Diplomarbeit

Tapyre

Entwicklung eines KI-integrierten Produktivitätstools mit Plugin-System

Eingereicht von

Christian Vorhofer
Raphael Ladinig

Eingereicht bei

Höhere Technische Bundeslehr- und Versuchsanstalt Anichstraße

Abteilung für Wirtschaftsingenieure/Betriebsinformatik

Betreuer

GREINÖCKER Albert, Mag. Dr. DI

Innsbruck, April 2026

Abgabevermerk:

Betreuer/in:

Datum:

Kurzfassung /Abstract

Eine Kurzfassung ist in deutscher sowie ein Abstract in englischer Sprache mit je maximal einer A4-Seite zu erstellen. Die Beschreibung sollte wesentliche Aspekte des Projektes in technischer Hinsicht beschreiben. Die Zielgruppe der Kurzbeschreibung sind auch Nicht-Techniker! Viele Leser lesen oft nur diese Seite.

Beispiel für ein Abstract (DE und EN)

Die vorliegende Diplomarbeit beschäftigt sich mit verschiedenen Fragen des Lernens Erwachsener – mit dem Ziel, Lernkulturen zu beschreiben, die die Umsetzung des Konzeptes des Lebensbegleitenden Lernens (LBL) unterstützen. Die Lernfähigkeit Erwachsener und die unterschiedlichen Motive, die Erwachsene zum Lernen veranlassen, bilden den Ausgangspunkt dieser Arbeit. Die anschließende Auseinandersetzung mit Selbstgesteuertem Lernen, sowie den daraus resultierenden neuen Rollenzuschreibungen und Aufgaben, die sich bei dieser Form des Lernens für Lernende, Lehrende und Institutionen der Erwachsenenbildung ergeben, soll eine erste Möglichkeit aufzeigen, die zur Umsetzung dieses Konzeptes des LBL beiträgt. Darüber hinaus wird im Zusammenhang mit selbstgesteuerten Lernprozessen Erwachsener die Rolle der Informations- und Kommunikationstechnologien im Rahmen des LBL näher erläutert, denn die Eröffnung neuer Wege zur orts- und zeitunabhängiger Kommunikation und Kooperation der Lernenden untereinander sowie zwischen Lernenden und Lernberatern gewinnt immer mehr an Bedeutung. Abschließend wird das Thema der Sichtbarmachung, Bewertung und Anerkennung des informellen und nicht-formalen Lernens aufgegriffen und deren Beitrag zum LBL erörtert. Diese Arbeit soll einerseits einen Beitrag zur besseren Verbreitung der verschiedenen Lernkulturen

leisten und andererseits einen Reflexionsprozess bei Erwachsenen, die sich lebensbegleitend weiterbilden, in Gang setzen und sie somit dabei unterstützen, eine für sie geeignete Lernkultur zu finden.

This thesis deals with the various questions concerning learning for adults – with the aim to describe learning cultures which support the concept of live-long learning (LLL). The learning ability of adults and the various motives which lead to adults learning are the starting point of this thesis. The following analysis on self-directed learning as well as the resulting new attribution of roles and tasks which arise for learners, trainers and institutions in adult education, shall demonstrate first possibilities to contribute to the implementation of the concept of LLL. In addition, the role of information and communication technologies in the framework of LLL will be closer described in context of self-directed learning processes of adults as the opening of new forms of communication and co-operation independent of location and time between learners as well as between learners and tutors gains more importance. Finally the topic of visualisation, validation and recognition of informal and non-formal learning and their contribution to LLL is discussed.

Gliederung des Abstract in **Thema**, **Ausgangspunk**, **Kurzbeschreibung**, **Zielsetzung**.

Projektergebnis Allgemeine Beschreibung, was vom Projektziel umgesetzt wurde, in einigen kurzen Sätzen. Optional Hinweise auf Erweiterungen. Gut machen sich in diesem Kapitel auch Bilder vom Gerät (HW) bzw. Screenshots (SW). Liste aller im Pflichtenheft aufgeführten Anforderungen, die nur teilweise oder gar nicht umgesetzt wurden (mit Begründungen).

Christian Vorhofer
Raphael Ladinig

CCA - COMPETENCE CENTRE
HTL Anichstraße

Erklärung der Eigenständigkeit der Arbeit

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe. Meine Arbeit darf öffentlich zugänglich gemacht werden, wenn kein Sperrvermerk vorliegt.

Ort, Datum Verfasser 1

Ort, Datum Verfasser 1

Inhaltsverzeichnis

1 Einführung in Neuronale Netzwerke

1.1 Künstliche Neuronen

Künstliche Neuronen bilden die Grundbausteine moderner neuronaler Netze und orientieren sich konzeptionell am Funktionsprinzip biologischer Nervenzellen. Ein künstliches Neuron erhält mehrere Eingangswerte $x_1, x_2, \ldots, x_n$, die jeweils mit Gewichten $w_1, w_2, \ldots, w_n$ multipliziert werden. Zusammen mit einem Bias-Term b entsteht die gewichtete Summe

$$z = \sum_{i=1}^{n} w_i x_i + b.$$

Um dem Modell die Fähigkeit zu geben, nichtlineare Zusammenhänge zu lernen, wird auf diese Summe eine Aktivierungsfunktion angewendet. Typische Aktivierungsfunktionen sind die Sigmoid-Funktion, die Tanh-Funktion oder im modernen Deep Learning vor allem die *Rectified Linear Unit* (ReLU). Das resultierende Ausgabe-Signal des Neurons lautet somit

$$y = \sigma(z).$$

Durch die Verschachtelung vieler solcher Neuronen in mehreren Schichten (sogenannten Layers) können sehr komplexe Funktionen modelliert werden. Das „Wissen“ des neuronalen Netzes ist in den Gewichten und Bias-Werten gespeichert, die während des Trainingsprozesses mithilfe von Optimierungsverfahren wie dem Gradientenabstieg angepasst werden.

Die Fähigkeit eines einzelnen Neurons, eine lineare Entscheidungsgrenze zu modellieren, wurde bereits früh durch das Perzeptron-Modell demonstriert. Erst durch die Kombination vieler Neuronen in tieferen Netzen wurde es möglich, hochkomplexe Muster wie Bildmerkmale oder sprachliche Zusammenhänge effizient zu verarbeiten. Damit stellen künstliche Neuronen die Grundlage aller modernen Deep-Learning-Architekturen dar, aus denen später fortgeschrittene Modelle wie Convolutional Neural Networks (CNNs), Rekurrente Neuronale Netze (RNNs) und Transformer hervorgegangen sind.

Goodfellow et al. (2016)

1.2 Feed-Forward Neural Networks (FNN)

Ein Feed-Forward Neural Network (FNN) ist die einfachste Form eines neuronalen Netzes und bildet die Grundlage vieler moderner Deep-Learning-Modelle. Der Name beschreibt bereits die wichtigste Eigenschaft: Informationen fließen nur in eine Richtung, nämlich vom Eingang (*Input Layer*) über eine oder mehrere verdeckte Schichten (*Hidden Layers*) zum Ausgang (*Output Layer*). Es gibt keine Rückkopplungen oder Schleifen.

Ein FNN besteht aus vielen künstlichen Neuronen, die miteinander verbunden sind. Jedes Neuron berechnet aus seinen Eingaben eine gewichtete Summe und wendet anschließend eine Aktivierungsfunktion wie ReLU, Sigmoid oder Tanh an. Dadurch kann das Netzwerk auch komplexe, nichtlineare Zusammenhänge erkennen.

Das Netzwerk „lernt“, indem es seine Gewichte anpasst. Dies geschieht über ein Verfahren namens *Backpropagation*. Dabei wird gemessen, wie weit die Vorhersage des Netzes vom tatsächlichen Ergebnis entfernt ist. Dieser Fehler wird dann genutzt, um die Gewichte so zu verändern, dass das Modell in zukünftigen Durchläufen bessere Ergebnisse liefert.

Obwohl FNNs im Vergleich zu neueren Architekturen wie CNNs, RNNs oder Transformern relativ einfach aufgebaut sind, bilden sie das Fundament des Deep Learning. Viele moderne Modelle lassen sich als Weiterentwicklungen dieses grundlegenden Prinzips verstehen.

Christian Vorhofer
Raphael Ladinig

Goodfellow et al. (2016)

1.3 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) sind eine spezielle Art von neuronalen Netzen, die besonders gut für die Verarbeitung von Bildern geeignet sind. Im Gegensatz zu einfachen Feed-Forward-Netzen berücksichtigen CNNs die räumliche Struktur von Daten. Das bedeutet, dass sie Muster wie Kanten, Formen oder Texturen erkennen können – unabhängig davon, wo sie im Bild auftreten.

Der wichtigste Baustein eines CNN ist die *Convolutional Layer*. In dieser Schicht wandern kleine Filter (auch *Kerne* oder *Kernels* genannt) über das Bild und berechnen lokale Merkmale. Ein einzelner Filter kann zum Beispiel lernen, horizontale Kanten zu erkennen, während ein anderer runde Formen erkennt. Mehrere solcher Filter erzeugen sogenannte Feature Maps, die unterschiedliche Aspekte des Bildes hervorheben.

Zusätzlich zu den Faltungsschichten verwenden CNNs oft *Pooling-Schichten*. Diese verkleinern die Bilddarstellung, indem sie z. B. aus einem 2x2-Bereich nur den größten Wert übernehmen (*Max-Pooling*). Dadurch wird das Modell robuster gegenüber kleinen Verschiebungen im Bild und reduziert gleichzeitig die Anzahl der Parameter.

Am Ende eines CNNs befinden sich meist ein oder mehrere vollständig verbundene Schichten (*Fully Connected Layers*), die auf Basis der erkannten Merkmale eine Entscheidung treffen, zum Beispiel welche Klasse ein Bild hat.

CNNs haben die Bildverarbeitung revolutioniert und sind nach wie vor ein zentraler Bestandteil moderner Computer-Vision-Systeme. Für Textverarbeitung werden sie allerdings seltener eingesetzt, da Sprache eher eine Sequenz als eine zweidimensionale Struktur ist.

Lecun et al. (1998)

Christian Vorhofer
Raphael Ladinig

1.4 Rekurrente Neuronale Netze (RNN, LSTM)

Rekurrente Neuronale Netze (RNNs) wurden entwickelt, um Daten zu verarbeiten, die aus Sequenzen bestehen – zum Beispiel Texte, Sprache, Musik oder Zeitreihen. Im Gegensatz zu Feed-Forward- oder Convolutional-Netzen besitzen RNNs eine Rückkopplung: Ein Teil der Ausgabe eines Zeitschrittes wird als Eingabe in den nächsten Schritt zurückgeführt. Dadurch können RNNs Informationen aus früheren Zeitpunkten speichern und haben eine Art „Gedächtnis“.

Ein einfaches RNN verarbeitet zu jedem Zeitpunkt einen Eingangswert und kombiniert diesen mit dem vorherigen Zustand des Netzwerks. Das Verfahren funktioniert gut bei kurzen Sequenzen, hat jedoch Schwierigkeiten bei langen Abhängigkeiten. Das liegt am sogenannten *Vanishing Gradient Problem*, bei dem wichtige Informationen beim Training schnell verloren gehen.

Um diese Schwächen auszugleichen, wurden **LSTMs** (Long Short-Term Memory) entwickelt. LSTMs besitzen spezielle Schaltelemente, sogenannte *Gates*. Diese Gates entscheiden, welche Informationen gespeichert, weitergegeben oder gelöscht werden. Dadurch können LSTMs deutlich länger relevante Zusammenhänge behalten und sind stabiler beim Training als einfache RNNs.

LSTMs wurden viele Jahre erfolgreich im Bereich der Sprachverarbeitung eingesetzt, zum Beispiel für maschinelle Übersetzung oder Textklassifikation. Heute spielen sie jedoch eine deutlich kleinere Rolle, da Transformer-Modelle effizienter trainierbar sind und besser mit langen Texten umgehen können.

Hochreiter & Schmidhuber (1997)

1.5 Die Transformer-Architektur

Transformer-Modelle wurden im Jahr 2017 mit dem Paper Attention Is All You Need eingeführt und haben die Sprachverarbeitung grundlegend

verändert. Im Gegensatz zu RNNs und LSTMs verarbeiten Transformer die Eingabe nicht schrittweise, sondern betrachten alle Wörter eines Satzes gleichzeitig. Dadurch können sie wesentlich besser mit langen Texten umgehen und lassen sich effizient auf modernen GPUs parallelisieren.

Das zentrale Konzept eines Transformers ist die sogenannte Self-Attention. Diese Technik ermöglicht es dem Modell, zu bestimmen, welche Wörter in einem Satz für die Bedeutung eines anderen Wortes wichtig sind. Ein Beispiel: Im Satz „Der Hund jagt die Katze, weil sie schnell ist“ muss das Modell erkennen, dass sich „sie“ auf „die Katze“ bezieht. Self-Attention macht genau das möglich, indem jedes Wort auf alle anderen Wörter „aufmerksam“ werden kann.

Ein weiterer wichtiger Bestandteil ist die Multi-Head Attention. Hierbei nutzt das Modell mehrere Attention-Mechanismen gleichzeitig, die jeweils unterschiedliche Arten von Beziehungen lernen können – etwa grammatische Strukturen, thematische Zusammenhänge oder Referenzen im Text. Diese Informationen werden anschließend kombiniert, um ein besonders aussagekräftiges Gesamtverständnis zu erzeugen.

Da Transformer keine natürliche Reihenfolge wie RNNs haben, benötigen sie Positional Encodings, die beschreiben, an welcher Stelle ein Wort im Satz steht. Diese Positionsinformationen werden zu den Eingabedaten addiert, sodass das Modell die Struktur des Satzes versteht.

Ein klassischer Transformer besteht aus zwei Teilen: einem Encoder, der den Text verarbeitet und in eine nützliche Repräsentation (Embedding) umwandelt, und einem Decoder, der beispielsweise Text generieren kann. Bei modernen Sprachmodellen wie GPT wird meist nur der Decoder verwendet, während für Suchsysteme wie Tapyre ausschließlich der Encoder relevant ist.

Transformer haben sich aufgrund ihrer Genauigkeit, Skalierbarkeit und Effizienz als Standard für alle modernen NLP-Systeme durchgesetzt. Sie bilden die Grundlage großer Sprachmodelle (LLMs) und leistungsstarker Embedding-Modelle, wie sie auch in diesem Projekt verwendet werden.

Ashish Vaswani (2017)

Christian Vorhofer
Raphael Ladinig

1.6 Bedeutung von Transformern für LLMs und Embeddings

Transformer-Modelle spielen heute eine zentrale Rolle in fast allen Bereichen der Sprachverarbeitung. Sie bilden die Grundlage großer Sprachmodelle (*Large Language Models, LLMs*) wie GPT, LLaMA oder PaLM und sind außerdem der Standard für die Erzeugung hochwertiger Text-Embeddings. Der Grund dafür liegt in den besonderen Eigenschaften der Transformer-Architektur.

Durch den Einsatz von Self-Attention können Transformer Zusammenhänge zwischen weit entfernten Wörtern erkennen, was besonders wichtig für längere Texte, komplexe Satzstrukturen oder wissenschaftliche Dokumente ist. Während frühere Modelle wie RNNs oder LSTMs oft Schwierigkeiten hatten, Informationen über viele Wörter hinweg zu behalten, können Transformer den gesamten Kontext gleichzeitig berücksichtigen. Dies führt zu deutlich besseren Ergebnissen bei allen Aufgaben, die ein tiefes Textverständnis erfordern.

Für Embedding-Modelle – also Modelle, die Texte in numerische Vektoren umwandeln – bieten Transformer einen weiteren entscheidenden Vorteil: Sie erzeugen Repräsentationen, die nicht nur die Bedeutung einzelner Wörter, sondern die gesamte semantische Struktur eines Satzes oder Dokuments erfassen. Deshalb eignen sich Transformer-Encoder besonders gut für Suchsysteme, Klassifikationsaufgaben oder Recommendation-Systeme.

Moderne Embedding-Modelle wie *SPECTER2*, *Sentence-BERT* oder *E5* basieren alle auf Transformer-Encodern. Sie ermöglichen es, dass ähnliche Texte in einem Vektorraum nahe beieinander liegen, während unterschiedliche Inhalte klar voneinander getrennt sind. Diese Eigenschaft ist essenziell für semantische Suche, wie sie auch in diesem Projekt eingesetzt wird.

Zusammenfassend lässt sich sagen, dass Transformer die Grundlage moderner Sprachverarbeitung bilden. Ohne Transformer wären sowohl leistungsfähige LLMs als auch präzise Embedding-Modelle nicht möglich – und Systeme wie Tapyre Paper Search könnten in dieser Form nicht existieren.

Christian Vorhofer
Raphael Ladinig

Ashish Vaswani (2017) Brown et al. (2020) Touvron et al. (2023) Nandakumar et al. (2023)

2 Einführung in Natural Language Processing (NLP)

Natural Language Processing (NLP) ist ein zentraler Bereich der Künstlichen Intelligenz, der sich mit der automatischen Verarbeitung menschlicher Sprache beschäftigt. Ziel ist es, Texte so zu analysieren und zu interpretieren, dass Computer sprachbasierte Aufgaben ausführen können – etwa Suchanfragen beantworten, Texte zusammenfassen oder Dokumente klassifizieren. Moderne Systeme wie Suchmaschinen, Chatbots oder Sprachassistenten bauen maßgeblich auf Methoden des NLP auf Jurafsky & Martin (2023).

Während frühe Ansätze vor allem statistische Modelle nutzten, basiert das heutige NLP überwiegend auf tiefen neuronalen Netzen. Eine entscheidende Rolle spielt dabei die Frage, wie Bedeutungen mathematisch repräsentiert werden können. Diese Repräsentationen werden als **Embeddings** bezeichnet.

2.1 Klassische NLP-Ansätze

Vor dem Aufkommen neuronaler Modelle wurden Texte meist mithilfe statistischer Verfahren dargestellt. Typische Beispiele sind Bag-of-Words, TF–IDF und N-Gramme. Diese Methoden berücksichtigen jedoch weder semantische Beziehungen noch Kontextinformationen. So wird nicht erkannt, dass „Auto“ und „Fahrzeug“ ähnliche Bedeutungen haben oder dass „Bank“ sowohl ein Sitzmöbel als auch ein Finanzinstitut bezeichnen kann Jurafsky & Martin (2023).

Für einfache Klassifikationsaufgaben sind solche Modelle oft ausreichend, stoßen jedoch bei komplexeren Anwendungen – etwa semantischer Suche oder Übersetzung – schnell an ihre Grenzen.

2.2 Einführung in Embeddings

Da Computer ausschließlich mit numerischen Daten arbeiten, müssen Texte in Zahlen überführt werden. Embeddings lösen dieses Problem, indem sie Wörter, Sätze oder ganze Dokumente als Vektoren in einem kontinuierlichen Raum darstellen. Dabei gilt:

- Ähnliche Bedeutungen sollen ähnliche Vektoren besitzen.
- Unterschiedliche Bedeutungen sollen weit voneinander entfernt liegen.
- Kontextinformationen sollen möglichst berücksichtigt werden.

Embeddings bilden die Grundlage vieler moderner NLP-Systeme und ermöglichen semantische Ähnlichkeitsanalysen sowie inhaltliche Textvergleiche.

2.3 Word Embeddings: Word2Vec und GloVe

Einen bedeutenden Fortschritt stellten Word Embeddings wie **Word2Vec** dar. Diese Modelle ordnen jedem Wort einen festen Vektor zu und basieren auf der Idee, dass Wörter, die in ähnlichen Kontexten auftreten, ähnliche Repräsentationen erhalten. Dadurch entstehen semantische Strukturen wie:

$$\text{Knig} - \text{Mann} + \text{Frau} \approx \text{Knigin}$$

Word2Vec Mikolov et al. (2013) und ähnliche Ansätze erfassen grundlegende semantische Beziehungen, ignorieren jedoch die Mehrdeutigkeit von Wörtern: Das Wort „Bank“ hat stets denselben Vektor, unabhängig vom Kontext.

2.4 Kontextualisierte Embeddings

Mit tiefen neuronalen Netzen entstanden Modelle, die Wortbedeutungen kontextabhängig repräsentieren. Ein Beispiel dafür ist **ELMo**, das für jedes Wort unterschiedliche Vektoren erzeugt – abhängig vom Satz, in dem es vorkommt. Diese Ansätze bilden eine Übergangsphase zwischen klassischen Embeddings und modernen Transformer-Modellen.

2.5 Embeddings mit der Transformer-Architektur

Mit der Einführung der Transformer-Architektur wurden neue Maßstäbe gesetzt. Transformer-Encoder wie BERT Devlin et al. (2018) oder wissenschaftsspezifische Modelle wie SPECTER Cohan et al. (2020) erzeugen hochqualitative, kontextualisierte Embeddings, indem sie den gesamten Satz oder sogar das gesamte Dokument berücksichtigen.

Dies führt zu:

- kontextabhängigen Wortvektoren,
- Satz- und Dokumentrepräsentationen als einzelne Vektoren,
- präziser semantischer Modellierung,
- robuster Erfassung langer und komplexer Zusammenhänge.

Transformer-Embeddings sind daher besonders gut geeignet, um wissenschaftliche Texte mit ihren komplexen Begrifflichkeiten und Strukturmerkmalen zu verarbeiten.

2.6 Relevanz für Tapyre Paper Search

Im Projekt *Tapyre Paper Search* dienen Embeddings dazu, wissenschaftliche Artikel in einem Vektorraum abzubilden. Dokumente mit thematischen Ähnlichkeiten liegen darin räumlich nahe beieinander, was eine präzise semantische Suche ermöglicht. Spezialisierte Modelle wie SPECTER2, die auf

wissenschaftlichen Publikationen trainiert wurden, verbessern die Erkennung fachlicher Zusammenhänge nochmals deutlich.

3 Einführung in Agentic AI

Agentic AI beschreibt ein neues Paradigma der künstlichen Intelligenz, bei dem Modelle nicht nur Antworten generieren, sondern eigenständig Handlungen planen, Tools verwenden, Entscheidungen treffen und komplexe Aufgaben in mehreren Schritten ausführen. Ein *Agent* ist dabei ein KI-System, das aktiv Ziele verfolgt, Informationen beschafft, Aktionen ausführt und basierend auf den Ergebnissen weitere Schritte plant. Während klassische Sprachmodelle rein reaktiv arbeiten, agiert Agentic AI proaktiv und interaktiv.

Dieses Konzept ist besonders relevant für Systeme wie Tapyre Paper Search, da Agenten autonom Texte analysieren, Datenbanken abfragen, externe Tools aufrufen oder Informationen aus verschiedenen Quellen kombinieren können. Moderne KI-Anwendungen setzen zunehmend auf agentische Systeme, um flexiblere und robustere Abläufe zu ermöglichen.

3.1 ReAct: Reasoning + Acting

Ein grundlegender Ansatz innerhalb von Agentic AI ist das **ReAct**-Framework (Reasoning + Acting). Dabei führt ein Agent nicht nur interne Überlegungen (*Reasoning*) aus, sondern trifft auch explizite Entscheidungen und führt konkrete Aktionen (*Acting*) aus. ReAct wurde von Yao et al. vorgestellt Yao et al. (2022).

Ein typischer ReAct-Agent arbeitet in zyklischer Struktur:

1. Der Agent überlegt (*Thought*), wie er vorgehen soll.
2. Er führt eine Aktion aus, z. B. eine API-Anfrage.
3. Er erhält eine Beobachtung (*Observation*).

4. Basierend darauf plant er den nächsten Schritt.

Diese Schleife ermöglicht es dem Agenten, komplexe Aufgaben flexibel in Teilschritte zu zerlegen und dynamisch auf neue Informationen zu reagieren.

3.2 Tool Usage

Ein wesentliches Merkmal agentischer Systeme ist die Fähigkeit, externe Werkzeuge (*Tools*) einzusetzen. Beispiele hierfür sind:

- Datenbanken (z. B. Qdrant, MySQL),
- Web-APIs (arXiv, Semantic Scholar),
- Dateisysteme,
- Suchfunktionen,
- Python-Skripte.

Der Agent wählt das passende Tool aus, übergibt Parameter, interpretiert die Ergebnisse und nutzt diese, um weitere Entscheidungen zu treffen. Dadurch wird das Sprachmodell zu einer Art Steuerzentrale, die verschiedene Systeme koordinieren kann.

3.3 Model Context Protocol (MCP)

Das **Model Context Protocol (MCP)** ist ein offener Standard, der definiert, wie KI-Modelle sicher und zuverlässig mit Tools und Datenquellen interagieren können. MCP legt fest:

- wie Tools strukturiert beschrieben werden,
- wie Kontext an Modelle übergeben wird,
- wie Modelle Aktionen anfordern,
- und wie Ergebnisse standardisiert zurückgegeben werden.

Der offene Standard von OpenAI OpenAI (2024) ermöglicht es, Agenten flexibel in Software-Systeme einzubetten und komplexe Pipelines ohne individuelle Integrationslogik anzubinden.

3.4 Agent-to-Agent Kommunikation

In größeren Systemen arbeiten oft mehrere Agenten gemeinsam an einer Aufgabe. Diese Agent-to-Agent-Kommunikation kann genutzt werden, um Aufgaben zu verteilen, Wissen auszutauschen oder verschiedene Strategien zu evaluieren. Beispiele hierfür sind:

- ein Analyse-Agent extrahiert Daten,
- ein Recherche-Agent sucht passende Quellen,
- ein Planungs-Agent entscheidet über das weitere Vorgehen,
- ein Evaluations-Agent überprüft Ergebnisse.

Durch die Spezialisierung der Rollen entsteht eine höhere Robustheit und Skalierbarkeit.

3.5 RAG: Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) verbindet Sprachmodelle mit externem Wissen. Statt Antworten frei zu generieren, sucht ein RAG-System zuerst nach relevanten Dokumenten und erzeugt anschließend eine fundierte Antwort auf Basis dieser Inhalte. Der Ansatz wurde von Lewis et al. eingeführt Lewis et al. (2020).

Ein RAG-Agent arbeitet typischerweise wie folgt:

1. **Retrieval:** Suche nach relevanten Dokumenten, z. B. über Vektorsuche in Qdrant.
2. **Generation:** Erzeugung einer Antwort mithilfe des Sprachmodells, unter Nutzung der gefundenen Informationen.

Für Tapyre Paper Search ist dieser Ansatz essenziell, da wissenschaftliche Dokumente zuerst semantisch abgerufen und dann weiter analysiert oder zusammengefasst werden.

3.6 Multi-Agent Systems

Multi-Agent-Systems (MAS) bestehen aus mehreren spezialisierten Agenten, die parallel oder kooperativ arbeiten. Vorteile solcher Systeme umfassen:

- höhere Robustheit durch Rollenverteilung,
- bessere Skalierbarkeit,
- parallele Problemlösung,
- Spezialisierung auf Teilprobleme.

MAS werden zunehmend in Forschung, Retrieval-Systemen und komplexen KI-Anwendungen eingesetzt.

4 Grundkonzepte der verwendeten Technologien

Für die Umsetzung von Tapyre Paper Search werden mehrere moderne Software- und Infrastrukturtechnologien eingesetzt, die zusammen eine performante und erweiterbare Architektur bilden. Dieses Kapitel erläutert die wichtigsten technischen Grundlagen und erklärt insbesondere, wie Daten gespeichert, verarbeitet und über REST-Schnittstellen ausgetauscht werden.

4.1 Docker und Containerisierung

Docker ist eine Plattform zur Containerisierung von Anwendungen Docker Inc. (2025). Im Gegensatz zu klassischen virtuellen Maschinen teilt sich ein Container den Kernel des Host-Betriebssystems. Trotzdem ist jede Anwendung logisch isoliert. Diese Isolation wird durch mehrere Linux-Technologien erreicht, insbesondere Namespaces, Control Groups und Union-Filesystems Docker Inc. (2025), Quirós (2024):

- **Namespaces**: isolieren Prozesse, Netzwerke, Benutzer und Dateisysteme.
- **Control Groups (cgroups)**: begrenzen CPU-, RAM- und I/O-Ressourcen.
- **Union-Filesystems (z. B. OverlayFS)**: ermöglichen effiziente Layer-basierte Images.

Docker ermöglicht reproduzierbare Umgebungen, schnelle Deployments und konsistente Konfigurationen. Für Projekte wie Tapyre bedeutet dies, dass Komponenten wie Qdrant, MySQL oder Python-Anwendungen unabhängig voneinander, aber dennoch einheitlich ausgeführt werden können.

4.2 MySQL als relationale Datenbank

MySQL ist ein relationales Datenbanksystem, das Daten strukturiert in Tabellen speichert. Das Datenmodell folgt einem relationalen Schema, bei dem Entitäten über Primär- und Fremdschlüssel miteinander verbunden sind. Für die interne Datenorganisation nutzt MySQL (InnoDB) hauptsächlich B+-Bäume, die als Clustered und Secondary Indexes realisiert sind Oracle Corporation (2024), Percona (2024):

- **Clustered Index**: InnoDB speichert Daten entlang des Primärschlüssels. Die Tabelle ist selbst ein B+-Baum.
- **Secondary Indexes**: weitere B+-Bäume, die Zeiger auf die Primärzeilen enthalten.
- **Effiziente Suche**: Durch die logarithmische Höhe der B+-Bäume können Suchanfragen schnell ausgeführt werden.

MySQL bietet außerdem ACID-Transaktionen, die Datenkonsistenz garantieren und in der klassischen Datenbankliteratur ausführlich beschrieben werden Gray & Reuter (1992):

- **Atomicity**: Eine Transaktion wird entweder vollständig ausgeführt oder verworfen.
- **Consistency**: Alle Daten erfüllen definierte Integritätsregeln.
- **Isolation**: Gleichzeitige Transaktionen beeinflussen sich nicht.
- **Durability**: Bestätigte Änderungen bleiben dauerhaft gespeichert.

In Tapyre wird MySQL zur Speicherung von Nutzerdaten, Metadaten, Logeinträgen und administrativen Strukturen eingesetzt.

4.3 Qdrant und Approximate Nearest Neighbor Search

Qdrant ist eine spezialisierte Datenbank zur Speicherung und Suche von Vektorrepräsentationen (Embeddings) und wird explizit als Vektor-Datenbank

für semantische Suche entwickelt Qdrant Technologies (2024*d*). Im Gegensatz zu relationalen Datenbanken speichert Qdrant keine Texte, sondern numerische Vektoren, die die Bedeutung eines Dokuments darstellen.

Speicherstruktur

Eine Qdrant-Collection besteht aus Qdrant Technologies (2024*b*):

- **Vektoren** (meist 768 oder 1024 Dimensionen),
- **Payload-Daten** wie Titel, DOI, Autoren oder Jahr,
- **Segmenten** zur internen Aufteilung der Daten.

Diese Struktur ist optimiert für sequentielles Lesen und schnelle Ähnlichkeitsabfragen Qdrant Technologies (2024*e*).

Approximate Nearest Neighbor (ANN)

Da ein exakter Vergleich aller Vektoren bei großen Datenmengen ineffizient wäre, verwendet Qdrant Approximate Nearest Neighbor (ANN)-Algorithmen. Der wichtigste davon ist der **HNSW-Index (Hierarchical Navigable Small World)**, ein Graph-basierter ANN-Algorithmus Malkov & Yashunin (2020), Qdrant Technologies (2024*c*,*a*):

- eine mehrschichtige Graphstruktur,
- wenige Knoten auf höheren Ebenen (grobe Orientierung),
- viele Knoten auf unteren Ebenen (feine Suche),
- Navigation vom groben zum feinen Bereich,
- ermöglicht extrem schnelle Annäherung an das richtige Suchergebnis.

HNSW kombiniert hohe Geschwindigkeit mit hoher Genauigkeit und eignet sich besonders gut für semantische Suchsysteme Malkov & Yashunin (2020), Qdrant Technologies (2024*e*).

Christian Vorhofer
Raphael Ladinig

Ähnlichkeitsmaße

Qdrant unterstützt verschiedene Metriken Qdrant Technologies (2024*e*):

- Kosinusähnlichkeit (Standard für NLP-Modelle),
- Euklidische Distanz,
- Skalares Produkt (Dot Product).

In Tapyre kommt hauptsächlich die Kosinusähnlichkeit zum Einsatz, da sie die semantische Nähe zwischen Dokumenten besonders gut abbildet.

4.4 Flask und REST-APIs

Flask ist ein leichtgewichtiges Webframework für Python und dient in Tapyre zur Bereitstellung von REST-APIs Ronacher & Contributors (2024). Eine REST-API (*Representational State Transfer*) basiert auf einem Architekturstil für verteilte Hypermedia-Systeme, der von Fielding in seiner Dissertation beschrieben wurde Fielding (2000). Sie ermöglicht die Kommunikation zwischen Anwendungsteilen über standardisierte HTTP-Methoden:

- **GET**: Anfrage von Daten
- **POST**: Erstellen neuer Daten
- **PUT/PATCH**: Aktualisieren von Daten
- **DELETE**: Löschen von Daten

REST-APIs verwenden häufig JSON als Datenaustauschformat und sind zustandslos: Jeder Request enthält alle notwendigen Informationen, um verarbeitet zu werden Fielding (2000).

Ein typischer Beispiel-Request:

```
POST /embed
{
    "text": "Deep learning improves scientific search."
}
```

Die API ruft daraufhin den Embedding-Prozess auf, speichert das Ergebnis oder gibt es zurück. Dieser Ansatz ermöglicht modulare, wartbare und gut testbare Kommunikationsstrukturen.

4.5 PyTorch und GPU-Beschleunigung

PyTorch ist ein Framework für Deep Learning und wird verwendet, um Embeddings zu berechnen Paszke et al. (2019). Da Transformer-Modelle wie SPECTER2 hunderte Millionen Parameter besitzen, werden GPUs genutzt, um Berechnungen massiv zu beschleunigen. CUDA ermöglicht hierbei die Ausführung linearer Algebraoperationen direkt auf der Grafikkarte NVIDIA Corporation (2025).

Relevante Vorteile von PyTorch Paszke et al. (2019):

- dynamische Rechengraphen,
- große Modellbibliothek,
- nahtlose GPU-Unterstützung,
- Integration in moderne NLP-Pipelines.

4.6 Zusammenfassung

Docker stellt reproduzierbare Umgebungen bereit Docker Inc. (2025), MySQL speichert strukturierte Daten effizient über B+-Bäume und Clustered Indexes Oracle Corporation (2024), Qdrant ermöglicht schnelle semantische Vektorsuche mithilfe von HNSW und ANN Qdrant Technologies (2024*d*), Malkov & Yashunin (2020), Flask dient als Kommunikationsschicht über REST-APIs Ronacher & Contributors (2024), Fielding (2000), und PyTorch führt rechenintensive Transformer-Modelle auf GPUs mithilfe von CUDA aus Paszke et al. (2019), NVIDIA Corporation (2025). Diese Technologien bilden zusammen die Basis für ein modernes, flexibles und leistungsfähiges Informationssystem wie Tapyre Paper Search.

5 Entwicklung von Tapyre als Agentic-AI-System

In diesem Kapitel wird die Entwicklung von Tapyre als *Agentic AI*-System beschrieben. Im Gegensatz zu klassischen, rein reaktiven Sprachmodellen (Input → Output) arbeitet Tapyre mit einem Agenten, der eigenständig Tools aufrufen kann, in einem ReAct-Loop (Reasoning + Acting) entscheidet und seine Funktionalität über Plugins dynamisch erweitert Yao et al. (2022), Schick et al. (2023), Wang et al. (2024). Die Kopplung zwischen Kernsystem, LLM und Plugins ist dabei bewusst lose gehalten, um das System leicht erweiterbar und wartbar zu machen Gamma et al. (1994).

Im Folgenden werden die wichtigsten Bausteine erläutert:

- abstrakte LLM-Schnittstelle und konkrete Implementation für Ollama,
- der Agent auf Basis von LangChain und dem ReAct-Paradigma,
- das Plugin-Konzept und die Abbildung auf LangChain-Tools,
- dynamisches Laden der Plugins zur Laufzeit,
- ein konkretes Beispiel-Plugin (`AppPlugin`) zum Starten von Desktop-Anwendungen,
- lose Kopplung und Erweiterbarkeit.

5.1 Abstraktion der LLM-Schnittstelle

Um das System unabhängig von einem konkreten Sprachmodell oder Anbieter zu halten, wird eine abstrakte LLM-Schnittstelle definiert. Jeder LLM-Typ (z. B. Ollama, OpenAI, etc.) muss nur diese Schnittstelle implementieren. Dadurch kann das Modell später einfach ausgetauscht werden, ohne dass

der Agent oder die Plugins angepasst werden müssen. Dieses Prinzip folgt etablierten Architekturmustern wie Interface- und Factory-Abstraktionen Gamma et al. (1994).

```python
from abc import ABC, abstractmethod
from langchain_core.language_models.chat_models import BaseChatModel
class LLM(ABC):
    @abstractmethod
    def getLLM(self) -> BaseChatModel:
        pass
```

Listing 5.1: Abstrakte LLM-Schnittstelle

Eine konkrete Implementierung für Ollama sieht dann wie folgt aus:

```python
from abstractions.llm import LLM
from langchain_core.language_models.chat_models import BaseChatModel
from langchain_community.chat_models import ChatOllama

class OllamaLLM(LLM):
    def __init__(self, model: str, host: str, temperature: float, max_tokens: int):
        self.llm = ChatOllama(
            model=model,
            base_url=host,
            temperature=temperature,
            max_tokens=max_tokens
        )
    def getLLM(self) -> BaseChatModel:
        return self.llm
```

Listing 5.2: OllamaLLM als konkrete Implementierung

Der Rest des Systems kennt nur das Interface `LLM` und arbeitet mit `BaseChatModel`-Instanzen. Welches konkrete Modell dahinter steckt, ist für den Agenten und die Plugins transparent.

5.2 Der Agent und der ReAct-Loop

Der eigentliche Agent ist als Abstraktion definiert und besitzt lediglich eine Methode `ask`, die eine Anfrage entgegennimmt und eine Antwort zurückliefert:

```python
from abc import ABC, abstractmethod
class Agent(ABC):
    @abstractmethod
    def ask(self, prompt: str) -> str:
```

```
        pass
```

Listing 5.3: Abstrakte Agent-Schnittstelle

Die konkrete Implementation `PluginAgent` verwendet LangChain und das Agentenmodell `ZERO_SHOT_REACT_DESCRIPTION`. Dieses Agentenmodell setzt das ReAct-Prinzip um: Das Modell plant in Gedanken (*Reasoning*), ruft bei Bedarf Tools auf (*Acting*) und verarbeitet die Ergebnisse iterativ weiter Yao et al. (2022), LangChain (2023).

```python
# simple_agent.py
from langchain.agents import initialize_agent, AgentType
from langchain.prompts import ChatPromptTemplate, SystemMessagePromptTemplate, HumanMessagePromptTemplat
from abstractions.agent import Agent
from abstractions.llm import LLM

class PluginAgent(Agent):
    def __init__(self, tools: list, llm: LLM, system_prompt: str, verbose: bool):

        self.llm = llm.getLLM()

        self.prompt = ChatPromptTemplate.from_messages([
            SystemMessagePromptTemplate.from_template(system_prompt),
            HumanMessagePromptTemplate.from_template("{input}")
        ])

        self.agent = initialize_agent(
            tools=tools,
            llm=self.llm,
            agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
            verbose=verbose,
            handle_parsing_errors=True,
            max_iterations=2,
            early_stopping_method="generate"
        )

    def ask(self, prompt: str) -> str:
        return self.agent.run(prompt)
```

Listing 5.4: PluginAgent mit ReAct-Agententyp

Wichtige Aspekte:

- **ReAct-Loop:** `AgentType.ZERO_SHOT_REACT_DESCRIPTION` sorgt dafür, dass das LLM selbst entscheidet, wann ein Tool benutzt werden soll. Es erzeugt intern eine Folge aus »Thought«, »Action« und »Observation«.
- **Tool-Auswahl:** Die Liste `tools` wird später aus den geladenen Plugins erzeugt. Das LLM sieht nur die Tool-Beschreibungen und entscheidet basierend darauf, welches Tool geeignet ist.

- **max_iterations:** Begrenzung des ReAct-Loops auf zwei Tool-Aufrufe, um Endlosschleifen zu vermeiden Schick et al. (2023).

5.3 Plugins als Tools: Loose Coupling durch Interfaces

Plugins stellen die eigentliche Funktionalität des Systems dar (z. B. das Starten von Programmen, Suchen in Datenbanken, etc.). Sie sind über eine abstrakte Basisklasse definiert und können damit beliebig erweitert werden, ohne dass der Kern des Systems angepasst werden muss. Die Architektur folgt klassischen Prinzipien der losen Kopplung Gamma et al. (1994).

```python
from abc import ABC, abstractmethod
from langchain_core.tools import Tool as LCTool

class Plugin(ABC):
    prefix: str
    name: str
    prompt: str

    @property
    def Promt(self) -> str:  # noqa: N802
        return self.prompt

    @abstractmethod
    def run(self, text: str) -> str:
        pass

    def full_name(self) -> str:
        return self.name

    def to_langchain(self, *, return_direct: bool = False):
        tool = LCTool.from_function(
            name=self.full_name(),
            func=self.run,
            description=self.prompt,
        )

        tool.return_direct = return_direct
        return tool

    def __call__(self, text: str) -> str:
        return self.run(text)
```

Listing 5.5: Abstrakte Plugin-Basisklasse

Wesentliche Punkte:

- **Interface-basiert:** Plugins müssen nur `run()` implementieren.
- **Loose Coupling:** Der Agent kennt nur die LangChain-Tools, nicht die konkrete Plugin-Klasse.
- **Tool-Integration:** `LCTool.from_function` macht aus `run()` ein Tool, das im ReAct-Loop genutzt werden kann LangChain (2023).

5.4 Dynamisches Laden der Plugins

Damit neue Plugins hinzugefügt werden können, ohne das Hauptprogramm zu ändern, werden sie dynamisch zur Laufzeit geladen. Dieses Prinzip folgt gängigen Entwurfsmustern für modulare und erweiterbare Systeme Gamma et al. (1994).

```
1  from __future__ import annotations
2
3  import inspect
4  from importlib.util import module_from_spec, spec_from_file_location
5  from pathlib import Path
6  from typing import List
7  from abstractions.plugin import Plugin
8
9
10 class PluginLoader:
11     def __init__(self, plugins_dir: str | Path | None = None) -> None:
12         if plugins_dir is None:
13             plugins_dir = Path(__file__).parent / ".." / "plugins"
14         self.plugins_dir = Path(plugins_dir).resolve()
15
16     def load(self) -> List[Plugin]:
17         if not self.plugins_dir.exists():
18             print(f"[PluginLoader] Folder not Found: {self.plugins_dir}")
19             return []
20
21         plugins: List[Plugin] = []
22         for file in sorted(self.plugins_dir.glob("*.py")):
23             if file.name.startswith("_"):
24                 continue
25
26             mod = self._import_module(file)
27             if not mod:
28                 continue
29
30             for _, cls in inspect.getmembers(mod, inspect.isclass):
31                 if cls.__module__ != mod.__name__:
32                     continue
33                 if not issubclass(cls, Plugin) or cls is Plugin:
34                     continue
```

```
35                 if inspect.isabstract(cls):
36                     continue
37                 try:
38                     instance = cls()
39                     plugins.append(instance)
40                 except TypeError as e:
41                     print(f"[PluginLoader] Couldn't load a instance of {cls.__name__}: {e}"
42
43         return plugins
44
45     def _import_module(self, file: Path):
46         module_name = f"plugins_{file.stem}_{abs(hash(str(file)))}"
47         spec = spec_from_file_location(module_name, file)
48         if not spec or not spec.loader:
49             print(f"[PluginLoader] Spec failed for: {file}")
50             return None
51         mod = module_from_spec(spec)
52         try:
53             spec.loader.exec_module(mod)  # type: ignore[attr-defined]
54             return mod
55         except Exception as e:
56             print(f"[PluginLoader] Error while importing  {file}: {e}")
57             return None
```

Listing 5.6: Dynamischer PluginLoader

Vorteile dieses Ansatzes:

- **Plugin-basiertes Design:** Funktionen werden modular ergänzt.
- **Keine Codeänderung im Kern:** Neue Funktionalitäten sind sofort nutzbar.
- **Reflection & Introspection:** Automatische Erkennung von Plugin-Klassen.

5.5 Beispiel: AppPlugin zur Steuerung lokaler Anwendungen

Ein konkretes Plugin ist das `AppPlugin`, das installierte Desktop-Anwendungen aus `.desktop`-Dateien ausliest und auf Kommando starten kann. Es implementiert die Plugin-Schnittstelle und stellt damit ein Werkzeug bereit, das der Agent im ReAct-Loop selbstständig nutzen kann Yao et al. (2022), LangChain (2023).

```python
from abstractions.plugin import Plugin
import subprocess
import os
import configparser
import shlex


class AppPlugin(Plugin):
    def __init__(self):
        self.prefix = "launch"
        self.name = "AppLauncher"
        self.prompt = (
            "Opens a specified application, you got the following apps to choose"
            + self.get_all_apps()
            + "."
        )

        self.app_map = self._build_app_map()

    def _build_app_map(self):
        desktop_dirs = set()

        user_dir = os.path.expanduser("~/.local/share/applications")
        desktop_dirs.add(user_dir)

        xdg_data_dirs_env = os.environ.get("XDG_DATA_DIRS")

        if xdg_data_dirs_env:
            for data_dir in xdg_data_dirs_env.split(":"):
                if data_dir:
                    app_dir = os.path.join(data_dir, "applications")
                    desktop_dirs.add(app_dir)

        app_map = {}
        for directory in desktop_dirs:
            if os.path.exists(directory):
                for file in os.listdir(directory):
                    if file.endswith(".desktop"):
                        file_path = os.path.join(directory, file)
                        try:
                            config = configparser.ConfigParser(interpolation=None)
                            config.read(file_path, encoding="utf-8")

                            if "Desktop Entry" in config:
                                de = config["Desktop Entry"]

                                if de.get("NoDisplay", "false").lower() == "true":
                                    continue
                                if de.get("Hidden", "false").lower() == "true":
                                    continue

                                name = de.get("Name")
                                exec_cmd = de.get("Exec")

                                if name and exec_cmd:
```

```
                                    exec_cmd = self._cleanup_exec(exec_cmd)
                                    app_map[name] = exec_cmd
                        except Exception:
                            pass
        return app_map

    def _cleanup_exec(self, exec_cmd: str) -> str:
        exec_cmd = exec_cmd.replace("%%", "%")
        for code in ("%f", "%F", "%u", "%U", "%i", "%c", "%k"):
            exec_cmd = exec_cmd.replace(code, "")
        return exec_cmd.strip()

    def get_all_apps(self) -> str:
        return ";".join(sorted(self._build_app_map().keys()))

    def run(self, text: str):
        text = text.strip()
        try:
            exec_cmd = self._find_exec_cmd(text)
            if not exec_cmd:
                return f"App '{text}' wurde nicht gefunden."

            parts = shlex.split(exec_cmd)
            subprocess.Popen(
                parts, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL
            )
            return f"Sucsessfully started '{text}'"
        except Exception as e:
            return f"Error Starting '{text}': {e}"

    def _find_exec_cmd(self, text: str) -> str:
        if text in self.app_map:
            return self.app_map[text]

        for name, cmd in self.app_map.items():
            if name.lower() == text.lower():
                return cmd

        for name, cmd in self.app_map.items():
            if text.lower() in name.lower():
                return cmd

        return ""
```

Listing 5.7: AppPlugin als konkretes Plugin

5.6 Zusammenspiel: Agent, Plugins und ReAct-Loop

Der typische Ablauf einer Anfrage kombiniert mehrere etablierte Forschungsrichtungen der Agentic AI:

1. ReAct-basiertes Reasoning Yao et al. (2022),
2. LLM-Tool-Use Schick et al. (2023),
3. modulare Softwarearchitekturen Gamma et al. (1994),
4. agentische Selbstorganisation Wang et al. (2024),
5. optional: multi-agentische Koordination Du et al. (2023), Hong et al. (2023).

Diese Architektur macht Tapyre zu einem echten *Agentic AI*-System: Das LLM dient als Steuerungsinstanz, die eigenständig Tools auswählt, Aktionen plant und iterativ Entscheidungen trifft, während die Plugin-Struktur maximale Erweiterbarkeit sicherstellt.

Appendix

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literaturverzeichnis

Ashish Vaswani, Noam Shazeer, N. P. J. U. L. J. A. N. G. L. K. I. P. (2017), 'Arxiv', https://arxiv.org/abs/1706.03762. Zugriff 2025.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M. et al. (2020), Language models are few-shot learners, *in* 'Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS)'.

Chen, W. (2023), 'Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks', *arXiv preprint arXiv:2305.10200* .

Cohan, A., Feldman, S., Beltagy, I., Downey, D. & Weld, D. (2020), 'Specter: Document-level representation learning for scientific papers', *arXiv preprint arXiv:2004.07180* .
URL: *https://arxiv.org/abs/2004.07180*

Couper, M. P. (2001), 'Web Survey Research: Challenges and Opportunities', Proceedings of the Annual Meeting of the American Statistical Association.

Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2018), 'Bert: Pre-training of deep bidirectional transformers for language understanding', *arXiv preprint arXiv:1810.04805* .
URL: *https://arxiv.org/abs/1810.04805*

Diekmann, A. (1999), *Empirische Sozialforschung. Grundlagen, Methoden, Anwendungen*, fifth edn, Rowohlts Enzyklopaedie, Reinbeck bei Hamburg.

Dillman, D. A., Tortora, R. & Bowker, D. (1998), Principles for Constructing Web Surveys, Technical report, SESRC.

Docker Inc. (2025), 'Docker engine security', https://docs.docker.com/engine/security/. Zugriff am: 05.12.2025.

Du, N., Liu, H., Li, Y. et al. (2023), 'Improving factuality and reasoning in language models through multiagent debate', *arXiv preprint arXiv:2305.14325* .

Fielding, R. T. (2000), Architectural Styles and the Design of Network-based Software Architectures, PhD thesis, University of California, Irvine.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learning*, MIT Press. http://www.deeplearningbook.org.

Gray, J. & Reuter, A. (1992), *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann.

Hochreiter, S. & Schmidhuber, J. (1997), *long short-term memory*. https://www.bioinf.jku.at/publications/older/2604.pdf.

Hong, B., Tang, Y., Li, H. et al. (2023), 'Metagpt: Meta programming for multi-agent collaborative framework', *arXiv preprint arXiv:2308.00352* .

Jurafsky, D. & Martin, J. H. (2023), *Speech and Language Processing*. Online version.
URL: *https://web.stanford.edu/ jurafsky/slp3/*

LangChain (2023), 'Langchain documentation', https://python.langchain.com/. Accessed 2024.

Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998), 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE* **86**(11), 2278–2324.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N. et al. (2020), 'Retrieval-augmented generation for knowledge-intensive nlp tasks', *arXiv preprint arXiv:2005.11401* .
URL: *https://arxiv.org/abs/2005.11401*

Loc (2004), 'Corporate value statement'. Einstiegsseite zum Unternehmensleitbild.
URL: *http://www.lockheedmartin.com/wms/findPage.do*

Malkov, Y. A. & Yashunin, D. A. (2020), 'Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs', *IEEE Transactions on Pattern Analysis and Machine Intelligence* **42**(4), 824–836.

Mikolov, T., Chen, K., Corrado, G. & Dean, J. (2013), 'Efficient estimation of word representations in vector space', *arXiv preprint arXiv:1301.3781* .
URL: *https://arxiv.org/abs/1301.3781*

Nachname, V. (2024), 'Titel der webseite', https://www.beispielseite.de. Zugriff am 15. November 2024.

Nandakumar, K., Cohan, A., Feldman, S., Downey, D. & Beltagy, I. (2023), Specter2: Building better document-level representations, *in* 'Findings of the Association for Computational Linguistics (ACL)'.

NVIDIA Corporation (2025), *CUDA C++ Programming Guide*. Zugriff am: 05.12.2025.

OpenAI (2024), 'Model context protocol', https://github.com/modelcontextprotocol. Accessed 2024.

Oracle Corporation (2024), *MySQL 8.4 Reference Manual: InnoDB Index Types*. Zugriff am: 05.12.2025.
URL: *https://dev.mysql.com/doc/refman/8.4/en/innodb-index-types.html*

Paszke, A., Gross, S., Massa, F. et al. (2019), Pytorch: An imperative style, high-performance deep learning library, *in* 'Advances in Neural Information Processing Systems 32 (NeurIPS 2019)'.

Percona (2024), 'Understanding mysql indexes: Types, benefits, and best practices'. Zugriff am: 05.12.2025.
URL: *https://www.percona.com/blog/understanding-mysql-indexes-types-best-practices/*

Qdrant Technologies (2024*a*), 'Hnsw indexing fundamentals', https://qdrant.tech/course/essentials/day-2/what-is-hnsw/. Zugriff am: 05.12.2025.

Qdrant Technologies (2024*b*), 'Qdrant concepts: Collections', https://qdrant.tech/documentation/concepts/collections/. Zugriff am: 05.12.2025.

Qdrant Technologies (2024*c*), 'Qdrant concepts: Indexing', https://qdrant.tech/documentation/concepts/indexing/. Zugriff am: 05.12.2025.

Qdrant Technologies (2024*d*), 'Qdrant documentation', https://qdrant.tech/documentation/. Zugriff am: 05.12.2025.

Qdrant Technologies (2024*e*), 'Similarity search in qdrant', https://qdrant.tech/documentation/concepts/search/. Zugriff am: 05.12.2025.

Quirós, G. (2024), 'How containers work: Layers, overlayfs, namespaces & cgroups'. Zugriff am: 05.12.2025.
URL: *https://k8studio.io/tutorials/container-architecture-namespaces-cgroups-overlayfs/*

Reips, U.-D. (2002), 'Standards for Internet-Based Experimenting', *Experimental Psychology* **1**(4), 243–256.

Ronacher, A. & Contributors, F. (2024), 'Flask documentation', https://flask.palletsprojects.com/. Zugriff am: 05.12.2025.

Schick, T., Dwivedi-Yu, J., Vu, T. et al. (2023), 'Toolformer: Language models can teach themselves to use tools', *arXiv preprint arXiv:2302.04761* .
URL: *https://arxiv.org/abs/2302.04761*

Titel der Website (n.d.), Website. (Abgerufen am: 2017-07-11).
URL: *http://www.musterseite.de*

Touvron, H., Lavril, T., Izacard, G., Martinet, X. et al. (2023), 'Llama: Open and efficient foundation language models', *arXiv preprint arXiv:2302.13971* .

Wang, Z., Zhu, C., Lin, Y. & Zhou, J. (2024), 'A survey on agentic large language models', *arXiv preprint arXiv:2401.05561* .

Yao, S., Deng, J. Z., Zhou, J., Wang, A., Lo, Y. & Goodman, N. (2022), 'React: Synergizing reasoning and acting in language models', *arXiv preprint arXiv:2210.03629* .
URL: *https://arxiv.org/abs/2210.03629*