielsetzung und Thesen

Die Erkenntnisse aus den Experimenten dieser Arbeit bestätigen die aufgestellte dritte These (T3) aus Kapitel 1.4. Eine Optimierung der Eingabeaufforderungen für die Webanwendungsentwicklung lässt sich ohne Änderung der Modellparameter erreichen und bewirken, für einige Modelle eine signifikante Verbesserung der Ergebnisse. Die Modelle wurden bereits mit Programmiersprachen, die für Webanwendungsentwicklung essenziell sind trainiert, wie beispielsweise mit PHP, diese Daten sind in Form von Programmcode und sind von den Modellen abrufbar. Entscheidend hierbei ist die Art und Weise wie die Gestaltung der Eingabeaufforderungen umgesetzt wird. Diese Optimierung erfolgt durch das DSPy Framework für viele Modelle automatisiert.

Dennoch hat das DSPy Framework, auf einige Modelle einen negativen Effekt. Der generierte Code, dieser Modelle schnitt in der Evaluation mit den HumanEval-XL Proben schlechter ab. Der Grund konnte zurzeit nicht geklärt werden. Eine Annahme ist, dass der deklarative Ansatz von DSPy mit den komplexeren Datenstrukturen und Modulketten die LLMs zu inkonsistenten Antworten leitet. In diesen Fällen sollte weitere Frameworks, wie beispielsweise *AdalFlow* oder *LamaIndex* für die Optimierung in Betracht gezogen werden.

Die Tests haben weiterhin gezeigt, dass einige Modelle nicht mit dem DSPy Framework zusammenarbeiten. Auch hier konnte nicht abschließend geklärt werden, ob die Fehlerquelle bei der eingesetzten Hardware zu suchen ist oder ob das Framework selbst Probleme mit den Antworten dieser Modelle hat. Bei Modellen, die dieses Verhalten zeigen, sollte geprüft werden, ob auch hier die Wahl auf ein anderes Framework fallen kann.

Gerade mit der Arbeit von Closed-Source-Modellen, bei der die Modelle gar nicht oder nur sehr bedingt angepasst werden können, sind die Optimierungen der Eingabeaufforderungen essenziell. Hierbei kann ein Framework, wie das DSPy eine hilfreiche Unterstützung bieten.

Alle Evaluierungen wurden mit den HumanEval-XL durchgeführt. Mit den gewonnenen Erkenntnissen aus dieser Arbeit lässt sich die zweite These (T2) nur bedingt bestätigen. Ein einzelner Benchmark hat nicht ausreichend Aussagekraft, um eine LLM hinreichend zu bewerten. Diese Benchmarks eignen sich um einen ersten Eindruck von den Modellen zu erhalten. Hierbei können erste Eindrücke über deren Stärken und Schwächen gesammelt werden. Wie auch in [34] wird die Auffassung vertreten, dass diese Probe grundlegende Codeproblematiken testen, die nicht mit den realen Anforderungen von Entwicklern übereinstimmen. Diese Aussagen deuten darauf hin, dass Programmierer andere Anforderungen an LLMs stellen, die zurzeit mit Benchmarks, nur in engen Grenzen abgedeckt werden können. Zu beachten ist weiterhin, dass die Benchmarks nicht frei von Fehlern sind, die eine verzerrtes Bild der LLM erzeugen können. Entwickler würden die Fehleingaben in den Eingabeaufforderungen berichtigen und einen erneuten Versuch unternehmen, Code generieren zu lassen.

Die menschliche Intelligenz ist in der Lage aus generiertem Code und eigenem Wissen, die Programmteile herauszufiltern welche zur Lösung eines Problems beitragen. Entwickler stellen meist sehr komplexe Anforderungen, um die ihnen gestellten Probleme zu lösen. Diese Komplexität wird durch die Benchmarks nicht abgedeckt.

In den letzten Jahren hat generative KI auch die Arbeit der Entwickler stark beeinflusst. So ist in [35] die Rede von einem Programmierer, der seine Aufgabe abbrach, Zitat: „*weil er keinen Zugang zu seinem virtuellen Assistenten hatte. 'Ich kann einfach nicht mehr ohne die Hilfe der KI programmieren'*“ so der Entwickler. In [36], einem Artikel von Gartner, wird behauptet das bis zum Jahr 2027, 80% der Ingenieure eine Weiterbildung für generative KI benötigen. Dies hebt ebenfalls den Trend der nächsten Jahre hervor, das generative KI die Softwareentwicklung maßgeblich beeinflussen wird. Zurzeit sollten Entwickler sich aber nicht voll auf die Ausgaben der LLMs verlassen. Dies bestätigt auch der Onlineartikel [37], in diesem heißt es, Zitat: „*ChatGPT ... liefert Code, der zwar nicht immer direkt nutzbar ist, nach einer Überarbeitung aber schon recht überzeugend läuft.*“.

Trotzdem wächst die Beliebtheit von KI Anwendungen bei Softwareentwicklern immer weiter und dies gilt auch für den Bereich der Webanwendungsentwicklung. Dabei nutzen viele Entwickler, KI Modellen von US amerikanischen Unternehmen wie Anthropic, Google oder OpenAI die bereits sehr gut entwickelte Chatbots und APIs anbieten, welche für die Integration in Tools zur Codegenerierung eingesetzt werden können. Zurzeit hat Mistral, eine europäische KI-Entwicklungsfirma einen neuen Chatbot, mit dem Name *Le Chat* herausgebracht, der als Konkurrenz zu den amerikanischen Konzernen gesehen wird. Die KI-Entwicklungsfirma *Deepseek* aus China hat ein Modell entwickelt, das wie Mistral, in Konkurrenz zu den amerikanischen Modellen steht. Neben diese kommerziellen Closed-Source-Modelle gibt es eine Reihe von Open-Source-Modellen und einige Entwicklungsfirmen wie Mistral oder Deepseek bieten

neben den kommerziellen, auch einige ihrer Modelle als Open-Source an. Diese Modelle zeigen ebenfalls hervorragende Resultate für die Generierung von Code, wie die Ergebnisse dieser Arbeit beweisen. Um so wichtiger ist es die Stärken und Schwächen der Modelle zu kennen und welche Methoden, zur effektiven Nutzung von Eingabeaufforderungen einzusetzen sind. Mit diesen Kenntnissen wird die, in der Arbeit aufgestellte dritte These (T3) bewiesen. KI kann und wird die Webanwendungsentwicklung maßgeblich beeinflussen.

Gerade die einfache und effiziente Suche mithilfe der LLMs bringt einen zeitlichen Vorteil, dessen Ergebnisse bereits mit einfachen Tests überprüft werden können. LLMs reduzieren die Suche nach geeigneten Frameworks und Technologien und liefern oft funktionsfähigen Code. Dies gilt für die Integration in IDE wie auch die Nutzung eines Chatbots.

7.2 Grenzen und Einschränkungen

Die in dieser Arbeit evaluierten und optimierten Modelle, waren hauptsächlich Open-Source-Modelle. Alle diese Modelle wurden von der Plattform Ollama bereitgestellt. Zwei Ausnahmen gab es bei der Evaluierung, hierbei handelt es sich um *Gemini 1.5* von Google und *ChatGPT 4* von OpenAI.

Für die Evaluierung der Modelle ist nur der *HumanEval-XL* angewandt worden. Die meisten Benchmarks prüfen LLMs mit Python oder Java Proben. Eine Evaluierung mit spezifischen Sprachen für Webanwendungsentwicklung wie PHP, bieten nur wenige Benchmarks, darunter der *HumanEval-XL*.

Die Auswahl der getesteten Frameworks, welche die Kommunikation zu den LLMs und somit zum Ollama-Server herstellten, waren begrenzt auf `langchain` und `DSPy`.

7.3 Impulse für zukünftige Forschungen

Die KI-Entwicklungsfirmen bringen immer neue Modelle auf dem Markt, die schon mit einer sehr kleinen Anzahl von Parametern gute Ergebnisse liefern. Hier könnte eine Studie zeigen, ob *Small Language Models* (SLM) mit den *Large Language Models* (LLM), im Bereich der Webanwendungsentwicklung konkurrieren können. Ein Beispiel für ein SLM ist das Modell *Phi* von Microsoft, welches beispielsweise unter [38] heruntergeladen werden kann. Wird ein kleines Modell für die Codegenerierung eingesetzt und dahin gehend optimiert, ist eine große Parameteranzahl nicht erforderlich. Es existieren bereits einige Studien zu diesem Thema. Beispielsweise wird in [39] eine allgemeine Verbesserung des Benutzererlebnisses untersucht und in [40] wird eine Reduzierung der Kosten in Produktionsumgebungen vorgestellt.

Des Weiteren sollte Unternehmen oder Entwickler weitere Benchmarks in Betracht ziehen und die Brauchbarkeit von LLMs für die Webanwendungsentwicklung evaluieren. Interessant wäre die Prüfung,

ob die Verwendung mehrerer Benchmarks zu einer besseren Bewertung der LLMs führt.

Da die meisten Benchmarks auf statische Analysen ausgelegt sind und die LLMs nur bedingtes oder klein deterministisches Verhalten zeigen, kann eine weitere Forschung dahin gehen, einen Benchmark zu erstellen, der auf die dynamischen Antworten der LLMs besser eingehen könnte. Speziell für die PHP-Entwicklung wäre eine Untersuchung interessant, bei der die Evaluierung mit externer Tool wie *SonarQube*, *PHPBench*, *PHPUnit* oder *PHPMetrics* durchgeführt und dadurch eine verbesserte Bewertung der LLMs erreicht wird. Analog kann so ein Benchmark auf andere relevante Programmiersprachen der Webentwicklung ausgeweitet werden, wie beispielsweise JavaScript, Java und Ruby.

Weil LLMs kein deterministisches Verhalten zeigen, könnte die Auswertung der generierten Codes nicht auch von einer anderen LLM übernommen werden, sodass LLMs einander selbst bewerten? Bei so einem Verfahren, wären Forschungen möglich, ob auf externe Tools und statische Tests ganz verzichtet werden kann.

Ein weiteres mögliches Themenfeld ist der Einführungsprozess von KI gestützten Softwareentwicklung für Webanwendungen unter Anwendung von Firmenrichtlinien. Der Fokus sollte hierbei auf das Auswahlverfahren der Modelle liegen mit Blick auf die möglichen Benchmarks und dem Evaluierungsverfahren. Wie lassen sich Entwickler frühzeitig in die Auswahl der Modelle einbeziehen und welche Möglichkeiten bestehen für Entwickler schon während dieses Prozesses ihre Kenntnisse im Bereich von generativer KI zu erweitern oder vertiefen, sodass effektives Entwickeln von Anfang an erfolgen kann?

7.4 Praktische Anwendung

In der kommenden Zeit wird die Entwicklung der LLM weiter voranschreiten und dies wird zu weiteren Verbesserungen bei den Ausgaben der LLMs führen, was langfristig zu noch effizienteren Entwickeln führt. Dennoch sollte der Focus auf eine einheitliche Vorgehensweise liegen. Somit werden nicht nur gleiche Standards beim Code geschaffen, zudem wird ein einheitliches Know-How der Mitarbeiter realisiert. Dies wird langfristig dazu führen, dass die Entwicklung von Webanwendungen mithilfe generativer KI, die Arbeit der Entwickler effizienter gestaltet und sich somit auf die Kosten und Ressourcen der Unternehmen auswirkt. Was sich positiv auf die Wettbewerbsfähigkeit der Softwareunternehmen auswirken kann.

Die Auswahl der Anbieter wird, durch deren ständig wachsender Anzahl der LLMs immer schwieriger. Die in dieser Arbeit vorgestellten Ergebnisse der evaluierten Modelle, können Unternehmen nutzen, um einen Einstieg in das Thema zubekommen und schlägt Modelle vor, welche sich für die Generierung von Programmcode in der Webanwendungsentwicklung eignen. Des Weiteren unterstützt diese Arbeit die Unternehmen den bei ersten Versuchen, weitere neue LLMs zu evaluieren und das optimale Modell für ihre Prozesse zu finden. Die Mehrheit der Entwickler und Unternehmen nutzen zurzeit kommerzielle Modelle großer Anbieter, da die Nutzung der Chatbots und die schnelle, unkomplizierte

Verwendung der APIs mit wenig Aufwand realisierbar ist. Durch das zunehmende breite Interesse an generativer KI, wurden Möglichkeiten geschaffen Open-Source-Modelle in der eigenen Infrastruktur einzubinden und diese zu verwenden. Dies kann, wie schon zuvor besprochen zur Senkung der Kosten beitragen. Besonders die Abrechnung der Tokens ist ein interessanter Punkt, der nicht immer eindeutig nachvollziehbar ist. Der Kostenaspekt zwischen *Open-Source* und *Closed-Source* Modellen, wurde schon aufgegriffen, diskutiert und bringt laut [40] eine große Ersparnis, was besonders für mittlere und große Unternehmen interessant werden kann. Wie in dieser Arbeit gezeigt, liefern die Open-Source-Modelle hervorragende Ergebnisse für die Aufgaben in der Webanwendungsentwicklung, die oft bessere sind, als die Ergebnisse der kommerziellen Closed-Source-Modelle.

Unabhängig von der Auswahl des Modells sollten Unternehmen für die Entwickler einheitliche Schnittstellen zu den Modellen bereitstellen. Zum Einem können die Eingabeaufforderungen, wie in dieser Arbeit beschrieben, mittels verschiedener Frameworks oder anderer Methoden angepasst werden. Zum anderen können die abgesetzten Prompts verwendet finden, um beispielsweise Modelle weiter, durch „Fine-Tuning“ zu optimieren. Eine weitere Möglichkeit besteht darin die Prompts auszuwerten und den Wissensstand der Entwickler zu prüfen und gegebenenfalls mit gezielten Schulungsmaßnahmen das Entwicklungsteam zu fördern.

Ein weiterer Vorteil dieser Methode ist, dass die Möglichkeit besteht Prompts mit verschiedenen Metadaten anzureichern und Ansätze wie *Personas*, *Beispielen* und *zusätzlichem Kontext* umgesetzt werden können. Vorstellbar wäre auch die Integration eines RAG, welches auf Daten der eigenen Entwickler und bereits umgesetzter Softwarelösungen zurückgreifen kann.

Die Abbildung 7.1 zeigt das zuvor beschriebene Beispiel für die Integration einer LLM in den Prozess der Programmcodegenerierung. Die Entwickler stellen ihre Eingabeaufforderungen an den Agenten. Das erfolgt über ein Eingabeformular oder als Plug-in einer IDE. Über das Plug-in wird der generierte Code direkt in das aktuelle Projekt übernommen. Das Formular bietet die Möglichkeit weitere Fragen an die LLM stellen, die keinen Code generieren sollen. Dies könnten allgemeine Fragen zur Architektur oder Fragen zur Beurteilung von Tools beinhalten. Mit der beschriebenen Architektur, werden alle Fragen und Probleme erfasst, mit der sich die Entwickler auseinandersetzen. Diese Informationen könnten bei der Suche eines geeigneteren Benchmarks behilflich sein.

In einem ersten Schritt könnte der Agent als einfacher Aufruf einer LLM implementiert werden, um ihn später durch einen echten Agenten zu ersetzen. Der Agent leitet den einfachen oder optimierten Prompt an die LLM weiter, wertet die Antwort aus und sendet das Ergebnis an den Entwickler zurück. Zusätzlich werden die Prompts aufgezeichnet und in einer Datenbank gespeichert. In der Abbildung wird die Datenbank als Logs bezeichnet. Für die grün dargestellten Elemente, in der Abbildung 7.1 sind in dieser Arbeit bereits Ansätze und Lösungsvorschläge vorgestellt und besprochen wurden.

Sobald ein Agent implementiert wurde und zum Einsatz kommt, können wertvolle Quellen zur Informationsbeschaffung an den Agenten angeschlossen werden. Dies kann als *Retrieval Augmented Generation*

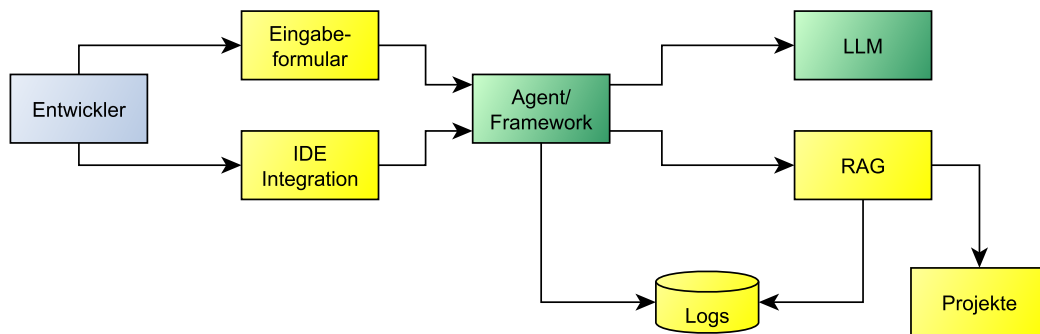


Abbildung 7.1: Anwendungsbeispiel für die Integration einer LLM

(RAG) implementiert werden, um auf die vorhandene firmenintern Daten zuzugreifen. Diese Daten können aus bereits gestellte und bewertete Eingabeaufforderungen der Entwickler oder aus vorhandenem Codes von abgeschlossenen Projekten bestehen. Mit dem, in dieser Arbeit vorgestellten DSPy Framework können unter anderem Agenten und RAG implementiert werden.

Abschließend wird in Abbildung 7.2 ein Beispiel für ein einfaches Chat-Formular gezeigt. Weitere Formularfelder können im Formular hinzugefügt und der LLM als zusätzliche Metadaten oder Anweisungen mitgeteilt werden.

Entwickler-Chatbot

Hallo! Wie kann ich Ihnen helfen?

Ich möchte ein Formular ausfüllen.

Gerne, bitte beschreiben Sie das Formular.

Wähle eine Sprache:

PHP

Programmieraufgabe:

Absenden

Abbildung 7.2: Anwendungsbeispiel für ein Eingabeformular für Prompts

Allerdings kann es hierbei auch zu Problemen kommen, beispielsweise wenn Entwickler falschen Code als richtig bewerten und diese Informationen der LLM zur Verfügung gestellt werden.

Es gibt Modelle welche besonders für die Webanwendungsentwicklung sehr gute Ergebnisse, auch ohne Optimierung liefern. Dennoch sollte eine Optimierung der Eingabeaufforderungen in Betracht gezogen werden, da diese mit relativ wenig Aufwand umsetzbar ist. Der Einsatz von generativer KI wird in allen Bereichen der Programmierung neue Maßstäbe setzen und nachhaltig verändern. Dieser Prozess hat bereits begonnen und immer mehr Unternehmen befassen sich mit diesem Thema und investieren in den Ausbau dieser Technologie. Dies bedeutet, dass eine Transformation auf Unternehmensebene durchzuführen ist, welche die gesamten Prozesse der Codegenerierung umfasst, auch wenn diese Transformation zu Beginn höhere Kosten verursacht.

Es wird festgestellt, dass durch die Evaluation mit Benchmarks eine relevante Kenngröße zur Ermittlung der optimalen LLM gemessen werden kann, obwohl durch die Benchmarks nicht alle Aspekte der Webanwendungsentwicklung abgedeckt werden können. Ebenfalls ist es wichtig, Kenntnisse über die Auswahl geeigneter Methoden in Form von Frameworks oder die Wahl der Sprache zu kennen, um die geeignete LLM auszuwählen.

Alle, in dieser Arbeit vorgeschlagenen Ansätze, können langfristig zu einer Kosteneinsparung beitragen, da sich der Ressourcenverbrauch reduzieren wird. Dies wird sich bei der Entwicklungszeit für Anwendungen, für die Fehlersuche und beim Implementieren von zusätzlichen Funktionen bemerkbar machen.

LITERATUR

- [1] Volker M. Banholzer. *Künstliche Intelligenz als Treiber der Veränderung in der Unternehmenskommunikation 4.0?* Bd. 1/2020. Technische Hochschule Nürnberg Georg-Simon-Ohm, 2020. URL: https://www.th-nuernberg.de/fileadmin/fakultaeten/amp/amp_docs/K%C3%BCnstliche_Intelligenz_und_die_Rolle_n_von_Unternehmenskommunikation_Banholzer_IKOM_WP_1_2020__fin-1.pdf.
- [2] *Digitale Transformation: Fallbeispiele und Branchenanalysen*. 2022. URL: https://library.oapen.org/bitstream/handle/20.500.12657/57358/978-3-658-37571-3.pdf?sequence=1&utm_source=textcortex&utm_medium=zenochat#page=70 (besucht am 19.10.2024).
- [3] Erin Yepis. *Developers want more, more, more: the 2024 results from Stack Overflow's Annual Developer Survey*. 24. Juli 2024. URL: <https://stackoverflow.blog/2024/07/24/developers-want-more-more-more-the-2024-results-from-stack-overflow-s-annual-developer-survey/> (besucht am 09.08.2024).
- [4] Rebeka Tóth, Tamas Bisztray und László Erdodi. *LLMs in Web Development: Evaluating LLM-Generated PHP Code Unveiling Vulnerabilities and Limitations*. 21. Apr. 2024. URL: <https://arxiv.org/abs/2404.14459v2> (besucht am 05.01.2025).
- [5] Juyong Jiang u. a. *A Survey on Large Language Models for Code Generation*. 1. Juni 2024. URL: <https://arxiv.org/abs/2406.00515> (besucht am 07.11.2024).
- [6] Juan David Velásquez-Henao, Carlos Jaime Franco-Cardona und Lorena Cadavid-Higueta. „Prompt Engineering: a methodology for optimizing interactions with AI-Language Models in the field of engineering“. In: *DYNA* 90.230 (3. Nov. 2023), S. 9–17. DOI: 10.15446/dyna.v90n230.111700. URL: <https://doi.org/10.15446/dyna.v90n230.111700>.
- [7] Mayank Mishra u. a. *Granite Code Models: A Family of Open Foundation Models for Code Intelligence*. 7. Mai 2024. URL: <https://arxiv.org/abs/2405.04324> (besucht am 08.11.2024).

- [8] Anton Lozhkov u. a. *StarCoder 2 and The Stack v2: The Next Generation*. 29. Feb. 2024. URL: <https://arxiv.org/abs/2402.19173> (besucht am 08.11.2024).
- [9] Sasikala C 1 Dr.M.Kalpana Devi 2,Tholhappiyan T 3, Sasikala Nataraj. „REVOLUTIONIZING WEB DEVELOPMENT WITH AN INTELLIGENT CHATBOT: a NOVEL APPROACH UTILIZING OPENAI'S GPT-3 AND ADVANCED NLP STRATEGIES“. In: *Machine Intennigence Research* 18.1 (17. Aug. 2024), S. 1098–1109. URL: <http://machineintelligenceresearchs.com/index.php/mir/article/view/90> (besucht am 08.11.2024).
- [10] Daoguang Zan u. a. *Large language models meet NL2Code: a survey*. 19. Dez. 2022. URL: <https://arxiv.org/abs/2212.09420> (besucht am 26.12.2024).
- [11] Pekka Ala-Pietilä u. a. *Eine Definition der KI: Wichtigste Fähigkeiten und Wissenschaftsgebiete*. 5. März 2019. URL: https://elektro.at/wp-content/uploads/2019/10/EU_Definition-KI.pdf (besucht am 10.09.2024).
- [12] Johanna Pahl. *Zeichnung einer biologische Zelle*. 26. Sep. 2024.
- [13] Yoav Goldberg. „A Primer on Neural Network Models for Natural Language Processing“. In: *Journal of Artificial Intelligence Research* 57 (20. Nov. 2016), S. 345–420. DOI: 10.1613/jair.4992. URL: <https://jair.org/index.php/jair/article/view/11030>.
- [14] Xavier Amatriain. *Prompt Design and Engineering: Introduction and Advanced Methods*. 24. Jan. 2024. URL: <https://arxiv.org/abs/2401.14423v3> (besucht am 12.10.2024).
- [15] Banghao Chen u. a. *Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review*. 23. Okt. 2023. URL: <https://arxiv.org/abs/2310.14735v5> (besucht am 26.12.2024).
- [16] Mark Chen u. a. *Evaluating Large Language Models Trained on Code*. 7. Juli 2021. URL: <https://arxiv.org/abs/2107.03374> (besucht am 28.10.2024).
- [17] Qiwei Peng, Yekun Chai und Xuhong Li. *HumanEval-XL: A Multilingual Code Generation Benchmark for Cross-lingual Natural Language Generalization*. 26. Feb. 2024. URL: <https://arxiv.org/abs/2402.16694> (besucht am 15.11.2024).
- [18] Qwen u. a. *QWen2.5 Technical Report*. 19. Dez. 2024. URL: <https://arxiv.org/abs/2412.15115> (besucht am 09.01.2025).
- [19] Binyuan Hui u. a. *QWen2.5-Coder Technical Report*. 18. Sep. 2024. URL: <https://arxiv.org/abs/2409.12186> (besucht am 09.01.2025).

- [20] DeepSeek-Ai u. a. *DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence*. 17. Juni 2024. URL: <https://arxiv.org/abs/2406.11931> (besucht am 09.01.2025).
- [21] Yi Cui. *WebApp1K: A Practical Code-Generation Benchmark for Web App Development*. 30. Juli 2024. URL: <https://arxiv.org/abs/2408.00019v1> (besucht am 09.01.2025).
- [22] *Modelcard Llama3.1-claude on Ollama*. URL: <https://ollama.com/incept5/llama3.1-claude> (besucht am 09.01.2025).
- [23] *Model Meta-Llama-3.1-8B-Claude on Hugging face*. URL: <https://huggingface.co/Undi95/Meta-Llama-3.1-8B-Claude> (besucht am 09.01.2025).
- [24] Meta-Llama. *Llama 3.1 Modelcard*. URL: https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md (besucht am 09.01.2025).
- [25] Meta. *Introducing Code Llama, a state-of-the-art large language model for coding*. 24. Aug. 2023. URL: <https://ai.meta.com/blog/code-llama-large-language-model-coding> (besucht am 25.02.2025).
- [26] Manuel Eberhardinger u. a. *From Code to Play: Benchmarking Program Search for Games Using Large Language Models*. 5. Dez. 2024. URL: <https://arxiv.org/abs/2412.04057v1> (besucht am 09.01.2025).
- [27] Xin Quan u. a. *Verification and Refinement of Natural Language Explanations through LLM-Symbolic Theorem Proving*. 2. Mai 2024. URL: <https://arxiv.org/abs/2405.01379v4> (besucht am 09.01.2025).
- [28] *Models*. URL: <https://platform.openai.com/docs/models> (besucht am 10.01.2025).
- [29] Md Kamrul Siam, Huanying Gu und Jerry Q. Cheng. *Programming with AI: Evaluating ChatGPT, Gemini, AlphaCode, and GitHub Copilot for Programmers*. 14. Nov. 2024. URL: <https://arxiv.org/abs/2411.09224v1> (besucht am 10.01.2025).
- [30] Ran Elgedawy u. a. *Ocasionally Secure: A Comparative Analysis of Code Generation Assistants*. 1. Feb. 2024. URL: <https://arxiv.org/abs/2402.00689v1> (besucht am 10.01.2025).
- [31] *Gemini-Modelle*. URL: <https://ai.google.dev/gemini-api/docs/models/gemini?hl=de> (besucht am 10.01.2025).
- [32] *GGUF*. URL: <https://huggingface.co/docs/hub/en/gguf> (besucht am 23.01.2025).

- [33] Ammar Ahmed u. a. *From Code to Compliance: Assessing ChatGPT's Utility in Designing an Accessible Webpage – A Case Study*. 7. Jan. 2025. URL: <https://arxiv.org/abs/2501.03572v1> (besucht am 10.01.2025).
- [34] Shudan Zhang u. a. *NaturalCodeBench: Examining Coding Performance Mismatch on HumanEval and Natural User Prompts*. 7. Mai 2024. URL: <https://arxiv.org/abs/2405.04520> (besucht am 21.02.2025).
- [35] FOCUS online. „Macht KI uns dümmer? Forscher untersuchen Auswirkungen von Chatbots auf Lern- und Denkfähigkeiten“. In: (21. Feb. 2025). URL: https://www.focus.de/wissen/macht-ki-uns-duemmer-forscher-untersuchen-auswirkungen-von-chatbots-auf-lern-und-denkfaehigkeiten_id_260737569.html.
- [36] *Gartner Says Generative AI will Require 80% of Engineering Workforce to Upskill Through 2027*. Gartner. 3. Okt. 2024. URL: <https://www.gartner.com/en/newsroom/press-releases/2024-10-03-gartner-says-generative-ai-will-require-80-percent-of-engineering-workforce-to-upskill-through-2027> (besucht am 23.02.2025).
- [37] Steffen Albrecht und Karlsruher Institut für Technologie (KIT). *Was ChatGPT für Bildung und Wissenschaft bedeutet*. 25. Apr. 2023. URL: <https://www.helmholtz.de/newsroom/artikel/was-chatgpt-fuer-bildung-und-wissenschaft-bedeutet/> (besucht am 22.02.2025).
- [38] *Microsoft/PHI-2 · Hugging Face*. URL: <https://huggingface.co/microsoft/phi-2> (besucht am 25.02.2025).
- [39] Mengya Hu u. a. *SLM meets LLM: Balancing latency, interpretability and consistency in hallucination detection*. 22. Aug. 2024. URL: <https://arxiv.org/abs/2408.12748> (besucht am 25.02.2025).
- [40] Chandra Irugalbandara u. a. *Scaling Down to Scale Up: A Cost-Benefit analysis of replacing OpenAI's LLM with open source SLMs in production*. 20. Dez. 2023. URL: <https://arxiv.org/abs/2312.14972> (besucht am 25.02.2025).

ANHANG

A Installationshinweise

A.1 Python

Da in dieser Arbeit Python verwendet wird, sollte zu den grundlegenden Paketen, für die Arbeit mit großen Sprachmodellen folgende Zusatzpakete installiert sein.

```
pip3 install langchain
pip3 install ollama
pip3 install transformer
```

Im Weiteren wird kein Hinweis auf verwendete Pakete gegeben. Diese sind evtl. den Fehlermeldungen während und nach der Programmausführung zu entnehmen.

A.2 Installation und Konfiguration von Ollama

Für die Installation von Ollama wird bei Linux folgendes Skript ausgeführt,

```
curl -fsSL https://ollama.com/install.sh | sh
```

Ollama kann in seiner Konfiguration angepasst werden, im Folgenden wurde der Pfad zur den Modellen geändert und die Erreichbarkeit von Ollama über Netzwerk eingestellt. Dazu wird die Datei `/etc/systemd/system/ollama.service` angepasst und der korrekte Host und IP-Adresse gesetzt.

5

```
diff --git a/ollama.service b/ollama.service
--- a/ollama.service
+++ b/ollama.service
@@ -10,3 +10,4 @@
RestartSec=3
Environment="PATH=/usr/local/bin:/usr/bin"
```

```
-  
+   Environment="OLLAMA_HOST=0.0.0.0"  
+   Environment="OLLAMA_MODELS=/home/ai/models "  
+
```

Listing 8.1: Ollama Hostanpassng für Netzwerkbetrieb

Nach der Installation kann die Funktionsfähigkeit geprüft werden, der folgenden Beispielaufruf lädt ein Modell von Ollama und startet dieses.

```
ollama run deepseek-coder-v2:16b
```

Im Anschluss kann über die Konsole mit dem Modell interagiert werden.

A.3 Open WebUI Installationshinweise

Hier wird Open WebUI als docker Container verwendet, es ist also erforderlich vorher docker zu installieren. Die Installation von Open WebUI, unter Debian kann mit folgendem Skript erfolgen.

```
# Pull Open WebUI container.  
docker pull ghcr.io/open-webui/open-webui:main  
  
# Run container.  
5 docker run -d --network=host -v open-webui:/app/backend/data \  
-e OLLAMA_BASE_URL=http://127.0.0.1:11434 --name open-webui \  
--restart always ghcr.io/open-webui/open-webui:main
```

Listing 8.2: Open WebUI installieren

Der Aufruf der UI, kann mittel Browser erfolgen. Hier wird die IP und der Port 8080 angegeben. Beispiel `http://192.168.2.45:8080`.

B Download der Hugging Face Modelle

Ein Hugging Face Modell kann, heruntergeladen werden. Soll das Modell auch nach dem Löschen des lokalen huggingface-Caches zur Verfügung stehen, sollte das Modell separat abgespeichert werden. Für den Download und speichern der Modelle kommt das Python-Skript, wie in Listing 8.3 gezeigt zur Anwendung. Hier ist zu erwähnen, dass ausreichend RAM zur Verfügung stehen muss, um die Modelle zu speichern. Mit diesem Script kann ein Modell an einem angegebenen Pfad abgespeichert werden und ist nach dem Cache löschen immer noch lokal vorhanden.

```

3  """
  The script downloads a model of Hugging Face and saves it locally.
  """

  __VERSION__ = "0.0.1"

  from transformers import AutoModelForCausalLM, AutoTokenizer

8  def load_model_from_huggingface(
    root_name: str, vendor_name: str
  ) -> None:
    """
13  Load model from Hugging Face and save local.

    :param root_name (str): Root name of model
    :param vendor_name (str): Vendor name of model
    """

18  # Personal access token from Hugging Face.
    access_token = "hf_XXXXXXXXXXXXXXXXXXXXXXXXXXXX"

    model_path_to_save = f"/hf/models/{vendor_name.replace('-', '_')}"

23  tokenizer = AutoTokenizer.from_pretrained(
    f"{root_name}/{vendor_name}"
  )
  model = AutoModelForCausalLM.from_pretrained(
    f"{root_name}/{vendor_name}",
28  is_decoder=True,
    token=access_token,
  )

  tokenizer.save_pretrained(model_path_to_save)
33  model.save_pretrained(model_path_to_save)

```

```
38 # Example from https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct.  
    load_model_from_huggingface(  
        root_name="Qwen", vendor_name="Qwen2.5-Coder-32B-Instruct"  
    )
```

Listing 8.3: Laden der Modelle von Hugging Face und lokal speichern

C Abfragen lokaler Modelle

Um die Modelle abzufragen, kommt der Code, welcher in Listing ?? zu sehen ist, zum Einsatz. Hier wird die Abfrage mit dem langchain-Framework gezeigt.

```
"""
2 Create answers by Ollama.
"""

__VERSION__ = "0.0.1"

7 from langchain_ollama.llms import OllamaLLM
  from langchain.prompts import PromptTemplate

MODEL_SETTING_MAX_TOKEN: int = 600
12 MODEL_SETTING_TOP_P: float = 0.95
MODEL_SETTING_TEMP: float = 0.2

MODEL_CONNECT_HOST: str = "192.168.178.140"
MODEL_CONNECT_PORT: str = "11434"
17

def run_model(problem: str, model: OllamaLLM) -> str:
    """
    22 Connect to ollama model server.

    :param problem (str): Prompt there is send to ollama server.
    :param model (OllamaLLM) Connected model.
    :return Result from ollama server by prompt.
    """
    27 prompt_template: PromptTemplate = PromptTemplate(
        input_variables=["user_prompt"],
        template="{user_prompt}",
    )
    prompt = prompt_template.format(user_prompt=problem)
    32 return model.invoke(prompt)

def connect_model(model_name: str) -> OllamaLLM:
    37 """
    Connect Ollama hosted model
```



```
42 :param model_name (str): Model name.
    :return OllamaLLM: Connected model.
    """
    return OllamaLLM(
        base_url=f"{MODEL_CONNECT_HOST}:{MODEL_CONNECT_PORT}",
        model=model_name,
        max_token=MODEL_SETTING_MAX_TOKEN,
47         temperature=MODEL_SETTING_TEMP,
        top_p=MODEL_SETTING_TOP_P,
    )

52 def execute_prompt(
    prompt: str,
    model_name: str,
    prompt_repetitions: int = 5,
) -> list[str]:
57     """
    Create prompts and save local.

    :param problem (str): Problem or prompt.
    :param model_name (str): Ollama model name.
62     :param prompt_repetitions (int, optional): _description_. Defaults
        to 5.
    :return (str): Model answer.
    """

    if prompt is None:
        return False

67

    current_count: int = 0
    max_count: int = prompt_repetitions
    model = connect_model(model_name=model_name)
    answers: list[str] = []
72     while current_count < max_count:
        answers.append(run_model(problem=prompt, model=model))
        current_count += 1

    return answers
```

Listing 8.4: Abfragen der Ollama Modelle mit langchain

Für die Optimierung mit dem DSPy-Framework wurde der Code, welcher in Listing 8.5 gezeigt wird angewandt.

```
"""
Create answers by Ollama with DSPy.
"""
4  __VERSION__ = "0.0.1"

import dspy

9  MODEL_SETTING_MAX_TOKEN: int = 600
MODEL_SETTING_TOP_P: float = 0.95
MODEL_SETTING_TEMP: float = 0.8 # eigentlich 0.2

14 MODEL_CONNECT_HOST: str = "192.168.178.140"
MODEL_CONNECT_PORT: str = "11434"

class BasicProgramming(dspy.Signature):
19     """
    Custom signature.

    Args:
        dspy (_type_): Base signature from dspy.
24     """

    question: str = dspy.InputField(desc="Eine Frage zu PHP
        Programmierung")
    answer: str = dspy.OutputField(desc="Generiere Programmcode")

29 def run_model(prompt: str, model: dspy.LM) -> str:
    """
    Connect to ollama model server.

34     :param prompt (str): Prompt there is send to ollama server.
    :return Result from ollama server by prompt.
    """

    model(messages=[{"role": "user", "content": "Du bist erfahrener PHP
        Entwickler"}])
    chain_elem = dspy.ChainOfThought(BasicProgramming)
```

```
39     pred = chain_elem(question=prompt)

    return pred.answer

44 def connect_model(model_name: str) -> dspy.LM:
    """
    Connect Ollama hosted model

    :param model_name (str): Model name.
    :return OllamaLLM: Connected model.
    """
    llm: dspy.LM = dspy.LM(
        model=f"ollama_chat/{model_name}",
        api_base=f"http://{MODEL_CONNECT_HOST}:{MODEL_CONNECT_PORT}",
54     api_key="",
        temperature=MODEL_SETTING_TEMP,
        cache=False,
        cache_in_memory=False,
        max_tokens=4096,
59     num_retries=2,
    )
    dspy.configure(lm=llm)

    return llm

64

def execute_prompt(
    prompt: str,
    model_name: str,
69     prompt_repetitions: int = 5,
) -> list[str]:
    """
    Create prompts and save local.

    :param problem (str): Problem or prompt.
    :param model_name (str): Ollama model name.
    :param prompt_repetitions (int, optional): _description_. Defaults
        to 5.
    :return (str): Model answer.
    """
79     if prompt is None:
        return False
```

```
84     current_count: int = 0
      max_count: int = prompt_repetitions
      model = connect_model(model_name=model_name)
      answers: list[str] = []
      while current_count < max_count:
          answers.append(run_model(prompt=prompt, model=model))
          current_count += 1
89
      return answers
```

Listing 8.5: Abfragen der Ollama Modelle mit DSPy

D Abfrage cloudbasierter Modelle

Das Listig 8.6 zeigt die Umsetzung der Abfrage von den Gemini Modellen mit Python. Um die Abfragen durchzuführen sind neben dem Projektnamen und des Standortes, noch ein gültiger GCP Account erforderlich.

```
"""
Create answers by Google Gemini.
"""

5  __VERSION__ = "0.0.1"

from google import genai
from google.genai import types

10  MODEL_SETTING_MAX_TOKEN: int = 600
    MODEL_SETTING_TOP_P: float = 0.95
    MODEL_SETTING_TEMP: float = 0.2

15  GOOGLE_PROJECT: str = "XX-XXXxx"
    GOOGLE_LOCATION: str = "us-central1"

ALLOWED_LLMS: list[dict] = [
20      {"model": "gemini-1.5-pro-002", "answer_folder": "gemini15"},
      {"model": "gemini-2.0-flash-exp", "answer_folder": "gemini2"},
]

25  def run_model(
        problem: str, client: genai.Client, model_name: str
    ) -> str:
    """
    Run Google model.

30      :param problem (str): Problem or prompt.
      :param client (genai.Client): Gemini client.
      :param model_name (str): Gemini model name.
      :return (str): Model answer.

35      """
    generate_content_config = types.GenerateContentConfig(
        temperature=MODEL_SETTING_TEMP,
```

```

    top_p=MODEL_SETTING_TOP_P,
    max_output_tokens=MODEL_SETTING_MAX_TOKEN,
    response_modalities=["TEXT"],
)

text: str = ""
for chunk in client.models.generate_content_stream(
    model=model_name,
    contents=[problem],
    config=generate_content_config,
):
    text += chunk.text

return text.rstrip()

def connect_client() -> genai.Client:
    """
    Get Gemini client.

    :return genai.Client: Gemini client.
    """
    return genai.Client(
        vertexai=True,
        project=GOOGLE_PROJECT,
        location=GOOGLE_LOCATION
    )

def execute_prompt(
    prompt: str,
    model_name: str,
    prompt_repetitions: int = 5,
) -> list[str]:
    """
    Execute model with problem.

    :param prompt (str): Prompts with sample.
    :param model_name (str): Model name.
    :param prompt_repetitions (int, optional): Count of repetitions.
        Defaults to 5.
    :return bool: Answers by model.
    """

```

```
80     if prompt is None:
        return False

        current_count: int = 0
        max_count: int = prompt_repetitions
85     client = connect_client()
        answers: list[str] = []
        while current_count < max_count:
            answers.append(
                run_model(
90                    problem=prompt,
                    client=client,
                    model_name=model_name
                )
            )
85         current_count += 1

        return answers
```

Listing 8.6: Ausführen der Prompts für Gemini Modelle.

Ähnlich wie die Gemini Modelle, werden auch die ChatGPT Modelle von OpenAI abgefragt. Auch hier ist ein gültiger Account und ein API Key notwendig. Das Listing 8.7 zeigt die Abfrage mittels Python.

```
3 """
Create answers by OpenAI ChatGPT.
"""

__VERSION__ = "0.0.1"

from openai import OpenAI

8
MODEL_SETTING_MAX_TOKEN: int = 600
MODEL_SETTING_TOP_P: float = 0.95
MODEL_SETTING_TEMP: float = 0.2

13 OPEN_AI_KEY: str = "OPEN_AI_PERSONAL_KEY"
OPEN_AI_ORG: str = "org-XXXX"
OPEN_AI_PROJECT: str = "proj_YYYY"

ALLOWED_LLMS: list[dict] = [
18     {"model": "gpt-3.5-turbo", "answer_folder": "gpt35"},
    {"model": "gpt-4-turbo", "answer_folder": "gpt4"},
]

23 def run_model(
    problem: str, client: OpenAI, model_name: str
) -> str:
    """
    Run OpenAI models.

28
    :param problem (str): Problem or prompt
    :param client (OpenAI): OpenAI client.
    :param model_name (str): OpenAI model name.
    :return str: Model answer.
33
    """
    completion = client.chat.completions.create(
        model=model_name,
        max_tokens=MODEL_SETTING_MAX_TOKEN,
        temperature=MODEL_SETTING_TEMP,
38         top_p=MODEL_SETTING_TOP_P,
        messages=[
            {
```



```
43         "role": "user",
            "content": problem,
        },
    ],
)

    return completion.choices[0].message.content

48

def connect_client() -> OpenAI:
    """
    Get OpenAI client.

53    :return OpenAI: OpenAI client.
    """

    return OpenAI(
        api_key=OPEN_AI_KEY,
58        organization=OPEN_AI_ORG,
        project=OPEN_AI_PROJECT,
    )

63 def execute_prompt(
    prompt: str,
    model_name: str,
    prompt_repetitions: int = 10,
) -> list[str]:
68     """
    Execute model.

    :param prompt (): Sample.
    :param model_name (str): Model name.
73    :param prompt_repetitions (int, optional): _description_. Defaults
        to 5.
    :return bool: Answers.
    """

    if prompt is None:
        return False

78

    current_count: int = 0
    max_count: int = prompt_repetitions
    client = connect_client()
    answers: list[str] = []
```

```
83 while current_count < max_count:
    answers.append(
        run_model(
            problem=prompt,
            client=client,
88         model_name=model_name
        )
    )
    current_count += 1

93 return answers
```

Listing 8.7: Ausführen der Prompts für OpenAI Modelle.

E Evaluation der Antworten von den Modellen

Das Listing 8.8 zeigt beispielhaft eine Suche nach Codezeilen in einer Antwort eines Modells. Die Methode `search_generated_code` muss auf jedes Modell angepasst sein.

```
"""
2  Helpers for large language model operations.
  """

  __VERSION__ = "0.0.1"

7  import subprocess
  import json
  import os

  from pass_at_k import pass_at_k
12

  DATA_BASE_PATH: str = f"{os.path.dirname(os.path.realpath(__file__))}/
    data"

  MODEL_RESPONSES_EVALUATED: list[str] = [
17      "chatgpt4",
        "genini15_flash",
        "genini15_pro",
        "deepseek_coder_v2",
        "deepseek_r1",
22      "llama31",
        "llama32",
        "llama33",
        "mistral_small",
        "qwen25_coder",
27      "codellama_13b",
        "codellama_70b",
    ]

32 def search_generated_code(content: str) -> str:
    """
        Search the generated code in string, it is one line of file.

        Args:
37         content (str): Answer from LLM.
```

```

Returns:
    str: Generated code.
"""
42 for index in [i for i in range(0, 5)]:
    if content.startswith('{ "result_' + str(index) + '":' ):
        generated_code: str = content.split(
            '{ "result_' + str(index) + '":'
47         )[1]
        generated_code = generated_code[:-2]
        generated_code = generated_code.strip()

        if len(generated_code.split(" ' 'php")) > 1:
            generated_code = generated_code.split(" ' 'php")[1]
52         generated_code = generated_code.split(" ' '")[0]

        if generated_code.startswith("<?php"):
            generated_code = generated_code.split("<?php")[1]
            generated_code = generated_code.split(">")[0]
57         if generated_code.startswith(" ' '\n<?php\n"):
            generated_code = generated_code.split("<?php")[1]
            generated_code = generated_code.split(">")[0]

        if len(generated_code.split(r"\n}\n")) > 0:
            generated_code = (
                generated_code.split(r"\n}\n")[0] + "\n}\n"
62             )

        return generated_code

67     return ""

72 def get_file_content(
    file_name: str, count_problems: int, answer_path: str
) -> tuple[list[str], int]:
    """
    Get the content from JSONL answer file.
77

    :param file_name (str): File name with answers.
    :param count_problems (int): Number of first samples from each test
        to be tested.

```

```

:param answer_path (str): Path to answer file.
:return tuple[list[str], int]: Generated code snippets from LLM and
    count of snippets.
82 """
    answers: list = []
    length: int = 0
    with open(f"{answer_path}/{file_name}.jsonl", encoding="utf-8") as
        answer_file:
            lines: str = answer_file.read()
87         length = 5
            for line in lines.split("\n"):
                if line != "":
                    answers.append(search_generated_code(line))
                    if len(answers) >= count_problems:
92                         return (answers, length)
    return (answers, length)

def evaluate(
97     problem_path: str,
    answer_path: str,
    num_consider_samples: int,
):
    """
102     Execute the evaluation.

    :param problem_path (str): Path to problem.
    :param answer_path (str): Path to answers.
    :param num_consider_samples (int): Numbers of consider samples.
107     :param search_generated_code (function): Function for extracting the
        program code.
    """
    num_consider_samples = 5
    summary_answer_pass_at: dict = {"1": 0.0, "2": 0.0, "3": 0.0, "4":
        0.0, "5": 0.0}
    with open(problem_path, "r", encoding="utf-8") as problems:
112         lines: list = problems.read().split("\n")

        total_samples: int = 0
        correct_samples: int = 0
        answers_per_sample: int = 0
117
        for line in lines:

```

```
problem_result: str = ""
if line != "":
    answer_file_name: str = (
122         json.loads(line).get("task_id", "").replace("/", "-")
    )

    problem_result = (
        json.loads(line).get("task_id", "")
127         + ";"
        + str(num_consider_samples)
    )
    test: str = json.loads(line).get("test", "")
    content_tuple: tuple = get_file_content(
132         file_name=answer_file_name,
        count_problems=num_consider_samples,
        answer_path=answer_path,
    )
    answers: list[str] = content_tuple[0]
137    answers_per_sample = content_tuple[1]

    count: int = 1
    # False == durchgefallen
    answer_result_fault: bool = False
142    ready_count: int = 0
    total_samples += 1
    for answer in answers:
        answer = answer.replace(r"\n", "\n")
        answer = answer.replace(r"\"", "'")

147        test = test.replace(r"\"", "'")

        code: str = test.replace("\n<?php\n", "\n")
        code += answer.replace("\n<?php\n", "\n")

152        try:
            result = subprocess.run(
                ["php", "-r", code],
                capture_output=True,
                text=True,
                check=False,
                timeout=5,
157            )
```

```

162         except subprocess.TimeoutExpired:
            pass

            if result.stderr.strip() == "":
                answer_result_fault = True
                ready_count += 1

167         count += 1

            if answer_result_fault:
                correct_samples += 1

172         if answer_file_name == "php-11":
            exit()

            problem_result += ";" + str(ready_count)
177         for k in range(1, 6): # range(1,6) -> [1, 2, 3, 4, 5]
            pass_at_k_value: float = pass_at_k(
                num_total_samples_n=answers_per_sample,
                num_correct_samples_c=ready_count,
                k=k,
182            )
            problem_result += ";" + f"{{(pass_at_k_value):.4f}}"
            summary_answer_pass_at[f"{k}"] += pass_at_k_value
        print(problem_result)

187        print(f"total samples;{{total_samples}}")
        print(f"correct samples;{{correct_samples}}")
        print(f"answers per sample;{{answers_per_sample}}")
        for key, value in summary_answer_pass_at.items():
            print(f"k={{key}};{{(value / 80):.4f}}")

192
# Example: Evaluate the sample results.
# The problem and the outputs of the LLM are read from the stored data,
# evaluated and the result is
# stored again.
197 evaluate(
    problem_path=f"{{DATA_BASE_PATH}}/problems/PHP_German.jsonl",
    answer_path=f"{{DATA_BASE_PATH}}/answers/deepseek_coder_v2",
    num_consider_samples=5,
)

```

Listing 8.8: Evaluation der Modellantworten: Suche nach Codeausschnitten

F Methodenerkennung unter PHP

Mit den folgenden Skripten wird gezeigt das es möglich ist PHP Funktionen welche durch LLMs erzeugt wurden zu scannen und beispielsweise deren Namen in Tests verwenden zu können. Das Listing 8.9 zeigt ein Skript, das beispielhaft verschiedene Funktionen implementiert die durch das Skript im Listing 8.10 zur Laufzeit erkannt werden.

```
<?php
// @file: methods.php

4 function fibfib_rekursiv($n) {}

function fibfib_iterativ($n) {}

function fib($x, $y) {}
9 function fibfib($n) {}
```

Listing 8.9: PHP Skript welches verschiedene Funktionen implementiert

```
<?php
// @file scan_methods.php

4 <?php

// include "method_names.php";
$filename = "method_names.php";
$code = file_get_contents($filename);
9 $tokens = token_get_all($code);

$all = [];
$search = "";
$func_name = False;
14 foreach ($tokens as $token) {
    if (
        gettype($token) == "array" &&
        count($token) > 1 &&
        trim($token[1]) != "" &&
19 trim($token[1]) != "<?php"
    ) {
        if ($token[1] == "function") {
            if (str_ends_with($search, ",")) {
                $search = substr($search, 0, strlen($search) - 1);
```

```
24     }
    $search = $search . ")";
    if ($search != ")") {
        $all[] = $search;
    }
29     $search = $token[1] . " ";
    $func_name = True;
} elseif ($func_name) {
    $func_name = False;
    $search = $search . $token[1] . "(";
34 } else {
    $search = $search . $token[1] . ",";
}
}
}
39
if (str_ends_with($search, ",")) {
    $search = substr($search, 0, strlen($search) - 1);
}
44 $search = $search . ")";
    $all[] = $search;

print_r($all);
```

Listing 8.10: PHP Skript das Methoden zur Laufzeit erkennt

Nach dem Ausführen des Skriptes wird die Ausgabe, welche in Listing ?? zu sehen ist erzeugt.

```
Array
(
3   [0] => function fibfib_rekursiv($n)
    [1] => function fibfib_iterativ($n)
    [2] => function fib($x,$y)
    [3] => function fibfib($n)
)
```

Listing 8.11: Erkannte PHP Methoden

Erklärung zur Verwendung von KI-Systemen

Ich erkläre, dass ich

- mich aktiv über die Leistungsfähigkeit und Beschränkungen der in meiner Arbeit eingesetzten KI-Systeme informiert habe;
- alle Inhalte aus wissenschaftlich anerkannten Quellen entnommen und entsprechend gekennzeichnet habe; alle Inhalte unter Anwendung wissenschaftlicher Methoden im Rahmen der vorliegenden Arbeit von mir selbst entwickelt wurden;
- mir bewusst bin, dass ich als Autor*in dieser Arbeit die Verantwortung für die in ihr gemachten Angaben und Aussagen trage.

Bei der Erstellung der Arbeit habe ich die folgenden auf künstlicher Intelligenz (KI) basierten Systeme in der im Folgenden dargestellten Weise benutzt:

Arbeitsschritt	Eingesetzte(s) KI-System(e)	Beschreibung der Verwendungsweise
Generierung von Ideen und Konzeption der Arbeit	- - - -	- - - -
Literatursuche	- - - -	- - - -
Literaturanalyse	- - - -	- - - -
Literaturverwaltung und Zitationsmanagement	- - - -	- - - -
Auswahl der Methoden und Modelle	- - - -	- - - -
Datensammlung und -analyse	- - - -	- - - -

KAPITEL 8. ANHANG

Generierung von Programmcodes	ChatGPT und Gemini	Erstellen grundlegender Anweisungen und Coderecherche für Python
Erstellung von Visualisierungen	- - - -	- - - -
Interpretation und Validierung	- - - -	- - - -
Strukturierung des Texts der Arbeit	ChatGPT	Generierung von Vorschlägen für den Aufbau der Kapitel
Formulierung des Texts der Arbeit	ChatGPT	Umformulieren von Absätzen, mit denen ich nicht zufrieden war
Übersetzung des Texts der Arbeit	DeepL	Übersetzung von Text für eigenen Code
Redigieren des Texts	ChatGPT und Duden Online	Korrektur von Rechtschreibfehlern und Grammatikfehlern und Umstrukturierung von Satzbau
Vorbereitung der Präsentation des Texts	- - - -	- - - -
Sonstiges	- - - -	- - - -

Temmen-Ringenwalde, 2. März 2025

Ort, Datum



Unterschrift