universität wien

Bachelorarbeit

ENTWURF UND UMSETZUNG EINER GRAPHBASIERTEN MULTI-AGENTENARCHITEKTUR ZUR VERBESSERTEN GENERIERUNG VON BLOGBEITRÄGEN MIT LARGE LANGUAGE MODELS UNTER NUTZUNG VON RETRIEVAL-AUGMENTED GENERATION

Verfasser

Marc Rodenbäck

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2026 2

Studienkennzahl lt. Studienblatt:	UA 033 526
Fachrichtung:	Informatik - Wirtschaftsinformatik
Betreuer:	Dipl.-Ing. Dr.techn. Marian Lux

Inhaltsverzeichnis

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Neuentwicklung und Evaluierung eines agentenbasierten Chatbots zur automatisierten Erstellung von Blogposts. Eine erste Gap-Analyse der ursprünglichen Umsetzung des Prototypen mit CrewAI deckte fundamentale Schwächen auf, wie mangelnde Transparenz, schwerfällige Bedienbarkeit und Schwierigkeiten bei der Wartung und Erweiterung. Zur Lösung dieser Limitierungen wurde das System auf eine graphenbasierte Architektur mit LangGraph migriert, woraus eine modulare 4-Schichten-Architektur hervorging. Im Backend erzwingt ein deterministischer Graph nun die Selbstreflexion der Agenten, während das Frontend als interaktiver Telegram-Bot mit Status-Streaming in Echtzeit realisiert wurde.

1 Einleitung

1.1 Motivation und Ausgangslage

Die rasante Entwicklung der Künstlichen Intelligenz und ihrer mannigfaltigen Einsatzmöglichkeiten hat in den vergangenen Jahren nicht nur die akademische Lehre, sondern auch die unternehmerische Praxis deutlich verändert. Moderne Sprachmodelle werden zunehmend als High-Tech-Werkzeuge in Geschäftsprozesse integriert.

Ich persönlich habe mich fast mein gesamtes Studium mit diesen Technologien auseinandergesetzt, sowohl in der theoretischen Lehre als auch in der alltäglichen Anwendung. Diese akademische Basis wurde durch meine letzte berufliche Tätigkeit als IT-Consultant in einem Systemhaus, dessen Dienstleistungen auch Data-Science- und AI-Lösungen umfassten, maßgeblich geprägt. In dieser Rolle durfte ich das Thema Künstliche Intelligenz sowohl aus vertrieblicher Perspektive als auch in der Kundenberatung aus unterschiedlichen Perspektiven beleuchten und die enormen Potenziale, aber auch die Herausforderungen bei der Integration von KI-Lösungen in Geschäftsprozesse kennenlernen.

Am Ende meines Studiums wollte ich mein theoretisches Basiswissen durch eine praktische Implementierung ergänzen. Das vorliegende Bachelorprojekt, die architektonische Neuentwicklung eines agentenbasierten Chatbots zur automatisierten Blogpost-Erstellung, bot einen passenden Rahmen dafür.

1.2 Zielsetzung der Arbeit

Das Hauptziel dieser Arbeit besteht in der Konzeption, Entwicklung und Evaluierung einer fähigen Multi-Agenten-Architektur. Die Basis ist ein bestehender, auf dem Framework CrewAI konzipierter Chatbot-Prototyp, der Blogbeiträge generieren soll. Wie die praktische Anwendung aber zeigte, stieß dieser Prototyp an Grenzen hinsichtlich Flexibilität, Effizienz, Wartbarkeit und User Experience (UX).

Um diese Einschränkungen abzubauen, wurde das System im Rahmen dieser Arbeit auf das graphenbasierte Framework LangGraph migriert. Ziel ist es,

einen expliziten, deterministischen Workflow zu erschaffen, der falsche Annahmen der Modelle unterdrückt und sich strikt an Einschränkungen durch den Nutzer hält. Ergänzend zur Restrukturierung des Backends, wird außerdem ein benutzerorientiertes Interface in Telegram umgesetzt, welches die bisherige textbasierten Eingaben durch eine einladende Oberfläche mit Status-Updates und Button-Navigation ersetzt.

2 Theoretische Grundlagen

2.1 Large Language Models und Prompt-Engineering

Large Language Models (LLMs) zeigen den aktuellen Stand der Technik im Bereich der Verarbeitung natürlicher Sprache (Natural Language Processing) auf. Die grundlegende Funktionsweise dieser auf künstlichen neuronalen Netzen basierenden Modelle beruht auf der statistischen Vorhersage von Textbausteinen, sogenannten Tokens. Anstatt Sprache auf einer rein regelbasierten oder semantischen Ebene zu verstehen, berechnen LLMs auf Basis der vorangegangenen Sequenz die mathematische Wahrscheinlichkeit für das jeweils nächste Token und generieren so iterativ zusammenhängende Texte [13].

Eine technische Einschränkung bei der Arbeit mit LLMs bildet das sogenannte Kontextfenster (Context Window). Dieses definiert die maximale Anzahl an Tokens, die das Modell in einem einzigen Durchlauf verarbeiten kann [13]. Da das Modell kein persistentes Gedächtnis über diese Grenze hinaus besitzt, müssen alle für eine Aufgabe relevanten Informationen, Instruktionen und bisherigen Konversationsverläufe zwingend innerhalb dieses Limits übergeben werden. Dies macht eine effiziente Steuerung und Begrenzung der Eingabedaten - insbesondere bei komplexen Workflows - zwingend erforderlich.

Um das Verhalten von LLMs effektiv zu steuern, kommt das sogenannte Prompt-Engineering zum Einsatz. Darunter versteht man die systematische Konzeption und semantische Programmierung von Eingabeaufforderungen, um die Ausgabequalität der Modelle zu maximieren. Während einfache Instruktionen bei komplexen Logik- oder Textgenerierungsaufgaben häufig zu inkorrekten oder fehlerhaften Aussagen (Halluzinationen) führen, können fortgeschrittene Techniken die Zuverlässigkeit des Systems merklich erhöhen.

Ein wesentlicher Durchbruch in diesem Bereich ist das Chain-of-Thought (CoT) Prompting [11]. Bei dieser Methode wird das Modell durch spezifische Anweisungen dazu gezwungen, seine Problemlösung nicht direkt als finales Ergebnis auszugeben, sondern den Weg dorthin in transparente, nachvollziehbare Zwischenschritte zu zerlegen [11]. Durch dieses erzwungene, schrittweise "Nachdenken"vor der eigentlichen Textgenerierung wird die Fehleranfälligkeit des Modells reduziert, da sich Schlussfolgerungen auf den explizit generierten Kontext der vorherigen Schritte beziehen. In dieser Arbeit bildet die genannte Methodik das Fundament für die Strukturierung der Workflows, um qualitativ hochwertige Blogposts zu generieren.

2.2 Retrieval-Augmented Generation

Moderne Large Language Models verfügen über eine gewaltige Wissensbasis, welche während ihrer Trainingsphase in den Modellparametern (parametrisches Gedächtnis) gespeichert wurde. Dennoch weisen sie zwei fundamentale Limitierungen auf: Ihr Wissensstand ist nach Abschluss des Trainings statisch und sie tendieren bei Wissenslücken zu plausibel klingenden, aber inkorrekten Aussagen (Halluzinationen). Um diese Schwächen zu überwinden, wurde das Konzept der Retrieval-Augmented Generation (RAG) entwickelt. RAG erweitert die KI-gestützte Textgenerierung um einen externen Abruf von Informationen aus anderen Wissensquellen (nicht-parametrisches Gedächtnis) [6].

Der typische Workflow bei RAG wird nach drei Phasen unterschieden: Indexierung, Abruf und Generierung [4]. Zunächst werden im Rahmen der Indexierung externe Quellen, wie lokale Dokumente oder Web-Inhalte, in kleinere Textabschnitte unterteilt. Diese "Chunks"werden anschließend durch ein Embedding-Modell in Vektoren transformiert und in einer Vektordatenbank (wie beispielsweise ChromaDB) gespeichert. Das Retrieval-System sucht daraufhin in der Datenbank nach den Vektoren mit der höchsten mathematischen Übereinstimmung (meist über die Kosinus-Ähnlichkeit). Somit werden die relevantesten Textabschnitte identifiziert [4].

In der letzten Phase der Generierung werden die Informationen als zusätzlicher Kontext in den Prompt des LLMs geladen [6]. Das Modell beantwortet dann nicht mehr nur aus seinem Trainingswissen, sondern wird auf die bereitgestellten Fakten geerdet (Grounded Generation). Für die Architektur der vorliegenden Arbeit ist RAG eine essenzielle Technologie: Durch die Kombination aus Dokumenten-Retrieval und Websuchen wird sichergestellt, dass die autonomen Agenten (insbesondere in der Rolle des Researchers) fundiert recherchieren können und die finalen Blogposts auf einer validen Faktenbasis beruhen.

2.3 Multi-Agenten-Systeme (MAS) und Graph-Architekturen

Die Lösung komplexerer Aufgaben ist für isolierte Large Language Models oftmals problematisch. Die Vermischung von Informationsbeschaffung, der Strukturierung und kreativer Textgenerierung in einem einzigen Prompt überfordert das System gerne, es wird also fehleranfällig. Einen Lösungsansatz für diese Problematik bilden Multi-Agenten-Systeme (MAS). Dabei wird eine umfassende Aufgabe in modulare Teilbereiche zerlegt und an spezialisierte KI-Agenten mit klar definierten Rollen, wie etwa *Researcher*, *Editor* oder *Writer*, delegiert [12]. Durch diese Arbeitsteilung können die Modelle gezielter auf ihre jeweilige Teilaufgabe fokussiert und besser mit passenden Modellen und Prompts ausgestattet werden.

Moderne MAS sind in der Lage Feedbackschleifen zu implementieren. Anstatt Text streng linear und ohne qualitative Kontrollinstanz zu generieren, evaluieren sich die Agenten selbst [10]. Möglich wird das z.B. durch einen zyklischen Prozess, bei dem ein generierender Agent (Actor) einen Entwurf erstellt, welcher von einem bewertenden Agenten (Evaluator) kritisiert wird, woraufhin

der Actor seinen Output selbstständig korrigiert [10]. Dieser Mechanismus ist dazu da, um Halluzinationen zu verringern und bildet die theoretische Basis für die in dieser Arbeit entwickelte *Fact-Checker-Loop*.

Um solch komplexe Workflows technisch robust und transparent abzubilden, etablieren sich zunehmend zustandsbasierte Graph-Architekturen. Solche Systeme (wie beispielsweise LangGraph) definieren den KI-Workflow explizit als gerichteten Graphen [5]. Hierbei repräsentieren die Knoten (Nodes) die ausführenden Agenten, während die Kanten (Edges) den Kontrollfluss und die Bedingungen definieren. Ein zentrales Zustandsobjekt (State) wird dabei kontinuierlich zwischen den Knoten weitergegeben und aktualisiert. Dadurch wird eine feingranulare Kontrolle über den Datenfluss ermöglicht, sowie eine transparente Fehlerverfolgung und die nahtlose Einbindung von Korrekturschleifen.

2.4 Prinzipien des UX/UI-Designs in der Mensch-Maschine-Interaktion

Mit dem rasanten Fortschritt generativer KI-Modelle vollzieht sich in der Human-Computer Interaction (HCI) ein Paradigmenwechsel: Klassische grafische Benutzeroberflächen werden zunehmend durch dialogorientierte Systeme, sogenannte Conversational User Interfaces (CUIs), ergänzt oder abgelöst [3]. Während diese textbasierten Schnittstellen eine natürliche und intuitive Kommunikation versprechen, bergen sie spezifische Herausforderungen für die Usability.

Ein zentrales Problem vieler textbasierter KI-Systeme ist die mangelnde Handlungsaufforderung offener Texteingabefelder. Nutzer stehen häufig vor der Schwierigkeit, die Fähigkeiten und Grenzen des Systems nicht richtig zu verstehen. Eine Studie von Luger und Sellen (2016) belegt, dass die Diskrepanz zwischen Nutzererwartung und tatsächlicher Systemleistung, kombiniert mit der Unklarheit darüber, welche Befehle das System verarbeiten kann, zu kognitiver Überlastung und Frustration führt [7].

Um diese Hürden zu überwinden, etablieren sich spezifische Design-Richtlinien für die Interaktion mit KI-Systemen. Eine fundamentale Regel, wie sie beispielsweise in den *Guidelines for Human-AI Interaction* postuliert wird, lautet: "Make clear what the system can do-[1]. Für die Konzeption des Frontends bedeutet dies, dass der Nutzer nicht durch ein leeres Textfeld alleingelassen werden darf. Stattdessen sollten Interaktionsmöglichkeiten durch strukturierte UI-Elemente wie z.B. vordefinierte Auswahlmenüs aktiv geführt werden.

Ein weiterer kritischer UX-Faktor bei der Orchestrierung komplexer Multi-Agenten-Systeme ist die Systemtransparenz. Wenn generative Prozesse mehrere Minuten in Anspruch nehmen, ohne dem Nutzer ein kontinuierliches Feedback über den Status des Fortschritts zu liefern, sinkt das Vertrauen in die Zuverlässigkeit des Systems drastisch [7]. Die vorliegende Arbeit versucht dieses Problem gezielt im Frontend-Design via Telegram anzugehen. Durch den Ersatz mühsamer Texteingaben durch flüssige Button-Navigation und die Implementierung eines Live-Status-Streamings wird die Ungewissheit eliminiert und der interne Workflow der Agenten für den Nutzer jederzeit nachvollziehbar visualisiert.

3 Systemarchitektur und Implementierung

In diesem Kapitel wird die technische Konzeption und praktische Umsetzung der Architektur vorgestellt. Der vollständige Code des Projekts ist in einem öffentlichen GitHub-Repository dokumentiert [9].

3.1 Konzeption der Gesamtarchitektur

Um den in der Gap-Analyse identifizierten Schwächen der Vorgängerversion entgegenzuwirken, wurde das System von Grund auf neu designet. Die zentrale architektonische Leitlinie bildet dabei die strikte Trennung von Zuständigkeiten. Die Applikation ist in eine modulare 4-Schichten-Architektur unterteilt, um Konfiguration, Workflow-Logik, Modellauswahl und das Benutzerinterface (UI) bestmöglich voneinander zu entkoppeln (vgl. Abbildung 1).

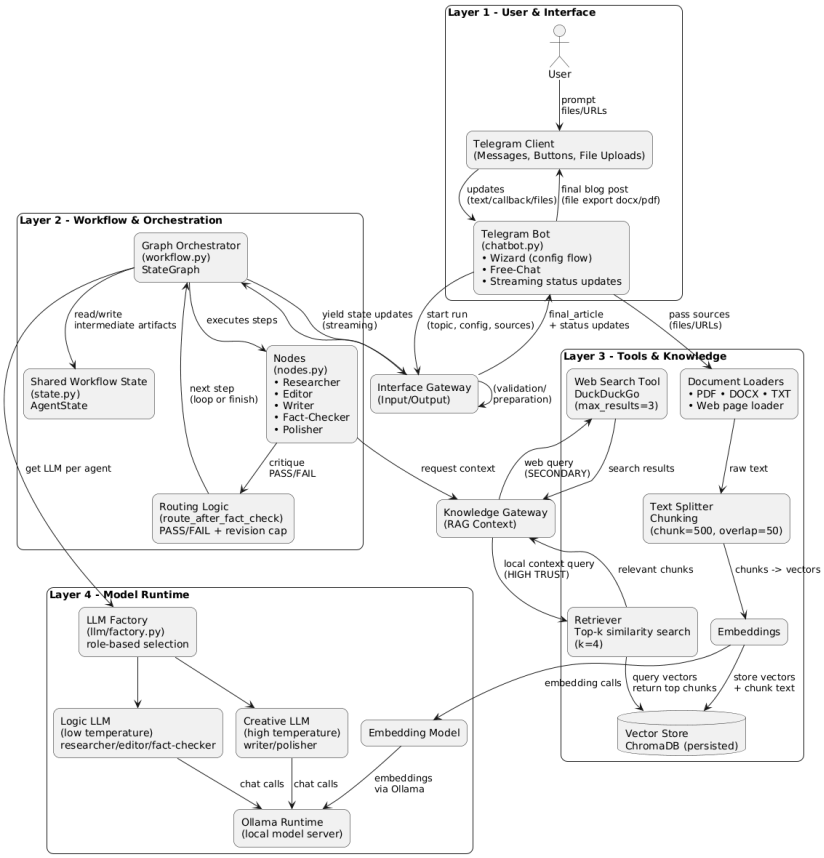

Abbildung 1: Schichtenarchitektur des Systems

Die Architektur wurde in verschiedene Ebenen gegliedert:

- **Layer 1 - User & Interface:** Diese Schicht kapselt die Interaktion mit dem Endanwender. Sie nutzt einen Telegram-Bot als Frontend, um Eingaben (Prompts, Konfigurationen, Dateiuploads) zu speichern und sowohl den Verarbeitungsstatus via Live-Streaming, als auch das fertige Ergebnis als Markdown-Datei zurückzugeben.

- **Layer 2 - Workflow & Orchestration:** Das Herzstück des Systems bildet der in LangGraph implementierte Graph. Dieser orchestriert die spezialisierten Agenten-Knoten (Researcher, Editor, Writer, Fact-Checker und Polisher) und verwaltet den Systemzustand in *Agent State*. Zudem ist hier die Routing-Logik verankert, die den Ablauf hinter dem Fact-Checker steuert.

- **Layer 3 - Tools & Knowledge:** Diese Schicht stellt die externe Informationsbeschaffung im Sinne der RAG sicher. Sie umfasst das Laden und Verarbeiten lokaler Dokumente, deren Vektorisierung und Speicherung in ChromaDB, sowie eine begleitende Websuche zur Verifizierung von Fakten.

- **Layer 4 - Model Runtime:** Die unterste Ebene abstrahiert die Anbindung der Large Language Models. Durch eine dedizierte *LLM Factory* werden je nach Agentenrolle spezifische Modelle (Logik vs. Kreativ) zugewiesen und über eine lokale Ollama-Laufzeitumgebung ausgeführt.

Durch die Implementierung wird die explizite Trennung der Schichten sichergestellt, sodass beispielsweise ein Austausch des Frontends (Layer 1) oder ein Wechsel der LLM-Provider (Layer 4) durchgeführt werden kann, ohne die Workflow-Logik (Layer 2) modifizieren zu müssen.

3.2 Backend: LangGraph-Workflow und Agenten-Rollen

Die logische Orchestrierung der Agenten erfolgt im Backend über das Framework LangGraph. Wie in Abbildung 2 detailliert dargestellt, wird der Prozess der Blogpost-Generierung als gerichteter Graph modelliert. Die Knoten (Nodes) repräsentieren dabei die LLM-Agenten, während die Kanten (Edges) den Datenfluss und die logischen Abhängigkeiten definieren.

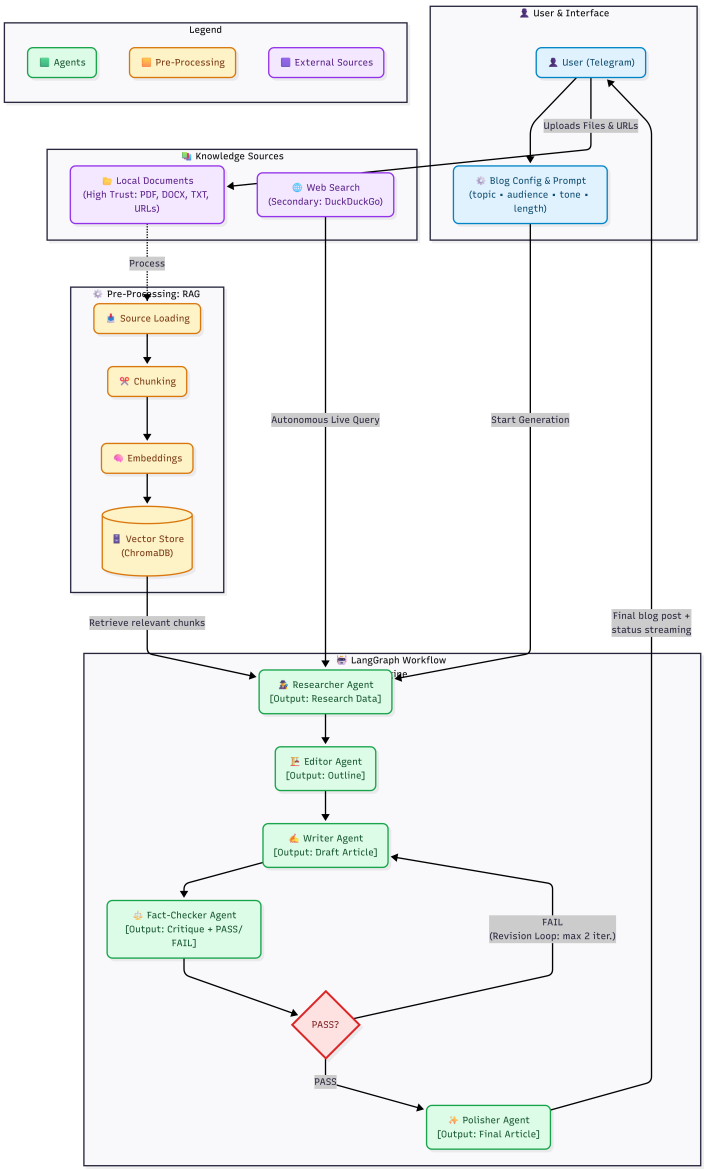

Abbildung 2: Detaillierter LangGraph-Workflow der Agenten

3.2.1 Zustandsverwaltung

Ein zentraler Unterschied zu intransparenten Frameworks wie CrewAI ist die Möglichkeit der expliziten Kontrolle der Zustände in LangGraph. Alle Agenten arbeiten auf einem geteilten Zustandsobjekt (`AgentState`), das bei jedem Schritt aktualisiert und an den nächsten Knoten weitergereicht [9]. Dies garan-

tiert maximale Transparenz und ermöglicht ein präzises Debugging. Der Zustand definiert exakt, welche Daten während des Durchlaufs erzeugt werden müssen:

```
from typing import TypedDict

class AgentState(TypedDict):

    topic: str
    target_len: str
    language_level: str
    information_level: str
    language: str
    tone: str
    additional_info: str
    source_documents: List[str]
    research_data: List[str]
    outline: List[str]
    draft: str
    critique: Optional[str]
    final_article: str
    revision_count: int
    current_status: str
```

Der **Researcher-Agent** befüllt diesen State zu Beginn mit gefundenen Fakten (`research_data`) , indem er die RAG-Pipeline abfragt. Daraufhin gestaltet der **Editor-Agent** durch den Inhalt eine klare Struktur (`outline`), bevor der **Writer-Agent** den ersten Textentwurf (`draft`) generiert.

3.2.2 Die Fact-Checker-Loop

Um das Risiko von Halluzinationen zu minimieren und die Textqualität zu steigern, wurde ein *Fact-Checker-Agent* in den Workflow eingebaut, der den vom Writer erstellten Entwurf kritisch gegen die gefundenen Recherche-Daten evaluiert und ein "PASSöder "FAIL"(im Fall "FAIL"mit konstruktiver Kritik) (`critique`) zurück gibt.

Hier unterscheidet sich die Architektur vom linearen Verarbeitungsfluss der originalen Implementierung mit CrewAI. Durch eine Bedingung wird eine Feedback-Schleife implementiert [9]. Die zugehörige Routing-Funktion entscheidet dynamisch, ob der Graph zum Polisher voranschreitet oder den Text zur Überarbeitung an den Writer zurückschiebt:

```
def route_after_fact_check(state: AgentState) -> str:


    if state.get("revision_count", 0) >= 2:
        return "polisher"
```

```
    critique = state.get("critique", "").strip().upper()

    if critique == "PASS" or critique.startswith("PASS") or critique == "":
        return "polisher"
    else:
        return "writer"
```

Durch die Deckelung auf maximal zwei Revisionsschleifen (`revision_count >= 2`) wird verhindert, dass sich der Prozess in Endlosschleifen verfängt. Erst wenn die Faktenprüfung "PASS"gibt oder das Limit erreicht wird, finalisiert der **Polisher-Agent** den Text, wendet das gewünschte Wording an und bereitet die Markdown-Formatierung für die Ausgabe vor.

3.3 Frontend: Telegram-Interface

Während der logische Aufbau des Systems im LangGraph-Backend liegt, erfordert die Interaktion mit dem Endnutzer ein intuitives und transparentes Frontend. Wie in der theoretischen Konzeption (vgl. Kapitel 2.4) dargelegt, scheitern viele textbasierte KI-Systeme an mangelnder Systemtransparenz und fehlender Führung des Users. Um diese Hürden zu überwinden, wurde das Frontend der Anwendung als interaktiver Telegram-Bot realisiert.

Anstatt den Nutzer mit einem leeren Textfeld zu konfrontieren, führt das System durch einen konfigurierbaren "Wizard". Über vordefinierte Buttons können Parameter wie Tonalität, Zielgruppe und Länge des Posts intuitiv ausgewählt werden, was die Bedienbarkeit deutlich verbessert [9].

3.3.1 Live-Streaming und Status-Transparenz

Eine wesentliche Herausforderung bei der Orchestrierung mehrerer Agenten ist die Verarbeitungsdauer, besonders da ein vollständiger Recherche- und Schreibzyklus mehrere Minuten in Anspruch nehmen kann. Um den von Luger und Sellen (2016) beschriebenen Vertrauensverlust durch Intransparenz zu verhindern, zeigt das Frontend ein kontinuierliches Live-Streaming der Zustände, in denen sich der Prozess zum jeweiligen Zeitpunkt befindet.

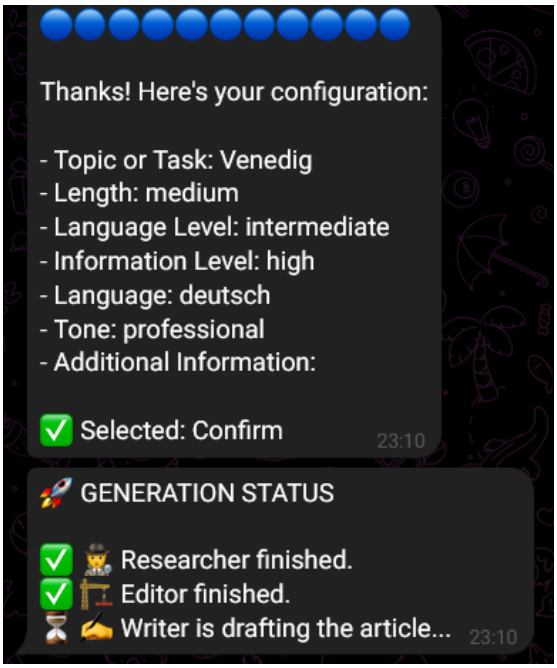

Abbildung 3: Telegram-Chat mit Status-Visualisierung

Sobald der Workflow im Graph gestartet wird, generiert das Backend bei jedem Übergang zwischen den Knoten ein Signal, dieses wird durch die Schnittstelle zum Interface empfangen, welches die bestehende Telegram-Nachricht entsprechend aktualisiert (vgl. Abbildung 3). Dies wird konkret über die `edit_message_text`-Methode der Telegram-API gelöst. Der Nutzer sieht so in Echtzeit, welcher Agent (z.B. *Researcher* oder *Fact-Checker*) aktuell arbeitet.

Ein vereinfachter Code-Ausschnitt illustriert diesen asynchronen Update-Mechanismus:

```
for node_name, state_update in output.items():
    final_state.update(state_update)

    if node_name == "researcher":
        status_text += " Researcher finished.\n"
        await status_msg.edit_text(
            status_text + "Editor is creating the outline..."
        )

    elif node_name == "editor":
        status_text += "Editor finished.\n"
```

```
        await status_msg.edit_text(
            status_text + "Writer is drafting the article..."
        )

    elif node_name == "writer":
        status_text += "Writer finished.\n"
        await status_msg.edit_text(
            status_text + "Fact Checker is verifying facts..."
        )

    elif node_name == "fact_checker":
        status_text += "Fact Checker finished.\n"
        critique = state_update.get("critique", "").strip().upper()
        rev_count = state_update.get("revision_count", 0)

        if (
            critique == "PASS"
            or critique.startswith("PASS")
            or critique == ""
            or rev_count >= 2
        ):
            await status_msg.edit_text(
                status_text
                + "Polisher is formatting the final text..."
            )
        else:
            await status_msg.edit_text(
                status_text
                + f"Fact Checker found errors! Writer is rewriting (Revision {rev_count})...
            )

    #[...]
```

Mit diesem Feedback wird die ursprüngliche Intransparenz des Systems beendet, wodurch der Nutzer ein unmittelbares Verständnis für die Arbeitsteilung der Agenten erhält, was die User Experience und die gefühlte Wartezeit drastisch verbessert. Nach erfolgreichem Durchlauf aller Agenten liefert der Bot den finalen Text als formatierte Markdown-Nachricht zum Download.

4 Methodisches Vorgehen

4.1 Frontend-First-Ansatz

Bevor die komplexe LangGraph-Logik entwickelt wurde, lag der erste Entwicklungsschritt in der Konzeption und Implementierung des Telegram-Bots.

Dieser Ansatz bot mehrere Vorteile:

- **Definition der Systemanforderungen:** Durch das frühzeitige Design der Buttons und des Wizard-Flows in Telegram konnten Testläufe während der Backend-Entwicklung deutlich beschleunigt werden.
- **Fokus auf Transparenz:** Die Notwendigkeit eines Live-Streamings wurde direkt zu Beginn der Entwicklung als Kernanforderung identifiziert und in den finalen Schritten des Projektes schließlich durch die darauf ausgerichtete Backend-Architektur mit minimalem Aufwand umgesetzt.
- **Entkopplung:** Durch die initiale Bereitstellung einer funktionierenden UI konnte das Backend im weiteren Verlauf modular und isoliert entwickelt werden, ohne die Schnittstelle zum Nutzer zu berühren.

Erst nachdem die Interaktion zwischen Nutzer und System durch das Frontend klar definiert war, wurde die Implementierung der Backend-Logik umgesetzt.

4.2 Framework-Wechsel: Von CrewAI zu LangGraph

In der Anfangsphase des Projekts wurde zunächst überlegt, das bereits im Vorprojekt bestehende Framework CrewAI für die Orchestrierung der Agenten beizubehalten. CrewAI zeichnet sich durch eine hohe Abstraktionsebene aus, die einen schnellen Einstieg in die Multi-Agenten-Entwicklung durch ein einfaches, rollenbasiertes Setup ermöglicht [2].

Aus Entwicklerperspektive erwies sich CrewAI als zu starr und intransparente "Blackbox". Der interne Zustand und der genaue Informationsfluss zwischen den Agenten werden weitgehend verborgen [2]. Dies erschwerte nicht nur das Debugging bei Fehlern massiv, bei Aktualisierungsversuchen zeigte sich außerdem der neu entstandenen Zwang auf die OpenAI-API zugreifen zu müssen, was eine Ansteuerung lokaler LLMs unmöglich machte. An dieser Stelle wurde die Entscheidung getroffen, einen Wechsel auf LangGraph zu vollziehen. Trotz eines höheren Entwicklungsaufwands, zeigten sich die Vorteile von LangGraph vor allem darin, dass die Implementierung spezifischer Kontrollstrukturen, wie der zyklischen *Fact-Checker-Loop*, ermöglicht wurde.

LangGraph adressiert die erwähnten Hürden nämlich durch einen expliziten graphenbasierten Ansatz. Anstatt den Kontrollfluss dem Framework zu überlassen, muss der Entwickler den Workflow als gerichteten Graphen mit Knoten und Kanten definieren [5]. Dies bot entscheidende Vorteile für die Implementierung:

- **Explizites State-Management:** Jeder Zwischenschritt der Agenten ist in einem typisierten State-Objekt greifbar und veränderbar [5].
- **Bedingtes Routing (Conditional Edges):** Komplexe Revisionsschleifen lassen sich programmatisch exakt steuern (z. B. maximal zwei Iterationen bei einem "FAIL"des Fact-Checkers).

- **natives Streaming:** Der Graphen-Status lässt sich problemlos abgreifen, was die zwingende Voraussetzung für die Telegram-Live-Status-Updates war.

Neben den pragmatischen Entwicklervorteilen war dieser Wechsel insbesondere aus wissenschaftlicher Perspektive in dieser Bachelorarbeit sinnvoll. Es eröffnete sich dadurch nämlich mehr Transparenz, Nachvollziehbarkeit und die Möglichkeit, Systemverhalten kausal zu erklären. Während ein abstrahiertes Framework, wie CrewAI, diese Analyse erschwert, zwingt LangGraph zur bewussten Modellierung eines deterministischen Multi-Agenten-Systems.

4.3 Definition und Durchführung der Testreihe

Um die Qualität und Performance der neu konzipierten Multi-Agenten-Architektur evaluierbar zu machen, wurde eine Testreihe konzipiert, die möglichst viele Benchmarks auswerten kann. Anstatt das System lediglich mit generischen Prompts zu prüfen, wurden einige spezifische Testfälle definiert.

Diese Testfälle unterschieden sich maßgeblich durch die RAG-Konstellation. Die Differenzierung erfolgte nach folgenden Zielsetzungen:

- **Aktualitäts-Check (Nur Websuche):** Evaluation, ob das System bei aktuelleren Themen (wie neuen Software-Releases) auf veraltetes Trainingswissen zurückgreift oder neuere Web-Informationen korrekt extrahiert.
- **Compliance-Check (Nur Dokumenten-Upload):** Ein gezielter Test zur Halluzinationsunterdrückung. Das System wurde mit einer fiktiven, logisch absurden Richtlinie konfrontiert (z. B. dem Verbot von Sauerstoff-produzierenden Pflanzen). Ziel war es zu prüfen, ob der *Fact-Checker* das vorgegebene Dokument strikt über das allgemeine Weltwissen des LLMs priorisiert.
- **Synthese-Test (Websuche und Dokument):** Die anspruchsvollste Metrik. Hier musste das System externe Optionen aus dem Web (z. B. reale Restaurant-Empfehlungen) mit den harten Restriktionen eines lokalen Dokuments (z. B. fiktive Budget-Limits) fehlerfrei abgleichen und filtern.
- **Baseline-Messung (Ohne externe Quellen):** Ein reiner Kreativitäts-Test zur Bewertung der strukturellen und sprachlichen Qualität des Modells (Strukturierung durch den *Editor*, Formulierung durch den *Writer*) ohne die kognitive Belastung durch einen RAG-Kontext.

4.3.1 Relevanz für das Benchmarking

Die methodische Definition dieser isolierten Metriken war für das Benchmarking der Architektur essenziell. Nur durch die gezielte Trennung der Informationsquellen ließ sich evaluieren, ob bei Prompts, den Modellen oder der Workflow-Logik Anpassungen vorgenommen werden sollten. Wenn das System

in der Baseline-Messung flüssig schreibt, aber im Compliance-Check an den vorgegebenen Fakten scheitert, erlaubt dieses Vorgehen somit eine klare Fehlereingrenzung.

Zudem bildeten diese vier Testfälle die standardisierte Grundlage, um im Verlauf der Arbeit die Output-Qualität zu unterschiedlichen Stufen im Entwicklungsprozess unter identischen Rahmenbedingungen objektiv miteinander vergleichen zu können und waren letztlich erforderlich, um das alte Projekt unter CrewAI gegen das aktuelle unter LangGraph zu messen.

4.4 KI-gestützte Entwicklung

Ein weiterer, wichtiger Ansatz in der Methodik dieser Arbeit ist die aktive Einbindung von Google Gemini 3.1 Pro in den Softwareentwicklungsprozess. Entsprechend dem Paradigma des AI-Assisted Software Engineering wurde das Tool iterativ als "Co-Pilot"genutzt, um die Entwicklerproduktivität zu steigern [8].

Ich nahm in diesem Projekt primär die Rollen des *Requirements Engineers* und *Solution Architects* ein, während die KI beim Prototyping, Brainstorming und bei repetitive Coding-Aufgaben unterstützt hat. Die konkrete Aufgabenteilung definierte sich wie folgt:

- **Menschlicher Lead (Architektur, Infrastruktur und UX):** Die strategische Systemkonzeption, die Definition der 4-Schichten-Architektur sowie der Entwurf der zugrundeliegenden Infrastruktur und Ordnerstruktur oblagen meiner Zuständigkeit. Ein besonderer Fokus lag auf der Konzeption und weiten Teilen der manuellen Implementierung des Telegram-Frontends (`chatbot.py`). Als QA-Tester definierte ich zudem die Testmetriken und evaluierte die Ergebnisse kritisch.
- **KI-gestütztes Bootstrapping (LangGraph und Prompting):** Aufgrund der hohen Komplexität von LangGraph übernahm die KI das initialen Prototypen und half mir, mich zügig mit dem Framework zurechtzufinden. Der allererste Systemprototyp ab der Graphen-Einführung wurde maßgeblich durch Gemini mitgestaltet und anschließend von mir orchestriert und angepasst. Ebenso lieferte die KI starke Unterstützung beim iterativen Prompt-Engineering der frühen Testphase.
- **Operative Code-Hygiene:** Auf der operativen Ebene übernahm die KI wiederkehrende, zeitintensive Aufgaben fast vollständig. Dazu gehörten die lückenlose Kommentierung des Quellcodes, der Aufbau der Logging-Infrastruktur, das Debugging sowie die UI-Formatierung im Frontend (z. B. der Einsatz von Emojis).

Diese methodische Vorgehensweise spiegelt moderne Best-Practices im Software Engineering wider. Die KI fungierte nicht als autonomer Entwickler, sondern als nützliches Werkzeug, das unter meiner Weisung unterstützt hat. Durch die konsequente Auslagerung von syntaktischer Fleißarbeit konnte die Entwicklungszeit deutlich verkürzt werden, womit es möglich war die gewonnene Zeit direkt in die Schärfung des Gesamtsystems fließen zu lassen.

Hinweis zur KI-gestützten Textredaktion

Neben der technischen Softwareentwicklung wurde Generative KI (Google Gemini 3.1 Pro) auch im iterativen Schreibprozess dieser Arbeit als unterstützendes Werkzeug eingesetzt. Entsprechend der Leitlinien zur guten wissenschaftlichen Praxis im Umgang mit KI-Technologien fungierte das Sprachmodell hierbei ausschließlich als digitales Lektorat. Der Einsatz beschränkte sich auf die Überprüfung der Orthografie, die sprachliche Glättung sowie die stilistische Feinjustierung der manuell verfassten Textentwürfe. Die inhaltliche Strukturierung, die wissenschaftliche Argumentation, die Auswertung der Testreihen sowie sämtliche Schlussfolgerungen stellen die alleinige geistige Eigenleistung des Autors dar.

5 Evaluierung

Nach der vollständigen Implementierung der Architektur bestand der abschließende Schritt in der Evaluierung des Systems. Ziel war es also zu prüfen, inwiefern die in der Gap-Analyse identifizierten Schwächen der Vorgängerversion durch den Neuentwurf neutralisiert wurden. Alle generierten Blogposts und Protokolle der durchgeführten Testreihen sind zur vollständigen Transparenz im öffentlichen GitHub-Repository des Projekts dokumentiert [9].

5.1 Behebung der architektonischen Schwächen (Backend)

Die Auswertung der Testfälle belegt, dass der neu eingeführte, zyklische Fact-Checker-Loop die ursprünglichen Defizite minimiert hat. Konfrontiert mit Restriktionen, wie etwa einem Budget-Limit (Testfall 3) oder absurden Firmenregeln (Testfall 2), hielt die LangGraph-Architektur die Vorgaben verlässlich ein. Widersprüchliches Trainingswissen der Modelle wurde durch die Faktenprüfung unterdrückt.

Darüber hinaus war die alte Version äußerst ineffizient und komplexe Aufgaben führten in der Blackbox-Architektur häufig dazu, dass sich die Agenten in unkontrollierbaren Endlosschleifen verfingen, was manuelle Systemabbrüche durch den Nutzer erzwang. Durch das State-System verläuft der Workflow nun deterministisch, ist aus Entwicklersicht jederzeit transparent nachvollziehbar und liefert den finalen Output im direkten Vergleich um ein Vielfaches schneller.

Ein weiterer Erfolgsfaktor ist der Umbau vom ineffizienten Öne-Size-Fits-AllAnsatz der Vorgängerversion, bei der ein einziges Modell (Llama 3.1) alle Aufgaben übernahm. Die neue Architektur nutzt stattdessen eine rollenspezifische *LLM-Factory*. Analytische Knoten (wie Researcher und Fact-Checker) greifen nun auf ein auf Logik optimiertes Modell (Qwen) zurück, Textgenerierung erfolgt hingegen durch ein kreatives Modell (Gemma). Außerdem ist es nun möglich, mit dem Parameter *temperature* die Flexibilität der Modelle auf den jeweiligen Agenten zu steuern.

5.2 Optimierung der Usability und Systemtransparenz (Frontend)

Neben den Problemen im Backend litt der Prototyp unter massiven Einschränkungen der Usability. Die Anwendung war für den User intransparent und schwer bedienbar, da die Nachrichten des Bots ohne Highlights und Absätze die Lesbarkeit erschwert haben und Konfigurationen mühsam in Einzelschritten als Freitext in den Chat abgetippt werden mussten. Zudem führte die im Durchschnitt hohe Verarbeitungsdauer gepaart mit mangelnder Transparenz über den Status des Prozesses (Nachricht durchgehend nur 'processing...') dazu, dass nicht absehbar war, ob und wann der Post fertig werden würde.

Diese Schwächen konnte vollständig eliminiert werden. Die Eingabeaufforderung wurde durch einen intuitiven Wizard-Prozess mit klickbaren Telegram-Inline-Keyboards ersetzt, was sowohl die Lesbarkeit als auch die Bedienbarkeit deutlich steigern konnte. Die Problem der fehlende Transparenz des Workflows wurde ebenfalls behoben, denn der Nutzer erhält nun kontinuierliche Status-Updates direkt im Chat. Der gesamte Prozess wird somit begleitet durch eine nachvollziehbare, visuelle Darstellung, was die User Experience (UX) auf ein professionelles Niveau hebt.

6 Adaptation Manual

Hinweis: Da sich der Quellcode und die Architektur dieses Projekts an internationale Open-Source-Standards richten, wurde die technische Systemdokumentation im Original auf Englisch verfasst. Das folgende Kapitel entspricht inhaltlich dem offiziellen Adaptation Manual aus dem zugehörigen GitHub-Repository [9] und dient als technischer Leitfaden zur Modifikation des Systems.

One of the primary design goals of this project is adaptability. The system architecture deliberately separates **configuration**, **workflow logic**, **agent behaviour**, **model selection**, **retrieval (RAG)**, and **interface**. This manual explains not only *what* can be adapted, but also *how* to implement typical modifications in a stable way.

6.1 Quick Orientation

Key adaptation surfaces (most stable → most invasive):

1. **Prompts & templates** (`config/agents.yaml`, `config/tasks.yaml`)
2. **Global settings** (`config/configs.yaml`, `.env`)
3. **Workflow topology & routing** (`graph/workflow.py`)
4. **Node logic / state schema** (`graph/nodes.py`)
5. **Model factory & provider switching** (`llm/factory.py`)

6. **Retrieval pipeline** (`tools/vectorstore.py`, `tools/search_tool.py`)
7. **Telegram UX / conversation states** (`chatbot.py`, `states.py`)

6.2 Configuration Adaptation (config Folder)

The `config/` folder is the declarative control layer. It enables changes without touching executable code and is the preferred place to start. `factory.py` fetches the models automatically.

6.2.1 `config/configs.yaml` - Global Runtime Settings

Purpose:

- Default models (LLM + embeddings)

How to adapt: Model swap:

```
llm_model: llama3.1:8b-instruct-q4_K_M
embedding_model: mxbai-embed-large
```

6.2.2 `config/agents.yaml` - Agent Roles, Persona, Constraints

Purpose:

- Defines each agent's identity and behavioural constraints through a system prompt or role prompt.

Typical agents:

- Researcher
- Editor
- Writer
- FactChecker
- Polisher

How to adapt: Example: Introduce citation style instructions (Polisher):

```
polisher:
  system_prompt: |
    If sources are provided in the briefing, add inline citations like [1], [2].
    Do not invent sources.
```

6.2.3 `config/tasks.yaml` - Step Outputs & Deliverables

Purpose:

- Specifies what each step must produce (structure, checklists, formatting constraints).

How to adapt: Make the editor output a strict outline format:

```
editor_task:
  description: |
    Create an outline with numbered H2 sections and bullet points per section.
    Include supporting evidence placeholders.
```

6.3 Workflow Adaptation (LangGraph)

The workflow is defined in:

- `graph/workflow.py`

Nodes are processing steps, edges define execution order. LangGraph enables conditional routing (revision loops).

6.3.1 Modifying the agent flow

How to adapt: Add a new node (example: SEO step after polishing):

```
workflow.add_node("seo", seo_node)
workflow.add_edge("polisher", "seo")
workflow.add_edge("seo", END)
```

Remove a node:

- delete `add_node(...)`
- reconnect the edges so the graph remains connected

6.3.2 Changing revision logic

The revision loop is controlled by conditional edges from the FactChecker node.
Possible changes:

- Increase max revision count
- Adjust PASS/FAIL criteria

How to adapt: Increase revision count:

- Identify where `revision_count` is incremented and compared.

6.4 Agent Behaviour Adaptation

Agent logic is implemented in:

- `graph/nodes.py`

Each node encapsulates one role:

- `research_node`
- `editor_node`
- `writer_node`
- `fact_check_node`
- `polisher_node`

6.4.1 The shared state (how information flows)

LangGraph nodes typically:

- read from `state` (inputs produced by earlier nodes)
- write new keys back into `state`

How to adapt: Add a new state field (example: `citations`):

1. In `research_node`, collect citations and store:
   - `state["citations"] = ...`
2. Update downstream prompts to include citations.
3. In `polisher_node`, enforce citation formatting.

6.4.2 Writer improvements (how rewrites are implemented)

Typical rewrite pattern:

- include `critique` and `previous_draft` in the prompt
- instruct the writer to apply only necessary edits

How to adapt: Add 'diff-style' rewrite discipline:

- instruct the writer to keep sections unchanged unless flagged
- or to produce a short change log

6.5 Model Adaptation (`llm/factory.py`)

Model configuration is handled via:

- `llm/factory.py`

Purpose:

- Role-specific specialization (logic vs creative)

6.5.1 Changing LLM models (Ollama)

Download new model:

`ollama pull <model_name>`

Then change it in `config/configs.yaml`

How to adapt:

- change `temperature` for the agents flexiblity
- change `keep_alive` to controll, how long the model stays loaded in memory

6.5.2 Using non-Ollama models

To switch providers:

1. Replace `Ollama` initialization with the desired provider wrapper
2. Ensure output formats remain compatible with downstream nodes

6.6 RAG Adaptation

Retrieval components are defined in:

- `tools/vectorstore.py`
- `tools/search_tool.py`

6.6.1 Vector store adaptation (Chroma)

Typical parameters to tune:

- `chunk_size`
- `chunk_overlap`
- top-k retrieval

How to adapt:

- Smaller chunks improve precision for factual queries
- Larger chunks improve coherence for long-form writing

A common tuning path:

1. Increase overlap if key facts get split
2. Increase `k` if answers miss important context

6.6.2 Web search adaptation

The search tool typically:

- queries a search provider (DuckDuckGo)
- filters results by domain / filetype

How to adapt:

- Extend blacklist/whitelist
- prefer authoritative domains (e.g., `.edu`, official docs)
- add a 'recency' bias if the topic is time-sensitive

6.7 Telegram Chatbot Adaptation

Conversation logic is defined in:

- `chatbot.py`

To adapt interaction flow:

1. Modify conversation states
2. Adjust question sequence
3. Update handlers and transitions

How to adapt:

- Add new configuration parameters by:
  - adding a new state (`states.py`)
  - updating the state machine transitions
  - passing the new field into the workflow inputs
- Change UI style:
  - replace free-text input with inline buttons

7 Fazit und Ausblick

Die vorliegende Bachelorarbeit befasste sich mit der Konzeption, Neuentwicklung und Evaluierung eines agentenbasierten Chatbots zur automatisierten Erstellung von Blogposts. Ausgangspunkt der Arbeit war die genaue Gap-Analyse eines bestehenden Prototyps, der auf dem Framework CrewAI basierte. Es zeigte sich vor allem durch Intransparenz, Ineffizienz und eine hohe Anfälligkeit für Halluzinationen, dass derart abstrahierte Frameworks für deterministische und fehlerresistente Aufgabenfolgen weniger geeignet sind, als ein System, welches auf einer Logik durch Graphen definiert ist.

7.1 Zusammenfassung der Ergebnisse

Der Wechsel zu der graphenbasierten Architektur von LangGraph und das neu gestaltete Telegram-Interface markieren den zentralen Erfolg dieses Projekts. Durch das explizite State-Management und die Einführung eines zyklischen *Fact-Checker-Loops* (Self-Reflection) konnte das Systemverhalten signifikant stabilisiert werden. Die Auswertung der definierten Testreihen belegt, dass die neue Architektur schneller, genauer und transparenter arbeitet.

Die UX wurde durch das neu konzipierte Telegram-Frontend auf ein professionelles Niveau gehoben. Der Ersatz freier Texteingaben durch klickbare Inline-Keyboards, die Einführung von visuellen Highlights und das Status-Streamings eliminierten die kognitive Überlastung während der Textgenerierung vollständig. Das resultierende System ist somit nicht nur im Backend robuster, sondern auch im Frontend intuitiv und nutzerzentriert bedienbar.

7.2 Ausblick

Obgleich das entwickelte Multi-Agenten-System die gesetzten Ziele vollumfänglich erfüllt, bietet die gewählte modulare Architektur eine ideale Grundlage für zukünftige Erweiterungen.

Ein möglicher Ansatzpunkt für die Weiterentwicklung ist die Integration von *Human-in-the-Loop*-Mechanismen (HITL). Da LangGraph den Systemzustand nach jedem Knotenpunkt pausieren und speichern kann, ließe sich das Telegram-Frontend dahingehend erweitern, dass der Nutzer selbst eine Rolle während des Workflows spielt und beispielsweise das generierte Outline des *Editor-Agenten* manuell absegnet oder per Chat-Nachricht korrigiert, bevor der *Writer-Agent* den finalen Text ausformuliert.

Zudem birgt die *LLM-Factory* das Potenzial für umfassende Modell-Benchmarks. Da die Logik komplett entkoppelt ist, könnten in zukünftigen Arbeiten gezielt kleinere, quantisierte Open-Source-Modelle gegen Giganten (wie GPT-4o) im direkten Zusammenspiel der Agenten-Rollen evaluiert werden.

Zusammenfassend demonstriert diese Arbeit, dass die erfolgreiche Implementierung von Generativer KI nicht allein von der Wahl des größten Sprachmodells abhängt, sondern maßgeblich von einer transparenten, durchdachten Softwarearchitektur und einer konsequenten Ausrichtung am Endanwender.

Literatur

[1] Amershi, S., Weld, D., Vorvoreanu, M., Fourney, A., Beshi, B., Suh, P., Iqbal, S., Bennett, P. N., Inkpen, K., Teevan, J., et al. Guidelines for human-ai interaction. In *Proceedings of the 2019 CHI conference on human factors in computing systems* (2019), pp. 1–13.

[2] CrewAI. Crewai: Framework for orchestrating role-playing, autonomous ai agents. `https://docs.crewai.com/`, 2024. Zugegriffen: 24. Februar 2026.

[3] Følstad, A., and Brandtzæg, P. B. Chatbots and the new world of hci. *interactions 24*, 4 (2017), 38–42.

[4] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., and Wang, H. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023).

[5] LangChain. Langgraph: Build resilient language agents as graphs. `https://langchain-ai.github.io/langgraph/`, 2024. Zugegriffen: 22. Februar 2026.

[6] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems 33* (2020), 9459–9474.

[7] Luger, E., and Sellen, A. "like having a really bad pa: The gulf between user expectation and experience of conversational agents. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (2016), ACM, pp. 5286–5297.

[8] Peng, S., Kalliamvakou, E., Cihon, P., and Demirer, M. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

[9] Rodenbäck, M. Baragmaschatbot-2.0-langgraph. `https://github.com/Marrody/BaRagmasChatbot-2.0-LangGraph`, 2026. Zugegriffen: 26. Februar 2026.

[10] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems 36* (2024).

[11] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems 35* (2022), 24824–24837.

[12] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, H., Liu, J., et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155* (2023).

[13] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).