Generative KI und Agentensysteme

HOTEL CONCIERGE CHATBOT

Kendric Scoles, Ivan Seidel,
Umut Can Özcelik

FHNW Olten
Bachelor Business AI, 3. Sem.
HS2025

Table of Contents

Einleitung

Der Hotel Concierge Chatbot ist ein KI-gestützter Chatbot, der speziell für das Hotel Les Trois Rois in Basel entwickelt wurde. Er soll Hotelgästen rund um die Uhr bei allgemeinen Fragen weiterhelfen. Zum Beispiel zu den angebotenen Dienstleistungen, Sehenswürdigkeiten in Basel oder zur Orientierung in der Stadt. Der Chatbot kann auch Auskünfte zu nahegelegenen Grossstädten wie Zürich oder Paris geben, die direkt mit der SBB erreichbar sind.

Technisch basiert das Projekt auf einem RAG. Das bedeutet, dass der Chatbot seine Antworten nicht nur aus einem trainierten Sprachmodell generiert, sondern zusätzlich auf externe Quellen wie Hotelinformationen oder PDF-Dokumente mit relevanten Informationen zugreift. Dadurch kann er präzisere und aktuellere Antworten liefern, die zum Kontext des Hotels und der Umgebung passen.

Hier ist noch ein Hinweis zu den RAG-Tools, und zwar die Fallback-Suche: Damit der Chatbot auch bei fehlenden oder unvollständigen Dokumenteninformationen weiter zuverlässige Antworten liefern kann, haben wir zusätzliche Tools als Fallback integriert.

- DuckDuckGo Search ist für allgemeine Websuchen, um aktuelle oder externe Informationen nachzuladen.
- WetterMeteo API ist für lokale Wetterabfragen oder wenn wetterbezogene Fragen gestellt werden.

Diese Tools können in der RAG-Pipeline oder im Tool-Handling-Teil des Projekts konfiguriert werden, um automatisch angesprochen zu werden, wenn keine passenden Informationen aus der Wissensdatenbank gefunden werden.

Für die Umsetzung verwenden wir LangChain als Framework, um die verschiedenen Komponenten miteinander zu verbinden. Als Sprachmodell kommt die Cerebras Inference API zum Einsatz, die leistungsfähige LLMs zur Verfügung stellt. Als einfache Benutzeroberfläche dient Gradio, über das Gäste direkt mit dem Chatbot interagieren können.

Ein weiterer Bestandteil des Projekts ist das Safeguarding. Dieses stellt sicher, dass der Chatbot nur angemessene und thematisch passende Antworten gibt. Dadurch bleibt die Kommunikation freundlich, sicher und professionell.

Insgesamt zeigt unser Projekt, wie generative KI praktisch eingesetzt werden kann, um den Gästeservice zu verbessern und gleichzeitig das Hotelpersonal zu entlasten. Der Hotel Concierge Chatbot ist ein gutes Beispiel dafür, wie moderne KI-Technologien sinnvoll in einem realen Umfeld wie der Hotellerie integriert werden können.

Problemstellung

Wir haben diesen Use Case ausgewählt, weil wir alle von unserer Gruppe schon einmal in mindestens einem Hotel gewesen sind und festgestellt haben, dass es am Empfang meistens keinen Chatbot hatte. Wenn keine Person anwesend war, ergaben sich folgende Situationen:

Situation 1: Eine Person kommt ins Hotel und es ist niemand da. Am Schalter steht entweder ein Schild mit der Aufschrift: „Es wird gleich jemand für Sie da sein“ oder ein Schild mit einer ähnlichen Formulierung wie „Der Concierge/die Rezeptionistin ist gerade abwesend. Rufen Sie unter folgender Nummer an, dann wird gleich jemand für Sie da sein“.

Situation 2: Eine Person kommt ins Hotel, aber es ist niemand da und auch keine weiteren Informationen zur Verfügung. Nur die Zeiten, in denen das Hotel betreut ist, sind bekannt. Das kann in kleinen Hotels mit weniger Sternen vorkommen.

In beiden Situationen müssen die Hotelgäste warten. Und da kommen wir mit unserem Hotel Concierge Chatbot ins Spiel, der die Wartezeit überbrückt und die gängigsten Fragen beantworten kann. So erhalten die Gäste Antworten auf ihre Fragen und haben etwas mehr Geduld, bis der Concierge zurückgekehrt ist.
Auch bereits eingecheckte Gäste profitieren von dem Chatbot. Wenn sie zum Beispiel schon vor Ort sind und die Stadt erkunden möchten, aber noch nicht wissen wohin, können sie den Chatbot fragen. Er wird ihnen auch diese Frage beantworten können.

Technische Architektur

Die Anwendung folgt einem einfachen End-to-End-Pfad. Die Gradio-UI nimmt eine Nutzereingabe entgegen, übergibt die Frage an die RAG-Antwortfunktion und liefert die Antwort an die UI zurück., falls der RAG-Pfad keine brauchbaren Informationen liefert, greift ein Tool-Fallback. Konkret ruft die UI-Funktion ‘ask(q)’ zunächst ‘answer_with_llm(q)’ auf und prüft die Antwortqualität. Wenn die Antwort leer oder als nicht ausreichend gestuft wird, wird ‘answer_with_tools(q)’ aktiviert, sodass Wetter, Web-Suche, Wikipedia oder Währungsumrechnung genutzt werden können. Diese Logik ist in ‘ui.py’ implementiert und sorgt dafür, dass der Nutzer entweder eine kontextbasierte oder eine durch Tools gestützte Antwort erhält.

Der RAG-Kern läuft in ‘rag_basic.py’. Das System lädt lokale Dateien (PDF,HTML,Markdown/TXT), segmentiert sie in Chunks und baut/lädt eine persistente FAISS-Vektorendatenbank, wobei über Änderungszeitpunkte entschieden wird, ob ein vorhandener Index wiederverwendet oder neu erstellt werden muss. Die Embeddings stammen aus ‘HuggingFaceEmbeddings’ mit dem Standardmodell ‘sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2’. Für den Chat Call wird eine OpenAI-kompatible Client Instanz mit Cerebras-Basis-URL und API Key verwendet.

Als Fallback ist ein schlanker Tool-Agent hinterlegt, der über die Cerebras-kompatible ‘ChatOpenAI’-Schnittstelle läuft und bei Bedarf Tools wie z.B. ‘DuckDuckGoSearchResults’ oder ‘OpenMeteo API’ aufruft. Der Agent ist streng auf überprüfbare Informationen ausgerichtet, soll nichts erfinden und antwortet standardmässig präzise auf Deutsch.

Für Debug und Demonstrationszwecke existierte ursprünglich eine zweite Oberfläche ‘ui_rag_only.py’, die zusätzlich den verwendeten Quellen aus dem Retrieval anzeigte. Dieses Feature wurde jedoch in der finalen Version auskommentiert (# vor die Code-Zeilen), da das System inzwischen stabil arbeitet und die Hauptoberfläche bewusst schlank gehalten wurde.

Implementierung und RAG

Die wichtigsten Konfigurationsparameter, einschliesslich Modellnamen, API-Basen, Daten- und Indexpfade sowie Chunking-Werte, werden in ‘rag_basic.py’ gesetzt. Standardmässig werden eine Chunk-Grösse von 700 Zeichen und eine Overlap von 120 Zeichen verwendet. Diese Parameter steuern die Textaufbereitung für die Embedding-Erstellung und das Retrieval und werden aus Umgebungsvariabeln befüllt, falls vorhanden.

Das Laden der lokalen Quellen ist robust gegen unterschiedliche Formate. Für PDFs wird zuerst der ‘PyMuPDFLoader’ versucht und bei Fehlern auf den ‘PyPDFLoader’ zurückgegriffen. HTML wird mit dem ‘BSHTMLLoader’ geladen, Text/Markdown mit dem ‘TextLoader’. Danach erzeugt der ‘RecursiveCharacterTextSplitter’ die Chunks aus denen ‘FAISS.from_documents(...)’ den Vektorindex erstellt oder, falls vorhanden und aktuell, wieder lädt und verwendet

Der Retrieval analysiert jede Nutzerfrage und erweiter automatisch um passende Synonyme. Anschliessend durchsucht das System das Vektorindex mit ‘similarity_search’ und bewertet die gefundenen Textstellen. Wenn keine passenden Treffer gefunden werden, führt das System einen zusätzlichen ‘Deep-Scan’ durch, der mithilfe von Mustern wie Zeitangaben oder Keywords dennoch verwertbare Textausschnitte aus den lokalen Dateien identifiziert.

Die eigentliche Antworterzeugung baut aus den abgerufenen Chunks einen kompakten Kontextstring, übergibt ihn an ‘_chat(...)’ mit einem RAG-Only-Systemprompt und erhält so eine Antwort, die strikt auf lokalen Kontext beschränkt ist. Nur wenn kein Kontext und kein Deep-Scan verwertbar sind, wird der Tool-Fallback aufgerufen. Somit stellt der Code sicher, dass lokale Unterlagen vorrangig verwendet werden und Halluzinationen minimiert bleiben.

Das Projekt arbeitet explizit lokalen Hotel- und Basel-Quellen in dem ‘data/’ Ordner. Dazu zählen unter anderem Factsheets zum Hotel sowie ein Basel-Transport-Factsheet mit Knotenpunkten, typischen Linien und SBB-Fernverkehrsfrequenzen. Diese Inhalte sind so gewählt, dass typische Concierge-Fragen wie Wege vom Hotel zur SBB und Hotel bezogene Fragen zuverlässig beantwortet werden können.

Evaluation & Testing

Für die Funktionsprüfung stellt ‘rag_basic.py’ eine kleine Diagnosefunktion bereit, die unter anderem die Anzahl der Index-Vektoren und die geladenen Quelldateinamen ausgibt. Anhand dieser Ausgabe lässt sich unmittelbar verifizieren, ob alle relevanten Dokumente eingebunden sind und der Index befüllt wurde. Ein sinnvoller ‘Smoke-Testablauf’ ist daher: Indexstatus prüfen, eine zentrale Hotelfrage stellen und kontrollieren, dass die Antwort inhaltlich mit den lokalen Dokumenten übereinstimmt, anschliessen eine Wegbeschreibung Hotel -> Sbb testen und schliesslich eine Zugfrequenzfrage prüfen. Damit deckt man die Kernpfade, lokaler Hotelkontext, ÖV-Kontext und SBB-Zeitlagen ab.

Zur Bewertung der Systemleistung werden zwei praxisnahe Kennzahlen verwendet. Die erste ist die ‘Context-Hit-Rate’, die angibt, wie häufig der Chatbot seine Antworten direkt aus den lokal gespeicherten Informationsdokumenten bezieht. Die zweite Kennzahl ist die ‘Fallback-Rate’, die zeigt, wie oft der Chatbot auf externe Tools wie Websuche oder Wetterabfragen zurückgreifen musste, weil im eigenen Wissensspeicher keine passenden Informationen gefunden wurden. Diese beiden werte ermöglichen eine einfache Einschätzung, wie zuverlässig das Retrieval funktioniert und in welchen Fällen externe Datenquellen notwendig sind.

Abschliessend ist erwähnenswert, dass der Tool-Agent selbst enge Leitplanken nutzt, um keine unbelegten Aussagen zu produzieren. Dadurch bleibt die Gesamtausgabe des Systems auch im Fallbackfall faktenorientiert und knapp.

Tracing & Performanceanalyse

Für eine gezielte Analyse des Systemverhaltens und der Modellleistung kann optional ein Tracing-System aktiviert werden. Die Implementierung nutzt dabei LangFuse, ein Framework zur Nachverfolgung von modellaufrufen, Prompts und Latency. Wenn die zugehörigen Umgebungsvariablen (LANGFUSE_PUBLIC_KEY, LANGFUSE_SECRET_KEY und LANGFUSE_HOST) in der .env-Datei gesetzt sind, wird das Tracing automatisch beim Start der Applikation initialisiert. Dies geschieht innerhalb der Funktion '(ask_once()' in 'chain_basic.py', wo das Framework die gesamte Prompt-Pipeline protokolliert, einschliesslich Eingabefrage, erzeugtem Prompt, Modellantwort und Latenzzeit.

Das Tracing liefert dadurch eine transparente Sicht auf die Leistung des RAG-Systems und der Cerebras-Schnittstelle. Entwickler können damit nachvollziehen, wie lange einzelne Schritte dauern, wie viel Kontext pro Anfrage verarbeitet wird und welche Prompt Struktur zu besonders präzisen Antworten führt. Diese Metriken helfen bei der Optimierung der Antwortzeiten, der Reduktion von Tokenverbrauch und der Verbesserung der Gesamtqualität der generierten Antworten.

Herausforderungen & Learnings

- Setup Probleme (API-Keys, LangChain Versionen, Local Host vs Deepnote ban)
    - Die API-Keys funtionierten, aber es kam bei hoher User-Usage (unserer und von anderen Personen) zu Verzögerung seitens Cerebras und unser Chatbot brauchte einen kurzen Moment, um einen Output zurückzugeben.
    - Es gab manchmal Herausforderungen beim Einrichten, denn bei jedem Laptop/Computer mussten am Anfang die richtigen Versionen installiert werden. Das führte manchmal zu Versionskonflikten zweischen Zum Beispiel bei LangChain und Gradio.
    - Der Hocalhost ging in seltenen Fällen nicht auf, obschon es funktionieren sollte. Als Lösung schlossen wir das Programm und öffneten es erneut.
    - Einmal haben wir aus Versehen Gradio im Deepnote gestartet und wir wurden für einen Tag gebannt. Von diesem Ereignis haben wir gelernt, dass Gradio nur Lokal ausgeführt werden soll, weil Deepnote Gradio nicht akzeptiert.
- Debugging
    - Debugging nahm eine hohe Zeit in Anspruch. Es gab aber keine nennenswerten Herausforderungen.
- Teamorganisation & Kommunikation

    - Da nicht alle Teammitglieder einander kannten, wussten wir anfangs nicht, wer welche Fähigkeiten hatte und konnten dies auch nicht direkt kommunizieren. So kam es manchmal zu Zeitverzögerungen vom Projekt.

Fazit & Ausblick

- Der Chatbot funktioniert, auch wenn es lange gedauert hat, ihn zum Laufen zu bringen. Wir sind mit dem Ergebnis zufrieden und können uns vorstellen, ihn bei zukünftigen Chatbot-Projekten als Vorlage zu verwenden.

Quellen & Anhänge

- https://www.cerebras.ai/, https://cloud.cerebras.ai/platform/.
  Cerebras ist unser API Key Provider.
- https://pypi.org/project/langchain-huggingface/
- gradio: Ist eine Python Library. Ein Rahmenwerk für die Erstellung webbasierter Schnittstellen für maschinelles Lernen. Für unser Interface verwendet.
- Langchain und gradio werden durch pip install -r requirements.txt heruntergeladen.
- Link GitHub Projekt: https://github.com/kendricscoles/hotel-concierge-bot/tree/main&
- Link Deepnote Projekt: https://deepnote.com/workspace/Fachhochschule-Nordwestschweiz-Bachelor-Business-AI-8a7f9eed-981d-4902-9a28-e7a9a4e7820c/project/hotelconciergechatbot-862a8d37-62e5-4794-9ec2-a7e0c044f3d8/notebook/4b98b76ba00f4a409e2be151b33e4cd2
- Screenshots (UI, Konsole, LangSmith Traces