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“ eines neuronalen Netzes ist dabei in den Gewichten und Bias-Werten gespeichert, die während des Trainingsprozesses mithilfe von Optimierungsverfahren wie dem Gradientenabstieg angepasst werden.

Die Fähigkeit eines einzelnen Neurons, lineare Entscheidungsgrenzen 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. Künstliche Neuronen stellen somit 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 (vgl. Goodfellow et al. (2016)).

Für diese Arbeit sind künstliche Neuronen besonders relevant, da sie die elementare Recheneinheit der verwendeten Transformer- und Embedding-Modelle darstellen. Alle später beschriebenen Verfahren zur semantischen Repräsentation wissenschaftlicher Texte basieren letztlich auf der Kombination und Optimierung dieser einfachen Bausteine.

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 die zentrale Eigenschaft dieser Architektur: Informationen fließen ausschließlich in eine Richtung, nämlich vom Eingang (*Input Layer*) über eine oder mehrere verdeckte Schichten (*Hidden Layers*) zum Ausgang (*Output Layer*). Rückkopplungen oder Schleifen sind nicht vorhanden.

Ein FNN besteht aus mehreren künstlichen Neuronen, die schichtweise miteinander verbunden sind. Jedes Neuron berechnet aus seinen Eingaben eine gewichtete Summe und wendet anschließend eine Aktivierungsfunktion wie ReLU, Sigmoid oder Tanh an. Dadurch ist das Netzwerk in der Lage, auch komplexe und nichtlineare Zusammenhänge zu modellieren.

Das Training eines FNNs erfolgt mittels des Verfahrens der *Backpropagation*. Dabei wird zunächst der Fehler zwischen der vorhergesagten Ausgabe des Netzes und dem tatsächlichen Zielwert berechnet. Anschließend wird dieser Fehler schrittweise durch das Netzwerk zurückpropagiert, um die Gewichte

so anzupassen, dass der Fehler in zukünftigen Durchläufen minimiert wird (vgl. Goodfellow et al. (2016)).

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 – einschließlich Transformer – lassen sich als Weiterentwicklungen dieses grundlegenden Prinzips verstehen, bei denen zusätzliche Mechanismen wie Attention oder spezielle Schichttypen eingeführt wurden (vgl. Goodfellow et al. (2016)).

In dieser Arbeit dienen FNNs vor allem als konzeptionelle Grundlage, um den Übergang von einfachen neuronalen Netzen zu komplexeren Architekturen wie Transformern nachvollziehbar zu machen.

1.3 Rekurrente Neuronale Netze (RNN, LSTM)

Rekurrente Neuronale Netze (RNNs) wurden entwickelt, um sequenzielle Daten wie Texte, Sprache, Musik oder Zeitreihen zu verarbeiten. Im Gegensatz zu Feed-Forward- oder Convolutional-Netzen besitzen RNNs Rückkopplungen, sodass Informationen aus vorherigen Zeitschritten in die Verarbeitung aktueller Eingaben einfließen können. Dadurch verfügen RNNs über eine Art internes „Gedächtnis“.

Ein einfaches RNN kombiniert zu jedem Zeitschritt den aktuellen Eingabewert mit dem vorherigen internen Zustand. Dieses Verfahren eignet sich gut für kurze Sequenzen, stößt jedoch bei längeren Abhängigkeiten an seine Grenzen. Ursache hierfür ist das sogenannte *Vanishing Gradient Problem*, bei dem Gradienten während des Trainings stark abnehmen und relevante Informationen verloren gehen.

Zur Lösung dieses Problems wurden Long Short-Term Memory Netze (LSTMs) entwickelt. LSTMs verfügen über spezielle Schaltelemente, sogenannte *Gates*, die steuern, welche Informationen gespeichert, weitergegeben oder verworfen werden. Dadurch sind LSTMs in der Lage, relevante Informationen über längere Zeiträume hinweg zu behalten und stabiler zu trainieren als klassische RNNs (vgl. Hochreiter & Schmidhuber (1997)).

LSTMs wurden über viele Jahre erfolgreich in der Sprachverarbeitung eingesetzt, etwa für maschinelle Übersetzung oder Textklassifikation. In modernen NLP-Systemen werden sie jedoch zunehmend durch Transformer-Modelle ersetzt, da diese effizienter trainierbar sind und besser mit sehr langen Texten umgehen können. Für diese Arbeit sind RNNs und LSTMs daher insbesondere im historischen Kontext relevant (vgl. Hochreiter & Schmidhuber (1997)).

1.4 Die Transformer-Architektur

Transformer-Modelle wurden im Jahr 2017 mit dem Paper *Attention Is All You Need* vorgestellt und haben die natürliche Sprachverarbeitung grundlegend verändert (vgl. Ashish Vaswani (2017)). Im Gegensatz zu RNNs und LSTMs verarbeiten Transformer die Eingabe nicht sequenziell, sondern betrachten alle Token eines Textes gleichzeitig. Dies ermöglicht eine effiziente Parallelisierung auf modernen GPUs und verbessert den Umgang mit langen Texten erheblich.

Das zentrale Konzept der Transformer-Architektur ist die sogenannte Self-Attention. Dieser Mechanismus erlaubt es dem Modell, zu bewerten, welche Wörter eines Satzes für die Bedeutung eines anderen Wortes besonders relevant sind. Dadurch können auch weit entfernte Abhängigkeiten innerhalb eines Textes effektiv modelliert werden (vgl. Ashish Vaswani (2017)).

Ein weiterer wichtiger Bestandteil ist die Multi-Head Attention. Dabei werden mehrere Attention-Mechanismen parallel eingesetzt, die jeweils unterschiedliche Arten von Beziehungen erfassen können, etwa syntaktische, semantische oder thematische Zusammenhänge. Die Ergebnisse dieser Attention-Köpfe werden anschließend zusammengeführt (vgl. Ashish Vaswani (2017)).

Da Transformer keine inhärente Reihenfolge der Eingabe besitzen, werden sogenannte Positional Encodings verwendet. Diese kodieren die Position eines Tokens im Text und werden den Eingaberepräsentationen hinzugefügt, sodass das Modell die Struktur der Sequenz berücksichtigen kann (vgl. Ashish Vaswani (2017)).

Ein klassischer Transformer besteht aus einem Encoder und einem Decoder. Während der Encoder den Eingabetext in eine semantisch aussagekräftige Repräsentation überführt, dient der Decoder der Textgenerierung. In vielen modernen Anwendungen – darunter auch die in dieser Arbeit betrachteten Such- und Embedding-Modelle – wird ausschließlich der Encoder verwendet (vgl. Ashish Vaswani (2017)).

Transformer bilden die Grundlage der in diesem Projekt eingesetzten Sprach- und Embedding-Modelle und sind daher zentral für das Verständnis der folgenden Kapitel (vgl. Ashish Vaswani (2017)).

1.5 Bedeutung von Transformern für LLMs und Embeddings

Transformer-Modelle sind heute die Basis nahezu aller modernen Anwendungen der natürlichen Sprachverarbeitung. Sie bilden das Fundament großer Sprachmodelle (*Large Language Models, LLMs*) sowie leistungsfähiger Embedding-Modelle (vgl. Ashish Vaswani (2017), Brown et al. (2020), Touvron et al. (2023)).

Durch den Einsatz von Self-Attention können Transformer auch weit entfernte Abhängigkeiten innerhalb eines Textes erfassen. Dies ist insbesondere bei langen oder komplexen Dokumenten von entscheidender Bedeutung, wie sie etwa in wissenschaftlichen Publikationen vorkommen. Frühere Architekturen wie RNNs oder LSTMs konnten solche Zusammenhänge nur eingeschränkt abbilden (vgl. Ashish Vaswani (2017)).

Für Embedding-Modelle bieten Transformer den Vorteil, dass sie nicht nur einzelne Wörter, sondern die semantische Bedeutung ganzer Sätze oder Dokumente in dichten Vektorrepräsentationen erfassen. Transformer-Encoder eignen sich daher besonders für Aufgaben wie semantische Suche, Textklassifikation oder Empfehlungssysteme (vgl. Nandakumar et al. (2023)).

Moderne Embedding-Modelle wie *SPECTER2*, *Sentence-BERT* oder *E5* basieren auf Transformer-Encodern. Sie ermöglichen es, inhaltlich ähnliche Texte

im Vektorraum nahe beieinander abzulegen, während thematisch unterschiedliche Dokumente klar getrennt sind. Diese Eigenschaft ist essenziell für die semantische Suche, wie sie im Rahmen dieses Projekts zur wissenschaftlichen Literatursuche eingesetzt wird (vgl. Nandakumar et al. (2023)).

Zusammenfassend lässt sich festhalten, dass Transformer die technologische Grundlage moderner Sprachverarbeitung darstellen. Ohne diese Architektur wären leistungsfähige LLMs sowie präzise Embedding-Modelle nicht realisierbar, wodurch auch das in dieser Arbeit entwickelte Suchsystem in dieser Form nicht möglich wäre (vgl. 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 Teilbereich der Künstlichen Intelligenz, der sich mit der automatischen Verarbeitung und Analyse menschlicher Sprache beschäftigt. Ziel ist es, Texte so zu modellieren, dass Computersysteme sprachbasierte Aufgaben ausführen können, etwa das Beantworten von Suchanfragen, das Zusammenfassen von Texten oder die Klassifikation von Dokumenten. Moderne Anwendungen wie Suchmaschinen, Chatbots oder Sprachassistenten basieren maßgeblich auf Methoden des NLP (vgl. Jurafsky & Martin (2023)).

Während frühe NLP-Ansätze überwiegend auf regelbasierten oder statistischen Verfahren beruhten, wird heutiges NLP fast ausschließlich durch tiefe neuronale Netze geprägt. Eine zentrale Fragestellung dabei ist, wie sprachliche Bedeutung in numerischer Form repräsentiert werden kann. Diese numerischen Repräsentationen werden als Embeddings bezeichnet und bilden die Grundlage moderner NLP-Systeme.

2.1 Klassische NLP-Ansätze

Vor dem Aufkommen neuronaler Sprachmodelle wurden Texte hauptsächlich mithilfe statistischer Verfahren repräsentiert. Zu den bekanntesten Ansätzen zählen Bag-of-Words, TF–IDF sowie N-Gramm-Modelle. Diese Methoden basieren auf der Häufigkeit von Wörtern oder Wortfolgen, berücksichtigen jedoch weder semantische Beziehungen noch den sprachlichen Kontext.

So wird beispielsweise nicht erkannt, dass Begriffe wie „Auto“ und „Fahrzeug“ eine ähnliche Bedeutung haben oder dass ein Wort wie „Bank“ je nach Kontext unterschiedliche Bedeutungen annehmen kann (vgl. Jurafsky & Martin (2023)). Für einfache Klassifikations- oder Zählaufgaben können diese Modelle ausreichend sein, bei komplexeren Anwendungen wie semantischer Suche oder maschineller Übersetzung stoßen sie jedoch schnell an ihre Grenzen.

Diese Einschränkungen waren ein wesentlicher Motivationsfaktor für die Entwicklung semantischer Repräsentationen in Form von Embeddings.

2.2 Einführung in Embeddings

Da Computer ausschließlich mit numerischen Daten arbeiten, müssen sprachliche Informationen in eine geeignete mathematische Form überführt werden. Embeddings lösen dieses Problem, indem sie Wörter, Sätze oder ganze Dokumente als Vektoren in einem kontinuierlichen Vektorraum darstellen. Ziel dieser Repräsentation ist es, semantische Beziehungen geometrisch abzubilden.

Dabei gelten folgende grundlegende Prinzipien:

- Ähnliche Bedeutungen sollen durch ähnliche Vektoren repräsentiert werden.
- Unterschiedliche Bedeutungen sollen im Vektorraum weit voneinander entfernt liegen.
- Kontextinformationen sollen – sofern möglich – in die Repräsentation einfließen.

Embeddings bilden die Grundlage vieler moderner NLP-Anwendungen, insbesondere für Aufgaben wie Ähnlichkeitsberechnungen, Clustering oder semantische Suche. Auch in dieser Arbeit spielen Embeddings eine zentrale Rolle, da sie die Basis für den Vergleich wissenschaftlicher Texte darstellen.

2.3 Word Embeddings: Word2Vec und GloVe

Einen wesentlichen Fortschritt stellten Word-Embedding-Modelle wie Word2Vec dar. Diese Modelle ordnen jedem Wort einen festen Vektor zu und beruhen auf der Annahme, dass Wörter, die in ähnlichen Kontexten auftreten, auch ähnliche Bedeutungen haben. Dadurch entstehen semantische Strukturen im Vektorraum, die sich beispielsweise durch einfache Vektorrechnungen ausdrücken lassen:

$$\text{Koenig} - \text{Mann} + \text{Frau} \approx \text{Koenigin}$$

Modelle wie Word2Vec (vgl. Mikolov et al. (2013)) erfassen grundlegende semantische Beziehungen zwischen Wörtern sehr effektiv. Ein zentrales Problem dieser Ansätze besteht jedoch darin, dass jedes Wort unabhängig vom Kontext immer durch denselben Vektor repräsentiert wird. Mehrdeutige Wörter wie „Bank“ können somit nicht kontextabhängig interpretiert werden.

Diese Einschränkung führte zur Entwicklung kontextualisierter Embedding-Modelle.

2.4 Kontextualisierte Embeddings

Mit dem Einsatz tiefer neuronaler Netze entstanden Modelle, die Wortbedeutungen abhängig vom jeweiligen Kontext darstellen können. Ein bekanntes Beispiel ist ELMo, das für ein Wort unterschiedliche Vektoren erzeugt, je nachdem, in welchem Satz es vorkommt. Dadurch kann zwischen verschiedenen Bedeutungen desselben Wortes unterschieden werden.

Diese kontextualisierten Embeddings stellten einen wichtigen Zwischenschritt dar und ebneten den Weg für leistungsfähigere Modelle auf Basis der Transformer-Architektur. Sie zeigten erstmals, dass eine dynamische, kontextabhängige Repräsentation von Sprache deutlich bessere Ergebnisse liefert als statische Wortvektoren.

Christian Vorhofer
Raphael Ladinig

2.5 Embeddings mit der Transformer-Architektur

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

Dies führt unter anderem zu folgenden Vorteilen:

- kontextabhängige Wortrepräsentationen,
- Satz- und Dokumentembeddings als einzelne Vektoren,
- präzisere semantische Modellierung,
- robuste Verarbeitung langer und komplexer Textstrukturen.

Insbesondere für wissenschaftliche Texte mit komplexer Terminologie und langen Abhängigkeiten sind Transformer-basierte Embeddings besonders geeignet. Aus diesem Grund kommen sie auch in dieser Arbeit zum Einsatz.

2.6 Relevanz für Tapyre Paper Search

Im Projekt *Tapyre Paper Search* werden Embeddings verwendet, um wissenschaftliche Publikationen in einem hochdimensionalen Vektorraum abzubilden. Dokumente mit ähnlichem inhaltlichem Schwerpunkt liegen dabei räumlich nahe beieinander, wodurch eine präzise semantische Suche ermöglicht wird.

Der Einsatz spezialisierter Modelle wie SPECTER2, die gezielt auf wissenschaftlichen Texten trainiert wurden, verbessert die Erkennung fachlicher Zusammenhänge zusätzlich. Die in diesem Kapitel beschriebenen NLP- und Embedding-Konzepte bilden somit die theoretische Grundlage für die im weiteren Verlauf der Arbeit vorgestellten Such- und Vergleichsmechanismen.

3 Einführung in Agentic AI

Agentic AI beschreibt ein modernes Paradigma der künstlichen Intelligenz, bei dem Modelle nicht nur reaktiv auf Eingaben antworten, sondern eigenständig Ziele verfolgen, Handlungen planen, Werkzeuge einsetzen und komplexe Aufgaben in mehreren Schritten ausführen. Ein *Agent* ist dabei ein KI-System, das Informationen beschafft, Entscheidungen trifft, Aktionen ausführt und auf Basis neuer Beobachtungen sein weiteres Vorgehen anpasst (vgl. Yao et al. (2022)).

Im Gegensatz zu klassischen Sprachmodellen, die primär als textbasierte Antwortgeneratoren fungieren, agieren agentische Systeme proaktiv und interaktiv. Sie kombinieren Sprachverarbeitung mit Planungs-, Entscheidungs- und Ausführungsmechanismen und können dadurch komplexe Workflows selbstständig bearbeiten.

3.1 ReAct: Reasoning + Acting

Ein grundlegender Ansatz innerhalb von Agentic AI ist das ReAct-Framework (Reasoning + Acting). Dieser Ansatz kombiniert explizite Gedankenschritte (*Reasoning*) mit konkreten Aktionen (*Acting*). ReAct wurde von Yao et al. vorgestellt und zielt darauf ab, Entscheidungsprozesse transparent mit tatsächlichen Handlungen zu verbinden (vgl. Yao et al. (2022)).

Ein typischer ReAct-Agent arbeitet in einer zyklischen Struktur:

1. Der Agent formuliert einen Gedankenschritt (*Thought*), um das weitere Vorgehen zu planen.
2. Er führt eine Aktion aus, beispielsweise eine Suchanfrage oder API-Abfrage.

3. Er erhält eine Beobachtung (*Observation*) als Ergebnis der Aktion.
4. Auf Basis dieser Information plant er den nächsten Schritt.

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

3.2 Tool Usage

Ein zentrales Merkmal agentischer Systeme ist die Fähigkeit, externe Werkzeuge (*Tools*) zu verwenden. Diese Tools erweitern die Fähigkeiten eines Sprachmodells erheblich, da sie den Zugriff auf externe Datenquellen und Rechenressourcen ermöglichen. Typische Beispiele sind:

- Datenbanken (z. B. Qdrant, MySQL),
- Web-APIs (z. B. arXiv oder Semantic Scholar),
- Dateisysteme,
- Suchfunktionen,
- Ausführung von Python-Skripten.

Der Agent wählt ein geeignetes Tool aus, übergibt die notwendigen Parameter, interpretiert die Ergebnisse und nutzt diese als Grundlage für weitere Entscheidungen. Dadurch fungiert das Sprachmodell als eine Art Steuerinstanz, die verschiedene Systeme orchestriert.

Im Kontext dieser Arbeit ermöglicht Tool Usage beispielsweise den Zugriff auf Desktop-Apps.

3.3 Model Context Protocol (MCP)

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

- wie Tools formal beschrieben werden,
- wie Kontextinformationen an Modelle übergeben werden,

- wie Modelle Aktionen anfordern,
- und wie Ergebnisse standardisiert zurückgegeben werden.

Der von OpenAI vorgestellte Standard erleichtert die Integration agentischer Systeme in bestehende Software-Architekturen und reduziert den Implementierungsaufwand komplexer Pipelines (vgl. OpenAI (2024)). Für agentische Systeme wie Tapyre bietet MCP eine strukturierte Grundlage zur Anbindung externer Dienste.

3.4 Agent-to-Agent Kommunikation

In umfangreicheren agentischen Systemen arbeiten häufig mehrere Agenten gemeinsam an einer Aufgabe. Diese sogenannte Agent-to-Agent-Kommunikation ermöglicht eine Aufteilung komplexer Problemstellungen auf spezialisierte Rollen. Beispiele hierfür sind:

- ein Analyse-Agent zur Extraktion relevanter Informationen,
- ein Recherche-Agent zur Suche nach geeigneten Quellen,
- ein Planungs-Agent zur Koordination des Workflows,
- ein Evaluations-Agent zur Überprüfung der Ergebnisse.

Durch diese Spezialisierung steigt sowohl die Robustheit als auch die Skalierbarkeit des Gesamtsystems. Fehler oder Unsicherheiten einzelner Agenten können durch andere Agenten ausgeglichen werden.

3.5 RAG: Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) verbindet Sprachmodelle mit externem, strukturiertem Wissen. Anstatt Antworten ausschließlich aus dem internen Modellwissen zu generieren, ruft ein RAG-System zunächst relevante Dokumente ab und erzeugt darauf aufbauend eine fundierte Antwort. Der Ansatz wurde von Lewis et al. eingeführt (vgl. Lewis et al. (2020)).

Ein typischer RAG-Agent arbeitet in zwei Phasen:

1. **Retrieval**: Suche nach relevanten Dokumenten, beispielsweise über eine Vektorsuche in einer Datenbank wie Qdrant.
2. **Generation**: Erzeugung einer Antwort unter Einbeziehung der abgerufenen Inhalte.

Für Tapyre Paper Search ist dieser Ansatz essenziell, da wissenschaftliche Dokumente zunächst semantisch identifiziert und anschließend analysiert, zusammengefasst oder weiterverarbeitet werden.

3.6 Multi-Agent Systems

Multi-Agent Systems (MAS) bestehen aus mehreren autonomen, oft spezialisierten Agenten, die parallel oder kooperativ zusammenarbeiten. Solche Systeme bieten mehrere Vorteile:

- höhere Robustheit durch Aufgabenverteilung,
- bessere Skalierbarkeit,
- parallele Bearbeitung komplexer Probleme,
- gezielte Spezialisierung einzelner Agenten.

Multi-Agent-Systeme werden zunehmend in Forschung, Retrieval-Systemen und komplexen KI-Anwendungen eingesetzt. Auch für zukünftige Erweiterungen von Tapyre Paper Search bieten sie eine flexible und leistungsfähige Architekturgrundlage.

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 ermöglichen. Dieses Kapitel erläutert die technischen Grundlagen und zeigt jeweils den praktischen Bezug zur Umsetzung des Systems (z. B. Deployment, Datenhaltung, semantische Suche, API-Schicht und GPU-beschleunigte Modellinferenz).

4.1 Docker und Containerisierung

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

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

Docker ermöglicht reproduzierbare Umgebungen, schnelle Deployments und konsistente Konfigurationen (vgl. Docker Inc. (2025)). Für Tapyre Paper Search ist dies insbesondere relevant, da Komponenten wie Qdrant, MySQL

und die Python-Dienste unabhängig voneinander, aber dennoch in einer einheitlichen Laufzeitumgebung betrieben werden können. Dadurch lassen sich Entwicklungs-, Test- und Produktionsumgebungen konsistent abbilden und Abhängigkeiten (z. B. Datenbankversionen) kontrollierbar halten.

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) insbesondere B^+-Bäume, die als Clustered und Secondary Indexes realisiert sind (vgl. Oracle Corporation (2024), Percona (2024)):

- **Clustered Index**: InnoDB speichert Daten entlang des Primärschlüssels; die Tabelle ist selbst als B^+-Baum organisiert.
- **Secondary Indexes**: weitere B^+-Bäume, die Verweise auf die Primärschlüsselzeilen enthalten.
- **Effiziente Suche**: durch die logarithmische Höhe der B^+-Bäume können Suchanfragen performant ausgeführt werden.

MySQL bietet außerdem ACID-Transaktionen, die Datenkonsistenz und Zuverlässigkeit sicherstellen (vgl. Gray & Reuter (1992)):

- **Atomicity**: Eine Transaktion wird entweder vollständig ausgeführt oder verworfen.
- **Consistency**: Integritätsregeln bleiben erhalten.
- **Isolation**: Parallel ausgeführte Transaktionen beeinflussen sich kontrolliert bzw. entsprechend des Isolation-Levels nicht unzulässig.
- **Durability**: bestätigte Änderungen bleiben dauerhaft gespeichert.

In Tapyre Paper Search wird MySQL zur Speicherung strukturierter Metadaten eingesetzt (z. B. Titel, Autoren, Kategorien, Importstatus). Das ermöglicht klassische Filter- und Verwaltungsabfragen (z. B. nach Datum, Kategorie oder Verarbeitungsstatus), während die semantische Suche separat über die Vektordatenbank erfolgt.

Christian Vorhofer
Raphael Ladinig

4.3 Qdrant und Approximate Nearest Neighbor Search

Qdrant ist eine spezialisierte Datenbank zur Speicherung und Suche von Vektorrepräsentationen (Embeddings) und ist für semantische Suchanwendungen optimiert (vgl. Qdrant Technologies (2024*d*)). Im Gegensatz zu relationalen Datenbanken steht hier nicht die textbasierte Volltextsuche im Vordergrund, sondern die Ähnlichkeitssuche in einem Vektorraum, in dem Dokumente durch numerische Repräsentationen beschrieben werden.

4.3.1 Speicherstruktur

Eine Qdrant-Collection besteht typischerweise aus Vektoren, Payload-Daten (Metadaten) sowie internen Segmenten zur Datenorganisation (vgl. Qdrant Technologies (2024*b*)):

- **Vektoren** (häufig 768 oder 1024 Dimensionen, abhängig vom Embedding-Modell),
- **Payload** wie Titel, DOI, Autoren oder Jahr,
- **Segmente** zur internen Aufteilung der Daten.

Diese Struktur ist auf schnelle Ähnlichkeitsabfragen und effiziente Datenzugriffe ausgelegt (vgl. Qdrant Technologies (2024*e*)). In Tapyre Paper Search erlaubt die Payload zudem das nachträgliche Filtern semantischer Treffer (z. B. nach Jahr oder Kategorie), während die eigentliche Ranking-Logik über Vektorsimilarität erfolgt.

4.3.2 Approximate Nearest Neighbor (ANN)

Da ein exakter Vergleich aller Vektoren bei großen Datenmengen ineffizient ist, verwenden Vektordatenbanken ANN-Algorithmen. Qdrant nutzt hierfür u. a. den HNSW-Index (Hierarchical Navigable Small World), einen graphbasierten ANN-Algorithmus (vgl. Malkov & Yashunin (2020), Qdrant Technologies (2024*c*,*a*)):

- mehrschichtige Graphstruktur,
- wenige Knoten auf höheren Ebenen (grobe Orientierung),
- viele Knoten auf unteren Ebenen (feine Suche),
- Navigation vom groben in den feinen Suchraum,
- schnelle Annäherung an die nächsten Nachbarn.

HNSW kombiniert hohe Geschwindigkeit mit hoher Treffergenauigkeit und eignet sich besonders für semantische Suchsysteme (vgl. Malkov & Yashunin (2020), Qdrant Technologies (2024*e*)). In Tapyre Paper Search ist dies zentral, da auch größere Paper-Sammlungen interaktiv durchsuchbar bleiben sollen.

4.3.3 Ähnlichkeitsmaße

Qdrant unterstützt verschiedene Distanz- bzw. Ähnlichkeitsmaße, darunter Kosinusähnlichkeit, euklidische Distanz und skalares Produkt (vgl. Qdrant Technologies (2024*e*)):

- **Kosinusähnlichkeit** (häufiger Standard für NLP-Embeddings),
- **Euklidische Distanz**,
- **Skalares Produkt** (Dot Product).

In Tapyre wird hauptsächlich die Kosinusähnlichkeit eingesetzt, da sie bei normalisierten Text-Embeddings eine robuste Näherung semantischer Ähnlichkeit bietet (vgl. Qdrant Technologies (2024*e*)).

4.4 Flask und REST-APIs

Flask ist ein leichtgewichtiges Webframework für Python und dient in Tapyre zur Bereitstellung von REST-APIs (vgl. Ronacher & Contributors (2024)). REST (*Representational State Transfer*) ist ein Architekturstil für verteilte Systeme, der u. a. von Fielding beschrieben wurde (vgl. Fielding (2000)). REST-basierte Schnittstellen verwenden standardisierte HTTP-Methoden:

- **GET**: Abfrage von Daten,
- **POST**: Erstellen neuer Ressourcen bzw. Auslösen von Verarbeitung,
- **PUT/PATCH**: Aktualisieren von Ressourcen,
- **DELETE**: Löschen von Ressourcen.

REST-APIs verwenden häufig JSON als Datenaustauschformat und sind zustandslos, d. h. jeder Request enthält alle für die Verarbeitung notwendigen Informationen (vgl. Fielding (2000)).

Ein vereinfachtes Beispiel für eine Anfrage an einen Embedding-Endpunkt:

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

In Tapyre Paper Search dient die API-Schicht als klar definierte Kommunikationsgrenze zwischen Komponenten (z. B. Frontend, Orchestrierung und Embedding-Service). Das erleichtert Tests, ermöglicht unabhängige Skalierung einzelner Dienste und reduziert Kopplung zwischen Systemteilen.

4.5 PyTorch und GPU-Beschleunigung

PyTorch ist ein Framework für Deep Learning und wird in Tapyre zur Berechnung von Embeddings eingesetzt (vgl. Paszke et al. (2019)). Da Transformer-Modelle wie SPECTER2 sehr rechenintensiv sind, wird die Ausführung auf GPUs genutzt, um Matrixoperationen massiv zu beschleunigen. CUDA stellt hierfür eine Plattform bereit, um entsprechende Operationen auf der Grafikkarte auszuführen (vgl. NVIDIA Corporation (2025)).

Relevante Vorteile von PyTorch sind unter anderem (vgl. Paszke et al. (2019)):

- dynamische Rechengraphen,
- breite Modell- und Tool-Unterstützung,
- nahtlose GPU-Nutzung,

- gute Integrierbarkeit in moderne NLP-Pipelines.

Für Tapyre Paper Search ist dies praktisch relevant, da die Embedding-Erzeugung typischerweise den größten Rechenanteil der Pipeline darstellt. GPU-Beschleunigung reduziert die Laufzeit der Indexierung und ermöglicht schnellere Aktualisierungen des Paper-Bestands.

4.6 Zusammenfassung

Docker stellt reproduzierbare Umgebungen für die komponentenbasierte Ausführung bereit (vgl. Docker Inc. (2025)), MySQL speichert strukturierte Metadaten effizient über Indizes und Transaktionen (vgl. Oracle Corporation (2024), Gray & Reuter (1992)), Qdrant ermöglicht performante semantische Vektorsuche mittels ANN/HNSW (vgl. Qdrant Technologies (2024*d*), Malkov & Yashunin (2020)), Flask dient als Kommunikationsschicht über REST (vgl. Ronacher & Contributors (2024), Fielding (2000)), und PyTorch führt rechenintensive Embedding-Modelle GPU-beschleunigt aus (vgl. Paszke et al. (2019), NVIDIA Corporation (2025)). Zusammen bilden diese Technologien die technische Grundlage für ein 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, die ausschließlich nach dem Prinzip *Input* $\rightarrow$ *Output* arbeiten, nutzt Tapyre einen Agenten, der eigenständig Entscheidungen trifft, Tools aufruft und Aufgaben iterativ in einem ReAct-Loop (Reasoning + Acting) ausführt (vgl. Yao et al. (2022), Schick et al. (2023), Wang et al. (2024)).

Der Agent fungiert dabei als steuernde Instanz zwischen dem Sprachmodell, externen Werkzeugen und der Systemumgebung. Die Kopplung zwischen Kernsystem, LLM und Plugins ist bewusst lose gehalten, um eine hohe Erweiterbarkeit, Wartbarkeit und Austauschbarkeit einzelner Komponenten zu gewährleisten. Dieses Architekturprinzip orientiert sich an etablierten Entwurfsmustern modularer Softwaresysteme (vgl. Gamma et al. (1994)).

Im Folgenden werden die wichtigsten Bausteine dieser Architektur erläutert:

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

5.1 Abstraktion der LLM-Schnittstelle

Um Tapyre unabhängig von einem konkreten Sprachmodell oder Anbieter zu halten, wird eine abstrakte LLM-Schnittstelle definiert. Jeder unterstützte LLM-Typ (z. B. Ollama oder cloudbasierte APIs) muss lediglich diese Schnittstelle implementieren. Dadurch kann das zugrunde liegende Sprachmodell ausgetauscht werden, ohne dass der Agent oder bestehende Plugins angepasst werden müssen.

Dieses Vorgehen folgt etablierten Architekturprinzipien wie Interface- und Factory-Abstraktionen, die eine klare Trennung von Schnittstelle und Implementierung vorsehen (vgl. Gamma et al. (1994)).

```
1 from abc import ABC, abstractmethod
2 from langchain_core.language_models.chat_models import BaseChatModel
3 class LLM(ABC):
4     @abstractmethod
5     def getLLM(self) -> BaseChatModel:
6         pass
```

Listing 5.1: Abstrakte LLM-Schnittstelle

Die Schnittstelle kapselt die Kommunikation mit dem Sprachmodell vollständig und stellt dem restlichen System eine einheitliche Interaktionsmöglichkeit zur Verfügung.

Eine konkrete Implementierung für das lokale LLM-Framework sieht wie folgt aus:

```
1  from abstractions.llm import LLM
2  from langchain_core.language_models.chat_models import BaseChatModel
3  from langchain_community.chat_models import ChatOllama
4
5  class OllamaLLM(LLM):
6      def __init__(self, model: str, host: str, temperature: float, max_tokens: int):
7          self.llm = ChatOllama(
8              model=model,
9              base_url=host,
10             temperature=temperature,
11             max_tokens=max_tokens
12         )
13     def getLLM(self) -> BaseChatModel:
14         return self.llm
```

Listing 5.2: OllamaLLM als konkrete Implementierung

Der restliche Systemkern arbeitet ausschließlich mit dem Interface `LLM` und erhält Instanzen von `BaseChatModel`. Welches konkrete Sprachmodell im Hintergrund verwendet wird, ist für den Agenten und die Plugins vollständig transparent.

Christian Vorhofer
Raphael Ladinig

5.2 Der Agent und der ReAct-Loop

Der Agent selbst ist ebenfalls als Abstraktion definiert und stellt lediglich eine zentrale Methode `ask` bereit, die eine Anfrage entgegennimmt und eine Antwort erzeugt:

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

Listing 5.3: Abstrakte Agent-Schnittstelle

Diese minimale Schnittstelle verdeutlicht die konzeptionelle Rolle des Agenten: Er fungiert als vermittelnde Instanz zwischen Benutzeranfrage, Sprachmodell und verfügbaren Tools.

Die konkrete Implementierung `PluginAgent` basiert auf dem Framework LangChain und verwendet den Agententyp `ZERO_SHOT_REACT_DESCRIPTION`. Dieser Agententyp setzt explizit das ReAct-Paradigma um, bei dem das Modell interne Gedankenschritte (*Reasoning*) mit konkreten Aktionen (*Acting*) kombiniert (vgl. Yao et al. (2022), LangChain (2023)).

```python
# simple_agent.py
from langchain.agents import initialize_agent, AgentType
from langchain.prompts import ChatPromptTemplate, SystemMessagePromptTemplate, HumanMessagePromptTemplate
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

Zentrale Aspekte dieser Implementierung sind:

- **ReAct-Loop**: Der Agent erzeugt intern eine Sequenz aus »Thought«, »Action« und »Observation«, wodurch Planung und Ausführung explizit miteinander verknüpft werden.
- **Tool-Auswahl**: Die verfügbare Tool-Liste wird dynamisch aus den geladenen Plugins generiert. Das LLM erhält ausschließlich deren Beschreibungen und entscheidet selbstständig, welches Tool geeignet ist.
- **Iterationsbegrenzung**: Die maximale Anzahl von Tool-Aufrufen wird begrenzt, um Endlosschleifen zu vermeiden und die Kontrolle über den Agentenlauf zu behalten (vgl. Schick et al. (2023)).

5.3 Plugins als Tools: Lose Kopplung durch Interfaces

Plugins stellen die eigentliche funktionale Erweiterbarkeit des Systems dar, etwa zum Starten von Anwendungen, zur Abfrage externer Datenquellen oder zur Interaktion mit dem Dateisystem. Sie sind über eine abstrakte Basisklasse definiert und können unabhängig vom Kernsystem implementiert werden.

```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:
```

```
31          return self.run(text)
```

Listing 5.5: Abstrakte Plugin-Basisklasse

Wesentliche Eigenschaften dieses Ansatzes sind:

- **Interface-basiertes Design**: Jedes Plugin implementiert lediglich die Methode `run()`.
- **Lose Kopplung**: Der Agent kennt ausschließlich die daraus erzeugten LangChain-Tools, nicht jedoch die konkrete Plugin-Implementierung.
- **Tool-Integration**: Mithilfe von `LCTool.from_function` wird die `run()`-Methode automatisch als Tool für den ReAct-Loop verfügbar gemacht (vgl. LangChain (2023)).

5.4 Dynamisches Laden der Plugins

Um neue Funktionalitäten ohne Änderungen am Hauptprogramm integrieren zu können, werden Plugins dynamisch zur Laufzeit geladen. Dieses Vorgehen entspricht gängigen Entwurfsmustern für modulare und erweiterbare Softwaresysteme (vgl. 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:
```

```
                    continue
                if inspect.isabstract(cls):
                    continue
                try:
                    instance = cls()
                    plugins.append(instance)
                except TypeError as e:
                    print(f"[PluginLoader] Couldn't load a instance of {cls.__name__}: {e}")

        return plugins

    def _import_module(self, file: Path):
        module_name = f"plugins_{file.stem}_{abs(hash(str(file)))}"
        spec = spec_from_file_location(module_name, file)
        if not spec or not spec.loader:
            print(f"[PluginLoader] Spec failed for: {file}")
            return None
        mod = module_from_spec(spec)
        try:
            spec.loader.exec_module(mod)  # type: ignore[attr-defined]
            return mod
        except Exception as e:
            print(f"[PluginLoader] Error while importing  {file}: {e}")
            return None
```

Listing 5.6: Dynamischer PluginLoader

Die Vorteile dieses Ansatzes sind:

- pluginbasierte Erweiterbarkeit ohne Neukompilierung,
- keine Änderungen am Kernsystem erforderlich,
- automatische Erkennung neuer Plugins mittels Reflection und Introspection.

5.5 Beispiel: AppPlugin zur Steuerung lokaler Anwendungen

Ein konkretes Beispiel für ein Plugin ist das `AppPlugin`. Dieses liest installierte Desktop-Anwendungen aus `.desktop`-Dateien aus und ermöglicht es dem Agenten, Anwendungen auf Benutzeranfrage zu starten.

```
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():
```

```
91              if name.lower() == text.lower():
92                  return cmd
93
94          for name, cmd in self.app_map.items():
95              if text.lower() in name.lower():
96                  return cmd
97
98          return ""
```

Listing 5.7: AppPlugin als konkretes Plugin

Das Plugin implementiert die abstrakte Plugin-Schnittstelle und wird dem Agenten als Tool zur Verfügung gestellt. Im ReAct-Loop kann das Sprachmodell selbstständig entscheiden, wann der Aufruf dieses Tools sinnvoll ist (vgl. Yao et al. (2022), LangChain (2023)).

5.6 Zusammenspiel von Agent, Plugins und ReAct-Loop

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

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

Durch diese Architektur wird Tapyre zu einem vollwertigen *Agentic AI*-System. Das Sprachmodell übernimmt die Rolle einer intelligenten Steuerungsinstanz, die eigenständig plant, Tools auswählt und Entscheidungen iterativ weiterentwickelt, während die Plugin-Struktur eine maximale Erweiterbarkeit des Gesamtsystems sicherstellt.

6 Architektur und Implementierung von Tapyre Paper Search

In diesem Kapitel wird die Architektur und Implementierung von Tapyre Paper Search beschrieben. Das System dient der automatisierten Verarbeitung, Indexierung und semantischen Durchsuchung wissenschaftlicher Publikationen. Ziel ist es, große Mengen an Forschungsarbeiten aus unterschiedlichen Quellen strukturiert aufzubereiten und sowohl klassisch über Metadaten als auch semantisch über Vektorrepräsentationen durchsuchbar zu machen.

Die Architektur folgt einem modularen Ansatz mit klar getrennten Verantwortlichkeiten für Datenbeschaffung, Textverarbeitung, Embedding-Erzeugung, Speicherung und Orchestrierung. Dieses Design orientiert sich an etablierten Softwarearchitekturprinzipien und ermöglicht eine hohe Wartbarkeit, Erweiterbarkeit sowie den einfachen Austausch einzelner Komponenten.

Abgrenzung zu klassischen Big-Data-Architekturen Im Rahmen dieser Diplomarbeit wurde bewusst auf den Einsatz klassischer Big-Data-Technologien wie Hadoop, Spark oder verteilte Data-Lake-Architekturen verzichtet. Der Grund dafür liegt in den begrenzten zeitlichen, personellen und infrastrukturellen Ressourcen, die im Rahmen einer Diplomarbeit realistisch zur Verfügung stehen.

Stattdessen wurde ein ressourceneffizienter Ansatz gewählt, der sich auf die für die Umsetzung des Projekts tatsächlich notwendigen Daten und Verarbeitungsschritte konzentriert. Es werden ausschließlich jene Informationen

gespeichert, die für die semantische Suche und Verwaltung wissenschaftlicher Publikationen erforderlich sind. Dies umfasst insbesondere strukturierte Metadaten sowie kompakte Vektorrepräsentationen der Inhalte.

Dieser bewusste Verzicht auf umfangreiche Big-Data-Frameworks reduziert die Komplexität des Systems erheblich, erleichtert Deployment und Wartung und ermöglicht eine vollständige Umsetzung auf begrenzter Hardware. Gleichzeitig bleibt die Architektur modular und erweiterbar, sodass bei steigenden Datenmengen oder erweiterten Ressourcen eine spätere Skalierung und Integration zusätzlicher Technologien möglich wäre.

Im Folgenden werden die zentralen Bausteine der Architektur erläutert:

- abstrakte Kernschnittstellen für Datenquellen, Embedding-Modelle und Datenbanken,
- konkrete Implementierungen für arXiv, Specter2, Qdrant und MySQL,
- die PDF-Verarbeitung als zentrale technische Herausforderung,
- die Pipeline zur Orchestrierung des Gesamtprozesses,
- das Zusammenspiel von strukturierter und semantischer Suche.

6.1 Abstraktion der Datenquellen

Um unterschiedliche wissenschaftliche Datenquellen flexibel integrieren zu können, wird eine abstrakte Schnittstelle für Datenprovider definiert. Diese legt fest, wie neue Dokumente geladen und bereitgestellt werden, ohne dass der restliche Verarbeitungsprozess von der konkreten Quelle abhängig ist.

```python
from abc import ABC, abstractmethod

class DataProvider(ABC):
    @abstractmethod
    def next(self):
        pass

    @abstractmethod
    def hasNext(self) -> bool:
        pass
```

Listing 6.1: Abstrakte Schnittstelle für Datenquellen

Die Schnittstelle definiert ein einheitliches Vertragsmodell für den Zugriff auf Papers und deren Metadaten. Dadurch können neue Quellen, etwa

PubMed oder lokale Archive, ergänzt werden, ohne bestehende Komponenten anpassen zu müssen. Dieses Vorgehen entspricht dem Open-Closed-Prinzip, bei dem Software für Erweiterungen offen, für Änderungen jedoch geschlossen bleibt.

6.2 als konkrete Datenquelle

Eine konkrete Implementierung dieser Schnittstelle stellt der Datenprovider für arXiv dar. Dieser ist für das Abrufen von Metadaten sowie das Herunterladen der zugehörigen PDF-Dokumente verantwortlich.

```
1  import os
2  import re
3  import time
4  import json
5  import requests
6  from pathlib import Path
7
8  from src.core.data_provider import DataProvider
9  from src.impl.logger import get_logger
10
11
12 class ArxivDataProvider(DataProvider):
13     def __init__(
14         self,
15         first_id: str = "",
16         last_id: str = "",
17         rate_limit_seconds: float = 3.0,
18         max_retries: int = 3,
19     ):
20         self.logger = get_logger(__name__)
21         self.logger.info("Initializing ArxivDataProvider")
22
23         self.first_id = first_id
24         self.last_id = last_id
25         self.current_id = first_id
26         self.last_pull = 0.0
27         self.finished = False
28         self.rate_limit_seconds = rate_limit_seconds
29         self.max_retries = max_retries
30
31         state_path = os.getenv("ARXIV_STATE_FILE", "/app/state/arxiv_state.json")
32         self.state_file = Path(state_path)
33
34         self.state_file.parent.mkdir(parents=True, exist_ok=True)
35
36         self._load_state()
37
38     def _load_state(self) -> None:
39         if not self.state_file.exists():
40             self.logger.info("[state] No state file found, starting from first_id.")
41             return
42
43         try:
44             with self.state_file.open("r", encoding="utf-8") as f:
45                 state = json.load(f)
46
47             saved_current = state.get("current_id")
48             saved_finished = state.get("finished", False)
49
50             if saved_current:
51                 self.logger.info(
```

```
                    f"[state] Restoring state from {self.state_file}: "
                    f"current_id={saved_current}, finished={saved_finished}"
                )
                self.current_id = saved_current
                self.finished = saved_finished
        except Exception as e:
            self.logger.error(f"[state] Failed to load state file: {e}", exc_info=True)

    def _save_state(self) -> None:
        tmp_file = self.state_file.with_suffix(".tmp")
        data = {
            "current_id": self.current_id,
            "first_id": self.first_id,
            "last_id": self.last_id,
            "finished": self.finished,
            "last_pull": self.last_pull,
        }

        try:
            with tmp_file.open("w", encoding="utf-8") as f:
                json.dump(data, f)

            os.replace(tmp_file, self.state_file)
            self.logger.debug(
                f"[state] Saved state: current_id={self.current_id}, finished={self.finished}"
            )
        except Exception as e:
            self.logger.error(f"[state] Failed to save state: {e}", exc_info=True)
            if tmp_file.exists():
                try:
                    tmp_file.unlink()
                except OSError:
                    pass

    def hasNext(self) -> bool:
        self.logger.debug(f"[hasNext] Current ID: {self.current_id}, finished={self.finished}")
        return not self.finished

    def next(self):
        """Fetch the next PDF from arXiv sequentially."""
        if self.finished:
            self.logger.info("[next] No more IDs to process     finished.")
            return None

        while True:
            next_id = self._get_next_id(self.current_id)
            self.logger.debug(f"[next] Next ID candidate: {next_id}")

            if next_id == self.last_id or next_id is None:
                self.logger.info("[next] Reached last ID or invalid next ID     marking finished.")
                self.finished = True
                self._save_state()
                return None

            self.current_id = next_id
            pdf_data = self._fetch_pdf(next_id)

            if pdf_data and len(pdf_data) > 0:
                self.logger.info(
                    f"[next] Successfully fetched PDF for {next_id} ({len(pdf_data)} bytes)"
                )
                self._save_state()
                break

            self.logger.warning(f"[next] No valid PDF found for {next_id}, continuing...")

        return next_id, pdf_data

    def _get_next_id(self, current_id: str) -> str | None:
        """Increment arXiv ID and force rollover at 10000."""
        match = re.match(r'arXiv:(\d{2})(\d{2})\.(\d{4,5})(?:v\d+)?', current_id)
        if not match:
            self.logger.warning(f"[get_next_id] Invalid current ID format: {current_id}")
            return None
```

```
127         yy, mm, num = map(int, match.groups())
128
129         ROLLOVER_LIMIT = 10000
130         num += 1
131
132         if num >= ROLLOVER_LIMIT:
133             num = 0
134             mm += 1
135             if mm > 12:
136                 mm = 1
137                 yy += 1
138
139             if yy > 99 or (yy == 7 and mm < 4):
140                 return None
141
142         next_id = f'arXiv:{yy:02d}{mm:02d}.{num:05d}'
143         return next_id
144
145     def _fetch_pdf(self, paper_id: str) -> bytes:
146         """Download PDF from arXiv with retries and rate limiting."""
147         attempts = 0
148         while attempts < self.max_retries:
149             now = time.time()
150             wait_time = self.last_pull + self.rate_limit_seconds - now
151             if wait_time > 0:
152                 self.logger.debug(f"[fetch_pdf] Rate limiting: sleeping {wait_time:.2f}s")
153                 time.sleep(wait_time)
154
155             self.last_pull = time.time()
156
157             match = re.match(r'arXiv:(\d{4,6}\.\d{4,5})', paper_id)
158             if not match:
159                 self.logger.error(f"[fetch_pdf] Invalid arXiv ID format: {paper_id}")
160                 return b""
161
162             arxiv_id = match.group(1)
163             url = f"https://arxiv.org/pdf/{arxiv_id}.pdf"
164             self.logger.info(
165                 f"[fetch_pdf] Fetching PDF from {url} (attempt {attempts+1}/{self.max_retries})"
166             )
167
168             try:
169                 response = requests.get(url, timeout=10)
170                 status = response.status_code
171
172                 if status == 200:
173                     self.logger.debug(
174                         f"[fetch_pdf] 200 OK for {arxiv_id} ({len(response.content)} bytes)"
175                     )
176                     return response.content
177
178                 elif status == 404:
179                     self.logger.warning(f"[fetch_pdf] 404 Not Found for {arxiv_id}")
180                     return b""
181
182                 else:
183                     attempts += 1
184                     backoff = 5 * 60 * 60
185                     self.logger.critical(
186                         f"[fetch_pdf] CRITICAL: Unexpected HTTP {status} for {arxiv_id}, "
187                         f"retrying in {backoff}s (attempt {attempts}/{self.max_retries})"
188                     )
189                     time.sleep(backoff)
190
191             except requests.exceptions.RequestException as e:
192                 attempts += 1
193                 backoff = min(600, 100 * attempts)
194                 self.logger.error(
195                     f"[fetch_pdf] Request exception for {arxiv_id}: {e}, "
196                     f"retrying in {backoff}s (attempt {attempts}/{self.max_retries})",
197                     exc_info=True
198                 )
199                 time.sleep(backoff)
200
201         self.logger.critical(f"[fetch_pdf] Max retries reached for {paper_id}, giving up.")
```

```
202         return b""
```

Listing 6.2: arXivDataProvider zur Anbindung der arXiv-API

Der Provider verarbeitet unter anderem Titel, Autoren, Abstracts, Kategorien sowie die URL des PDF-Dokuments. Durch die klare Trennung zwischen Datenbeschaffung und nachgelagerter Verarbeitung bleibt das System robust gegenüber API-Änderungen und leicht auf weitere Quellen übertragbar.

6.3 PDF-Verarbeitung und Textextraktion

Die Umwandlung wissenschaftlicher PDF-Dokumente in maschinenlesbaren Text stellt eine der größten technischen Herausforderungen dar. PDFs enthalten häufig mehrspaltige Layouts, mathematische Formeln, Fußnoten sowie Seitenköpfe, die eine saubere Textextraktion erschweren.

```
1  from src.core.pdf_converter import PdfConverter
2  from src.impl.logger import get_logger
3  import fitz
4  from datetime import datetime, timedelta, timezone
5  import re
6  from typing import Optional
7  import unicodedata
8
9
10 class FitzPdfConverter(PdfConverter):
11     def __init__(self):
12         self.logger = get_logger(__name__)
13         self.logger.debug("FitzPdfConverter initialized.")
14
15     def pdf_to_string(self, pdf) -> str:
16         if isinstance(pdf, str):
17             self.logger.info("Opening PDF from file path.")
18         elif isinstance(pdf, bytes):
19             self.logger.info("Opening PDF from bytes stream.")
20             self.logger.debug("PDF bytes length: %s", len(pdf))
21         else:
22             self.logger.error("Invalid input type for pdf_to_string: %s", type(pdf))
23             raise ValueError("Input must be a file path or PDF binary content")
24
25         doc = None
26         full_text = ""
27         error_pages = 0
28
29         try:
30             if isinstance(pdf, str):
31                 doc = fitz.open(pdf)
32             else:
33                 doc = fitz.open(stream=pdf, filetype="pdf")
34
35             page_count = len(doc)
36             self.logger.debug("PDF opened successfully. Page count: %d", page_count)
37
38             for i, page in enumerate(doc, start=1):
39                 try:
40                     text = page.get_text()
41                     full_text += text
42                     if i % 10 == 0 or i == page_count:
```

```
                        self.logger.debug(
                            "Extracted text up to page %d/%d (current page length=%d).",
                            i, page_count, len(text)
                        )
                except Exception as e:
                    error_pages += 1
                    self.logger.error(
                        "Failed to extract text from page %d/%d: %s     skipping this page.",
                        i, page_count, e
                    )
                    continue

            if error_pages > 0:
                self.logger.warning(
                    "Text extraction finished with %d page errors (total pages=%d).",
                    error_pages, page_count
                )

            self.logger.info(
                "Finished extracting text from PDF. Total length: %d characters.",
                len(full_text)
            )
            return full_text

        finally:
            if doc is not None:
                doc.close()
                self.logger.debug("PDF document closed after text extraction.")

    def pdf_metadata(self, pdf) -> dict:
        if isinstance(pdf, str):
            self.logger.info("Opening PDF from file path for metadata.")
        elif isinstance(pdf, bytes):
            self.logger.info("Opening PDF from bytes stream for metadata.")
            self.logger.debug("PDF bytes length: %s", len(pdf))
        else:
            self.logger.error("Invalid input type for pdf_metadata: %s", type(pdf))
            raise ValueError("Input must be a file path or PDF binary content")

        doc = None
        try:
            try:
                if isinstance(pdf, str):
                    doc = fitz.open(pdf)
                else:
                    doc = fitz.open(stream=pdf, filetype="pdf")
            except Exception as e:
                self.logger.error(
                    "Failed to open PDF for metadata extraction: %s", e, exc_info=True
                )
                return {}

            metadata = doc.metadata or {}
            self.logger.debug("Raw metadata keys: %s", list(metadata.keys()))

            expected_keys = [
                'title', 'author', 'subject', 'keywords',
                'creator', 'producer', 'creationDate',
                'modDate', 'trapped'
            ]

            complete_metadata = {key: metadata.get(key) for key in expected_keys}

            if complete_metadata.get('creationDate'):
                original = complete_metadata['creationDate']
                parsed = self.parse_pdf_date(original)
                complete_metadata['creationDate'] = parsed
                self.logger.debug("Parsed creationDate: %s -> %s", original, parsed)

            if complete_metadata.get('modDate'):
                original = complete_metadata['modDate']
                parsed = self.parse_pdf_date(original)
                complete_metadata['modDate'] = parsed
                self.logger.debug("Parsed modDate: %s -> %s", original, parsed)

```

```
118             self.logger.info("Metadata extracted successfully.")
119             return complete_metadata
120
121         finally:
122             if doc is not None:
123                 doc.close()
124                 self.logger.debug("PDF document closed after metadata extraction.")
125
126     def clean_string(self, text: str) -> str:
127         before_len = len(text)
128         text = unicodedata.normalize("NFKC", text)
129
130         text = text.replace('\r\n', '\n').replace('\r', '\n')
131
132         WHITESPACE_CHARS = [
133             "\u00A0",
134             "\u2007",
135             "\u202F",
136             "\u2009",
137             "\u2002", "\u2003", "\u2004", "\u2005", "\u2006",
138             "\u2008", "\u200A",
139             "\u3000",
140             "\u180E",
141             "\u200B", "\u200C", "\u200D", "\u2060",
142         ]
143         for ch in WHITESPACE_CHARS:
144             text = text.replace(ch, ' ')
145
146         text = text.replace('    ', '-')
147         text = text.replace('    ', '-')
148         text = text.replace('    ', '-')
149         text = text.replace('  ', '')
150
151         text = text.replace('\n', ' ')
152
153         text = ' '.join(text.split())
154         after_len = len(text)
155         self.logger.debug("Cleaned string: length %d -> %d.", before_len, after_len)
156         return text
157
158     def chunk_string(self, text: str, chunk_size=500, overlap=50) -> list[str]:
159         self.logger.info("Chunking text with chunk_size=%d and overlap=%d.", chunk_size, overlap)
160         chunks = []
161         start = 0
162         text_len = len(text)
163         while start < text_len:
164             end = min(start + chunk_size, text_len)
165             chunks.append(text[start:end])
166             start += chunk_size - overlap
167
168         self.logger.info("Produced %d chunks from text of length %d.", len(chunks), text_len)
169         return chunks
170
171     def parse_pdf_date(self, pdf_date_str) -> Optional[datetime]:
172         if not pdf_date_str or not str(pdf_date_str).startswith('D:'):
173             self.logger.warning("PDF date string missing or not starting with 'D:': %s", pdf_date_str)
174             return None
175
176         original = pdf_date_str
177         pdf_date_str = pdf_date_str[2:]
178
179         match = re.match(
180             r"(\d{4})(\d{2})(\d{2})(\d{2})(\d{2})(\d{2})([+-])(\d{2})'?(\d{2})?",
181             pdf_date_str
182         )
183         if not match:
184             self.logger.warning("Failed to parse PDF date string: %s", original)
185             return None
186
187         year, month, day, hour, minute, second, tz_sign, tz_hour, tz_minute = match.groups()
188         dt = datetime(int(year), int(month), int(day), int(hour), int(minute), int(second))
189
190         offset = timedelta(hours=int(tz_hour or 0), minutes=int(tz_minute or 0))
191         if tz_sign == '-':
192             offset = -offset
```

```
193
194         parsed = dt.replace(tzinfo=timezone(offset))
195         self.logger.debug("Parsed PDF date '%s' into datetime '%s'.", original, parsed)
196         return parsed
```

Listing 6.3: PDF-zu-Text-Konvertierung mit PyMuPDF

Die Implementierung basiert auf der Bibliothek PyMuPDF (fitz) und extrahiert den Text seitenweise. Dabei wird bewusst auf komplexe Layout-Rekonstruktionen verzichtet, um eine hohe Verarbeitungsgeschwindigkeit zu gewährleisten. Diese Lösung stellt einen praxisnahen Kompromiss zwischen Textqualität und Performance dar und eignet sich besonders für die Verarbeitung großer Paper-Sammlungen, wie sie in diesem Projekt anfallen.

6.4 Abstraktion der Embedding-Erzeugung

Für die semantische Suche müssen Texte in hochdimensionale Vektoren überführt werden. Um unterschiedliche Embedding-Modelle flexibel einsetzen zu können, wird eine abstrakte Embedder-Schnittstelle definiert.

```
1 from abc import ABC, abstractmethod
2
3 class Embedder(ABC):
4     @abstractmethod
5     def embed(self, text: str):
6         pass
```

Listing 6.4: Abstrakte Embedder-Schnittstelle

Diese Schnittstelle kapselt die konkrete Modellimplementierung vollständig. Dadurch können verschiedene Modelle getestet, ausgetauscht oder parallel verwendet werden, ohne dass Änderungen an der Pipeline oder den Datenbankschichten notwendig sind. Diese Abstraktion ist insbesondere für experimentelle Evaluierungen von Vorteil.

6.5 Specter2 als semantisches Embedding-Modell

Als konkrete Implementierung kommt das Embedding-Modell Specter2 zum Einsatz. Dieses Transformer-basierte Modell wurde speziell für wis-

senschaftliche Texte trainiert und erzeugt Vektorrepräsentationen, die den inhaltlichen Kern eines Papers erfassen.

```python
from src.core.embedder import Embedder
from src.impl.logger import get_logger
from typing import List, Union
from transformers import AutoTokenizer, AutoModel
import torch


class Specter2Embedder(Embedder):
    def __init__(self, model_name: str = "allenai/specter2_base"):
        self.logger = get_logger(__name__)
        self.logger.info("[Specter2Embedder] Initializing model: %s", model_name)

        if torch.cuda.is_available():
            self.device = torch.device("cuda")
            device_name = torch.cuda.get_device_name(0)
            cap = torch.cuda.get_device_capability(0)
            self.logger.info("[CUDA] device=%s cap=sm_%d%d torch_cuda=%s",
                             device_name, cap[0], cap[1], torch.version.cuda)
            torch.backends.cudnn.benchmark = True
        elif torch.backends.mps.is_available():
            self.device = torch.device("mps")
            device_name = "Apple MPS (Metal)"
        else:
            self.device = torch.device("cpu")
            device_name = "CPU"

        self.logger.info("[Specter2Embedder] Using device: %s", device_name)

        try:
            self.tokenizer = AutoTokenizer.from_pretrained(model_name)
            self.model = AutoModel.from_pretrained(
                model_name,
                use_safetensors=True,
                trust_remote_code=False,
            )
            self.model.to(self.device)
            self.model.eval()
            self.logger.info("[Specter2Embedder] Model and tokenizer loaded successfully.")
        except Exception as e:
            self.logger.exception("[Specter2Embedder] Failed to load model '%s': %s", model_name, e)
            raise

    @torch.inference_mode()
    def embed(self, text: Union[str, List[str]]) -> torch.Tensor:
        if isinstance(text, str):
            self.logger.debug("[Specter2Embedder] Received single string for embedding.")
            text = [text]
        else:
            self.logger.debug("[Specter2Embedder] Received batch of %d texts for embedding.", len(text))

        try:
            encoded_input = self.tokenizer(
                text,
                padding=True,
                truncation=False,
                return_tensors="pt",
            )
            encoded_input = {k: v.to(self.device, non_blocking=True) for k, v in encoded_input.items()}

            self.logger.debug(
                "[Specter2Embedder] Tokenized input successfully (seq_len=%d).",
                int(encoded_input["input_ids"].shape[1])
            )

            if self.device.type == "cuda":
                with torch.amp.autocast(device_type="cuda", dtype=torch.float16):
                    model_output = self.model(**encoded_input)
            else:
                model_output = self.model(**encoded_input)


```

```
            embeddings = model_output.last_hidden_state[:, 0, :]  # CLS
            self.logger.debug("[Specter2Embedder] Generated embeddings shape: %s", tuple(embeddings.shape))

            return embeddings.detach().to("cpu")

        except Exception as e:
            self.logger.exception("[Specter2Embedder] Error during embedding: %s", e)
            raise
```

Listing 6.5: Specter2Embedder zur Erzeugung semantischer Vektoren

Durch den Einsatz eines domänenspezifischen Modells wird eine deutlich bessere semantische Abbildung wissenschaftlicher Inhalte erreicht als mit generischen Sprachmodellen. Dies ist insbesondere für die thematische Suche und Ähnlichkeitsbewertung von Publikationen entscheidend.

6.6 Abstraktion der Datenhaltung

Das System unterscheidet bewusst zwischen strukturierter Datenhaltung für Metadaten und vektorbasierten Repräsentationen für semantische Suche. Eine abstrakte Datenbankschnittstelle definiert die hierfür notwendigen Operationen.

```
from abc import ABC, abstractmethod

class Database(ABC):
    def __init__(self):
        self.connect()

    @abstractmethod
    def connect(self):
        pass

    def __del__(self):
        self.disconnect()

    @abstractmethod
    def disconnect(self):
        pass
```

Listing 6.6: Abstrakte Datenbankschnittstelle

Diese Trennung erlaubt es, unterschiedliche Datenbanktypen gezielt für ihre jeweiligen Stärken einzusetzen und bei Bedarf auszutauschen.

Christian Vorhofer
Raphael Ladinig

6.7 MySQL für strukturierte Metadaten

Metadaten wie Titel, Autoren, Kategorien oder Statusinformationen werden in einer relationalen Datenbank gespeichert.

```python
import os
from datetime import datetime
from sqlalchemy import create_engine, text
from sqlalchemy.orm import sessionmaker, scoped_session
from sqlalchemy.exc import SQLAlchemyError, IntegrityError, DataError

from src.models.paper import Paper
from src.models.base import Base
from src.core.database import Database
from src.impl.logger import get_logger


class MySQLDatabase(Database):
    def __init__(self):
        self.logger = get_logger(__name__)
        self.logger.info("[MySQLDatabase] Initialization started")

        user = os.getenv("MYSQL_USER")
        password = os.getenv("MYSQL_PASSWORD")
        host = os.getenv("MYSQL_HOST", "mysql_db")
        database = os.getenv("MYSQL_DATABASE")

        self.logger.debug("[MySQLDatabase] Env vars - USER: %s, HOST: %s, DB: %s", user, host, database)

        if not all([user, password, database]):
            raise ValueError("[MySQLDatabase] Missing env vars: MYSQL_USER, MYSQL_PASSWORD, MYSQL_DATABASE")

        self.db_url = f"mysql+pymysql://{user}:{password}@{host}/{database}"
        self.logger.info("[MySQLDatabase] Connection URL: %s", self.db_url)

        self.engine = None
        self.Session = None

        try:
            super().__init__()
            self.logger.info("[MySQLDatabase] super().__init__ successful")
        except Exception as e:
            self.logger.exception("[MySQLDatabase] Error calling super(): %s", e)
            raise

        try:
            self.logger.info("[MySQLDatabase] Connection established successfully")
        except Exception as e:
            self.logger.exception("[MySQLDatabase] Error while establishing connection: %s", e)
            raise

    def connect(self):
        self.logger.info("[MySQLDatabase] Creating engine and session...")
        self.engine = create_engine(self.db_url, echo=False, pool_pre_ping=True)
        self.Session = scoped_session(sessionmaker(bind=self.engine))
        self.logger.info("[MySQLDatabase] Engine and Session created. Creating tables...")
        Base.metadata.create_all(self.engine)
        self.logger.info("[MySQLDatabase] Tables created")

    def disconnect(self):
        self.logger.info("[MySQLDatabase] Disconnecting from database...")
        if self.Session:
            self.Session.remove()
        if self.engine:
            self.engine.dispose()
        self.logger.info("[MySQLDatabase] Disconnected")

    def get_session(self):
        return self.Session()

    def add_paper(
```

```
67          self,
68          arxiv_id: str,
69          text: str,
70          title: str = None,
71          author: str = None,
72          subject: str = None,
73          keywords: str = None,
74          creator: str = None,
75          producer: str = None,
76          creation_date: datetime = None,
77          modification_date: datetime = None,
78          trapped: str = None,
79          skip_if_exists: bool = True,
80      ):
81          session = self.get_session()
82          try:
83              if skip_if_exists and self.paper_exists(arxiv_id):
84                  self.logger.info("[MySQLDatabase] Paper %s already exists, skipping insert.", arxiv_id)
85                  return arxiv_id
86
87              self.logger.info("[MySQLDatabase] Inserting paper: %s", arxiv_id)
88              paper = Paper(
89                  arxiv_id=arxiv_id,
90                  text=text,
91                  title=title,
92                  author=author,
93                  subject=subject,
94                  keywords=keywords,
95                  creator=creator,
96                  producer=producer,
97                  creation_date=creation_date,
98                  modification_date=modification_date,
99                  trapped=trapped,
100             )
101             session.add(paper)
102             session.commit()
103             self.logger.info("[MySQLDatabase] Paper saved: %s", arxiv_id)
104             return arxiv_id
105
106         except IntegrityError as e:
107             session.rollback()
108             self.logger.warning("[MySQLDatabase] IntegrityError on %s, skipping. Detail: %s", arxiv_id, e)
109             return None
110
111         except DataError as e:
112             session.rollback()
113             author_len = len(author) if author is not None else 0
114             self.logger.warning("[MySQLDatabase] DataError on %s (author_len=%d), skipping. Detail: %s", arxiv_id, author_len, e)
115             return None
116
117         except SQLAlchemyError as e:
118             session.rollback()
119             self.logger.exception("[MySQLDatabase] Unexpected SQLAlchemyError on %s: %s", arxiv_id, e)
120             return None
121
122         finally:
123             session.close()
124
125     def get_paper_by_arxiv_id(self, arxiv_id: str):
126         session = self.get_session()
127         try:
128             self.logger.info("[MySQLDatabase] Fetching paper by ID: %s", arxiv_id)
129             return session.query(Paper).filter_by(arxiv_id=arxiv_id).first()
130         finally:
131             session.close()
132
133     def get_all_papers(self):
134         session = self.get_session()
135         try:
136             self.logger.info("[MySQLDatabase] Fetching all papers")
137             return session.query(Paper).all()
138         finally:
139             session.close()
140
141     def delete_paper(self, arxiv_id: str):
```

```
142             session = self.get_session()
143             try:
144                 self.logger.info("[MySQLDatabase] Deleting paper with ID: %s", arxiv_id)
145                 paper = session.query(Paper).filter_by(arxiv_id=arxiv_id).first()
146                 if paper:
147                     session.delete(paper)
148                     session.commit()
149                     self.logger.info("[MySQLDatabase] Paper deleted: %s", arxiv_id)
150                     return True
151                 self.logger.info("[MySQLDatabase] Paper not found: %s", arxiv_id)
152                 return False
153             except SQLAlchemyError as e:
154                 session.rollback()
155                 self.logger.exception("[MySQLDatabase] Error deleting paper %s: %s", arxiv_id, e)
156                 return False
157             finally:
158                 session.close()
159
160         def get_statistics(self):
161             session = self.get_session()
162             try:
163                 self.logger.info("[MySQLDatabase] Gathering database statistics")
164                 total_papers = session.query(Paper).count()
165                 latest_paper = session.query(Paper).order_by(Paper.creation_date.desc()).first()
166                 earliest_paper = session.query(Paper).order_by(Paper.creation_date.asc()).first()
167
168                 db_name = (self.engine.url.database if self.engine else None) or os.getenv("MYSQL_DATABASE")
169
170                 db_size = None
171                 if self.engine and db_name:
172                     with self.engine.connect() as conn:
173                         result = conn.execute(
174                             text("""
175                                 SELECT ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS size_mb
176                                 FROM information_schema.tables
177                                 WHERE table_schema = :db
178                             """),
179                             {"db": db_name},
180                         )
181                         db_size = result.scalar_one_or_none()
182
183                 stats = {
184                     "total_papers": total_papers,
185                     "latest_paper_date": latest_paper.creation_date if latest_paper else None,
186                     "earliest_paper_date": earliest_paper.creation_date if earliest_paper else None,
187                     "database_size_mb": db_size,
188                 }
189                 self.logger.info("[MySQLDatabase] Statistics: %s", stats)
190                 return stats
191             finally:
192                 session.close()
193
194         def paper_exists(self, arxiv_id: str) -> bool:
195             session = self.get_session()
196             try:
197                 return session.get(Paper, arxiv_id) is not None
198             finally:
199                 session.close()
```

Listing 6.7: MySQL-Datenbankanbindung für Paper-Metadaten

Relationale Datenbanken eignen sich besonders für konsistente Speicherung, relationale Abfragen und Filteroperationen, wie sie bei Metadaten häufig erforderlich sind.

6.8 Qdrant als Vektordatenbank

Für die semantische Suche werden die erzeugten Embeddings in einer spezialisierten Vektordatenbank gespeichert. Hierfür kommt Qdrant zum Einsatz, das effiziente Approximate-Nearest-Neighbor-Suchen ermöglicht.

```python
import os
import uuid
from qdrant_client import QdrantClient
from qdrant_client.http.models import PointStruct
from qdrant_client.models import Distance, VectorParams
from src.core.database import Database
from src.impl.logger import get_logger


class QdrantDatabase(Database):
    def __init__(self):
        self.logger = get_logger(__name__)
        self.collection_name = os.getenv("QDRANT_COLLECTION", "chunks")
        self.host = os.getenv("QDRANT_HOST", "qdrant_db")
        self.port = int(os.getenv("QDRANT_PORT", 6333))
        self.vector_size = int(os.getenv("VECTOR_SIZE", 768))
        self.client = None

        self.logger.info(
            "[QdrantDatabase] Initialized with host=%s, port=%d, collection=%s, vector_size=%d",
            self.host, self.port, self.collection_name, self.vector_size
        )

        try:
            super().__init__()
            self.logger.debug("[QdrantDatabase] Super init successful")
        except Exception as e:
            self.logger.exception("[QdrantDatabase] Error during super init: %s", e)
            raise

    def connect(self):
        self.logger.info("[QdrantDatabase] Connecting to Qdrant at %s:%d...", self.host, self.port)
        self.client = QdrantClient(host=self.host, port=self.port)

        try:
            self.client.get_collection(self.collection_name)
            self.logger.info("[QdrantDatabase] Collection '%s' already exists.", self.collection_name)
        except Exception:
            self.logger.warning("[QdrantDatabase] Collection '%s' not found. Creating a new one...", self.collection_name)
            try:
                self.client.recreate_collection(
                    collection_name=self.collection_name,
                    vectors_config=VectorParams(
                        size=self.vector_size,
                        distance=Distance.COSINE
                    )
                )
                self.logger.info("[QdrantDatabase] Collection '%s' created successfully.", self.collection_name)
            except Exception as e:
                self.logger.exception("[QdrantDatabase] Failed to create collection '%s': %s", self.collection_name, e)
                raise

    def disconnect(self):
        if self.client:
            self.client = None
            self.logger.info("[QdrantDatabase] Disconnected from Qdrant.")
        else:
            self.logger.warning("[QdrantDatabase] Disconnect called, but client was already None.")

    def add_chunk(
        self,
        arxiv_id: str,
        embedding,
        text: str,
```

```
65          max_retries: int = 3,
66          base_delay: float = 1.0,
67      ):
68          if not arxiv_id or embedding is None or text is None:
69              self.logger.error(
70                  "[QdrantDatabase] Missing required fields (arxiv_id=%s, embedding=%s, text=%s).",
71                  arxiv_id,
72                  type(embedding),
73                  "present" if text else "None",
74              )
75              raise ValueError("Both arxiv_id, embedding, and text are required.")
76
77          if hasattr(embedding, "tolist"):
78              embedding_list = embedding.tolist()
79          else:
80              embedding_list = embedding
81
82          if isinstance(embedding_list[0], (list, tuple)):
83              embedding_list = embedding_list[0]
84
85          pk = str(uuid.uuid4())
86          self.logger.debug(
87              "[QdrantDatabase] Adding chunk with ID %s for paper %s", pk, arxiv_id
88          )
89
90          point = PointStruct(
91              id=pk,
92              vector=embedding_list,
93              payload={
94                  "arxiv_id": arxiv_id,
95                  "text": text,
96              },
97          )
98
99          attempt = 0
100         while attempt < max_retries:
101             attempt += 1
102             try:
103                 self.client.upsert(
104                     collection_name=self.collection_name,
105                     points=[point],
106                     wait=True,
107                 )
108                 self.logger.info(
109                     "[QdrantDatabase] Added chunk %s for paper %s (attempt %d/%d)",
110                     pk,
111                     arxiv_id,
112                     attempt,
113                     max_retries,
114                 )
115                 return pk
116
117             except (
118                 ResponseHandlingException,
119                 httpx.RemoteProtocolError,
120                 httpcore.RemoteProtocolError,
121             ) as e:
122                 if attempt >= max_retries:
123                     self.logger.exception(
124                         "[QdrantDatabase] Giving up on chunk %s for paper %s after %d attempts: %s",
125                         pk,
126                         arxiv_id,
127                         attempt,
128                         e,
129                     )
130                     raise
131
132                 delay = min(base_delay * (2 ** (attempt - 1)), 30.0)
133                 self.logger.warning(
134                     "[QdrantDatabase] Transient error when upserting chunk %s "
135                     "for paper %s (attempt %d/%d): %s     retrying in %.1fs",
136                     pk,
137                     arxiv_id,
138                     attempt,
139                     max_retries,
```

```
                    e,
                    delay,
                )
                time.sleep(delay)

            except Exception as e:
                self.logger.exception(
                    "[QdrantDatabase] Failed to upsert chunk %s for paper %s (no retry): %s",
                    pk,
                    arxiv_id,
                    e,
                )
                raise

    def get_similar(self, embedding: list[float], top_k: int = 5):
        self.logger.debug("[QdrantDatabase] Searching for top %d similar embeddings.", top_k)

        if hasattr(embedding, "tolist"):
            embedding = embedding.tolist()

        if isinstance(embedding, list) and isinstance(embedding[0], list):
            embedding = embedding[0]

        try:
            search_result = self.client.search(
                collection_name=self.collection_name,
                query_vector=embedding,
                limit=top_k
            )
            self.logger.info("[QdrantDatabase] Found %d similar vectors.", len(search_result))
            return search_result
        except Exception as e:
            self.logger.exception("[QdrantDatabase] Error during similarity search: %s", e)
            raise

    def get_statistics(self):
        self.logger.info("[QdrantDatabase] Fetching collection statistics for '%s'", self.collection_name)
        try:
            stats = self.client.get_collection(self.collection_name)
            data = {
                "points_count": stats.points_count,
                "segments_count": getattr(stats, "segments_count", None),
            }
            self.logger.info("[QdrantDatabase] Statistics: %s", data)
            return data
        except Exception as e:
            self.logger.exception("[QdrantDatabase] Failed to fetch collection stats: %s", e)
            raise
```

Listing 6.8: Qdrant-Datenbank für semantische Suche

Durch den Einsatz HNSW-basierter Indexstrukturen können auch sehr große Paper-Sammlungen performant durchsucht werden, was eine zentrale Voraussetzung für semantische Suchanwendungen darstellt.

6.9 Pipeline zur Orchestrierung des Gesamtprozesses

Die Pipeline bildet das zentrale Orchestrierungselement des Systems. Sie verbindet Datenquelle, PDF-Verarbeitung, Embedding-Erzeugung und Spe-

icherung zu einem konsistenten und reproduzierbaren Ablauf.

```python
from tqdm.auto import tqdm
from src.impl.simple_data_provider import SimpleDataProvider
from src.impl.fitz_pdf_converter import FitzPdfConverter
from src.impl.specter_2_embedder import Specter2Embedder
from src.impl.mysql_database import MySQLDatabase
from src.impl.qdrant_database import QdrantDatabase
from src.impl.logger import get_logger
import os


class Pipeline:
    def __init__(self, relational_db, vector_db, data_provider, pdf_converter, embedder):
        self.mysql_db = relational_db
        self.qdrant_db = vector_db
        self.data_provider = data_provider
        self.pdf_converter = pdf_converter
        self.embedder = embedder
        self.logger = get_logger(__name__)
        self.logger.debug(
            "Pipeline initialized with %s, %s, %s, %s, %s",
            type(relational_db).__name__,
            type(vector_db).__name__,
            type(data_provider).__name__,
            type(pdf_converter).__name__,
            type(embedder).__name__,
        )

        self._tqdm_disable = False

        self._paper_bar_format = "{l_bar}{bar}| {n_fmt} papers [{elapsed}<{remaining}, {rate_fmt}]"
        self._chunk_bar_format = (
            "  {l_bar}{bar}| {n_fmt}/{total_fmt} chunks "
            "[{elapsed}<{remaining}, {rate_fmt}{postfix}]"
        )

    def process(self):
        self.logger.info("Pipeline processing started.")

        with tqdm(
            desc="Papers",
            unit="paper",
            dynamic_ncols=True,
            mininterval=0.5,
            bar_format=self._paper_bar_format,
            disable=self._tqdm_disable,
        ) as paper_bar:
            while self.data_provider.hasNext():
                result = self.data_provider.next()
                if result is None:
                    self.logger.warning("Data provider returned None     no more papers or an error occurred.")
                    break

                arxiv_id, pdf_data = result
                if arxiv_id is None or pdf_data is None:
                    self.logger.warning(
                        "Received invalid paper result (arxiv_id=%s, pdf_data=%s). Skipping...",
                        arxiv_id,
                        "None" if pdf_data is None else "bytes",
                    )
                    continue

                if self.mysql_db.paper_exists(arxiv_id):
                    self.logger.info("Paper %s already exists     skipping.", arxiv_id)
                    continue

                self.logger.info("Processing paper: %s", arxiv_id)

                try:
                    text = self.pdf_converter.pdf_to_string(pdf_data)
                except Exception as e:
                    self.logger.error("Failed to extract text for %s: %s", arxiv_id, e, exc_info=True)
                    continue

```

```
74                  if not text or not text.strip():
75                      self.logger.warning("No text extracted for %s     skipping.", arxiv_id)
76                      continue
77
78                  try:
79                      metadata = self.pdf_converter.pdf_metadata(pdf_data)
80                  except Exception as e:
81                      self.logger.error("Failed to extract metadata for %s: %s", arxiv_id, e, exc_info=True)
82                      metadata = {}
83
84                  cleaned_text = self.pdf_converter.clean_string(text)
85                  chunks = self.pdf_converter.chunk_string(cleaned_text)
86
87                  if not chunks:
88                      self.logger.warning("No chunks generated for %s     skipping embedding.", arxiv_id)
89                      paper_bar.update(1)
90                      continue
91
92                  self.mysql_db.add_paper(
93                      arxiv_id=arxiv_id,
94                      text=text,
95                      title=metadata.get("title"),
96                      author=metadata.get("author"),
97                      subject=metadata.get("subject"),
98                      keywords=metadata.get("keywords"),
99                      creator=metadata.get("creator"),
100                     producer=metadata.get("producer"),
101                     creation_date=metadata.get("creationDate"),
102                     modification_date=metadata.get("modDate"),
103                     trapped=metadata.get("trapped"),
104                 )
105                 self.logger.info("Created DB record for paper %s (%d chunks)", arxiv_id, len(chunks))
106
107                 try:
108                     embeddings = self.embedder.embed(chunks)
109                 except Exception as e:
110                     self.logger.error("Failed to embed chunks for %s: %s", arxiv_id, e, exc_info=True)
111                     paper_bar.update(1)
112                     continue
113
114                 with tqdm(
115                     total=len(chunks),
116                     desc=f"Embedding {arxiv_id}",
117                     unit="chunk",
118                     dynamic_ncols=True,
119                     mininterval=0.1,
120                     leave=False,
121                     bar_format=self._chunk_bar_format,
122                     disable=self._tqdm_disable,
123                 ) as chunk_bar:
124                     for idx, (chunk, embedding) in enumerate(zip(chunks, embeddings), start=1):
125                         self.qdrant_db.add_chunk(arxiv_id, embedding.tolist(), chunk)
126                         chunk_bar.update(1)
127                         if idx % 25 == 0 or idx == len(chunks):
128                             chunk_bar.set_postfix_str(f" last={idx}")
129                             self.logger.debug("Embedded %d/%d chunks for %s.", idx, len(chunks), arxiv_id)
130
131                 paper_bar.set_postfix_str(f"last={arxiv_id} chunks={len(chunks)}")
132                 paper_bar.update(1)
133
134         self.logger.info("Pipeline processing finished.")
```

Listing 6.9: Pipeline zur Verarbeitung und Indexierung von Papers

Der Gesamtprozess gliedert sich in folgende Schritte:

1. Abruf neuer Papers aus der Datenquelle,
2. Download und Textextraktion aus PDF-Dokumenten,
3. Aufteilung des Textes in Chunks und Erzeugung von Embeddings,

4. Speicherung von Metadaten und Vektorrepräsentationen.

Durch die klare Trennung dieser Schritte bleibt die Pipeline leicht erweiterbar, testbar und für agentische Erweiterungen geeignet.

6.10 Zusammenspiel der Komponenten

Das Zusammenspiel der beschriebenen Komponenten ermöglicht eine skalierbare und flexible Paper-Search-Plattform. Während relationale Datenbanken effiziente Metadatenabfragen erlauben, stellt die Vektordatenbank eine leistungsfähige semantische Suche bereit. Die modulare Architektur erlaubt es, neue Datenquellen, Embedding-Modelle oder Datenbanken mit minimalem Implementierungsaufwand zu integrieren.

Damit bildet Tapyre Paper Search die technische Grundlage für eine moderne, agentenfähige Forschungsplattform, die klassische Informationsretrieval-Ansätze mit aktuellen Methoden der semantischen Suche kombiniert.

6.11 Analyse der Performance-Charakteristika und Systemeinschränkungen

Diese Section analysiert beobachtete Performance-Effekte während des praktischen Betriebs von Tapyre Paper Search. Neben ressourcenbedingten Einschränkungen werden insbesondere Limitierungen bei der semantischen Suche, anfängliche Implementierungsineffizienzen sowie periodische Unterbrechungen des Systems betrachtet. Ziel ist es, die gemessenen Verläufe einzuordnen und deren Ursachen transparent darzustellen.

6.11.1 Limitierungen bei der semantischen Suche

Im praktischen Betrieb zeigte sich, dass die Indexierung und Speicherung der Embeddings auch bei großen Datenmengen stabil funktionierte. Ab einer

Größenordnung von etwa 500 000 Publikationen traten die wesentlichen Einschränkungen jedoch bei der semantischen Abfrage (Querying) auf. Die Antwortzeiten stiegen deutlich an und in einzelnen Fällen kam es zu Instabilitäten bis hin zu Abstürzen der Vektordatenbank Qdrant.

Die Ursache lag primär in den begrenzten verfügbaren Arbeitsspeicherressourcen der Serverumgebung. Da der Server parallel von mehreren Personen genutzt wurde, musste der verfügbare RAM für Qdrant stark limitiert werden. Unter diesen Bedingungen kann die effiziente Ausführung von ANN-Abfragen – insbesondere HNSW-basierter Suchverfahren – beeinträchtigt werden, da Indexstrukturen und Caches nicht vollständig im Speicher gehalten werden können. Dies führt zu vermehrten I/O-Zugriffen sowie deutlichen Latenzspitzen, wodurch sich die Query-Zeiten im Extremfall stark verlängern.

6.11.2 Anfängliche Implementierungsineffizienzen und Optimierungen

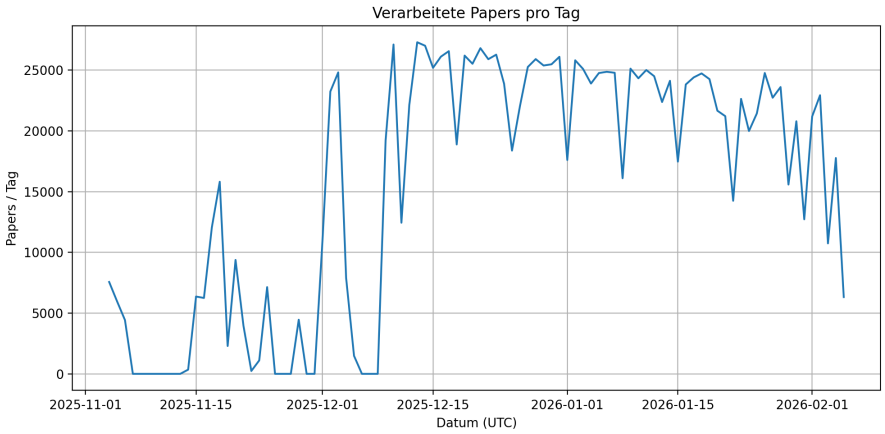

Abbildung 6.1: Anzahl der pro Tag verarbeiteten wissenschaftlichen Publikationen im Testbetrieb

Abbildung 6.1 zeigt die Anzahl der täglich verarbeiteten wissenschaftlichen Publikationen während des Testbetriebs. Zu Beginn des Beobachtungszeitraums

ist eine vergleichsweise geringe Verarbeitungsrate erkennbar. Diese anfängliche Phase ist primär auf noch ineffiziente Implementierungen innerhalb der Pipeline zurückzuführen.

Insbesondere betraf dies nicht optimal gewählte Batch-Größen, redundante Verarbeitungsschritte sowie fehlende Parallelisierung einzelner Komponenten. Im Verlauf der Entwicklung wurden diese Schwachstellen schrittweise identifiziert und durch gezielte Refaktorierungen behoben. Die anschließend steigende Tagesleistung verdeutlicht, dass Performanceverbesserungen vor allem durch Implementierungsoptimierungen erreicht wurden und die zugrunde liegende Architektur grundsätzlich skalierbar ausgelegt ist.

6.11.3 Periodische Verarbeitungseinbrüche durch geplante Unterbrechungen

In Abbildung 6.1 sind zudem wiederkehrende Einbrüche in der täglichen Verarbeitungsrate erkennbar. Diese treten in regelmäßigen Abständen auf und sind insbesondere an Donnerstagen zu beobachten. Die Ursache hierfür liegt nicht in technischen Instabilitäten oder Fehlern des Systems, sondern in geplanten Unterbrechungen des Pipeline-Betriebs.

Aufgrund der gemeinsamen Nutzung der Serverinfrastruktur musste die Pipeline an Donnerstagen für mehrere Stunden deaktiviert werden, um Rechenressourcen für andere Nutzer freizugeben. Während dieser Zeit fand keine Indexierung oder Embedding-Erzeugung statt, was sich direkt in der reduzierten Tagesverarbeitung widerspiegelt. Diese Einbrüche sind daher als organisatorische Randbedingung des Entwicklungsumfelds zu interpretieren.

6.11.4 Kumulative Verarbeitung und Gesamtstabilität des Systems

Abbildung 6.2 zeigt die kumulative Anzahl der verarbeiteten Publikationen über die Zeit. Der nahezu monotone und annähernd lineare Anstieg

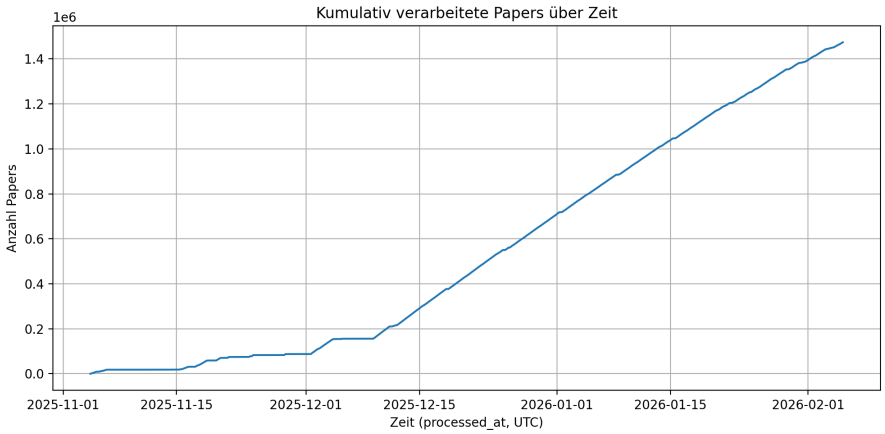

Abbildung 6.2: Kumulative Anzahl der verarbeiteten wissenschaftlichen Publikationen

bestätigt, dass der Import- und Indexierungsprozess zuverlässig und kontinuierlich durchgeführt werden konnte. Es traten keine systematischen Abbrüche oder Datenverluste auf.

Gleichzeitig verdeutlicht die Darstellung, dass die zunehmende Datenmenge keinen negativen Einfluss auf die Stabilität der Pipeline hatte. Die später beobachteten Performance-Probleme betrafen primär die semantische Abfrageleistung der Vektordatenbank unter stark begrenzten Arbeitsspeicherressourcen, nicht jedoch die Speicherung oder Indexierung selbst.

6.11.5 Zusammenfassende Einordnung

Zusammenfassend lassen sich die beobachteten Performance-Charakteristika auf drei Hauptfaktoren zurückführen: anfängliche Implementierungsineffizienzen, bewusst limitierte Hardware-Ressourcen sowie geplante Betriebsunterbrechungen. Keiner dieser Faktoren stellt eine grundsätzliche Einschränkung der Architektur von Tapyre Paper Search dar.

Vielmehr zeigen die Ergebnisse, dass das System auch unter realistischen und eingeschränkten Rahmenbedingungen stabil betrieben werden konnte. Die Analyse liefert wertvolle Erkenntnisse für zukünftige Optimierungen

und zeigt auf, welche Maßnahmen bei erweitertem Ressourcenrahmen oder in einer dedizierten Betriebsumgebung sinnvoll wären.

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*