Integration von LLM-Agenten in MLOps: Erkenntnisse für die Automatisierung maschineller Lernpipelines

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering

Autoren	Cyrill Gurtner gurtncyr@students.zhaw.ch
	Nils Rechberger rechbnil@students.zhaw.ch
Betreuer	Frank-Peter Schilling scik@zhaw.ch
Studienprogramm	PA HS 24_scik_150
Eingereicht am	20. Dezember 2024

Inhaltsverzeichnis

1 Zusammenfassung

In der vorliegenden Arbeit wird die Integration von Large Language Models (LLMs) in Machine Learning Operations (MLOps) untersucht. Vorstellung dieser Integration ist die Steigerung der Automatisierung und Effizienz maschineller Lernpipelines. Ziel war die Entwicklung eines LLM-Agenten, der typische Aufgaben wie Datenvorverarbeitung, Modelltraining und Monitoring eigenständig ausführt und dabei auf moderne Sprachmodelle zurückgreift. MLOps, ein Paradigma zur Standardisierung von ML-Workflows, sieht sich mit Herausforderungen konfrontiert, die sich aus der wachsenden Komplexität und dem Bedarf an Skalierbarkeit ergeben. Manuelle Prozesse sind in der Regel mit Verzögerungen und höheren Kosten verbunden. LLMs werden zwar bereits in Bereichen wie der Textgenerierung und Codeerstellung erfolgreich eingesetzt, doch das Potenzial, das sie für MLOps-Prozesse bieten, ist bisher nur unzureichend erforscht. Die vorliegende Arbeit widmet sich daher der Frage, wie LLM-Agenten die Automatisierung fördern.

Die Forschungsstrategie verbindet eine modulare, containerisierte Infrastruktur mit der Integration führender LLMs wie GPT-4o und Llama3. Diese Modelle wurden speziell für die Automatisierung typischer MLOps-Aufgaben wie Datenhandling, Training und Deployment getestet. Ein Novum stellt der Einsatz von Prompt-Engineering dar, das eine gezielte Ausrichtung der Modelle auf spezifische Aufgaben ermöglicht. Experimente basierend auf Kaggle-Datensätzen dienten der Validierung des Ansatzes. Zur Nachverfolgung und Dokumentation der Experimente wurde MLflow genutzt, während Docker-Container eine isolierte und reproduzierbare Umgebung gewährleisteten.

Die Resultate der Untersuchung legen nahe, dass LLMs in der Lage sind, einfache bis mittelschwere Aufgaben mit einer gewissen Zuverlässigkeit zu automatisieren. GPT-4o wies dabei konsistente Leistungen auf, während Llama3 in komplexeren Szenarien Defizite zeigte. Die Automatisierung komplexer Aufgaben stösst jedoch an die Grenzen der aktuellen Modellgenerationen. Die Studie betont die Relevanz der Auswahl eines geeigneten Modells, um die Automatisierung effektiv zu implementieren. Es wird betont, dass spezialisierte Modelle und adaptive Multi-Agenten-Systeme zukünftige Forschungsschwerpunkte sein sollten.

Die Untersuchung leistet einen Beitrag zur wissenschaftlichen und praktischen Weiterentwicklung von MLOps-Prozessen. Es wird demonstriert, dass LLM-Agenten die Effizienz und Standardisierung in der KI-Entwicklung fördern können. Für Unternehmen ergeben sich daraus Potenziale zur Reduktion von Zeit- und Kostenaufwand. Zukünftige Arbeiten sollten sich daher auf die Integration in realweltliche Szenarien und die Weiterentwicklung von Modellarchitekturen konzentrieren, um die Effizienz und Skalierbarkeit weiter zu steigern. Die Arbeit zeigt, dass LLM-Agenten ein vielversprechender Ansatz sind, um die Herausforderungen moderner MLOps-Systeme zu adressieren und die Automatisierung in der Praxis voranzutreiben.

2 Abstract

This thesis analyses the integration of large language models (LLMs) in machine learning operations (MLOps). The aim of this integration is to increase the automation and efficiency of machine learning pipelines. The aim was to develop an LLM agent that performs typical tasks such as data pre-processing, model training and monitoring independently, using modern language models. MLOps, a paradigm for standardising and automating ML workflows, faces challenges resulting from growing complexity and the need for scalability. Manual processes are usually associated with delays and higher costs. While LLMs are already successfully used in areas such as text generation and code creation, the potential they offer for MLOps processes has not yet been sufficiently explored. This thesis therefore addresses the question of how LLM agents can promote and evaluate automation.

The research strategy combines a modular, containerised infrastructure with the integration of leading LLMs such as GPT-4o and Llama3. These models have been tested specifically for the automation of typical MLOps tasks such as data handling, training and deployment. A novel feature is the use of prompt engineering, which aligns the models to specific tasks. Experiments based on Kaggle data sets were used to validate the approach. MLflow was used to track and document the experiments, while Docker containers ensured an isolated and reproducible environment.

The results show that LLMs can reliably automate simple to moderately difficult tasks. GPT-4o performed consistently, while Llama3 showed deficits in more complex scenarios. However, the automation of complex tasks reached the limits of the current model generations. The study emphasises the importance of choosing the right model to implement automation effectively. It emphasises that specialised models and adaptive multi-agent systems should be the focus of future research.

The study contributes to the scientific and practical development of MLOps processes. It demonstrates that LLM agents can promote efficiency and standardisation in AI development. For companies, this results in potential for reducing time and costs. Future work should focus on the integration into real-world scenarios and the further development of model architectures in order to further increase efficiency and scalability. The work shows that LLM agents are a promising approach to address the challenges of modern MLOps systems and drive automation in practice.

3 Einleitung

3.1 Ausgangslage

Im Rahmen des Forschungsgebiets Machine Learning Operations (MLOps) erfolgt eine Übertragung bewährter Praktiken aus den Development Operations (DevOps) auf Modelle des Machine Learning (ML). Ziel von MLOps ist es, die produktive Nutzung von Machine Learning in der Praxis zu optimieren, indem standardisierte Prozesse und technologische Fähigkeiten implementiert werden [1]. Diese sollen es ermöglichen, ML-Systeme schnell, effizient und mit hoher Zuverlässigkeit zu entwickeln, bereitzustellen und zu betreiben.

Das MLOps-Paradigma deckt dabei den gesamten Lebenszyklus eines ML-Projekts ab. Neben den Kernaufgaben der Entwicklung, Implementierung und Bereitstellung zählen hierzu auch Planung, Design, Datenerhebung, das Training der Modelle sowie deren kontinuierliche Überwachung. Diese umfassende Herangehensweise soll gewährleisten, dass ML-Lösungen nicht nur technisch funktional sind, sondern auch robust und nachhaltig betrieben werden können. Gerade vor dem Hintergrund des anhaltenden KI-Booms, der sowohl Grossunternehmen als auch kleine und mittlere Unternehmen (KMU) betrifft, gewinnt MLOps zunehmend an Bedeutung. Die wachsende Zahl von Investitionen in KI-bezogene Unternehmen verdeutlicht diesen Trend: Laut der National Venture Capital Association haben im Jahr 2019 allein in den USA 1.356 Unternehmen Investitionen in Höhe von insgesamt 18,457 Milliarden US-Dollar erhalten, eine Steigerung gegenüber den 1.281 Unternehmen, die im Vorjahr eine Finanzierung von 16,8 Milliarden US-Dollar erhielten [2].

Die steigende Bedeutung von MLOps ist eng mit der Bewältigung immer komplexerer Herausforderungen in der Entwicklung und Implementierung von ML-Projekten verbunden. Insbesondere in einem Umfeld, in dem die Erwartungen an die Leistungsfähigkeit und Stabilität von ML-Modellen kontinuierlich steigen, ist es essenziell, Projekte effizient umzusetzen. Verzögerungen durch repetitive oder ineffiziente Prozesse können nicht nur die Durchlaufzeit von der Entwicklung bis zur Bereitstellung verlängern, sondern auch den gesamten Workflow beeinträchtigen. Die Entwicklung effektiver MLOps-Strategien ist daher nicht nur ein technisches Erfordernis, sondern auch eine betriebswirtschaftliche Notwendigkeit, um im Wettbewerb um innovative KI-Anwendungen bestehen zu können.

Darüber hinaus stellt die zunehmende Vielfalt an ML-Anwendungen eine Herausforderung für die Skalierbarkeit und Wiederverwendbarkeit von Lösungen dar. In hochgradig dynamischen Umfeldern wie der Industrie 4.0 können standardisierte MLOps-Prozesse dazu beitragen, branchenspezifische Anforderungen effizient zu erfüllen. Die Automatisierung spielt in diesem Kontext eine Schlüsselrolle. Automatisierte MLOps-Pipelines ermöglichen nicht nur eine drastische Reduktion manueller Eingriffe, sondern fördern auch die Konsistenz und Wiederholbarkeit von ML-Workflows. Werkzeuge wie Continuous Integration (CI) und Continuous Deployment (CD) werden in Verbindung mit spezialisierten ML-Plattformen wie beispielsweise MLflow[1] immer häufiger eingesetzt, um den operativen Aufwand zu minimieren und gleichzeitig die Qualität zu erhöhen.

[1] https://mlflow.org/

3.2 Zielsetzung

Diese Arbeit verfolgt das Ziel, MLOps-Prozesse durch den Einsatz von LLM-Agenten zu automatisieren und dabei die Effizienz und Skalierbarkeit zu erhöhen. Dabei greifen wir auf die jüngsten Fortschritte in der Verarbeitung natürlicher Sprache durch LLMs zurück. Diese Modelle haben sich nicht nur durch ihre aussergewöhnlichen Fähigkeiten im Bereich der Textgenerierung, sondern auch durch ihr bemerkenswertes Potenzial als Programmierwerkzeuge ausgezeichnet. Tatsächlich erreichen LLMs in der Programmierung inzwischen Leistungen, die mit denen menschlicher Entwickler vergleichbar sind. Ein Beispiel hierfür ist die o1-Vorschau von OpenAI[2], welche in 16,9 % der Kaggle-Wettbewerbe mindestens das Niveau einer Bronzemedaille[3] erreicht hat, was je nach Wettbewerb und Teilnehmeranzahl einer Platzierung unter den besten 40 bis 10 % entspricht [3].

Der Einsatz von LLMs ermöglicht es, komplexe Aufgaben in einfache, systematische Schritte zu unterteilen. Durch präzise formulierte Eingabeaufforderungen, sogenannte Prompts, können die Fähigkeiten dieser Modelle gezielt angesteuert und auf bestimmte Aufgaben ausgerichtet werden. Diese Prompts stellen dabei nicht nur den Mechanismus zur Steuerung der Modelle dar, sondern auch eine Schnittstelle, über die externe Systeme eingebunden und gesteuert werden können. Über die einfache Nutzung von LLMs als Chatbots hinausgehend, eröffnet diese erweiterte Funktionalität neue Möglichkeiten, indem sogenannte LLM-Agenten entwickelt werden. Solche Agenten sind in der Lage komplexe Arbeitsabläufe zu bewältigen und interaktiv mit anderen Softwarekomponenten zu arbeiten.

Die Integration von LLMs in MLOps-Prozesse bietet somit ein erhebliches Potenzial zur Automatisierung und Effizienzsteigerung. Insbesondere in der Entwicklung von ML-Modellen, wo wiederkehrende Aufgaben häufig zeit- und ressourcenintensiv sind, kann der Einsatz von LLM-Agenten die Durchlaufzeiten deutlich verkürzen und die Zuverlässigkeit der Implementierung erhöhen. Dies trägt nicht nur zur Optimierung bestehender Arbeitsabläufe bei, sondern erweitert auch die Möglichkeiten, innovative und skalierbare Lösungen im Bereich des maschinellen Lernens zu realisieren.

3.3 Aufgabenstellung

Die zentrale Aufgabe dieser Arbeit liegt in der Konzeption und Umsetzung eines LLM-Agenten auf Basis von LLMs, der alltägliche Aufgaben im Bereich MLOps effizient automatisieren und bewältigen kann.

Neben diesen praktischen Aufgabenstellungen zielt die Arbeit darauf ab, die Leistungsfähigkeit eines LLM-Agenten bei der Bearbeitung von Aufgaben unterschiedlicher Komplexität zu untersuchen. Dabei wird evaluiert, inwiefern die Fähigkeit des LLM-Agenten, spezifische Anforderungen zu erfüllen, von der zugrunde liegenden Architektur und den Eigenschaften des verwendeten LLM abhängt. Ein zentraler Aspekt dieser Untersuchung ist die Analyse der Qualität des vom LLM-Agenten generierten Codes. Die Ergebnisse dieser Evaluierung sollen Aufschluss darüber geben, in welchen Szenarien der Einsatz eines LLM-Agenten besonders geeignet ist und wo potenzielle Grenzen liegen.

[2] https://openai.com/index/introducing-openai-o1-preview/
[3] https://www.kaggle.com/progression

Vor der praktischen Umsetzung ist es wichtig, geeignete Anwendungsfälle (Use Cases) zu evaluieren. Diese Evaluation bildet die Grundlage für die strategische Planung und Vorbereitung der Implementierung. Die ausgewählten Use Cases sollen dabei ein möglichst breites Spektrum typischer MLOps-Aufgaben abdecken, um die vielseitigen Einsatzmöglichkeiten des entwickelten LLM-Agenten zu demonstrieren und dessen praktische Relevanz zu untermauern.

3.4 Anforderungen

Die Implementierung des LLM-Agenten basiert auf GPT Engineer[4], einem Open-Source-Projekt, das für die Automatisierung typischer Softwareentwicklungsaufgaben konzipiert wurde. Dieses Framework bietet leistungsstarke Werkzeuge, um moderne Technologien wie Prompt-Engineering und die Feinabstimmung (fine-tuning) grosser Sprachmodelle effektiv einzusetzen.

Trotz der vielseitigen Einsatzmöglichkeiten des Frameworks ist GPT Engineer primär als generisches Werkzeug für Interaktionen mit LLMs konzipiert. Dies macht eine gezielte Anpassung erforderlich, um die spezifischen Anforderungen an einen LLM-Agenten zu erfüllen. Eine zentrale Herausforderung dabei ist die Auswahl eines geeigneten LLMs, das sowohl die Komplexität als auch die Vielfalt der MLOps-Aufgaben bewältigen kann.

[4]https://gpt-engineer.readthedocs.io/en/latest/index.html

4 Theoretische Grundlagen

4.1 Stand der Forschung

Am 22. November 2022 präsentierte OpenAI ChatGPT[1], was einen entscheidenden Impuls für die Weiterentwicklung und den verstärkten Einsatz von KI-Technologien markierte, die aktuell primär auf LLMs basieren. Die Fähigkeit von ChatGPT, menschliche Sprache nicht nur präzise zu verstehen, sondern auch adäquate und relevante Antworten zu generieren, ist ein zentraler Aspekt seiner Funktionalität. Diese Fähigkeit allein erklärt jedoch nicht den aussergewöhnlichen Erfolg von LLMs. Ein wesentlicher Faktor für die breite Akzeptanz war die rasante Demonstration ihres Potenzials durch eine grosse Nutzerbasis. Wie die Analyseplattform Exploding Topics berichtet, verzeichnete ChatGPT innerhalb der ersten fünf Tage nach der Veröffentlichung eine beeindruckende Nutzerzahl von über einer Million. Bis August 2024 stieg die Zahl der aktiven Nutzer auf 121,3 Millionen an [4]. Diese Zahlen unterstreichen nicht nur die Popularität von LLMs, sondern auch ihren Einfluss auf die Etablierung neuer Technologien und Geschäftsmodelle.

Die Einführung von ChatGPT durch OpenAI öffnete neue Marktsegmente und führte zu einer verstärkten Kommerzialisierung von LLMs. Zahlreiche Unternehmen folgten diesem Beispiel und brachten konkurrierende Modelle auf den Markt, wie beispielsweise LLama[2] vom Meta. Diese Produkte spiegeln die Dynamik und Innovationskraft wider, die dieser neu entstandene Sektor hervorbrachte.

Alle genannten Modelle folgen dem Paradigma des "Language-Models-as-a-Service"(LMaaS), das es Nutzerinnen und Nutzern gegen eine Gebühr ermöglicht, über ein Interface auf die Modelle zuzugreifen. Entwickler, die diese Systeme in spezifischen Kontexten nutzen wollen, stehen jedoch vor besonderen Herausforderungen, darunter die Evaluierung, das Benchmarking und die Qualitätsprüfung der LLMs [5]. Zur Unterstützung dieses Prozesses stellen moderne Anbieter APIs zur Verfügung, die es ermöglichen, LLMs in Anwendungen zu integrieren. Diese Entwicklung führte zu einem weiteren Angebot: AI-as-a-Service. Dabei handelt es sich um Frameworks, die auf bestehende LLMs zurückgreifen und deren Fähigkeiten auf individuelle Anwendungsfälle anpassen.

Parallel dazu gewann auch die Open-Source-Community im Bereich der KI-Entwicklung zunehmend an Bedeutung. Eine Schlüsselrolle spielt dabei die Plattform Hugging Face[3], die sich als zentraler Ort für die Zusammenarbeit an maschinellen Lernmodellen etabliert hat. Sie bietet nicht nur Zugang zu einer Vielzahl von Modellen und Datensätzen, sondern ermöglicht auch das Teilen und Optimieren individuell entwickelter Anwendungen. Der Prozess des sogenannten Finetunings, bei dem vortrainierte Modelle für spezifische Aufgaben optimiert werden, hat sich hierbei als besonders effektive Methode erwiesen. Diese Technik wird zunehmend als integraler Bestandteil des Trainingsprozesses von Basismodellen eingesetzt, die in generativen KI-Systemen Anwendung finden [6]. Voraussetzung für diese Anpassungen ist jedoch die Verfügbarkeit geeigneter Datensätze, um das Modell gezielt zu spezialisieren.

[1] https://openai.com/index/chatgpt/
[2] https://www.llama.com/
[3] https://huggingface.co/

Ein Unternehmen, das frühzeitig das Potenzial dieser Entwicklungen erkannte und über die notwendigen Ressourcen verfügte, um ein marktfähiges Produkt zu schaffen, war Microsoft. Mit der Einführung von Copilot[4] bot Microsoft eines der ersten LLM-basierten Systeme speziell für Entwickler und Programmierer an. Dieses Produkt demonstrierte eindrucksvoll, wie LLMs genutzt werden können, um produktive und kreative Prozesse in der Softwareentwicklung zu unterstützen. Es repräsentiert zugleich die zunehmende Integration von KI-Technologien in professionelle Arbeitsumgebungen und hebt den Stellenwert solcher Lösungen als transformative Werkzeuge für die Zukunft der Arbeitswelt hervor.

4.2 Forschungslücke

Es lässt sich somit festhalten, dass sowohl etablierte LLMs als auch spezialisierte Modelle einen erheblichen Einfluss auf die moderne Softwareentwicklung ausüben. Die Bandbreite ihrer Anwendungen reicht von der Unterstützung bei alltäglichen Programmieraufgaben bis hin zur Automatisierung komplexer Entwicklungsprozesse. Dabei erweist sich die Fähigkeit der Modelle, spezifische Anforderungen zu adressieren, als entscheidender Vorteil, insbesondere in der Anpassung an branchenspezifische oder unternehmensinterne Bedürfnisse.

Eine zentrale Fragestellung bleibt jedoch, wie sich spezialisierte LLMs effektiv in bestehende Programmierumgebungen integrieren und für die Automatisierung wiederkehrender Aufgaben einsetzen lassen. Die Herausforderungen betreffen dabei nicht nur die technische Einbettung in Entwicklungs-Workflows, sondern auch die Gewährleistung einer nahtlosen Interaktion zwischen menschlichen Entwicklern und den KI-Systemen. Zusätzlich ist sicherzustellen, dass die Codevorschläge qualitativ hochwertig, sicher und den Standards der Softwareentwicklung entsprechend sind, um den Anforderungen an Zuverlässigkeit und Effizienz gerecht zu werden.

In diesem Kontext gehen wir mit der Entwicklung unseres LLM-Agenten über die blosse Evaluierung bestehender Modelle hinaus. Unser Ansatz bietet eine praxisnähere Weiterentwicklung, indem wir die Integration projektspezifischer Frameworks ermöglichen. Diese Spezialisierung erlaubt es, nicht nur generische Aufgaben zu bewältigen, sondern auch massgeschneiderte Lösungen für spezifische Softwareprojekte bereitzustellen. Durch die Nutzung solcher Frameworks wird die Anpassung an reale Arbeitsumgebungen erleichtert und die praktische Relevanz des LLM-Agenten erhöht.

4.3 Theorie

4.3.1 MLOps Lebenszyklus

Der MLOps-Lebenszyklus umfasst typischerweise verschiedene Phasen, die von der Datenverwaltung über die Modellentwicklung und -validierung bis hin zur Modellbereitstellung, -überwachung und -wartung reichen. Diese Phasen bilden einen iterativen Prozess, der es ermöglicht, Modelle kontinuierlich zu aktualisieren und an veränderte Daten oder Geschäftsanforderungen anzupassen. Dadurch wird nicht nur die Modellleistung langfristig gesichert, sondern auch der gesamte Entwicklungsprozess effizienter gestaltet.

Wie Salama et al. [1] in ihrer Arbeit erläutern, lässt sich der MLOps-Lebenszyklus in klar definierte Schritte unterteilen, die eine Grundlage für die praktische Umsetzung dieses Paradigmas bilden. Abbildung 4.1 zeigt eine typische Darstellung des MLOps-Lebenszyklus:

[4] https://copilot.microsoft.com/

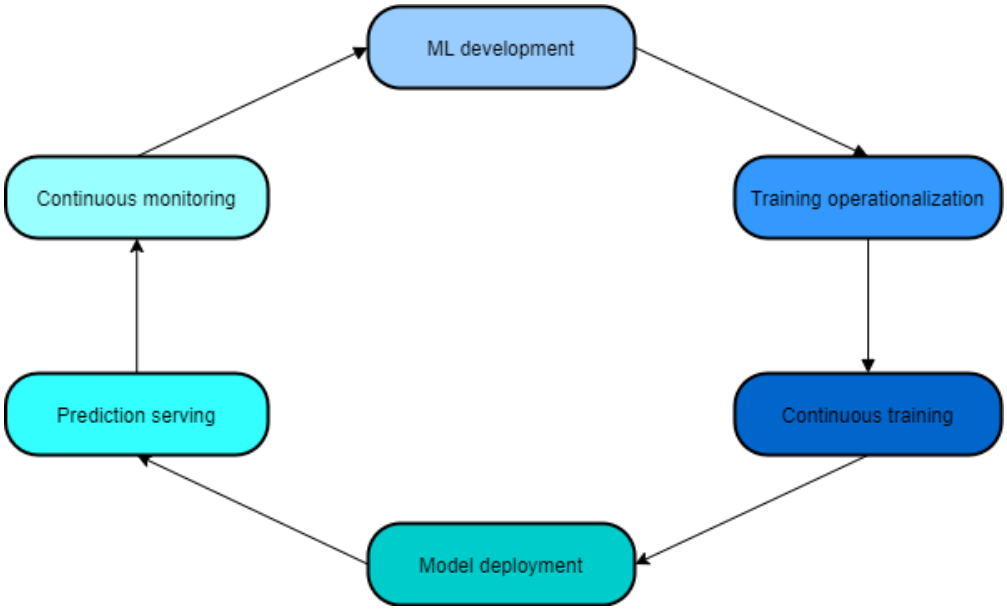

Abbildung 4.1: Typischer MLOps-Lebenszyklus (nach Salama et al. [1]).

4.3.2 Large Language Models

Large Language Models basieren auf einer Reihe theoretischer Grundlagen des maschinellen Lernens, insbesondere der neuronalen Netzwerke und der Wahrscheinlichkeitsmodellierung von Sprache. Ihre Funktionsweise zeichnet sich durch die Fähigkeit aus, kohärente, relevante und kontextuell adäquate Texte zu generieren. Diese Leistung beruht massgeblich auf der Skalierung sowohl der Modellarchitektur als auch der Datenmengen, die während des Trainings genutzt werden. Die moderne Architektur von LLMs, insbesondere Transformer-Modelle, wurde durch Vaswani et al. aufgezeigt [7]. Diese bahnbrechende Entwicklung hat die Sprachmodellierung grundlegend verändert, indem sie die sequentielle Verarbeitung traditioneller Modelle durch Selbstaufmerksamkeitsmechanismen ersetzte. Diese Mechanismen ermöglichen es, Beziehungen zwischen Wörtern unabhängig von deren Distanz im Text zu modellieren, wodurch tiefere Kontextverknüpfungen hergestellt werden können.

Durch die Approximation der bedingten Wahrscheinlichkeitsverteilung eines Tokens optimieren LLMs die Maximum-Likelihood-Schätzung (MLE) während des Trainings auf umfangreichen Textkorpora. Wie Brown et al. in ihrer Arbeit zu GPT-3 zeigen, führt die Skalierung von Parametern und Daten zu einer signifikanten Verbesserung der Generalisierungsfähigkeit, selbst wenn die zugrunde liegenden Mechanismen unverändert bleiben [8]. Diese Fortschritte wurden durch Regularisierungstechniken wie Dropout und Layer-Normalisierung unterstützt, die die Stabilität und Effizienz während des Trainings gewährleisten [9].

Trotz dieser bemerkenswerten Erfolge stützen sich LLMs weiterhin ausschliesslich auf statistische Darstellungen von Sprache, ohne ein inhärentes Weltwissen oder ein echtes Verständnis von Bedeutung und Intentionalität zu besitzen. Bender et al. kritisieren in ihrer Arbeit, dass LLMs überzeugende, aber häufig fehlerhafte oder irreführende Texte generieren können, da ihnen die Fähigkeit fehlt, den tieferen Kontext und die Absichten hinter einem Text zu erfassen [10]. Diese Einschränkungen werfen weiterhin grundlegende Fragen zu ihrer Effizienz, den ethischen Implikationen und der theoretischen Interpretation ihrer Fähigkeiten auf.

4.3.3 LLMs-Agents

Da das Forschungsfeld der LLM-Agenten noch relativ jung ist, gibt es noch keine standardisierte oder etablierte Praxis in Forschung und Industrie. Ein vielversprechendes Anwendungsfeld ist der Einsatz von LLM-Agenten in Betriebssystemen (Operating Systems, OS), bei dem der LLM-Agent die Rolle eines „Betriebssystem-Gehirns“ übernimmt und so ein Betriebssystem mit einer Art „Seele“ ausgestattet wird. Diese Idee wurde von Mei et al. in ihrer Arbeit über AIOS vorgestellt, wo sie beschreiben, wie LLM-Agenten die Interaktion und Entscheidungsprozesse eines Betriebssystems auf eine intelligente, kontextuell angepasste Weise ermöglichen können [11].

Ein weiteres potenzielles Entwicklungsfeld für omni-fähige LLM-Agenten stellt der Einsatz von LLM-Agent-Kollektiven dar. Diese Frameworks basieren auf der Zusammenarbeit mehrerer LLM-Agenten, die jeweils auf eine spezifische Aufgabe oder Funktion abgestimmt sind. Der Zusammenschluss mehrerer spezialisierter Modelle eröffnet neue Möglichkeiten, indem er es ermöglicht, die Agenten in einer dynamischen Architektur miteinander agieren zu lassen. Durch Mechanismen wie Agentenauswahl zur Inferenzzeit und frühzeitiges Stoppen (Early Stopping) kann die Effizienz und Leistung der gesamten Agentengruppe verbessert werden. Liu et al. beschreiben, dass diese dynamischen Architekturen die Interaktion von LLM-Agenten über mehrere Runden hinweg optimieren und so eine bessere Nutzung von Ressourcen und eine höhere Leistung erzielen können [12].

Diese Ansätze zeigen das Potenzial auf, wie ein einzelner LLM-Agent in der Zukunft spezifische Aufgaben mit hoher Effizienz und Anpassungsfähigkeit lösen könnte. Ein einzelner LLM-Agent, der für eine konkrete Funktion innerhalb eines Systems verantwortlich ist, kann durch kontinuierliche Anpassung und Feinabstimmung auf bestimmte Anwendungsfälle optimiert werden. Durch seine Fähigkeit, aus einer Vielzahl von Datenquellen zu lernen und auf neue, unerforschte Szenarien zu reagieren, bietet er nicht nur eine hohe Flexibilität, sondern auch die Möglichkeit, komplexe und dynamische Aufgaben eigenständig zu bewältigen.

4.4 Hypothese

Ein LLM-Agent, der speziell für die Durchführung von MLOps-Aufgaben entwickelt und implementiert wird, kann die Effizienz und Qualität von MLOps-Prozessen signifikant verbessern, wobei die Leistungsfähigkeit des Agenten in Bezug auf Codequalität, Funktionalität und Innovationsgrad in direktem Zusammenhang mit der zugrunde liegenden Architektur und den spezifischen Eigenschaften des verwendeten LLM steht. Die Fähigkeit des LLM-Agenten, Anforderungen unterschiedlicher Komplexität zu erfüllen, wird in Abhängigkeit von der Aufgabenstellung und den ausgewählten Use Cases variieren, wobei sich klare Grenzen für den praktischen Einsatz des Agenten bei besonders komplexen oder dynamischen MLOps-Aufgaben abzeichnen werden.

5 Vorgehen

Das Aufgabengebiet bei MLOps umfasst eine Vielzahl von abwechslungsreichen und anspruchsvollen Tätigkeiten. Daher muss unser Agent in der Lage sein, flexibel auf die unterschiedlichsten Anforderungen zu reagieren. Dabei geht es nicht nur darum, die Komplexität der jeweiligen Aufgabe zu erfassen, sondern auch den Kontext zu verstehen, in dem der Agent agiert und Entscheidungen trifft.

Als Basis für die Entwicklung unseres Agenten verwenden wir das Framework GPT-Engineer [1]. Dieses Projekt verwendet grosse Sprachmodelle (LLMs) wie GPT-4 von OpenAI [2], um den Prozess der Softwareentwicklung zu automatisieren. GPT-Engineer umfasst mehrere Python-Skripte, die in enger Interaktion mit dem LLM Code generieren, Anforderungen klären und Spezifikationen erstellen [13].

Das Framework bietet zwei zentrale Funktionalitäten, die wir für unseren Agenten nutzen: Preprompts und die einfache Integration offener LLMs. Preprompts ermöglichen es, den Kontext, in dem der Agent arbeiten soll, bereits vor der eigentlichen Aufgabenbearbeitung klar zu definieren. Die Integration offener LLMs wiederum bietet eine benutzerfreundliche Methode, neue Modelle schnell und einfach an GPT-Engineer anzubinden - ohne umfangreiche Anpassungen an der Konfiguration vornehmen zu müssen.

Um zu evaluieren, inwieweit ein MLOps-Agent die Automatisierung von MLOps-Aufgaben unterstützen kann, wurden 4 Kaggle-Challenges ausgewählt, die typische MLOps-Aufgaben repräsentieren. Diese Aufgaben wurden aufgrund ihrer Komplexität und ansteigendem Schwierigkeitsgrad ausgewählt.[3].

Zusätzlich haben wir zwei LLM-Modelle getestet, um ein umfassenderes Bild von der Leistung des Agenten zu erhalten. Die Wahl fiel auf zwei der derzeit bekanntesten LLMs: GPT-4o und Llama3.

Um unseren Agenten schnell und ohne aufwändige Installation einsatzbereit zu machen, haben wir ihn in einem Docker-Container implementiert. Dies hat zwei wesentliche Vorteile: Erstens ermöglicht es eine schnelle, plattformunabhängige Bereitstellung und Konfiguration des Agenten. Zweitens sorgt die Containerisierung für eine isolierte Umgebung, die unabhängig von anderen Systemen betrieben werden kann.

Für die technische Dokumentation der Datensätze, Modelle und Metriken haben wir uns für das MLflow Framework entschieden. MLflow ist eine Open-Source-Plattform, die speziell entwickelt wurde, um Praktiker und Teams im Bereich des maschinellen Lernens bei der Bewältigung komplexer Prozesse zu unterstützen. Sie deckt den gesamten Lebenszyklus von Machine-Learning-Projekten ab und stellt sicher, dass jede Phase überschaubar, nachvollziehbar und reproduzierbar bleibt [14].

[1] https://gpt-engineer.readthedocs.io/en/latest/index.html
[2] https://openai.com/index/gpt-4/
[3] Dies ist keine offizielle Klassifizierung von Kaggle.com

5.1 Preprompts

GPT-Engineer stellt standardmässig mehrere Preprompt-Dateien zur Verfügung, die dazu dienen, das verwendete LLM-Modell in den richtigen Kontext zu setzen. Diese Dateien spielen eine zentrale Rolle bei der Steuerung der Interaktion zwischen Benutzer und Modell, da sie den Rahmen für die Bearbeitung der Aufgaben definieren.

Einige dieser Dateien werden zu Beginn eines Arbeitsprozesses automatisch eingebunden und an das LLM übergeben, um eine grundlegende Orientierung zu gewährleisten. Andere Preprompts werden nur unter bestimmten Bedingungen verwendet, z.B. wenn bestimmte Parameter gesetzt sind. Ein Beispiel hierfür ist der Parameter -c (für clarify), der definiert, wie das Modell mit Mehrdeutigkeiten im Prompt umgehen soll. Ist dieser Parameter aktiviert, sorgt der entsprechende Preprompt dafür, dass das LLM gezielt Rückfragen stellt oder Vorschläge macht, um unklare Anforderungen zu klären.

Zusätzlich können benutzerdefinierte Preprompts erstellt werden, um spezifische Anforderungen oder Kontexte abzudecken. Dies ermöglicht eine flexible Anpassung an individuelle Arbeitsabläufe und die Integration von domänenspezifischem Wissen. Der modulare Aufbau von GPT-Engineer erleichtert nicht nur die Auswahl und Kombination von Preprompts, sondern auch deren Erweiterung durch eigene Templates.

Durch diese vielseitige Verwendung von Preprompts bietet GPT-Engineer ein hohes Mass an Anpassungsfähigkeit und erleichtert die effiziente Lösung komplexer Aufgaben in unterschiedlichen Kontexten.

5.2 Verwendete LLMs

Auch bei der Auswahl der LLMs bietet GPT-Engineer eine benutzerfreundliche Möglichkeit zur Integration verschiedener Modelle. Neben dem Standardmodell GPT-4o können über den Anbieter OpenRouter[4] zahlreiche weitere Modelle angebunden werden.

OpenRouter stellt eine OpenAI-kompatible Vervollständigungs-API zur Verfügung, die den Zugriff auf über 277 Modelle und Anbieter ermöglicht. Diese Modelle können entweder direkt oder über das OpenAI SDK aufgerufen werden. Darüber hinaus stehen verschiedene SDKs von Drittanbietern zur Verfügung, die die Integration weiter erleichtern und zusätzliche Funktionalitäten bieten [15].

Die Integration eines neuen Modells in GPT-Engineer erfolgt ohne grossen Konfigurationsaufwand. Die Benutzer müssen lediglich die Zugangsdaten oder API-Schlüssel des gewünschten Modells angeben, um die Verbindung herzustellen. Dadurch ist es möglich, flexibel zwischen verschiedenen Modellen zu wechseln und unterschiedliche Fähigkeiten oder Architekturen zu testen - beispielsweise um die Performance für bestimmte Aufgaben zu optimieren oder neue Entwicklungen im Bereich der LLMs zu evaluieren.

Ein weiterer Vorteil der OpenRouter-Integration ist die standardisierte Schnittstelle, die sicherstellt, dass unterschiedliche Modelle ohne grössere Anpassungen in bestehende Workflows eingebunden werden können. Dies fördert die Skalierbarkeit und macht den GPT-Engineer zu einem vielseitigen Werkzeug für unterschiedlichste Anwendungsfälle im Bereich der LLMs und darüber hinaus.

[4] https://openrouter.ai/

5.3 Systemarchitektur

Fasst man alle Komponenten unseres Agenten zusammen, so ergibt sich folgende Systemarchitektur (siehe Abb. 5.1). Diese Architektur verbindet die einzelnen Bausteine - von der Modellintegration über die Preprompt-Verwaltung bis hin zur containerisierten Auslieferung - zu einem kohärenten und leistungsfähigen Gesamtsystem.

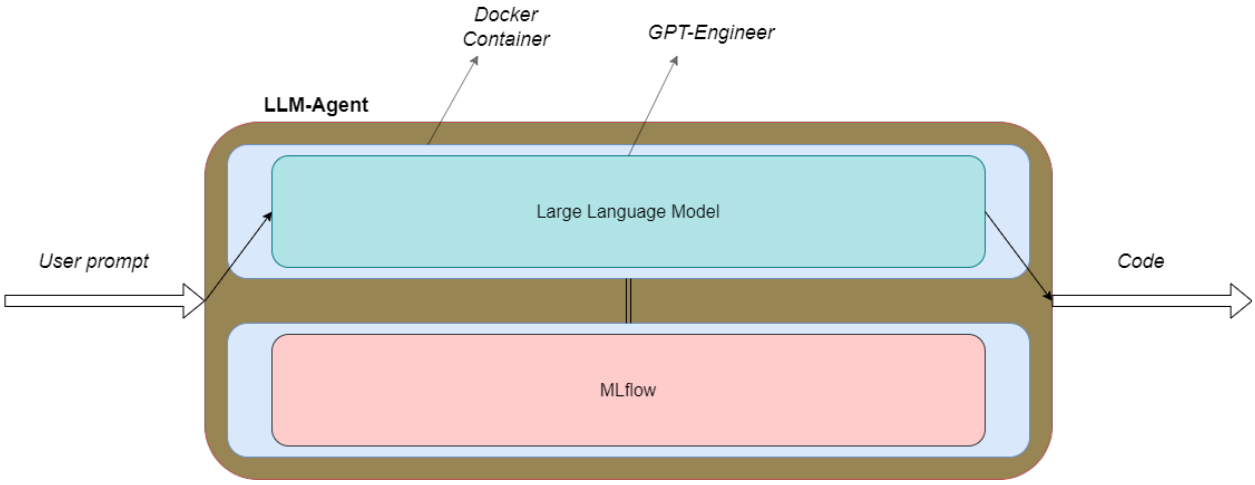

Abbildung 5.1: Die Systemarchitektur des LLM-Agenten basierend den einzelnen Komponenten.

Die Abbildung zeigt, wie die verschiedenen Elemente miteinander interagieren: Die LLMs werden über eine standardisierte API-Schnittstelle eingebunden, während Preprompts und Konfigurationsparameter den Kontext und die Aufgabensteuerung definieren. Ergänzt wird dies durch die Verwendung von Docker für eine isolierte und plattformunabhängige Umgebung und MLflow für die Verwaltung der experimentellen Daten und Modellmetriken.

Diese modulare Systemarchitektur ermöglicht Flexibilität, Skalierbarkeit und einfache Erweiterbarkeit, so dass der Agent an vielfältige Anforderungen und neue Entwicklungen im Bereich MLOps angepasst werden kann.

6 Implementation

Basierend auf der skizzierten Systemarchitektur beschreiben wir in den folgenden Abschnitten die implementierten Projektteile, insbesondere die Entwicklung der Containerumgebung und Preprompts.

6.1 Architektur

Die Gesamtarchitektur des Projekts ist in Abbildung 6.1 veranschaulicht. Basierend auf einer klaren, modularen Struktur werden alle wesentlichen Komponenten – von den Grundkonfigurationen über die Experimente bis hin zur Modellierung und dem Tracking in MLflow – in separaten Ordnern gekapselt. Diese Organisation erleichtert nicht nur die Weiterentwicklung und Wartung, sondern ermöglicht auch eine rasche Orientierung im Code.

Im Folgenden wird zunächst die Verzeichnisstruktur im Überblick dargestellt und in den folgenden Abschnitten wird näher auf Details der Containerumgebung, der Preprompts und der Implementierung der Experimente eingegangen.

```
.
├── README.md
├── docker-compose.yml
├── gpte
│   ├── Dockerfile
│   ├── experiments
│   │   ├── experiment-easy-digit-recognizer
│   │   ├── experiment-easy-house-prices
│   │   ├── experiment-easy-titanic
│   │   ├── experiment-hard-detect-misinformation
│   │   ├── experiment-hard-object-recognition
│   │   ├── kaggle.json
│   │   ├── preprompts
│   │   │   ├── clarify
│   │   │   ├── entrypoint
│   │   │   ├── file_format
│   │   │   ├── file_format_diff
│   │   │   ├── file_format_fix
│   │   │   ├── generate
│   │   │   ├── improve
│   │   │   ├── philosophy
│   │   │   └── roadmap
│   │   └── run.py
│   └── requirements.txt
└── mlflow
    ├── Dockerfile
    ├── app
    │   └── log_mlflow_with_localhost.py
    └── requirements.txt
```

Abbildung 6.1: Projektstruktur

6.2 Docker

Die gesamte Experimentier- und Ausführungsumgebung wurde in Docker-Containern gekapselt, um eine reproduzierbare sowie isolierte Ausführungsumgebung sicherzustellen. Dies erlaubt es, sowohl auf Entwicklungs- als auch auf Produktionssystemen ein identisches Setup bereitstellen zu können. Hierfür wurden zwei Haupt-Services in Containern umgesetzt:

- **MLflow-Container**: Dient der Nachverfolgbarkeit (Tracking) von Experimenten, dem Logging von Parametern, Metriken und Modellen sowie der Modellversionierung. Über den MLflow-Server können alle durchgeführten Experimente und deren Resultate zentral eingesehen werden. Der Container basiert auf einer `python:3.12-slim`-Image-Variante und installiert die benötigten Pakete für MLflow. Anschliessend wird der MLflow-Server über einen definierten Port nach aussen verfügbar gemacht.
- **GPTE-Container (LLM-Experimente)**: Hier laufen die skriptbasierten Experimente mittels Large Language Models (LLMs). Der Container enthält neben der eigentlichen Experimentlogik (Python-Skripte in `run.py` und den jeweiligen `experiment`-Verzeichnissen) auch Konfigurationen für den Zugriff auf Kaggle-Datasets. Das Basis-Image `python:3.12` wird um notwendige Systemabhängigkeiten (z. B. `build-essential`, `git`, `curl`) und Python-Pakete erweitert. Die Konfiguration der `kaggle.json`-Datei ermöglicht den automatisierten Download von Datensätzen. Die Einbindung des gesamten Projektverzeichnisses in den Container (`./:/app`) ermöglicht es, sowohl auf lokale Projektdateien als auch auf dynamisch generierte Inhalte (z. B. generierter Code durch LLM-Tools) zuzugreifen.

Die in `docker-compose.yml` definierte Orchestrierung ermöglicht das parallele Starten beider Container. Der MLflow-Service wird auf einem lokalen Host-Port (z.,B. `127.0.0.1:8000`) verfügbar gemacht, um über den Browser auf das MLflow UI zugreifen zu können. Der GPTE-Container greift direkt auf die im Projekt eingebundenen Ordnerstrukturen zu und kann somit Experimente starten, die Ergebnisse via MLflow loggen sowie generierte Artefakte lokal ablegen.

Dieser Aufbau erlaubt ein einfaches Deployment, Wiederholbarkeit der Experimente, klar definierte Schnittstellen zwischen den Services sowie eine problemlose Erweiterbarkeit. Durch die Nutzung von Containern und Docker-Compose kann die gesamte Infrastruktur mit wenigen Befehlen gestartet, gestoppt und neu aufgesetzt werden.

6.3 Ausführbarkeit der Experimente

Die Ausführbarkeit der Experimente wird über das Python-Script `run.py` gewährleistet. Dieses Script erlaubt es, Experimente parametrisiert zu starten und unterschiedliche Modelle sowie spezifische Experimente auszuwählen. Der Ablauf ist dabei so gestaltet, dass sowohl die einzelnen Experimente als auch alle Experimente gemeinsam über einen einheitlichen Befehl gestartet werden können.

6.3.1 Parametrisierung

Das Script `run.py` bietet folgende zentrale Parameter, die bei der Ausführung verwendet werden können:

`--experiment`: Mit diesem Parameter wird ein bestimmtes Experiment gezielt angesprochen. Zum Beispiel kann `-experiment experiment-easy-titanic` genutzt werden, um nur das Titanic-Experiment auszuführen.

`--model`: Dieser Parameter spezifiziert das zu verwendende Modell (z.B. `gpt-4o`). Dies ermöglicht den direkten Vergleich verschiedener Large Language Models (LLMs) im gleichen Experiment.

Wird der Parameter `--experiment` nicht angegeben, so führt `run.py` alle im Projektverzeichnis gefundenen Experimente nacheinander aus.

6.3.2 Ausführung mit einheitlichen Optionen

Zusätzlich zu den oben genannten Parametern setzt `run.py` bei jedem Lauf standardisierte Optionen, um die Ergebnisse vergleichbar zu machen. Dies sind insbesondere:

`--use-custom-preprompts`: Mit dieser Option wird sichergestellt, dass die Pre-Prompts verwendet werden, um die Eingaben für das Modell konsistent und reproduzierbar zu gestalten. Zudem stellen die Pre-Prompts sicher, dass die Experimente immer das gleiche Verständnis vom Kontext und der Ausführungsumgebung haben.

`--temperature 0`: Diese Einstellung ist besonders wichtig, da sie die Zufälligkeit bei der Modellgenerierung stark reduziert. Eine Temperatur von 0 bedeutet, dass das Modell deterministisch antwortet. Es werden stets die wahrscheinlichsten Tokens ohne zufällige Variation ausgegeben. Dies ermöglicht eine verlässliche Reproduzierbarkeit der Ergebnisse bei wiederholten Durchläufen der gleichen Eingabe.

6.3.3 Beispielaufruf

Ein typischer Aufruf von `run.py`, um ein Experiment mit einem bestimmten Modell auszuführen, sieht wie folgt aus:

```
python run.py --experiment experiment-easy-titanic --model gpt-4o
```

6.4 Preprompt Engineering

Um sicherzustellen, dass alle Experimente unter einheitlichen Bedingungen und mit klarem Kontext ausgeführt werden, kommen sogenannte Preprompts zum Einsatz. Diese Preprompts sind kurze Textbausteine oder Anweisungen, die dem Modell vor jeder eigentlichen Anfrage (Prompt) präsentiert werden. Auf diese Weise wird ein konsistentes Ausgangsniveau geschaffen, in dem alle Experimente wissen, woher die Daten stammen, wie die Verbindung zur MLflow-Instanz aufzubauen ist und wie allgemein vorzugehen ist.

Durch die Einbindung dieser Preprompts werden folgende wichtige Aspekte sichergestellt:

- Alle Experimente operieren unter denselben Rahmenbedingungen und Vorgaben. Dies umfasst die Kenntnis darüber, dass die Daten über die Kaggle-API abgerufen werden, sowie wo die heruntergeladenen Datensätze im Dateisystem zu finden sind.
- Der Zugriff auf MLflow ist klar definiert. Die Preprompts geben beispielsweise vor, dass die MLflow-Instanz unter einer bestimmten URL erreichbar ist (z.B. `http://mlflow:5000`), und dass entsprechende Logging-Operationen – wie das Setzen von Parametern, das Registrieren von Modellen oder das Logging von Metriken – konsistent gehandhabt werden.
- Die Experimente erhalten Richtlinien für die Strukturierung und Ausgabe von Code, etwa welche Dateinamen- und Ordnerstruktur zu verwenden ist oder wie Codeänderungen im vereinbarten Format (z.B. Unified Diff) bereitzustellen sind.
- Durch die einheitlichen Preprompts werden generelle Prinzipien, wie Modularität, Wiederverwendbarkeit und Best Practices in der Datenvorverarbeitung, dem Feature-Engineering oder der Modellierung, in allen Experimenten gleichermassen verankert.

Kurzum: Die Preprompts fungieren als einheitliche Wissens- und Anforderungsebene, sodass jede Ausführung eines Experiments denselben Startpunkt und denselben Handlungsspielraum besitzt. Dadurch entsteht ein konsistenter, reproduzierbarer Kontext, der über alle Experimente hinweg für Vergleichbarkeit und Nachvollziehbarkeit sorgt.

7 Experimente

Basierend auf der entwickelten Systemarchitektur wurde der Agent mit verschiedenen Modellen auf einer Reihe von Kaggle-Challenges getestet, die typische MLOps-Aufgaben repräsentieren (siehe Abb. 7.1). Diese Experimente dienten dazu, die Leistungsfähigkeit des Agenten in realitätsnahen Szenarien zu evaluieren und die Effektivität der integrierten Komponenten zu validieren.

Verschiedene LLMs wurden verwendet, um ihre Eignung für die Automatisierung spezifischer Aufgaben zu untersuchen. Die ausgewählten Kaggle-Herausforderungen wurden so gewählt, dass sie ein breites Spektrum an Anforderungen abdecken – von der Datenvorverarbeitung bis zum Modelltraining.

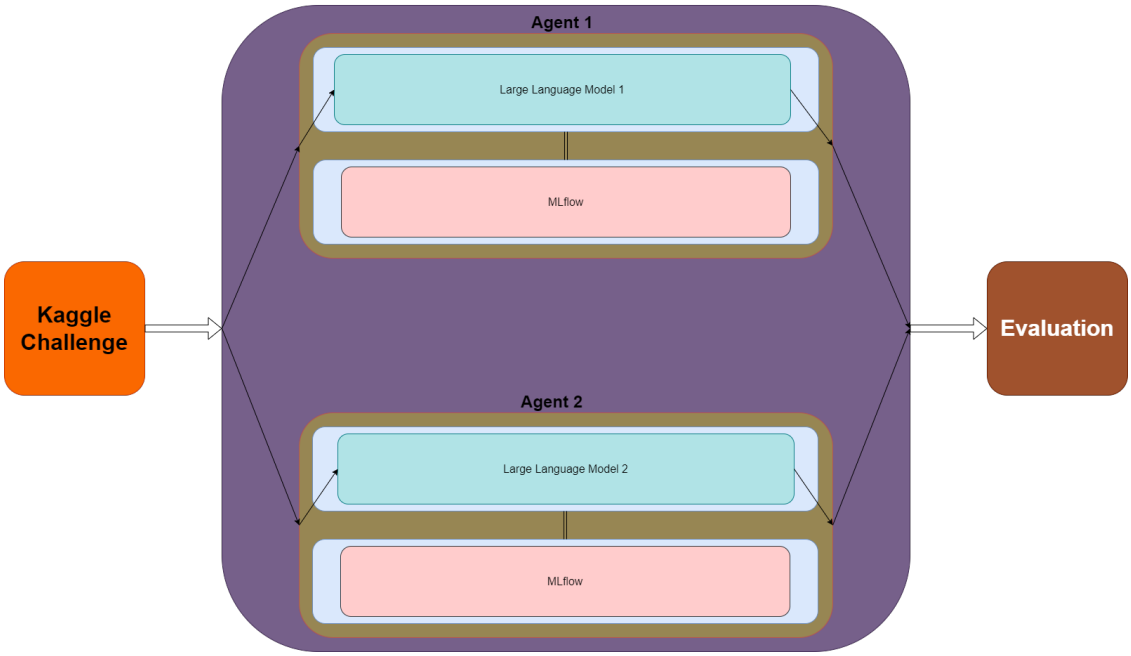

Abbildung 7.1: Experimentelle Evaluation der verschiedenen LLM-Agenten.

7.1 Titanic - Machine Learning from Disaster

Die Challenge *Titanic - Machine Learning from Disaster* zielt darauf ab, ein prädiktives Modell zu entwickeln, das vorhersagt, welche Passagiere des berühmten Schiffsunglücks überlebt haben. Basierend auf verschiedenen Passagiermerkmalen wie Alter, Geschlecht, Klasse und anderen demografischen Daten soll das Modell die Überlebenschancen der Passagiere bestimmen.

Kategorie: Klassifizierung
Challenge: `https://www.kaggle.com/competitions/titanic`

7.1.1 Metrik zur Auswertung

Die Hauptmetrik zur Auswertung für dieses Experiment ist die Klassifizierungsgenauigkeit (Accuracy), die den Prozentsatz der korrekt vorhergesagten Überlebenden im Verhältnis zur Gesamtzahl der Passagiere misst. Für jeden Passagier im Testset soll vorhergesagt werden, ob dieser überlebt (1) oder nicht (0).

7.1.2 Lösungsansatz

Ein möglicher Lösungsansatz umfasst folgende Schritte:

Datenvorverarbeitung Behandlung fehlender Werte, Codierung kategorialer Variablen (z.B. Geschlecht, Klasse) mittels One-Hot-Encoding und Standardisierung numerischer Merkmale wie Alter und Einkommen.

Feature-Engineering Erstellung zusätzlicher Features, wie beispielsweise die Anzahl der Familienmitglieder an Bord oder die Decknummer.

Modellwahl Einsatz von Klassifikationsalgorithmen wie Logistic Regression, Random Forest oder Gradient Boosting, um die Überlebenschancen vorherzusagen.

Hyperparameter-Optimierung Feinabstimmung der Modellparameter mittels Grid Search oder Random Search, um die beste Performance zu erzielen.

Evaluierung Validierung des Modells mittels Kreuzvalidierung und Überprüfung der Genauigkeit auf einem separaten Testdatensatz.

7.2 House Prices - Advanced Regression Techniques

Die Challenge *House Prices - Advanced Regression Techniques* fordert die Teilnehmer auf, den Verkaufspreis von Häusern anhand vielfältiger Merkmale vorherzusagen. Hierbei werden verschiedene fortgeschrittene Regressionsmethoden eingesetzt, um die Genauigkeit der Vorhersagen zu maximieren. Der Wettbewerb beinhaltet umfangreiche Datenvorverarbeitungs- und Feature-Engineering-Aufgaben, um die bestmöglichen Modelle zu entwickeln.

Kategorie: Regression
Challenge: `https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques`

7.2.1 Metrik zur Auswertung

Die Hauptmetrik zur Auswertung für dieses Experiment ist der Root-Mean-Squared-Error (RMSE) zwischen dem logarithmierten vorhergesagten Wert und dem logarithmierten beobachteten Verkaufspreis. Diese Metrik stellt sicher, dass Fehler bei teuren und günstigen Häusern gleichermassen berücksichtigt werden.

7.2.2 Lösungsansatz

Ein möglicher Lösungsansatz umfasst folgende Schritte:

Datenvorverarbeitung Behandlung fehlender Werte, Codierung kategorialer Variablen mittels One-Hot-Encoding oder Label-Encoding und Standardisierung numerischer Merkmale.

Feature-Engineering Erstellung neuer Features wie beispielsweise das Verhältnis von Wohnfläche zu Grundstücksfläche oder die Anzahl der Renovierungen.

Modellwahl Einsatz von Regressionsalgorithmen wie Linear Regression, Ridge Regression, Lasso Regression, Random Forest Regressor oder Gradient Boosting Regressor.

Hyperparameter-Optimierung Optimierung der Modellparameter mittels Techniken wie Grid Search oder Bayesian Optimization, um die beste Performance zu erzielen.

Evaluierung Validierung des Modells mittels Kreuzvalidierung und Überprüfung des RMSE auf einem separaten Testdatensatz.

7.3 Digit Recognizer

Die *Digit Recognizer*-Challenge konzentriert sich auf die Erkennung handgeschriebener Ziffern aus einem umfangreichen Bilddatensatz. Ziel ist es, ein Modell zu entwickeln, das in der Lage ist, die korrekten Ziffernlabels für gegebene Testbilder präzise vorherzusagen. Diese Aufgabe erfordert den Einsatz von Bildverarbeitungstechniken und tiefen neuronalen Netzwerken zur Maximierung der Klassifizierungsgenauigkeit.

Kategorie: Klassifizierung mit Bildverarbeitung
Challenge: https://www.kaggle.com/competitions/digit-recognizer

7.3.1 Metrik zur Auswertung

Die Hauptmetrik zur Auswertung für dieses Experiment ist die Kategorisierungsgenauigkeit (Accuracy), die den Prozentsatz der korrekt klassifizierten Ziffern im Verhältnis zur Gesamtzahl der Testbilder misst.

7.3.2 Lösungsansatz

Ein möglicher Lösungsansatz umfasst folgende Schritte:

Datenvorverarbeitung Normalisierung der Pixelwerte, Reshaping der Bilddaten und ggf. Datenaugmentation zur Erweiterung des Trainingsdatensatzes.

Modellwahl Einsatz von Convolutional Neural Networks (CNNs) wie LeNet, AlexNet oder modernen Architekturen wie ResNet, um die Ziffern zu erkennen.

Training Verwendung von Optimierungsalgorithmen wie Adam oder SGD und Einsatz von Techniken wie Dropout oder Batch Normalization zur Verbesserung der Generalisierungsfähigkeit.

Hyperparameter-Optimierung Feinabstimmung von Lernrate, Anzahl der Epochen, Batch-Grösse und anderen Modellparametern.

Evaluierung Validierung des Modells mittels Kreuzvalidierung und Überprüfung der Genauigkeit auf einem separaten Testdatensatz.

7.4 CIFAR-10

Die Challenge *CIFAR-10* ist ein etabliertes Computer-Vision-Dataset, das für die Objekterkennung verwendet wird. Es ist ein Teil des umfangreichen *80 Million Tiny Images*-Datensatzes und besteht aus 60.000 32x32 Farbbildern, die jeweils einer von 10 Objektklassen zugeordnet sind, mit 6.000 Bildern pro Klasse.

Das Ziel dieser Challenge ist es, ein Erkennungsmodell für Objektbilder zu entwickeln. Für jedes Bild im Testset soll ein Label für die gegebene ID vorhergesagt werden. Die vorhergesagten Labels müssen exakt mit den offiziellen Labels übereinstimmen.

Kategorie: Mehrklassen-Objekterkennung mit Bildverarbeitung
Challenge: `https://www.kaggle.com/competitions/cifar-10`

7.4.1 Metrik zur Auswertung

Die Hauptmetrik zur Auswertung für dieses Experiment ist die Klassifizierungsgenauigkeit (Accuracy), die den Prozentsatz der korrekt vorhergesagten Labels im Verhältnis zur Gesamtzahl der Testbilder misst. Für jedes Bild im Testset soll vorhergesagt werden, welche der 10 Objektklassen es repräsentiert.

7.4.2 Lösungsansatz

Ein möglicher Lösungsansatz umfasst folgende Schritte:

Datenvorverarbeitung Behandlung fehlender Werte, Normalisierung der Pixelwerte, Reshaping der Bilddaten und gegebenenfalls Datenaugmentation zur Erweiterung des Trainingsdatensatzes.

Feature-Engineering Nutzung von Techniken wie Histogram of Oriented Gradients (HOG), Principal Component Analysis (PCA) oder die Integration von vortrainierten Convolutional Neural Networks (CNNs) zur Extraktion relevanter Merkmale.

Modellwahl Einsatz von Deep Learning-Architekturen wie Convolutional Neural Networks (CNNs) oder fortgeschrittenen Architekturen wie ResNet, DenseNet oder VGGNet zur Klassifizierung der Bilder.

Training Verwendung von Optimierungsalgorithmen wie Adam oder Stochastic Gradient Descent (SGD) und Einsatz von Regularisierungstechniken wie Dropout oder Batch Normalization zur Verbesserung der Generalisierungsfähigkeit des Modells.

Hyperparameter-Optimierung Feinabstimmung von Lernrate, Anzahl der Epochen, Batch-Grösse, Anzahl der Layer und anderen Modellparametern mittels Methoden wie Grid Search, Random Search oder Bayesian Optimization, um die beste Performance zu erzielen.

Evaluierung Validierung des Modells mittels Kreuzvalidierung und Überprüfung der Genauigkeit auf einem separaten Testdatensatz.

7.5 Kaggle Score

Neben der Fülle an Daten und Herausforderungen eröffnet Kaggle den Nutzerinnen und Nutzern die Möglichkeit, die Resultate ihrer Arbeit bewerten zu lassen und diese mit den Resultaten anderer Teilnehmerinnen und Teilnehmer zu vergleichen. Die Einreichung der Ergebnisse erfolgt in einem projektspezifischen Abgabeformat. Die Bewertung der Ergebnisse erfolgt anhand einer aufgabenbezogenen Bewertungsmethode, aus der sich ein finaler Score ableitet.

Dies bietet den Nutzerinnen und Nutzern gleich mehrere Vorteile: Ein Vorteil besteht in der Möglichkeit, mehrere Versionen von Modell-Reultaten einzureichen, wodurch verschiedene Iterationen und Optimierungen direkt miteinander verglichen werden können. Des Weiteren bietet der Score durch die Einbindung in das Leaderboard die Möglichkeit, die eigene Leistung im direkten Vergleich mit anderen Teilnehmenden zu evaluieren und eine globale Platzierung zu erlangen. Diese Transparenz schafft nicht nur einen kompetitiven Anreiz, sondern fördert auch den Wissensaustausch innerhalb der Community[1].

[1] https://www.kaggle.com/discussions?sort=hotness

8 Resultate

8.1 Bewertungsraster

Zur Bewertung des vom Agent generierten Codes wurden drei Hauptkriterien definiert, die die Qualität der Problemlösung durch den Agenten abbilden sollen. Jedes Kriterium erhält eine Bewertung auf einer Skala von 1 bis 10, um die Ergebnisse quantitativ zu erfassen.

- **Codequalität**: Dieses Kriterium umfasst Aspekte wie Struktur, Dokumentation, Formatierung und die Wahl aussagekräftiger Variablennamen. Die Bewertung orientiert sich am Konzept des „Clean Code“, wie es von Robert C. Martin in seinem gleichnamigen Buch beschrieben wird [16]. Je besser die Prinzipien des Clean Code umgesetzt sind, desto höher fällt der Score aus.
- **Funktionalität**: Neben der Erfüllung der Hauptaufgaben des Agents werden hier auch ergänzende Funktionen bewertet. Dazu gehören beispielsweise die Aufbereitung von Daten, die Nachverfolgung von Modellen mittels MLflow sowie die korrekte Evaluation der Ergebnisse.
- **Innovationsgrad**: Dieses Kriterium bewertet einerseits die Einhaltung bewährter Praktiken („Best Practices“) im Programmierprozess, wie typische Muster, Namenskonventionen und etablierte Vorgehensweisen. Andererseits wird auch der kreative Umgang mit Aufgabenstellungen gewürdigt. Da dieser Punkt stark subjektiv ist, sollte die Bewertung mit Bedacht und unter Berücksichtigung individueller Interpretationen erfolgen.
- **Kaggle-Score**: Sofern ein korrektes Abgabefile generiert wurde, wird dieses bei der jeweiligen Kaggle-Challenge eingereicht, um den entsprechenden Kaggle-Score zu ermitteln. Da der Kaggle-Score je nach Challenge unterschiedlich interpretiert wird (z. B. ist bei Accuracy ein hoher Wert besser, während bei RMSE ein niedriger Wert vorteilhaft ist), erfolgt eine Normalisierung des Scores, wie in de Formel 8.1 und 8.1. Diese ermöglicht es, den Kaggle-Score einheitlich in die finale Bewertung einzubeziehen.

  Höhere Werte sind besser:

$$\text{NormScore} = 10 \cdot \frac{\text{Score} - \text{Min}}{\text{Max} - \text{Min}} \tag{8.1}$$

  Niedrigere Werte sind besser:

$$\text{NormScore} = 10 \cdot \left(1 - \frac{\text{Score} - \text{Min}}{\text{Max} - \text{Min}}\right) \tag{8.2}$$

  Dabei gelten folgende Parameter:
  – Score: Der erreichte Kaggle-Score des Modells.
  – Min: Der schlechteste Score (unter allen Teilnehmern).
  – Max: Der beste Score (unter allen Teilnehmern).

8.1.1 Berechnung des finalen Scores

Nachdem der Kaggle-Score normiert wurde, wird der finale Score als einfacher Durchschnitt zwischen allen Kategorien berechnet, wie in 8.3 zu sehen:

$$\text{FinalScore} = \frac{\text{Code-Qualität} + \text{Funktionalität} + \text{Innovationsgrad} + \text{NormScore}}{4} \tag{8.3}$$

8.2 Titanic - Machine Learning from Disaster

8.2.1 GPT-4o

Code-Qualität

Der Code ist in einer verständlichen Struktur mit klaren Funktionen und Modulen aufgebaut. Die Variablennamen sind selbsterklärend (z. B. `download_data`, `preprocess_data`) und die Kommentare beschreiben den Zweck jeder Funktion und jedes Abschnitts. Der Code behandelt mögliche Fehlerquellen wie fehlende Daten oder MLflow-Verbindungsprobleme. Eine detailliertere Dokumentation der Modellwahl sowie ausführlichere Kommentare zu Funktionen könnten jedoch noch weiter optimiert werden.

Bewertung: 7/10

Funktionalität

Die Vorverarbeitung ist umfassend und berücksichtigt sowohl numerische als auch kategorische Merkmale. Der Einsatz von `SimpleImputer`, `StandardScaler` und `OneHotEncoder` erweist sich in diesem Kontext als sinnvoll. Das Training des `RandomForestClassifiers` ist solide aufgebaut, allerdings werden keine Hyperparameter-Optimierungen oder Evaluierungen alternativer Modelle vorgenommen. Es sei an dieser Stelle angemerkt, dass die Modellwahl für diese Art von Aufgabe als zu komplex erachtet werden könnte.

Bewertung: 7/10

Innovationsgrad

Der Code folgt einem konventionellen Ansatz, wobei auf bewährte Tools und Methoden zurückgegriffen wird. Allerdings erscheint die Modellwahl etwas überqualifiziert. Die Modularisierung des Codes erlaubt eine leichte Anpassung desselben. Dennoch wäre eine Erweiterung um die Möglichkeit der Hyperparameter-Optimierung wünschenswert.

Bewertung: 6/10

Kaggle-Score

- **Kaggle-Score**: 0.76555
- **Normalisierter Score**: 7.7

8.2.2 Llama3

Code-Qualität

Die Llama3-Implementierung konnte keine lauffähige Datei erstellen. Der Versuch, mehrere Funktionen und Metriken zu importieren, führt zu einer endlosen Schleife. Dies deutet darauf hin, dass grundlegende Mechanismen zur Organisation und Modularisierung des Codes fehlen. Die redundanten Imports verhindern, dass der Code überhaupt ausgeführt werden kann. Es fehlt eine klare Trennung zwischen verschiedenen Modulen und keine Datei ist ausführbar.

Bewertung: 0/10

Kaggle-Score

Da keine funktionsfähige `submission.csv` erstellt wurde, ist eine Bewertung auf Kaggle nicht möglich.

- **Kaggle-Score**: Nicht verfügbar
- **Normalisierter Score**: 0.0

8.3 House Prices - Advanced Regression Techniques

8.3.1 GPT-4o

Code-Qualität

Auch hier ist der Code nachvollziehbar strukturiert und mit erläuternden Kommentaren versehen. Die einzelnen Schritte, wie beispielsweise die Datenvorbereitung, das Modelltraining sowie die Evaluation, sind klar voneinander abgegrenzt. Die Benennung der Variablen ist aussagekräftig und die Nutzung von Modulen wie `Pipeline` und `ColumnTransformer` gewährleistet eine übersichtliche Darstellung. Des Weiteren überprüft der Code MLflow-Verbindungen und behandelt Fälle, in denen Modelle nicht registriert werden können.

Bewertung: 8/10

Funktionalität

Die Trennung von numerischen und kategorischen Merkmalen sowie deren spezifische Verarbeitung, insbesondere die Skalierung und das One-Hot-Encoding, sind in sinnvoller Weise umgesetzt. Die Verarbeitung berücksichtigt zudem fehlende Werte, welche durch Imputer ergänzt werden, wodurch die Robustheit des Verfahrens erhöht wird. Der Einsatz eines `RandomForestRegressors` stellt eine bewährte Methode zur Lösung von Regressionsproblemen dar. Allerdings ist auch hier keine intuitive Anpassung der Hyperparameter möglich.

Bewertung: 7/10

Innovationsgrad

Der Code folgt Standardmethoden und implementiert einen klassischen Machine-Learning-Workflow. Die Modularität des Codes erlaubt eine leichte Anpassung des Workflows. Allerdings fehlen innovative Ansätze wie fortgeschrittenes Feature-Engineering oder der Einsatz moderner Algorithmen.

Bewertung: 6/10

Kaggle-Score

- **Kaggle-Score**: 0.15080
- **Normalisierter Score**: 8.5

8.3.2 Llama3

Code-Qualität

Der Code von Llama3 zeigt eine grundsätzlich brauchbare Struktur, jedoch fehlen entscheidende technische Aspekte, die die Flexibilität und Skalierbarkeit verbessern könnten. Die Verwendung von Modulen wie `ColumnTransformer` und `Pipelines` für die Vorverarbeitung von Daten ist ein positives Merkmal, jedoch gibt es keine Mechanismen zur dynamischen Anpassung der Pipeline oder der Parameter.

Beispiel für die mangelnde Flexibilität ist der `RandomForestRegressor`, welcher mit fixen Standardparametern initialisiert wird:

```
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

Hier fehlen Parameter wie `max_depth` oder `min_samples_split`, die durch ein Konfigurationsfile oder Argumente dynamisch gesetzt werden könnten. GPT-4o implementiert eine flexiblere Herangehensweise, indem Hyperparameter aus einer Konfigurationsdatei gelesen werden, was Anpassungen ohne Codeänderungen erlaubt. Ein Beispiel für eine fehlende dynamische Anpassung wäre:

```
params = {"n_estimators": 200, "max_depth": 10, "random_state": 42}
model = RandomForestRegressor(**params)
```

Darüber hinaus fehlt im Code die Behandlung von Fehlerfällen, wie z. B. einer unterbrochenen Verbindung zu MLflow oder Problemen bei der Datenverarbeitung. So wird nicht überprüft, ob alle erwarteten Spalten in den Datenframes vorhanden sind, was potenziell zu Laufzeitfehlern führen kann.

Bewertung: 6/10

Funktionalität

Die grundlegenden Schritte – wie die Trennung numerischer und kategorialer Merkmale, die Anwendung spezifischer Transformationen (z. B. One-Hot-Encoding und Skalierung) sowie die Nutzung eines `RandomForestRegressor` – sind korrekt umgesetzt. Jedoch zeigt sich ein Mangel an Flexibilität bei der Anpassung von Hyperparametern und der Verarbeitungspipeline.

Ein Beispiel für die starren Transformationen im Preprocessing:

```
preprocessor = ColumnTransformer( transformers=[
    ('num', numerical_transformer, numerical_features),
    ('cat', categorical_transformer, categorical_features)
] )
```

Die Liste der `numerical_features` und `categorical_features` ist statisch definiert. Es fehlt eine dynamische Methode, um Spalten automatisch zu erkennen oder flexibel auszuwählen. Bei GPT-4o wird die Spaltendefinition beispielsweise durch eine Funktion implementiert, die Datenframe-Metadaten verwendet:

```
def detect_features(df):
    numerical = df.select_dtypes(include=['int64', 'float64']).columns
    categorical = df.select_dtypes(include=['object', 'category']).columns
    return numerical, categorical

numerical_features, categorical_features = detect_features(train_df)
```

Ausserdem fehlt die Möglichkeit, Pipelines oder Modelle über eine Konfigurationsdatei (`JSON` oder `YAML`) anzupassen. Ein solcher Ansatz würde eine bessere Benutzerfreundlichkeit gewährleisten und den Workflow modularer machen.

Bewertung: 5/10

Innovationsgrad

Der Code von Llama3 orientiert sich strikt an Standardmethoden und implementiert einen klassischen Machine-Learning-Workflow, ohne signifikante Innovationen einzuführen. Es fehlen moderne Ansätze wie:

Einsatz moderner Algorithmen Der `RandomForestRegressor` ist ein bewährter Algorithmus, jedoch könnten fortgeschrittenere Methoden wie Gradient Boosting (z. B. `XGBoost` oder `LightGBM`) implementiert werden, die in Kaggle-Wettbewerben oft bessere Ergebnisse erzielen.

Fortgeschrittenes Feature-Engineering Beispielsweise die Erzeugung von Interaktionstermen oder die automatische Auswahl wichtiger Features mittels eines `SelectFromModel`.

Pipeline-Erweiterbarkeit Die Pipeline könnte durch eine Cross-Validation-Komponente erweitert werden, um die Generalisierbarkeit des Modells besser zu bewerten.

Ein fehlendes Beispiel ist die Integration von Cross-Validation in die Pipeline:

```
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(
    model, X_train, y_train, cv=5, scoring="neg_root_mean_squared_error"
)
print("Cross-Validation RMSE:", -np.mean(cv_scores))
```

Der Mangel an solchen Techniken zeigt, dass der Innovationsgrad des Codes im Vergleich zu GPT-4o gering ist.

Bewertung: 5/10

Kaggle-Score

Die experimentelle Implementierung mit Llama3 konnte keine funktionierende `submission.csv` generieren, was eine direkte Evaluierung auf Kaggle verhinderte. Obwohl Vorhersagen getroffen wurden, scheint der Codefehler in der Generierung der CSV-Datei zu liegen:

```
submission_df = pd.DataFrame({"Id": test_df["Id"], "SalePrice": predictions})
submission_df.to_csv("submission.csv", index=False)
```

Ein potenzieller Fehler liegt in der Verarbeitung der Testdaten, da `test_df` möglicherweise keine Spalte `Id` enthält. Die fehlende Robustheit in der Fehlerbehandlung hat einen erheblichen Einfluss auf die Nutzbarkeit der Ergebnisse.

- **Kaggle-Score**: Nicht verfügbar
- **Normalisierter Score**: 0.0

8.4 Digit Recognizer

8.4.1 GPT-4o

Code-Qualität

Der Code zeichnet sich durch eine übersichtliche und logische Struktur aus. Kommentare stellen die wichtigsten Funktionen erklärend dar, jedoch könnten detailliertere Beschreibungen der Preprocessing- und Modellschritte hinzugefügt werden, um die Nachvollziehbarkeit zu verbessern. Die modulare Struktur des Codes trägt zu seiner Wartungsfreundlichkeit bei. Der Workflow ist in klar abgegrenzte Funktionen für das Herunterladen, Vorverarbeiten, Trainieren und Evaluieren aufgeteilt.

Bewertung: 8/10

Funktionalität

Der Vorgang der Aufteilung der Daten in Trainings- und Validierungsdaten wird korrekt ausgeführt. Die Verwendung eines LogisticRegression-Modells erweist sich als eine grundlegende, jedoch funktionale Wahl. Es gibt grundlegende Fehlerprüfungen für MLflow-Verbindungen und Datenhandling.

Bewertung: 8/10

Innovationsgrad

Der hier dargelegte Ansatz verbleibt auf einem grundlegenden Niveau und verzichtet auf innovative Strategien, wie beispielsweise fortgeschrittenes Feature-Engineering oder den Einsatz moderner Modelle, wie neuronaler Netze. Die Wahl eines logistischen Regressionsmodells ist als solide, wenngleich wenig kreativ zu bewerten.

Bewertung: 5/10

Kaggle-Score

- **Kaggle-Score**: 0.90396
- **Normalisierter Score**: 9.0

8.4.2 Llama3

Code-Qualität

Der Code von Llama3 weist eine Grundstruktur auf, die den Anforderungen der Aufgabe entspricht, jedoch fehlt es an modularer Organisation und flexibler Handhabung von Parametern.

Ein positiver Punkt ist die konsequente Verwendung von MLflow zur Protokollierung von Parametern und Metriken. Allerdings ist die Implementierung weniger robust als bei GPT-4o. So fehlen beispielsweise Mechanismen zur Validierung von Eingabedaten. Ein Beispiel für fehlende Robustheit:

```
X_train = train_df.drop("Label", axis=1) / 255.0
y_train = train_df["Label"]
X_test = test_df / 255.0
```

Hier wird eine direkte Division durch `255.0` durchgeführt, ohne Überprüfung, ob die Spalten korrekt geladen oder vollständig sind. Bei GPT-4o wird durch eine separate Funktion für das Preprocessing sichergestellt, dass solche Fehler erkannt und behandelt werden.

Ein weiterer Schwachpunkt ist die redundante und nicht dokumentierte Speicherung von Artefakten wie der `submission.csv`, ohne zu prüfen, ob die Datei bereits existiert oder überschrieben wird. Die fehlende Trennung von Preprocessing, Training und Evaluation erschwert die Wartbarkeit und Wiederverwendbarkeit des Codes.

Bewertung: 6/10

Funktionalität

Die grundlegenden Schritte, wie das Herunterladen von Daten mit der Kaggle-API, die Normalisierung von Pixelwerten und die Aufteilung der Daten in Trainings- und Validierungsdaten, wurden korrekt implementiert.

Ein klarer Schwachpunkt ist die fehlende Dynamik bei der Wahl und Anpassung der Modelle. Der Code setzt standardmässig auf `LogisticRegression`, ohne alternative Modelle oder Hyperparameter-Optimierung zu ermöglichen. Ein Beispiel:

```
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

Im Vergleich dazu bietet GPT-4o eine klar strukturierte Funktion für das Trainieren und Evaluieren von Modellen, die eine spätere Erweiterung durch zusätzliche Algorithmen erleichtert. Ein weiteres Problem ist die fehlende Robustheit der `test.py`-Datei. Diese lädt das Modell direkt aus MLflow, jedoch ohne jegliche Prüfung, ob das Modell korrekt registriert oder geladen wurde. Dies könnte zu Laufzeitfehlern führen, wenn MLflow nicht ordnungsgemäss konfiguriert ist. Zudem ist die Generierung der `submission.csv` fehlerhaft, da die Datei keine sinnvollen Vorhersagen enthält, was eine direkte Evaluierung auf Kaggle unmöglich macht. Dies deutet auf einen grundlegenden Fehler in der Pipeline hin, entweder bei der Modellinferenz oder beim Speichern der Ergebnisse.

Bewertung: 5/10

Innovationsgrad

Der Ansatz von Llama3 beschränkt sich auf die Verwendung von Standardtechniken wie `LogisticRegression` und Normalisierung. Moderne Methoden, wie etwa Convolutional Neural Networks (CNNs), die für Bilddaten deutlich besser geeignet wären, wurden nicht implementiert. Zudem fehlt die Nutzung von fortgeschrittenem Feature-Engineering oder einer robusteren Datenaugmentation.

GPT-4o verwendet ähnliche Standardmethoden, bietet jedoch eine klarere Trennung der Schritte und ermöglicht eine flexiblere Anpassung. Llama3 bleibt in Bezug auf Innovation deutlich hinter den Möglichkeiten zurück.

Bewertung: 5/10

Kaggle-Score

Die experimentelle Implementierung mit Llama3 konnte keine funktionsfähige `submission.csv` generieren, was eine direkte Evaluierung auf Kaggle verhinderte. Es wurde lediglich die Datei `sample_submission.csv` kopiert, jedoch nicht mit den berechneten Werten aus dem Modell abgefüllt.

- **Kaggle-Score**: Nicht verfügbar
- **Normalisierter Score**: 0.0

8.5 CIFAR-10

8.5.1 GPT-4o

Code-Qualität

Der Code von GPT-4o ist insgesamt gut strukturiert und modular aufgebaut. Schritte wie das Herunterladen und Vorverarbeiten der Daten sowie das Trainieren und Bewerten des Modells sind klar voneinander getrennt. Ein positiver Punkt ist die Verwendung eines vortrainierten VGG16-Modells, was die Komplexität der Implementierung erhöht, jedoch auch den potenziellen Nutzen moderner Ansätze aufzeigt. Allerdings führt ein Fehler in der Datenpipeline zu einem Laufzeitfehler:

```
TypeError: All values in column x_col=id must be strings.
```

Dieser Fehler tritt auf, weil der Spaltenwert `x_col` nicht korrekt vorbereitet wurde, was auf eine fehlende Validierung und Datenprüfung hinweist. Ein besserer Ansatz wäre, den Inhalt der Spalte vor der Übergabe an `flow_from_dataframe` zu überprüfen und zu formatieren:

```
train_df["id"] = train_df["id"].astype(str)
```

Bewertung: 7/10

Funktionalität

Die grundlegende Pipeline ist solide aufgebaut, mit der Nutzung von `ImageDataGenerator` für die Datenaugmentation und der Integration von MLflow zur Nachverfolgung von Metriken und Modellen. Dennoch scheitert die Funktionalität an der fehlenden Prüfung und Verarbeitung der Testdaten. Der Fehler in der `create_submission`-Funktion führt dazu, dass keine korrekten Vorhersagen generiert werden können. Auch fehlen Mechanismen zur Hyperparameter-Optimierung.

Ein Vorschlag zur Verbesserung wäre die Verwendung einer Validierungsfunktion für die Datengeneratoren:

```
def validate_dataframe(df, required_columns):
    missing_columns = [
        col for col in required_columns if col not in df.columns
    ]
    if missing_columns:
        raise ValueError(f"Missing required columns: {missing_columns}")
```

Bewertung: 6/10

Innovationsgrad

Die Nutzung von VGG16 als Basismodell ist ein guter Ansatz für Bildklassifizierungsprobleme und zeigt, dass moderne Modelle genutzt werden können. Dennoch fehlen weitergehende Innovationen, wie der Einsatz von Transfer Learning mit feinabgestimmten Layern oder die Integration eines automatisierten Feature-Engineering-Prozesses. Die Implementierung bleibt somit im Bereich eines soliden, aber nicht aussergewöhnlichen ML-Workflows.

Bewertung: 6/10

Kaggle-Score

Aufgrund der fehlerhaften Datenverarbeitung konnte keine funktionsfähige `submission.csv` erstellt werden, was eine Bewertung auf Kaggle unmöglich macht.

- **Kaggle-Score**: Nicht verfügbar
- **Normalisierter Score**: 0.0

8.5.2 Llama3

Code-Qualität

Der Code von Llama3 zeigt eine grundlegende Struktur mit der Integration von MLflow zur Nachverfolgung von Metriken und der Nutzung von PyTorch für die Modellimplementierung. Allerdings fehlen wichtige Validierungs- und Fehlerbehandlungsmechanismen, insbesondere bei der Datenverarbeitung und beim Umgang mit Artefakten wie `submission.csv`. Auch die Strukturierung ist weniger klar als bei GPT-4o, da Schritte wie Datenvorverarbeitung und Modelltraining direkt im `main.py`-Skript integriert sind, was die Modularität einschränkt.

Bewertung: 6/10

Funktionalität

Die Verwendung eines benutzerdefinierten CNN-Modells zeigt, dass der Ansatz für Bilddaten geeignet ist. Allerdings fehlen hier fortgeschrittene Techniken wie Datenaugmentation oder die Optimierung von Hyperparametern. Darüber hinaus ist die Datenpipeline nicht robust genug, da keine Mechanismen implementiert wurden, um sicherzustellen, dass die Eingabedaten vollständig und korrekt formatiert sind.

Bewertung: 6/10

Innovationsgrad

Das benutzerdefinierte CNN-Modell ist eine innovative Ergänzung im Vergleich zur Nutzung eines vortrainierten Modells, wie es bei GPT-4o der Fall ist. Allerdings fehlen weitergehende Optimierungen und fortgeschrittene Ansätze, wie etwa der Einsatz von Transfer Learning oder die Nutzung moderner Architekturen wie ResNet oder EfficientNet. Auch die fehlende Automatisierung in der Pipeline beschränkt den Innovationsgrad.

Bewertung: 7/10

Kaggle-Score

Auch hier konnte keine korrekte `submission.csv` generiert werden. Obwohl Vorhersagen auf dem Testdatensatz durchgeführt wurden, fehlen Validierungsmechanismen, die sicherstellen, dass die Ergebnisse in das korrekte Format überführt werden.

- **Kaggle-Score**: Nicht verfügbar
- **Normalisierter Score**: 0.0

8.6 Bewertungsraster

Die nachfolgende Tabelle 8.1 präsentiert die erzielten Bewertungen und ermöglicht einen direkten Vergleich der Leistung der eingesetzten Modelle. Sie illustriert die Unterschiede in Codequalität, Funktionalität und Innovationsgrad für die betrachteten Herausforderungen.

Tabelle 8.1: Vergleich der Ergebnisse von GPT-4o und Llama3 für verschiedene Challenges.

Challenge	Kriterium	GPT-4o	Llama3
Titanic - Machine Learning from Disaster	Code-Qualität	7.0	0.0
	Funktionalität	7.0	0.0
	Innovationsgrad	6.0	0.0
	Kaggle-Score	7.7	0.0
House Prices - Advanced Regression Techniques	Code-Qualität	8.0	6.0
	Funktionalität	7.0	5.0
	Innovationsgrad	6.0	5.0
	Kaggle-Score	8.5	0.0
Digit Recognizer	Code-Qualität	8.0	6.0
	Funktionalität	8.0	5.0
	Innovationsgrad	5.0	5.0
	Kaggle-Score	9.0	0.0
CIFAR-10	Code-Qualität	7.0	6.0
	Funktionalität	6.0	6.0
	Innovationsgrad	6.0	7.0
	Kaggle-Score	0.0	0.0

Die Tabelle 8.2 zeigt die durchschnittlichen Endbewertungen der einzelnen Agenten für die jeweiligen Herausforderungen und ermöglicht eine differenzierte Gegenüberstellung der Ergebnisse.

Tabelle 8.2: Durchschnittlicher Score der Agents pro Challenge

Challenge	**GPT-4o**	**Llama3**
Titanic - Machine Learning from Disaster	6.9	0.0
House Prices - Advanced Regression Techniques	7.4	4.0
Digit Recognizer	7.5	4.0
CIFAR-10	4.8	4.8

9 Diskussion

9.1 Qualität der Pipeline und MLflow Integration

Die Integration von MLflow war ein zentraler Bestandteil der Experimente und zeigte, dass GPT-4o eine konsistente und robuste Implementierung liefert, wie in Tabelle 9.1 verglichen wird. Parameter, Modelle und Metriken wurden zuverlässig protokolliert. Bei Llama3 fehlten jedoch grundlegende Mechanismen zur Sicherstellung einer stabilen Verbindung, was zu wiederholten Ausfällen führte.

Tabelle 9.1: Vergleich der MLflow-Integration zwischen GPT-4o und Llama3

Aspekt	**GPT-4o**	**Llama3**
MLflow-Verbindungsprüfung	Implementiert (inkl. Fallback bei Fehlern)	Nicht implementiert, führt zu Ausfällen
Modellregistrierung	Erfolgreich mit Fehlerbehandlung	Keine Registrierung möglich
Nutzung erweiterter MLflow-Funktionen	Grundlegende Parameter- und Metrikprotokollierung	Keine erweiterten Funktionen genutzt
Automatisierung	Automatische Modellprotokollierung	Manuelle oder fehlende Schritte

9.2 Codequalität

Die Analyse der Codequalität in Tabelle 9.2 offenbarte deutliche Unterschiede zwischen den beiden Modellen. GPT-4o lieferte strukturierte und gut dokumentierte Lösungen, während Llama3 oft unstrukturierte oder fehlerhafte Codes erzeugte.

Tabelle 9.2: Vergleich der Codequalität zwischen GPT-4o und Llama3

Aspekt	**GPT-4o**	**Llama3**
Strukturierung	Modular und nachvollziehbar	Unstrukturiert, redundante Imports
Fehlerbehandlung	Robuste Behandlung (z. B. MLflow-Validierung)	Keine Fehlerprüfung
Dokumentation	Grundlegende Kommentare, ausbaufähig	Kaum Dokumentation
Datenvalidierung	Implementiert	Fehlend, führt zu Laufzeitfehlern

9.3 Modellqualität

Die Modelle zeigten eine hohe Abhängigkeit von der Experimentkonfiguration. Während GPT-4o robuste Ergebnisse bei Titanic und House Prices erzielte, scheiterte Llama3 bereits an grundlegenden Modellanforderungen, wie in Tabelle 9.3 beschrieben.

Tabelle 9.3: Vergleich der Modellqualität zwischen GPT-4o und Llama3

Aspekt	GPT-4o	Llama3
Vorhersagegenauigkeit	Hoch (z. B. 0.76555 bei Titanic)	Keine validen Vorhersagen
Modellkomplexität	Nutzung bewährter Modelle (z. B. Logistic Regression, Random Forest)	Fehlende Implementierung oder ineffizient
Hyperparameter-Optimierung	Fehlend, jedoch Anpassungsmöglichkeiten vorhanden	Keine Unterstützung
Komplexe Daten (z. B. CIFAR-10)	Grundansätze vorhanden, jedoch fehlerhaft	Überforderung bei Bildverarbeitung

9.4 Schwierigkeitsgrenze

Die Analyse der Experimente zeigte, dass die LLMs mit zunehmender Komplexität der Aufgaben deutliche Schwierigkeiten haben. Während einfache Aufgaben wie Titanic und House Prices gut gelöst wurden, scheiterten beide Modelle bei CIFAR-10 und Detecting Misinformation. Dies ist in 9.4 weiter beschrieben.

Tabelle 9.4: Analyse der Schwierigkeitsgrenze bei GPT-4o und Llama3

Aufgabentyp	GPT-4o	Llama3
Einfache Aufgaben	Erfolgreich (z. B. Titanic)	Fehlerhaft
Mittlere Aufgaben	Teilerfolg (z. B. Sentiment Analysis)	Schwache Ergebnisse
Schwierige Aufgaben	Grundsätzliche Implementierung vorhanden (CIFAR-10), aber fehlerhaft	Vollständiges Scheitern
Abhängigkeit von Pre-Prompts	Hohe Abhängigkeit, aber robuster	Sehr hohe Abhängigkeit, scheitert ohne klare Vorgaben

10 Ausblick

Eine Analyse des abschliessenden Bewertungsrasters 8.2 zeigt die signifikante Abhängigkeit der Qualität eines LLM-Agenten von der Wahl des zugrundeliegenden LLM-Modells. Die durchgeführten Experimente demonstrieren, dass LLMs dazu fähig sind, Codeprobleme bis zu einem gewissen Schwierigkeitsgrad zu lösen. Mit zunehmender Komplexität der Aufgabenstellung zeigt sich jedoch eine drastische Abnahme der Lösungsqualität. Dies manifestiert sich in Fällen, in denen der LLM-Agent nicht mehr in der Lage ist, ausführbaren bzw. funktionsfähigen Code zu generieren. Es ist besonders auffällig, dass einfache, klar formulierte Aufgabenstellungen von den getesteten Modellen (mit Ausnahme von LLama3, siehe Referenz 8.2.2) mit hoher Genauigkeit bewältigt werden, während komplexere Szenarien häufig die Grenzen der aktuellen Modellgeneration aufzeigen.

Das Ziel, die Effizienz und Skalierbarkeit von MLOps-Prozessen durch den Einsatz von LLM-Agenten zu erhöhen, konnte im Rahmen dieser Arbeit bedingt erreicht werden (siehe 3.2). Die Implementierung eines modularen und containerisierten LLM-Agenten ermöglichte die Automatisierung diverser typischer MLOps-Aufgaben, wie beispielsweise die Codegenerierung, das Modell- und Metriktracking mittels MLflow. Die entwickelte Methode zeichnet sich durch ihre Flexibilität und Systemunabhängigkeit aus, was vielversprechende Einsatzmöglichkeiten in verschiedenen Anwendungsgebieten eröffnet. Unser Ansatz zur modularen und anpassbaren Architektur eines LLM-Agents zeigt praxisnahe Lösungswege für die Automatisierung von MLOps-Prozessen auf. Der Einsatz von Technologien wie MLflow zur Nachvollziehbarkeit und Dokumentation der Ergebnisse setzt hierbei neue Standards in der Forschung und Anwendung.

Die aufgestellte Hypothese, dass die Leistungsfähigkeit eines LLM-Agents in direktem Zusammenhang mit den spezifischen Eigenschaften des verwendeten LLMs steht, wurde durch die empirische Untersuchung bestätigt. Die Ergebnisse von GPT-4o und Llama3 demonstrieren die signifikante Modellabhängigkeit. Während GPT-4o konsistente Ergebnisse bei mittlerem Schwierigkeitsgrad lieferte, wiesen die Experimente mit Llama3 deutliche Schwächen in der Codequalität und der Generierung funktionaler Lösungen auf. Diese Diskrepanzen betonen die Notwendigkeit, für spezifische Aufgabenstellungen gezielt passende Modelle auszuwählen und deren Grenzen im Vorfeld zu analysieren.

10.1 Wissenschaftlicher Beitrag

Der vorliegende Beitrag leistet einen Beitrag zur Weiterentwicklung des Forschungsfeldes rund um LLMs und deren Integration in MLOps-Pipelines. Die Ergebnisse der Untersuchung bilden eine fundierte Grundlage für zukünftige Forschungsarbeiten und eröffnen neue Perspektiven für die Automatisierung maschineller Lernprozesse. Der LLM-Agent bildet die Grundlage für vertiefende Untersuchungen in zukünftigen Studien. Diese sollen erforschen, inwiefern innovative Prompt-Strategien oder spezialisierte Modellarchitekturen die Effizienz und Zuverlässigkeit weiter steigern können.

Die in dieser Arbeit vorgestellten Methoden bieten potenziellen Nutzen für Unternehmen, die MLOps-Abläufe effizienter gestalten möchten. Insbesondere die containerisierte Architektur erlaubt eine schnelle Implementierung und Anpassung an spezifische Bedürfnisse. Zukünftige Arbeiten sollten die Integration fortschrittlicher Modellarchitekturen und die Anwendung auf

realweltliche Datenpipelines vertiefen. Die Entwicklung adaptiver Multi-Agenten-Systeme könnte darüber hinaus weitere Potenziale für die Skalierbarkeit und Effektivität von MLOps-Prozessen eröffnen.

10.2 Limitationen

Die Limitationen dieser Arbeit betreffen insbesondere die Skalierbarkeit der Ansätze auf hochkomplexe und dynamische MLOps-Aufgaben. Zudem war die Evaluation auf eine begrenzte Auswahl von LLMs und spezifische Use Cases beschränkt. Für eine umfassendere Validierung wären breiter angelegte Experimente mit zusätzlichen Modellen und heterogeneren Aufgabenstellungen notwendig.

11 Schlussfolgerung

Die vorliegende Arbeit trägt wesentlich zur wissenschaftlichen und praktischen Weiterentwicklung von MLOps-Pipelines bei, indem sie aufzeigt, wie LLMs als MLOps-Agenten eingesetzt werden können, um Effizienz und Automatisierung zu steigern. Die Untersuchung hat demonstriert, dass die Integration von LLMs in typischen MLOps-Workflows, wie Datenvorverarbeitung, Modelltraining und Monitoring, eine bemerkenswerte Reduktion manueller Eingriffe ermöglicht und somit die Standardisierung und Wiederholbarkeit fördert.

Für die Praxis bedeutet dies, dass Unternehmen und Entwickler von optimierten Prozessen profitieren können, die sowohl zeit- als auch ressourcensparend sind. Gleichzeitig zeigt die Arbeit, dass die Flexibilität und Skalierbarkeit solcher Pipelines durch den modularen Ansatz und die Verwendung von Containern und MLflow gestärkt werden. Die gewonnenen Erkenntnisse bieten eine solide Grundlage für weitere Forschung und die Implementierung spezialisierter LLM-Agenten in komplexeren MLOps-Szenarien.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] K. Salama, J. Kazmierczak und D. Schut, „Practitioners guide to MLOps: A framework for continuous delivery and automation of machine learning.," Google Cloud, White paper, Mai 2021, S. 37. (besucht am 30. 10. 2024).

[2] C. O'Brien. „AI startups raised $18.5 billion in 2019, setting new funding record," VentureBeat. (14. Jan. 2020), Adresse: https://venturebeat.com/ai/ai-startups-raised-18-5-billion-in-2019-setting-new-funding-record/ (besucht am 30. 10. 2024).

[3] J. S. Chan, N. Chowdhury, O. Jaffe u. a., *MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering*, 2024. DOI: 10.48550/ARXIV.2410.07095. Adresse: https://arxiv.org/abs/2410.07095 (besucht am 30. 10. 2024).

[4] „Number of ChatGPT users (oct 2024)," Exploding Topics. (30. März 2023), Adresse: https://explodingtopics.com/blog/chatgpt-users (besucht am 31. 10. 2024).

[5] E. La Malfa, A. Petrov, S. Frieder u. a., „Language-Models-as-a-Service: Overview of a New Paradigm and its Challenges," *Journal of Artificial Intelligence Research*, Jg. 80, S. 1497–1523, 26. Aug. 2024, ISSN: 1076-9757. DOI: 10.1613/jair.1.15865. Adresse: https://www.jair.org/index.php/jair/article/view/15865 (besucht am 31. 10. 2024).

[6] „What is fine-tuning? | IBM." (15. März 2024), Adresse: https://www.ibm.com/topics/fine-tuning (besucht am 31. 10. 2024).

[7] A. Vaswani, N. Shazeer, N. Parmar u. a., *Attention Is All You Need*, 2017. DOI: 10.48550/ARXIV.1706.03762. Adresse: https://arxiv.org/abs/1706.03762 (besucht am 19. 11. 2024).

[8] T. B. Brown, B. Mann, N. Ryder u. a., *Language Models are Few-Shot Learners*, 2020. DOI: 10.48550/ARXIV.2005.14165. Adresse: https://arxiv.org/abs/2005.14165 (besucht am 19. 11. 2024).

[9] J. L. Ba, J. R. Kiros und G. E. Hinton, *Layer Normalization*, 2016. DOI: 10.48550/ARXIV.1607.06450. Adresse: https://arxiv.org/abs/1607.06450 (besucht am 19. 11. 2024).

[10] E. M. Bender, T. Gebru, A. McMillan-Major und S. Shmitchell, „On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, Ser. FAccT '21, Virtual Event, Canada: Association for Computing Machinery, 2021, S. 610–623, ISBN: 9781450383097. DOI: 10.1145/3442188.3445922. Adresse: https://doi.org/10.1145/3442188.3445922.

[11] K. Mei, Z. Li, S. Xu, R. Ye, Y. Ge und Y. Zhang, *AIOS: LLM Agent Operating System*, 2024. DOI: 10.48550/ARXIV.2403.16971. Adresse: https://arxiv.org/abs/2403.16971 (besucht am 03. 11. 2024).

[12] Z. Liu, Y. Zhang, P. Li, Y. Liu und D. Yang, *Dynamic LLM-Agent Network: An LLM-agent Collaboration Framework with Agent Team Optimization*, 2023. DOI: 10.48550/ARXIV.2310.02170. Adresse: https://arxiv.org/abs/2310.02170 (besucht am 03. 11. 2024).

[13] „Welcome to GPT-ENGINEER's Documentation — gpt-engineer 0.3.1 documentation." (), Adresse: https://gpt-engineer.readthedocs.io/en/latest/# (besucht am 19. 09. 2024).

[14] „MLflow: A Tool for Managing the Machine Learning Lifecycle.“ (), Adresse: https://mlflow.org/docs/latest/index.html (besucht am 27. 11. 2024).

[15] „Quick start,“ OpenRouter. (), Adresse: https://openrouter.ai (besucht am 27. 11. 2024).

[16] R. C. Martin, *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code* (mitp Professional), Online-Ausg. Verlagsgruppe Hüthig Jehle Rehm, 2013, 1 S., ISBN: 9783826655487 9783826696398.