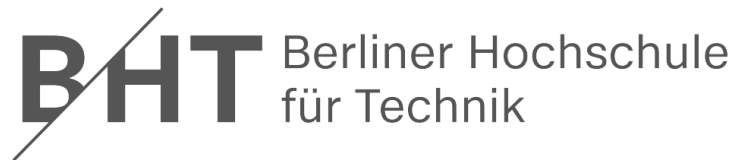


Evaluierung und Optimierung von Large Language Models für die Entwicklung von Webanwendungen

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 1. Februar 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, 1. Februar 2025

Ort, Datum

Unterschrift

ABSTRACT

Abstract in Englisch.

ZUSAMMENFASSUNG

Zusammenfassung in Deutsch.

Inhaltsverzeichnis

Abstract	i
Abbildungsverzeichnis	vi
Tabellenverzeichnis	vii
Listings	viii
Abkürzungsverzeichnis	ix
1 Einleitung	1
1.1 Hintergrund und Kontext	1
1.2 Problemstellung	2
1.3 Stand der Forschung	3
1.4 Zielsetzung und Forschungsfragen	3
1.5 Aufbau der Arbeit	4
1.6 Abgrenzung	4
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	12
2.2 Natural Language Processing	12
2.3 Large Language Model	13
2.3.1 Grundlagen	13
2.3.2 Grenzen und Probleme bei LLMs	14
2.3.3 Verständnis für die LLMs	15
2.4 Koordinationsstrategien für LLMs	15
2.4.1 Orchestrierung von LLMs	16
2.4.2 Multi-Agenten-Systeme	17
2.5 Prompt Engineering	18

2.5.1	Prompt-Techniken	18
2.5.2	Grenzen beim Prompt-Engineering für LLMs	19
2.6	Grundlagen der Webentwicklung	20
2.6.1	Programmiersprachen	20
2.6.2	Entwicklung	20
2.7	Benchmark für LLM	21
2.7.1	pass@k Methode	21
3	Konzeption / Design	23
3.1	Definition der Evaluierungsziele	23
3.2	Auswahl der LLMs und deren Konfiguration	24
3.3	Design der Experimente	27
3.4	Konzeption des Prompt-Engineerings	28
3.5	Evaluationsumgebung	29
4	Implementierung	31
4.1	Lokale Modelle	31
4.1.1	Ergebnisse generieren	31
4.2	Benchmark Codeevaluation	32
4.3	Codeevaluation mit Frameworks	33
4.3.1	PHP Codeevaluation	33
4.4	Optimierung der Antworten	34
5	Evaluation	35
5.1	Modellbewertung mit HumanEval Benchmark	35
5.1.1	Evaluierung nach Sprache	38
5.1.2	Nachteile der Evaluierung	38
5.2	Optimierung der Ergebnisse	41
6	Lessons Learned	43
6.1	Evaluierung der großen Sprachmodelle	43
6.1.1	Lokale Ressourcen	43
6.1.2	Auswertung des Benchmarks	44
6.2	Erweiterte Codeevaluation	44
6.2.1	PHPUnit	45
6.2.2	PHPMetrics	45
6.2.3	SonarQube	45
7	Diskussion und Ausblick	47
7.1	Impulse für zukünftige Forschungen	48
7.2	Praktische Anwendung	49

7.2.1	Anwendung für Entwickler	49
8	Fazit	51
	Literatur	53
	Glossar	56
	Anhang	57
A	Installationshinweise	57
A.1	Python	57
A.2	Installation und Konfiguration von Ollama	57
A.3	Open WebUI Installationshinweise	58
A.4	Download der Hugging Face Modelle	58
A.5	Abfragen lokaler Modelle	59
A.6	Abfrage cloudbasierter Modelle	62
A.7	Evaluation der Antworten von den Modellen	67

Abbildungsverzeichnis

2.1	LLMs im Kontext der Forschungsbereiche von KI	7
2.2	Biologische Nervenzelle	10
2.3	Künstliche Nervenzelle	10
3.1	Codegeneration	28
5.1	Ergebnisse der pass@k-Methode für die Modelle	37
5.2	Ergebnisse der pass@k-Methode für Llama Modelle verschiedener Sprachen . . .	39

Tabellenverzeichnis

3.1	Auswahl der LLMs für die Evaluierung	25
3.2	Einstellungen der Modellparameter	26
5.1	Ergebnisse der pass@1 und pass@5 Methode	36

Listings

3.1	Beispiel für einen Test aus dem HumanEval-XL Benchmark	27
3.2	Prompt Beispiel für eine Aufgabe aus dem HumanEval-XL Benchmark	29
4.1	Codesnippet zur Ausführung des PHP Interpreters	32
4.2	Berechnung der pass@k Metrik in Python	33
5.1	Wortlaut der Aufgabe 20 im HumalEval-XL Benchmark	39
5.2	Wortlaut des Tests 20 im HumalEval-XL Benchmark	40
5.3	Fehler bei der Auswertung durch fehlerhafte Anführungszeichen	40
6.1	Beispiel für Bewertungskriterien	44
8.1	Ollama Hostanpassng für Netzwerkbetrieb	57
8.2	Open WebUI installieren	58
8.3	Laden der Modelle von Hugging Face und lokal speichern	59
8.4	Abfragen der Ollama Modelle	59
8.5	Ausführen der Prompts für Gemini Modelle.	62
8.6	Ausführen der Prompts für OpenAI Modelle.	64
8.7	Evaluation der Modellantworten: Suche nach Codeausschnitten	67

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 Kundendatendaten 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 Forschungsfragen

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 Zielsetzung bereits aus dem Titel „*Evaluierung und Optimierung von Large Language Models für die Entwicklung von Webanwendungen*“ dieser Arbeit herleiten. Sie untersucht die Möglichkeiten mit natürlicher Sprache, Code zu generieren. Wie in [5, vgl. Seite 2] wird auch in dieser Arbeit die Kurzform NL2Code, für Language-to-Code verwendet. Diese Arbeit soll eine Auswahl von

Modellen evaluieren und dessen Brauchbarkeit für die Entwicklung von Webanwendungen aufzeigen. Um die Antworten der Modelle zu optimieren, soll eine Evaluation von Methodiken erfolgen, bei der deren Anwendung eine Verbesserung der Antworten ersichtlich ist.

Als erstes Ziel soll jedoch die Evaluierung des generierten Codes der Modelle stehen und dessen Tauglichkeit für die Programmierung untersucht werden. Sobald Code geschrieben wurde und von anderen Programmierenden überarbeitet oder verstanden werden muss, erfordert dies erheblich mehr Zeit als das ursprüngliche Schreiben des Codes. Daher ist es essenziell, dass der Code gut strukturiert ist und durch Kommentare ergänzt wird, um die Verständlichkeit und Wartbarkeit zu erleichtern. Viele Evaluierungsdatensätze prüfen lediglich die korrekte Funktionalität des Codes.

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

- Z1** Können LLMs hochwertigen Code für die Entwicklung von Webanwendungen generieren und welche Modelle eignen sich für die Entwicklung von Webanwendungen?
- Z2** Wie gut sind die Ergebnisse, hinsichtlich Syntax, Funktionalität, Codequalität und Coding-Standards?
- Z3** Welches Prompt-Design kann helfen die Codequalität und die Genauigkeit der Ergebnisse zu verbessern?

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

verwendeten Programmiersprachen wie JavaScript, HTML, CSS und 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. Es wird ein Überblick über die wichtigsten Teilgebiete gegeben, wobei die einzelnen Bereiche nur oberflächlich behandelt werden können, um eine grundlegende Einführung zu bieten.

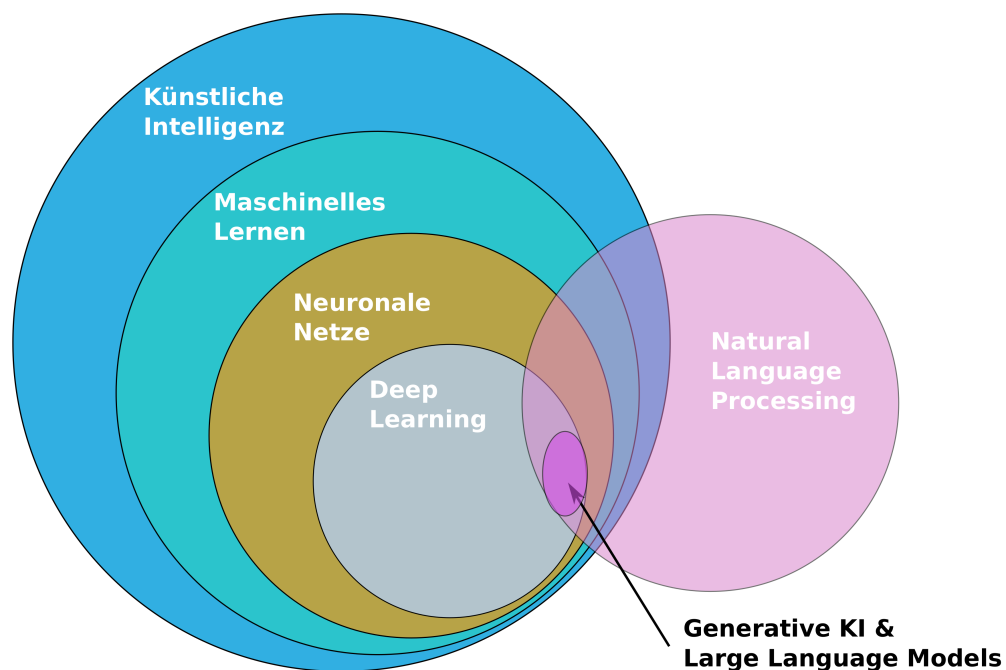


Abbildung 2.1: LLMs im Kontext der Forschungsbereiche von KI

Die Forschungsfelder der großen Sprachmodelle (Large Language Models, LLM) sind eng mit den Teilbereichen Deep Learning 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“ „Deep Learning“ 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 „Deep Learning“ 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 „Deep Learning“ 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

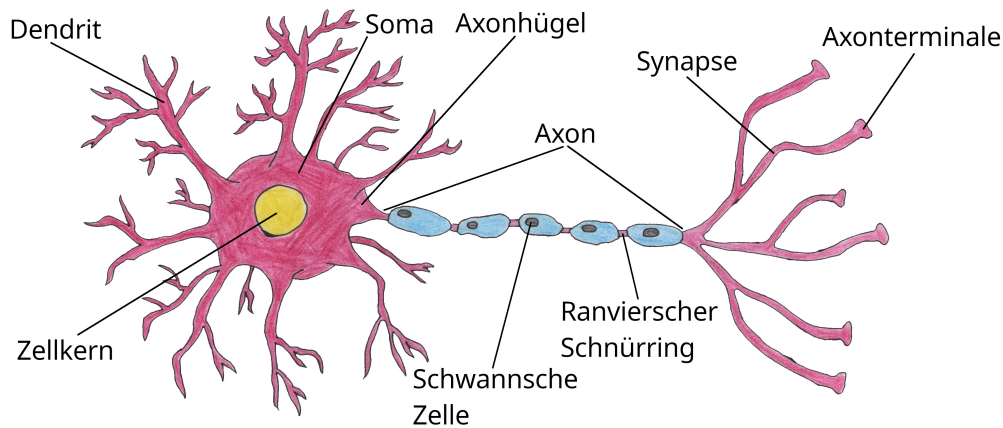


Abbildung 2.2: Biologische Nervenzelle

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.

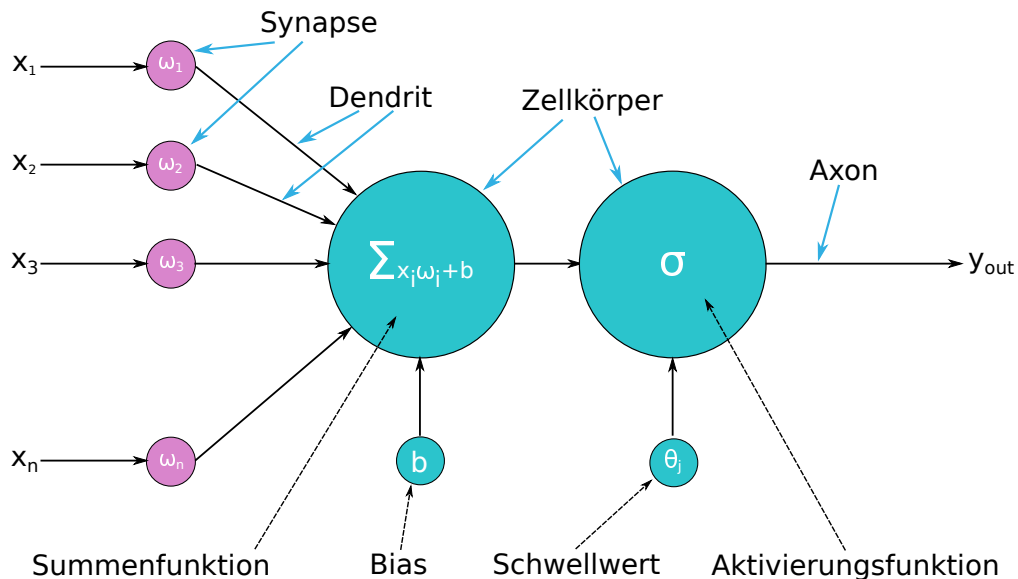


Abbildung 2.3: Künstliche Nervenzelle

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

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

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 folgenden 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 benötigt. 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 plausible 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. Diese Unwissenheit 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 Koordinationsstrategien für LLMs

Es gibt mehrere Ansätze, dass LLMs komplexe Probleme gemeinsam bearbeiten. Dabei übernehmen die LLMs unterschiedliche Aufgaben oder Teilaufgaben des Problems und tragen somit zur Gesamtlösung bei. Durch diese modulare Struktur ist es möglich, dass für jedes Teilproblem eine spezialisierte LLM einzusetzen. Ein weiterer Vorteil dieser Methoden ist die Verteilung des Problems auf mehrere Systeme, die voneinander unabhängig sind. Durch diese Zusammenarbeit können die LLMs effizienter und genauer antworten

Des Weiteren sind die LLMs in der Lage mit externen Ressourcen zusammenarbeiten und Ergebnisse abfragen. So ist es möglich die Fähigkeiten der LLMs zu erweitern z.B. durch Datenbankabfragen, Web-

Suche oder Steuerung von Hardware. Ein wichtiger Aspekt für die Koordination zwischen LLMs ist die Kommunikation. Diese ist entscheidend für die erfolgreiche Bearbeitung des Problems und koordinieren die LLMs.

In den folgenden Kapiteln werden zwei Ansätze für die Zusammenarbeit von mehreren LLMs erläutert. Zum einen die *Orchestrierung*, zum anderen die *Multi-Agenten-System (MAS)*.

2.4.1 Orchestrierung von LLMs

Bei der Orchestrierung von LLMs wird die Steuerung, der Agenten mittels eines zentralisierten Systems umgesetzt, es erfolgt eine koordinierte Nutzung. Meist wird ein Problem in Teilprobleme zerlegt und die Agenten bearbeiten Teilprobleme meist parallel. Die zentrale Steuerung entscheidet welche Teilaufgabe, welcher Agent am besten geeignet ist für die Lösung der Teilaufgabe.

Die zentrale Rolle in der Orchestrierung von LLMs übernimmt dabei der Orchestrator. Dieser steuert die Aufgabenverteilung, koordiniert und kombiniert die Ergebnisse und leitet sie in die entsprechenden Agenten oder erstellt daraus die Antwort, außerdem kann er zusätzliche Aufgaben wie Fehlerbehandlung, Skalierung, Datenschutz und Sicherheit ausführen.

Im Bereich der Softwareentwicklung mit Spezialisierung auf internetbasierte Anwendungen, bei der bestimmte Standards erwartet, spezielle Frameworks und Bibliotheken eingesetzt werden, könnte eine Orchestrierung bei der Umsetzung der Programmcodeerstellung wie folgt beschrieben, helfen. Bei der Lösung von Anforderungen sind nicht immer alle Agent beteiligt, vielmehr sucht der Orchestrator die jeweiligen optimalen Agenten aus.

Der Orchestrator übernimmt auch hier die oben beschriebenen Aufgaben. Ein Frontend-Agent nutzt eines der großen Sprachmodelle, um Nutzeranforderungen in die Benutzeroberflächen der Anwendungen zu implementieren und könnte das Design verwalten. Gleichzeitig wäre es möglich, dass dieser Agent Tools wie React.js oder Vue.js unterstützen. Für die serverseitigen Anwendungen ist der *Backend-Agent* verantwortlich und verwaltet die Logik der Anwendung. Er könnte mit Frameworks wie Node.js, Express und Django umgehen. Um die Anwendung mit einer Datenbank auszustatten, kann ein *Datenbank-Agent* eingesetzt werden. Er kennt verschiedenen Datenbanken wie MySQL oder PostgreSQL. Dieser verwaltet die Datenbank und deren Abfragen. Der *Test-Agent* testet die Anforderung die von durch den Frontend-, Backend- oder Datenbank-Agent umgesetzt wurden.

Ein letzter wichtiger Agent könnte noch der NLP-Agent sein. Dieser Agent nimmt natürliche Sprachanweisungen und Anforderungen entgegen, übersetzt diese in technische Anforderungen als Prompt für die Sprachmodelle. Die Ergebnisse der Bearbeitung werden zum Schluss von dem Agenten in eine vom

Menschlichen verständliche Sprache überführt und zurückgegeben.

2.4.2 Multi-Agenten-Systeme

Multi-Agenten-Systeme (MAS) bestehen aus mehreren Agenten. Ein Agent ist eine autonome Einheit, die mit der Umwelt interagieren kann. Im Gegensatz zur Orchestrierung sind Multi-Agenten-Systeme in ihrer Steuerung dezentralisiert. Alle Agenten haben unterschiedliche Lösungsansätze für ein Problem. Je nach deren Fähigkeit hat dieser auch seine ganz eigenen Ziele, welche zu den anderen Agenten entweder kollaborativ oder kompetitiv ausgerichtet sind. Die Hauptarbeit zur Lösungsfindung eines Problems übernimmt der Agent, mit dem besten Lösungsansatz für das Problem. Die anderen Agenten können den ausführenden Agenten unterstützen. Um die beste Lösung zu finden, müssen die Agenten untereinander kommunizieren. Teil der Kommunikation kann es sein, einfache Informationen austauschen, um eine gemeinsame Strategie fest zulegen oder um zu Verhandeln, welcher Agent die Lösung eines Problems übernimmt.

Im Bereich der Webentwicklung mit MAS, könnte ein derartiges System wie folgt aussehen. Ein *Frontend-Agent* ist für das Design und die Benutzeroberfläche verantwortlich. Hierbei erzeugt dieser Agent Ausgaben in HTML, JavaScript und CSS um die Oberflächen zu erstellen. Dazu kann der Agent Frameworks, wie React verwenden und auf externe Designer Tool zugreifen. Ein weiterer Agent ist der *Backend-Agent*, welcher für die serverseitige Anwendung zuständig ist. Er erstellt seine Funktionen in PHP, Python oder NodeJS. Der Backend-Agent hat Zugriff auf Frameworks und externe Bibliotheken der gewählten Programmiersprache. Er erstellt und verwaltet zudem die Datenbankoperationen (CRUD-Operations). Hinzu kommt noch ein *Test-Agent*, welcher automatisierte Tests durchführt. Um die Funktionalität der Anwendung zu gewährleisten, arbeitet der Test-Agent mit dem Frontend- und Backend-Agent eng zusammen. Der Test-Agent stellt sicher, dass jegliche Codeänderung getestet wird und führt Unit-, Inetragtions- und End-to-End-Tests durch. Wird ein Fehler festgestellt, kann der Test-Agent ein Ticket erstellen oder direkt mit dem Frontend- oder Backend-Agenten kommunizieren.

Ein weiterer Agent könnte ein *Deploment-Agent* sein. Dieser führt automatische Depolyments in verschiedene Umgebungen (QA, Test oder Produktion) durch. Er ist in den Continuous Integration (CI) und Continuous Deployment (CD) Workflow integriert, welche die Bereitstellung auf verschiedenen Servern (VMware, Bare-Metal) und Cloud-Umgebungen (AWS, Azure, Google) bewerkstelligt. Des weitere könnten beispielsweise Security-Agent, Monitoring-Agent und Optimierungs-Agent Einsatz finden.

Auch hier kann ein NLP-Agent zum Einsatz kommen und die Kommunikation zwischen Mensch und System managen.

2.5 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.

2.5.1 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.

Prompttemplates

Um die Eingaben für Programmierer oder Nutzer und dessen Anfragen an LLMs zu vereinfachen, können Propmt-Templates verwendet werden. Diese eignen sich im besonderen Maße für automatisierte Prompts.

```
1  Erstelle eine [Programmiersprache] Funktion namens [Funktionsname], die
    [Beschreibung der Funktion]
    . Die Funktion soll folgende Parameter verwenden:
    [Liste der Parameter]
    . Die Funktion soll folgende Bedingungen erfüllen:
6  [Liste der Bedingungen: z.B. Fehlerbehandlung, bestimmte Algorithmen]
    . Die Ausgabe der Funktion soll in folgendem Format sein:
    [Erwartetes Ausgabeformat]
```

2.5.2 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.

2.6 Grundlagen der Webentwicklung

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

2.6.1 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. Die meisten visuellen Anwendungen erstellen HTML (**H**yper**T**ext **M**arkup **L**anguage) Code als Grundgerüst und generieren CSS (**C**ascading **S**tyle **S**heets) Dateien für das Layout, die als Standardformatierungssprache gilt. Anwendungen die als RestAPI (**A**pplication **P**rogramming **I**nterface) fungieren liefern meist Ausgaben in Form von JSON (**J**ava**S**cript **O**bject **N**otation) aus. Neben JSON Format gibt es weitere beispielsweise XML (**X**ML **A**in't **M**arkup **L**anguage) oder YAML (**Y**AML **A**in't **M**arkup **L**anguage).

2.6.2 Entwicklung

Bei der Entwicklung von Webseiten werden längst schon die selben Prozesse und Tools verwendet wie bei anderen Softwareprojekten. Auch hier finden Tolls wie GitLab¹ und Jenkins² Anwendung. Gerade in der Entwicklung von cloudbasierten Anwendungen kommen Containertools wie Docker³ in Verbindung mit Kubernetes⁴ zum Einsatz. Diese Tools lassen sich hervorragend in CI/CD Pipelines integrieren. An deren Anfang steht auch hier der Entwickler, welcher durch KI Unterstützung erhalten kann.

¹Gitlab ist eine webbasierte Anwendung die Issue-Traking, CI/CD Pipelines, Dokumentation und mehr für Entwickler anbietet.

²Jenkins ist ein webbasiertes Tool für die kontinuierliche Integration welches viele Build-Tools, wie Ant und Maven integriert, Testtols wie JUnit und Emma bietet, sowie Verwaltungssystem wie CVS, Subversion und Git unterstützt. Jenkins kann durch viele Plugins erweitert werden.

³Durch die Containerisierung mit Docker können Anwendungen und deren Umgebungen einfach bereitgestellt und bei bedarf skaliert werden. Docker bietet eine Vielzahl von einsatzbereiten Container an, die einzeln oder in Clustern laufen können.

⁴Kubernetes ist Orchestrierungstool für Dockercontainer das von Google entwickelt wurde. Neben den Container-Anwendungen verwaltet Kubernetes auch die Umgebung für Container, wie beispielsweise Netzwerke.

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 Werkzeug welches die Entwicklung effizienter gestalten kann.

Die Codegenerierung für Designelemente kann ebenso mittels NL2Code erfolgen wie komplexe Backend-funktionalitäten. Ebenso kann die vorherige Konzeption durch eine LLM erfolgen.

2.7 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.

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.

2.7.1 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.

Die pass@k Methode 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 HumenEval-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 für die Codegenerierung von webbasiertem Code festgehalten. Dies umfasst die Festlegung der verwendeten LLMs, die geprüfte Programmiersprachen, Framework zur Erstellung und Auswertung der Tests und Systeme für die Bereitstellung und Verarbeitung der Ergebnisse. Die Evaluierung der Modelle erfolgt in deutscher Sprache, was die Prompts und die Tests betrifft. Allein die Methodenbezeichnung ist in englischer Sprache.

3.1 Definition der Evaluierungsziele

Ausgehend von den in Kapitel 1.4 gestellten Ziele dieser Arbeit, werden hier das Konzept und das Design für die Evaluation beschrieben. Wie werden die Ziele erreicht und welche Komponenten sind dazu erforderlich, um die Daten zu erheben und auszuwerten.

Die Evaluation soll zeigen, ob die generierten Codes korrekt funktioniert und es nicht zu Laufzeitfehlern oder Deadlocks kommt. Des Weiteren wird evaluiert, ob der Code den gängigen Programmierstil entsprechen und welche Qualität der generierte Code ausweist. Hier liegt vor allen der Fokus auf Lesbarkeit und Einhaltung von Codingstandards. Als letzter Punkt soll die Dokumentation innerhalb des Codes geprüft werden. Je besser diese Ausfällt je leichter sind spätere Refactorings möglich.

Was nicht im Fokus liegt und in den Test vernachlässigt wird, sind die Performance der Modelle und die Architekturen der Systeme auf denen die Modelle laufen. Es geht in erster Linie um den generierten Code und dessen Brauchbarkeit für die reelle Verwendung in der Praxis.

Für die Messung der Fehler wird die `pass@k` Methode angewandt. Diese Methode baut auf einen mitgelieferten Test auf, der für jedes Problem vorhanden ist. Der Test, zusammen mit dem generierten Code ergeben einen ausführbaren testbaren Code. Jeder Test wird in einer Liste notiert und dazu das Ergebnis ob positiv oder negativ. Daraus kann dann mit der `pass@k` Methode die repräsentative

Zuverlässigkeit des Modells für jedes Problem und anschließend für das gesamte Modell errechnet werden. Während die Evaluierung der Model mit dem Aufgaben des HumanEval-XL erfolgten, wird die Optimierung der Prompts an eigen erstellte Aufgaben erfolgen. Diese sind komplexer als die allgemeinen Aufgaben aus dem HumanEval-XL Benchmark. Hierbei soll untersucht werden, mit welchem Ansätzen Prompts im Bereich der Codegenerierung im Bereich der Webprogrammierung erfolgen kann. Der Aufbau der Aufgaben orientiert sich am HumanEval Benchmark. Es wird neben der Aufgabe auch ein Unittest vorgegeben. Somit kann der generierte Code mit Unittests geprüft werden. Um diese Test ausführen zu können, sind PHP Dateien erforderlich. Somit müssen die generierten Codes in Dateien gespeichert werden und können im Anschluss geprüft werden.

3.2 Auswahl der LLMs und deren Konfiguration

Für die Evaluation werden experimentell einige 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 Referenz soll das kommerzielle Modell *Gemini 1.5* dienen, welches durch stetige Verbesserung und einer großen Nutzerzahl erstellt wurde.

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 und bietet verbesserte Fähigkeiten zur Codegenerierung und -optimierung. 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 in beiden Arbeiten das Modell mit verschiedenen Open-Source und Close-Source Modellen verglichen.

Die Modelle **Llama 3.1-Claude** und **Llama 3.1** gehören mit 8 Milliarden Parametern zu den kleineren Modellen von MetaAI. 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 unter [24] zu finden.

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 [25] wurde Mistral, mit verschiedenen Modelle zur Spielecodegenerierung verglichen. Während in [26] eine Evaluation für natürlichsprachlicher Erklärungen, Mistral mit anderen Modellen verglichen wurde.

Mit **Gemini 1.5** präsentiert Google ein Modell zur Verarbeitung von natürlicher Sprache und stellt es zur freien Nutzung zur Verfügung. Genau wie ChatGPT nutzt auch Google die Nutzereingaben, um neue Modelle zu trainieren, was zur Weiterentwicklung für und somit zum neuen Modell **Gemini 2**. Wie in [27] beschrieben, setzen auch die Gemini Modelle die Transformer-Architektur ein, was sie dazu befähigt, komplizierte Sprachmuster zu erkennen und präzise Vorhersagen zutreffen. In [27] wird Gemini 1.5 mit Aufgaben zur Codegenerierung mit ChatGPT und Copilot vergleichen. Nach [28] kann sich das Gemini-Ultra-Modell beim MMLU-Benchmark sogar mit menschlichen Experten messen und erschließt eine breite Palette von Anwendungsbereichen. Hier wurden die Fähigkeiten zur Codegenerierung an Sicherheitsfragen im E-Commerce Bereich getestet. Ein Überblick über die Gemini Modelle ist unter [29] zu finden.

Neben den genannten Quellen sind die Herstellerseite eine gute Quelle weiterführende Informationen einzuholen.

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
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	43 GB	DE/EN	X	Ollama
Codellama	13b	q4_0	7,4 GB	DE	X	Ollama
Mistral Small	22b	q4_0	12 GB	DE	X	Ollama
Gemini 1.5 Pro	k.A.	k.A.	k.A.	DE	-	online

Tabelle 3.1: Auswahl der LLMs für die Evaluierung

Die Einstellung für die Abfragen der Probleme wurden bei allen Modellen identisch gewählt.

Für die Abfragen der Testprobleme wurde eine *temperature* von 0.2 gewählt. Ein niedriger Wert veranlasst die Modelle deterministischere und standardisierte Antworten zu geben und verhindert Kreativität und Zufälligkeit. Die Generierung von Programmcode soll konsistenten und präzisen Code liefern.

Ein hoher *top_p* Wert verlangt von den Modellen eine Antwort die mit hoher Wahrscheinlichkeit richtig ist. Für die Codegenerierung sollten die wahrscheinlichsten und syntaktisch korrekten Token angewandt werden. Für die Abfragen wird hier ein Wert von 0.95 angesetzt.

Die maximale Anzahl Token sollte bei der Generierung von Code zwischen 20 und 1000 Token eingestellt werden, je nach Umfang der Antworten. Da hier nicht nur die Funktionsfähigkeit geprüft wird, sondern auch Struktur und Coding-Standards wird ein hoher Wert ausgewählt, sodass beispielsweise Kommentare ebenfalls enthalten sind können. Somit wird *max_token* auf sechshundert festgelegt.

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

Modell	Temp.	max. Token	Top-p
Qwen2.5-coder	0.2	600	0.95
Deepseek-Coder-v2	0.2	600	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
Mistral Small	0.2	600	0.95
Gemini 1.5 Pro	k.A.	k.A.	k.A.

Tabelle 3.2: Einstellungen der Modellparameter

Hinzu kommen weitere Parameter für die Modelle. 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.

Alle Prompts welche die Probleme enthalten, werden mit Python Skripten abgefragt. Dies gilt für die offenen lokalen wie auch für die kommerziellen Modelle. Die Abfragen werden jeweils an die entsprechenden APIs abgesetzt. Als Framework für die Abfragen der lokalen Modelle, kommt das Python *langchain* Framework zu Einsatz. Für die Modelle der Gemini-Reihe wird die Google eigene Bibliothek, *google.genai* verwendet. Dasselbe trifft für die Modelle von OpenAI zu. Hier kommt die Python Bibliothek *openai* zum Einsatz.

3.3 Design der Experimente

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

- **task_id**: Kennung der Datenprobe
- **prompt**: Anfrage für das Modell, Funktionsheader und Docstring
- **entry_point**: Einstiegspunkt für den Test
- **test**: Test für die geforderte Funktion
- **description**: Beschreibung der Aufgabe
- **language**: Kennung der Programmiersprache
- **canonical_solution**: Lösung für das Problem
- **natural_language**: Ländersprache in der Datei

Die Abbildung 3.1 zeigt den Aufbau und damit alle wichtigen Bereiche des Benchmark-Tests.

Um die Modelle zu evaluieren und ihre Fähigkeiten hinsichtlich der im Web vorherrschenden Programmiersprachen zu untersuchen, werden die Tests in den Programmiersprache(n) PHP (und JavaScript) vorgenommen. Dafür sollen die Modelle mehrfach einfache Funktionen generieren.

Der Benchmark liefert die Tests mit. Dazu werden bereits in den Prompts die Namen der Methoden und die zu übergebenen Parameter angegeben, welche zu erstellen ist. Der jeweilige Test verwendet dann diesen Namen und übergibt die geforderten Parameter. Das Listing 3.1 zeigt ein Beispiel für einen mitgelieferten HumanEval-XL Test.

```
function compare($x, $y) {  
2   return $x == $y;  
}  
$arg00 = [3, 1, 2, 4, 5];  
$x0 = median($arg00);  
$v0 = 3;  
7 if (!compare($x0, $v0)) {  
    throw new Exception("\Error at 1th assert statement.\");  
}
```

Listing 3.1: Beispiel für einen Test aus dem HumanEval-XL Benchmark

Um die Modelle untereinander zu vergleichen, bekommen alle Modelle dieselben Prompts. Von jedem Prompt werden pro Modelle zehn Varianten erstellt. Die Ergebnisse werden in einer Liste chronologisch gespeichert. Diese Ergebnisse werden dann mittels der `pass@k` Metrik geprüft.

HumanEval-XL Benchmark

Tasks: Generation 1 2 >> 3 (Test: 1 2 3 4)

<?php	Deklaration ①
<pre>/** * Sie sind ein erfahrener PHP-Programmierer und hier ist Ihre Aufgabe. * ... */</pre>	Kommentar ②
<pre>function belowZero(\$operations) { // Initialize the current balance to zero \$balance = 0; // Iterate over each operation in the list foreach (\$operations as \$operation) { // Add or subtract the operation amount from the balance \$balance += \$operation; // If the balance goes below zero, return True immediately if (\$balance < 0) { return true; } } // If we've iterated over all operations and never gone below zero, // then return False return false; }</pre>	Ergebnis ③
<pre>function compare(\$x, \$y) { return \$x == \$y; } \$args00 = []; \$x0 = belowZero(\$args00); \$v0 = false; if (!compare(\$x0, \$v0)) { throw new Exception("Error at 1th assert statement."); ... }</pre>	Test ④

PHP (Problem 0)

Abbildung 3.1: Codegeneration

3.4 Konzeption des Prompt-Engineerings

Die Prompts im HunamEval-XL Benchmark sind als Few-Shot-Prompts verfasst. Sie neben der eigentlichen Aufgabe sind noch Beispiel für die Eingabedaten und erwarteten Ergebnisse angegeben. Das Listing 3.2

zeigt ein Beispiel für einen Prompt.

```
1 <?php
    /**
    * Sie sind ein erfahrener PHP-Programmierer und hier ist Ihre Aufgabe.
    * Gib den Median der Elemente in der Liste l zurück.
6 * >>> median([3, 1, 2, 4, 5])
    * 3
    * >>> median([-10, 4, 6, 1000, 10, 20])
    * 15.0
    *
11 */
function median($l){
```

Listing 3.2: Prompt Beispiel für eine Aufgabe aus dem HumanEval-XL Benchmark

Alle Prompts im Benchmark sind als Code-Kommentare aufgebaut. Als letzte Zeile ist der Methodenname angegeben. Somit soll sichergestellt werden, dass die erstellten Tests funktionieren.

3.5 Evaluationsumgebung

Die freien Modelle laufen auf einem Debian 12 System, welches mit 16 CPUs und 32 GB RAM ausgestattet ist. Um zusätzlichen Speicher zu erhalten, wurde eine 100 GB Swap Partition genutzt. Für die Bereitstellung ist das freie Framework Ollama zum Einsatz gekommen.

Die Hardware Auf die kommerziellen Modelle kann kein Einfluss auf die Systeme genommen werden.

Alle erstellten Codes und Ergebnisse werden dokumentiert und im Anhang eingefügt, sodass eine die Evaluierungen nachvollzogen werden können. Des Weiteren werden alle Daten und Dateien unter <https://github.com/willi-pahl/master-thesis> bereitgestellt.

Mein roter Faden

Hier kommen noch Optimierungsangaben, die stehen zurzeit nicht fest.

IMPLEMENTIERUNG

4.1 Lokale Modelle

Für das Ausführen von Modellen zum Testen werden in dieser Arbeit zwei Techniken angewandt. Zum einen mittels Ollama Framework, das mit einer Web-GUI erweitert werden kann, zum anderen durch Dateien, welche beispielsweise mit dem Python Framework Langchain abgefragt werden können.

Die Modelle werden auf einem Debian 12 Server mit 32 GB RAM und einem 16 Kernprozessor ausgeführt. Um große Modelle zu testen wurde mit einer Swap-Partition von 100 GB gearbeitet, um die Ausführung größerer Modelle zu ermöglichen.

Für das Testen der lokalen Modelle wird das Ollama Framework angewandt. Dies ermöglicht eine neben einer Benutzeroberfläche im Browser eine Anbindung an einer API. Diese lässt sich beispielsweise mittels Python abfragen. Auf dieser Weise 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 es können Interaktionen mit dem Modell erfolgen.

Zusätzlich kann ein grafisches Tool zum Testen installiert werden. Mit deren Hilfe können die Modelle leicht getestet werden. Mit Open WebUI wird ein Browser basierendes Tool eingesetzt, dass auf dem Ollama-Server installiert 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.

4.1.1 Ergebnisse generieren

Nachdem die Modelle bereitstehen, erfolgt das Generieren der Ergebnisse für jedes einzelne Modell. Mit Prompts, die aus den Aufgaben des HumanEval-XL Benchmark bestehen, werden nun die Modelle mehrmals hintereinander abgefragt. Die vollständig generierten Antworten der Modelle werden, für eine spätere Auswertung im JSONL-Dateiformat gespeichert. Für jedes Problem erfolgen fünf Abfragen an

jedes Modell. Für die Evaluierung der lokalen Modelle wurden Modelle von Ollama verwendet, siehe Kapitel 3.2 für die Auswahl der Modelle.

4.2 Benchmark Codeevaluation

Die Auswertung der Antworten wird für jedes Modell angepasst, da die Antworten von Modell zu Modell variieren. Hier werden unterschiedliche Möglichkeiten für Codesnippets angewendet, beispielsweise (`““php`) oder (`““php \n <?php`), welche für die Ausführung anzupassen sind. Somit wird immer nur die Methode zur Extrahierung des Codes aus den Antworten geändert, siehe Beispielcode im Listing 8.7.

Mit den vorliegenden Codesnippets und den Tests aus dem jeweiligen Problem, wird ein ausführbarer PHP-Code erstellt und mittel Python ausgeführt. Die Tests werfen Exceptions, wenn der Code die Anforderungen nicht erfüllt oder Laufzeitfehler auftreten. Dann gibt die Aufgabe als nicht bestanden. Das Listing 4.1 zeigt den Ausschnitt im Code zur Ausführung des PHP Interpreters in Python.

```
for answer in answers:
    answer = answer.replace(r"\n", "\n")
3     try:
        result = subprocess.run(
            ["php", "-r", f"{test}{answer}"],
            capture_output=True,
            text=True,
8            check=False,
            timeout=5,
        )
    except subprocess.TimeoutExpired:
        pass
13

    if result.stderr.strip() == "":
        answer_result_fault = True
```

Listing 4.1: Codesnippet zur Ausführung des PHP Interpreters

Die unbehandelte Exception in Zeile 12, führt lediglich dazu, dass die Programmausführung nicht unterbrochen wird. Ein abgelaufener *Timeout* deutet auf eine Endlosschleife hin und somit gilt dieser Test als nicht bestanden. Der hier abgefangene *Timeout* wird in der nachfolgenden *if* Konstrukte mit ausgewertet. Aus den erhaltenen Ergebnissen, berechnet die *pass@k* Methode eine Vorhersage zur Zuverlässigkeit des jeweiligen vorgegebenen Problems, hinsichtlich Codegenerierung. Anschließend wird der Durchschnitt für die Zuverlässigkeit des Modells errechnet.

Umsetzung der pass@k Metric

Die Auswertung der Ergebnisse erfolgt mittel Python. Dazu wird der generierte Code mit den vorgegebenen Tests aus dem HumenEval-XL zusammengefügt. Der so erzeugte Code wird dann in der Python-Umgebung ausgeführt. Die Ergebnisse aus den Tests werden dann mittel pass@k Methode analysiert. In Python steht, hierfür die Bibliothek *pass_at_k* zur Verfügung. Das Listing 4.2 zeigt Umsetzung der Methode. Der PHP Code wurde auf einen Debian 11 mit PHP in der Version 8.2.26 evaluiert.

```
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 4.2: Berechnung der pass@k Metrik in Python

4.3 Codeevaluation mit Frameworks

Neben dem bekannten Evaluationen mit beispielsweise dem HumanEval Benchmark, wird hier eine weitere Testmethodik überprüft, die mit verschiedenen Validierungstools der jeweiligen Programmiersprache ausgeführt wird. Für die Erstellung der Abfragen wird das Python-Skript verwendet, was schon im Kapitel 4.2 vorgestellt wurde.

4.3.1 PHP Codeevaluation

Der Test wird bei den erweiterten Problemen durchgeführt und beginnt mit den Unit-Tests die mit *PHPUnit* durchgeführt werden. Im Anschluss wird *PHPMetrics* ausgeführt. Hierbei wird geprüft, ob die Codekomplexität und Wartbarkeit überprüft. Sind diese Tests bestanden, wird der Code noch gegen eine *SonarQube* Server validiert. Die Ausführung der Tests wird mithilfe eines Python-Skripts durchgeführt. Es wird eine PHP Datei erstellt, die mit den Frameworks geprüft wird.

4.4 Optimierung der Antworten

Mein roter Faden

Mögliche Optimierungsstrategien

- mit dem Model Programming (DSPY) Ansatz: Python Bibliothek vorhanden pypi.org | DSPy.

EVALUATION

Die Evaluation der Ergebnisse erfolgt im ersten Schritt anhand des HumanEval-XL Benchmarks. Dieser Benchmark wird in [17] vorgestellt und erweitert den HumanEval [16]. Der HumanEval-Benchmark evaluiert nur Python während der HumanEval-XL weitere Programmiersprachen und in verschiedenen Landessprachen unterstützt, darunter auch die deutsche Sprache. Neben Python sind auch Prompts für PHP und JavaScript enthalten, welche für die Webentwicklung wichtig sind. Die Datensätze des HumanEval-XL sind unter <https://github.com/FloatAI/humaneval-xl> einsehbar und bestehen jeweils aus 80 Tests. Für jedes Problem werden zehn Lösungsvorschläge generiert, die im Anschluss auf die Aspekte der Syntaktik und Semantik evaluiert werden.

5.1 Modellbewertung mit HumanEval Benchmark

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.

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

Dabei ist n die Gesamtanzahl der Versuche, c die Anzahl der korrekten Lösungen unter den n Versuchen und k gibt die Anzahl der Lösungen an die betrachtet wurden. Für die Berechnung der $pass@k$ Metrik wird die Formel 2.3 verwendet, welche in [16] vorgeschlagen wird.

Für alle Probleme wurden jeweils zehn Abfragen erstellt und bewertet. Welche Modelle getestet an der

Evaluation beteiligt waren und welche Ergebnisse ermittelt wurden, wird in Tabelle 5.1 gezeigt.

Model	pass@1	pass@5
Llama3.3:70b	0,575	0,7
Deepseek-coder-v2:	0,53	0,5875
Gemini Flash 1.5	0,5	0,55
Llama3.1:8b (T600)	0,45	0,5302
Llama3.1:8b (T1200)	0,45	0,53472
Llama3.1-claude:8b	0,375	0,508929
Llama3.2:3b	0,275	0,4114
Mistral-small 22b	0,2125	0,3061
Qwen 2.5-Coder 32b	0,1875	0,239
Codellama:13b	0,0	0,025

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

Das Modell `code1lama` wurde ebenfalls getestet, hat aber beim Generierung von den Lösungen der PHP Probleme nicht gut abgeschnitten. Viele der Anforderungen wurden in Python erstellt und viele Tests sind als nicht bestanden gewertet wurden. Aus diesem Grund wird es bei den weiteren Betrachtungen nicht mehr beachtet.

Des Weiteren wurden eine Reihe von Llama-Modellen getestet, unter anderem einige verschiedene Llama3.1 Modelle. Bei dem *Llama3.1 8b* wurden zwei verschiedene Versuche durchgeführt mit jeweils unterschiedlicher Tokenlängen. Anders als bei *ChatGPT 3.5* und *Gemini 1.5* können die kleineren Modelle schon mit geringerer Tokenlänge valide Antworten erzeugen. Die unterschiedliche Tokenlängen haben kaum einen Einfluss auf das Ergebnis. Hier liegt der Unterschied bei nicht mal einen Prozent.

Ein Ansatz zur Promptoptimierung, wurde bei den Tests ebenfalls evaluiert. So wurde ein Llama3.1 Modell mit den Systemprompts des *Claude Sonnet 3.5* Modell von der Firma Anthropic erstellt. Dieser Ansatz konnte mit der vorliegenden Evaluierung nicht bestätigt werden. Der Unterschied liegt hier bei plus einem Prozent bei der pass@1 Evaluierung und minus drei Prozent bei der pass@5 Evaluation. Mögliche Ursachen hierfür könnten in einer fehlerhaften Erstellung des Modells oder die Aufforderungen sind nicht kompatibel mit einem Llama3.1 Modell. Eine weitere Möglichkeit sind unzureichendes Wissen über die Architektur des Quellmodells oder fehlendes implizite Informationen im Zielmodell. Ebenso könnte die Unbrauchbarkeit der Systemprompts für Codegenerierung die Ursache sein. Warum es keinen signifikanten Unterschied gab, könnte in dieser Arbeit nicht abschließen geklärt werden. Die Ergebnisse sind in Tabelle 5.1 und in der Abbildung 5.1 zu sehen.

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 0,12 und 0,7 beim pass@5 liefert dieses Modell die

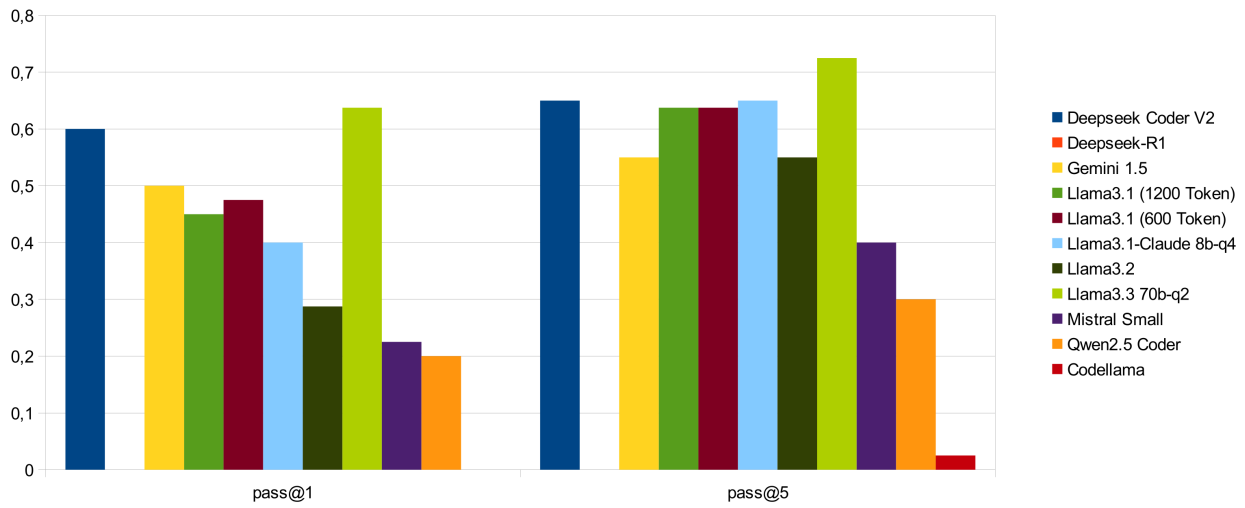


Abbildung 5.1: Ergebnisse der pass@k-Methode für die Modelle

besten Ergebnisse. Bei dem vorliegenden Modell handelt es sich um ein Modell mit Quantisierung, was die Größe des Modells auf rund 26 GB reduziert. Bei diesem Modell wurde einige Komponenten einer 2-Bit Quantisierung unterzogen, bei einer fast gleichbleibenden Ausgabenqualität, weitere Informationen sind [30] zu entnehmen. Das originale Modell mit ca. 43 GB ließ sich nicht mit den verfügbaren lokalen Ressourcen testen.

Zusätzlich wurden zwei weitere Modelle aus dem Open-Source-Bereich in die Evaluation einbezogen: *Mistral-Small 22b*, entwickelt von der Firma MistralAI, sowie *Qwen 2.5 Coder 32b*. Die Leistung dieser beiden Modelle fiel im Vergleich zu anderen Modellen jedoch geringer aus. Insbesondere lagen die Trefferquoten von *Mistral-Small 22b* und *Qwen 2.5 Coder 32b* bei etwa der Hälfte der durchschnittlichen Trefferquoten der getesteten Llama-Modelle.

Das Modell *Deepseek Coder V2 16b* welches an die Leistung von den Llama3.1 Modellen heranreicht und ist ebenfalls aus dem Open-Source-Bereich.

Ein weiteres Modell ist Gemini 1.5. von Google und gehört zu den Cloused-Source-Modellen.

Einige Aufgaben wurden von Gemini aber wie ein Chat behandelt. So hat das Modell Antworten generiert, die einer Konversation mit einem Chatbot ähneln, so wurde beispielsweise der folgende Text generiert, der einen Bezug zur zuvor gegebenen Antwort des Modells herstellt.

```
function greatestCommonDivisorRecursive($a, $b) {
    // ... (Rest der Funktion bleibt ähnlich)
```


}

Generiert von Gemini 1.5

Diese generierten Codeausschnitte haben die Tests nicht bestanden und wirken sich somit negativ auf die Auswertung aus.

5.1.1 Evaluierung nach Sprache

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

Da das Training der Modelle in englischer Sprache erfolgt, lag die Vermutung nahe, dass eine Evaluierung in dieser Sprache, wesentlich bessere Ergebnisse liefern sollte. Dies zeigen auch die Auswertungen in [17, S. 11] (Tabelle 12), bei denen die Programmiersprache PHP in der englischen Version im Durchschnitt besser abschnitt.

Diese Vermutung konnte mit den durchgeführten Tests der ausgewählten Modelle nur bedingt bestätigt werden. Das Llama3.3 zeigt besser Werte, in der englische Version. Während das Llama3.2 Modell eine bessere Trefferquote in Deutsch bei steigendem k-Wert für die $\text{pass}@k$ -Methode zeigt. In der Abbildung 5.2 werden die Ergebnisse der $\text{pass}@k$ -Methode mit $k = 1 \dots 5$ für die ausgewählten Llama-Modelle zusammengefasst.

Dieser Test zeigt auch das eine deutschsprachige Verwendung der Modelle nicht zwangsläufig schlechteren Code hervorbringt. Daraus wird der Schluss gezogen, das bei der Optimierung der Prompts eine Übersetzung ins Englische nicht unbedingt zu einer signifikanten Verbesserung führt.

5.1.2 Nachteile der Evaluierung

Trotz seines Einsatzes bei der Evaluierung von Modellen, zeigt der Benchmark-Test einige Fehler. Dadurch resultieren Nachteile, die sich bei der Bewertung der Modelle negativ auswirken.

Einige **Fragen sind nicht eindeutig formuliert** und können von den Modellen falsch interpretiert werden. Ein Beispiel im HumenEval-XL Benchmark ist die Aufgabe 20. Der genaue deutsche Wortlaut ist wie folgt, *Überprüfen Sie, ob zwei Wörter dieselben Zeichen enthalten.*, der gesamte Prompt ist in Listing 5.1 zu sehen. In der hier durchgeführten Evaluierung wurde diese Aufgabe im $\text{pass}@1$ nur einmal bestanden. Der

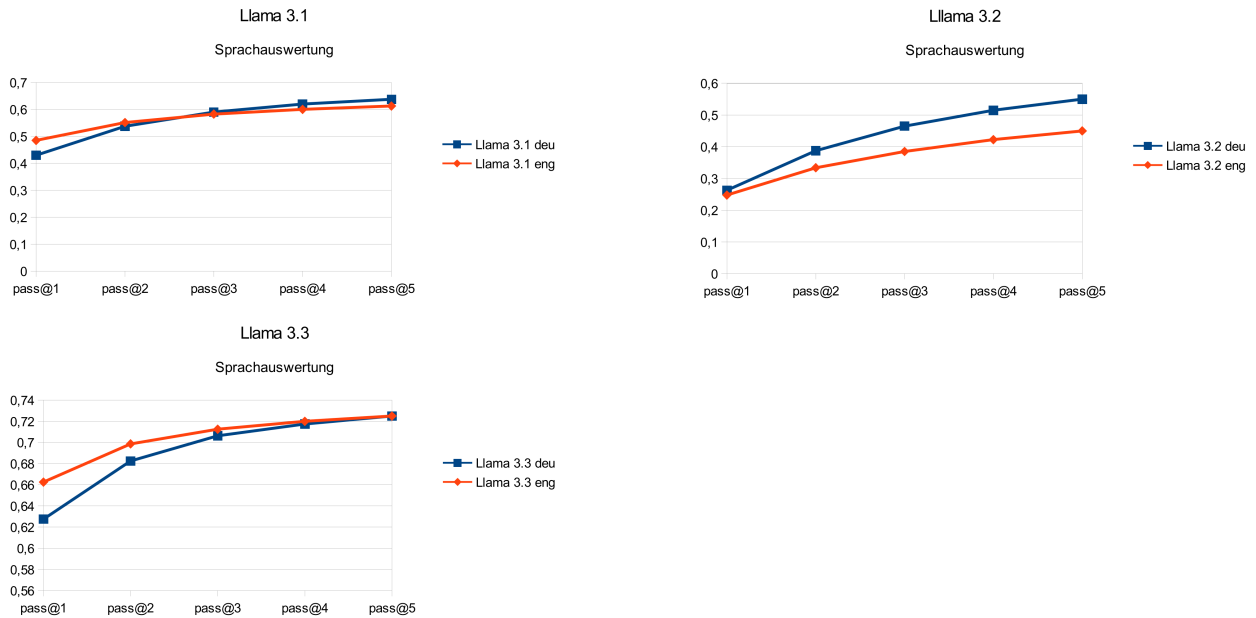


Abbildung 5.2: Ergebnisse der pass@k-Methode für Llama Modelle verschiedener Sprachen

Wortlaut hält die Modelle an, einen Vergleich in der Methode selbst durchzuführen. In der, durch den Benchmark vorgegebenen Test, werden aber zwei Zeichenketten verglichen. D.h. als erwartetes Ergebnis sollte eine Zeichenkette erstellt werden, welche die doppelten Zeichen zweier zu testenden Zeichenketten enthält. Das Listing 5.2 zeigt einen Teil des Tests und die erwarteten Ergebnisse.

```

1 <?php
  /**
   * Sie sind ein erfahrener PHP-Programmierer und hier ist Ihre Aufgabe.
   * * Überprüfen Sie, ob zwei Wörter dieselben Zeichen enthalten.
   * >>> same_chars('eabcdzzzz', 'dddzzzzzzzddedddabc')
6 * True
   * >>> same_chars('abcd', 'dddddddabc')
   * True
   * >>> same_chars('dddddddabc', 'abcd')
   * True
11 * >>> same_chars('eabcd', 'dddddddabc')
   * False
   * >>> same_chars('abcd', 'dddddddabce')
   * False
   * >>> same_chars('eabcdzzzz', 'dddzzzzzzzdddabc')
  
```

```

16 * False
   */
   function sameChars($s0, $s1){

```

Listing 5.1: Wortlaut der Aufgabe 20 im HumalEval-XL Benchmark

```

function compare($x, $y) {
2   return $x == $y;
}

$arg00 = "eabcdzzzz";
$arg01 = "dddzzzzzzzzddeddabc";
7 $x0 = sameChars($arg00, $arg01);
  $v0 = true;
  if (!compare($x0, $v0)) {
    throw new Exception("Error at 1th assert statement.");
  }

```

Listing 5.2: Wortlaut des Tests 20 im HumalEval-XL Benchmark

Eine weitere Fehlerquelle ist das ***Zusammenführen der Antworten der Modelle und Tests aus dem Benchmark***. Meist ist das Problem 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 den Einsatz einer weiteren Pythoncodezeile, wie in Listing 5.3 zeigt konnten die Bewertung einiger Modelle verbessert werden. So hat sich die Bewertung für das *pass@1* des Deepseek-Coder-V2 um 7% von 0,53 auf 0,6 verbessert. Ebenso wurde für das Llama3.1 Modell eine Verbesserung von 0,45 auf 0,475 ermittelt.

```

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

```

Listing 5.3: Fehler bei der Auswertung durch fehlerhafte Anführungszeichen

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. Bei der Formulierung der Tests könnten Randbedingungen nicht korrekt betrachtet wurden sein, was dazu führen

kann, dass der Einsatz der generierten Codes zu Fehlern führen könnte. Ebenfalls können die Tests falsche Parameter vorgeben, wodurch korrekt generierter Code als falsch bewertet wird. Die eben genannten Fehler konnten im vorliegenden Ergebnissen nicht nachgewiesen werden, ganz auszuschließen sind sie aber nicht.

5.2 Optimierung der Ergebnisse

Als Ziel der Optimierung gilt das die LLMs effizienten, präzisen und korrekten Code zu generieren. Ein Ansatz dies zu erreichen ist die Prompts zur Codegenerierung mithilfe einer LLMs zu erstellen oder zu verbessern.

Mein roter Faden

Hier kommen noch Optimierungsangaben, die stehen zurzeit nicht fest.

LESSONS LEARNED

In diesem werden die Arbeitsprozesse reflektiert und Fehler die während der Arbeit auftraten analysiert. Des Weiteren werden Vorschläge gegeben, um das Arbeiten mit Sprachmodellen zu verbessern. Zuerst wird der Evaluierungsprozess betrachtet, um anschließend die Optimierung der Prompts zu diskutieren.

6.1 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.

6.1.1 Lokale Ressourcen

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

Der größere Speicher wurde notwendig während des Laden und Speichern der Modelle auf die lokale SSD. Zudem 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. Dieser wurde auch für die Ausführung größerer Modelle benötigt.

Eine weitere Verbesserung war 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 das Austauschen der Grafikkarte wurde der SWAP für die Berechnungen der Token während die Antwort erstellt wurde nicht mehr benötigt.

Durch diese Anpassungen bestand die Möglichkeit größere Modelle zuladen und bereit zustellt. Des Weiteren konnte eine wesentliche Verbesserung der Antwortzeit festgestellt werden. Eine genaue Messung wurde hier nicht durchgeführt. So wurden bei dem Deekseek-Coder-V2 Modelle eine Verbesserung der Berechnungszeit für den Benchmark von etwa 24 Stunden auf circa eine Stunde beobachtet.

Dennoch war die Berechnungszeit einiger Modelle, welche mehr als 12 GB groß waren sehr hoch. Sodass die Wartezeit, das Auswerten der Evaluierung verzögerte.

6.1.2 Auswertung des Benchmarks

Bei der Auswertung des Benchmarks traten einige Fehler auf, die beseitigt werden konnten. Ein Großteil waren kleinere Fehler, die bei der Auswertung aufgetreten, wie in Kapitel 5.1.2 bereits angesprochen. Der gravierendste Fehler trat aber bei der Berechnung des `pass@k` für das gesamte Modell auf. Hier wurde eine falsche Python-Methode implementiert, sodass das Ergebnis, um ein Zehntel niedriger war, als das wirkliche Ergebnis. Erst durch Vergleiche Ergebnisse anderer Arbeiten und Herstellerangaben, ist der Fehler aufgefallen und nach intensiver Suche wurde dieser Fehler gefunden und beseitigt.

6.2 Erweiterte Codeevaluation

Bei den vordefinierten Prüfungen der HumanEval Benchmarks, wird geprüft, ob der Code lauffähig ist, nicht aber die Codestruktur oder Kommentare. Ein Problem bei der Nutzung des von der LLM generiertem Code ist, dass Entwickler diesen einfach kopieren und in ihre Programme implementieren. Es wird also nur die Funktionalität des Codes geprüft, nicht aber Strukturen und Kommentare um die Lesbarkeit und Verständlichkeit zu erhöhen. Dieses Vorgehen mag zu schnellen Erfolgen in der Programmentwicklung führen, wird aber beim Refactoring oder Fehlersuche erhebliche Defizite mit sich bringen.

Aus diesem Grund sollte der erstellte Code nicht nur auf die Funktionalität geprüft werden. Dafür sollten weitere Test-Frameworks der jeweiligen Programmiersprache zur Anwendung kommen. Es gibt mehrere Frameworks zur Prüfung der Codequalität unter PHP. Zwei bekannte Frameworks die auch in dieser Arbeit Anwendung finden, sind die Frameworks `phpunit` und `phpmetrics`. Mit ihnen wird der, durch die LLMs generierten Codes geprüft.

Um PHPUnit und PHPMetrics für die Evaluierung zu verwenden, müssen weitere Angaben und Einträge im Benchmark erfolgen. So muss ein PHP-Unittest enthalten sein, dieser kann den einfachen benutzerdefinierten Test ersetzen. Des Weiteren sind die Kriterien für die Metrik Messung, für jeden Test erforderlich. Die Kriterien können wie in Listing 6.1 dargestellt, aussehen.

```
1  criteria = {  
    "Lines of code": lambda x: int(x) > 12,  
    "Logical lines of code by method": lambda x: float(x) > 7,  
    "Lack of cohesion of methods": lambda x: float(x) > 3,  
    "Average Cyclomatic complexity by class": lambda x: float(x) > 10,  
6  "Average Weighted method count by class": lambda x: float(x) > 20,  
    "Average bugs by class": lambda x: float(x) > 0.1,  
    "Critical": lambda x: int(x) > 0,  
    "Error": lambda x: int(x) > 0,  
    "Warning": lambda x: int(x) > 0,  
11 "Information": lambda x: int(x) > 0,  
    }
```

Listing 6.1: Beispiel für Bewertungskriterien

Mit den erweiterten Tests werden die Benchmarks, um die folgenden Punkte erweitert.

- **unittest**: Unittests für die geforderte Funktion, unterschied zu den einfachen Tests
- **metrics**: Kriterien für den Metriktest

6.2.1 PHPUnit

Eines der bekanntesten spezielles Framework für Unit-Tests in PHP, was als Industriestandard gilt. Mit diesem Framework können neben der Prüfung auf funktionsfähigen Code auch Randfälle betrachtet und Fehlerbehandlungen im Code getestet werden. Als Grundlage für die Auswahl des Tools wird auf Studie [31] verwiesen.

6.2.2 PHPMetrics

Ein PHP Framework für die Codeanalyse, welches detaillierte Berichte über die Codequalität, Komplexität des Codes und über dessen Wartbarkeit erzeugt. PHPMetrics wird in verschiedenen Arbeiten eingesetzt, um die Codequalität zu ermitteln. So auch in [32], bei der verschiedene Open Source LMS verglichen werden.

6.2.3 SonarQube

Als letztes Tool soll SonarQube zur statischen Codeanalyse und Codeprüfung zum Einsatz kommen. Es werden verschiedene Programmiersprachen unterstützt, darunter auch PHP und JavaScript. In der Arbeit

[33] wird die Prüfung der Codequalität mit SonarQube, ChatGPT3.5 und ChatGPT4 verglichen. Als Schlussfolgerung aus dem Ergebnis dieser Arbeit, wird auch hier die Codeanalyse durch eine LLM nicht erfolgen, sondern ebenfalls durch SonarQube.

DISKUSSION UND AUSBLICK

Mein roter Faden

Struktur des Kapitels

1. **Einleitung:** Eine kurze Einführung in die Diskussion und den Ausblick.
2. **Zusammenfassung der Ergebnisse:** Eine kurze Übersicht über die wichtigsten Ergebnisse und in Relation mit den Forschungsfragen stellen.
3. **Diskussion der Ergebnisse:** Eine Analyse und Interpretation der Ergebnisse. Vergleich mit Stand der Forschung und früherer Arbeiten.
4. **Grenzen und Einschränkungen:** Eine Diskussion der Limitationen der Studie. Z.B. begrenzte Datenbasis, Grenzen der eingesetzter Tools und Technik.
5. **Impulse für zukünftige Forschung:** Vorschläge für weitere Studien. Verbesserungsmöglichkeiten der Methoden usw. und Zukunft des Forschungsfeldes und evtl. Trends.
6. **Praktische Anwendung:** Eine Diskussion der möglichen Anwendungen der Ergebnisse. In welchen Unternehmen und welche realen Anwendungen können die Ergebnisse eingesetzt werden.

Mein roter Faden

Unterschied Diskussion/Ausblick und Fazit

Aspekt	Diskussion und Ausblick	Fazit
Funktion	Kritische Analyse und Zukunftsperspektive	Zusammenfassung und Abschluss
Zeitperspektive	Zukunftsorientiert	Rückblickend
Detaillierungsgrad	Detailreichere Auseinandersetzung mit Ergebnissen	Knapp und prägnant

Während „Diskussion und Ausblick“ die Ergebnisse kritisch reflektiert und auf zukünftige Entwicklungen verweist, fasst das „Fazit“ die Arbeit kompakt zusammen und beantwortet die Forschungsfrage. Beide Kapitel sind komplementär, aber klar voneinander zu unterscheiden.

Wie in [34] beschrieben,

7.1 Impulse für zukünftige Forschungen

Ein interessantes Feld für die Forschung ist die Nutzung generativer KI und welche Auswirkungen dies auf das menschliche Denken und Handeln hat. In der Studie [35] wird von einem System 0 gesprochen, welches neben den bekannten

1. System 1: schnelles, intuitives und automatisches Denken
2. System 2: langsames, analytisches und reflektierteres Denken

eingeführt wird. Hierbei handelt es sich um ein Denken, welches die KI für den Menschen übernimmt. Entscheidungen und Daten werden durch die KI übernommen. Ein externes System, ähnlich wie eine USB-Festplatte eines PCs.

Inwieweit können auch *Small Language Models* für Programmieraufgaben eingesetzt werden. Könnte der enorme Energiebedarf und Ressourcen der LLMs durch SLMs ersetzt werden? Siehe Small Language Models (SLMs) oder Small but Powerful: A Deep Dive into Small Language Models (SLMs). Eine weitere Forschung kann die Evaluation sein, ob Finetuned SLMs, wie Phi-2, Google Gemini Nano oder Metas Llama-2-13b bessere Ergebnisse liefern, als die LLMs.

Ein weiteres Feld kann sich mit der Einführung einer KI in Firmen befassen und Fragen wie,

- Wie können Entwickler bestmöglich vorbereitet werden, um die Einführung von KI reibungslos zu ermöglichen?

- Wie kann Datensicherheit und Datenqualität sichergestellt werden?
- Evaluierung von Kosten/Nutzen für die Einführung von KI in Softwareunternehmen.

evaluieren.

Mein roter Faden: noch was zum Testen

Ein Tool zur Orchestrierung von Multi-Agenten-Systemen OpenAI Swarm, gefunden auf Golem | Karrierewelt.

7.2 Praktische Anwendung

Blaupause für Prompting Das Geheimnis hinter LLM-Halluzinationen [S. 16 ff.] noch testen und evaluieren.

7.2.1 Anwendung für Entwickler

Zur Optimierung des generierten Codes kann auch die freie Wahl der Softwarekomponenten durch die LLMs betragen. Wie in [16] beschrieben können Nutzer, anstatt in Suchmaschinen beispielsweise die Vorteilen und Nachteile von PyTorch und Tensorflow zu vergleichen, kann das die LLM übernehmen und als Prompt wird nur `# import machine learning package` angegeben.

Wie in [le-2024] beschrieben nimmt das Lesen von Programm zehn mal mehr Zeit in Anspruch, als Code zu schreiben. Diese Arbeit kann ebenfalls durch eine LLM übernommen werden.

FAZIT

Mein roter Faden

Struktur des Kapitels

1. **Zusammenfassung:** kurze Wiederholung der Zielsetzung d. Arbeit, Überblick der wichtigsten Ergebnisse aus Eval. und Optimierung, Fragestellung beantwortet?
2. **Reflexion:** Stärken und Schwächen d. Arbeit, Diskussion über mögliche Fehlerquellen, Einschätzung Optimierungsansätze oder Benchmarks

Mein roter Faden

Unterschied Diskussion/Ausblick und Fazit

Aspekt	Diskussion und Ausblick	Fazit
Funktion	Kritische Analyse und Zukunftsperspektive	Zusammenfassung und Abschluss
Zeitperspektive	Zukunftsorientiert	Rückblickend
Detaillierungsgrad	Detailreichere Auseinandersetzung mit Ergebnissen	Knapp und prägnant

Während „Diskussion und Ausblick“ die Ergebnisse kritisch reflektiert und auf zukünftige Entwicklungen verweist, fasst das „Fazit“ die Arbeit kompakt zusammen und beantwortet die Forschungsfrage. Beide Kapitel sind komplementär, aber klar voneinander zu unterscheiden.

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-Higuaita. „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] 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).
- [26] 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).
- [27] 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).
- [28] Ran Elgedawy u. a. *Ocassionally Secure: A Comparative Analysis of Code Generation Assistants*. 1. Feb. 2024. URL: <https://arxiv.org/abs/2402.00689v1> (besucht am 10.01.2025).
- [29] *Gemini-Modelle*. URL: <https://ai.google.dev/gemini-api/docs/models/gemini?hl=de> (besucht am 10.01.2025).
- [30] *GGUF*. URL: <https://huggingface.co/docs/hub/en/gguf> (besucht am 23.01.2025).

- [31] Radziah Mohamad, Noraniah Yassin und Easter Sandin. „Comparative Evaluation of Automated Unit Testing Tool for PHP“. In: *International Journal of Software Engineering and Technology* 2 (Dez. 2016), S. 7–11.
- [32] Rini Anggrainingsih u. a. „Comparison of maintainability and flexibility on open source LMS“. In: Aug. 2016, S. 273–277. DOI: 10.1109/ISEMANTIC.2016.7873850.
- [33] Igor Regis Da Silva Simões und Elaine Venson. *Evaluating Source Code Quality with Large Language Models: a comparative study*. 7. Aug. 2024. URL: <https://arxiv.org/abs/2408.07082> (besucht am 30.12.2024).
- [34] Sandro Hartenstein und Andreas Schmietendorf. „KI-gestützte Modernisierung von Altanwendungen: Anwendungsfelder von LLMs im Software Reengineering“. In: *Softwaretechnik-Trends* Band 44, Heft 2. Gesellschaft für Informatik e.V., 2024. URL: <https://dl.gi.de/handle/20.500.12116/44181> (besucht am 15.08.2024).
- [35] Massimo Chiriatti u. a. „The case for human–AI interaction as system 0 thinking“. In: *Nature Human Behaviour* 8.10 (22. Okt. 2024), S. 1829–1830. DOI: 10.1038/s41562-024-01995-5. URL: <https://www.nature.com/articles/s41562-024-01995-5>.

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.

3

```
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"
-
8 +   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 Beispielaufwurf 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`.

A.4 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.

```

__VERSION__ = "0.0.1"

3 from transformers import AutoModelForCausalLM, AutoTokenizer

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

8     :param root_name (str): Root name of model
    :param vendor_name (str): Vendor name of model
    """
    # Personal access token from Hugging Face.
13    access_token = "hf_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"

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

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

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

```

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

A.5 Abfragen lokaler Modelle

Um die Modelle abzufragen, kommt der Code, welcher in Listing 8.4 zu sehen ist, zum Einsatz.

```
__VERSION__ = "0.0.1"
```

```
3 from langchain_ollama.llms import OllamaLLM
  from langchain.prompts import PromptTemplate

  from src.execute.execute import (
      read_problem,
8      write_log,
      MODEL_SETTING_MAX_TOKEN,
      MODEL_SETTING_TEMP,
      MODEL_SETTING_TOP_P,
  )
13
  MODEL_CONNECT_HOST: str = "192.168.178.140"
  MODEL_CONNECT_PORT: str = "11434"

18 INSTALLED_LLMS: list[dict] = [
    {"model": "qwen2.5-coder:32b", "answer_folder": "qwen25_coder"},
    # more models ...
]

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

    :param problem (str): Prompt there is send to ollama server.
28 :return Result from ollama server by prompt.
    """
    prompt_template: PromptTemplate = PromptTemplate(
        input_variables=["user_prompt"],
        template="{user_prompt}",
33    )
    prompt = prompt_template.format(user_prompt=problem)

    return model.invoke(prompt)

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

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

53 def execute_prompt(
    task_id: int,
    model_name: str,
    language: str,
    problem_filename: str,
58    answer_sub_folder: str,
    prompt_repetitions: int = 10,
) -> bool:
    """
    Create prompts and save local.

63
    :param task_id (int): Problem id
    :param model_type (str): Type of model.
    :returns bool: True prompt create.
    """

68    problem: dict = read_problem(
        task_id=task_id, language=language, problem_filename=
        problem_filename
    )
    prompt: str = problem.get("prompt", None)

73    if prompt is None:
        return False

    current_count: int = 0
    max_count: int = prompt_repetitions
78    model = connect_model(model_name=model_name)
```



```

while current_count < max_count:
    answer = run_model(problem=prompt, model=model)
    write_log(
        answer=answer,
83         key=f"result_{current_count}",
        task_id=f"{language}/{task_id}",
        sub_folder=answer_sub_folder,
    )
    current_count += 1
88 return True

```

Listing 8.4: Abfragen der Ollama Modelle

A.6 Abfrage cloudbasierter Modelle

```

__VERSION__ = "0.0.1"
2
from google import genai
from google.genai import types

from src.execute.execute import (
7     read_problem,
    write_log,
    MODEL_SETTING_MAX_TOKEN,
    MODEL_SETTING_TOP_P,
    MODEL_SETTING_TEMP,
12 )

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

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

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

```

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

generate_content_config = types.GenerateContentConfig(
32     temperature=MODEL_SETTING_TEMP,
    top_p=MODEL_SETTING_TOP_P,
    max_output_tokens=MODEL_SETTING_MAX_TOKEN,
    response_modalities=["TEXT"],
)

37
text: str = ""
for chunk in client.models.generate_content_stream(
    model=model_name,
    contents=[problem],
42     config=generate_content_config,
):
    text += chunk.text

return text.rstrip()

47
def connect_client() -> genai.Client:
    """
    Get Gemini client.

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

57
def execute_prompt(
    task_id: int,
    model_name: str,
    language: str,
    problem_filename: str,
62     answer_sub_folder: str,
    prompt_repetitions: int = 10,
```

```
) -> bool:
    """
    Execute model with problem.

67
    :param task_id (int): Problem task id.
    :param model_name (str): Model name.
    :param language (str): Programming language.
    :param problem_filename (str): File name with problem content.
72
    :param answer_sub_folder (str): Sub folder for save answer.
    :param prompt_repetitions (int, optional): Count of repetitions.
        Defaults to 10.
    :return bool: _description_
    """

    problem: dict = read_problem(
77
        task_id=task_id, language=language, problem_filename=
            problem_filename
    )

    prompt: str = problem.get("prompt", None)

    if prompt is None:
82
        return False

    current_count: int = 0
    max_count: int = prompt_repetitions
    client = connect_client()
87
    while current_count < max_count:
        answer: str = ""
        answer = run_model(problem=prompt, client=client, model_name=
            model_name)
        write_log(
92
            answer=answer,
            key=f"result_{current_count}",
            task_id=f"{language}/{task_id}",
            sub_folder=answer_sub_folder,
        )
        current_count += 1
97
    return True
```

Listing 8.5: Ausführen der Prompts für Gemini Modelle.

```
__VERSION__ = "0.0.1"

3 from openai import OpenAI

from src.execute.execute import (
    read_problem,
    write_log,
8     MODEL_SETTING_MAX_TOKEN,
    MODEL_SETTING_TOP_P,
    MODEL_SETTING_TEMP,
)

13 OPEN_AI_KEY: str = (
    "OPEN_AI_PERSONAL_KEY"
)
OPEN_AI_ORG: str = "org-XXXX"
OPEN_AI_PROJECT: str = "proj-YYYY"

18 ALLOWED_LLMS: list[dict] = [
    {"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.

    :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,
        top_p=MODEL_SETTING_TOP_P,
38         messages=[
            {
```

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

    return completion.choices[0].message.content

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

    :return OpenAI: OpenAI client.
53     """
    return OpenAI(
        api_key=OPEN_AI_KEY,
        organization=OPEN_AI_ORG,
        project=OPEN_AI_PROJECT,
58     )

def execute_prompt(
    task_id: int,
    model_name: str,
63     language: str,
    problem_filename: str,
    answer_sub_folder: str,
    prompt_repetitions: int = 10,
) -> bool:
68     """
    Execute model.

    :param task_id (int): _description_
    :param model_name (str): _description_
73     :param language (str): _description_
    :param problem_filename (str): _description_
    :param answer_sub_folder (str): _description_
    :param prompt_repetitions (int, optional): _description_. Defaults to
        10.
    :return bool: _description_
```

```

78     """
    problem: dict = read_problem(
        task_id=task_id, language=language, problem_filename=
            problem_filename
    )
    prompt: str = problem.get("prompt", None)
83
    if prompt is None:
        return False

    current_count: int = 0
88    max_count: int = prompt_repetitions
    client = connect_client()
    while current_count < max_count:
        answer: str = ""
        answer = run_model(problem=prompt, client=client, model_name=
            model_name)
93        write_log(
            answer=answer,
            key=f"result_{current_count}",
            task_id=f"{language}/{task_id}",
            sub_folder=answer_sub_folder,
98        )
        current_count += 1
    return True

```

Listing 8.6: Ausführen der Prompts für OpenAI Modelle.

A.7 Evaluation der Antworten von den Modellen

Das Listing 8.7 zeigt beispielhaft eine Suche nach Codezeilen in einer Antwort eines Modells. Die Methode `search_generated_code` muss auf jedes Modell angepasst sein.

```

def search_generated_code(content: str) -> str:
    """
    Search the generated code in string, it is one line of file.
    :param content (str): Answer from LLM.
5    :returns str: Generated code.
    """
    for index in [i for i in range(0, 10)]:

```

```
10  if content.startswith('{result_' + str(index) + '":'):
    generated_code: str = content.split(
        '{result_' + str(index) + '":')[1]
    generated_code = generated_code[:-2]
    generated_code = generated_code.strip()

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

20  if generated_code.startswith("<?php"):
    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"

25  return generated_code
return ""
```

Listing 8.7: Evaluation der Modellantworten: Suche nach Codeausschnitten

ANLAGEN

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	- - - -	- - - -

KAPITEL 8. ANHANG

Datensammlung und -analyse	- - - -	- - - -
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, 1. Februar 2025

Ort, Datum

Unterschrift