Bachelorarbeit

Build a Multi-Agent approach for a Large Language Model (LLM) to improve responses and by using Retrieval-Augmented Generation (RAG) to consider own data in queries

Verfasserin

Anna Hansl

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2025

Studienkennzahl lt. Studienblatt:	A 033 526
Fachrichtung:	Wirtschaftsinformatik
Betreuerin / Betreuer:	Dipl.-Ing. Dr.techn. Marian Lux

Contents

Abstract

This thesis presents a local, open-source, multi-agent blog post creation chatbot integrating Retrieval-Augmented Generation (RAG) to enhance response accuracy. Built with CrewAI and Ollama, the system runs almost entirely offline while remaining adaptable to various use cases.

A specialized agent team—Researcher, Editor, Writer, and Proofreader—collaborates to generate high-quality blog posts, supported by a fact-checking tool to reduce misinformation. Evaluation was done using a list of specific requirements to be met and a list of challenging scenarios to be faced, which shows improved factual consistency and adaptability, though challenges remain in speed optimization and handling complex queries.

This research highlights the potential of multi-agent RAG systems to improve chatbot reliability and output quality while maintaining local execution and transparency, specifically using the framework CrewAI.

1 Motivation

When ChatGPT was released to the public in 2022, the interest in large language models and chatbots went through the roof[1]. Within months, most major tech firms had their own tools and developed their own models. Nowadays, the market on online large language model tools is huge, with hundreds of models all trained to do different things being publicly available. But, one thing the market is not oversaturated with is local models, and tools that use them. So, for this Bachelor thesis project, a local, open-source, multi-agent blog post creation chatbot that can easily be extended to a new use case and uses retrieval-augmented generation (RAG) was created.

The problem most online tools share is that they are a) online, and b) often require users to create an account to use them. The fact that they are only available online means that cannot be used when the user is not connected to the internet, which can make whole workflows and tools useless when the user is offline. Many users also dislike that they are forced to create accounts because they want to stay anonymous, which these tools do not allow[2].

There are a few open-source locally executable models that can be downloaded via Ollama, but they often are just that: large language models. They often suffer from hallucinations because the models have to fit on household computers for users to really be able to use them, forcing the used data sets to be very small. They also do not allow users to use retrieval-augmented generation, which is an AI framework that allows users to include external knowledge sources with information that is used for output generation, which helps against misinformation[3].

[1] ChatGPT, *ChatGPT*. URL: `https://chatgpt.com/`, Last accessed 22 February 2025.

[2] Wiktor Kowalski, *Why Using Local LLMs Might Be Better Than ChatGPT, Claude, or Gemini: A Simple Cost Breakdown*. URL: `https://scand.com/company/blog/local-llms-vs-chatgpt-cost-comparison/`, Last accessed 22 February 2025.

[3] Wiktor Kowalski, *Why Using Local LLMs Might Be Better Than ChatGPT, Claude,*

The created tool solves both of these problems, as it is both a local, open-source tool that can be locally executed and includes the two architectural LLM approaches of retrieval-augmented generation and multi-agent systems. This means that users can download the tool via GitHub, adjust a single value given that they have the required dependencies installed, boot the system, and can start configuring their own blog post articles. They can choose the topic, the language, and much more, and add links to websites or documents as external knowledge sources that are used when the article is created. This is done by a team of agents—a researcher, an editor, a writer and a proofreader—who work together to create the perfect blog article. And, if the user is not satisfied with the blog post creation use case, they can easily adapt it by changing a few lines in the code according to an adaptation manual which is included in this thesis and on GitHub.

To do this, the framework CrewAI was used, which focuses on multi-agent systems, and includes tools to support retrieval-augmented generation. It also allows users to use downloadable, locally executable models. As a user interface, the framework python-telegram-bot was chosen, as this provides the necessary infrastructure to support several users at the same time, provides an interface for different types of documents being sent, and allows everyone who has a Telegram profile to use the tool.

A tool that is very similar to this functionality is LM Studio. It is specifically designed for downloading and running local LLMs, and also allows retrieval-augmented generation. The biggest difference between the two is that this implementation is multi-agent, which LM Studio does not support at the moment. LM Studio also runs entirely offline, which means that it does not support websites as RAG sources, and cannot be extended to support online models like GPT-4[4] or Gemini[5]. It also only supports a relatively small number of LLMs, which is in contrast to this implementation, as CrewAI allows the support of many different models, which can even be mixed[6][7]. LM Studio was tested using a number of prompts, showing this project to be more suitable for this use case.

To measure success, a list of requirements for the chatbot and a list of challenging tasks and scenarios it should be able to fulfill were created. The requirements are that the tool is a chatbot, local, open-source, multi-agent, supports RAG and is easily adaptable. The tasks and scenarios it should be able to withstand are two users being handled at the same time, rewriting a blog post as if it was written from a different perspective, writing a text about a topic that has very little to do with the provided RAG source, several tools being used at the same time and remembering conversation history.

or Gemini: A Simple Cost Breakdown. URL: `https://scand.com/company/blog/local-llms-vs-chatgpt-cost-comparison/`, Last accessed 22 February 2025.

[4] ChatGPT, *ChatGPT.* URL: `https://chatgpt.com/`, Last accessed 22 February 2025.

[5] Google, *Google Gemini.* URL: `https://gemini.google.com/?hl=de`, Last accessed 22 February 2025.

[6] LM Studio, *LM Studio Docs.* URL: `https://lmstudio.ai/docs`, Last accessed 22 February 2025.

[7] CrewAI, *Introduction.* URL: `https://docs.crewai.com/introduction`, Last accessed 22 February 2025.

2 Foundation

In this project two large language model architectural approaches, retrieval-augmented generation (RAG) and multi-agent systems, were used to create a chatbot. To establish a common foundation, a few concepts will be explained in this section.

2.1 Large Language Models (LLM)

Large language models are AI systems that are able to use human language. The most prominent example is GPT-4 by OpenAI, which is a chatbot that has taken over the world since it was published in November 2022[8]. Humans can interact with it like with another human being and it is able to do a lot of tasks outside of mere text creation, like code or solve easy mathematical problems. It does so by always predicting the next word in a sentence, a program, or a calculation[9].

2.1.1 Tokenization & Embeddings

This prediction process is based on tokenization and embeddings, and on the large data sets being used to create it. These data sets are composed of millions of pages of text, which are all tokenized. Tokenization is the process of splitting something into smaller parts, for example into sentences, words or letters[8]. Most advanced models, like GPT-4[10] or the Llama model[11] that is used in the implementation use a form of byte-pair tokenization, as this is language-agnostic, and can also be used to work with things like emojis or languages with no spaces between words, like Japanese or Chinese[12].

These tokens are then embedded, meaning they are mapped to a continuous vector space. These vectors can then be used for calculations, like vector similarity and dissimilarity. When predicting the next word in a sentence, these calculations for similarity are necessary to accurately predict the next word. After embedding, the vectors are processed using the Transformer architecture, which is a technology proposed by Google engineers which understands the relationship between words without understanding the words themselves. So, in a sentence, it can be used to focus attention on single words, which are then weighted more in the vector. For example, if the two sentences "I love AI" and "I hate AI" are compared, the vectors might be similar because 2 out of 3 words are the same, while meaning the complete opposite. With transformers, a lot

[8] ChatGPT, *ChatGPT.* URL: `https://chatgpt.com/`, Last accessed 22 February 2025.

[9] Mark Stevenson, *Large language models: How the AI behind the likes of chatgpt actually works*, 2024. URL: `https://theconversation.com/large-language-models-how-the-ai-behind-the-likes-of-chatgpt-actually-works-244701`, Last accessed 22 February 2025.

[10] ChatGPT, *ChatGPT.* URL: `https://chatgpt.com/`, Last accessed 22 February 2025.

[11] Ollama, *Llama3.1.* URL: `https://ollama.com/library/llama3.1`, Last accessed 22 February 2025.

[12] HuggingFace, *Byte-Pair Encoding tokenization - Hugging Face.* URL: `https://huggingface.co/learn/nlp-course/chapter6/5`, Last accessed 22 February 2025.

of attention can be placed on the verbs, and the vectors are more different[7]. So when a user sends a query, this query is tokenized and embedded, and based on vector similarity and dissimilarity, the next word is predicted, and then the next, and so on.

2.1.2 Quantization

Quantization is a technique used in LLMs to reduce embedding vector size, which are usually stored in large data types like float32. This is done by transforming the existing vectors in float32 to smaller data types, like 8-bit sequences, by mapping them to a smaller vector space. The technique is very useful for reducing memory usage and retrieval time, but reduces the overall performance. An example would be binary quantization, which reduces the f32 values to 1 bit-values. This reduces necessary storage space to 3.125%, so 1/32, and allows 40 times faster retrieval, while only reducing overall retrieval performance to 96%[13].

Another example would be scalar quantization. In scalar quantization the f32 values are reduced to an arbitrary sequence length. For example, int8 scalar quantization is used to convert f32 values to int8 values, reducing memory usage to a fourth and making it much faster. In detail, this quantization works by looking at int8 as 256 levels, (in this case -127 to 128, but is 0 to 255 when uint is used), and looking at the whole dataset of embeddings, finding the minimum and the maximum, and assigning each embedding a level within this, relative to the min and the max. For example, if the maximum value in a dataset is 7 and the minimum is -3, the levels range across this span of 10, and if a value is e.g. 2, so in the middle, it would be situated at 0. This is a great way of saving memory and speeding up retrieval, but its accuracy preservation is relatively low[14]. This is where Q8_0 , a specialized version of int8 scalar quantization, comes into play: it is specifically made for the use case of being used for tasks involving transformers and large language models, and is therefore optimized in this area. In it, an associated scale and offset is used to improve the expressiveness of the quantization, so the quantized information is more recognizable and more easily usable, as the context is not changed/it is only quantized for one specific context/use case. Other quantization methods include q3, q5 and many more, where each allows a different number of levels. For example, in q5_0, the values are transformed into 5-bit integers, which in turn reduces the levels to 32 (-15 to 16 or 0 to 31)[15].

[13] Aamir Shakir, *Binary and Scalar Embedding Quantization for Significantly Faster & Cheaper Retrieval.* URL: `https://huggingface.co/blog/embedding-quantization`, Last accessed 22 February 2025.

[14] Aamir Shakir, *Binary and Scalar Embedding Quantization for Significantly Faster & Cheaper Retrieval.* URL: `https://huggingface.co/blog/embedding-quantization`, Last accessed 22 February 2025.

[15] TheBloke, *TheBloke/Llama-2-13B-chat-GGML · Hugging Face.* URL: `https://huggingface.co/TheBloke/Llama-2-13B-chat-GGML`, Last accessed 22 February 2025.

2.2 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation is a two-phase AI framework used for retrieving facts from external sources, like websites, documents or databases. These external sources are provided by the user, either when the query is placed or when the chatbot is configured, like a chatbot that can use company-specific databases. RAG is made up of two phases, retrieval and generation, which is visualized in Figure 1. Before the retrieval phase, the content from the external source is retrieved, tokenized and embedded. They are first split into larger chunks, as LLMs have limited token windows, and these chunks are then embedded[3].

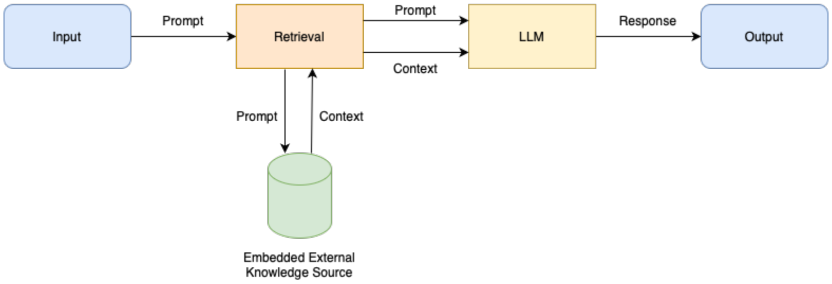


Figure 1: A visualization of the RAG workflow

In the retrieval phase, the user query is used to retrieve all relevant embeddings, so the important chunks, that the LLM might need. For this, it is also embedded, and vector similarities and dissimilarities are used. The embeddings that are relevant for answering the query are returned in the form of context. Before they are returned, they are reranked, so sorted by how useful they are for answering the query. The second phase is the generation phase, where the retrieved context and the original user query are given to the LLM, which then generates an answer. This framework increases answer quality and reduces LLM hallucinations.[3].

2.3 Multi-Agent Systems (MAS)

Multi-Agent Systems are an AI architectural approach that consists of several LLM agents that work together to perform a task. Agents are different LLM instances that all have different sub-tasks that are defined using prompts, and can use different tools to perform these tasks and interact with each other[16]. For example, in this implementation, there are the four agents Researcher, Editor, Writer and Proofreader that all interact with each other. There are many architecture types of MAS that are all different, and these contrasts make them very

[16] Kerem Aydin, *What is an LLM Agent and how does it work?*. URL: https://medium.com/@aydinKerem/what-is-an-llm-agent-and-how-does-it-work-1d4d9e4381ca, Last accessed 22 February 2025.

adaptable to different use cases. The main architectural difference is that they either have a centralized or a decentralized network, so that there is a common knowledge source that all agents share or that the agents just communicate with their neighboring agents. There are also different types of hierarchies, types of communications and much more[17].

2.4 Prompt Engineering

Prompt Engineering is the practice of designing and refining LLM queries to improve performance and output. As LLMs cannot truly understand the meaning of a text, it is important to say exactly what you mean without leaving any room for misunderstanding. When doing that, the LLM can accurately predict the next words based on the provided query and respond with a satisfying answer. To achieve this behavior, several techniques have been developed over the last few years. The one most people unknowingly use is zero-shot prompting, which is just posing a simple query without providing any example, while few-shot prompting includes a few examples. The most popular actual technique is chain-of-thought prompting, where the model is encouraged to think step by step to solve complex tasks, and is very popular in math, logic and problem-solving, but there are many more[9].

3 Related Work

As LLMs are some of the most popular technologies at the moment, many big IT companies, like OpenAI, Meta, Microsoft or Google, have developed models and frameworks. In this project the framework CrewAI[18] was used for the multi-agent system and RAG implementation, and a Llama model[19] was used as a large language model, but many other valid alternatives exist. These will be discussed in this chapter, along with important technological improvements that were needed to bring the technology to where it is today.

The only tool that can really be compared to this implementation is LM Studio, as this combines two of the three big aspects, as it is local and supports RAG. It is a tool that can be used to download LLMs, chat with them, and allows users to send documents as a secondary source. It can be downloaded and used as a normal app on a computer, making its user interface very easy. But, it is not very customizable, does not support multiple agents, and does not allow the user to create tasks and workflows. It also works totally offline, so does not support using online models, cannot accept website links as RAG sources and has no fact-checking included[20].

[17] Anna Gutowska, *What is a multiagent system?*, 2024. URL: `https://www.ibm.com/think/topics/multiagent-system`, Last accessed 22 February 2025.

[18] CrewAI, *Agents.* URL: `https://docs.crewai.com/concepts/agents`, Last accessed 22 February 2025.

[19] Ollama, *Llama3.1.* URL: `https://ollama.com/library/llama3.1`, Last accessed 22 February 2025.

[20] LM Studio, *LM Studio Docs.* URL: `https://lmstudio.ai/docs`, Last accessed 22 Febru-

3.1 Large Language Models

The idea of a large language model has been around for quite some time, but the technology that paved the way for all modern models, Transformers, was only introduced in 2017. Back then, a team of Google AI specialists developed this new understanding of contextualized relationships and used it to revolutionize the world of natural language processing[7]. The next year, a similar change happened when ELMo and BERT where introduced. ELMo introduced embeddings that are dependent on context, so where the meaning of a word changes depending on the rest of the sentence[5], while BERT allowed for bidirectional pretraining, so that the model was able to consider both the left and the right side of a word for context[2]. Then, in 2019 and 2020, OpenAI published papers on GPT-2 and GPT-3, with GPT-2 introducing the idea of LLMs as we know them[6], and GPT-3 first exploring the role a large amount of data plays in improving accuracy[1].

As mentioned above, a Llama model was used, specifically `llama3.1:8b-instruct-q8_0`[21]. This model was downloaded using Ollama, which is a tool for running open-source LLMs locally. Ollama offers many different models, but even more models exist on the internet that can be accessed via APIs or websites. For example, OpenAI offers the GPT models[22], which are the most prominent ones on the market, and the ones that are most used in everyday life. The problem with them is that they cannot be downloaded and therefore also not executed locally, and to use them in a project, an API key is necessary, which means that users need to register and pay money[23]. Another big LLM family is Google's Gemini, which mainly focuses on other input types like images or videos but can also handle text[24]. But, just like GPT, an API key is necessary to use it in a program, and it can also not be run locally[25]. In January 2025, the Chinese company DeepSeek released an open-source downloadable LLM with reportedly much lower training and computing costs while still maintaining a performance similar to Llama2 or GPT-4. But it was trained for programming use cases rather than general ones[26], and also has the problem of censoring certain topics[27], which is why it was not chosen for this project.

The main decision was between different Llama models, as they are all open-

ary 2025.

[21] Ollama, *Llama3.1.* URL: `https://ollama.com/library/llama3.1`, Last accessed 22 February 2025.

[22] ChatGPT, *ChatGPT.* URL: `https://chatgpt.com/`, Last accessed 22 February 2025.

[23] OpenAI, *Pricing.* URL: `https://platform.openai.com/docs/pricing`, Last accessed 22 February 2025.

[24] Google, *Google Gemini.* URL: `https://gemini.google.com/?hl=de`, Last accessed 22 February 2025.

[25] Google, *Get a Gemini API key.* URL: `https://ai.google.dev/gemini-api/docs/api-key?hl=en`, Last accessed 22 February 2025.

[26] DeepSeek, *deepseek.* URL: `https://www.deepseek.com/`, Last accessed 22 February 2025.

[27] Donna Lu, *We tried out DeepSeek. It worked well, until we asked it about Tiananmen Square and Taiwan.* URL: `https://www.theguardian.com/technology/2025/jan/28/we-tried-out-deepseek-it-works-well-until-we-asked-it-about-tiananmen-square-and-taiwan`, Last accessed 22 February 2025.

source and can be downloaded and executed locally. Llama3.2[28] and Llama3.3[29] are not licensed to be used in the EU because of the AI Act and therefore could not be considered for this project[30]. So, the choice was between different Llama3.1 models. Llama3.1 also includes models of size 70B and 405B, while the model used in this implementation is of size 8B. At the time of release, the 405B model was the largest and most powerful open source LLM available on the market, but since it has a size of 243 GB, the smaller alternative which was specifically designed for instruct-interactions was chosen. It has a size of 8.5 GB and includes 8.03 billion parameters[31].

3.2 Multi-Agent System Frameworks

The most prominent alternative to CrewAI is LangGraph by LangChain, a San Francisco based company that focuses on providing a platform and product that enables developers to develop LLM products from scratch. Its product LangGraph was specifically built for managing graph-based agent workflows, therefore being very useful for multi agent systems. It represents the system as a graph, with the nodes being LLM calls, function executions and API calls, and the edges being the flow between those nodes[32].

This system has many advantages, like being a part of the LangChain universe, and therefore easily interacting with other systems and having great community support, and being very flexible and customizable. This framework was not chosen as users need to understand the graphs and the logic flows to be able to adapt them, so, while it may not require extensive programming knowledge, a steep learning curve when using the framework is required. Secondly, including RAG in this framework requires a manual definition of the retrieval logic in the workflow graph, while CrewAI has tools specifically developed for RAG[33][34][35].

But there are several other multi-agent frameworks out there as well. Examples would be OpenAI Swarm, Autogen, Magentic-One, and many more.

[28] Ollama, *llama3.2.* URL: https://ollama.com/library/llama3.2, Last accessed 22 February 2025.

[29] Ollama, *llama3.3.* URL: https://ollama.com/library/llama3.3, Last accessed 22 February 2025.

[30] Maria Stasimioti, *Meta rolls out multimodal llama 3.2 - but not in Europe*, 2024. URL: https://slator.com/meta-rolls-out-multimodal-llama-3-2-but-not-in-europe/, Last accessed 22 February 2025.

[31] Ollama, *Llama3.1.* URL: https://ollama.com/library/llama3.1, Last accessed 22 February 2025.

[32] LangChain, *LangGraph.* URL: https://www.langchain.com/langgraph, Last accessed 22 February 2025.

[33] LangChain, *LangGraph.* URL: https://www.langchain.com/langgraph, Last accessed 22 February 2025.

[34] Mehul Gupta, *Magentic-one, autogen, langgraph, Crewai, or openai swarm: Which Multi-AI Agent Framework is best?.* URL: https://medium.com/data-science-in-your-pocket/magentic-one-autogen-langgraph-crewai-or-openai-swarm-which-multi-ai-agent-framework-is-best-6629d8bd9509, Last accessed 22 February 2025.

[35] CrewAI, *Introduction.* URL: https://docs.crewai.com/introduction, Last accessed 22 February 2025.

OpenAI Swarm is a framework that focuses on being easy to use, even for non-programmers, but is just available with OpenAI LLM models[36]. Autogen is a Microsoft framework that mainly features two agents, the User and the Assistant, but is not very intuitive and complicated to set up, and therefore not easy to adapt[37]. Magentic-One is a newer Microsoft framework that features 5 agents and is more similar to OpenAI Swarm, but it is not very flexible and has very little community support and documentation[38][39].

3.3 Retrieval-Augmented Generation Frameworks

While some of the ideas behind RAG have been around for some time, the paper that revolutionized its usage in LLMs was only released in 2020. It was released by Lewis et al. and introduced the concept of RAG as we know it today, so a system that collects relevant files from a knowledge source for a query and uses the fetched documents to generate an answer[4]. Since 2020, advances have been made in several parts of RAG. These range from enhanced retrieval models like ColBERT, which increase the amount of data the retrieval part can handle, to domain-specific applications, so adapting RAG systems to a specific use case like law or medicine, and finally, developing multi-modal RAG, so applications that can also use input formats like images, videos and audios[40].

Nowadays, there are many online models and tools that support and include RAG, or which can be specialized to a specific use case where an external knowledge source, like a database, is available. For example, OpenAI offers a ChatGPT Retrieval Plugin that can be used to combine the chatbot with a database of information[41]. The three main RAG libraries and frameworks are FARM, a framework from Deepset focusing on transformer-based NLP pipelines[42], Haystack, an end-to-end RAG framework from Deepset[43], and

[36] OpenAI, *OpenAI Swarm.* URL: `https://github.com/openai/swarm`, Last accessed 22 February 2025.

[37] Microsoft, *Autogen.* URL: `https://microsoft.github.io/autogen/dev//index.html`, Last accessed 22 February 2025.

[38] Adam Fourney, Gagan Bansal, Hussein Mozannar, Victor Dibia, Saleema Amershi, *Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks.* URL: `https://www.microsoft.com/en-us/research/articles/magentic-one-a-generalist-multi-agent-system-for-solving-complex-tasks/`, Last accessed 22 February 2025.

[39] Mehul Gupta, *Magentic-one, autogen, langgraph, Crewai, or openai swarm: Which Multi-AI Agent Framework is best?.* URL: `https://medium.com/data-science-in-your-pocket/magentic-one-autogen-langgraph-crewai-or-openai-swarm-which-multi-ai-agent-framework-is-best-6629d8bd9509`, Last accessed 22 February 2025.

[40] Custom AI Studio, *This history of retrieval-augmented generation in 3 minutes...!.* URL: `https://medium.com/@custom_aistudio/this-history-of-retrieval-augmented-generation-in-3-minutes-f7f07073599a`, Last accessed 5 February 2025.

[41] OpenAI, *chatgpt-retrieval-plugin.* URL: `https://github.com/openai/chatgpt-retrieval-plugin`, Last accessed 22 February 2025.

[42] deepset, *FARM.* URL: `https://github.com/deepset-ai/FARM`, Last accessed 22 February 2025.

[43] deepset, *Haystack.* URL: `https://haystack.deepset.ai`, Last accessed 22 February 2025.

REALM, which is a Google toolkit for answering questions with RAG[44][45]. Despite all of these being very valid options, none of them were chosen for this project, as CrewAI already has very good RAG tools and support that can easily be adapted and understood.

4 Architecture

The idea behind the project is to provide a local, open-source, multi-agent chatbot that supports RAG and can easily be adapted to new use cases. For implementing the requirements local and open-source, a Llama LLM was chosen that can be downloaded and executed locally. For the multi-agent and RAG requirement, the framework CrewAI was chosen, which also allows the system to be easily adaptable.

CrewAI is a framework that focuses on multi-agent systems and allows for very easy system creation and configuration. To create a system, a number of agents and tasks need to be created, and can be specified in a YAML file to improve readability.

As a user interface, the python-telegram-bot framework was chosen. Here, users can register a bot using the Telegram bot BotFather, and when they run their bot program, its functionality can be accessed at the registered location on Telegram. This user interface was chosen because it can support several users at the same time, and enables users to send many types of documents without having to consider any uploads or anything. In Figure 2, the relationship between the individual parts is visualized. When starting the program, the chatbot is activated. It can then be accessed via Telegram, where the user can either chat with the default Llama LLM or configure a blog article. When configuring a blog article, they can add links to websites or documents that are then passed to the crew, which is the multi-agent system, in the form of RAG-tools. When the blog post configuration is confirmed, the crew starts with the blog post creation. The crew, including all its agents and tasks, is first created using the config files, in which each agent and task is specified. Then, the input variables are used to execute all tasks, using the RAG tools. In the end, the finished blog article is returned to the chatbot, which in turn sends it to the user.

4.1 Architecture of the Multi-Agent System

One of the three main parts of the system is the multi-agent system, which was implemented using CrewAI. It is essentially made up of agents, tasks, and a file that combines the two.

[44] Ming-Wei Chang, Kelvin Guuk, *REALM: Integrating Retrieval into Language Representation Models.* URL: `https://research.google/blog/realm-integrating-retrieval-into-language-representation-models/`, Last accessed 22 February 2025.

[45] Hazal Simsek, *Compare Top 12 Retrieval Augmented Generation (RAG) Tools / Software.* URL: `https://research.aimultiple.com/retrieval-augmented-generation/`, Last accessed 22 February 2025.

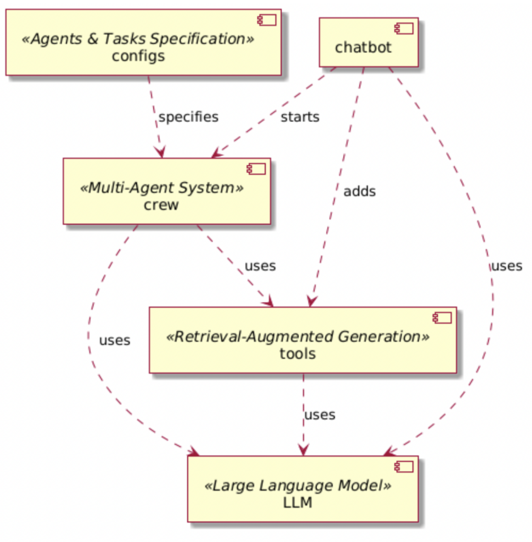


Figure 2: Component Diagram of the Whole System

4.1.1 Agents

For the blog post generation use case, four agents were chosen, each with a different role, goal and backstory. They each perform their tasks in a given order, starting with the Researcher, then the Editor, then the Writer and finally, the Proofreader, as these four roles sum up the people that work on a blog article. Initially, a Factchecker was also considered, but it increased the already long blog article creation time immensely, and therefore its functionality was absorbed by the Proofreader. In the `agents.yaml`, each agent is specified using a role, a goal, and a backstory. For each one, the topic/task the blog article should be about is included in the specification, for example the Researcher's role is `Senior Data Researcher who is an expert on {topic}`, where `{topic}` is the given topic or task. The four agents are described below.

1. **Researcher:** The first agent that is created and used is the Researcher, whose goal it is to uncover cutting-edge developments. If RAG-tools are included, the information provided by them is used, like the text from a website or a PDF document. If they are not provided, it is just the whole knowledge base of the specified LLM, which it can fact-check using the fact-checking tool.

2. **Editor:** The second agent is the Editor, whose goal it is to create a bullet point outline for the article. Like a real editor who gives their writers structures for their article, this Editor also does just that, using the research given to it by the Researcher.

3. **Writer:** The third agent is the Writer, whose goal it is to write blog arti-

cles based on the list of bullet points created by the editor. An important point here is that the writer completely adheres to the given list, without adding any points itself.

4. **Proofreader:** The fourth and final agent is the Proofreader, whose goal it is to proofread and fact-check the article created by the Writer. Its job is to polish the article to perfection, so that the reader has a positive experience, and remove all false statements.

4.1.2 Tasks

Another important part of the CrewAI framework are the tasks, which specify exactly what each agent has to do. They are specified in the `tasks.yaml` and when they are created, each is assigned an agent. The tasks all have a description, an expected output and a corresponding agent. They can either be executed sequentially, so in the order they are defined in, or hierarchically, meaning the task order is assigned based on the tasks themselves. In this implementation, a sequential process was chosen, so the system starts with the research task, then the editor task, and so on. All tasks have specified outputs they should produce, and receive all information created by the tasks before them. For example, the editor task plans each article and returns a list of bullet points, which is then transformed into a blog article by the writer task. The four tasks are described below.

1. **Research task:** The research task corresponds to the Researcher, and its description is that it conducts thorough research on the given topic, paying close attention to finding current information. It returns a list of 10 bullet points including the most relevant information on the topic, which is then given to the editor task.

2. **Editor task:** The editor task corresponds to the Editor, and its description is that it plans and structures a blog article about the given topic, while paying close attention to the list of most important points created by the researcher task. It returns a list of bullet points, each about the topic of a paragraph in the final article.

3. **Writer task:** The writer task corresponds to the Writer, and its description is that it writes a blog article about the given topic, based on the list of bullet points given to it by the editor task. It returns a blog article.

4. **Proofreader task:** The proofreader task corresponds to the Proofreader, and its description is that it proofreads and fact-checks the article created by the writer task, without semantically changing anything (unless the information is incorrect, then it is changed). It returns a corrected blog article, which is then returned to the user.

4.2 Architecture of Retrieval-Augmented Generation

The second big part of the program is the RAG-System. For this, CrewAI offers a list of tools that enable users to add links to websites and documents. How these tools are built and how they work will be explained in the Implementation section. The four tools that can be added are the `WebsiteSearchTool`, the `PDFSearchTool`, the `DOCXSearchTool` and the `TXTSearchTool`. When adapting and extending the project, more tools can be added as well, but these were chosen as they are most commonly used.

- **Website RAG Search** – `WebsiteSearchTool` The `WebsiteSearchTool` first extracts the information from the provided website, for which the link can be either customized when the tool is created or provided during runtime, and then uses the embedding model to vectorize the contents of the website and then gives it to the agent. By default, it is configured with OpenAI, but can easily be customized. In this implementation, for the LLM, `llama3.1:8b-instruct-q8_0` was used, and for the embedding model, `mxbai-embed-large` was chosen. The link to the website is given to the tool upon its creation, before the tool is added to a list of tools after the chatbot user sends a link to a website of their choice.
- **PDF RAG Search** – `PDFSearchTool` The main difference between the `PDFSearchTool` and the `WebsiteSearchTool` is that the information is not extracted from a website, but a provided PDF, which allows users to use this tool offline as well (for the `WebsiteSearchTool` an internet connection is necessary). It is configured with a path to the PDF which leads to the document's place of storage, `Documents`, which is given to the tool upon its creation. The `PDFSearchTool` itself is added to the list of tools mentioned above when a PDF is sent via the Telegram chatbot, which is saved in the `documents` folder within the project, and this new path is then used for creating the tool.
- **DOCX RAG Search** – `DOCXSearchTool` The `DOCXSearchTool` and the `PDFSearchTool` are very similar, the only difference is that the document in question when the `PDFSearchTool` is used is a PDF, and when the `DOCXSearchTool` is used, it is a DOCX, so a Microsoft Word Document.
- **TXT RAG Search** – `TXTSearchTool` Like the `DOCXSearchTool`, the `TXTSearchTool` is very similar to the `PDFSearchTool`, with the only difference being that instead of a DOCX or a PDF, the document in question is a simple TXT file.

4.3 Architecture of the Large Language Model and the Embedding Model

While not being one of the three essential parts of the implementation, the chosen large language model also plays a big role in the implementation. As a

LLM, `llama3.1:8b-instruct-q8_0`, a Llama model, was used. It is an open-source model that can be downloaded via Ollama and executed locally, which was one of the requirements of the project. It is accessed at several places in the implementation, specifically in the chatbot, the crew and the RAG tools. In the chatbot, the LLM is accessed when the user is not configuring the blog post, as this is the default mode. In the crew, all agents need to be configured with a LLM, and the LLM itself is then accessed within the CrewAI framework. Similarly, the RAG tools also need to be configured with a LLM, which is also accessed when the tool is used within the framework.

As an embedding model, `mxbai-embed-large` was chosen. The embedding model is only used in the RAG tools, where it is needed to embed the provided external knowledge source so that the LLM can then use the data.

4.3.1 Chosen Large Language Model

When configuring the agents, a `llm` attribute can be added. This is the LLM that is then used in the framework, and defaults to GPT-4 from OpenAI[46]. As this is a model that is accessed via the internet and requires a token to be used, another option was chosen for this implementation: a local Ollama model, specifically `llama3.1:8b-instruct-q8_0`. It first has to be downloaded, either via the command line, the Ollama website or Hugging Face, and then has to be served. It is then available at the local endpoint "http://localhost:11434", where a user can see the sentence "Ollama is running".

Ollama itself is a tool for running open-sourced LLMs locally, ranging from normal large language models to embedding models and many more. It allows users to, once downloaded, run these models locally, which is a great advantage for creating offline tools and testing. Another great advantage of Ollama is that it downloads model weights, configurations and datasets at once, making the process very easy for users which are new to the world of LLMs. It even offers a WebUI for users that feel less comfortable in the usual command line setting, but it has to be downloaded separately[47].

The model that was used for this implementation is `llama3.1:8b-instruct-q8_0`, which was developed by Meta and was released in July 2024. The quantization used is Q8_0 and within the model itself, a mixture of Q8_0 and F32 is used[48].

4.3.2 Chosen Embedding Model

`mxbai-embed-large` is a tool offered by Mixedbread, a San Francisco-based AI firm focused on producing tools and models for search and retrieval, like `mxbai-embed-large`. `mxbai-embed-large` is one of their three embedding models, and while it was trained as an English embedding model, it has worked

[46] ChatGPT, *ChatGPT*. URL: `https://chatgpt.com/`, Last accessed 22 February 2025.

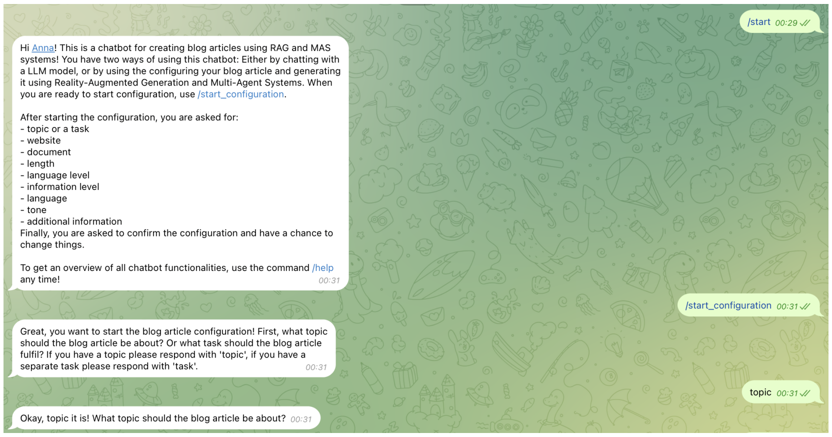
[47] Stephen M. Walker II, *Ollama: Easily run LLMs locally.* URL: `https://klu.ai/glossary/ollama`, Last accessed 22 February 2025.

[48] Ollama, *Llama3.1.* URL: `https://ollama.com/library/llama3.1`, Last accessed 22 February 2025.

on tests with German websites and documents as well. It was trained using a dataset of more than 700 million pairs by using contrastive training, with fine-tuning being done with more than 30 million triplets using the AngIE loss function. The two limitations listed online are the recommended maximum sequence length, as longer sequences may be truncated which could lead to a loss of information, and the language, as the model was exclusively trained on English text. Despite this being a fault listed online, it was not noticeable in the tests, as it also worked well with German[49].

4.4 Architecture of the Telegram Bot

The third big part of the implementation was the Telegram chatbot. For this, the framework python-telegram-bot was used. It can be downloaded via the command line and allows users to specify asynchronous functions that can then be accessed via a Telegram chat. Additionally, a token must be configured by the BotFather-bot. Here, a name and a username for the new bot must be chosen, and then the BotFather returns a token that must be used in the code. For this implementation, the name "RAG-MAS-BlogChatbot" was chosen, with the username being "ragmasbot". So, when a Telegram user wants to chat with the chatbot, they can search for the "ragmasbot" and start a conversation with it.



/start 00:29

Hi Anna! This is a chatbot for creating blog articles using RAG and MAS systems! You have two ways of using this chatbot: Either by chatting with a LLM model, or by using the configuring your blog article and generating it using Reality-Augmented Generation and Multi-Agent Systems. When you are ready to start configuration, use /start_configuration.

After starting the configuration, you are asked for:
- topic or a task
- website
- document
- length
- language level
- information level
- language
- tone
- additional information

Finally, you are asked to confirm the configuration and have a chance to change things.

To get an overview of all chatbot functionalities, use the command /help any time! 00:31

/start_configuration 00:31

Great, you want to start the blog article configuration! First, what topic should the blog article be about? Or what task should the blog article fulfil? If you have a topic please respond with 'topic', if you have a separate task please respond with 'task'. 00:31

topic 00:31

Okay, topic it is! What topic should the blog article be about? 00:31

Figure 3: An exemplary conversation start

After a conversation with the chatbot was started, the user has two choices: either they chat with the Llama model, or they start the blog article configuration by typing `/start_configuration`. Then, they will be asked a series of questions, as can be seen in Figure 3. If the user sends a link or a document, a RAG tool is added to the MAS implementation that the agents can use. After

[49] MixedbreadAI, *Mxbai-embed-large-V1 - docs.* URL: `https://www.mixedbread.ai/docs/embeddings/mxbai-embed-large-v1`, Last accessed 22 February 2025.

answer confirmation, the MAS is created and started and generates the blog article, which is then returned to the user after a few minutes. When this is done, the user can still chat with the local Llama model, which can also change things about the configured blog article, or start the configuration again. Either way, the chatbot remembers the conversation up until the last `/start` or `/clear`, so the already configured articles can be changed or improved.

5 Implementation

The implementation can again be split up into two big python files, a few smaller ones and three YAML files. It can be accessed via GitHub under the link https://github.com/annavalentinakatharina/RAG-MAS-Blog-Chatbot and is public.

5.1 Functionality of the `chatbot.py` file

When the program is started via the command `crewai run`, a Telegram bot is created and started. After the Telegram user types `/start` or, if it is the user's first interaction with the chatbot, just enters the chat, a conversation is started, and the user receives a personalized message explaining the process to them. For the conversation management, a `ConversationHandler` is used, which is provided by the telegram library, and is made up of an `entry_point`, `states` and `fallbacks`. The command `/start` is defined as the `entry_point` of the conversation. After the start, the user's answers are then always filtered, with the default mode being that the user just interacts with the local Llama model like they would with a chatbot on the internet. While doing so, it is in the first state, `CHAT`, and always returns to this. The states command which asynchronous function is being called, which always process the answer and return a state. So, for example, while being in the state `CHAT`, the user's answers are always given directly to the LLM, and the LLM's answers are returned to the user, while the function returns the state `CHAT`. Parallel to the states, the fallback functions can be called. These fallback functions, so `/start_configuration`, `/help`, `/clear`, `/cancel` and `/chat`, can be called by typing the corresponding words, and a specific behavior is started:

- **`/start_configuration`:** The process of the blog post configuration is started.
- **`/help`:** The user receives a list of all possible fallback functions and a general bot explanation. It then returns to the default chatbot interaction mode. This is also mentioned when the user first starts the conversation.
- **`/clear`:** The conversation history is cleared. It then returns to the default chatbot interaction mode.
- **`/cancel`:** The conversation is ended. For the user to interact with the chatbot again, they have to again start the conversation using `/start`.

- `/chat`: While in the blog post configuration mode, users can use this command to return to the default chatbot interaction mode.

All asynchronous functions receive an `Update` and a `Context` as input variables. Both are part of the telegram library and serve different purposes, but together handle when several users use the chatbot at the same time. The `Update` manages the interaction with the Telegram user, so includes the user message and is used to send the reply message to the user. The `Context` provides all conversation information, like the `user_data`, which is where all information that is collected about the user's blog post and the conversation history is saved. So, for example, the topic is saved like this: `context.user_data['topic'] = update.message.text`.

When users start the blog post configuration, they are asked a number of questions. This is managed by the states. In total, there are 13 states defined, `CHAT`, `TOPIC_OR_TASK`, `TOPIC`, `TASK`, `WEBSITE`, `DOCUMENT`, `LENGTH`, `LANGUAGE_LEVEL`, `INFORMATION`, `LANGUAGE`, `TONE`, `ADDITIONAL` and `CONFIRM`.

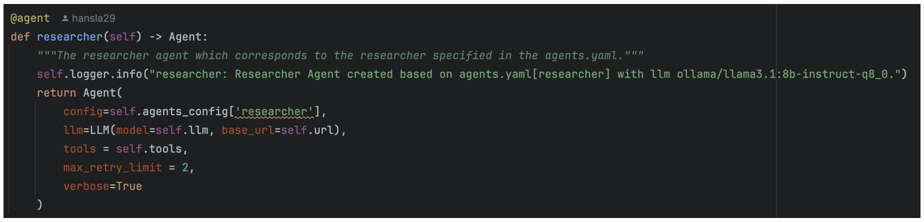
So, for example, in the asynchronous function `/start_configuration`, the user response asks if the user wants a topic or a task as can be seen in Figure 3. The function returns the state `TOPIC_OR_TASK`. Then the user is asked for either a topic or a task, which coincidentally returns the states `TOPIC` and `TASK`. These interactions are repeated until the confirmation is reached, with all inputs being saved in the `context.user_data[]`, except for the links and documents, which are used to create new RAG tools depending on the type of document/link and stored in a list called `tools`. The documents themselves are saved in the directory `documents`.

When the user confirms the data, a new `BaRagmasChatbot` is created, using `tools` as an input variable. It then "kicks off" the crew, which uses the data stored in `context.user_data[]` as input data. After some time, the crew returns a finished blog post and the function returns the state `CHAT`, allowing users to commence their conversation with the default model.

5.2 Functionality of `crew.py`, `agents.yaml` and `tasks.yaml`

The interaction with the framework CrewAI is essentially split into three parts: the `agents.yaml`, the `tasks.yaml`, and the `crew.py`. While the two YAML files are both configuration files and therefore placed in the configs folder in this implementation, the `crew.py` file unwraps them and makes agents out of the specified agents, and tasks out of specified tasks. It is also possible to create the crew without the YAML files by just adding all the information manually to the agents in the `crew.py`, but that would make the code harder to understand, read and maintain.

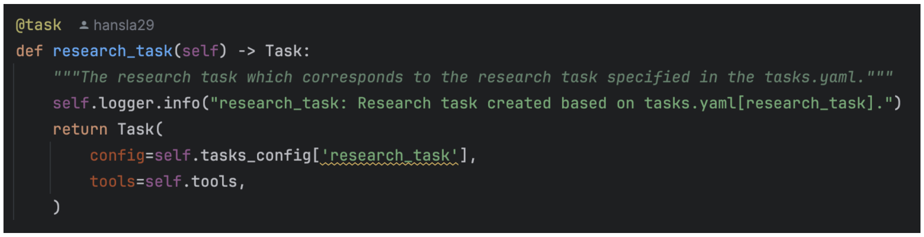
Within the `crew.py` file, the class `BaRagmasChatbot` is defined. Each agent and task is defined and created using a function. In Figure 4, the function `researcher()` is shown, which returns an `Agent`. First, a message is logged, and then the `Agent` is returned. This is defined using the variables `config`, which is where in the `agent.yaml` file the corresponding definition is situated,



```python
@agent  hansla29
def researcher(self) -> Agent:
    """The researcher agent which corresponds to the researcher specified in the agents.yaml."""
    self.logger.info("researcher: Researcher Agent created based on agents.yaml[researcher] with llm ollama/llama3.1:8b-instruct-q8_0.")
    return Agent(
        config=self.agents_config['researcher'],
        llm=LLM(model=self.llm, base_url=self.url),
        tools = self.tools,
        max_retry_limit = 2,
        verbose=True
    )
```

Figure 4: The `researcher()` function

`llm`, which refers to the LLM that the agent should use and because it is an Ollama LLM, the `base_url` also needs to be added, `tools`, which is the list of RAG tools and the `FactCheckTool`, `max_retry_limit`, which is how often the agent's tasks can be retried in case of error before they are aborted, and `verbose`, which allows the agent to write detailed log messages. The tasks are created similarly. In Figure 5, the function `research_task()` is shown, which returns a `Task`. It also starts with a logger message, and then defines where the config is located, and which tools are allowed, which are the ones also given to each agent.

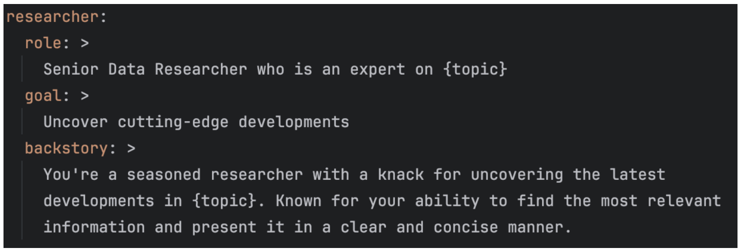


```python
@task  hansla29
def research_task(self) -> Task:
    """The research task which corresponds to the research task specified in the tasks.yaml."""
    self.logger.info("research_task: Research task created based on tasks.yaml[research_task].")
    return Task(
        config=self.tasks_config['research_task'],
        tools=self.tools,
    )
```

Figure 5: The `research_task()` function

In `agents.yaml` and `tasks.yaml`, the agents and tasks are defined. For the configuration of the agents, a number of attributes can be used, as can be seen in Figure 6. They range from `role`, `goal` and `backstory`, which are used in this implementation to options like `max_iterations` (maximum possible iterations), `respect_context_window` (boolean that forces the program to keep under context window size), and `code_execution_mode` (mode for code execution, can be either "safe" or "unsafe"). These are essentially the same as the manually added ones, like `llm`. For the configuration of the tasks, there is also a number of attributes available. For this implementation, `description`, `expected_output` and `agent` were used, but there are many others available, like `output_file`, which defines a file path for storing task output, or `context`, which defines other task results that should be used for this task[50].

[50] CrewAI, *Agents*. URL: `https://docs.crewai.com/concepts/agents`, Last accessed 22 February 2025.



```yaml
researcher:
  role: >
    Senior Data Researcher who is an expert on {topic}
  goal: >
    Uncover cutting-edge developments
  backstory: >
    You're a seasoned researcher with a knack for uncovering the latest
    developments in {topic}. Known for your ability to find the most relevant
    information and present it in a clear and concise manner.
```

Figure 6: The Researcher configuration in `agents.yaml`

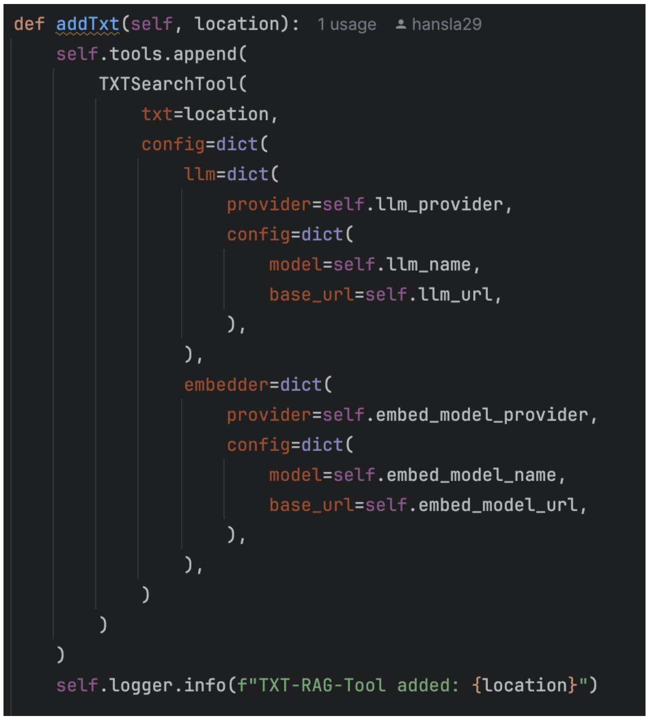
5.3 Functionality of the RAG tools

CrewAI offers a list of tools that manage the RAG-Approach and are very easily configured. Out of a wide range of options, four RAG tools were chosen, specifically the website, the PDF, the DOCX and the TXT RAG tools. To configure either RAG tool, a LLM and an embedding model have to be included. In Figure 7, the function `addTxtTool(location)` can be seen, which adds a `TXTSearchTool` to the tool list. The definition of the tool includes the file `location`, and a `config`, which defines the `llm` and the `embedder`, which are both defined using the models from the `configs.yaml` file. These tools are then used by the agents, which use them when fulfilling their assigned tasks.

Additionally, a CrewAI custom tool was created, which does not really fall under the category of RAG but belongs to the tools and is included in the `tools`. It is a `FactCheckTool` that can be used by all agents and is designed to check facts and stop agents from hallucinating. It works by using the DuckDuckGo-API to search the given sentence and save the texts and sources of the top ten responses. Then, for each sentence, it gives each text and the statement to the Llama model and asks it to respond with yes if the sentence is true according to the text, which returns ”True” if it is. If not, it returns ”False”. Originally, it was planned to also include a source, but as this confused the agents too much and caused the tool to become very slow and the articles to become very short, it was removed.

5.4 Functionality of the rest of the implementation

The configuration file, `configs.yaml`, is used to store information that is generally used throughout the whole implementation but should be easily changeable. This includes the Telegram chatbot token that is created by the BotFather, the LLM that is used by the agents and its corresponding URL, and the LLM and embedding model used within `chatbot.py`, and their corresponding URLs. In the other files, these values are imported as instance variables and used in a file. The reason for this file is that one of the requirements of the implementation is that it should be easy to adapt, and changing the large language model, the chatbot token and the embedding model are where changes are the most common.



```
def addTxt(self, location):
    self.tools.append(
        TXTSearchTool(
            txt=location,
            config=dict(
                llm=dict(
                    provider=self.llm_provider,
                    config=dict(
                        model=self.llm_name,
                        base_url=self.llm_url,
                    ),
                ),
                embedder=dict(
                    provider=self.embed_model_provider,
                    config=dict(
                        model=self.embed_model_name,
                        base_url=self.embed_model_url,
                    ),
                ),
            )
        )
    )
    self.logger.info(f"TXT-RAG-Tool added: {location}")
```

Figure 7: The `addTxtTool(location)` function

Another part of the application is the logger, which is included in each file, and has different levels. It is defined in the `logger_config.py`, and logs to the `app.log` file. Currently, the logger level is `logging.WARN`. It is also included in the tests directory, where a separate `app.log` exists.

6 How to use and adapt the Implementation

This explanation is close to identical to the explanations given at https://github.com/annavalentinakatharina/RAG-MAS-Blog-Chatbot/tree/main[51].

6.1 Using the implementation

Prerequisites:

- Python $\geq$ 3.10 and $\leq$ 3.13
- Pip and virtual environment support

Step by step tutorial:

[51] Anna Hansl, *RAG-MAS-Blog-Chatbot*. URL: `https://github.com/annavalentinakatharina/RAG-MAS-Blog-Chatbot/tree/main`, Last accessed 22 February 2025.

1. Clone this git repository and open the project:
   ```
   git clone https://github.com/annavalentinakatharina/RAG-MAS-Blog-Chatbot.git
   cd RAG-MAS-Blog-Chatbot
   ```
2. Create your .venv and activate it:
   ```
   python3 -m venv .venv
   source .venv/bin/activate
   ```
3. Install the dependencies:
   ```
   pip install -r requirements.txt
   ```
4. Now, download the ollama cli tool:
   ```
   brew install ollama
   ```
   If you're using Linux, `brew` is not available. Alternatively use this command:
   ```
   curl -fsSL https://ollama.com/install.sh | sh
   ```
   A third alternative would be to download the tool via the Ollama website https://ollama.com/download. Either way, the tool has to be started. After running serve, wait a few seconds and then press enter:
   ```
   ollama serve &
   ```
5. The next step is downloading the ollama model and the embedding model (If you want to use a different model, just choose a different model, but additionally go to 6.2.2 How to change your model)
   ```
   ollama pull llama3.1:8b-instruct-q8_0
   ollama pull mxbai-embed-large
   ```
6. Add your Telegram chatbot token into the `src/ba_ragmas_chatbot/config/configs.yaml` file in line 2, `token={your_token}`, replacing `{your_token}`. To receive your Telegram chatbot token, you first need to register your chatbot using the BotFather-bot on Telegram. This also shows you where you can access your chatbot once it is running.
7. To kickstart the chatbot, run this from the root folder:
   ```
   crewai run
   ```
8. When using the chatbot, the bot is started when you first communicate with the chatbot, or, if you have already had a conversation with the bot, using the command `/start`.
9. To stop the chatbot, use `Ctrl + C`

6.2 How to adapt the Implementation to a new Use Case

As there are only so many blogs and blog posts in the world, a blog article generator is not a very broad use case, but one of the objectives of the project was creating an open-source, local and easily adaptable tool. And for this, it is a great starting point and example.

6.2.1 Multi-Agent System Adaptation

To adapt the multi-agent system to a new use case, the files `agents.yaml`, `tasks.yaml` and `crew.py` need to be changed.

1. Go to agents.yaml and replace the existing agents with the new ones, each including a role, a goal and a backstory.
2. Secondly, go to tasks.yaml and replace the existing tasks with new ones, at least one per agent. Each should include a description, an expected_output and an agent.
3. Thirdly, go to crew.py and change the functions that return either an agent or a task, like researcher(). Here, change the name of the function, the text of the logger message and the corresponding YAML agent or task. This has to be done until all tasks are mentioned in the crew.py.
4. Remove all unnecessary agent and task functions.

6.2.2 How to change the model

In a world of change, large language models do not stay the best model on the market very long. So, another thing that has to be easy when adapting the implementation is having the possibility to choose a different large language model and a different embeddings model. This implementation is made with Ollama models, but it is of course also possible to use models from a different provider, like OpenAI or Google.

Switch to a different Ollama LLM First, download the model via the command line, and then go to the file src/ba_ragmas_chatbot/config/configs .yaml. Secondly, go to line 5, llm: ollama/llama3.1:8b-instruct-q8_0. Here, go to ollama/llama3.1:8b -instruct-q8_0 and replace it with new provider and model, e.g. ollama/llama3.2. Thirdly, go to line 10, name: llama3.1:8b-instruct-q8_0. Here, replace the name of the model (e.g. llama3.1:8b-instruct-q8_0 with llama3.2).

Switch to a different Ollama embedding model First, download the model via the command line, then go to the file src/ba_ragmas_chatbot/config/ configs.yaml. Here, go to line 14, name: mxbai-embed-large and replace mxbai-embed-large with your model of choice.

Switch to a non-Ollama model First, do the same steps as with an Ollama model, but also change line 11, provider_name: ollama/line 15, provider_name: ollama and the url-attributes(if they are even needed, maybe they can just be removed). Secondly, go to all the agents in the crew.py and change the llm attribute, where the base_url part needs to be removed.

Thirdly, go to chatbot.py, and go to the functions addWebsite(), addPDF(), addDOCX() and addTxt(). Here, go to the most indented part, where the attributes model and url are situated, and remove the url attribute. Next, if an API key is necessary to access the model, create a file called .env in the root folder, and add the line OPENAI_API_KEY={key}. Add your key instead of {key}. Now, the switch to a non-Ollama model is finished.

6.3 RAG Adaptation

To adapt the RAG part to a new input type, a new tool adding function needs to be implemented in the `chatbot.py`. To see all possible input types, please go to the Tools list on https://docs.crewai.com/introduction[52] and decide on one. To now explain how to add a new input type in detail, let's copy the function `addWebsite(url)`, paste it to the end of the file and adapt it to `addCSV(csv)`, which can be used to add a `CSVSearchTool`. Start by looking at the online documentation, and find out what additional input variable is necessary for this tool. In the case of `CSVSearchTool`, it is `csv='path/to/your/csvfile.csv`[53]. Now, change the following lines in the pasted function `addCSV(csv)`:

1. Line 1: Rename the function to `addCSV` and change `url` to `csv`.
2. Line 3: Replace `WebsiteSearchTool` with `CSVSearchTool`.
3. Line 4: Replace `website` with `csv`, and `url` with `csv`.
4. Line 23: In the log message, replace `Website` with `CSV`, and {`url`} with {`csv`}.

Additionally, the function that receives the input document also needs to be changed. To do this, go to the `VALID_MIME_TYPES` and add the new mime type. Now, go to the function `document()`, and extend the match-case to your new mime type, with the new tool being added as a case.

6.4 Telegram Chatbot Adaptation

To adapt the Telegram chatbot to a new use case, the conversation needs to be changed. To do this, please do the following steps in `chatbot.py`:

- First, go to line 15, remove all unwanted states and add all new states, and change the number at the end to the new amount of states.

- Next, decide on an order the questions should be asked in. When creating the functions in the next task, always ask for the next state at the end and return the next state.

- Next, for all new functions, let's look at the way a function should be changed by first copying `tone()` and pasting it below. Now, let us change the following lines:

  1. Line 1: Change the function name to the new function name, e.g. `confidentiality()`
  2. Line 2: Change the comment description to the new state
  3. Line 7: Change the first word in the log message from `tone` to the new function name
  4. Line 9: Change `tone` to the new state, e.g. `confidentiality`

[52] CrewAI, *Agents*. URL: `https://docs.crewai.com/concepts/agents`, Last accessed 22 February 2025.

[53] CrewAI, *CSV RAG Search*. URL: `https://docs.crewai.com/tools/csvsearchtool`, Last accessed 22 February 2025.

5. Line 10: Change the response to asking for the new state in the chosen order, e.g. `additional information`.
6. Line 12: Change the first word in the log message from `tone` to the new function name
7. Line 13: Change `WEBSITE` to the new next state, e.g. `ADDITIONAL` (information)
8. Line 15: Change the first word in the log message from `tone` to the new function name
9. Line 16: Change `tone` to the new state
10. Line 17: Change the response to asking for the new state in the chosen order, e.g. `additional information`.
11. Line 19: Change the first word in the log message from `tone` to the new function name
12. Line 20: Change `WEBSITE` to the new next state, e.g. `ADDITIONAL` (information)
13. Line 23: Change `tone` to the new state
14. Line 24: Change the first word in the log message from `tone` to the new function name
15. Line 25: Change `TONE` to the new state, e.g. `CONFIDENTIALITY`

- Do this for all new functions and remove the ones that you don't want anymore, but please refrain from changing the logic of `website()`, `document()`, `no_document()`, `start_configuration()` and `confirm()`, as they are essential for chatbot functionality.

- Next, check the function order and make sure that all functions return the correct next state, especially `start_configuration()`, where the first state needs to be called, and `confirm()`, which should be the last function called.

- Now, adapt the function `start_bot`, where the `ConversationHandler` should include just the new states and their corresponding function.

To adapt the fallback functions, change the individual functions, and if you want to add or remove any, change the `fallbacks` in the `ConversationHandler`.

7 Evaluation & Discussion

For evaluating the tool, several requirements and use cases were discussed in the beginning. As it is a tool based on large language models whose nature is that they are not deterministic, simple unit tests or a similar automated form of testing were not an option. So, to measure success, a number of requirements was discussed, along with a list of challenging tasks and scenarios the chatbot should be able to fulfill. All of the agreed requirements were met, as the tool is

1. in the form of a chatbot, as it is a python-telegram-bot

2. local, as a downloadable and locally executable model was chosen

3. open-source, as the code is available on GitHub
4. multi-agent, as a number of agents is used to fulfil the tasks
5. RAG, as CrewAI's RAG tools are used by the agents
6. and can easily be adapted, as easily understandable technology was chosen and an adaptation manual provided here in the text and on GitHub

So, the hard requirements for the tool were met. The task examples were also all fulfilled at least once. They were:

- **Two users being handled at the same time:** As all functions within the chatbot are asynchronous, this requirement was easily met.
- **Rewriting a blog post from a different perspective:** This was tried a few times, and the chatbot was able to complete the task, even with the language from the original blog post and the configured language differing (e.g. English and German).
- **Writing a text about a topic that has very little to do with the provided RAG website or link:** This was also tried a few times, and while the tool was sometimes confused by the fact that the given knowledge hat nothing to do with the chosen topic, it mostly managed to produce good articles. But, sometimes, the resulting blog posts were very short or referred to the external source, stating that the input had very little to do with the query.
- **Several tools being used at the same time:** This was also tried a few times, and while it works well when different tools are in use, the same tool twice with different knowledge sources can cause it to struggle. It has worked well before, but sometimes it just considers the source that seems more relevant or the one that was sent first.
- **Remembering the conversation history:** The tool is able to remember the conversation history and can adapt small parts of the configured articles. For small changes or translations, the default Llama model was mostly used, but it also works if it is given to the multi-agent team again. But in this case, the danger always is that other parts of the article are changed or rewritten, as the agents are given a lot of creative freedom.

Additionally, a **number of unit tests** were created to test the chatbot.py file.

Compared to other existing technologies, the project differs from all in at least one aspect. For example, while online tools like ChatGPT[54] allow users to upload external sources, they mostly are single agent systems and cannot be downloaded and executed locally. Similarly, just the Llama LLM accessed via Ollama might be a very good local model, but it does not enable users to upload external sources and is single-agent, as is mostly the case when using

[54] ChatGPT, *ChatGPT*. URL: `https://chatgpt.com/`, Last accessed 22 February 2025.

local models. So, while the blog post generation tool might not be perfect, it combines many positive aspects that the other tools are lacking.

In comparison, LM Studio was tested a few times with similar queries. It was able to answer most of the prompts correctly, but was way more prone to hallucinations due to the lack of fact-checking. It specifically struggled with only being given a few pieces of information, where this project was able to use other information from online sources using the fact-checking tool and creating complete blog articles. In this case, LM Studio would produce shorter articles. They also had less of a clear structure compared to the ones being produced by this implementation.

The two biggest faults of the tool are that it is very slow and that the Telegram chatbot can be hard to adapt. Its slowness is partly due to the architecture, but mostly because of the model used, which is generally very slow. It also depends on the hardware, and on a laptop like the one it was developed on, which has an above average GPU and CPU but is still not very fast, it takes some time. The Telegram chatbot, on the other hand, is relatively hard to adapt, and requires at least a small understanding of programming. This is because of the conversation flow, which follows a script, and for it to be adapted, the whole file, or at least most of it, needs to be changed. But, one has to remember here that other user interfaces, like a website, might be even harder to adapt, as at least the Telegram chatbot can be adapted very easily if an example is available.

One lesson learned is that the projects based on LLMs can have mixed results because they are not deterministic. So, while a query might lead to a perfect result one time, the next time it could very well return something completely different. Using prompt engineering, these abnormal cases might be decreased, but they can never be completely eradicated. But, this is not a problem that is singular to this implementation, as even OpenAI's ChatGPT has the disclaimer "ChatGPT can make mistakes. Verify important information." on its website[55].

Another lesson learned is that prompt engineering is important, as even the smallest difference in words will allow the implementation to work perfectly or completely ruin it. For example, when building the fact-check tool, the agents tried to use the tool to "check facts" in the sense that they could ask it questions, and it would give them answers. This was because of an unclear tool description. It made the text creation incredibly slow because they would always receive just a "True" or "False" and would try to use the tool several times before learning that it works differently. So, using prompt engineering, the tool description was reworked and the tool fixed so that the agents can understand that the tool "checks facts" in the sense that it verifies them, and returns either "True" or "False".

[55] ChatGPT, *ChatGPT*. URL: `https://chatgpt.com/`, Last accessed 22 February 2025.

8 Conclusion & Future Work

In conclusion, a local, open-source, multi-agent blog post creation tool that offers RAG and is easily extendible was created using the frameworks CrewAI and python-telegram-bot, using an local Ollama model. The chatbot demonstrated effective use of its agents and tools in a number of challenging scenarios and generally produced good blog articles. An important factor to success was the fact-checking tool, which reduced hallucinations as much as possible, along with the RAG tools and the multi-agent system.

8.1 Assumptions & Limitations

To decrease hallucinations and misinformation, the system relies heavily on the external knowledge sources provided by the user and the fact-checking tool. Here, the assumption is that the given sources are true, or at least reflect the information the user wants to have included in their blog post. Another assumption is that CrewAI works correctly. If there was any kind of error in the framework, the chatbot tool would not be able to recognize it and might deliver nonsense. A big assumption is also that the model used provides meaningful answers and does not actively provide misinformation. Especially if the data used for model creation was not correctly checked or deliberately censored, this can be the case.

Limitations of the implementation are that it strongly depends on the used frameworks and libraries, including CrewAI, python-telegram-bot, Ollama, the DuckDuckGo API and many more. If any of these change their functionality at a level basic enough for the chatbot to be affected, the whole program would stop working. Another limitation of the project is, of course, that it is a LLM-based system. The outcomes are always non-deterministic, and cannot be predicted in any way. So, it is prone to hallucinations, and while they are reduced by using the fact-checking tool, they can never be completely eradicated. Another big limitation is the local model. It is not updated automatically, and neither is its knowledge base, so the more time passes, the more outdated the information it provides becomes.

8.2 Future Work

Possible next steps in the project:

- Testing different models and flows, and comparing what difference a different model makes in regard to speed and accuracy.
- A different use case closer to real world RAG applications can also be very interesting, especially if a permanent database is added in the background.
- Improving the fact-checking tool and providing more tools might also be a good idea.

Future work outside of the project:

- Future work may include rebuilding the chatbot with a different multi-agent framework, like LangGraph.

- An interesting thesis project would be a tool that configures the template CrewAI system for the user and returns a zip file including the configured project. The user could be asked questions like what agents they want, their attributes, the tasks and their respective attributes, what questions they want asked in their configuration, and so on. This could then be created by the project and then be downloaded by the user.

References

[1] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. Curran Associates Inc.

[2] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 10 2018.

[3] Gupta, S., Ranjan, R., and Singh, S. A comprehensive survey of retrieval-augmented generation (rag): Evolution, current landscape and future directions, 10 2024.

[4] Patrick Lewis, Ethan Perez, A. P. F. P. V. K. N. G. H. K. M. L. W.-t. Y. T. R. S. R. D. K. Retrieval-augmented generation for knowledge-intensive nlp tasks, 5 2020.

[5] Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations.

[6] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners.

[7] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L., and Polosukhin, I. Attention is all you need.

[8] Webster, J. J., and Kit, C. Tokenization as the initial phase in nlp. In *Proceedings of the 14th Conference on Computational Linguistics - Volume 4* (USA, 1992), COLING '92, Association for Computational Linguistics, p. 1106–1110.

[9] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2022), NIPS '22, Curran Associates Inc.