Assignment zu Anwendungsaspekte des Machine Learning – Retrieval-Augmented Generation (RAG) und Agentensysteme

vorgelegt am 31.08.2025

Fakultät Wirtschaft und Gesundheit

Studiengang Wirtschaftsinformatik

Kurs WWI2022B

von

Felix Neu

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Theoretische Recherche & Einordnung

1.1 RAG-Systeme

1.1.1 Einleitung und Motivation

Große Sprachmodelle (LLMs) wie GPT-3 oder GPT-4 haben in den letzten Jahren enorme Fortschritte gezeigt. Dennoch sind sie durch zwei wesentliche Einschränkungen gekennzeichnet: ihr Wissensstand ist auf das Training eingefroren, und sie neigen zu sogenannten Halluzinationen, also plausibel klingenden, aber falschen Inhalten[1]. Klassische Frage-Antwort-Systeme (QA-Systeme) setzten dagegen auf reines Information Retrieval, lieferten jedoch starre Textpassagen und keine generativen, kontextsensitiven Antworten[2].

Retrieval-Augmented Generation (RAG) verbindet beide Ansätze, indem es Nutzereingaben über semantisches Retrieval mit relevanten Dokumentpassagen anreichert und diese in die Prompting-Pipeline eines LLM einspeist. So entstehen faktenbasierte, aktuelle und zugleich kohärente Antworten[3].

Die Motivation für RAG liegt insbesondere in drei Aspekten:

- **Reduktion von Halluzinationen** durch explizite Wissensquellen,
- **Aktualität** durch Erweiterbarkeit der Datenbasis ohne erneutes Training,
- **Nachvollziehbarkeit** durch Verweise auf verwendete Dokumente und Quellen.

Diese Eigenschaften machen RAG besonders geeignet für wissensintensive Domänen wie Recht, Medizin oder Unternehmenswissen[4].

1.1.2 Architektur eines RAG-Systems

Die Architektur eines typischen Retrieval-Augmented Generation (RAG)-Systems folgt einer klaren Pipeline (siehe Abbildung 1). Zunächst wird die Eingabe eines Nutzers in kleinere Textsegmente zerlegt (*Chunking*), um auch längere Dokumente für die Weiterverarbeitung handhabbar zu machen. Dieser Schritt ist entscheidend, da zu große Segmente häufig zu irrelevanten Retrieval-Ergebnissen führen, während zu kleine Segmente den Kontext fragmentieren[5]. Die Segmente werden anschließend in semantische Vektoren transformiert (*Embeddings*), die die inhaltliche Bedeutung des Textes abbilden, anstatt nur auf exakten Wortübereinstimmungen zu basieren. Diese

[1] Vgl. Lewis et al. 2021
[2] Vgl. Şakar, Emekci 2025
[3] Vgl. Lewis et al. 2021
[4] Vgl. Şakar, Emekci 2025
[5] Vgl. Schwaber-Cohen, Patel 2025

Vektorrepräsentationen werden in einer Vektordatenbank gespeichert, die auf Ähnlichkeitssuche optimiert ist und damit die Grundlage für das Retrieval bildet.

Bei einer Anfrage werden die inhaltlich ähnlichsten Segmente über semantisches Retrieval (*Top-k*) identifiziert. Optional können diese Ergebnisse über Re-Ranking oder Kontextkompression weiter gefiltert werden, um die Antwortqualität zu erhöhen und das Tokenbudget effizient zu nutzen[6]. Die so gewonnenen Passagen werden zusammen mit der ursprünglichen Nutzerfrage in den Prompt des Sprachmodells integriert. Das Large Language Model (LLM) generiert schließlich eine Antwort, die auf der Kombination von Eingabe und den abgerufenen Kontextinformationen basiert und somit sowohl kohärent als auch faktenbasiert ist[7].

Abb. 1: Schematische Architektur eines RAG-Systems

1.1.3 Kernkomponenten und Herausforderungen von RAG

Retrieval-Augmented Generation (RAG) kombiniert Informationsabruf mit generativer Modellierung. Im Zentrum stehen drei technische Bausteine – Embeddings mit Vektordatenbanken, Prompting zur Kontextintegration sowie die Wahl des Sprachmodells –, die zugleich spezifische Herausforderungen mit sich bringen.

Embeddings & Vektordatenbanken. Texte werden in hochdimensionale Vektoren transformiert, die semantische Ähnlichkeiten abbilden. Für die Speicherung und effiziente Abfrage dienen Systeme wie FAISS, Pinecone oder Weaviate. FAISS etwa nutzt GPU-Beschleunigung, um Ähnlichkeitssuchen im Milliardenbereich zu ermöglichen[8]. Die Segmentierung der Daten (Chunking) ist dabei kritisch: Kleine Chunks verbessern die Präzision, größere enthalten mehr Kontext, können aber Rauschen verstärken[9].

Prompting & Kontextkonstruktion. Die abgerufenen Dokumente werden in den Prompt integriert, typischerweise über Retrieval-Templates oder Re-Ranking[10]. Ziel ist, relevanten Kontext präzise und kompakt einzubinden, um konsistente und überprüfbare Antworten zu erzeugen.

Modellwahl & Kosten. Die Wahl des Sprachmodells bestimmt sowohl Qualität als auch Wirtschaftlichkeit. Proprietäre Modelle wie GPT-4o liefern hohe Performance und Multimodalität[11],

[6] Vgl. Şakar, Emekci 2025
[7] Vgl. Lewis et al. 2021
[8] Johnson, Douze, Jégou 2021
[9] Vgl. Schwaber-Cohen, Patel 2025
[10] Vgl. Lewis et al. 2021
[11] OpenAI 2025

sind jedoch mit API-Kosten verbunden. Open-Source-Modelle wie LLaMA 3 bieten wachsende Leistungsfähigkeit bei geringeren Betriebskosten und höherer Flexibilität[12].

Herausforderungen. RAG-Systeme stehen vor drei Hauptproblemen:

- *Tokenlimit:* Das Kontextfenster beschränkt die Menge an Dokumenten, die gleichzeitig berücksichtigt werden können.
- *Qualität des Retrievals:* Unpräzise Embeddings oder Rankings wirken sich unmittelbar negativ auf die Antwortqualität aus.
- *Latenz & Kosten:* Die Kombination aus Vektorsuche und Modellinferenz verursacht sowohl höhere Antwortzeiten als auch Betriebskosten.

1.2 Agentensysteme

1.2.1 Was ist ein Agent?

Im Kontext von LLM-basierten Systemen bezeichnet der Begriff *Agent* eine Architektur, die ein Sprachmodell um Entscheidungslogik erweitert und es befähigt, in einer interaktiven Umgebung zu handeln. Anders als klassische Prompt-Chains, die deterministische Abfolgen von Eingaben und Ausgaben definieren, können Agenten auf den Nutzerkontext reagieren und ihr Verhalten dynamisch anpassen[13].

Park et al. beschreiben Agenten als simulierte Entitäten, die Wahrnehmungen verarbeiten, Erinnerungen speichern und basierend darauf Handlungen planen[14]. Diese konzeptionelle Sichtweise verdeutlicht, dass Agenten nicht nur einfache Textverarbeitungswerkzeuge sind, sondern eigenständige softwarebasierte Akteure, die innerhalb einer Umgebung Entscheidungen treffen.

In der Praxis unterscheidet sich ein Agent somit von einer Prompt-Chain vor allem durch seine Flexibilität: Während Chains eine starre Sequenz vorgegebener Schritte ausführen, können Agenten in Abhängigkeit vom Kontext verschiedene Strategien wählen und gegebenenfalls externe Ressourcen einbeziehen[15]. Damit bilden sie die Grundlage für Systeme, die über reine Textgenerierung hinausgehen und komplexere Aufgaben übernehmen können.

[12] Grattafiori et al. 2024
[13] Vgl. Wu et al. 2023
[14] Park et al. 2023
[15] Vgl. Weng 2023

1.2.2 Kernfähigkeiten von Agentensystemen

Während im vorherigen Abschnitt die grundlegende Definition und Abgrenzung von Agenten erläutert wurde, steht hier die Frage im Vordergrund, durch welche Fähigkeiten sich Agentensysteme konkret auszeichnen. Dazu gehören insbesondere der gezielte Einsatz externer Werkzeuge, die Koordination mehrstufiger Aufgaben, die Verteilung von Teilaufgaben an spezialisierte Subagenten sowie Mechanismen des Speicherns und Planens.

Tool-Nutzung. Eine der wichtigsten Erweiterungen gegenüber klassischen Prompt-Chains ist die Fähigkeit von Agenten, externe Tools einzusetzen. Darunter fallen API-Aufrufe, der Zugriff auf Datenbanken, das Ausführen von Code oder auch der Umgang mit Dateien. Diese Integration erweitert die Handlungsmöglichkeiten des Modells erheblich, da Informationen oder Funktionen genutzt werden können, die nicht in den Modellparametern enthalten sind[16]. Ohne Tool-Nutzung wären Agenten weitgehend auf textuelle Transformationen beschränkt; mit ihr können sie jedoch Aufgaben wie Rechnen, aktuelle Informationsabfragen oder das Bearbeiten von strukturierten Daten übernehmen.

Orchestrierung. Agenten sind in der Lage, komplexe Aufgaben in einzelne Schritte zu zerlegen und diese zu koordinieren. Damit können auch Aufgaben bearbeitet werden, die nicht linear, sondern aus mehreren abhängigen Teilschritten bestehen. Wu et al. zeigen, dass insbesondere in Multi-Agent-Szenarien Konversationen zwischen Agenten als Mechanismus dienen können, um Aufgaben zu planen und koordiniert zu bearbeiten[17]. Dies eröffnet die Möglichkeit, Arbeitsabläufe dynamisch zu strukturieren und Entscheidungen situativ anzupassen, statt vorab starre Prozessketten zu definieren.

Routing. Ein weiteres Merkmal ist die Verteilung von Aufgaben an spezialisierte Subagenten. Routing-Mechanismen entscheiden, welcher Teil des Gesamtsystems eine Anfrage bearbeitet. Dadurch lassen sich Agenten mit unterschiedlichen Rollen oder Stärken kombinieren, etwa für Recherche, Datenverarbeitung oder Interaktion mit dem Nutzer. Dieser Ansatz ist besonders relevant in komplexen Umgebungen, in denen unterschiedliche Fähigkeiten parallel benötigt werden[18].

Memory und Planning. Schließlich benötigen Agenten die Fähigkeit, vergangene Interaktionen zu speichern und daraus künftige Handlungen abzuleiten. Park et al. beschreiben, dass sowohl Kurzzeitspeicher (z. B. Konversationskontext) als auch Langzeitspeicher (persistente Wissensrepräsentationen) entscheidend für glaubwürdige und konsistente Agenten sind[19]. Weng ergänzt, dass Planning auf der Zerlegung von Zielen in Teilaufgaben basiert und durch Selbstreflexion iterativ verbessert werden kann[20]. In Kombination erlauben Memory und Planning den Aufbau von Systemen, die über mehrere Interaktionen hinweg kohärent bleiben und komplexe Vorhaben strategisch verfolgen.

[16] Vgl. Weng 2023
[17] Wu et al. 2023
[18] Vgl. Wu et al. 2023
[19] Park et al. 2023
[20] Weng 2023

Die genannten Fähigkeiten bilden die Grundlage für den praktischen Einsatz von Agentensystemen. In der folgenden Sektion wird dargestellt, wie diese Konzepte in konkreten Frameworks umgesetzt werden.

1.2.3 Frameworks für Agentensysteme

Die zuvor beschriebenen Fähigkeiten von Agenten lassen sich nur dann praktisch umsetzen, wenn eine geeignete technische Infrastruktur vorhanden ist. Frameworks spielen hierbei eine entscheidende Rolle, da sie die Interaktion zwischen Sprachmodellen, Entscheidungslogik und externen Tools kapseln und dadurch wiederverwendbare Strukturen bereitstellen. Besonders relevant sind derzeit *LangChain Agents*, die sich als Standard in der Community etabliert haben, sowie *CrewAI*, ein Framework, das speziell auf die Zusammenarbeit mehrerer Agenten in Teams ausgerichtet ist. Beide verfolgen unterschiedliche Konzepte und adressieren damit verschiedene Anwendungsfelder.

Kriterium	**LangChain Agents**	**CrewAI**
Architektur	Zentralisierte Steuerung: Agents bauen auf Chains und Tools auf[21].	Rollenbasierte Teams von Agenten, die über definierte Rollen und Interaktionen zusammenarbeiten[22].
Modularität	Sehr granular, viele vordefinierte Komponenten (Chains, Tools, Memory), allerdings mitunter komplex in der Anwendung[23].	Optimiert für Kollaboration, weniger Einzelbausteine, klare Rollen- und Aufgabenverteilung[24].
Use Cases	Typisch: Chatbots, RAG-Systeme, Automatisierung, Data Analysis[25].	Typisch: Multi-Agent-Kollaboration wie Research-Teams, Content-Generierung oder Projektbearbeitung[26].
Einstiegshürden	Große Community, umfangreiche Dokumentation, aber hohe Lernkurve durch API-Komplexität[27].	Leichter Einstieg mit CLI-Tools und Beispielskripten, jedoch kleinere Community und begrenzte Ressourcen[28].

Tab. 1: Vergleich von LangChain Agents und CrewAI anhand zentraler Kriterien

Der Vergleich zeigt, dass *LangChain Agents* durch ihre hohe Modularität und Flexibilität überzeugen, jedoch eine komplexe API-Struktur mit entsprechender Lernkurve aufweisen[29]. *CrewAI*

[21] Vgl. LangChain, Inc. 2025; Chase 2024.
[22] Vgl. CrewAI 2025.
[23] Vgl. Chase 2024.
[24] Vgl. Kipkemboi 2025.
[25] Vgl. ZealousSystemPvtLtd 2025.
[26] Vgl. Moura 2025.
[27] Vgl. LangChain, Inc. 2025.
[28] Vgl. Kipkemboi 2025.
[29] Vgl. LangChain, Inc. 2025; Chase 2024

setzt dagegen auf kollaborative Rollenmodelle, wodurch die Einstiegshürde sinkt und Multi-Agent-Workflows schnell umgesetzt werden können[30].

Während LangChain typischerweise für Chatbots, RAG-Systeme und Datenanalysen genutzt wird[31], adressiert CrewAI vor allem szenariengetriebene Anwendungen wie Research- oder Content-Crews[32]. Damit ergänzen sich beide Frameworks: LangChain bietet maximale Anpassbarkeit, CrewAI einen pragmatischen Zugang zu kollaborativen Agentensystemen.

1.2.4 Fazit

Zusammenfassend lässt sich festhalten, dass Agentensysteme einen entscheidenden Schritt über klassische Prompt-Chains hinaus darstellen. Während Prompt-Chains starre Abfolgen definieren, ermöglichen Agenten durch Entscheidungslogik, Tool-Nutzung und Memory adaptive und kontextsensitive Interaktionen[33]. Ihre zentralen Fähigkeiten – vom Einsatz externer APIs über die Orchestrierung mehrstufiger Prozesse bis hin zum Routing und langfristigen Gedächtnis – eröffnen neue Möglichkeiten für komplexe Aufgabenstellungen[34].

Der Vergleich der Frameworks zeigt unterschiedliche Schwerpunktsetzungen: *LangChain Agents* bieten ein hochgradig modulares Toolkit, das sich besonders für anspruchsvolle, flexible Anwendungen eignet, jedoch eine hohe Einarbeitung erfordert[35]. *CrewAI* verfolgt dagegen einen kollaborativen Ansatz mit rollenbasierten Multi-Agent-Teams, wodurch Einstieg und Umsetzung deutlich einfacher werden, auch wenn dies mit geringerer Granularität einhergeht[36].

Insgesamt wird deutlich, dass Agentensysteme ein zentrales Konzept moderner LLM-Anwendungen darstellen. Die Wahl des Frameworks hängt stark von den Projektzielen ab: maximale Flexibilität und Integration sprechen für LangChain, während schnelle Kollaboration und pragmatische Umsetzungen für CrewAI vorteilhaft sind. Damit bilden Agentensysteme nicht nur ein theoretisches Konstrukt, sondern eine praxisrelevante Grundlage für die nächste Generation KI-gestützter Anwendungen.

[30] Vgl. CrewAI 2025; Kipkemboi 2025
[31] Vgl. ZealousSystemPvtLtd 2025
[32] Vgl. Moura 2025
[33] Vgl. Weng 2023; Park et al. 2023
[34] Vgl. Wu et al. 2023
[35] Vgl. LangChain, Inc. 2025
[36] Vgl. CrewAI 2025; Kipkemboi 2025

2 Implementierung eines KI-Systems mit RAG- und Agentenkomponenten

Dieses Kapitel dokumentiert die Architektur des Assignments, welches im Repository unter https://github.com/fe-neu/FN-AGENTS-PACKAGE zu finden ist. Beispiele und Demos sind ebenfalls dort enthalten.

Die Aufgabe, ein KI-System zu entwickeln, wurde ohne die Nutzung spezieller Frameworks wie in Sektion 1.2 beschrieben umgesetzt. Der Grund hierfür war das Ziel, ein tieferes Verständnis für die zugrunde liegenden Technologien zu erlangen. Dementsprechend wurden verschiedene Funktionen, die Frameworks üblicherweise bereitstellen, eigenständig implementiert.

Die im Kern verwendeten Python-Packages waren: `openai`, `pypdf`, `numpy` sowie einige kleinere Utility-Packages.

2.1 RAG-System

2.1.1 Allgemeine Implementierung

Da zum einen keine Frameworks für RAG-Systeme verwendet wurden und zum anderen das Kernprinzip von RAG sowohl für das eigentliche RAG-System als auch für das Langzeitgedächtnis eingesetzt werden sollte, wurden im ersten Schritt einige grundlegende Klassen implementiert, um Basisfunktionen bereitzustellen. Zu finden sind diese in `fn_package.retrieval.core`

`VectorStore` ist eine eigens implementierte In-Memory-Vektordatenbank, die `numpy`-Matrizen verwendet, um Embeddings von Inhalten zu speichern.

`BaseRecord` ist eine abstrakte Basisklasse, deren Instanzen sich im `VectorStore` ablegen lassen. Sowohl das RAG-System als auch das Langzeit-Memory-System besitzen eigene Implementierungen dieser Klasse mit spezifischen, für ihre jeweilige Anwendung relevanten Features.

`Retriever` ist die Klasse, die Informationen letztendlich aus dem `VectorStore` abruft. Dabei wird die Cosine Similarity genutzt. Implementiert wurde sowohl das Retrieval mit dem Parameter `k` als auch die Möglichkeit, alle Dokumente oberhalb eines Ähnlichkeitsschwellwerts zurückzugeben.

`Embedder` ist im Wesentlichen ein Wrapper für die OpenAI-Embeddings-API und erzeugt die Embeddings, die im `VectorStore` gespeichert werden sollen. Das verwendete Embedding-Modell sowie die dazugehörigen Dimensionen können innerhalb der OpenAI-Embedding-Familie über die `.env`-Datei angepasst werden. Standardmäßig wird `text-embedding-3-small` mit 1536 Dimensionen verwendet.

2.1.2 RAG-System

Das eigentliche RAG-System wurde in `fn_package.retrieval.rag` implementiert. Hier findet sich auch der `ChunkRecord`, der neben den Funktionen eines Records zusätzlich Referenzen auf den vorherigen sowie den nachgelagerten Chunk enthält. Außerdem wird gespeichert, aus welcher Datei ein jeweiliger Chunk stammt.

Der `Chunker` ist dafür verantwortlich, einzelne Strings in Chunks zu zerlegen. Dabei kommt das Paket `tiktoken` zum Einsatz, sodass das Chunking nicht zeichenbasiert, sondern tokenbasiert erfolgt. Standardmäßig wird eine Chunkgröße von 200 Tokens mit einem Overlap von 50 Tokens verwendet. Diese Werte lassen sich jedoch bei Bedarf leicht anpassen und sind in Ihrer aktuellen Form nicht zwingend optimal.

Der `Parser` ist lediglich dafür zuständig, PDF-Dokumente zu verarbeiten. Hierfür wurde das Paket `pypdf` verwendet.

Der `RagService` stellt letztendlich die API zur Interaktion mit dem RAG-System dar. In dieser Klasse ist auch die Pipeline implementiert, die von einem PDF bis hin zum betriebsbereiten RAG-System reicht (siehe Abbildung 2).

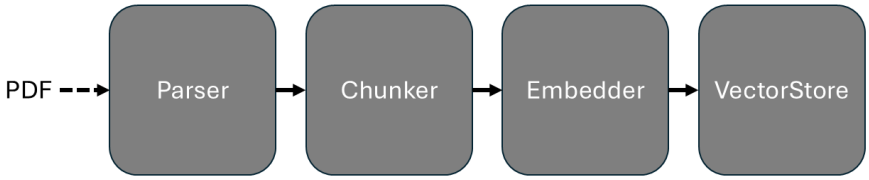

Abb. 2: Pipeline vom PDF zum betriebsbereiten System

Darüber hinaus ist hier auch die Pipeline zum Retrieven von Dokumenten implementiert (siehe Abbildung 3). Eine einfache Methode erstellt zudem direkt einen informativen String aus den einzelnen Record-Objekten, der alternativ zu den vollständigen Record-Objekten verwendet werden kann.

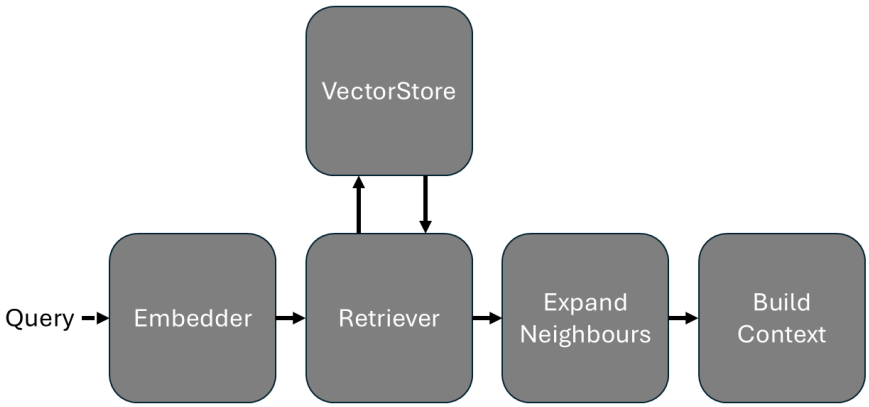

Abb. 3: Pipeline zum Retrieven von Records

2.1.3 Langzeitgedächtnis

Das Langzeitgedächtnis wurde implementiert, um Informationen über den Nutzer, wie zum Beispiel persönliche Daten, Interessen oder Vorlieben, zu speichern. Hierfür wurde eine eigene Datenklasse `MemoryRecord` entwickelt, die neben den Basisfunktionen zusätzlich den Zeitpunkt der Erstellung einer Erinnerung sowie deren Relevanz als Fließkommazahl zwischen 0 und 1 speichert.

Außerdem wurde ein `CSVStorage` implementiert, der Erinnerungen permanent als CSV-Dateien ablegt und diese beim Initialisieren erneut lädt, sodass Erinnerungen dauerhaft persistent bleiben.

Der zentrale Unterschied zwischen dem regulären RAG-System und dem Memory-System liegt in der Art, wie Records aus dem `VectorStore` abgerufen werden. Der `MemoryRetriever`, der vom herkömmlichen `Retriever` erbt, besitzt zusätzlich die Fähigkeit, die Similarity-Scores anhand des Alters einer Erinnerung sowie ihrer Relevanz zu modifizieren. Diese Multiplikatoren lassen sich zudem konfigurieren. Die genaue Formel lautet:

$$\text{Score}(h) \;=\; s_{\text{base}}(h) \cdot e^{-\lambda_{\text{rec}} \cdot \Delta t(h)} \cdot \big(1 + \lambda_{\text{imp}} \cdot \big(i(h) - 0.5\big)\big)$$

- h: ein Treffer (*Hit*), bestehend aus Embedding-Ähnlichkeit und Metadaten
- $s_{\text{base}}(h)$: ursprünglicher Cosine-Similarity-Score des Treffers
- $\Delta t(h)$: Alter der Erinnerung in **Tagen** seit ihrer Erstellung
- λ_{rec}: Gewichtung für die *Recency* (jüngere Erinnerungen sind wichtiger)
- $i(h)$: Importance-Wert des Treffers im Bereich $[0, 1]$

- λ_{imp}: Gewichtung für die *Importance*
- $e^{-\lambda_{\text{rec}} \cdot \Delta t(h)}$: Recency-Faktor, exponentieller Abfall der Relevanz mit zunehmendem Alter
- $1 + \lambda_{\text{imp}} \cdot (i(h) - 0.5)$: Importance-Faktor, Verschiebung des Scores abhängig von der Wichtigkeit

Im Beispiel-Chat werden Erinnerungen bei jeder Abfrage angefügt. Auch hier kann wieder mit den Parametern `k` und `threshold` optimiert werden.

2.2 Agents

2.2.1 Architektur

Das in diesem Projekt implementierte Agentensystem folgt dem Prinzip eines Multi-Agent-Systems. In der aktuellen Implementierung wurden Tool-Nutzung, Routing, Parallelisierung und Orchestrierung realisiert.

Abbildung 4 zeigt eine vereinfachte Darstellung der Architektur des Systems. Dieses ist in vier Schichten untergliedert:

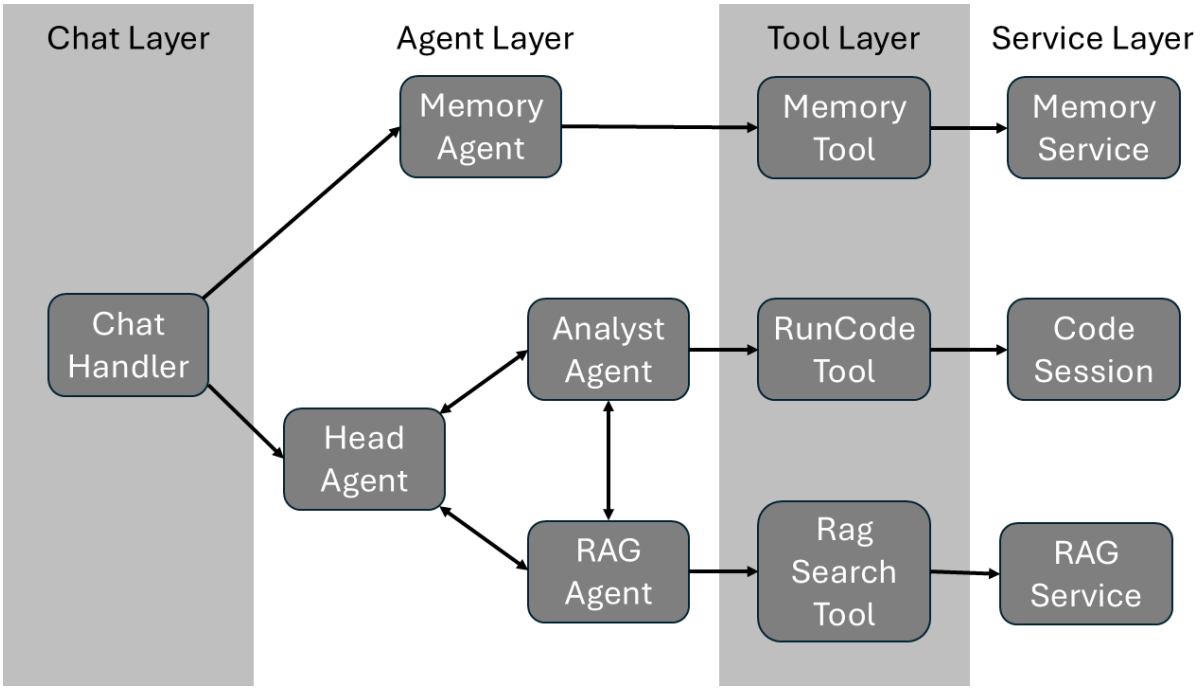

Abb. 4: Vereinfachte Architekturdarstellung des Multi-Agent-Systems

- **Chat Layer:** Die oberste Schicht, die für das Management der Konversation verantwortlich ist und direkt mit dem Nutzer interagiert.

- **Agent Layer:** Die Schicht, in der sich die Agents befinden und gegebenenfalls miteinander interagieren.
- **Tool Layer:** Tools, mit denen die Agents interagieren können (werden weiter unten genauer erläutert).
- **Service Layer:** Services, die in der Regel von Tools aufgerufen werden, um Tool Calls umzusetzen.

Die einzelnen Agents und deren Features werden in den folgenden Abschnitten detaillierter erläutert.

2.2.2 Agents

In diesem Projekt werden vier spezialisierte Agenten eingesetzt, die jeweils eine klar abgegrenzte Rolle einnehmen und über unterschiedliche Tools verfügen. Im Folgenden werden diese Agenten in der Reihenfolge HeadAgent, RAGAgent, AnalystAgent und MemoryAgent vorgestellt.

HeadAgent

Rolle: Der HeadAgent übernimmt die Rolle des Koordinators und Entscheidungsträgers. Er empfängt standardmäßig alle Nachrichten des Nutzers und entscheidet, ob Aufgaben selbst bearbeitet oder an spezialisierte Agenten weitergegeben werden. Ziel ist es, eine effektive Zusammenarbeit zwischen allen Agenten sicherzustellen, ohne in die Details ihrer Arbeit einzugreifen.

Tools:

- `HandOverTool` – Übergibt Nachrichten oder Aufgaben an andere Agenten oder den Nutzer.
- `ThinkTool` – Ermöglicht interne Überlegungen, die nicht nach außen kommuniziert werden.

RAGAgent

Rolle: Der RAGAgent ist auf die Informationsbeschaffung und -aufbereitung spezialisiert. Er durchsucht den Dokumentenspeicher, extrahiert relevante Inhalte und liefert faktenbasierte, objektive Antworten mit Quellenangaben.

Tools:

- `HandOverTool` – Gibt Ergebnisse oder Aufgaben weiter.
- `ThinkTool` – Dient zur internen Gedankenspeicherung.
- `RagSearchTool` – Führt Suchanfragen im Vektorspeicher aus und ruft relevante Dokumente ab.

AnalystAgent

Rolle: Der AnalystAgent ist der Datenanalysespezialist. Er verfügt über exklusiven Zugang zu einer persistenten Python-Ausführungsumgebung mit Zustandsbeibehaltung über mehrere Ausführungen hinweg. Damit analysiert er tabellarische Daten, führt Berechnungen durch und erstellt Visualisierungen. Er darf ausschließlich die Pakete `numpy`, `pandas` und `matplotlib` verwenden.

Tools:

- `HandOverTool` – Übergibt Ergebnisse oder Aufgaben an andere Agenten.
- `ThinkTool` – Speichert interne Überlegungen.
- `RunCodeTool` – Führt Python-Code in der Sitzung aus.
- `GetCodeHistoryTool` – Gibt den bisher ausgeführten Code zurück.
- `GetFileTreeTool` – Zeigt die verfügbaren Dateien im Arbeitsverzeichnis an.
- `ResetCodeSessionTool` – Setzt die Python-Umgebung zurück (Variablen werden gelöscht, Dateien bleiben erhalten).

MemoryAgent

Rolle: Der MemoryAgent ist ausschließlich für das Management des Langzeitgedächtnisses zuständig. Er entscheidet, ob neue Informationen gespeichert oder verworfen werden sollen. Dieser Agent arbeitet parallel zu den anderen Agenten um die tatsächliche Konversation nicht zu bremsen. Seine erstellten Langzeit Erinnnerungen werden aber allen Agenten zur Verfügung gestellt.

Tools:

- `CreateMemoryTool` – Legt einen neuen Memory-Eintrag im Memory Service ab.

Literaturverzeichnis

Chase, Harrison (2024): What is an AI agent? LangChain Blog. URL: https://blog.langchain.com/what-is-an-agent/ (Abruf: 22.08.2025).

CrewAI (2025): CrewAI Documentation. CrewAI. URL: https://docs.crewai.com/en/introduction (Abruf: 22.08.2025).

Grattafiori, Aaron et al. (2024): The Llama 3 Herd of Models. DOI: 10.48550/arXiv.2407.21783. arXiv: 2407.21783[cs]. URL: http://arxiv.org/abs/2407.21783 (Abruf: 21.08.2025).

Johnson, Jeff; Douze, Matthijs; Jégou, Hervé (2021): Billion-Scale Similarity Search with GPUs. In: *IEEE Transactions on Big Data* 7.3, S. 535–547. ISSN: 2332-7790. DOI: 10.1109/TBDATA.2019.2921572. URL: https://ieeexplore.ieee.org/document/8733051 (Abruf: 21.08.2025).

Kipkemboi, Tony (2025): Build your First CrewAI Agents. CrewAI. URL: https://blog.crewai.com/getting-started-with-crewai-build-your-first-crew/ (Abruf: 22.08.2025).

LangChain, Inc. (2025): LangChain Documentation. URL: https://python.langchain.com/docs/introduction/ (Abruf: 22.08.2025).

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. DOI: 10.48550/arXiv.2005.11401. arXiv: 2005.11401[cs]. URL: http://arxiv.org/abs/2005.11401 (Abruf: 21.08.2025).

Moura, João (2025): What Matters in AI Agents. CrewAI. URL: https://blog.crewai.com/what-matters-in-ai-agents/ (Abruf: 22.08.2025).

OpenAI (2025): GPT-4o Documentation. URL: https://platform.openai.com (Abruf: 21.08.2025).

Park, Joon Sung; O'Brien, Joseph C.; Cai, Carrie J.; Morris, Meredith Ringel; Liang, Percy; Bernstein, Michael S. (2023): Generative Agents: Interactive Simulacra of Human Behavior. DOI: 10.48550/arXiv.2304.03442. arXiv: 2304.03442[cs]. URL: http://arxiv.org/abs/2304.03442 (Abruf: 22.08.2025).

Şakar, Tolga; Emekci, Hakan (2025): Maximizing RAG efficiency: A comparative analysis of RAG methods. In: *Natural Language Processing* 31.1, S. 1–25. ISSN: 2977-0424. DOI: 10.1017/nlp.2024.53. URL: https://www.cambridge.org/core/product/identifier/S2977042424000530/type/journal_article (Abruf: 21.08.2025).

Schwaber-Cohen, Roie; Patel, Arjun (2025): Chunking Strategies for LLM Applications | Pinecone. URL: https://www.pinecone.io/learn/chunking-strategies/ (Abruf: 21.08.2025).

Weng, Lilian (2023): LLM Powered Autonomous Agents. Section: posts. URL: https://lilianweng.github.io/posts/2023-06-23-agent/ (Abruf: 22.08.2025).

Wu, Qingyun; Bansal, Gagan; Zhang, Jieyu; Wu, Yiran; Li, Beibin; Zhu, Erkang; Jiang, Li; Zhang, Xiaoyun; Zhang, Shaokun; Liu, Jiale; Awadallah, Ahmed Hassan;

White, Ryen W.; Burger, Doug; Wang, Chi (2023): AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. DOI: 10.48550/arXiv.2308.08155. arXiv: 2308.08155[cs]. URL: http://arxiv.org/abs/2308.08155 (Abruf: 22. 08. 2025).

ZealousSystemPvtLtd (2025): Top 5 Use Cases for AI Agents Built with LangChain. URL: https://www.zealousys.com/blog/use-cases-for-ai-agents-built-with-langchain/ (Abruf: 22. 08. 2025).

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Thema: *Assignment zu Anwendungsaspekte des Machine Learning – Retrieval-Augmented Generation (RAG) und Agentensysteme* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

(Ort, Datum) (Unterschrift)