Assignment

Anwendungsaspekte des Machine Learning

vorgelegt am 31. August 2025

Fakultät Wirtschaft und Gesundheit

Studiengang Wirtschaftsinformatik

Kurs WWI2022A

von

Anna Benndorf

Inhaltsverzeichnis

Abkürzungsverzeichnis

LLM	=	Large Language Model
Question-Answering-System	=	QA-System
Retrieval-Augmented Generation-System	=	RAG-System

Abbildungsverzeichnis

Tabellenverzeichnis

1 Theoretische Recherche und Einordnung

Im Folgenden werden die theoretischen Grundlagen zu Retrieval-Augmented Generation-Systemen (RAG-Systeme) und Agentensystemen beschrieben.

1.1 RAG-Systeme

Zunächst werden RAG-Systeme und die Architektur eines typischen RAG-Systems erläutert. Anschließend werden die Unterschiede zu reinen generativen Modellen sowie zu klassischen Question-Answering-Systemen (QA-Systemen) genannt. Darüber hinaus werden die Rolle der einzelnen Komponenten und die Herausforderungen bei der Umsetzung eines RAG-Systems erklärt.

Grundsätzlich beschreibt RAG einen Ansatz, bei dem Large Language Models (LLMs) nicht mehr nur auf ihr trainiertes Wissen zugreifen, sondern zusätzlich Informationen aus externen Quellen einbeziehen.[1] Im Gegensatz zu klassischen „Closed-Book"-Modellen, wie GPT-3.5 oder GPT-4 im Standardmodus, können RAG-Systeme also auf Dokumente, Datenbanken oder andere Wissensquellen zugreifen. Dadurch verbessern sich vor allem die Aktualität und Verlässlichkeit der generierten Antworten.

Durch das Abrufen passender Inhalte aus einer Wissensquelle und die anschließende Generierung der Antwort durch ein LLM wird die semantische Suche mit der Textgenerierung kombiniert.[2] Dies reduziert das Risiko von Halluzinationen und verbessert die Skalierbarkeit und Wartbarkeit von KI-Systemen.

1.1.1 Architektur eines typischen RAG-Systems

In diesem Abschnitt wird dir Architektur eines typischen RAG-Systems betrachtet. Ein solches RAG-System besteht aus mehreren aufeinanderfolgenden Schritten:[3]

1. Datenvorbereitung und Chunking: Die Ausgangsdaten (z. B. eine PDF) werden in kleinere Textabschnitte („Chunks") zerlegt, damit sie später leichter verarbeitet werden können.
2. Embedding und Speicherung: Die Chunks werden durch ein Embedding-Modell (z. B. mit text-embedding-3-small) in Vektoren umgewandelt und in einer Vektordatenbank abgelegt.

[1] Vgl. hierzu und im Folgenden Lewis et al. 2021, S. 1 ff.; vgl. dazu auch Gao et al. 2024, S. 1 ff.
[2] Vgl. hierzu und im Folgenden Gupta et al. 2024, S. 1 ff.; vgl. dazu auch IBM Research 2023
[3] Vgl. hierzu und im Folgenden Pederson 2025; vgl. dazu auch: LangChain Docs o. J.; Gao et al. 2024, S. 3

3. Retriever: Wenn eine Anfrage, ein Prompt, gestellt wird, sucht das System semantisch passende Chunks zur Anfrage heraus.
4. Antwortgenerierung: Das LLM erhält die gefundenen Inhalte, den Kontext, und die Anfrage und generiert auf Basis dessen die finale Antwort.

In der folgenden Abbildung 1 ist die Architektur eines solchen RAG-Systems zur Veranschaulichung dargestellt.

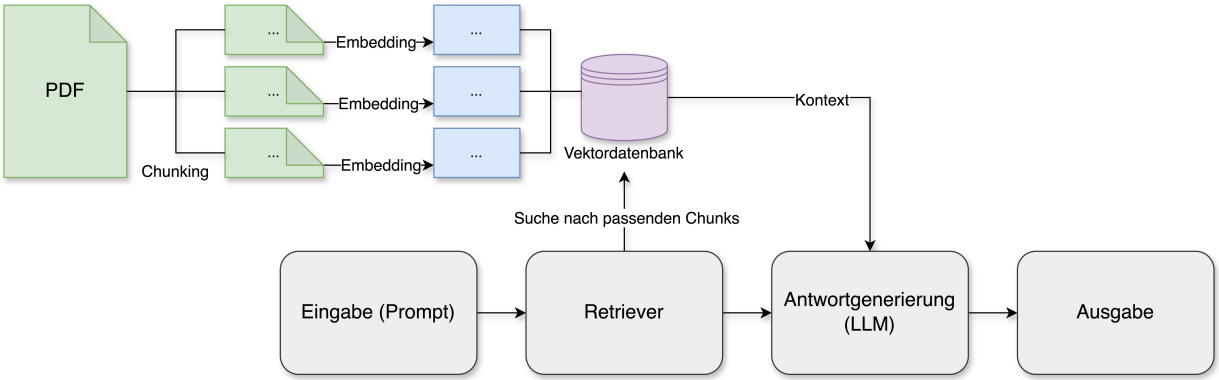

Abb. 1: Architektur eines typischen RAG-Systems

1.1.2 Unterschiede zu reinen generativen Modellen und klassischen QA-Systemen

Es gibt einige Unterschiede von RAG-Systemen zu reinen generativen Modellen und klassischen QA-Systemen. Generative Sprachmodelle ohne Retrieval-Mechanismus antworten ausschließlich auf Basis ihrer gelernten Trainingsdaten.[4] Sie können keine nach dem Trainingszeitpunkt veröffentlichten Informationen berücksichtigen. Zudem kann es vorkommen, dass generative Modelle bei Unsicherheiten plausible, aber falsche Antworten halluzinieren. Im Gegensatz dazu arbeiten klassische QA-Systeme meist regelbasiert, um passende Textstellen zu finden.[5] Ein RAG-System kombiniert beide Ansätze: Es sucht gezielt nach relevanten Inhalten und das LLM formuliert basierend darauf eine kontextualisierte Antwort.[6] Dies ist besonders bei komplexen Fragen oder heterogenen Datenquellen von Vorteil.

1.1.3 Rolle der einzelnen Komponenten eines RAG-Systems

Im folgenden Abschnitt werden die Rollen der einzelnen Komponenten eines RAG-Systems betrachtet. Dazu zählen Embeddings und Vektordatenbaken, Prompting und Kontextkonstruktion sowie Modellwahl und Kostenaspekte.

[4] Vgl. hierzu und im Folgenden Lewis et al. 2021, S. 1 f.; vgl. dazu auch: Gao et al. 2024, S. 1 f.
[5] Vgl. GeeksforGeeks 2025
[6] Vgl. hierzu und im Folgenden Lewis et al. 2021, S. 1 f.; vgl. dazu auch GeeksforGeeks 2025

Embeddings sind numerische Repräsentationen von Texten im Vektorraum.[7] Sie ermöglichen es, inhaltlich ähnliche Textabschnitte (Chunks) unabhängig von ihrer Wortwahl zu identifizieren. Dabei ist die Qualität des Embedding-Modells entscheidend, denn gute Modelle ordnen ähnliche Inhalte dicht beieinander an. Eine Vektordatenbank wie Qdrant oder Pinecone kann sehr viele solcher Chunks effizient durchsuchen.

In RAG-Systemen wird der abgerufene Kontext zusammen mit der Anfrage als erweiterter Prompt an das Sprachmodell übergeben, um eine kontextbasierte Antwort zu ermöglichen.[8] Dabei ist darauf zu achten, dass der Prompt nicht zu lang wird, damit das Tokenlimit des Modells nicht überschritten wird.[9]

Im Rahmen dieses Assignments ist die Nutzung von gpt-4o-mini und text-embedding-3-small vorgeschrieben, da beide kostengünstig, schnell und leistungsfähig sind.[10] Außerdem kann durch das Retrieval-Prinzip sogar ein kleineres Modell präzise Antworten liefern, sofern der Kontext gut gewählt ist.[11]

1.1.4 Typische Herausforderungen bei der Umsetzung

Da RAG-Systeme auch einige Herausforderungen mit sich bringen, werden in diesem Abschnitt typische Herausforderungen bei der Umsetzung eines RAG-Systems dargestellt.

Zunächst stellt die Größe des Kontextfensters eine Herausforderung dar, denn Sprachmodelle haben ein begrenztes Kontextfenster (z. B. 8.000 bis 32.000 Tokens).[12] Die abgerufenen Inhalte müssen deshalb gegebenenfalls vor dem Einfügen gefiltert, zusammengefasst oder priorisiert werden.[13] Außerdem treten Herausforderungen in Bezug auf die Latenz und die Performance des Systems auf. Durch den mehrstufigen Ablauf innerhalb des RAG-Systems entstehen zusätzliche Latenzen.[14] Abhilfe können Optimierungen wie Caching, asynchrone Verarbeitung oder Reranking schaffen, um die Performance zu verbessern.[15] Des Weiteren spielt die Retrieval-Qualität eine Rolle. Die Effektivität eines RAG-Systems hängt stark davon ab, wie gut die Embeddings und Chunkings gewählt sind.[16] Fehlerhafte Retrievals führen zu irrelevanten oder falschen Antworten. Selbst mit gutem Kontext kann das LLM halluzinieren oder

[7] Vgl. hierzu und im Folgenden Pederson 2025; vgl. dazu auch Qdrant o. J.
[8] Vgl. Khatib 2024
[9] Vgl. IBM Cloud Docs 2025
[10] Vgl. OpenAI API-Nutzung im Projekt – Leitblatt für Studierende
[11] Vgl. AWS o. J.
[12] Vgl. Bleiweiss 2024
[13] Vgl. Tribe AI 2025
[14] Vgl. milvus o. J. a
[15] Vgl. Gosh 2024
[16] Vgl. hierzu und im Folgenden Ma et al. 2024

wichtige Inhalte übersehen. Dies kann vor allem bei mehrdeutigen Fragen oder widersprüchlichen Quellen kann dies vorkommen.[17] Dementsprechend wirkt sich dies auf die Antwortgenauigkeit des Systems aus.

1.2 Agentensysteme

Im Folgenden wird sich mit Agentensystemen befasst. Es wird zunächst beschrieben, was ein Agent im Kontext von LLMs ist. Anschließend werden die zentralen Fähigkeiten von Agentensystemen, sowie typische Frameworks erläutert. Abschließend werden zwei dieser Frameworks anhand bestimmter Kriterien miteinander verglichen.

1.2.1 Was ist ein Agent im Kontext von LLMs?

Ein LLM-Agent ist ein KI-System, das mithilfe eines Sprachmodells nicht nur Inhalte generiert, sondern auch eigenständig entscheidet, welche Schritte zur Lösung einer Aufgabe erforderlich sind.[18] Im Gegensatz zu einer einfachen Prompt-Chain, besitzt ein Agent eine Entscheidungslogik.[19] Denn ein Agent analysiert Aufgaben, nutzt Werkzeuge (Tools), kann Zwischenschritte planen und mit einem Gedächtnis (Memory) arbeiten. Demnach können Agenten flexibel reagieren und ihren Ablauf dynamisch anpassen.[20]

1.2.2 Zentrale Fähigkeiten von Agentensystemen

Agentensystemen verfügen über verschiedene Fähigkeiten. In diesem Abschnitt werden die folgenden zentralen Fähigkeiten vorgestellt: Tool-Nutzung, Orchestrierung, Routing und Memory und Kontextverarbeitung.

Agenten können mit externen Systemen interagieren und beispielsweise APIs abfragen oder Dateien durchsuchen.[21] Dabei entscheidet der Agent selbst, welches Tool wann aufgerufen wird. Ein bekanntes Konzept dafür ist das ReAct-Framework, das Schlussfolgerung und Handeln kombiniert. Zudem sind Agenten in der Lage, mehrstufige Aufgaben zu planen und zu koordinieren.[22] Dazu definieren sie Teilschritte, prüfen Zwischenergebnisse und modellieren iterative Lösungswege. Dieser Prozess wird Orchestrierung genannt, kann sowohl synchron als auch asynchron erfolgen und ist meist Teil eines Multi-Agenten-Systems. In solchen Multi-

[17] Vgl. Barnett et al. 2024, S. 4; vgl. dazu auch Bommena 2025
[18] Vgl. Rajnerowicz 2025
[19] Vgl. hierzu und im Folgenden Chase 2024
[20] Vgl. Sahota 2023
[21] Vgl. hierzu und im Folgenden Sahota 2023; vgl. dazu auch: milvus o. J. b; Yao et al. 2023, S. 2 ff.
[22] Vgl. hierzu und im Folgenden Zhang et al. 2025, S. 3; vgl. dazu auch Anthropic 2025

Agenten-Systeme werden spezialisierte Subagenten eingesetzt, um Aufgaben zu lösen.[23] Ein zentraler Agent verteilt die Anfragen dabei an die jeweils passenden Subagenten. Dieses Prinzip ermöglicht skalierbare Systeme. Des Weiteren kann ein Agent Informationen aus früheren Interaktionen, zum Beispiel über Vektordatenbanken oder Zwischenspeicher, behalten.[24] Dies ist für konsistente Interaktionen, Planung und Kontextanpassungen notwendig.

1.2.3 Typische Frameworks

Als Nächstes werden drei typische Frameworks für Agentensysteme vorgestellt:

LangChain ist eines der bekanntesten Open-Source-Frameworks für den Aufbau von LLM-Agenten und eignet sich gut für Entwickler.[25] Es ermöglicht die Erstellung mehrschrittiger Workflows, bei denen Agenten Informationen abrufen, Kontext beibehalten und externe Tools einbinden können. LangChain bietet außerdem Module wie Chains, Memory, Tools und Agentenlogik an und ist daher ideal für maßgeschneiderte Agentenlösungen.

Microsoft AutoGen ist ebenfalls ein Open-Source-Framework, das mehrere spezialisierte KI-Agenten in einem Team koordiniert, um gemeinsam komplexe Aufgaben zu lösen.[26] Anstatt alles einem einzelnen Agenten zu überlassen, werden verschiedene Aufgaben auf mehrere Agenten aufgeteilt. Die Kommunikation zwischen den verschiedenen Agenten erfolgt über strukturierte Nachrichten, ähnlich wie in menschlichen Teams. Nutzer von AutoGen können das Framework flexibel konfigurieren und so maßgeschneiderte Multi-Agenten-Workflows erstellen.

CrewAI ist ein Framework zur Orchestrierung spezialisierter KI-Agententeams, die ähnlich wie menschliche Teams zusammenarbeiten.[27] Dabei erhält jeder Agent erhält eine Rolle, ein Ziel und eine Hintergrundgeschichte. Dadurch können die Agenten autonom Aufgaben delegieren, kommunizieren und sich an veränderte Kontexte anpassen.

1.2.4 Vergleich: LangChain Agents vs. CrewAI[28]

Im Folgenden werden die beiden Frameworks LangChain Agents und CrewAI anhand der folgenden Kriterien miteinander verglichen: Architektur, Modularität, typische Use Cases sowie technischer Einstieg und Lernkurve.

---

[23] Vgl. hierzu und im Folgenden Zhang et al. 2025, S. 2 ff.; vgl. dazu auch: Gu et al. 2025, S. 4 ff.; Ahmed 2025
[24] Vgl. hierzu und im Folgenden agix 2025
[25] Vgl. hierzu und im Folgenden Rajnerowicz 2025
[26] Vgl. hierzu und im Folgenden Jens.Marketing 2023
[27] Vgl. hierzu und im Folgenden Rajnerowicz 2025
[28] Vgl. hierzu und im Folgenden TechLatest.Net 2024; vgl. dazu auch: Tiwari 2024; CrewAI o. J.

LangChain Agents ist modular aufgebaut und umfasst Bestandteilen wie Chains, Retrievers und Memory. Im Gegensatz dazu ist CrewAI ein Framework mit rollenbasierten Agententeams, den sogenannten „Crews“ und strukturierten Aufgaben, den „Tasks“. Außerdem weist LangChain eine hohe Modularität auf, da die einzelnen Komponenten beliebig miteinander kombiniert werden können. Auch CrewAI ist modular aufgebaut, und zwar in Form von Crews, Tasks und Flows. Zudem werden Rollen und Aufgaben klar voneinander getrennt. Des Weiteren kann LangChain vielseitig eingesetzt werden. Typische Use Cases sind beispielsweise Dokumentenzusammenfassungen oder Chatbots. CrewAI wird dagegen besonders bei Multi-Agent-Szenarien, wie beispielsweise Marktforschung oder Problemlösungen verwendet. Für die Verwendung und den technischen Einstieg in LangChain sind bereits umfangreiche Dokumentationen und gute Tutorials vorhanden. Im Gegensatz dazu gibt es für die Verwendung von CrewAI weniger Dokumentationen, jedoch erste Tutorials. Da LangChain eine Vielzahl an Funktionen und Kombinationsmöglichkeiten anbietet, ist die Lernkurve im Vergleich zu CrewAI etwas steiler. Dies gilt jedoch nur, wenn der CrewAI-Nutzer ein gutes Verständnis von Rollenlogik und Aufgabendelegation hat.

Die folgende Tabelle 1 zeigt diesen Vergleich in einer übersichtlichen Darstellung.

Kriterium	**LangChain Agents**	**CrewAI**
Architektur	- Modular aufgebaut mit Bestandteilen wie Chains, Retrievers, Memory etc.	- Framework mit rollenbasierten Agententeams (Crews) und strukturierten Aufgaben (Tasks)
Modularität	- Hohe Modularität - Komponenten können beliebig kombiniert werden	- Modular in Form von Crews, Tasks und Flows - Klare Trennung zwischen Rollen und Aufgaben
Typische Use Cases	- Vielseitige LLM-Anwendungen wie Dokumentenzusammenfassung und Chatbots	- Multi-Agent-Szenarien wie Marktforschung und Problemlösung
Technischer Einstieg und Lernkurve	- Umfangreiche Dokumentation und gute Tutorials sind vorhanden - Steilere Lernkurve durch eine Vielzahl an Funktionen und Kombinationsmöglichkeiten	- Weniger Dokumentation, aber erste Tutorials vorhanden - Etwas flachere Lernkurve bei Verständnis von Rollenlogik und Aufgabendelegation

Tab. 1: Vergleich LangChain Agents vs. CrewAI

Zusammenfassend lässt sich sagen, dass sich LangChain Agents besonders für Entwickler eignet, die flexible und individuell anpassbare LLM-Agenten entwickeln möchten. CrewAI richtet sich an Nutzer, die strukturierte Multi-Agentensysteme mit klaren Rollen umsetzen wollen.

2 Implementierung eines RAG-Systems und eines Agentensystems

In diesem Kapitel werden die Implementierung eines KI-Systems mit RAG und die Implementierung eines Agentensystems erläutert.

2.1 RAG-System

Im Folgenden wird die Architektur des RAG-Systems beschrieben und dargestellt. Zudem wird die Modellkonfiguration des Systems beschrieben.

2.1.1 Dokumentation

Als Erstes wird das PDF-Dokument von GitHub heruntergeladen und lokal gespeichert. Der PyPDFLoader liest den Inhalt seitenweise aus und wandelt jede Seite in ein Document-Objekt um. Diese enthält sowohl den extrahierten Text als auch zugehörige Metadaten. Mit dem RecursiveCharacterTextSplitter werden die langen Seitentexte in Chunks zerlegt. Anschließend wird jeder Chunk unter Verwendung des OpenAI-Modells text-embedding-3-small in einen Vektor umgewandelt. Diese Vektoren werden dann im DocArrayInMemorySearch gespeichert, einer In-Memory Vektordatenbank. Der Retriever durchsucht diese Vektordatenbank nach den ähnlichsten Chunks. Wenn eine Benutzerfrage eingeht, werden mithilfe von ChatPromptTemplate die Benutzerfrage und die aus dem Retriever stammenden relevanten Chunks zu einem Prompt kombiniert. Dieser Prompt wird dann an das Sprachmodell ChatOpenAI mit der Modellkonfiguration gpt-4o-mini übergeben, welches auf Basis des Prompts eine Antwort generiert. Der StrOutputParser extrahiert aus der Modellantwort den Antwort-Text, welcher dann ausgegeben wird.

Die folgende Abbildung 2 zeigt eine graphische Veranschaulichung der Architektur.

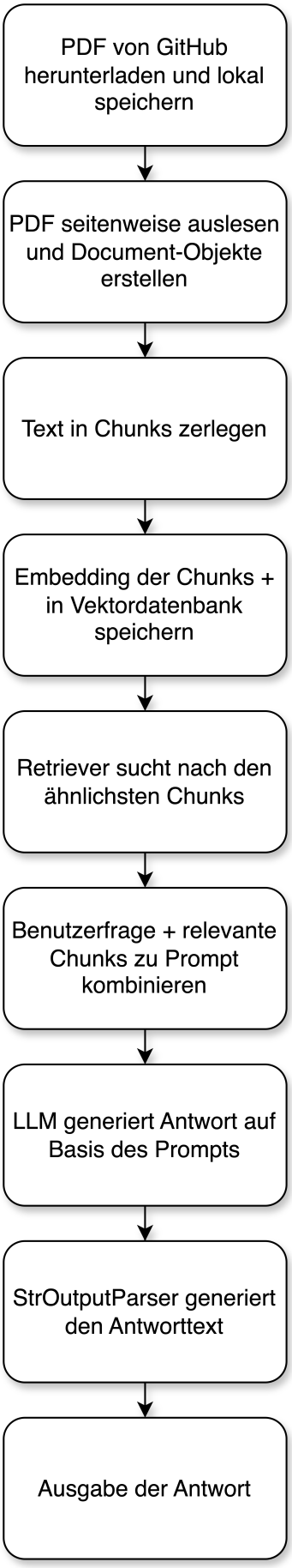

Abb. 2: Architektur des RAG-Systems

2.1.2 Modellkonfiguration

Es wird Sprachmodell gpt-4o-mini mit einer Temperatur von 0,2 verwendet, um eine geringe Varianz und damit präzisere Antworten zu erzielen. Die maximale Antwortlänge ist auf 200 Tokens begrenzt und die Ausgabe wird durch den StrOutputParser als reiner Text extrahiert. Für die Vektorisierung der Chunks wird das Embedding-Modell text-embedding-3-small verwendet. Die Chunkgröße ist auf 800 Tokens mit einer Überlappung von 120 Tokens begrenzt, um den Kontext nicht zu verlieren. Beim Retrieval kommt Maximal Marginal Relevance zum Einsatz. Es werden die vier relevantesten und zugleich vielfältigsten Chunks aus einer Vorauswahl von 20 Kandidaten (fetch_k = 20) in den Prompt integriert.

2.1.3 Code

Der Code für das RAG-System ist in Anhang 1 zu finden und auf GitHub unter folgendem Link: https://github.com/x8bean/Machine-Learning-Assignment

2.2 Code des Agentensystems

Der Code für das Agentensystem ist in Anhang 2 zu finden und auf GitHub unter folgendem Link: https://github.com/x8bean/Machine-Learning-Assignment

Anhang

Anhang 1: Code RAG-System

1) Pakete installieren und OpenAI-API-Key laden

```
# Pakete installieren

# Schnittstelle zu OpenAI-Modellen
!pip -q install langchain_openai

# Zusätzliche LangChain-Komponenten (Loader, Vectorstores)
!pip -q install langchain-community

# PDF-Parser
!pip -q install pypdf

# In-Memory-Vektorstore
!pip -q install docarray

# Tokeenizer für OpenAI-Modelle
!pip -q install tiktoken

# OpenAI-API-Key laden
from google.colab import userdata
OPENAI_API_KEY = userdata.get('apikey_ab')

# Bricht ab, falls kein API-Key vorhanden ist
assert OPENAI_API_KEY
```

2) PDF aus GitHub laden

```
# Bibliothek für HTTP-Anfragen (Dateien, APIs etc.)
import requests

# LangChain-Loader für PDFs
# Nutzt pypdf, um Seiten auszulesen und in Dokument-Objekte (Text + Metadaten) zu konvertieren
from langchain_community.document_loaders import PyPDFLoader

# GitHub Raw-Link zur PDF
url = "https://raw.githubusercontent.com/x8bean/Machine-Learning-Assignment/main/Wissensquelle.pdf"

# Download + Speichern als temporäre Datei in Colab
pdf_path = "/content/tmp.pdf"

# Öffnet die Zieldatei im Schreib-/Binärmodus ("wb"), lädt die PDF von der angegebenen URL herunter (mit 30 Sekunden Timeout) und schreibt den Inhalt direkt in diese Datei
with open(pdf_path, "wb") as f:
    f.write(requests.get(url, timeout=30).content)
```

```
# PDF in LangChain-Dokumente laden
loader = PyPDFLoader(pdf_path)
page_docs = loader.load()

# Überprüfung, ob die PDF die Vorgabe von ≤ 10 Seiten erfüllt
assert len(page_docs) <= 10, f"PDF hat {len(page_docs)} Seiten (>10)."

# Anzahl der Seiten wird ausgegeben
print("Seiten geladen:", len(page_docs))
```

3) Chunking

```
# Teilt Texte rekursiv anhand von Trennzeichen (Absatz, Satz, Wort) in Chunks
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Initialisiert den Textsplitter mit Regeln für Größe, Überlappung und Trennzeichen-Priorität
splitter = RecursiveCharacterTextSplitter(
    chunk_size=800, # Maximale Zeichen pro Chunk
    chunk_overlap=120, # Überlappung zwischen Chunks (verhindert Informationsverlust)
    separators=["\n\n", "\n", ".", " ", ""]
)

# Wendet das Chunking auf alle Seiten-Dokumente an, erzeugt Liste von kleine-
ren Document-Objekten
chunks = splitter.split_documents(page_docs)

# Anzahl der erzeugten Chunks wird ausgegeben
print("Chunks erstellt:", len(chunks))
```

4) Embedding

```
# Schnittstelle zu OpenAI-Embedding-API, um Text in Embeddings zu verwan-
deln, die den Sinn des Textes darstellen
from langchain_openai import OpenAIEmbeddings

# In-Memory-Vektorstore, speichert Embedding-Vektoren und führt Ähnlichkeitssuche durch
from langchain_community.vectorstores import DocArrayInMemorySearch

# Erstellt Embedding-Funktion, die bei Aufruf OpenAI-API für Vektorisierung nutzt
embeddings = OpenAIEmbeddings(
    model="text-embedding-3-small", # Modellauswahl für die Umwandlung von Text in Em-
beddings
    openai_api_key=OPENAI_API_KEY   # API-Schlüssel für die Authentifizierung bei OpenAI
)

# Wandelt alle Chunks in Embeddings um und speichert sie im Vektorstore
vectorstore = DocArrayInMemorySearch.from_documents(chunks, embedding=embeddings)

# Erzeugt Retriever, der für eine Suchanfrage die Top-4 relevantesten Chunks ausgibt
retriever = vectorstore.as_retriever(
    search_type="mmr", # Maximal Marginal Relevance: bedeutet, dass nicht nur die relevan-
testen Abschnitte ausgewählt werden, sondern auch möglichst unterschiedliche, um Wieder-
holungen zu vermeiden
```

```
    search_kwargs={"k": 4, "fetch_k": 20} # k=4: gibt die 4 besten Treffer zurück (werden später ins Prompt eingefügt); fetch_k=20: zieht zuerst die 20 besten Kandidaten, bevor daraus die 4 ausgewählt werden
)

5) Implementierung des RAG-Systems

# Schnittstelle zu OpenAI-Chat-LLMs
from langchain_openai import ChatOpenAI

# Generiert strukturierte Prompts aus Template-Texten mit Platzhaltern
from langchain.prompts import ChatPromptTemplate

# Wandelt LLM-Ausgabe in reinen Text-String um
from langchain_core.output_parsers import StrOutputParser

# Um mehrere Verarbeitungsschritte miteinander zu verbinden:
# RunnableParallel: führt mehrere Aufgaben gleichzeitig aus
# RunnablePassthrough: gibt die Eingabe unverändert weiter
from langchain_core.runnables import RunnableParallel, RunnablePassthrough

# Prompt-Template mit Platzhaltern für Kontext und Frage
template = """
Beantworte die Frage basierend auf dem Kontext.
Wenn Du die Frage nicht beantworten kannst, antworte "Ich weiß es nicht".

Context: {context}

Question: {question}
"""

# Erzeugt Prompt-Objekt, das bei Aufruf Platzhalter durch echte Werte ersetzt
prompt = ChatPromptTemplate.from_template(template)

# Initialisiert LLM mit Parametern
llm = ChatOpenAI(
    model="gpt-4o-mini", # Chat-Modellauswahl
    openai_api_key=OPENAI_API_KEY, # API-Schlüssel für die Authentifizierung bei OpenAI
    temperature=0.2, # Zufälligkeit der Ausgabe steuern
    max_tokens=200 # Antwortlänge begrenzen
)

# Erstellt Parser, der die reine Textausgabe extrahiert
parser = StrOutputParser()

# Kombiniert Retrieval und Frage in einer parallelen Struktur
setup = RunnableParallel(
    context=retriever, # Führt semantische Suche aus und liefert Chunks als Kontext
    question=RunnablePassthrough() # Leitet die Frage unverändert weiter
)

# Verknüpft alle Schritte: Retrieval --> Prompt --> LLM --> Ausgabe
chain = setup | prompt | llm | parser

6) Beispiel
```

```
# Beispielabfrage 1
print(chain.invoke("Wovon handelt das Dokument?"))

# Beispielabfrage 2
print(chain.invoke("Welche Unterschiede gab es bei den Zuschauerzahlen zwischen Männern und Frauen während der WM 2014?"))
```

Anhang 2: Code Agentensystem

Installation der Pakete

```
# Offizielles OpenAI-SDK, um über Python mit den OpenAI-APIs zu kommunizieren
!pip install openai

# HTTP-Client-Bibliothek für Python, um Web-APIs wie OpenWeatherMap abzufragen
!pip install requests

# Bibliothek für Datenvalidierung/-modellierung, die dafür sorgt dafür, dass Ausgaben in einem festen Schema (Structured Output) vorliegen
!pip install pydantic

# Hilfsbibliothek, um Listen/Tabellen als schön formatierte Texttabellen auszugeben
!pip install tabulate
```

Imports und Basis-Setup

```
# Openai-client-klasse importieren (für Chat-Completion Aufrufe)
from openai import OpenAI

# Pydantic-Basisklasse für strikt typisierte/validierte Datamodelle
from pydantic import BaseModel

# Sorgt für strukturierte Tabellen in der Konsole
from tabulate import tabulate

# Datum und Zeit für Zeitstempel
from datetime import datetime

# Lädt 'requests' für HTTP-Anfragen (z.B. API-Aufrufe) und 'json' zum Umwandeln zwischen JSON-Text und Python-Datenstrukturen
import requests, json

# OpenAI-Key laden
from google.colab import userdata
OPENAI_API_KEY = userdata.get('apikey_ab')

# OpenWeatherMap API-Key
from google.colab import userdata
OWM_API_KEY = userdata.get('owmkey')

# Standard-Stadt für die Abfrage
CITY = "Stuttgart"

# OpenAI-Client initialisieren
client = OpenAI(api_key=OPENAI_API_KEY)
```

Tool-Nutzung: Wetter-Tool

```
# Definiert eine Funktion, die für eine Stadt aktuelles Wetter von OpenWeatherMap abruft
def get_weather(city: str):
    # Sendet eine HTTP-GET-Anfrage an die OWM-API mit Stadtname, metrischen Einheiten, deutscher Sprache und API-Key
    resp = requests.get(
        f"http://api.openweathermap.org/data/2.5/weather?q={city}&units=metric&lang=de&appid={OWM_API_KEY}",
        timeout=15 # Beendet den Request automatisch, wenn nach 15 Sekunden keine Antwort kommt
    )

    # Wandelt die API-Antwort aus JSON in ein Python-Objekt (Dictionary) um
    data = resp.json()
    # Prüft, ob die Antwort fehlerfrei ist (HTTP-Code & API-Code)
    if resp.status_code != 200 or (data.get("cod") not in (200, None)):
        msg = data.get("message", f"HTTP {resp.status_code}") # Holt die Fehlermeldung aus der Antwort oder nutzt den HTTP-Statuscode
        raise RuntimeError(f"Fehler beim Abruf: {msg}") # Bricht die Funktion mit einer Fehlermeldung ab, wenn die Abfrage scheitert

    # Liest die aktuelle Temperatur in °C aus den Wetterdaten aus
    temp = float(data["main"]["temp"])

    # Liest die Kurzbeschreibung des aktuellen Wetters aus
    desc = data["weather"][0]["description"]

    # Erstellt ein Dictionary mit heutigem Datum und Temperatur
    point = {"date": datetime.now().strftime("%Y-%m-%d"), "temp_c": temp}

    # Gibt die Wetterdaten als strukturiertes Dictionary zurück
    return {"city": city, "temperature_c": temp, "description": desc, "points": [point]}


# Tool-Spezifikation für das LLM (Function Calling)
tools = [{ # Definiert eine Liste mit einem Tool, das das LLM aufrufen darf
    "type": "function", # Legt fest, dass es sich bei diesem Tool um eine Funktion handelt
    "function": {
        "name": "get_weather", # Name des Tools, der im LLM-Aufruf referenziert wird
        "description": "Get current weather for a city (temperature + short description).", # Beschreibung für das LLM, wofür das Tool gedacht ist
        "parameters": {  # Legt das JSON-Schema für die Parameter fest, die das LLM übergeben muss
            "type": "object", # Parameter werden als JSON-Objekt übergeben
            "properties": {"city": {"type": "string"}}, # Erlaubtes Feld: "city" als String
            "required": ["city"], # Das Feld "city" ist Pflicht
            "additionalProperties": False # Keine weiteren Felder außer den definierten sind erlaubt
        },
        "strict": True # Erzwingt strikte Einhaltung des Schemas beim Funktionsaufruf
    }
}]
```

Prompt-Chaining

```
# 1. Schritt: Erster LLM-Call entscheidet, ob/wie das Tool aufzurufen ist

# Erstellt die Nachrichtenliste, die als Gesprächskontext an das LLM geschickt wird
messages = [
    # Systemnachricht: definiert Rolle und Stil der Antworten
    {"role": "system", "content": "You are a concise weather assistant. Keep outputs short."},
    # Benutzerfrage mit der gewünschten Stadt (CITY)
    {"role": "user", "content": f"What's the current weather in {CITY}?"}
]

# Ruft das OpenAI-API auf, um eine Chat-Antwort vom Modell zu generieren
c1 = client.chat.completions.create(
    model="gpt-4o-mini", # Modellauswahl
    messages=messages, # Übergibt den Gesprächskontext an das Modell
    tools=tools, # Übergibt die Liste verfügbarer Tools, die das Modell aufrufen darf
    max_tokens=200 # Beschränkt die maximale Länge der Modellantwort
)

# 2. Schritt: Tool tatsächlich ausführen und Ergebnis zurück in den Chat geben

# Iteriert über alle vom Modell vorgeschlagenen Tool-Aufrufe
for tc in c1.choices[0].message.tool_calls or []:
    messages.append(c1.choices[0].message) # Fügt die Assistant-Nachricht mit dem Tool-
Aufruf zum Nachrichtenverlauf hinzu
    args = json.loads(tc.function.arguments) # Parst die vom Modell als JSON gesendeten Ar-
gumente in ein Python-Dictionary
    result = get_weather(**args)  # Führt das Wetter-Tool mit den Argumenten aus und spei-
chert das Ergebnis
    messages.append({"role": "tool", "tool_call_id": tc.id, "content": json.dumps(re-
sult)})  # Fügt das Tool-Ergebnis als 'tool'-Nachricht in den Nachrichtenverlauf ein

Structured Output

# Definiert ein Pydantic-Datenmodell für einen einzelnen Wetterdatenpunkt
class WeatherPoint(BaseModel):
    date: str                # Datum als Zeichenkette
    temp_c: float            # Temperatur in Grad Celsius als Fließkommazahl

# Definiert ein Pydantic-Datenmodell für den kompletten Wetterbericht
class WeatherReport(BaseModel):
    city: str                    # Name der abgefragten Stadt
    temperature_c: float         # Aktuelle Temperatur in Grad Celsius
    description: str             # Kurze Wetterbeschreibung
    points: list[WeatherPoint] = []  # Liste mit Wetterpunkten (hier nur ein Eintrag für heute)
    avg_temp_c: float            # Durchschnittstemperatur (kann vom Modell berechnet wer-
den)
    note: str                    # Kurze Empfehlung oder Bemerkung basierend auf der Tempera-
tur

# 3. Schritt: Zweiter LLM-Call: finale Antwort erzeugen und strikt ins Schema parsen

# Ruft das OpenAI-API auf und parsed die Antwort direkt ins WeatherReport-Schema
c2 = client.beta.chat.completions.parse(
    model="gpt-4o-mini",             # Modellwahl
    messages=messages,               # Gesamter Nachrichtenverlauf (inkl. Tool-Ergeb-
nis) als Kontext
```

```
    tools=tools,                    # Liste der verfügbaren Tools
    response_format=WeatherReport,      # Erwartetes Ausgabeformat: Instanz des Weather-
Report-Pydantic-Modells
    max_tokens=200                  # Maximale Länge der Antwort in Tokens begrenzen
)
final = c2.choices[0].message.parsed    # Extrahiert die geparste Modellantwort als Weat-
herReport-Objekt


# Ausgabe bestehend aus JSON und TABELLE

print("JSON Output:")                   # Überschrift für die JSON-Ausgabe
print(final.model_dump_json(indent=2))    # Gibt den WeatherReport formatiert als JSON-
String aus (mit Einrückung)

# Prüft, ob die Liste der Wetterpunkte nicht leer ist
if final.points:
    table = [["Datum", "Temp (°C)"]] + [[p.date, p.temp_c] for p in final.points]  # Baut eine Ta-
bellenstruktur aus den Wetterpunkten
    print("\nTabelle:")                                      # Überschrift für die Tabelle
    print(tabulate(table, headers="firstrow", tablefmt="github"))               # Gibt die Ta-
belle im Markdown-kompatiblen Format aus
```

Literaturverzeichnis

Agix 2025. *Memory-Augmented LLMS: How to build ChatGPT that remembers past conversations* https://agixtech.com/memory-augmented-llms-chatgpt/ (Zugriff vom 31.07.2025).

Ahmed 2025. *How to Design Multi-Agent LLM Systems for Complex Research Tasks effectively* https://medium.com/%40sahin.samia/how-to-design-multi-agent-llm-systems-for-complex-research-tasks-effectively-91da52a92ccc (Zugriff vom 31.07.2025).

Anthropic 2025. *How we built our multi-agent research system* https://www.anthropic.com/engineering/built-multi-agent-research-system (Zugriff vom 31.07.2025).

AWS o. J. *Was ist Retrieval-Augmented Generation (RAG)?* https://aws.amazon.com/what-is/retrieval-augmented-generation/ (Zugriff vom 30.07.2025).

Barnett, Scott; Kurniawan, Stefanus; Thudumu, Srikanth; Brannelly, Zach; Abdelrazek, Mohamed 2024. *Seven Failure Points When Engineering a Retrieval Augmented Generation System,* Geelong: o. Verl.

Bleiweiss, Amit 2024. *Tips for Building a RAG Pipeline with NVIDIA AI LangChain AI Endpoints* https://developer.nvidia.com/blog/tips-for-building-a-rag-pipeline-with-nvidia-ai-langchain-ai-endpoints/ (Zugriff vom 30.07.2025).

Bommena, Srinivas 2025. *We thought RAG solved Hallucinations.Turns Out, we just changed the problem* https://medium.com/@srinib100/we-thought-rag-solved-hallucinations-turns-out-we-just-changed-the-problem-d985137ee4f7 (Zugriff vom 30.07.2025).

Chase, Harrison 2024. *What is an AI agent?* https://blog.langchain.com/what-is-an-agent/ (Zugriff vom 31.07.2025).

CrewAI o. J. *What is CrewAI?* https://docs.crewai.com/en/introduction (Zugriff vom 31.07.2025).

Gao, Yunfan; Xiong, Yun; Gao, Xinyu; Jia, Kangxiana; Pan, Jinliu; Bi, Yuxi; Dai, Yi; Sun, Jiawei; Wang, Meng; Wang, Haofen 2024. *Retrieval-Augmented Generation for Large Language Models: A Survey*, o. O.: o. Verl.

GeeksforGeeks 2025. *RAG vs Traditional QA* https://www.geeksforgeeks.org/nlp/rag-vs-traditional-qa/ (Zugriff vom 30.07.2025).

Gosh, Bijit 2024. *Strategies for Optimal Performance of RAG* https://medium.com/@bijit211987/strategies-for-optimal-performance-of-rag-6faa1b79cd45 (Zugriff vom 30.07.2025).

Gu, Zhouhong; Zhu, Xiaxuan; Cai, Yin; Shen, Hao; Chen, Xingzhou; Wang, Qingyi; Li, Jialin; Shi, Xiaoron; Guo, Haoran; Huang, Wenxuan; Feng, Hongwei; Xiao, Yanghua; Ye, Zheyu; Hu, Yao; Cao, Shaosheng 2025. *AGENTCROUPCHAT-V2: Divide-and-Conquer Is What LLM-Based Multi-Agent System Need*, o. O.: o. Verl.

Gupta, Shailja; Ranjan, Rajesh; Narayan, Surya 2024. *A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions,* o. O.: o. Verl.

IBM Cloud Docs 2025. *Verfahren zur Überwindung von Längenbegrenzungen für Kontexte* https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/fm-context-length.html?context=wx (Zugriff vom 30.07.2025).

IBM Research 2023. *What is retrieval-augmented generation?* https://research.ibm.com/blog/retrieval-augmented-generation-RAG (Zugriff vom 30.07.2025).

Jens Marketing 2023. *AutoGen: Ein LLM-Framework von Microsoft* https://jens.marketing/autogen-microsoft/ (Zugriff vom 31.07.2025).

Khatib, Mayada 2024. *RAG: a simple practical example using llama index and HuggingFace* https://medium.com/@mayadakhatib/rag-a-simple-practical-example-using-llama-index-and-huggingface-fab3e5aa7442 (Zugriff vom 30.07.2025).

LangChain Docs o. J. *Retrieval augmented generation (RAG*) https://python.langchain.com/docs/concepts/rag/ (Zugriff vom 30.07.2025).

Lewis, Patrick; Perez, Ethan; Piktus, Aleksandra; Petroni, Fabio; Karpukhin, Vladimir; Goyal, Naman; Küttler; Heinrich; Lewis, Mike; Yih, Wen-tau; Rocktäschel, Tim; Riedel, Sebastian; Kiela, Douwe 2021. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, o. O.: o. Verl.

Ma, Tengyu; Gangasani, Vivek; Phan, Wen 2024. *RAG architecture with Voyage AI embedding models on Amazon SageMaker JumpStart and Anthropic Claude 3 models* https://aws.amazon.com/de/blogs/machine-learning/rag-architecture-with-voyage-ai-embedding-models-on-amazon-sagemaker-jumpstart-and-anthropic-claude-3-models/ (Zugriff vom 30.07.2025).

milvus o. J. a. *What are the individual components of latency in a RAG pipeline (e.g., time to embed the query, search the vector store, and generatet he answer), and how can each be optimized?* https://milvus.io/ai-quick-reference/what-are-the-individual-components-of-latency-in-a-rag-pipeline-eg-time-to-embed-the-query-search-the-vector-store-and-generate-the-answer-and-how-can-each-be-optimized (Zugriff vom 30.07.2025).

milvus o. J. b. *What ist he LangChain agent, and how does it work?* https://milvus.io/ai-quick-reference/what-is-the-langchain-agent-and-how-does-it-work (Zugriff vom 31.07.2025).

OpenAI API-Nutzung im Projekt – Leitblatt für Studierende

Pederson 2025. *Retrieval-Augmented Generation (RAG)* https://www.pinecone.io/learn/retrieval-augmented-generation/ (Zugriff vom 30.07.2025).

Qdrant o. J. *How does Vector Search Work in Qdrant?* https://qdrant.tech/documentation/overview/vector-search/?utm_source=chatgpt.com (Zugriff vom 30.07.2025).

Rajnerowicz, Casimir 2025. *What Are AI Agents and How to Use Them in 2025?* https://www.v7labs.com/blog/ai-agents-guide (Zugriff vom 31.07.2025).

Sahota, Harpreet 2023. *Implementing Agents in LangChain* https://www.comet.com/site/blog/implementing-agents-in-langchain/?utm_source=chatgpt.com (Zugriff vom 31.07.2025).

TechLatest.Net 2024. *Understanding the LangChain Framework* https://medium.com/%40techlatest.net/understanding-the-langchain-framework-8624e68fca32 (Zugriff vom 31.07.2025).

Tiwari, Pankaj 2024. Understanding CrewAI: A Deep Dive into Multi-Agent AI Systems https://medium.com/accredian/understanding-crewai-a-deep-dive-into-multi-agent-ai-systems-110d04703454 (Zugriff vom 31.07.2025).

Tribe AI 2025. *Reducing Latency and Cost at Scale: How Leading Enterprises Optimize LLM Performance* https://www.tribe.ai/applied-ai/reducing-latency-and-cost-at-scale-llm-performance (Zugriff vom 30.07.2025).

Yao, Shunyu; Zhao, Jeffrey; Yu, Dian; Du, Nan; Shafran, Izhak; Narasimhan, Karthik; Cao, Yuan 2023. „ReAct: Synergizing Reasoning And Acting In Language Models“, in *ICLR.*

Zhang, Wentao; Cui, Ce; Zhao, Yilei; Hu, Rui; Liu, Yang; Zhou, Yahui; An, Bo 2025. *AgentOrchestra: A Hierarchical Multi-Agent Framework for General-Purpose Task Solving*, o. O.: o. Verl.

Erklärung

Ich versichere hiermit, dass ich das vorliegende Assignment im Modul „Anwendungsaspekte des Machine Learning“ selbstständig verfasst habe.

Kornwestheim, 12.08.2025 A. Benndorf

(Ort, Datum) (Unterschrift)