UNIVERSITÄT ZU LÜBECK

Using AutoGPT for Information Retrieval Agents

Nutzung von AutoGPT für Informations-Recherche Agenten

Masterarbeit

verfasst am
Institut für Informationssysteme

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Jakob Horbank

ausgegeben und betreut von
Prof. Dr. Ralf Möller

mit Unterstützung von
Thomas Asselborn

Lübeck, den 15. April 2024

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Jakob Horbank

Zusammenfassung

In recent years, large language models (LLMs) have showed impressive capablities for text generation tasks. Researchers have started to leverage LLMs to control agents. AutoGPT is such an attempt at creating an autounous goal-oriented system. While there is work that experiments with AutoGPT for web search, local information retrieval (IR) has not been researched yet. This thesis presents an analysis of AutoGPT and a AutoGPT agent that uses retrieval augmented generation (RAG) for IR over local documents. A question and answer (QA) benchmark is generated from a humanities journal to test the agent, and examine its failure points. The thesis concludes, that while using a RAG agent for IR is an interesting approach there is no inherent benefit to static IR systems that use LLMs.

Abstract

In recent years, large language models (LLMs) have showed impressive capablities for text generation tasks. Researchers have started to leverage LLMs to control agents. AutoGPT is such an attempt at creating an autounous goal-oriented system. While there is work that experiments with AutoGPT for web search, local information retrieval (IR) has not been researched yet. This thesis presents an analysis of AutoGPT and a AutoGPT agent that uses retrieval augmented generation (RAG) for IR over local documents. A question and answer (QA) benchmark is generated from a humanities journal to test the agent, and examine its failure points. The thesis concludes, that while using a RAG agent for IR is an interesting approach there is no inherent benefit to static IR systems that use LLMs.

Contents

1

Introduction

For a long time, information retrieval (IR) has been a heavily researched topic in computer science. Retrieving information from a large set of documents is an essential task in many domains. For example, a researcher might want to discover relevant parts of a new article for a given topic. Ideally he can chat with a retrieval system in natural language and receive correct information from the article. In recent years, there has been a large increase in language modeling capabilities. Large language models (LLMs) have been applied to a variety of applications such as general chatbots, writing helpers and coding assistants. There are different approaches to leverage IR for LLMs systems such as retrieval augmented generation (RAG). A RAG system retrieves documents from a database before using them as context for an LLM answer generation. This method enables on-demand inclusion of information into the answer generation. Recently, efforts were increased in using LLMs as agent controllers. In comparison to static systems, agent systems act autonomously and goal-directed. They constantly perceive the environment through sensors and perform actions through actuators. AutoGPT is an open-source project that uses GPT-4 to provide agent actions. While there are experiments with AutoGPT for navigating the web, using it for IR has not been researched yet.

This thesis presents a LLM agent built with AutoGPT, that uses RAG to answer user prompts. The agent is benchmarked with a small set of question and answer (QA) pairs from a humanities journal. A qualitative evaluation of the IR agent benchmark is given in the results.

1.1 Contributions of this Thesis

This thesis has three main contributions:

1. An analysis of AutoGPT in the context of IR.

2. A custom IR agent that uses RAG for IR.
3. A small humanities journal QA benchmark to test the IR agent.

In the analysis, I conclude that to research IR applications for AutoGPT, an adaption of the default abilities for RAG is needed. Due to the complexity of the default agent, I further reason that it is best to build a custom agent with the baseline Forge agent.

The custom IR agent is built with the Forge agent baseline included in the AutoGPT project. A custom agent logic is implemented to define the agent loop. The agent has abilities to ingest a PDF into its memory, retrieve documents for it and use them to answer a user query. Rather than following a fixed pipeline, the agent choose these actions to dynamically perform the RAG steps.

The benchmarking dataset contains eight QA pairs from a humanities journal. I used GPT-4 to generate candidate questions and then handpicked answers from the journal articles. The IR is compared to GPT-3, GPT-4 and Perplexity AI using cosine similarity between the ground truth and generated answers. There is no service that clearly outperforms the other ones. A qualitative evaluation of the agent explains the common failure points. The main problems of the agent are irrelevant retrieval results, inconsistent LLM response formats and choosing wrong abilities.

1.2 Related Work

With the rise of LLMs, they have been introduced into different domains. LLM agents are a promising way to fulfill user goals autonomously, hinting in the direction of real computer intelligence. Language models have also seen rising popularity in IR applications. Many popular search engines have introduced language models to present the retrieved data to the user. In the following, notable work that is of interest for this work is listed.

LLM Agents

After LLMs like GPT-3 and then GPT-4 were made public, researchers started testing different approaches to integrate them into agents. In particular, the introduction of tools to extend the language model capabilities Toolformer [27] trains a language model to use different APIs in a self-supervised way. For example, the model can use a calculator to externalize math problems, a task category that LLMs struggle with. HuggingGPT [28] uses generative pre-trained transformer (GPT) repeatedly in multiple rounds. For each step, a Huggingface model can be selected by the model to answer the prompt. In a virtual social setting, multiagent communication was demonstrated in [21]. Agents living together in a virtual village have different goals and communicate through natural language.

Building on top of these research experiments with tool usage and access to external knowledge, multiple applications of language model agents have been presented. AutoGPT [29] is an attempt to make LLM autonomous. The AutoGPT agent has a variety of abilities such as web search, image generation, code execution and file operations. It uses GPT-4 to generate an action proposal from a list of abilities. An introduction to AutoGPT is given in chapter 3. I will use the included Forge agent to build a custom agent in this thesis.

Besides AutoGPT, other LLM agent projects have been published. BabyAGI is a minimal task-driven autonomous agent [18]. With OpenAI Assistants [2], users can build custom assistants that can use OpenAI models, tools and knowledge to answer user queries.

LLM Information Retrieval

Because of their strong natural language capabilities, many developers work on tools that use these models for IR

Perplexity AI [22] is an online service that leverages a language model to provide a search that generates an answer from different sources on the internet. The content of the answer is then linked to the found sources, so the user can see and verify the result.

1.3 Structure of this Thesis

First, I will introduce the main backgrounds that are relevant to this work in chapter 2. The steps of an IR system are explained, as well as different strategies to evaluate such systems. Furthermore, I will give an overview of the concept of an *agent* and its main components. Additionally, the language modeling section covers the base to understand the current LLMs. I will go from the introduction of the transformer architecture to current chat models like ChatGPT. Finally, the combination of language models and agents to *LLM agents* is introduced. Chapter 3 contains an introduction to AutoGPT and its core elements, as well as an analysis of its current IR capabilities. First, an overview of the AutoGPT project and its components is given. I will then explain the default agent in more detail, looking at how the LLM agent ideas are implemented. After gaining an understanding of the AutoGPT project, we will then look at its default capabilities for IR tasks. In chapter 4, a custom AutoGPT agent that uses RAG for IR is presented. I will introduce the concept of RAG first, then give a detailed explanation of the IR agent. In chapter 5, will deal with how LLM agents can be benchmarking and present a test dataset for the RAG agent from chapter 4. The benchmark uses handcrafted question-and-answer pairs from a scientific humanities journal.

2

Backgrounds

Different branches of research were used to build upon this work. These topics and how they are connected to my work are explained in more detail in this chapter.

In section 2.1, I will describe the process of *IR systems*. IR systems take a user query and return a set of relevant documents. Generally these systems follow a fixed series of steps. To enable retrieval in the first place, the information source need to be stored and indexed in a database. After a user query is received, it is rewritten to improve retrieval Then the most relevant documents get retrieved from the index. Before presenting the results to the user, the relevant documents are re-ranked.

The notion of *agents* is described in section 2.2. Agents try to act as rational and goal-oriented decision-makers. In comparison to classical computer algorithms, agents observe, plan and act autonomously in order to complete a task. I will explain the basic concept of an agent and list the most important types of agents.

Section 2.3 deals with *language modeling* and the current developments of LLMs. Built upon the transformer architecture, modern LLMs can handle complex tasks such as text generation. Moreover, the natural language interface of LLMs can be used for tasks in other domains. An example of such a domain is *LLM agents*, explained in section 2.4. Here, the internal knowledge of the model is used to generate a decision to act towards a goal. This section gives an introduction to proposed frameworks and ideas for LLM agents.

2.1 Information Retrieval

In this section an introduction to information retrieval is given. It is based on the book "Introduction to Information Retrieval" from Manning, Raghavan, and Schuetze [17]. Retrieval of information is essential for humans. Information can be anything, like things

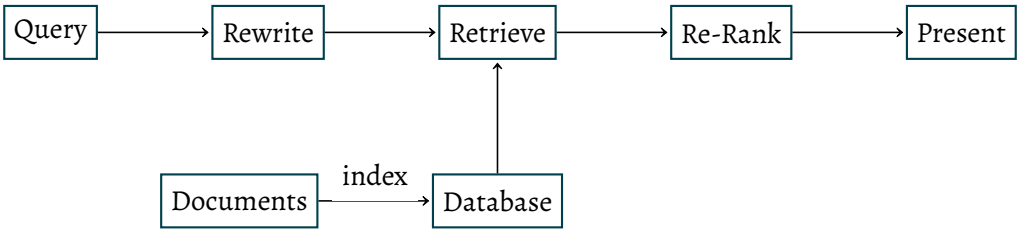

Figure 2.1: Every IR process starts with a user query. The database is then queried to retrieve relevant documents to the query. Before presenting an answer to the user, the retrieved documents are ranked by another relevance measure.

seen by the eye, thoughts or an article from a book. A broad definition of IR is:

> "Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)." (Manning, Raghavan, and Schuetze [17])

Modern IR systems extend this definition with a step dealing with the presentation of the information to the user.

Retrieving the best documents for a query from a large database filled with information of different kinds is a classic problem in computer science. There has been extensive research on database architectures and indexing algorithms. Popular applications of IR algorithms are search engines that enable fast access to billions of documents on the internet. Before retrieving information from an IR system, the data is preprocessed and stored. Then a user query initiates a series of steps that can rewrite the query, retrieve relevant documents, re-rank the relevant documents and finally present the information to the user.

Information Retrieval Steps

IR systems consist of multiple steps. In particular for common steps make up such an IR pipeline, rewrite, retrieve, rerank and present [35]. Typically, the user interacts with an IR system by writing out a query that describes the information need. A problem with queries written by humans is that they are not optimized for the following steps of the retrieval system. Human writing sentences can include spelling mistakes missing words or even wrong descriptions of information. In IR systems researchers try to mitigate such problems with query rewriting. A core query rewriting technique is to expand the query with more words to improve the retrieval accuracy.

To retrieve documents that correlate to the search query, different algorithms and models have been proposed. For a long time, functions like the BM25 [25] were used to

retrieve the best matching documents from a corpus. Term-based bag-of-words functions like BM25 rank documents based on word occurrences in every document. With neural network approaches becoming more popular, the focus has shifted toward mapping documents to high-dimensional vectors called embeddings. A similarity matrix can then be used to calculate the distance between to embeddings corresponding to the query and a document.

After a set of relevant documents was retrieved with an efficient retrieval method, the documents are re-ranked by their relevance. This time the algorithms used for ranking the documents are specialized towards quality rather than efficiency. In addition, the re-ranking phase can include task-specific ranking strategies to meet user demands.

In the last step, the information is presented to the user. LLMs have become a popular method to create user-friendly responses from retrieved documents [22].

Indexing

Before documents can be retrieved from a database, they have to be stored. How documents are stored is a crucial part of creating an IR system. While some data sources are delivered in structured formats like JSON or XML, academic texts are published in journals, articles or conference proceedings. All of these mediums are distributed as PDFs which is an unstructured data format. The PDF format does not save the content A preprocessing step is required before storing the documents to extract information from unstructured formats. There are different techniques to preprocess text before storing it. With *tokenization*, sentences are split up into words, phrases or symbols. Words with little information such as articles and prepositions are removed from the word list. The words are called *stop words*. Furthermore, a *stemmer* can be used to group words that have the same stem. All of these preprocessing steps help to capture the semantics of a text or sentence.

Cosine Similarity of Documents

Often times, one needs a useful measure of how similar two documents are. Think of the retrieval step where documents that are similar to the query are retrieved from the database. The *cosine similarity* is a measure of similarity between two n-dimensional vectors. It is defined as

$$S_C(A, B) := \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} \tag{2.1}$$

where A and B are the n-dimensional vectors and $\|\cdot\|$ is the euclidean norm. The resulting value ranges from -1 meaning opposite, to 1 meaning same. When the vectors are

orthogonal the measure is 0 meaning the vectors are not correlated. For IR, the vectors A and B represent the two documents to be compared.

Evaluation

Information retrieval systems are generally evaluated by categorizing documents as *relevant* or *non-relevant* concerning an information need of the user [17]. The *gold standard* or *ground truth* defines which documents are relevant and non-relevant. One has to consider that the relevance decision is made according to an *information need* and not the query. This distinction shows, that a clear information need has to be present in order to effectively evaluate an IR system.

Other properties of IR systems can be measured. The indexing and search speeds are important to enable effective usage of an IR system. Another property is the degree of complexity allowed by the query language. Nonetheless, the ultimate measure is the user utility. Here, all dimensions of evaluation culminate into a single idea, which makes this very hard to measure objectively. Each user has different tastes and needs for an IR system.

2.2 Agents

The introduction to agents in this section is based on the book "Artificial Intelligence: A Modern Approach" from Russel and Norvig [26]. Rationality is viewed as a core component of intelligence. In computer science, computational entities that act rationally are called agents. What it means to act rational is, and will stay an open research question for a long time. However, to use the notion of an agent in practical work a more concrete definition has emerged. An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators [26]. A human perceives the world through eyes and can act with hands or speech, a software can perceive and act through software interfaces. Both examples can be called agents.

A key step when constructing an agent is to define the *task environment*. The task environment consists of a performance measure, the environment, the actuators and the sensors [26]. Environments can be physical but also purely virtual, we are then using a software agent.

Agent Types

An agent consists of an agent architecture and an agent program. While the agent architecture makes perceptions and actions through sensors and actuators available, the

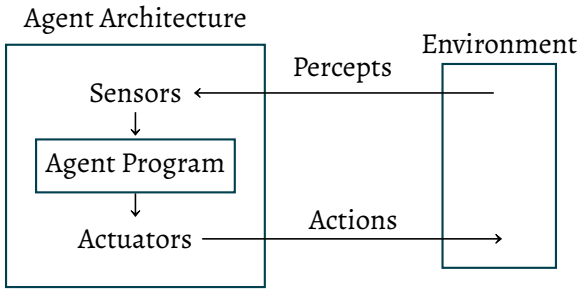

Figure 2.2: An agent as defined in [26]. The agent program runs on the agent architecture. Agents perceive their environment through sensors and act upon it using actuators. The agent program continuously maps perceptions to action and defines the agent type.

agent program implements the mapping from perceptions to actions. Agent programs can be categorized into four basic types that almost all agents are based on [26]. What kind of agent is useful depends on the given task environment.

Simple reflex agents Simple reflex agents are the simplest kind of agent. They choose actions based on the current perceptions looking at the perception history. This means that even a bit of unobservability can break the agent.

Model-based reflex agents Model-based reflex agents deal with partially observable environments by keeping an internal state of the world. This internal state depends on the perception history. Modeling physical or mental states is a complex topic that is heavily researched.

Goal-based agents In many cases, the perception history is not enough to choose the best action. To be able to do that a goal is required. Goal-based agents search for action sequences that end in a goal state. This process is called planning.

Utility-based agents Even with a goal in mind, there are cases where more than one action sequence leads to the goal state. To decide which action to select, utility-based agents choose the action that maximizes the expected utility. The utility function of the agent should ideally match the performance measure of the task environment.

To improve the performance of an agent it *has to learn*. All agent types can be extended to learning agents. For an agent, learning means modifying each component such that it better aligns with feedback from a new critic component [26]. As a result of these modifications, the agent performance improves.

2.3 Language Modeling

Language modeling is an important research area of computer science. In recent years, developments have significantly sped up with the introduction of deep learning into language modeling. The transformer network [30] has been the base for many advancements in recent years. Deriving from the transformer network, massively scaled-up pre-trained transformer models [7] enabled a big leap in the capabilities of language processing models.

Transformer Networks

Using recurrent structures such as recurrent neural networks and long short-term memory [8] was the dominant strategy in sequence modeling before transformer networks [30]. Every sentence token was represented as a hidden state that is a function of all previous hidden states. While this approach has a reasonable motivation, the sequential nature constrains computation speed for a single training example. This limitation is especially hindering for longer sequences, where batching the training data is only possible to a memory limit. Attempts to minimize sequential computation included convolutional networks that can be computed in parallel [11]. For these models, however, the number of operations required to relate two tokens grows in the distance of their positions in the sequence. Attention mechanisms allow modeling token relationships independent of their distance in a sequence.

The transformer network [30] shown in Figure 2.3 was the first model that removed all recurrent structures and only relies on attention mechanisms. In particular, they use multi-headed self-attention layers. Attention mechanisms learn dependencies between tokens in sentences without regard to their distance. Their non-sequential characteristics allow for better parallelization. Self-attention is an attention mechanism that computes relations between different positions of the same sequence to find a representation of the sequence.

Since the transformer architecture does not use any recurrent structures, the order of the tokens in the sequence must be manually injected. Positional encodings are added to the input embeddings to achieve this.

Similar to previous seqence-to-sequence models, a transformer consists of an encoder and a decoder. The encoder has two sub-layers, a multi-head attention block and a fully connected feed-forward block. The decoder is similar to the encoder but has an additional multi-head attention layer that attends to the outputs of the encoder. The first attention layer is masked, and the decoder input sequence is shifted to the right, so only previous positions can be attended by the decoder.

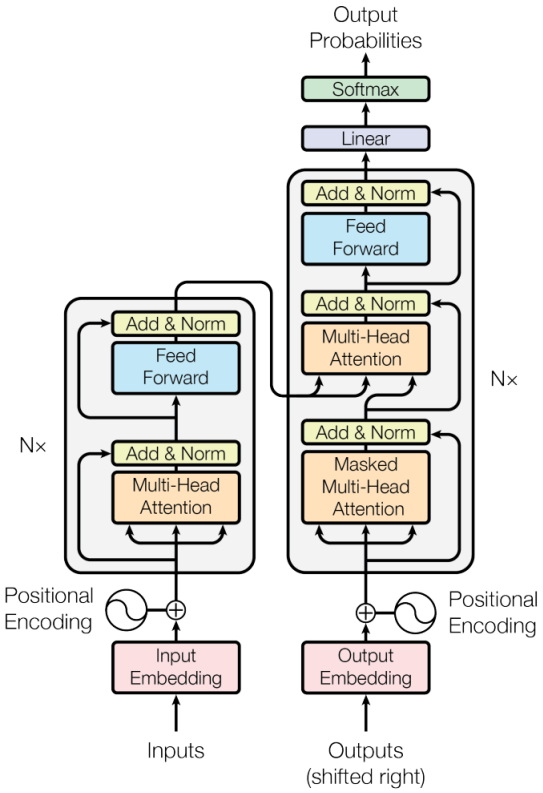

Figure 2.3: The encoder-decoder architecture of a full transformer network [30]. The encoder (left) can attend overall positions to learn rich embeddings. The decoder (right) can generate output sequences. The first decoder attention layer can only attend to previous positions of the input sequence, while the second can attend to the outputs of the encoder. This combines a sequence-to-sequence with autoregressive properties into the decoder.

The full architecture has the capabilities of a classic sequence-to-sequence model used for tasks like language translation. All positions in the decoder can attend to all positions in the encoder. For other tasks, however, the individual encoder and decoder sections are a better fit, as explained in the section 2.3 and section 2.3.

Generative Pre-Trained Transformers

Decoder-only transformers are the base of generative pre-trained transformers (GPT) [24]. The decoder used for GPT does not rely on the outputs of an encoder, because it is designed for generation tasks. In Figure 2.4 we can see that only the masked multi-head attention layer is kept in comparison to the full transformer network. The simple architecture enables faster pre-training and fine-tuning.

GPT is trained in two phases. In the unsupervised pre-training phase, the model is

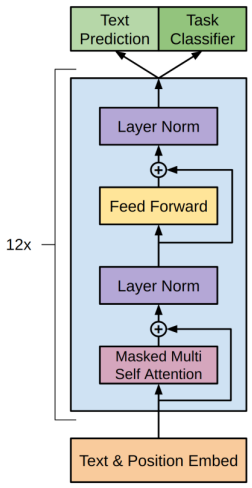

Figure 2.4: The GPT architecture [24]. A transformer encoder without connections to a previous encoder is used. The attention layer can only attend to previous positions in the input sequence. The decoder is stacked 12 times before generating the final output.

trained with big datasets of unlabeled text data. The training objective here is to accurately generate the next token of a sequence.

After the first pre-training phase, the model needs to be fine-tuned for a specific task. By including custom tokens in the fine-tuning dataset, the decoder can learn to handle a variety of different tasks such as text generation or classification.

Newer research on generation models focuses on scaling the network. The *scaling law* states that by simply increasing the number of parameters of the network, the model capabilities increase. Furthermore, after a certain level of scaling, abilities emerge that the model was not directly trained on.

As LLMs have reached parameter counts of over a billion, only very few companies have the hardware to train or even run inference tasks.

Embedding Models

The left-to-right nature of the transformer decoder suits text generation tasks that only depend on previous tokens. However, there are dependencies between tokens in both directions, which means a decoder-only model can not capture all relations.

The input sequence is transformed continuously while being processed through the transformer. Each transformation of the vector leads to a different intermediate representation. Each representation encodes different features and relationships of tokens in the sentence. These representations are called embeddings. Semantically similar documents are represented with embedding vectors that have a small distance, while seman-

tically different documents have a high distance.

Bidirectional encoder representations from transformers (BERT) [9] uses a transformer encoder, to create rich embeddings of input sequences. In pre-training, the bidirectional encoder is trained with two unsupervised tasks. For the first task, random tokens in the sequence are masked out to predict the masked tokens from the remaining tokens in the sequence. In the second task, sentence-level relationships are learned by predicting the next sentence. After pre-training, BERT can be fine-tuned on different downstream tasks leveraging the rich bidirectional embeddings learned in pre-training.

Instruction-Tuned Models

Language models are trained to predict the next token of a sequence. While a lot of knowledge is captured in the weights of the model, most information on the internet is not formatted as conversations between an assisting chatbot and a user. Because of this, language models need to be prompted in a specific way to be effective. The prompt needs to be written such that its continuation yields the desired output. This is not optimal for human users, as one would rather communicate in conversations. The training objective of LLMs is different from the objective "follow the user's instructions helpfully and safely" [20]. Researchers say that the language model is not *aligned*. A popular attempt at aligning language models is reinforcement learning with human feedback (RLHF) [20].

For reinforcment learning with human feedback (RLHF), handcrafted prompts are used to fine-tune GPT-3. The outputs of the model are collected into a set and ranked by humans. This set is then used to train a reward model. With this reward model, the language model is further fine-tuned. The resulting model is called *InstructGPT* and performs better than the baseline GPT-3 model.

Large language models are trained to predict the next token of a sequence, not to follow the instructions of the user. This leads to some unwanted results such as toxic, harmful or fabricated answers that are not true. When an LLM wrongly answer with high confidence, the model is "hallucinating". Even the latest LLM like GPT-4 suffer from this problem [19].

2.4 Large Language Model Agents

With the rise of LLMs and their impressive capabilities in various language tasks, efforts began to leverage them in agent systems. Experiments to give GPT access to tools [28, 27], that LLMs can use external functions if they are prompted with a description of how to access them.

For practical applications, [35] proposes a distinction between static and dynamic agents. A static agent is characterized as a fixed pipeline that mimics the user behavior. While this approach works, it cannot deal with complex and sometimes random human actions. The other type of agent can dynamically execute actions that are presented to it. The most prominent way of using LLM for agents is to build a prompt that contains all the information about the task, and then ask it to propose the next action. The prompt can contain descriptions of possible abilities, the task, guidelines, information about the environment and more.

Experiments in [15] showed that LLMs focus more on the first and last lines of the prompt in their answer generation. This needs to be respected when constructing long prompts to use LLMs for agents.

Similar to the view of human memory, different implementations of memories for agents have been proposed. For short-term memory, the message history format used by instruct language models can be used [13]. The messages of a conversation between the user and the agent are recorded and change the generation behavior of the LLM that controls the agent.

Long-term memory is generally represented as some external store where information can be stored and retrieved [13]. This memory can include past conversations between the agent and the user, as well as additional data that the agent should have access to. The information can be present from the beginning or added at runtime by a user prompt.

For LLM applications vector databases are a popular choice, because they integrate well into the embedding-based language modeling concept [12]. Vector databases store data together with their high-dimensional embedding. To create the embedding, models like BERT are used. The embeddings allow for fast similarity search with a query sentence.

Agent Construction

Most LLM agents share the same construction pattern. A construction framework for LLM agents consisting of four modules is proposed in [31]. Their framework is composed of a profiling module, a memory module, a planning module and an action module.

Before the actual execution of steps, an agent is given a profile that describes the role of the agent. This profile is usually written into the prompt that is sent to the language model. Profiles can be handcrafted by simply adding a "You are an x" statement to the prompt. Furthermore, profiles can be generated by LLMs which saves time for multiple agents but does not allow for precise control in the profile crafting process.

Information is stored in the memory module. Memory is used to keep a mental state

of the environment and store new observations. LLM agent memory types draw inspiration from human short-term and long-term memories. Short-term memory is generally represented by the in-context information given in the language model prompt. It is restricted by the context window length of the transformer architecture. Long-term memory represents an external storage of information. For LLM agents, a popular choice is a vector database, that allows for quick query and retrieval of information.

Planning strategies for agents can be separated into planning with and without feedback. Planning without feedback uses single-path or multi-path reasoning. For single-path planning, techniques like chain-of-though (CoT) and zero-shot-CoT prompt LLMs to "think step-by-step" [32]. These prompts can be used to produce a full plan with all steps from a single prompt. Other single-path approaches like HuggingGPT use the language model repeatedly after each step to generate the next one [28]. For multi-path reasoning CoT can be used again, now generating multiple plans in a tree-like structure [34]. For complex tasks, the agent should be able to incorporate feedback into the planning process. The feedback can come from the environment, humans or from a feedback model. All of these changes have to be represented in the language model prompt.

To act out the agents' decisions in the environment, the agent needs an action module. There are two main strategies for acting out actions. For the first strategy, the agent uses the task and its memory as prompts to choose the next action in a continuous loop. As the agent proceeds, the memory builds up with the action history. This allows for dynamic action choices while the agent is running. The second strategy uses the task to create a fixed action plan that is then executed strictly. Here, the agent can not adapt dynamically. The agent space defines the set of possible actions that can be performed. Agents can leverage external tools and internal knowledge. Types of external tools can include APIs, databases or other models that specialize in a specific task. The internal knowledge of a language model is also important for the action module. When deciding which action should be used in a step, The planning, conversation and commonsense understanding capabilities are crucial to make a good decision. The outcomes of actions can change the environment, the internal state and also trigger new actions.

3

An Analysis of AutoGPT

After its initial release, AutoGPT quickly gained a lot of traction in the open-source community. The project tries to "make GPT fully autonomous", by using GPT as an agent "brain". AutoGPT has sparse documentation that is quickly outdated, as the project is unfinished and often changes. Therefore, I will give a brief introduction to the main components of AutoGPT and analyze the project in the context of IR to build the foundations for the agent creation chapter.

Section 3.1 introduces the main components of the AutoGPT project. What initially grabbed the attention of many people is not the whole project, but the *AutoGPT agent* component of AutoGPT. The project later introduced the *Forge agent*, an agent framework to build custom agents. It defines a minimal agent without logic that abstracts away all the boilerplate. Additional, the AutoGPT project contains a benchmarking system. The *benchmarking system* can be used to build level-based test suites, to test the capabilities of an agent.

In section 3.2, the project and its components are analyzed in the context of information retrieval. The AutoGPT agent is built to be a "generalist agent" that can do a variety of tasks and therefore is complex. Because of the complexity, I opted to create a custom Forge agent to research the use of AutoGPT for IR. To work with local data, the agent has the ability to read text files and put the output into the prompt. This defines the key limitation of the agent, for longer texts the context window of the LLM is too short. Therefore, I chose to use a RAG approach in combination with AutoGPT.

3.1 Main Components

In this section, the main components of the AutoGPT project are introduced. The project provides two agents, the AutoGPT agent and the Forge agent. While the AutoGPT agent

is built to be a ready-to-use "generalist agent", the Forge agent provides a minimal agent for developers that want to implement a custom agent. Furthermore, AutoGPT comes with a benchmarking system to test agent abilities.

To make collaborating on agent projects easier, the open-source community created the *Agent Protocol* [1]. The Agent Protocol defines an API schema that handles the communication with an agent. On a high level, the protocol defines endpoints to create tasks and trigger the next step for a task. The two important concepts of the agent protocol are tasks and steps:

Task A task describes a goal for the agent. A task has an input prompt and contains a list of steps.

Step A step describes a single action of the agent. Every step has to be linked to a parent task and has its own input. Additionally, a variable signals if this is the last step. If this variable is false, the next step is requested automatically.

The AutoGPT agent and Forge agent both implement the agent protocol.

AutoGPT Agent

The AutoGPT agent made up the whole project when AutoGPT was first released. It is the default agent and comes with pre-defined logic. Currently, the AutoGPT agent is optimized for OpenAI GPT models. The prompts are tailored in ways that benefit the characteristics of GPT-3 and GPT-4. The official documentation describes the default agent as

> "A *generalist* agent, meaning it is not designed with a specific task in mind. Instead, it is designed to be able to execute a wide range of tasks across many disciplines, as long as it can be done on a computer." (*AutoGPT Agent Introduction* [3])

This generalist design can be seen in the list of available abilities.

Code Execution The agent can execute arbitrary Python code given as a string or a file. The execution is contained in a docker container that can access the agent workspace.

File Operations To access files in the agent workspace, it has abilities to list, read and write files.

Image Generation The agent can make calls to different image-generation services that transform a prompt into an image.

Web Search Web search is the most common application of the default agent. The agent has abilities to query a search engine and scrape website content using a headless browser.

Other Further abilities are present in the default agent. The user can be prompted to give more information about the task, the system time can be accessed. When the agent ends its run, a finishing ability is chosen.

During the development of the default agent, different abilities have been present and were then removed again. For example, efforts were made to introduce long-term memory to the agent. There were abilities to save agent conversations into a database for later retrieval. The idea was to improve the agent's performance by using past conversations as context. However, experiments showed that using retrieved long-term memories of conversations did not improve performance substantially [23].

The AutoGPT agent is modeled after the classic agent architecture. After the start, the user is asked to enter a task the AutoGPT agent should perform. Then the agent enters a loop of prompting the LLM, executing the proposed action handling the result of the action and updating the agent state.

Forge Agent

The Forge SDK handles the boilerplate code that implements the agent protocol. On running, the server with the corresponding endpoints gets started, and the agent can be used over the API endpoints. AutoGPT also comes with a chatbot web app that builds the appropriate HTTP requests to the agent endpoints. The user of the Forge agent has to create the actual agent logic, create custom prompt templates for the used model and add abilities to interact with external resources. Because the Forge agent is still under development and by no means a polished product, some internals also need to be tweaked to achieve the desired agent behavior.

By default, the Forge agent comes with abilities to read and write text files and to search a search engine as well as scraping a webpage. Each ability is specific by a name that the LLM can use to call it. Additionally, a short description of what the ability does, input parameters and return types are described. The information about an ability is formatted into a string before adding it to the step prompt.

File System The file system abilities allow operations on files located in the workspace of the agent. The agent can only operate in the defined workspace, this prevents unwanted effects when the agent proposes unexpected actions. Abilities to read, write and list files are present. All the abilities have a path parameter that the large language model has to populate when generating a step proposal with actions from this category.

Web The web search functionality is split up into abilities to call a search engine and to read a webpage The web-search ability requires a search string that is sent to the en-

```
Reply only in JSON with the following format:

{
      "thoughts": {
            "text":  "thoughts",
            "reasoning": "reasoning behind thoughts",
            "plan": "- short bulleted
                     - list that conveys
                     - long-term plan",
            "criticism": "constructive self-criticism",
            "speak": "thoughts summary to say to user",
      },
      "ability": {
            "name": "ability name",
            "args": {
                  "arg1": "value1", etc...
            }
      }
}
```

Listing 3.1: The system format of the Forge agent. The language model is asked to only answer in this format. The model generates thoughts before creating the output for the user (speak). Thoughts include reasoning, a plan and criticism. After the model generated the thoughts, it generates an ability proposal with the corresponding arguments.

gine. For the webpage ability a URL is needed, and if specific information should be extracted the LLM also has to provide a question.

Finish The "finish" ability terminates the agent loop. The agent is asked to choose this ability if the initial user prompt can be answered and give a reason.

The Forge agent comes with a built-in template engine that is used to populate the prompts before sending them to the large language model. Some templates are included by default:

Task-Step This template is used to create the prompt that is sent to the language model for each step. It includes the current task description and placeholders for extra information. The template always needs to be populated with the list of available abilities, allowing the language model to choose one of them.

System-Format The system format defines how the model should respond to prompts.

The Forge system format is depicted in Listing 3.1. The responses of the model need to be parsed according to this definition. Language models have different capabilities in answering in a structured manner, so the format has to be tuned for every model.

Techniques Techniques is a collection of prompting techniques that have been shown to improve generation quality. By default, the Forge agent has templates for few-shot, expert and chain-of-thought prompting.

The core of an agent is its logic. The Forge agent comes without any logic, its default behavior is to write a boilerplate text into a file in the workspace.

Benchmarking System

To evaluate AutoGPT and other agent systems that implement the agent protocol, the AutoGPT project has implemented a benchmarking system. The system consists of a set of tasks that the agent has to complete. The tasks are designed to test different aspects of the agent and are divided into different topics. Some tasks depend on the previous successful completion of other tasks. A task consists of an input prompt and an expected output. The output is defined by certain words that should be contained.

3.2 Analysis for Information Retrieval

Before I describe the use of AutoGPT as an IR agent in the next chapter, I will reason about my choices in this section.

The AutoGPT agent was not built with IR as a primary objective in mind. As stated in section 3.1, it is designed to be a "generalist agent". The AutoGPT agent has abilities to retrieve information from the web. With the web searching and website scraping abilities, the agent can look up and extract information about a given topic. For IR from local data sources there are no specific abilities present. The agent can simply read text files and include the content in the next step prompt. This means that every other file type has to be converted into plain text. Another inherent problem with this approach is the limited context window size of the language model. If the relevant information is contained in a long text, including the full text in the prompt will not be possible. Furthermore, the AutoGPT agent has a lot of other abilities that are irrelevant for IR. While they can be disabled, they still increase the complexity and points where the agent can break. This leads to more difficulties when using the agent for a specific task like local IR.

The Forge agent has similar default abilities for retrieval of information from the web, as well as read and write abilities. Therefore, the same problems with context window

length apply here. As the agent only contains boilerplate implementation for the agent protocol but no default logic, it is better suited as a starting point for a custom IR agent.

The benchmarking system can be used to test the abilities of agents. The level-based system is put in place to save money when testing, as later stage tests do not get executed if a required test in the earlier stage failed. To evaluate IR systems, a larger dataset of QA pairs is often used. For large dataset evaluations, the benchmarking system is not suited. It is built towards testing if an agent *can* achieve a goal rather than *how good* the produced answer is.

These arguments lead to the conclusion that to examine the use of AutoGPT for IR, it is best to create a custom agent built on the Forge agent. To enable local IR over large documents specific abilities will be introduced. The creation of the agent is described in the next chapter.

4

A Retrieval Augmented Generation Agent for IR

Large language models are excellent at generating text in a variety of styles. But when prompted about specific topics the answer quality suffers. Different approaches try to tackle this problem. A promising method is RAG, which combines in-context learning prompting techniques with a vector database. In this section, I present an agent that is designed to answer user prompts in a retrieval-augmented fashion.

To create the IR agent, I used the Forge SDK [4] that is included in AutoGPT. The Forge agent SDK is a tool to create a custom agent without having to write the boilerplate code. In comparison with the AutoGPT agent, it comes with less predefined abilities and has no initial logic. On the contrary, fewer parts can break, and it is not under an ongoing re-factoring process, compared to the AutoGPT agent. Additionally, the AutoGPT agent is not optimized for IR over a set of documents but operates more as a general-purpose agent.

I extended the Forge agent with abilities that are needed for RAG. To store the documents for retrieval, I used a *ChromaDB* vector database. The 'ingest' ability of the agent can be used to ingest a PDF file that is saved in the workspace. The PDF is converted into plain text and then chunked into smaller documents using the SentenceSplitter from LlamaIndex [16], which produces chunks of roughly equal length while respecting sentence boundaries.

4.1 Methods To Improve LLM Generation

Different approaches try to embed information sources into the generated text. One approach is to fine-tune the language model on a dataset that contains the information. The creation of fine-tuning data is an expensive task, as data needs to be gathered and cleaned to produce good results in training. There is some emerging work on generating

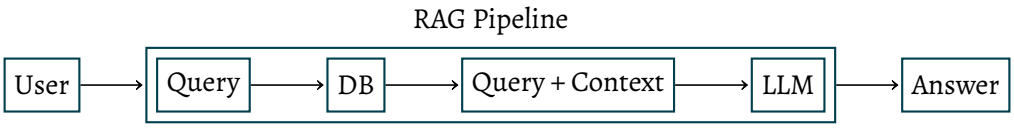

Figure 4.1: Instead of directly answering the user prompt with an LLM, a RAG system leverages context to generate better answers. After receiving the user prompt, it is used to retrieve relevant documents from a database. These documents are appended to the language model prompt as context. The model is instructed to answer only using the provided context.

synthetic datasets by using large language models as data augmentation tools. But even if the fine-tuning data quality is sufficient, it is still a challenge to make an LLM expert in a specific domain. Rather than learning new knowledge, fine-tuning is best suited to guide the model toward a certain answering style. Furthermore, fine-tuning models with billions of parameters is only possible on expensive hardware and therefore not feasible for on-demand tasks.

Following up on the challenges of fine-tuning researchers have searched for ways to optimize prompts to get the best model performance. These methods make use of a phenomenon called in-context learning. In-context learning describes the observation that giving a large language model more context surrounding the prompt can drastically improve the response quality. [32] found that adding a sentence that suggests step-by-step thinking to solve a problem makes the model better at solving logical questions. For GPT models, [33] showed an improvement after giving the language model a hint that it is an expert in a certain topic. The findings in the area of prompting techniques and in-context learning are suggesting that prompt optimizations have a high potential of getting the most out of large language models.

A more promising approach is RAG [14]. Before generating an answer to a prompt, a vector database is searched for relevant information. The prompt then includes the information of the returned documents as context.

4.2 Retrieval Augmented Generation

LLMs can embed a large amount of knowledge into the weights during the pre-training phase. It is possible to generate good answers for different kinds of prompts. But when it comes to specific domain knowledge, the exactness of LLM starts to degrade.

When trying to prompt models about very specific topics common problems such as hallucinations start to show. Not only does the model generate wrong information, it

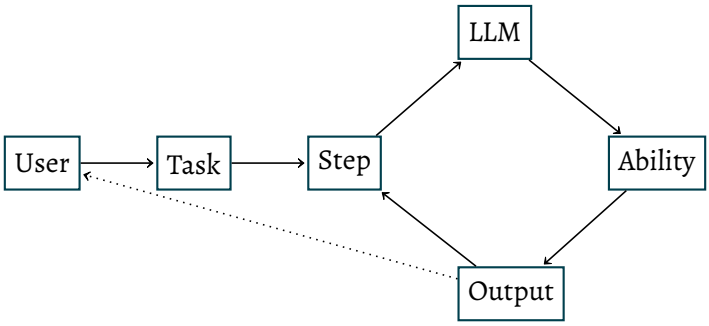

Figure 4.2: The IR agent receives the user prompt at the start of each cycle. Next, the step prompt is constructed with the action history, list of abilities and extra information. After the LLM responds to the step prompt, the proposed action is executed and the step output is presented to the user. The loop ends if the LLM proposes the 'finish' ability.

does so with strong confidence. A promising method to make the model answer based on specific sources is called RAG.

The RAG method combines an information store with in-context learning. A corpus of documents is stored in a database. When the user prompts the system, the database is queried with that prompt matching documents are returned. These documents are then included in the prompt to the LLM as context.

To store the information a vector database is used. A vector database uses embedding models to embed each document into a high-dimensional vector space. After a user query, the query string gets embedded by the same model, and then a metric such as the cosine similarity is used to find semantically close documents. The top-k documents are then returned.

4.3 Agent

In this section, I will describe in detail how the IR agent works.

Main Loop

The agent program is defined in the agent step execution function. Every time a new step request is given this function is called. For the first user input, a task is created, then the agent proceeds to complete steps until the task goal is reached. A single cycle of the agent follows a fixed set of actions. First, the step input is updated for the current step. The user can give updated instructions in each step to further guide the agent. If no new input is given, the agent uses the input of the parent task as the step input.

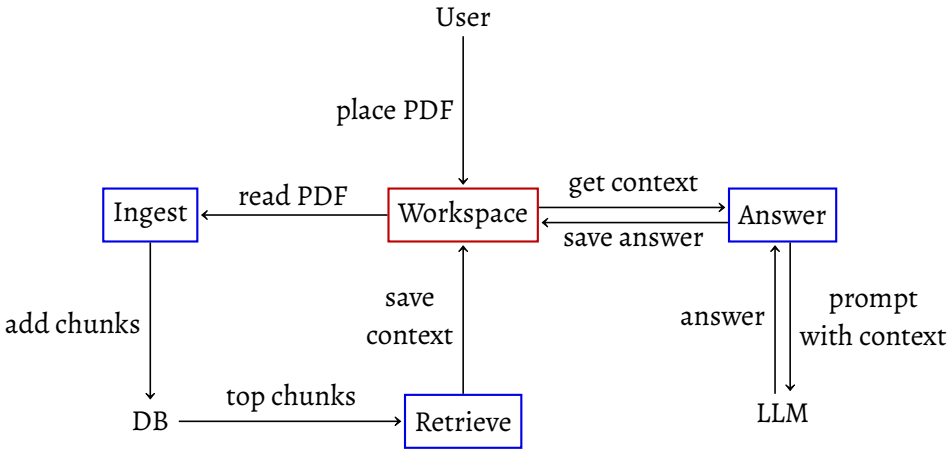

Figure 4.3: The RAG agent manipulates the workspace folder with abilities. To ingest a PDF, the user has to place the file into the workspace. After chunking the PDF, the chunks are ingested into the vector database. With the retrieval ability, the agent can retrieve the top five chunks for a query and save them in a workspace file. The retrieved documents are read from the workspace by the Answer ability. The generated answer is then written back into a file in the workspace.

After the input is updated the agent constructs the message history for the language model. The first message is always the same system format specification. The agent uses the default Forge agent format shown in Listing 3.1. The second message is the populated step prompt, which describes the current perceived environment of the agent. When the step prompt is constructed, it is sent to the language model. The generated answer is parsed into the thoughts and ability parts.

The response is parsed into the thoughts of the LLM and the proposed ability. If a valid ability is proposed, it is executed.

Agent Memory

There are three places that the agent stores memories, the workspace, the vector database and the prompt context.

The workspace is used to store larger amounts of information between agent steps. It can be viewed as a paper where a person writes notes on while working. Abilities can modify anything that is inside the workspace. For each task started by a user, the agent creates a new workspace. Access from abilities to the workspace is shown in 4.3. After retrieving the top documents from the vector database, the agent stores them in workspace text file. When the answering ability is chosen, the workspace file is read and used to populate the RAG template. After the LLM answers the RAG prompt, the answer is written

back into another workspace file.

The vector database can be viewed as a storage for external information, similar to a library. It is used to store large amount of information that is indexed upon addition to the database. A user can prompt the agent to ingest a PDF into the vector database with an ability. This starts the indexing process described in section 4.2 When the retrieval ability is activated, the agent retrieves the top 5 documents from the vector database.

The third storage is for storing information about the internal state of the agent. Here, the agent keeps track of a step history for each task. Each history includes the LLM output with the thoughts and proposed ability. Furthermore, all ability outputs are saved. This information is used to build the step history that is appended to the step prompt.

In summary, there are three places where information are stored

Step Prompt

The main step of the agent is to populate its step prompt template As denoted in section 2.2, an agent perceives its environment through sensors. The agent describes the current environment in the step prompt. An example step prompt is shown in listings 4.4 and 4.5. It includes the current step input and instructions on how to answer. The available resources are denoted, and the action history is appended.

If the agent is in its first step, the action history is empty. Otherwise, a list of the previous actions is compiled to give the agent a sense of its current state inside the task context. Each entry has the proposed ability with its parameters and the ability output. If the ability produced an error, this is also denoted. In the best practices section, the agent is prompted to react to errors and to not use the same action with the same arguments again.

In addition to the action history, the available abilities are added to the prompt. The abilities enable the agent to act out its decisions in the environment. For each ability, the name, parameters, return type and a short description is given.

Step Prompt Answer

The answer to the step prompt is from JSON into a usable format. An example answer to the step prompt from listings 4.4 and 4.6 is shown in listing 4.6. Given the history of actions, the LLM decided to choose the finish ability. This is the correct choice as the retrieval and answering action have already happened.

```
Answer as an expert in Planning.
Your task is:

Continue

Answer in the provided format.

Your decisions must always be made independently without seeking user
    assistance. Play to your strengths as an LLM and pursue simple
    strategies with no legal complications.


## Abilities
You have access to the following abilities you can call:

- answer_with_context(prompt: str, context_file: str, output_file: str) ->
    file. Usage: Answer a prompt using content from a textfile as context,

- retrieve_context_from_memory(query: string, output_file: string) -> file.
    Usage: Retrieve the most relevant information for a query and save it
    to a .txt file,

- finish(reason: string) -> None. Usage: Use this to shut down once you
    have accomplished all of your goals, or when there are insurmountable
    problems that make it impossible for you to finish your task.,
```

Listing 4.4: The step prompt is the core of each agent step. It describes the current environment representation of the agent. First, the current task is presented to the agent. Next, available resources and abilities are described. Finally, some hints for best practices are denoted. In this example, there is no step history as this is the first step.

```
## History of Abilities Used

- Step 1:
  Input: How is the performative distinction between chants and readings
      maintained in the breviary?
  Proposed Ability: {
    name: retrieve_context_from_memory
    args: {
      query: performative distinction between chants and readings in a
          breviary,
      output_file: breviary_context.txt
    }
  }
  Output: file: breviary_context.txt

- Step 2:
  Input: None
  Proposed Ability:{
    name: answer_with_context,
    args:
    {
      prompt: How is the performative distinction between chants and
          readings maintained in the breviary?,
      context_file: breviary_context.txt,
      output_file: output.txt
    }
  }
  Output: file: output.txt
```

Listing 4.5: The second part of the step prompt contains the action history. Here, all previous actions are listed with its name, arguments and output. With the action history, the agent can perveices its progress at achieving the goal.

```
{
  'ability': {
    'name': 'finish',
    'args': {
      'reason': 'Task completed successfully. The required explanation has
          been provided, and there are no outstanding queries or unresolved
          problems. It's time to shutdown.',
    }
  },
  'thoughts': {
    'criticism': 'A potential area for improvement could be the initial
       framing of the query for the memory retrieval step. Ensuring the
       query is as precise relevance of the retrieved context.',
    'plan': '- Reflect on the precision of the information retrieval
             - Evaluate the effectiveness of the answer provided
             - Contemplate potential follow-up actions or queries that
                 could deepen or expand upon the provided explanation
             - Decide on the next step, whether it's concluding the task or
                 seeking further clarification or details',
    'reasoning': 'The task required an understanding and explanation  of a
       specific aspect of the breviary, which is complex and likely
       requires specific knowledge not innately known. Therefore,
       leveraging the vector database to first retrieve context is aligned
       with the best practices outlined. Following that, using the
       `answer_with_context` ability ensures the answer is enriched and
       accurate based on the  sourced information.',
    'speak': "Using our resources, I've successfully provided an
       explanation of how chants and readings are differentiated in a
       breviary. The process involved retrieving context and then crafting
       an answer based on that context.',
    'text': "I've completed the task of explaining the performative
       distinction between chants and readings in the breviary using the
       provided resources. The first step involved retrieving relevant
       context from memory about the topic. The second step was answering
       the query with the help of the retrieved context.'
  }
}
```

Listing 4.6: The answer to the step prompt from listing 4.4.

Abilities

In section 2.2, I described how agents use actuators to perform actions in the environment. The IR agent is a software agent that uses functions to modify its workspace. In particular, it has abilities that enable retrieval augmentation capabilities. These abilities will be described in the following.

Ingest The *Ingest* ability should be used when a document should be a source of information for the agent. A file with the specified filename has to exist in the agent workspace before calling this function. This ability converts the PDF into plain text, creates text chunks of the same length and calls the functions to embed them into the vector database.

Retrieve The *Retrieve* ability accepts to query string and an output file path. With the query string, the agent queries its memory and gets the most relevant documents for it. The documents are formatted and saved to the specified file path.

Answer This ability uses the contents of a text file to populate the augmented generation template. The populated template is then sent to the LLM to generate an answer based on that context.

As the LLM tends to choose web search abilities to collect information, I removed the corresponding abilities. The agent is therefore forced to retrieve information from local sources.

Document Ingestion

To enable RAG, the agent can ingest documents into its vector database. The "ingest" ability can be called with a PDF filename as the argument. For the ability to succeed, a PDF file with the exact filename has to be present in the agent workspace. The user has to manually place the file in the workspace before prompting the agent. If a PDF is found, it is converted into plain text. Then, the text is split up into smaller chunks that can be mapped to embeddings with an embedding model. The SentenceSplitter utility from the LlamaIndex framework [16] is used, to create the chunks. The splitter splits up a text recursively and then joins words back together to generate chunks of similar length while respecting sentence borders.

Answering with Context

In a successful run, the agent generates an answer to the prompt with relevant context. For this ability to succeed, a text file containing the relevant documents has to be present

Listing 4.7: The prompt template to generate an answer with context. The language model is prompted to indicate which document was used to create the answer. captionpos

```
After every answer sentence, include a document id '<id>' token for the
    corresponding context document. If no context document was used to base
    the sentence, use the '<?>' token.

Task:
...

Answer only using the provided context:
...
```

in the agent workspace. The relevant documents are then used to populate a prompt template, which is shown in listing 4.7. Additionally, the model is prompted to append a token that indicates which context document was used to generate a sentence.

5

An IR Agent Benchmark

The evaluation of LLM agents is a difficult task, as evaluating LLMs alone already presents a challenge. There are different approaches to evaluating systems built around language models. Subjective evaluation is based on human feedback. As LLM systems are generally built to serve humans, this is an important part of evaluation. On the other hand, quantitative metrics that can be computed are used for objective evaluation. Different metrics are used for different tasks. Another important method is benchmarks. Benchmarks are a set of tasks or an environment that the agent is to move in.

When talking about agents, different properties can be evaluated. LLM agents can be evaluated end-to-end, where only the output is relevant for measuring performance. Other methods also use intermediate outputs to evaluate agent decisions on a step level.

5.1 Benchmark Considerations

Before creating the test benchmark for the IR agent some considerations have to be made, mainly about the price of API calls, and which parts of the agent to benchmark.

The current AutoGPT agents are tailored for GPT-4 as their decision-making LLM. As the agents call the LLM API in every step, runs can get expensive quickly. While this can be handled when running the agent manually step by step, benchmarking the agent on a large scale dataset can be dangerous. The agent runs can fail, which would mean that the money was wasted, or an agent can get stuck in loops that don't stop. Smaller local language models are not capable enough to replace GPT-4 as the decision LLM. Therefore, a benchmark with a large dataset was not feasible in my case.

Another question is which parts and characteristics of the agent we want to benchmark. One option is to evaluate the agent end to end, using a set QA pairs. The end-to-end test ignores the inner works of the agent and only looks at input and output. To better

understand the problems of the agent one might want to analyze every step of the agent. While a step by step analysis does not matter for the user of the agent, it can give useful insights for future research.

After considering the different benchmarking options, I decided to create a small dataset contains QA pairs based on a scientific journal. The agent is tested end-to-end first, and then examples from the dataset are used for quality evaluation on a step level. I will describe the creation of the dataset in the next section.

5.2 Benchmark Creation

In this section, I will describe the creation of the small IR dataset containing QA pairs from a scientific journal. For QA pair generation, the journal *Manuscripts and Performances in Religions, Arts and Sciences* from Brita et al. [6] was chosen.

Large language models have delivered good results for general knowledge questions, but can struggle with domain-specific tasks. They can confidently generate answers that are completely made up but seem correct to a non-expert user. For this reason, the IR agent uses RAG to gather context from a local vector database, before generating an answer. The retrieval does not have to be evaluated, as the agent uses an external library that is tested separately. What needs to be tested is the answer quality of the chatbot. As the built-in benchmarking system of AutoGPT was unfinished and not ready to be used for custom agents, I had to test question-and-answer pairs by hand in order to evaluate the IR agent. I adapt the end-to-end evaluation based on cosine similarity proposed in [5], to compare the IR agent to other chatbot services.

Data

The first step in creating the benchmark was to find a suited information source. We need documents that contain domain-specific expert knowledge to evaluate the answer quality of the IR agent. The humanities journal *Manuscripts and Performances in Religions, Arts and Sciences* Brita et al. [6] was chosen as an information source. I used this journal as it was published recently, and therefore unlikely used in the pre-training stages of GPT-4. This makes it a good fit for testing, as the agent has to use its retrieval capabilities instead of relying on pre-training knowledge of GPT-4. The journal presents a collection of papers that offer perspectives on how manuscripts can be studied as objects and actors in various kinds of performances. From the journal, I handpicked 10 question and answer pairs for evaluation. I am not even close to being an expert in the field, so I tried to pick questions that are specific topic, but can be verified by simply reading the article. While this set of

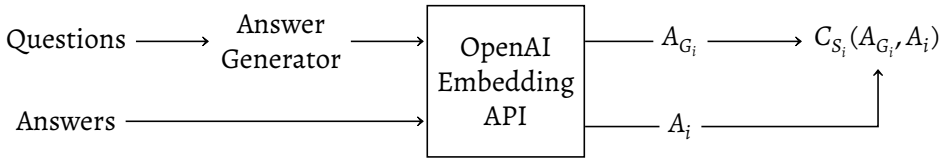

Figure 5.1: The E2E benchmark uses cosine similarity to evaluate how well the generated answer matches the true answer based on their "meanings". Generated answer and true answers are passed through the OpenAI Embedding endpoint. For question i, A_{G_i} is the embedding of the generated answer, while A_i embeds the true answer.

pairs can not be used to evaluate the general capabilities of the agent, it can be used to compare the IR agent to other services and make qualitative statements with examples.

Generation of Question and Answer Pairs

The QA dataset consists of 10 pairs that were chosen manually. To first find possible candidates, I pasted multiple articles in a ChatGPT chat. I used ChatGPT-3.5 with the prompt:

```
Article:

"..."

Generate 20 question and answer pairs from the article
for an information retrieval chatbot test dataset.
```

As LLMs can make up information while answering with high confidence, I first checked the information of the QA pairs. In the end, I kept the proposed questions and changed the answer. To make sure that the answers do not contain made up content, I looked for the exact passage in the article that contains the answer to the question. The length of the passage was chosen rather arbitrary. I tried to extract the passages such that only the answer to the question is included. This was not possible every time, as the answer was split up over multiple sentences in some instances.

The questions in the dataset follow a certain pattern. In particular, they ask for specific facts from the journal articles. In order to give the correct answer, one has to know the journal.

Benchmark Application

To put the evaluation results into perspective, other services for IR were tested on the QA dataset. I tested GPT-3.5-Turbo, GPT-4-Turbo and Perplexity for comparison. For

all tools, the answer quality was measured by calculating the cosine similarity between their embeddings. I used the OpenAI Embeddings API [10] to generate the embeddings of all questions and model answers. All services are only receiving the question with any additional information. The full process is depicted in figure 5.1.

6

Results

In chapter 5 I described the creation for a benchmark to test the IR agent created in chapter 4. This chapter presents the results. First, the results for four different chatbot services will be compared section 6.1. I compare the result of GPT-3-Turbo, GPT-4-Turbo, Perplexity AI and the IR agent. The small size of the dataset does not allow general statements about answer quality for the IR agent. Therefore, section 6.2 looks at exemplary question and answer pairs to compare the answer quality. While generating the answer for the benchmark, the IR agent had some failed runs. The patterns that lead to failed runs are described in section 6.3.

While working with AutoGPT and developing the IR agent, a couple of weaknesses of current LLM agent systems became visible. The most important challenge for LLM agents is the non-deterministic generation of large language models. It makes it significantly harder to build a software system around it because every time the language model is called, there has to be a fallback mechanism in case something fails. For fixed pipeline systems this is true as well, but error handling is easier as the task steps are known beforehand.

6.1 Benchmark Results

In chapter 5 a benchmark to test the IR agent was created. The benchmark consists of a small question and answer with 10 pairs. Testing the IR agent was done manually using the AutoGPT web interface. I pasted the question in the prompt window without any extra guidance for the agent.

As services to compare the agent to I chose the free version of Perplexity AI, GPT-3.5-Turbo and GPT-4-Turbo. To generate the answers of Perplexity AI, I pasted the question the prompt window in the same way as with the IR agent. For the GPT models a used the

API functions.

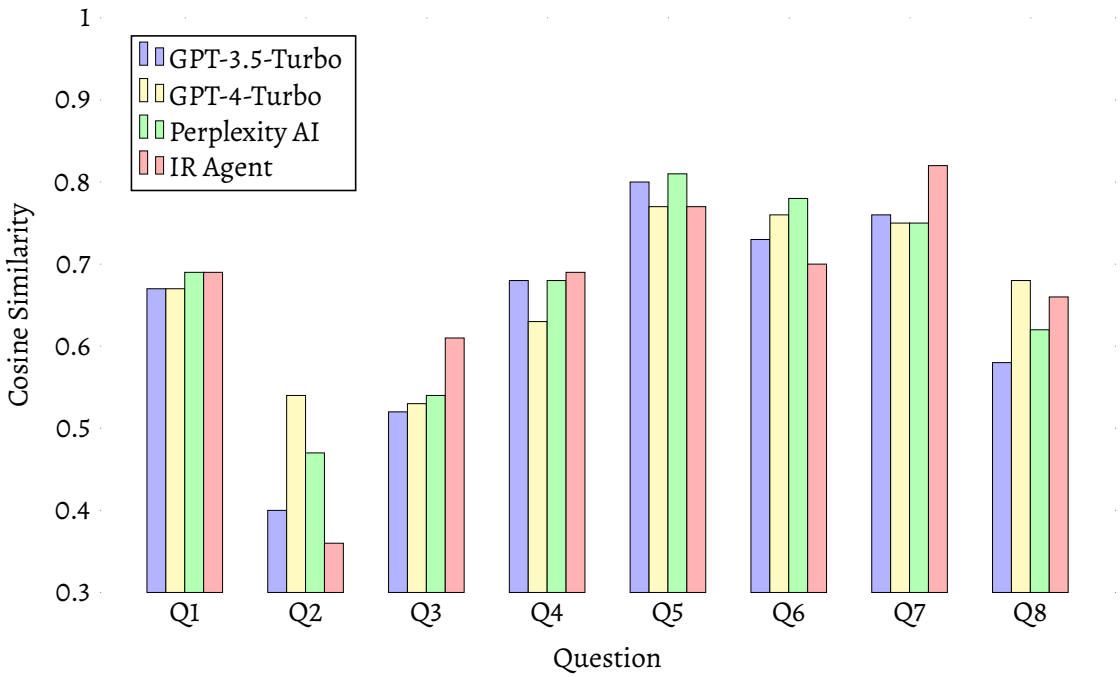

Figure 6.1: Semantic distances between the gold standard answer and the generated answer. The semantic distance is calculated as the cosine distance of the sentence embeddings.

The benchmarking results are shown in figure 6.1. We can observe, that all tested services perform similar for each question. There is no service that outperforms the others. Differences in similarity are visible between question, hinting at some questions being more difficult to answer than others. The main reasons for these results are the choice of question-answer pairs, and using the cosine similarity to measure the answer quality.

The 10 questions used in the benchmark are not nearly enough to capture all kinds of questions. Furthermore, as the answers are direct text exempts from the humanities journal.

6.2 Qualitative Evaluation

This section will give a qualitative evaluation of the benchmark results shown in section 6.1. I will discuss some examples for QA pairs that had good results and some with bad results. While these example do not allow for statistically relevant statements about the agent performance, they can still provide useful insights.

Good Example

The agent was able to generate a good answer for question eight:

> Who was likely the user of the portable breviary from San Zeno Maggiore?

The corresponding answer is:

> "The fact that the breviary transmits texts decidedly pertinent to the abbot reveals quite clearly that the book was intended for no less than the abbot of San Zeno Maggiore."

The generated answer of the IR agent contains the correct statement:

> "The user of the portable breviary from San Zeno Maggiore was likely the abbot for personal use rather than for the monastic community's collective ritual practices. <id:1>"

Here, the retrieval worked better, and the agent added a document ID after the answer, indicating it used document one from the five retrieved documents as context. Indeed, document one contains the relevant information to answer the question:

> "late-fifteenth-century breviary from the Abbey of San Zeno [...] it was probably made for the abbot's personal use rather than for the monastic community's collective ritual practices."

In this case all agent steps have worked. First, when adding the article to the database the required information stayed together in a single chunk. Then, the agent was able to generate a query to the database that yielded the relevant chunk. Finally, the RAG call to the LLM produced an answer that relied on that specific chunk.

A Bad Example

The worst answers were produced for short questions that do not contain a lot of context. All tested services performed worst for question two, which asks for a term definition:

> "What is a qersu?"

The article passage used as ground truth is:

> "Some consultations took place in a sacred enclosure by the river called a qersu."

This is a hard example, as the question contains minimal context for the term "qersu". As we will see, the word is used to describe multiple things.

In figure 6.1 we can see that GPT-4 gave the best answer, while the IR agent produced the worst. The IR agent answer is not a wrong term definition, but the agent completely fails at answering the question. The output of the IR agent is

> "I'm sorry, but the term 'qersu' is not mentioned in the provided context documents. <?>"

This would be a good result, if the agent did not have access to any sources containing the necessary information. In this case, the journal was added to the agent database, so ideally an actual answer defining the term should be generated. Answers of this kind hint at problems in the retrieval step. I will look at retrieval problems in more detail in section 6.3.

In comparison, the GPT-4 answer includes the sentence:

> "[...] In Hittite culture, a "qersu" was essentially a type of temple or sanctuary. [...]"

Remember that GPT-4 was prompted without additional context, which means that this information was learned during the pre-training phase of GPT-4.

Interestingly, Perplexity AI gives an answer and provides sources, but returns a different definition of the term:

> "A "qersu" is a type of coffin characterized by [...] in Ancient Egypt."

It also provides a source that supports the claim, but chooses a different one from the web. Note that I did not upload the journal as a source for Perplexity AI, this might have yielded a better answer.

We have seen that there are different reasons for bad answer of the IR agent. In the next section, I will put these reasons in several categories.

6.3 Points of Failure

When testing the IR agent, different types of errors occurred. In this section I will describe the three main problems I found, ability choice, complying to the system format and retrieval errors.

Due to the non-deterministic generation of LLMs, different answer can be generated for the same prompt. The answer is likely semantically similar to the previous one, but when a model has to choose between two abilities, a small difference can decide if the agent succeeds or fails at completing the task.

Another challenge when developing agents is the natural language interface of LLMs. While it makes describing the task easier for humans, it complicates the internal communication in the software. For example, AutoGPT needs the agent to answer in a structured system format. While this works most of the time, it is not guaranteed that the answer complies with the format. A small deviation such as a missing bracket can break the parsing process.

Retrieval problems can have different reasons but have the same result. The agent tries to generate an answer using context documents that are non-relevant to the question.

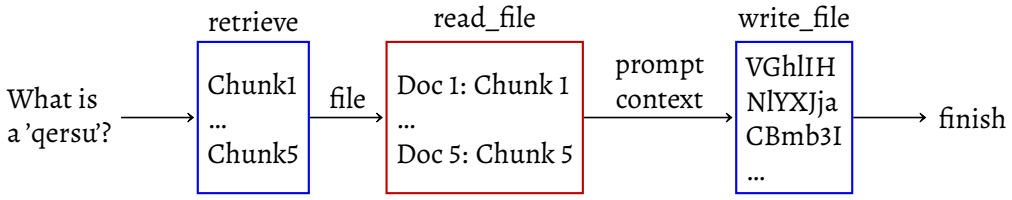

Figure 6.2: An agent run that failed because of an incorrect action choice. The agent read the context file into the next prompt, instead of using it for the answering ability. In the last step it still produced an output file, so for an outside observer that only looks at input and output, the agent behaved correctly, but the answer is just random characters.

Ability Choice

The agent sometimes chooses abilities that are not suited to progress towards the goal. For example, when the agent has read and write abilities, the agent decides to read the file containing the retrieved context in some instances. The correct choice would have been to proceed with the "answer" ability. In some instances, choosing the wrong ability leads to useless answers, as shown in figure 6.2.

A solution to this problem is to either remove the read and write abilities, or to guide the agent with explicit instruction in the prompt on which ability to use. For a true autonomous agent, these solutions are not ideal, because we want the agent to make the decisions with minimal user intervention. Fixes like removing abilities bring the agent closer to fixed pipeline solutions. Another solution could be to retry the same or modified prompts. This is feasible if the user does not demand fast responses. In the case of IR, the agent should answer reasonably fast and give correct answers. Therefore, I opted to remove all abilities that were not needed for RAG.

Furthermore, the model choice is important for decision-making. GPT-4 is a lot better than GPT-3 at making good decisions, following tasks and providing criticism and reasoning. For this reason, GPT-4 was used as the agent controller LLM.

LLM Answer Format

After receiving the answer to the step prompt, the agent has to parse it. The parsing step can only work if the answer is generated in the exact format specification. It is not certain that an LLM adheres to the system format.

Larger models like GPT-4 are better than smaller ones like GPT-3, but also cost more. Other models improve at structured answers with fine-tuning but might lose capabilities in other areas as a result. A software based solution could be to catch common mistakes from LLMs such as missing brackets or missing commas in JSON. While some mistakes

```
These manuscripts are significant evidence of the different aspects of a
||
5 Chardonnens 2007.


Introduction | 7

performance and its adaptability to different periods and contexts. A manu-
script of this kind which is produced communally prepares and shapes the
    col-
lective performance.
Two different aspects of the temporality of a performance are highlighted by
Mathieu Ossendrijver in his article on Babylonian and Assyrian sources pro-
duced during the first millennium BCE.
```

Listing 6.3: An example of bad chunking. Additional information splits the article content into two parts. This can hinder the quality of the produced embeddings, as the chunk has semantically mixed content.

can be mitigated with this approach, it is not feasible for large-scale applications as there are too many places where the format can break. Similar to ability choice mistakes, another solution is to retry and hope for a parsable answer. I chose this option to develop and test the IR agent.

Document Retrieval

There are two main reasons for failed retrieval of context for a user question, bad chunking and implicit query rewriting. In section 6.2 we have seen an example question where the agent could not answer the question given the retrieved context. This should not happen, as the dataset questions are based on the content the agent has access to. An answer of this kind hints at problems in the retrieval step before answering. I identified two main reasons for bad retrieval of information for a question, bad chunking and implicit query rewriting.

Bad chunking happens in the ingestion step, where to source document is added to the vector database. The chunking strategy of the IR agent is very simple, the full journal text is split at sentence level, until the resulting splits reach a desired length. In some instances, this strategy splits articles and paragraphs at places where the semantics do not change. For other cases, one chunk contains unrelated sections of a text. Both types of bad chunking decrease the quality of the embeddings, because the chunk does not have

a single, clear “meaning”,

Implicit query rewriting happens every time the IR agent chooses the retrieve ability. The retrieval ability has a “query” parameter that is set when the LLM answers the step prompt. It is the task of the language model to convert the user prompt into a query to the vector database. Which query retrieves the best documents for the following answer generation is not clear.

7

Conclusion

The recent advances in language modeling with large language models (LLMs) sparked research in building autonomous agents that use LLMs to make decisions. LLM agent research focuses on modeling human interaction and gathering information from the web. While there are information retrieval (IR) services like Perplexity AI that use LLMs, there has not been extensive research on LLM agents for IR. Most of these services use static pipelines which follow a fixed sequence of steps when producing an answer to a user prompt. In this thesis I investigated the usage of AutoGPT for an IR agent. Specifically, I studied the application on local IR in contrast to most LLM agent research which deals with web scraping tasks.

I started out with an introduction to the AutoGPT project and analyzed the components in the context of information retrieval. The distinctive feature of AutoGPT and similar projects is the In the analysis, I concluded that the best way to test AutoGPT for local IR, would be to create an agent built upon the AutoGPT Forge agent.

The IR agent uses retrieval augmented generation (RAG), a method that improves prompt answering by using retrieved documents as context.

To evaluate the IR agent I had to create a benchmark. As the IR agent is only accessible in the web interface and the computation speed is slow, I opted to use a small set of question and answer to perform a qualitative evaluation of the agent.

Using language model agents for IR tasks is an interesting approach, but needs to overcome some key challenges for usage in real-world applications. Most considerations about the agent structure are offloaded to the controlling LLM. Therefore, the reasoning capabilities of the LLM is the deciding factor for an LLM agent. This creates the need to use more expensive language model like GPT-4 rather than a cheaper one, as the controlling LLM. Combined with frequent API calls that use long prompt messages for every step, this leads to API costs adding up quickly. Especially in longer runs, where the ac-

tion history gets longer with each past step. The agent implementation mostly deals with calling the LLM and parsing its answer into the next action. There is no way to really say what led to a specific action sequence besides trial and error experimentation with different prompt templates. For IR systems in production, fixed RAG pipelines still seem to be a better fit. To create reliable systems with traceable answers, LLMs should be used for specific tasks not as controllers for the entire system.

7.1 Future Work

The IR agent could be extended with web searching capabilities to include external information on demand. This would add an extra dimension as the agent has to decide when local information is enough and when to look for online resources.

Work could be done to research how unstructured data formats can be split up, such that the semantic structure is kept together in chunks. For RAG, the ingestion of documents into the vector database is a crucial step. The popular chunking methods for unstructured data like PDFs simply split the text to get a certain chunk size. In listing 6.3 we have seen a chunk where the article text is split in the middle by metadata about the section and publisher. Semantically split chunks could improve the embeddings used for retrieval. Such methods could improve the performance of RAG pipelines.

To improve LLM agents, more work on improving LLMs in general has to be done. In section 6.3 I described the points of failure regarding the IR agent. They are mostly related to the capabilities of the underlying LLM. At this point in research, the construction of LLM agents is merely a software engineering problem. The solutions are either prompt or fine-tuning related. Most parts of the 3 project described in chapter 3 tries to handle the underlying LLM. The code for agent logic is relatively small, as most of the agent decision logic explained in section 2.2 is offloaded to the LLM. In short, the crucial factor for performance the base capability of the LLM.

Bibliography

[1] *Agent Protocol*. AIEF. (Visited on 03/01/2024).

[2] *Assistants Overview - OpenAI API*. (Visited on 03/20/2024).

[3] *AutoGPT Agent Introduction*. (Visited on 03/19/2024).

[4] *AutoGPT Forge GitHub*. (Visited on 03/01/2024).

[5] Banerjee, D., Singh, P., Avadhanam, A., and Srivastava, S. *Benchmarking LLM Powered Chatbots: Methods and Metrics*. Aug. 8, 2023. DOI: 10.48550/arXiv.2308.04624. arXiv: 2308.04624 [cs]. (Visited on 03/21/2024). preprint.

[6] Brita, A., Karolewski, J., Husson, M., Miolo, L., and Wimmer, H., eds. *Manuscripts and Performances in Religions, Arts, and Sciences*. De Gruyter, Nov. 20, 2023. ISBN: 978-3-11-134355-6. DOI: 10.1515/9783111343556. (Visited on 02/29/2024).

[7] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language Models Are Few-Shot Learners. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. (Visited on 03/08/2024).

[8] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. Dec. 11, 2014. DOI: 10.48550/arXiv.1412.3555. arXiv: 1412.3555 [cs]. (Visited on 03/06/2024). preprint.

[9] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: *Proceedings of NAACL-HLT*. 2019.

[10] *Embeddings - OpenAI API*. (Visited on 03/22/2024).

[11] Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. *Convolutional Sequence to Sequence Learning*. July 24, 2017. DOI: 10.48550/arXiv.1705.03122. arXiv: 1705.03122 [cs]. (Visited on 03/06/2024). preprint.

[12] Han, Y., Liu, C., and Wang, P. *A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge*. Oct. 18, 2023. DOI: 10.48550/arXiv.2310.11703. arXiv: 2310.11703 [cs]. (Visited on 04/05/2024). preprint.

[13] Hatalis, K., Christou, D., Myers, J., Jones, S., Lambert, K., Amos-Binks, A., Dannenhauer, Z., and Dannenhauer, D. Memory Matters: The Need to Improve Long-Term Memory in LLM-Agents. In: *Proceedings of the AAAI Symposium Series* 2(1):277–

280, 1 2023. ISSN: 2994-4317. DOI: 10 . 1609 / aaaiss . v2i1 . 27688. (Visited on 04/05/2024).

[14] 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. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474.

[15] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. *Lost in the Middle: How Language Models Use Long Contexts*. Nov. 20, 2023. DOI: 10.48550 / arXiv . 2307 . 03172. arXiv: 2307 . 03172 [cs]. (Visited on 02/29/2024). preprint.

[16] *LlamaIndex SentenceSplitter*. (Visited on 03/24/2024).

[17] Manning, C., Raghavan, P., and Schuetze, H. *Introduction to Information Retrieval*. Online edition. Cambridge University Press, Apr. 2009. ISBN: 0-521-86571-9.

[18] Nakajima, Y. *BabyAGI*. Apr. 2023. (Visited on 03/08/2024).

[19] OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., et al. *GPT-4 Technical Report*. Mar. 4, 2024. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774 [cs]. (Visited on 04/05/2024). preprint.

[20] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training Language Models to Follow Instructions with Human Feedback. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 27730–27744.

[21] Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. Generative Agents: Interactive Simulacra of Human Behavior. In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST'23. ACM, Oct. 2023. DOI: 10.1145/3586183.3606763.

[22] *Perplexity AI*. Perplexity AI. (Visited on 03/20/2024).

[23] Pwuts *AutoGPT Forum Post*.

[24] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. *Improving Language Understanding by Generative Pre-Training*. OpenAI, 2018.

[25] Robertson, S. and Zaragoza, H. The Probabilistic Relevance Framework: BM25 and Beyond. In: *Foundations and Trends in Information Retrieval* 3(4):333–389, Apr. 1, 2009. ISSN: 1554-0669. DOI: 10.1561/1500000019. (Visited on 03/06/2024).

[26] Russel, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Fourth Edition, global edition. Pearson, 2022. ISBN: 978-1-292-40113-3.

[27] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. *Toolformer: Language Models Can Teach Themselves to Use Tools*. Feb. 9, 2023. DOI: 10.48550/arXiv.2302.04761. arXiv: 2302.04761 [cs]. (Visited on 03/18/2024). preprint.

[28] Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. *HuggingGPT: Solving AI Tasks with ChatGPT and Its Friends in Hugging Face*. Dec. 3, 2023. DOI: 10.48550/arXiv.2303.17580. arXiv: 2303.17580 [cs]. (Visited on 03/18/2024). preprint.

[29] Significant Gravitas *AutoGPT*. Mar. 2023. (Visited on 03/18/2024).

[30] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017.

[31] Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al. *A Survey on Large Language Model Based Autonomous Agents*. Aug. 2023. DOI: 10.48550/ARXIV.2308.11432. arXiv: 2308.11432 [cs.AI]. preprint.

[32] Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., Chi, E., Le, Q. V., and Zhou, D. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 24824–24837.

[33] Xu, B., Yang, A., Lin, J., Wang, Q., Zhou, C., Zhang, Y., and Mao, Z. *ExpertPrompting: Instructing Large Language Models to Be Distinguished Experts*. May 23, 2023. DOI: 10.48550/arXiv.2305.14688. arXiv: 2305.14688 [cs]. (Visited on 03/14/2024). preprint.

[34] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In: *Advances in Neural Information Processing Systems* 36:11809–11822, Dec. 15, 2023. (Visited on 03/20/2024).

[35] Zhu, Y., Yuan, H., Wang, S., Liu, J., Liu, W., Deng, C., Chen, H., Dou, Z., and Wen, J.-R. *Large Language Models for Information Retrieval: A Survey*. Jan. 19, 2024. DOI: 10.48550/arXiv.2308.07107. arXiv: 2308.07107 [cs]. (Visited on 03/06/2024). preprint.