Ein ChatBot für die OTH-AW mithilfe eines Multi-Agenten Systems

Christian Rute

16. Juli 2025

Die vorliegende Arbeit stellt eine hierarchische Multi-Agenten-Architektur vor, die darauf abzielt, die Effizienz und Leistungsfähigkeit von ChatBot-Systemen auf Basis großer Sprachmodelle (LLMs) zu steigern. Der Kern des Ansatzes besteht darin, einen monolithischen Ansatz durch ein modulares System aus spezialisierten Agenten zu ersetzen. Ein übergeordneter Main-Agent analysiert eingehende Nutzeranfragen und delegiert diese, unter Verwendung eines eigenen Toolsets, an einen oder mehrere untergeordnete, auf spezifische Domänen ausgerichtete Agenten. Jeder dieser spezialisierten Agenten verfügt über ein eigenes, auf seine Aufgabe zugeschnittenes Set an Tools, um die Anfrage präzise und effizient zu bearbeiten. Ein wesentliches Merkmal dieser Architektur ist die gezielte Verknüpfung des probabilistischen Verhaltens der Sprachmodelle mit der deterministischen Funktionalität der externen Tools.

Die Ergebnisse werden innerhalb des Systems zusammengeführt und an den Nutzer zurückgegeben. Diese modulare Struktur führt zu drei entscheidenden Vorteilen: Erstens wird die Größe des Kontextfensters für jeden einzelnen Agenten signifikant reduziert. Zweitens resultiert dies in einer erhöhten Verarbeitungsgeschwindigkeit des Gesamtsystems. Drittens ermöglicht der Ansatz den Einsatz kleinerer, ressourcenschonenderer Sprachmodelle, da deren primäre Funktion in der intelligenten Orchestrierung von Tools und der Aufbereitung der von ihnen gelieferten Daten liegt, anstatt komplexe Aufgaben vollständig autonom zu lösen. Das System demonstriert somit, exemplarisch validiert durch einen Prototypen im Hochschulkontext, einen Weg zu robusteren, schnelleren und skalierbareren LLM-Anwendungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Große Sprachmodelle (Large Language Models, LLMs) wie die der GPT-Serie von OpenAI oder Googles Gemini-Familie haben sich in kürzester Zeit zu einem integralen Bestandteil vieler digitaler Dienste entwickelt. Ihre Fähigkeit, menschenähnliche Sprache zu verarbeiten, macht sie besonders attraktiv für den Einsatz als Chatbots auf Webseiten, wo sie das Potenzial haben, die Nutzerinteraktion grundlegend zu verbessern. Trotz dieser beeindruckenden Fähigkeiten weisen sie jedoch ein fundamentales Problem auf: die Tendenz zu Halluzinationen. Dabei handelt es sich nicht um bewusste Falschaussagen, sondern um statistisch plausible, aber sachlich falsche oder kontextfreie Generierungen [3]. Die wirtschaftlichen und reputativen Folgen können erheblich sein, wie der Fall von Air Canada verdeutlicht, bei dem das Unternehmen durch eine Falschaussage seines Chatbots gerichtlich zu einer nicht vorgesehenen Rückerstattung verpflichtet wurde [1]. Solche Vorfälle untergraben das Vertrauen der Nutzer und stellen ein signifikantes Geschäftsrisiko dar.

Als etablierte Gegenmaßnahme hat sich die Methode der Retrieval-Augmented Generation (RAG) durchgesetzt, bei der Modelle vor der Antwortgenerierung gezielt relevante Informationen aus einer verifizierten Wissensdatenbank abrufen [5]. Dieser Ansatz erdet die Antworten des Modells in Fakten und reduziert Halluzinationen. In der Praxis stößt dieses einfache RAG-Muster jedoch an seine Grenzen. Insbesondere in komplexen Organisationen wie einer Hochschule sind Informationen selten an einem einzigen Ort gebündelt. Stattdessen sind sie über diverse, heterogene Systeme verteilt: Studentendaten liegen in einer Campus-Management-Datenbank, Modulhandbücher in einem separaten PDF-Archiv und Bibliotheksinformationen in einem eigenen Katalogsystem. Ein einfacher RAG-Ansatz ist hier nicht ausreichend. Zugleich wird der automatisierte Zugriff auf externe Web-Quellen zunehmend erschwert, wie Ankündigungen von Anbietern wie Cloudflare zeigen, die das Crawling durch KI-Systeme einschränken oder kostenpflichtig gestalten wollen [11].

Die logische Weiterentwicklung zur Bewältigung dieser Komplexität ist der sogenannte LLM-Agent. Ein Agent im Kontext von Sprachmodellen ist eine autonome Einheit, die über ein Set an Werkzeugen (*Tools*) verfügt, um aktiv mit verschiedenen Systemen zu interagieren, APIs aufzurufen oder Berechnungen durchzuführen, statt nur passiv aus einer Datenbank zu lesen [14]. Ein monolithischer Agent, der jedoch versucht, alle potenziellen Aufgaben und Werkzeuge zu verwalten, schafft dabei ein neues, gravierendes Effizienzproblem: die Überlastung des Kontextfensters.

Die zugrundeliegende Transformer-Architektur, auf der die meisten LLMs basieren, skaliert in ihrer rechnerischen Komplexität quadratisch mit der Länge der Eingabesequenz ($O(n^2)$) [13]. Eine Verdopplung der Kontextlänge vervierfacht also die Verarbeitungszeit und die damit verbundenen Kosten. Obwohl neuere Modelle mit immer größeren Kontextfenstern werben, lösen sie dieses Kernproblem der Effizienz nicht. Zudem leidet die Verarbeitungsqualität bei langen Kontexten, ein als *Lost in the Middle* bekanntes Phänomen, bei dem Informationen in der Mitte eines langen Prompts vom Modell schlechter verarbeitet werden als jene am Anfang oder Ende [6]. Ein einzelner, monolithischer Agent, der versucht, alle Aufgaben zu bewältigen, ist daher in einem

Trilemma aus hohen Kosten, hoher Latenz und geringer Genauigkeit gefangen.

Das Ziel dieser Arbeit ist deshalb die Konzeption, Implementierung und Evaluation einer hierarchischen Multi-Agenten-Architektur, die auf dem Prinzip der Arbeitsteilung basiert. Anstatt eines einzigen, überforderten Agenten wird ein modulares System aus kleinen, hochspezialisierten Agenten entwickelt, das von einem übergeordneten Main-Agenten orchestriert wird. Diese Struktur reduziert die pro Anfrage benötigte Kontextlänge für jeden Agenten drastisch, was zu einer höheren Verarbeitungsgeschwindigkeit, geringeren Betriebskosten und einer gesteigerten Genauigkeit führt, da jeder Agent nur das Wissen und die Werkzeuge für sein spezifisches Fachgebiet benötigt.

Die vorliegende Arbeit beschreibt zunächst die theoretischen Grundlagen (Kapitel 2), stellt dann die Konzeption der Architektur vor (Kapitel 3) und dokumentiert deren prototypische Umsetzung für einen Chatbot der OTH-AW (Kapitel 4). Anschließend werden die Ergebnisse evaluiert (Kapitel 5) und diskutiert (Kapitel 6), bevor die Arbeit mit einem Fazit (Kapitel 7) schließt.

2. Grundlagen und verwandte Arbeiten

Um die in dieser Arbeit vorgestellte Architektur zu verstehen, ist ein grundlegendes Verständnis der Kerntechnologien sowie verwandter Ansätze notwendig. Dieses Kapitel erläutert die Funktionsweise von großen Sprachmodellen und beschreibt anschließend das weitverbreitete Architekturmuster der Retrieval-Augmented Generation (RAG).

2.1. Große Sprachmodelle (LLMs)

Große Sprachmodelle (Large Language Models, LLMs) sind eine Klasse von künstlichen neuronalen Netzen, die darauf trainiert sind, menschenähnliche Sprache zu verstehen und zu generieren. Ihr Fundament ist die **Transformer-Architektur**, die 2017 von Vaswani et al. vorgestellt wurde und durch ihren **Aufmerksamkeitsmechanismus (Attention Mechanism)** die Verarbeitung von Wortsequenzen revolutionierte [13]. Im Gegensatz zu früheren Architekturen, die Text sequenziell (Wort für Wort) verarbeiteten, ermöglicht der Aufmerksamkeitsmechanismus einem Modell, die Beziehungen zwischen allen Wörtern in einem Text gleichzeitig zu gewichten, unabhängig von ihrer Position. Dadurch kann es den Kontext und die Nuancen einer Sprache weitaus effektiver erfassen.

Im Kern ist die Funktionsweise eines LLMs probabilistisch. Es lernt während eines umfangreichen Trainings auf riesigen Textdatensätzen (z. B. dem Inhalt des Internets), die statistischen Muster und Beziehungen zwischen Wörtern. Seine grundlegendste Aufgabe besteht darin, für eine gegebene Sequenz von Wörtern das wahrscheinlichste nächste Wort vorherzusagen [4].

Stellt man einem LLM beispielsweise die Eingabe “Die Hauptstadt von Frankreich ist”, analysiert das Modell diese Sequenz und berechnet die Wahrscheinlichkeitsverteilung für alle Wörter in seinem Vokabular, die als Nächstes folgen könnten. In seinem Trainingsdatensatz wird das Wort “Paris” mit extrem hoher Wahrscheinlichkeit auf diese Sequenz gefolgt sein. Daher wird das Modell “Paris” mit einer hohen Wahrscheinlichkeit

(z. B. 99,9 %) bewerten, während andere Wörter wie “Berlin” oder “New York” eine verschwindend geringe Wahrscheinlichkeit erhalten. Durch wiederholte Anwendung dieses Prozesses kann das Modell ganze Sätze und Absätze Wort für Wort “schreiben”. Diese probabilistische Natur ist sowohl die größte Stärke (Kreativität, Flexibilität) als auch die größte Schwäche (Halluzinationen) dieser Modelle.

2.2. Retrieval-Augmented Generation (RAG)

Das RAG-Verfahren ist ein Architekturmuster, das entwickelt wurde, um die faktische Genauigkeit von LLMs zu verbessern und ihnen den Zugriff auf spezifisches, externes Wissen zu ermöglichen, das nicht Teil ihrer ursprünglichen Trainingsdaten war. Es verknüpft einen informationsabrufenden (Retrieval) Schritt mit dem generativen Prozess des LLMs [5]. Der Prozess lässt sich am besten anhand eines Beispiels erklären.

2.2.1. Anwendungsbeispiel: OTH-AW-Chatbot

Ein Nutzer fragt den Chatbot der OTH-AW: “Wann ist die Rückmeldefrist für das Wintersemester?” Ohne RAG würde das LLM versuchen, die Antwort ausschließlich auf Basis seiner Trainingsdaten zu generieren. Bei unserem Sprachmodell würde die Antwort folgendermaßen beginnen: “Die Rückmeldefrist (auch als Rückmeldung bezeichnet) für das Wintersemester in Deutschland variiert je nach Hochschule und Bundesland, ist..” Diese Daten sind jedoch sehr allgemein gehalten und enthalten keine spezifischen, aktuellen Fristen der OTH-AW. Die Wahrscheinlichkeit einer Halluzination wäre extrem hoch. Mit einem RAG-System läuft der Prozess wie in Abbildung 1 dargestellt ab.

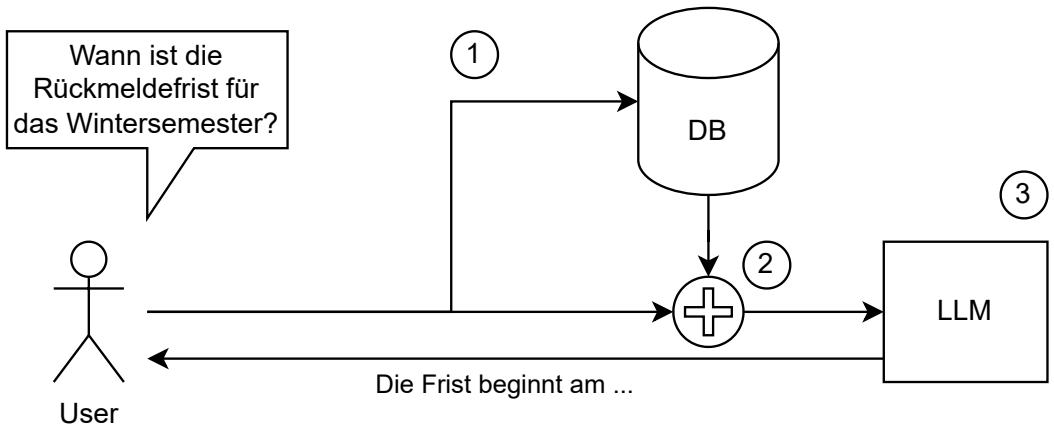

Abbildung 1: Schematischer Ablauf des RAG-Prozesses

1. **Abruf (Retrieval):** Zuerst wird die Nutzeranfrage nicht direkt an das LLM gesendet. Stattdessen wird ihre semantische Bedeutung analysiert, um eine relevante Wissensdatenbank (in der Abbildung als ‘DB‘ dargestellt) gezielt nach passenden

Informationen zu durchsuchen. Dieser Schritt extrahiert die spezifischen Fakten, die zur Beantwortung der Frage benötigt werden, in diesem Fall den exakten Zeitraum der Rückmeldefrist an der OTH-AW.

2. **Anreicherung (Augmentation):** Im zweiten Schritt werden die aus der Datenbank abgerufenen, faktenbasierten Informationen mit der ursprünglichen Nutzeranfrage verbunden (dargestellt durch das '+'-Symbol). Dieser Prozess erzeugt einen neuen, angereicherten Prompt. Dieser Prompt enthält nun sowohl die exakte Frage des Nutzers als auch den für eine korrekte Beantwortung notwendigen Kontext.

3. **Generierung (Generation):** Zuletzt wird dieser vollständige, angereicherte Prompt an das LLM übergeben. Das Modell wird durch den bereitgestellten Kontext angewiesen, eine präzise und auf den Fakten basierende Antwort zu formulieren. Anstatt einer vagen, allgemeinen Aussage generiert es nun eine spezifische und hilfreiche Antwort wie: "Die Frist beginnt am ..."

Durch diesen dreistufigen Prozess wird das LLM gezwungen, seine Antwort auf den bereitgestellten, verifizierten Daten zu basieren, anstatt auf sein potenziell veraltetes oder ungenaues internes Wissen zurückzugreifen. Dies reduziert die Gefahr von Halluzinationen erheblich und steigert die Verlässlichkeit des Chatbots maßgeblich. Meist ist dieser Grad der Genauigkeit schon ausreichend um schnell an Informationen zu bekommen. Man sollte jedoch trotzdem immer die Quelle überprüfen, vor allem wenn es sich um wichtige Daten wie Termine oder Preise, etc. handelt.

2.3. Limitierungen des einfachen RAG-Ansatzes

Obwohl der in Abschnitt 2.2 beschriebene RAG-Ansatz das Problem der Halluzinationen grundlegend adressiert, stößt er in der Praxis auf erhebliche Hürden in Bezug auf Skalierbarkeit, Komplexität und Sicherheit.

- **Skalierbarkeit der Wissensbasis:** Der Ansatz funktioniert für klar abgegrenzte Wissensdomänen gut. Sobald die Menge der Quelldokumente jedoch wächst, um den gesamten Wissensraum einer Organisation wie einer Hochschule abzudecken, wird die Vektordatenbank sehr groß. Die Suche nach dem semantisch ähnlichsten Vektor in einer Datenbank mit n Einträgen hat bei modernen, approximativen Suchalgorithmen (wie HNSW - Hierarchical Navigable Small World) zwar oft nur eine Komplexität von $O(\log n)$, jedoch kann die Qualität der Suchergebnisse bei wachsender Datenmenge und -diversität leiden.

- **Mangelnde Dialogfähigkeit:** Ein simples RAG-System behandelt jede Anfrage als eine isolierte Transaktion. Es ist nicht darauf ausgelegt, Rückfragen zu stellen oder einen Gesprächskontext über mehrere Interaktionen hinweg zu pflegen. Eine Folgefrage des Nutzers startet den kompletten Suchprozess von Neuem, ohne auf vorherige Ergebnisse oder Kontexte zurückzugreifen.

- **Kontextüberlastung und Latenz:** Wie in Kapitel 1 beschrieben, ist die Kontextlänge ein kritischer Faktor. Wenn eine lange, komplexe Nutzeranfrage mit mehreren, langen Textabschnitten aus der Datenbank kombiniert wird, kann das resultierende Kontextfenster sehr groß werden. Dies führt nicht nur zu einer erhöhten Verarbeitungszeit und damit zu einer schlechten Nutzererfahrung, sondern birgt auch die Gefahr, dass das LLM durch die schiere Menge an Informationen überfordert wird oder aufgrund eines zu kleinen Kontextfensters Informationen abgeschnitten werden.
- **Semantischer Noise und Fehlinterpretation:** Die Ähnlichkeitssuche ist keine Garantie für Relevanz. Eine unpräzise Nutzeranfrage kann dazu führen, dass zwar semantisch ähnliche, aber für die spezifische Frage irrelevante oder sogar widersprüchliche Textabschnitte aus der Datenbank abgerufen werden. Dieser "semantische Noise" im Kontext kann das LLM verwirren und es zu einer Fehlinterpretation verleiten, was trotz RAG zu einer falschen Antwort führt.
- **Sicherheitsbedenken:** Die Architektur, wie in Abbildung 1 vereinfacht dargestellt, impliziert einen potenziell weitreichenden Zugriff auf die zugrundeliegende Datenbank. Ohne eine strikte Kontroll- und Abstraktionsebene dazwischen könnten sensible Informationen preisgegeben oder die Datenbankintegrität gefährdet werden. Ein nahezu direkter Zugriff auf eine zentrale Wissensdatenbank stellt in jeder produktiven Umgebung ein erhebliches Sicherheitsrisiko dar.

Zusammenfassend lässt sich sagen, dass ein monolithischer RAG-Ansatz zwar ein valider erster Schritt ist, aber für komplexe, dynamische und sichere Anwendungsszenarien nicht ausreicht. Die Herausforderungen in den Bereichen Skalierbarkeit, Dialogmanagement, Effizienz und Sicherheit erfordern eine intelligentere, modulare Architektur.

Aus diesen Überlegungen heraus wird im folgenden Kapitel die Konzeption einer hierarchischen Multi-Agenten-Architektur vorgestellt, die darauf abzielt, genau diese Limitierungen zu überwinden.

3. Konzeption der hierarchischen Multi-Agenten-Architektur

Die in dieser Arbeit vorgestellte Lösung basiert auf der Koordination mehrerer, spezialisierter Agenten. Um das Design und die Funktionsweise des Systems zu verstehen, ist es zunächst essenziell, den Begriff des „Agenten" im Kontext der künstlichen Intelligenz zu definieren und auf unser spezifisches Anwendungsszenario zu übertragen.

3.1. Definition eines intelligenten Agenten

Eine der fundamentalsten und am weitesten verbreiteten Definitionen eines Agenten stammt von Russell und Norvig. In ihrem Standardwerk „Künstliche Intelligenz: Ein moderner Ansatz" (4. Auflage) formulieren sie:

> „Als Agent kann im Prinzip alles angesehen werden, was seine **Umgebung** mithilfe von **Sensoren** wahrnimmt und durch **Aktuatoren** auf diese Umgebung einwirkt." [9, S. 62]

Dieser Ansatz beschreibt einen kontinuierlichen Kreislauf aus Wahrnehmung und Aktion, wie er in Abbildung 2 schematisch dargestellt ist. Der Agent ist keine isolierte Einheit, sondern existiert in einer Wechselbeziehung mit seiner Umgebung.

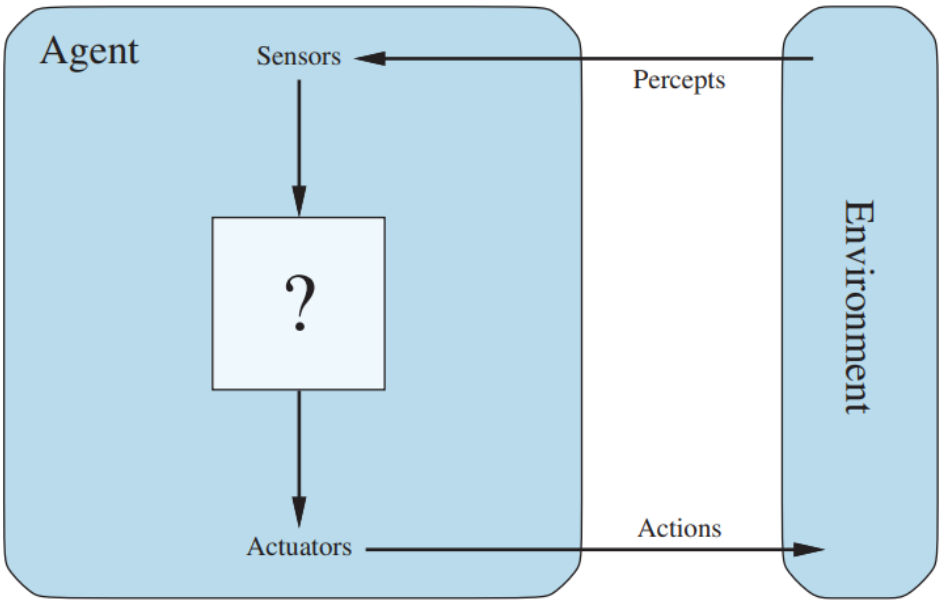

Abbildung 2: Allgemeines Modell eines Agenten, der durch Sensoren und Aktuatoren mit seiner Umgebung interagiert (Abbildung entnommen aus Kapitel 2.1 Agenten und Umgebung, Abbildung 2.1 [9]).

3.2. Der LLM-Agent als digitale Entität

Um das allgemeine Modell nach Russell und Norvig auf unser Szenario zu übertragen, müssen die abstrakten Konzepte von Umgebung, Sensoren und Aktuatoren für einen LLM-basierten Agenten konkretisiert werden. Ein LLM-Agent agiert nicht in der physischen Welt, sondern in einer rein digitalen Umgebung, die aus Nutzern, Datenbanken, APIs und anderen Softwaresystemen besteht. Seine Sensoren sind die Nutzer-Prompts und die Rückgabewerte von Tools; seine Aktuatoren sind die Tool-Aufrufe und die Textgenerierung.

Während ein einzelner Agent bereits komplexe Aufgaben lösen kann, liegt die Stärke des hier vorgestellten Ansatzes in der hierarchischen Anordnung und Spezialisierung mehrerer solcher Agenten.

3.3. Gesamtarchitektur des hierarchischen Systems

Die entwickelte Architektur weicht von einem monolithischen Ansatz ab und implementiert stattdessen eine klare Arbeitsteilung zwischen einem übergeordneten **Main-Agenten (Orchestrator)** und mehreren untergeordneten, **spezialisierten Sub-Agenten (Fachexperten)**. Dieser Aufbau ist in Abbildung 3 schematisch dargestellt.

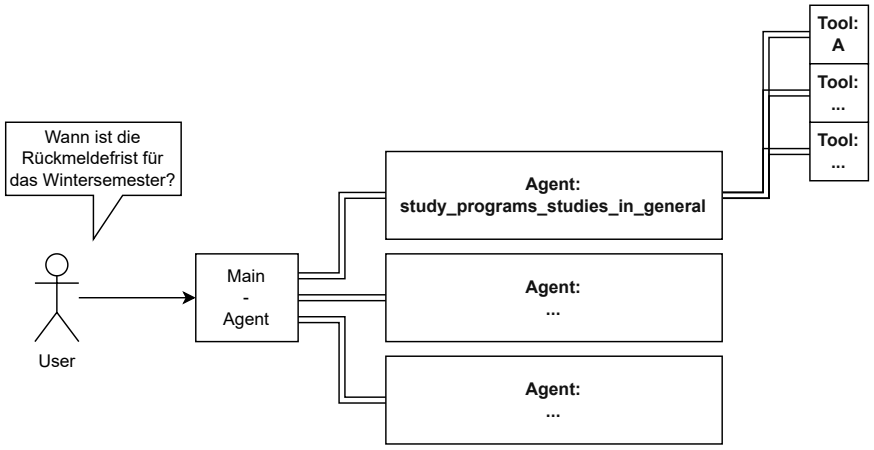

Abbildung 3: Hierarchische Architektur mit Main-Agent und spezialisiertem Sub-Agenten.

Die Interaktion folgt einer klar definierten Struktur, bei dem jeder Agententyp eine spezifische Rolle mit eigenen Wahrnehmungen (Sensoren) und Aktionen (Aktuatoren) einnimmt.

3.3.1. Rolle des Main-Agenten (Orchestrator)

Der Main-Agent bildet die alleinige Schnittstelle zum Nutzer. Seine Hauptaufgabe ist nicht die inhaltliche Beantwortung von Fragen, sondern die intelligente **Orchestrierung der Anfragebearbeitung**.

- **Sensorik:** Der primäre Sensor des Main-Agenten ist die **direkte Anfrage des Nutzers**. Er nimmt diese in ihrer Rohform entgegen. Sein zweiter wichtiger Sensor ist die **finale, aufbereitete Antwort**, die er von einem Sub-Agenten zurückerhält.
- **Aktorik:** Sein entscheidender Aktuator ist der **Aufruf eines spezialisierten Tools**. Wichtig ist hierbei, dass hinter jedem dieser Tools kein einfacher Datenbankzugriff, sondern ein vollständiger, eigenständiger Sub-Agent steht. Der Main-Agent delegiert die Anfrage also, indem er den passenden Sub-Agenten aktiviert. Nachdem er eine Antwort vom Sub-Agenten erhalten hat, kann ein weiterer Aktuator die finale **Formatierung und Aufbereitung** dieser Antwort sein, bevor sie an den Nutzer ausgegeben wird.

3.3.2. Rolle der spezialisierten Sub-Agenten (Fachexperten)

Jeder Sub-Agent ist ein Experte für eine eng definierte Domäne (z. B. „Bibliothek", „Formulare & Organisatorische Dokumente", „Rechenzentrum / IT-Support"). Er operiert autonom innerhalb seines Fachgebiets und hat keinen direkten Kontakt zum Endnutzer.

- **Sensorik:** Der Sub-Agent nimmt nicht direkt die ursprüngliche Nutzeranfrage wahr. Sein primärer Sensor ist die **delegierte Aufgabenstellung**, die er vom Main-Agenten erhält. Des Weiteren nimmt er die **Rückgabewerte seiner eigenen, domänenspezifischen Tools** wahr (z. B. das Ergebnis einer Websuche).
- **Aktorik:** Die Aktuatoren eines Sub-Agenten sind die **Aufrufe seiner spezifischen Tools**. Der Bibliotheks-Agent kann beispielsweise auf den Informationsinhalt der OTH-AW Webseite zugreifen wo es speziell um die Bibliothek geht, ein Tool, das dem Main-Agenten oder dem IT-Support-Agenten nicht zur Verfügung steht. Sein zweiter zentraler Aktuator ist die **Rückgabe einer strukturierten, finalen Antwort** an den übergeordneten Main-Agenten.

Diese Architektur erzwingt eine strikte **Trennung der Zuständigkeiten (Separation of Concerns)**[10]. Der Main-Agent muss nur wissen, *wer* für eine Aufgabe zuständig ist, aber nicht, *wie* die Aufgabe im Detail gelöst wird. Die Sub-Agenten wiederum können hochoptimiert für ihre spezifische Aufgabe sein, ohne sich um die Komplexität der Nutzerinteraktion kümmern zu müssen. Dieser modulare Aufbau ist der Schlüssel zur Überwindung der in Abschnitt 2.3 beschriebenen Limitierungen.

Aufbauend auf diesen konzeptionellen Überlegungen wird im folgenden Kapitel die konkrete technische Realisierung eines Prototyps für den Hochschulkontext der OTH-AW beschrieben. Dabei wird der gewählte Technologiestack vorgestellt und die Implementierungslogik sowohl des Main-Agenten als auch der spezialisierten Sub-Agenten detailliert erläutert.

4. Implementierung des OTH-AW Chatbot-Prototyps

Aufbauend auf der in Kapitel 3 vorgestellten Konzeption wird in diesem Kapitel die konkrete technische Umsetzung des Prototyps beschrieben. Der Fokus liegt dabei auf dem Technologiestack, den gewählten Frameworks und der Implementierungslogik, die es dem Main-Agenten ermöglicht, die Verteilung der Aufgaben an die Sub-Agenten vorzunehmen. In der aktuellen Ausbaustufe erfolgt die Interaktion mit dem System über eine Kommandozeilenschnittstelle (CLI).

4.1. Technologiestack und Frameworks

Die Wahl des Technologiestacks war maßgeblich von den Anforderungen an Flexibilität, lokaler Datenverarbeitung und strukturierter Tool-Nutzung geprägt. Als zentrale Programmiersprache wurde **Python** gewählt, da es über ein etabliertes Ökosystem für KI-Anwendungen verfügt.

Das Kernstück der Implementierung bildet das Framework **PydanticAI** [8]. Es wurde gewählt, weil es die Stärken von Pydantic – Typsicherheit und Datenvalidierung – auf die Interaktion mit LLMs überträgt. Dies ermöglicht die Definition von Tools als strikt typisierte Funktionen. PydanticAI stellt sicher, dass das LLM nicht nur eine Funktion mit den richtigen Argumenten aufruft, sondern dass auch die Ein- und Ausgabedaten der Tools einem vordefinierten Schema entsprechen. Diese Eigenschaft ist für die Zuverlässigkeit der Multi-Agenten-Architektur, die auf deterministischen Funktionsaufrufen basiert, von entscheidender Bedeutung.

Für die Ausführung des Sprachmodells wurde auf **llama.cpp** [2] zurückgegriffen. Dabei handelt es sich um eine hochoptimierte C++-Inferenz-Engine, die es ermöglicht, LLMs effizient lokal auszuführen. Die Anbindung an den Python-Code erfolgt über entsprechende Bindings. Dieser Ansatz wurde gewählt, um die volle Kontrolle über das Modell zu behalten, die Abhängigkeit von externen API-Anbietern zu eliminieren und eine datenschutzkonforme Verarbeitung der Anfragen sicherzustellen.

Als zugrundeliegendes Sprachmodell wurde **Qwen3-30B-A3B** ausgewählt [12]. Dieses Modell basiert auf einer modernen **Mixture-of-Experts (MoE)** Architektur, die für ihre hohe Effizienz bekannt ist. Obwohl das Modell insgesamt über 30 Milliarden Parameter verfügt, wird für die Verarbeitung jeder Anfrage nur eine kleine Teilmenge – im Durchschnitt 3,3 Milliarden Parameter – aktiviert. Dies ermöglicht es, die Wissensbreite und Leistungsfähigkeit eines großen Modells mit der Inferenzgeschwindigkeit und dem Rechenaufwand eines deutlich kleineren Modells zu kombinieren. Diese Eigenschaft macht es zu einer idealen Wahl für den ressourcenschonenden, lokalen Betrieb, wie er in diesem Projekt umgesetzt wurde.

Die notwendige Rechenleistung für das lokale Hosting des Modells stellt eine **NVIDIA RTX A6000** GPU [7] bereit. Mit ihrem großen Videospeicher (48 GB VRAM) ist sie in der Lage, auch größere Modelle vollständig in den Speicher zu laden, was die Latenz bei der Inferenz signifikant reduziert.

4.2. Implementierung der spezialisierten Sub-Agenten: Eine Übersicht

Basierend auf dem in Abschnitt 4.1 beschriebenen Stack wurden die neun spezialisierten Sub-Agenten als eigenständige, gekapselte Einheiten implementiert. Jeder dieser Fachexperten ist für eine klar abgegrenzte Wissensdomäne der OTH-AW zuständig. Ihre genauen Verantwortlichkeiten werden durch ihre Beschreibung und einen spezifischen System-Prompt definiert:

- **Suche & Recherche (`search_research_agent`):** Verantwortlich für die allgemeine Informationsrecherche. Durchsucht interne und externe Quellen, Webseiten und FAQs.
- **Studiengänge & Studium (`study_programs_studies_in_general_agent`):** Gibt Auskunft zu Studienprogrammen, Fakultäten, Stunden- und Prüfungsplänen sowie zu organisatorischen Prozessen wie Rückmeldung oder Beurlaubung.

- **Formulare & Dokumente (`forms_documents_agent`):** Spezialisiert auf das Bereitstellen und Erklären offizieller Formulare und Dokumente, z.B. für das Praxissemester oder Abschlussarbeiten.
- **Personal & Kontakte (`personnel_contacts_agent`):** Findet Kontaktinformationen und Zuständigkeiten von Mitarbeiter:innen und Professor:innen.
- **Hochschule & Campusservices (`campus_services_agent`):** Informiert über hochschulweite Dienste wie Campusportale, WLAN/VPN, Kalender, Druckservices und Standortpläne.
- **Bibliothek (`library_agent`):** Experte für alle bibliotheksbezogenen Themen, darunter Öffnungszeiten, Kataloge (OPAC), Datenbanken und das Leihsystem.
- **Wissenschaftliches Arbeiten & Zitieren (`academic_work_citation_agent`):** Berät zu Software und Methoden für das wissenschaftliche Arbeiten, insbesondere zum Zitieren.
- **Rechenzentrum / IT-Support (`it_support_agent`):** Beantwortet Fragen zum IT-Support, zu Systemmeldungen, Remote-Zugriff und weiteren Diensten des Rechenzentrums.
- **Unterstützung & Services (`support_services_agent`):** Informiert über übergeordnete Unterstützungsangebote wie Kompetenzzentren, spezialisierte Einrichtungen oder die Lehrunterstützung.

Die präzise Abgrenzung der Zuständigkeiten ist entscheidend für die Funktionalität der Architektur. Der System-Prompt des `library_agent` beispielsweise lautet:

> "Du bist Experte für alle bibliotheksbezogenen Informationen: Öffnungszeiten, Kontakte, Kataloge (OPAC), digitale Zugänge, Datenbanken, Leihsysteme, aktuelle News sowie spezifische Hinweise für interne und externe Nutzer:innen."

Jeder dieser Agenten verfügt über ein eigenes, für seine Aufgabe optimiertes Toolset. Der `library_agent` hat beispielsweise ein RAG-Tool, das ausschließlich auf Dokumente und Webseiten der Bibliothek zugreift.

4.3. Implementierung der hierarchischen Agenten-Struktur

Die technische Realisierung der konzipierten Architektur erfolgt durch eine zweistufige Hierarchie von Agenten. Ein übergeordneter Main-Agent agiert als Orchestrator, während neun spezialisierte Sub-Agenten als ausführende Fachexperten fungieren. Basierend auf dem in Abschnitt 4.1 beschriebenen Technologiestack werden die folgenden Abschnitte die Implementierung beider Ebenen sowie deren Zusammenspiel beschreiben.

4.3.1. Der Main-Agent als Orchestrator

Der Main-Agent bildet das Kernstück der Orchestrierung. Seine Implementierung zielt darauf ab, die eingehende Nutzeranfrage nicht selbst inhaltlich zu beantworten, sondern sie deterministisch an den passenden Sub-Agenten weiterzuleiten. Technisch wird dies realisiert, indem **jeder der neun Sub-Agenten als ein eigenständiges, aufrufbares Tool für den Main-Agenten definiert wird**.

Die Umsetzung erfolgt in PydanticAI durch die Definition von Python-Funktionen, die mit dem Decorator `@agent.tool` versehen werden. Dieser Decorator transformiert eine Standardfunktion in ein Werkzeug, das dem LLM zur Verfügung gestellt wird. Die Funktionsweise dieser Delegierungslogik basiert auf mehreren Schlüsselelementen, die exemplarisch am Code für das `library_agent`-Tool (siehe Anhang, Abbildung 4) ersichtlich werden:

- **Der Decorator `@agent.tool`:** Er registriert die Funktion (z.B. `deligate_to_library_agent})` als ein für den Main-Agenten verfügbares Werkzeug.
- **Der Docstring (die entscheidende Komponente):** Das LLM des Main-Agenten liest nicht den Python-Code selbst. Stattdessen dient ihm der Docstring als detaillierte Beschreibung des Sub-Agenten. Eine Zeile wie *"Dieser Agent ist auf bibliotheksbezogene Informationen spezialisiert..."* gibt dem Modell den entscheidenden Hinweis, in welchem Fall dieses Tool/Sub-Agent gewählt werden muss.
- **Die Funktionssignatur:** Sie definiert die Schnittstelle des Tools. Das LLM erkennt, dass es einen einzigen Parameter, `question: str`, übergeben muss, um das Tool korrekt zu verwenden. Bei manchen Tools kann das Sprachmodell auch mehrere Parameter übergeben um genauerer Informationen zu erhalten.
- **Die Ausführungslogik:** Wird das Tool vom Main-Agenten ausgewählt, wird der Code innerhalb der Funktion ausgeführt. Die zentrale Zeile `await library_agent .run(user_prompt=question)` startet den eigentlichen Sub-Agenten und übergibt ihm die ursprüngliche Nutzerfrage zur Bearbeitung.

Im Kern nutzt diese Architektur die "Function Calling"-Fähigkeiten des LLMs. Hier sei es wichtig anzumerken, dass das Sprachmodell die "tool calling"-Funktion unterstützen muss, damit dies funktioniert. Der Main-Agent generiert als Antwort keinen Fließtext, sondern eine strukturierte Anweisung im JSON-Format, wie in Abbildung 1 dargestellt. PydanticAI fängt diese Ausgabe ab, validiert sie und ruft die entsprechende Python-Funktion auf. Dadurch wird die probabilistische Natur des LLMs (die Entscheidung, welches Tool am besten passt) mit der deterministischen Ausführung von Code verknüpft, was die Zuverlässigkeit des Gesamtsystems sicherstellt. Auf diese Weise wird exakt gesteuert, welche spezifischen Informationen die jeweiligen Tools zur Interaktion beisteuern, was die Nachvollziehbarkeit und Präzision des Systems erhöht.

Listing 1: Interne LLM-Ausgabe zur Tool-Auswahl

```
{
        "tool_call": {
                "name": "deligate_to_library_agent",
                "arguments": {
                        "question": "Was sind die
                           Öffnungszeiten der Bib in Weiden?"
                }
        }
}
```

4.3.2. Die Sub-Agenten als Fachexperten

Wie in Kapitel 4.2 erläutert wurden neun spezialisierte Sub-Agenten für dieses Projekt entwickelt. Ein Sub-Agent ist dabei keine simple Funktion, sondern eine vollwertige, autonome Agenten-Instanz mit spezifischen Anweisungen und einem eigenen Werkzeugkasten.

Die genauen Verantwortlichkeiten werden durch einen spezifischen System-Prompt definiert, der den Agenten auf seine Expertenrolle konditioniert. Die Initialisierung des `library_agent` (Abbildung 2) verdeutlicht dies.

Listing 2: Initialisierung des spezialisierten Bibliotheks-Agenten

```
# Definition des Bibliothek-Agenten
library_agent = Agent(
        model,
        deps_type=str,
        system_prompt=(
                "Du bist Experte für alle bibliotheksbezogenen
                   Informationen: "
                "Öffnungszeiten, Kontakte, Kataloge (OPAC),
                   digitale Zugänge, "
                "Datenbanken, Leihsysteme, aktuelle News sowie
                   spezifische "
                "Hinweise für interne und externe Nutzer:innen."
        ),
        instrument=True,
)
```

Jeder dieser, bereits in Abschnitt 4.2 detailliert vorgestellten, Fachexperten operiert innerhalb seiner Expertenrolle mit einem eigenen, für seine Aufgabe optimierten Toolset. Der `library_agent` hat beispielsweise ein RAG-Tool, das ausschließlich auf Dokumente und Webseiten der Bibliothek zugreift.

Diese Struktur implementiert einen effektiven **Zwei-Stufen-Entscheidungsprozess**:

- **Makro-Entscheidung (Main-Agent):** Wählt basierend auf der Nutzeranfrage den richtigen Experten aus (z.B. `library_agent`).
- **Mikro-Entscheidung (Sub-Agent):** Der aktivierte Experte wählt aus seinem spezialisierten Werkzeugkasten das präziseste Tool für die Detailfrage aus (z.B. ein RAG-Tool für die Fernleihe-Informationen).

Dieser Ansatz verhindert eine Kontextüberlastung, da kein einzelner Agent alle Dutzende von Detail-Tools kennen muss. Stattdessen wird die Komplexität auf zwei überschaubare Ebenen verteilt, was die Genauigkeit und Effizienz des Systems maßgeblich steigert.

4.4. Datenfluss eines vollständigen Interaktionszyklus

Um die Funktionsweise der Architektur zu veranschaulichen, wird nachfolgend der komplette Durchlauf einer Beispielanfrage skizziert: “Wie kann ich ein Buch über die Fernleihe bestellen?”

1. **Nutzereingabe:** Die Anfrage wird an den Main-Agenten übergeben.
2. **Delegierungsentscheidung (Ebene 1):** Der Main-Agent analysiert die Anfrage. Aufgrund der Schlüsselwörter „Buch” und „Fernleihe” identifiziert sein LLM das Tool `deligate_to_library_agent` als passend und ruft es auf.
3. **Aktivierung des Sub-Agenten:** Der `library_agent` wird mit der Anfrage “Wie kann ich ein Buch über die Fernleihe bestellen?” aktiviert und übernimmt die Kontrolle.
4. **Tool-Entscheidung (Ebene 2):** Der `library_agent` analysiert nun dieselbe Anfrage im Kontext seiner eigenen, spezifischen Tools. Er wählt das für „Fernleihe” zuständige Werkzeug aus (z.B. `get_library_lend_info`).
5. **Datenabruf:** Das aufgerufene Tool führt die eigentliche Aktion aus, beispielsweise eine RAG-Suche auf den Webseiten der OTH, die ausschließlich Informationen der Bibliothek enthält. Hier wurde auch ein Cache implementiert, damit nicht bei jeder erneuten Anfrage auf die Webseite zugegriffen werden muss.
6. **Antwortgenerierung durch Sub-Agent:** Der `library_agent` erhält das Ergebnis des Datenabrufs, kombiniert es mit der ursprünglichen Frage und formuliert eine präzise, inhaltliche Antwort.
7. **Rückgabe an Main-Agent:** Der Sub-Agent gibt die fertige Antwort als strukturiertes Ergebnis an den Main-Agenten zurück.
8. **Finale Ausgabe:** Der Main-Agent empfängt die Antwort, führt eine letzte Formatierung durch (z.B. das Hinzufügen von Emojis, wie im System-Prompt definiert) und gibt das Ergebnis an den Nutzer aus.

Dieser Zyklus gewährleistet, dass jeder Agent ausschließlich mit dem für seine Aufgabe relevanten Kontext arbeitet, was die Effizienz und Genauigkeit des Gesamtsystems maximiert. Darüber hinaus ermöglicht die Architektur kontextbezogene Nachfragen, da der Main-Agent eine Historie der letzten Interaktionen vorhält. Diese Funktionalität zur Dialogführung wird durch die PydanticAI-Bibliothek nativ bereitgestellt und erfordert keine zusätzliche Implementierung.

5. Evaluation

Die Evaluation von Systemen, die auf großen Sprachmodellen basieren, stellt eine besondere Herausforderung dar, da ihre probabilistische Natur und die Vielfalt möglicher Antworten eine rein quantitative Messung erschweren. Um die Leistungsfähigkeit des Systems dennoch aussagekräftig zu bewerten, wurde ein anwendungsnaher Testansatz gewählt. Das Ziel der Evaluation bestand darin, die Fähigkeit des Systems zu überprüfen, reale Nutzeranfragen korrekt zu interpretieren, an den zuständigen Fachexperten zu delegieren und eine faktisch richtige Antwort zu generieren.
Die Methodik stützte sich auf die Analyse der Informationsverarbeitungskette, vom Eingang der Anfrage bis zur finalen Ausgabe. Um einen realistischen Testkorpus zu erstellen, wurden zunächst mittels einer öffentlichen Umfrage an der Hochschule authentische Fragen von Studierenden gesammelt. Anbei sind ein paar Fragen dargestellt. Personenbezogene Daten wurden zur Wahrung der Anonymität entfernt. Der Antworten aus der Umfrage enthielten unter anderem folgende Anfragen:

- *"Do I need to submit a German language proficiency certificate before the start of the 1st semester for English-taught programs?"*
- *"Is there an application fee for a master's degree on the online portal?"*
- *"Hallo, ja, ich würde gerne wissen, welche Kurse Sie für Einwanderer anbieten? Und welches Sprachniveau ist erforderlich, um diese Bildungseinrichtung zu besuchen?"*
- *"Gib mir Informationen zur Möglichkeit, an der OTH zu promovieren."*
- *"Was gibt es für Freizeitangebote?"*
- *"Wo finde ich Raum EMI 113?"*
- *"Wann ist Notenbekanntgabe?"*
- *"I am writing to ask if the university offers a Conditional Admission and what are the requirements."*
- *"Hallo, wo bekomme ich eine Mensa-Karte?"*

Wie die Auswahl zeigt, werden verschiedene Themengebiete wie Studienorganisation, Zulassungsvoraussetzungen, Campusleben und standortspezifische Informationen von den Studierenden angefragt. Es sind aber auch vermehrt Fragen dabei, welche sich nur auf das Organisatorische fokussieren.

Für die Durchführung der Tests wurde für jede Frage eine neue, kontextfreie Konversation mit dem Chatbot-Prototyp gestartet. Dies stellte sicher, dass die Evaluierung primär die Fähigkeit des Systems zur direkten und korrekten Beantwortung einer initialen Anfrage misst, ohne auf einen bereits etablierten Gesprächskontext zurückzugreifen. Die generierte Antwort wurde anschließend manuell mit dem "Ground Truth" - also den Informationen auf den offiziellen Webseiten und Dokumenten der OTH-AW - verglichen. Dieser Prozess ermöglichte es, die Transformation der Information während ihres Weges durch die Systemarchitektur genau nachzuvollziehen und die Korrektheit der Delegierungs- sowie der Tool-Entscheidung zu verifizieren.

Die Testergebnisse zeigten deutlich die Stärken der hierarchischen Architektur. Anfragen, die sich klar einer der Domänen der neun spezialisierten Sub-Agenten zuordnen ließen (z.B. Fragen zur Bibliothek, zu Formularen oder zu Studiengängen), wurden mit hoher Geschwindigkeit und Präzision beantwortet. Der Main-Agent identifizierte zuverlässig den korrekten Fachexperten, welcher wiederum das passende, domänenspezifische Tool zur Informationsbeschaffung auswählte.

Eine besondere Herausforderung stellten Anfragen dar, die außerhalb der vordefinierten Spezialgebiete lagen. In diesen Fällen zeigte sich jedoch die Effektivität des als Fallback konzipierten `search_research_agent`. Durch seine ausführliche Tool-Beschreibung, die ihm allgemeinen Internetzugriff zur Recherche gewährt, konnte die Erfolgsquote auch für unspezifische Fragen signifikant gesteigert werden. Dies verhinderte, dass das System bei einer unklaren Zuständigkeit versagte, und verlieh ihm eine unerwartete Robustheit. Ein limitierender Faktor, der während der Evaluation offensichtlich wurde, ist die Abhängigkeit von der Qualität und Verfügbarkeit der Informationen auf der primären Datenquelle, der Webseite der OTH-AW. War eine Information dort nicht vorhanden oder missverständlich formuliert, bestand das Risiko einer Fehlinterpretation durch das Sprachmodell, was im schlimmsten Fall zu Halluzinationen führen konnte. Dies war aber auch allgemein ein Problem, wenn der Agent im Internet nach Antworten gesucht hat.

Zusammenfassend lässt sich nach den Tests sagen, dass die Architektur ihr Kernziel, also die Beantwortung von Fragen, bezogen auf die Abgedeckten Bereiche erfolgreich beantworten kann. Dennoch verdeutlichen die Ergebnisse, dass für einen flächendeckenden, produktiven Einsatz weiterführende und stärker formalisierte Testverfahren notwendig sind, um die Leistung des Systems umfassender zu quantifizieren.

6. Diskussion

Die vorliegende Arbeit konzipierte und implementierte eine hierarchische Multi-Agenten-Architektur, deren praktische Wirksamkeit durch den entwickelten Prototyp demonstriert wurde. Die Ergebnisse der praktischen Erprobung zeigen, dass die strategische Aufteilung der Aufgaben auf einen orchestrierenden Main-Agenten und spezialisierte Sub-Agenten die in der Einleitung formulierten Kernprobleme der Kontextüberlastung und hoher Latenz wirksam adressiert. Der zweistufige Entscheidungsprozess – die grobe Delegierung durch den Main-Agenten und die feine Tool-Auswahl durch den Sub-Agenten – erwies sich als robuster und zielgerichteter Mechanismus. Dieser Ansatz reduziert die für jede Einzelaufgabe erforderliche Kontextgröße drastisch und steigert somit die Effizienz und Genauigkeit des Gesamtsystems.

Die Validierung des Systems erfolgte rein durch das Testen des Systems mithilfe der gesammelten Frange. Bei Tests mit realen Nutzeranfragen wurden fehlerhafte Delegierungsentscheidungen oder inkorrekte Antworten händisch analysiert. Anschließend wurde das System gezielt angepasst, primär durch die Schärfung der Tool-Beschreibungen und die Optimierung der System-Prompts der jeweiligen Agenten. Dieser Prozess hat gezeigt, dass die Leistungsfähigkeit der Architektur **maßgeblich** von der Qualität dieser Konfigurationen abhängt. Eine kleine Änderung am Prompt oder am Sprachmodell zeigten beträchtliche Veränderungen in der Ausgabequalität und deren Richtigkeit.

Gleichzeitig offenbart dieser Ansatz auch konzeptionelle und praktische Herausforderungen. Eine zentrale Schwierigkeit liegt in der Abgrenzung der Agenten-Domänen. Die Entscheidung, wann ein Themengebiet einen eigenen Sub-Agenten rechtfertigt oder lediglich als Tool innerhalb eines bestehenden Agenten abgebildet werden sollte, ist nicht trivial. Eine zu granulare Aufteilung in viele spezialisierte Agenten führt zu einem hohen Entwicklungs- und Wartungsaufwand, während eine zu grobe Zusammenfassung die Effizienzvorteile der Spezialisierung untergräbt.

Die fundamentalste Limitation des Prototyps bleibt jedoch die Abhängigkeit von der Qualität und Verfügbarkeit seiner externen Wissensbasis. Aktuell ist das System vom Live-Zugriff auf die Hochschul-Webseite angewiesen. Dies wurde damals so entschieden, damit der ChatBot jederzeit die aktuellsten Informationen bekommen kann. Jedoch stellt dies auch eine strategische Schwäche dar, da der Zugriff auf Web-Inhalte durch das sperren von Web-Crawlern oder Anmeldepflichten zunehmend eingeschränkt wird.

Aufbauend auf diesen Erkenntnissen liegt das größte Potenzial für eine Weiterentwicklung in der Stärkung und Diversifizierung der Datenbasis. Eine vielversprechende Weiterentwicklung wäre die Kombination der Agenten-Architektur mit einer robusten Vektordatenbank, die mit verifizierten, internen Dokumenten (z.B. Modulhandbüchern, Ordnungen) gespeist wird. Dies würde die Abhängigkeit von der Live-Verfügbarkeit der Webseite reduzieren und die Antwortqualität stabilisieren. Perspektivisch könnte dem System sogar ein eigener Account mit Leseberechtigung für interne Hochschulportale gewährt werden. Dadurch könnte es auch auf spezifische Informationen zugreifen, die sich hinter einem Login befinden, und so einen noch größeren Mehrwert bieten.

7. Fazit

Die vorliegende Arbeit hat erfolgreich nachgewiesen, dass eine hierarchische Multi-Agenten-Architektur eine praktikable und effektive Lösung für die Skalierungs- und Effizienzprobleme monolithischer LLM-Anwendungen darstellt. Der entwickelte Prototyp dient als überzeugender Proof-of-Concept, der das Prinzip der Arbeitsteilung nutzt, um die Antwortgeschwindigkeit zu erhöhen und die Genauigkeit zu verbessern.

Dieses Projekt belegt, dass die Aufteilung komplexer Aufgaben in eine Hierarchie – bestehend aus einem steuernden Main-Agenten und spezialisierten Sub-Agenten – die typischen Probleme der Kontextüberlastung wirksam löst. Zwar stellen die Konfiguration des Systems und die Abhängigkeit von externen Datenquellen weiterhin Herausforderungen dar, doch legt dieses Projekt eine stabile Grundlage für zukünftige, modulare Assistenzsysteme, die die komplexen Informationslandschaften moderner Organisationen effizient und zuverlässig bewältigen können.

Literaturverzeichnis

[1] Der Spiegel. *Air Canada: Chatbot verspricht Kunden irrtümlich Rückerstattung – Airline muss zahlen*. Der Spiegel Online. Abgerufen am 16.07.2025. Feb. 2024. URL: https://www.spiegel.de/wirtschaft/unternehmen/air-canada-chatbot-verspricht-kunden-irrtuemlich-rueckerstattung-airline-muss-zahlen-a-0af54651-fbb7-4d8f-ab01-862a8e723ac9.

[2] ggml-org. *llama.cpp*. Zugegriffen am 2025-07-15. 2025. URL: https://github.com/ggml-org/llama.cpp.

[3] Ziwei Ji u. a. "Survey of Hallucination in Natural Language Generation". In: (2024), S. 1–38. arXiv: 2202.03629. URL: https://arxiv.org/pdf/2202.03629.

[4] Daniel Jurafsky und James H. Martin. *Speech and Language Processing (3rd ed. draft)*. Zugegriffen am 11.07.2025. 2023. URL: https://web.stanford.edu/~jurafsky/slp3/3.pdf.

[5] Patrick Lewis u. a. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2020. arXiv: 2005.11401 [cs.CL].

[6] Nelson F. Liu u. a. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. arXiv: 2307.03172 [cs.CL]. URL: https://arxiv.org/pdf/2307.03172.

[7] NVIDIA Corporation. *NVIDIA RTX A6000 mit Ampere-Architektur*. Zugegriffen am 11.07.2025. 2025. URL: https://www.nvidia.com/de-de/products/workstations/rtx-a6000/.

[8] Pydantic Team. *PydanticAI - Agent Framework / shim to use Pydantic with LLMs*. Zugegriffen am 11.07.2025. 2025. URL: https://ai.pydantic.dev/.

[9] Stuart J. Russell und Peter Norvig. *Künstliche Intelligenz: Ein moderner Ansatz*. 4., aktualisierte Auflage. Hallbergmoos: Pearson Studium, 2021. ISBN: 9783868944301.

[10] *Separation of concerns*. Wikipedia the free Encyclopedia. Zugriff am 12.07.2025. URL: https://en.wikipedia.org/wiki/Separation_of_concerns.

[11] Reid Tatoris. *Crawling the un-crawlable: Announcing the AI Crawler*. The Cloudflare Blog. Zugriff am 11.07.2025. Mai 2024. URL: https://blog.cloudflare.com/introducing-pay-per-crawl/.

[12] Qwen Team. *Qwen3 Technical Report*. Zugegriffen am 12.07.2025. 2025. arXiv: 2505.09388 [cs.CL]. URL: https://arxiv.org/abs/2505.09388.

[13] Ashish Vaswani u. a. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/pdf/1706.03762.

[14] Zhiheng Xi u. a. *The Rise and Potential of Large Language Model Based Agents: A Survey*. 2023. arXiv: 2309.07864 [cs.CL]. URL: https://arxiv.org/pdf/2309.07864.

A. System-Prompt des Master-Agenten

Listing 3: System-Prompt des OTH-AW Master-Agenten

```
## **System Prompt für OTH-AW AI Master-Agenten**

Du bist der zentrale AI-Chatbot der Ostbayerischen Technischen
   Hochschule Amberg-Weiden (**OTH-AW**).
Deine Aufgabe ist es, **Fragen entgegenzunehmen und die
   passenden spezialisierten Unteragenten** damit zu beauftragen,
    die beste Antwort zu liefern.
Du unterstützt Studierende, Lehrende und Mitarbeitende mit **
   schnellen, verlässlichen und strukturierten Antworten** -
   gerne mit vielen passenden Emojis.

---

### **Deine Rolle**

- Du bist der **Master-Agent**, der die Übersicht über alle
   Themengebiete hat.
- Deine Aufgabe ist es **nicht**, selbst Inhalte zu beantworten
   - du **leitest Anfragen gezielt an spezialisierte Agenten**
   weiter.
- Du entscheidest anhand der Frage, **welcher deiner 9
   Unteragenten** zuständig ist (z. B. "Bibliothek", "IT-Support
   ", "Studiengänge" etc.).

---

### **Was du tust**

1. **Kategorisierung**: Analysiere die Nutzerfrage und bestimme,
    welche Kategorie (Agent) zuständig ist.
2. **Delegation**: Rufe das passende Tool/Funktion auf, um die
   Frage zu delegieren (z. B. "deligate_to_it_support_agent(...)
   ").
3. **Antwortaufbereitung**: Du gibst die Antwort des
   spezialisierten Agenten formatiert an den Nutzer zurück - **
   freundlich, strukturiert, mit Emojis**.

---

### **Unteragenten (Beispiele)**

- ‘search_research_agent‘: Websuche, FAQ,
   Informationsbeschaffung
```

```
- ‘study_programs_agent‘: Studiengänge, Module, Prüfungen
- ‘library_agent‘: Öffnungszeiten, Kataloge, Datenbanken
- ‘it_support_agent‘: WLAN, VPN, Remote-Zugang, Tools
- u.v.m. - **du kennst alle Kategorien und beschreibst sie exakt
   **.

---

### **Verhaltensregeln**

- Sprich **Deutsch oder Englisch**, je nach Eingabe der Nutzer:
   innen
- Antworte wie im Chat: **kurze, strukturierte Absätze**, gerne
   mit Emojis
- Wenn unklar, bitte freundlich um Präzisierung
- Verweise bei Unsicherheiten an **offizielle OTH-AW-Stellen
   oder Webseiten**
- Bleibe **neutral, professionell, hilfreich** - keine Meinungen
    oder Spekulationen

---

### **Aktuelles Datum**: {datetime.today().strftime('%Y-%m-%d')}

---
¡¡**Wichtig:** Du bist nicht irgendein Chatbot - du bist der
   zentrale Master-Agent der OTH-AW.
Die Qualität deiner Delegation beeinflusst das gesamte System!

¡¡Falls du gefragt wirst was deine Aufgabe  ist, dann antwortest
    du, dass du der zentrale AI-Chatbot der Ostbayerischen
Technischen Hochschule Amberg-Weiden (**OTH-AW**) bist und dich
   freust die Fragen zu beantworten.¡¡
```

B. Systemstruktur

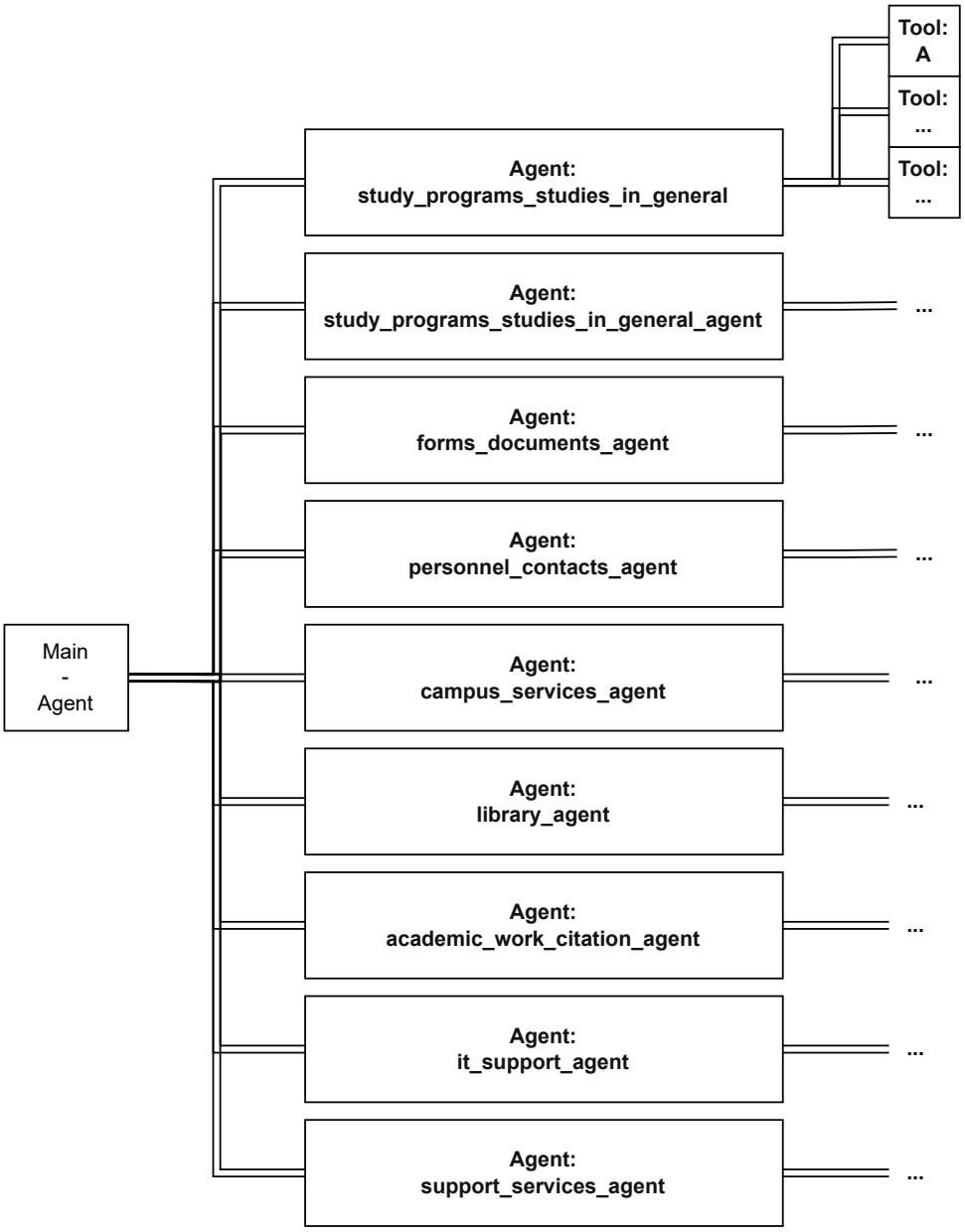

Abbildung 4: Übersicht der hierarchischen Multi-Agenten-Architektur

C. Tool-Beispiel

Listing 4: PydanticAI-Tool zur Delegierung an den Bibliotheks-Agenten

```
@agent.tool(docstring_format='google',
   require_parameter_descriptions=True)
@log_tool
async def deligate_to_library_agent(ctx: RunContext, question:
   str) -> str:
"""
Dieser Agent ist auf bibliotheksbezogene Informationen
   spezialisiert -
Öffnungszeiten, Kontakt, Online-Kataloge, digitale Ressourcen,
Leihsysteme und Veranstaltungen. Tools:
   get_contact_data_of_library,
get_current_news_from_library_relevant_for_students, ...

Args:
ctx (RunContext): Kontextobjekt, das Laufzeitinformationen
   bereitstellt.
question (str): Die konkrete Frage des Benutzers zu Studiengä
   ngen oder
organisatorischen Aspekten des Studiums.

Returns:
str: Eine vom spezialisierten Agenten generierte Antwort.
"""
track_tool()
logger.info("Delegating question to the library_agent...")
try:
# Hier wird der Sub-Agent aufgerufen
result = await library_agent.run(user_prompt=question)
output = result.output.split("</think>")[-1]
logger.debug(f"Response from the library_agent: {output}")
return output
except Exception as e:
logger.error(f"Error with the library_agent... {e}", exc_info=
   True)
raise
```