SMU Data Science Review

Volume 8
Number 1 *Spring 2024*

Article 6

Building Effective Large Language Model Agents

Sydney Holder
Southern Methodist University, sholder@smu.edu

Shreyash Taywade
AT&T, st6300@att.com

Building Effective Large Language Model Agents

Sydney A. Holder[1], Shreyash Taywade[2]
[1] Master of Science in Data Science, Southern Methodist University,
Dallas, TX 75275 USA
[2] AT&T Chief Data Office, 2900 W Plano Parkway, Plano, TX 75205 USA
sholder@smu.edu, st6300@att.com

Abstract. The advancement of large language models (LLMs) has significantly expanded the influence of artificial intelligence across various sectors. This paper explores building LLM agents to power applications and examines what is necessary to build an efficient and helpful AI assistant. The research investigates the core components necessary to create specialized agents, facilitate collaboration in problem-solving, and improve human task performance. The development and application of tools designed to augment the capabilities of LLM agents are also explored. The paper addresses the potential risks of the unknowns, such as hallucinations, which can compromise the success of agent-based solutions within LLM applications. The potential application areas for LLM agents and the broader implications of these findings on AI research and application development are discussed.

1 Introduction

Large Language Models (LLMs) are artificial intelligence (AI) designed to process and understand natural human language. These models employ deep learning architectures like transformers to process and generate text based on their input. They are trained on vast amounts of text data from diverse sources, such as books, articles, and websites, allowing them to learn patterns, relationships, and context within the language. As a result, large language models can generate contextually relevant, coherent, and human-like text in response to user inputs. LLMs have a wide range of applications, including natural language understanding, language translation, content generation, sentiment analysis, and assisting with various tasks through conversational interfaces.

A large language model agent is created by engineering a prompt to define a role for the LLM to respond in. An agent is responsible for deciding what steps to take to complete a task. It uses a language model as a reasoning engine to make these decisions. Autonomous language agents can automatically solve complicated tasks and interact with environments, humans, or other agents.

Designing and implementing an extensive language model application that successfully leverages the capabilities of agents can be a complex yet rewarding task. This paper investigates various aspects of multi-agent-based applications, including the methodologies and techniques used for creating agents, the strategic deployment of

multiple agents within an application, the degree of autonomy granted to agents, and the evaluation of agent effectiveness in alignment with specific use cases. Furthermore, the paper will delve into the challenges and potential drawbacks of developing large language model applications that rely heavily on agents. It will discuss the factors that may contribute to the success or failure of such applications, including scalability, cost, ethical considerations, and the impact of agent interactions on the overall system performance.

By providing a thorough analysis of those topics, this paper aims to contribute to understanding large language model agent applications and their effective integration of the agents. It ultimately aims to assist researchers, developers, and business units in making informed decisions when designing and deploying agent-based systems..

2 Literature Review

Numerous studies have explored and implemented a broad spectrum of cognitive approaches that can be integrated into prompts for agent generation, thereby enhancing their overall performance and adaptability. This paper examines these studies to glean insights into the most promising approaches and their potential applications.

Large Language Model (LLM) agents are intelligent systems capable of processing vast amounts of unstructured text data to generate coherent and contextually appropriate responses. These agents form the backbone of various problem-solving applications, from facilitating consensus-building during web discussions to simulating economic situations (Horton, 2023). These agents' key attributes are their ability to learn and adapt, effectively leveraging their training datasets to generate diverse responses or solutions (Askari et al., 2023).

The efficacy of LLM agents does not solely depend on the number of agents deployed; it is also contingent on how effectively these agents are designed and trained. For instance, less advanced large language models consistently chose similar options despite the different endowments provided during a fairness judgment experiment. In contrast, more advanced models displayed variation based on specific endowments. This highlights the importance of quality over quantity; having well-trained AI systems capable of understanding context-specific nuances will likely lead to better performance outcomes (Askari et al., 2023).

Integrating multiple intelligent agents into larger systems or applications is another crucial aspect of LLMs. For example, the 'Agent for Recommending Information Relevant to Web-based Discussion' utilized GPT-3 to generate queries and a BERT-based model for classifying information based on relevance within the ongoing discussion context (Kinoshita & Shiramatsu, 2022). Similarly, the Auto-GPT framework incorporated supervised or imitation-based learners into the Auto-GPT scheme without requiring fine-tuning of foundational LLMs, significantly enhancing performance in online decision-making benchmarks (Yang et al., 2023).

The ability of these agents to interact effectively with their environment is also critical. This includes their capacity to parse complex instructions into executable plans, a feature that HTML-T5++ exhibits, outperforming human-level performance on MiniWoB and achieving the highest zero-shot performance on CompWoB among all tested models (Furuta et al., 2023). Furthermore, LLM agents have also been successfully applied in the recruitment sector to automate resume screening (Gan et al., 2024), in data analytics tasks through the TaskWeaver framework (Qiao et al., 2023), and in the creation of multiple specialized agents to form an AI team tailored to various tasks through the AutoAgents framework (Chen et al., 2023).

Edward Chang and Emily Chang (2023) delve into how effective communication between different agents can enhance the overall success rate of LLM applications. They observed that agents incorporated their extensive knowledge in various agent-to-agent conversations, basing their discussions on reason. Their adept use of cultural references and literary connections enriched the conversation and broadened the scope of comprehension, making their points more relatable. In various cases, the agents enhanced the depth of the conversation and fostered interconnectedness between topics (Chang et al.; E., 2023).

WebArena is a realistic environment for evaluating the performance of autonomous agents on web-based tasks based on high-level natural language commands. The benchmark focuses on diverse, long-horizon tasks humans routinely perform online and evaluates task completions' functional correctness. The success rate achieved by these agents needs further improvement, highlighting the challenges in handling complex tasks within realistic environments (Zhou S. et al., 2023).

Experimentation with agents has proven promising for conducting pilot studies via simulation before testing in the real world. This approach offers insights into human behavior at lower costs and in less time than other methods. By endowing these AI agents with baseline beliefs and putting them through various scenarios, the author demonstrates that this approach can qualitatively recover findings from experiments with actual humans (Horton, 2023).

An agent was developed to perform relevant-information-recommendation. This agent retrieves necessary discussion data from ongoing web discussions, generates query terms, and recommends related information based on the phase of the current discussion progress (Kinoshita & Shiramatsu, 2022).

The “GPT-in-the-loop” approach is a novel method that combines the advanced reasoning capabilities of generative pre-trained transformers with multi-agent systems. Researchers achieved superior decision-making and adaptability without extensive training when integrating GPT-4 into agents. The authors use the framework for the Internet of Things to incorporate GPT into agent-driven applications. The results show that GPT's iterative approach showcases significant adaptability and improvement compared to traditional methods and human-generated solutions (Nascimento et al., 2023).

SPRING is a novel approach that employs an LLM agent to read a paper about a game and use what is learned to play the game. The SPRING agent can reason and make decisions in complex open-world game environments by extracting critical information from the paper and incorporating it into a QA summarization framework. The framework demonstrates improved performance compared to traditional reinforcement learning methods while requiring no training, showcasing the potential of LLMs for understanding and reasoning with human knowledge in gaming contexts (Wu et al., 2023).

AutoML-GPT is a framework that integrates agents into the automated machine learning (AutoML) process, aiming to simplify model development by automating various stages of the machine learning workflow. It consists of two types of agents: the Reasoning agent and the Coding agent. The Reasoning agent understands human requests and plans tool usage sequences, while the Coding agent reads documentation and modules, generates AutoML code, and executes it. AutoML-GPT demonstrated competitive performance compared to other renowned AutoML frameworks while significantly reducing the time and effort required for machine learning tasks (Tsai et al., 2023).

AGENTS is an open-source library containing key and helpful features for building LLM agents. The framework provides support for various scenarios such as single-agent systems (e.g., chat bot), multi-agent systems (e.g., cooperative or competitive interactions), human-agent interaction (e.g., debating), web navigation tasks (e.g., customer service agent using web search engines), among others (Zhou, W. et al., 2023).

LLMs are intelligent systems capable of processing vast amounts of unstructured text data to generate coherent and contextually appropriate responses. They have been successfully applied in a range of problem-solving applications. However, areas still need improvement, particularly in their ability to solve complex real-world tasks and their interaction with their environment. Despite these challenges, the potential of LLMs in computational social science, economic simulations, and consensus building during web discussions remains promising (Liu et al., 2023).

LLMs are intelligent systems capable of processing vast amounts of unstructured text data to generate coherent and contextually appropriate responses. They have been successfully applied in a range of problem-solving applications. However, areas still need improvement, particularly in their ability to solve complex real-world tasks and their interaction with their environment (Suri et al., 2024). Despite these challenges, the potential of LLMs in computational social science, economic simulations, and consensus building during web discussions remains promising (Li et al., 2024).

3 Analysis of Agents

3.1 Range of Autonomy in Agents

Large language model (LLM) agents exhibit a range of autonomy, from the least to the most. The least autonomous agents focus on specific tasks and rely heavily on short descriptions. They may have limited interaction with their environment and other agents and limited long-term memory capabilities.

As we move towards more autonomous LLM agents, they incorporate features such as tool usage, long-short-term memory, multi-agent communication, human-agent interaction, and symbolic control. These advanced agents can interact with environments or other agents over time while maintaining a record of their previous actions and decisions.

In addition to these frameworks explicitly designed for developing language models with varying degrees of autonomy in performing tasks or interacting with humans or other AI entities in real-world applications like customer service consulting, programming, writing, teaching, etc., there are also projects such as Auto-GPT, BabyAGI, and SuperAGI aimed at achieving AGI by enabling users build customize test tune deploy state-of-the-art without much coding effort required from them (Liu et al., 2023).

3.2 Components of LLM Agents

3.2.1 Cognitive Function

An indispensable segment of an LLM agent is its cognitive function or "brain," which is an intricate complex that can be elucidated into four significant parts. Each part is pivotal in how the LLM agent processes information, makes decisions, and interacts with its environment. The components mimic the cognitive functions of the human brain, enabling the agent to understand, process, and generate language in a sophisticated and human-like manner.

1. **General Knowledge:**

   Large language models, such as GPT-4, are the base for establishing an intelligent agent. These models, grounded in machine learning principles, utilize transformer-based architecture characterized by self-attention mechanisms. GPT-4, trained exhaustively on an expansive corpus of internet text, becomes an agent's foundational source of general knowledge. This broad base empowers the agent to converse on, comprehend, and generate coherent narratives across diverse topics. It can respond to factual inquiries and even demonstrate creative writing capabilities without necessitating supplementary information integration.

However, it is pivotal to understand that GPT-4, acting as a base for general knowledge, essentially operates as an advanced pattern recognition system. It learns from the vast data it has been trained on to generate contextually accurate and pertinent responses. This learning process involves a complex data encoding and decoding sequence, where the model learns to predict the probability of a word occurrence given the preceding words in a sentence. Through a multitude of processing layers and millions of parameters, the model eventually acquires the ability to generate contextually relevant and coherent sentences.

Nonetheless, there are inherent limitations when utilizing the general knowledge provided by an LLM in isolation. While an LLM agent constructed on the GPT-4 model can answer a wide range of inquiries with impressive accuracy, it must improve in specific or private domains, such as internal company data and processes. Furthermore, while the model can generate creative text based on the patterns it has learned, it does not know which documents specifically contributed to its training set, nor does it have access to any classified, proprietary, or confidential information. Thus, the utilization of LLM agent outputs necessitates an understanding of these limitations. A scholarly appreciation of these parameters will ensure the agent's capabilities are leveraged effectively and responsibly.

2. Specific Knowledge

Integrating RAG and Prompting techniques into an LLM agent significantly enhances its functionality, giving it a distinct personal touch, mainly when dealing with domain-specific or private data. With its complex cognitive structure, the LLM agent is notably enhanced by a component known as Specific Knowledge, facilitated by a pipeline known as Retrieval-Augmented Generation (RAG). This component effectively supplements the general knowledge of the LLM with domain-specific expertise, thereby diversifying and deepening its understanding of a variety of subjects.

RAG is a sophisticated tool that enables an LLM to acquire and process domain-specific, private, or even novel information. The distinct advantage of a RAG lies in its ability to utilize a knowledge retriever to source relevant information for the task at hand. Given the sheer volume of textual data, vector embeddings are commonly employed to extract similar text from a vector database. However, the versatility of a RAG pipeline allows for querying SQL databases, Graph databases, and even internet searches. This flexibility significantly enhances the depth of understanding of an LLM agent on previously unfamiliar subjects.

3. Thought Process

The thought process of an LLM (Large Language Model) agent is created through a combination of training on massive amounts of text data, prompt design, and integration with various tools or APIs. The agent's reasoning mechanism is designed to respond similarly to how a human would react to prompts, making it capable of understanding and processing natural language input.

In multi-agent systems, LLMs are used as proxies for human participants in surveys or experiments who respond to questions posed through prompts. This enables researchers to study these agents' behavior and decision-making processes in different scenarios. Incorporating LLMs into agents has two benefits: an enhanced reasoning mechanism for each agent and more efficient communication across diverse multi-agent landscapes (Askari et al., 2023).

Using LLM-based agents in strategic game experiments has several advantages. These models can capture latent social information like economic laws, decision-making heuristics, and common social preferences. They are trained on large corpora containing written text where people reason about and discuss such matters.

Prompt design plays an essential role in the performance of language-based agents. Adapting LLMs for decision-making tasks often involves non-trivial prompt design and memory retrieval mechanisms that dynamically construct the agent's context. Moreover, integrating prior knowledge from various sources helps improve their overall performance.

LLM-powered autonomous agents have significant potential applications across multiple domains like customer service consulting programming service, reducing the human effort required roles such as customer service consulting programming writing teaching, etc. reducing human effort required roles such customer service consulting programming writing teaching, etc., reducing the human effort required roles.

4. Personality

Finally, the personality of an LLM Agent is shaped by a combination of prompting and fine-tuning. Personality in humans is a unique blend of characteristics and qualities that define an individual's behaviors and attitudes. Creating a personality for an LLM agent infuses uniqueness and consistency into the agent's responses, thereby giving it a distinctive character.

Prompt Engineering is a technique that involves crafting prompts that set the tone, style, and context for the agent's responses. This can elicit different personalities, such as formal, humorous, or empathetic, or even different ways an agent might respond, such as adopting a pirate's speech pattern. Prompts guide the agent's responses while fine-tuning adjusts the agent's behavior to align with specific characteristics or traits. This creates a unique "personality" for the agent, making it more relatable and engaging to users.

Fine-tuning the parameters of the LLM allows the developer to adjust the agent’s behavior to align better with the desired characteristics. It helps to mold characteristics, influence the behavior, and shape the agent's responses. This process is integral to imbuing the LLM agent with a unique 'personality' that can make interactions more engaging, relatable, and user-friendly.

Fine-tuning is training the LLM on a specific dataset after training it on a broad dataset. The specific dataset is typically smaller and more specialized, often curated to reflect the desired traits or characteristics. This procedure adjusts the LLM's output, guiding it towards generating responses that align with the given personality traits. Fine-tuning can also adjust the LLM's responses to specific prompts or situations. For instance, an LLM agent can be fine-tuned to respond more positively or neutrally, depending on the desired personality trait. It can also be trained to exhibit a specific attitude or tone, such as humor, empathy, or enthusiasm.

3.2.2 Action-ability

The second primary component of an LLM agent is the ability to take action on thoughts or generate thoughts through observation. Agents can act by utilizing tools, which are instrumental in empowering them to transform their thoughts into actionable tasks. Thus, tools play a pivotal role in the functionality of the LLM agent. There are several types of tools that the agent can utilize, each contributing distinct capabilities to the agent's operational proficiency.

Some of the most common tools for LLM Agents allow the agent to generate, execute, and test code. These tools can be powered by code interpreters, which are instrumental in writing and executing code snippets in various programming languages, such as Python, JavaScript, or SQL. These interpreters transform abstract algorithms into functional programs, enabling the LLM agent to carry out specific tasks efficiently and effectively (Yuan et al., 2024).

Another tool that gives an LLM agent the ability to take action and gather information is the ability to make API calls. This tool allows the agent to access real-time data, providing a dynamic dimension to the agent's capabilities. For instance, an agent can call an API to obtain current weather statistics for a specific location, thereby providing users with timely and relevant information.

A newer but important capability that agents can now access is the ability to ‘see.’ Recent advancements in LLM models give them the ability to understand and even generate images, which, when given to an LLM agent, takes it to the next level.

By harnessing the power of these tools, an LLM agent can not only process and reason with information but also take direct action based on its thoughts, significantly enhancing its utility and efficiency.

3.3 Agent Examples

Various agent types can include:

1. Reasoning/Planning Agent: A reasoning agent is responsible for understanding human requests, planning the sequence of tool usage, monitoring subtasks, and combining tools to achieve the desired outcome.

2. Coding Agent: A coding agent can perform actions such as reading documentation and module-related packages within a code base, generating code for executing tasks, and returning code execution results.

3. Master Agent/Group Chat Manager: These agents focus on managing interactions and relationships. They usually have the most information and context, so they can effectively guide problem-solving and/or discussion.

4. Data Agent: A data agent leverages an LLM to assist with data analysis tasks. They can extract insights from datasets, generate reports, and provide recommendations. One use case where one might find a data agent is in financial analysis. Here, a data agent can answer questions about a company's performance based on financial statements.

Overall, large language model agents combine reasoning abilities with coding skills and memory management features that allow them to understand natural languages and perform complex tasks autonomously by leveraging various APIs/tools available on the web or in any specific environment they operate in.

4 Methodology

4.1 Define the problem

The initial step in building a large language model application involves clearly defining the problem statement and understanding the requirements. This process includes identifying the specific task or tasks the model is expected to perform and understanding the context in which it will be used. It is crucial to consider the user's needs and the type of interaction they will have with the model. This step also involves understanding the model's use's ethical, safety, and fairness considerations.

When building LLM agents, creating an environment where the agent can deeply understand the application domain and clearly define the problem is essential. This is because an agent might be designed to handle a specific aspect of the more significant problem or domain. For example, in a customer service scenario, one agent might handle billing inquiries, another technical support, and another product information. This division of labor requires a clear understanding of the domain and problem to ensure that each agent is trained with the appropriate data and fine-tuned to handle its specific tasks.

Understanding the application domain and obtaining a clear definition of the problem are fundamental steps in developing LLM agent(s). They guide the choice of data and resources for training and fine-tuning the models, and they are particularly crucial when building and fostering collaboration among LLM agents (Wu et al., 2024). For this paper, we will build an LLM agent that acts as a personal assistant and

demonstrates the use of cognitive function and actionability as described in the key components section above.

4.2 Model Selection

Choosing an appropriate large language model is a pivotal component in the process of constructing LLM agents. This selection notably influences the agent's functionality, flexibility, and efficiency. The selection of an apt model for a specific task necessitates an in-depth examination of several elements. The initial step is to comprehend the nature and complexity of the task. For instance, tasks that necessitate understanding context and human-like text generation may be suited to models such as Generative Pretrained Transformers (GPT).

Conversely, for tasks that require comprehension of relationships between entities, models like Bidirectional Encoder Representations from Transformers (BERT) might be more fitting (Chen et al., 2023). Once the task is clearly understood, the volume and quality of available training data should be considered. Larger, more intricate models typically necessitate more data; however, data quality is equally important. High-quality, pertinent data can enhance model performance, irrespective of the model's size.

Creating an LLM agent collaboration involves amalgamating multiple LLM agents to accomplish intricate tasks collectively. This collaboration can be horizontal, where all agents possess the same complexity level and work on different task components, or vertical, where more straightforward agents relay their outputs to more complex agents for additional processing. Such collaboration can augment the agents' performance by enabling them to specialize in various tasks and share their expertise.

4.3 Fine-Tuning

Once you have chosen the appropriate LLM, the next step is the fine-tuning process. Fine-tuning is a method used to improve the performance of a pre-trained model on a new task. You train the model on a specific task while keeping the pre-trained weights fixed. This helps the model adapt to the new task's nuances without losing the general language understanding it has acquired from pre-training.

Tuning a large language model for a specific task is a complex, multi-step process. It begins with pre-training, where the model is trained on a large corpus of text data to learn the underlying patterns and structures of the language. This process allows the model to develop a broad understanding of the language, including its grammar, common phrases, and even some context-based decision-making.

4.4 Architecture

For successful LLM agent solutions, especially when there is agent collaboration, the architecture must be carefully designed, distinct roles for each agent must be clearly defined, and effective agent communication must be ensured. The agents should undergo coordinated training, and their performance should be evaluated and optimized

regularly. With an appropriate build, rigorous methodologies, and continuous optimization, one can construct LLM agents that excel in various tasks.

Collaboration among LLM agents could include sharing insights, learning from each other's experiences, or coordinating to handle complex tasks. This requires resources such as a communication infrastructure that permits the agents to exchange information and coordination mechanisms for their actions. It also necessitates data that mirrors the collaborative aspects of the problem, such as multi-agent interaction transcripts or data about successful collaborations.

4.5 Implementation

In this study, an LLM Agent was developed to serve as a personal assistant to illustrate the principal characteristics of LLM agents. This assistant, constructed using the AutoGen library, can perform diverse tasks such as managing a calendar, emails, and a to-do list. AutoGen, an open-source framework developed by Microsoft, facilitates the creation of Language Model applications employing multiple agents that engage in conversation to solve tasks (Wu et al. et al., 2023). This framework streamlines the orchestration, automation, and optimization of complex LLM workflows, enabling developers to construct diverse conversation patterns that consider conversation autonomy, agent count, and conversation topology.

The implementation, detailed in the appendix, starts by importing the necessary modules and initializing the AI assistant and user proxy. The assistant is configured with a series of configurations, a termination message function, and parameters including temperature, timeout, and cache seed. The user proxy is set up with parameters such as the maximum number of consecutive auto-replies and the working directory.

The assistant is designed to perform several functions delineated in the script. These functions are decorated with @user_proxy.register_for_execution() and @assistant.register_for_llm(), indicating their registration for execution by the user proxy and inclusion in the assistant's LLM. The following tools were created:

1. ask_planner: This function asks a planner agent to create a plan for finishing a given task and verify the plan's execution result.
2. add_event: Function to add an event to a calendar.
3. get_events: Function to get calendar events between two times.
4. get_emails: Function to retrieve emails from a given path.
5. draft_email: Function to draft an email and save it to a given ‘outbox’ path.
6. add_todo: Function to add a to-do task to a task list.
7. get_todos: Function to get to-dos from the task list.
8. get_files: Function to get a list of file names in a specific directory.
9. read_file: Function to read the content of a given file.

The tools equip an LLM agent with the ability to gather information about its environment, interact with it, and execute tasks. These tasks range from managing a

calendar and handling emails to reading and writing data and files. Based on user commands, the developed agent can perform a spectrum of tasks, including email and calendar management, file handling, and task planning. It employs Microsoft's agent library AutoGen to create the AI assistant and execute the tasks.

4.6 Results

The code initiates a chat with the assistant, asking it to check recent emails, draft responses, and update the calendar or to-do list accordingly. This demonstrates the assistant's ability to handle complex tasks involving multiple functions. The assistant successfully performs the tasks, showcasing the potential of large language models in automating routine tasks.

The code output demonstrates the assistant's ability to handle complex tasks involving multiple functions. The assistant checks recent emails, drafts responses, and updates the calendar or to-do list accordingly. The assistant also summarizes class notes and drafts an email to share the notes. The assistant's responses are based on executing the functions defined in the script, demonstrating the assistant's ability to manage a calendar, handle emails, and manage a to-do list.

4.7 Future Enhancements

The AI personal assistants described above are already capable of performing a variety of tasks, such as managing a calendar, handling emails, and managing a to-do list. However, the addition of more agents and tools could potentially enhance its capabilities and make it a more complex and effective AI assistant.

Incorporating more agents could allow for the distribution of tasks among different AI entities, each specialized in a particular domain. For example, one agent could be dedicated to managing the calendar, another to handling emails, and another to managing the to-do list. This would allow for more efficient task management, leading to faster response times and improved performance. Also, adding more tools could expand the range of tasks the AI assistant can perform. For instance, tools could be added to manage finances, track health and fitness data, or even provide personalized news updates. This would make the AI assistant more versatile and useful to the user.

As delineated in the provided code, the AI assistant is already proficient in executing various tasks, such as managing a calendar, emails, and a to-do list. However, integrating additional agents and tools could augment its capabilities, creating a more sophisticated and effective AI assistant.

Incorporating more agents could facilitate the distribution of tasks among different AI entities, each specialized in a specific domain. For instance, one agent could be dedicated to managing the calendar, another to handling emails, and another to managing the to-do list. This would allow for more efficient task management, potentially leading to faster response times and improved performance.

Adding more tools could broaden the range of tasks the AI assistant can perform. For instance, tools could be added to manage finances, track health and fitness data, or even provide personalized news updates. This would enhance the versatility of the AI assistant, making it more useful to the user.

It is important to remember that adding more agents and tools could potentially enhance the capabilities of the AI assistant; careful consideration would need to be given to the design and implementation of these additions to ensure that they improve the overall effectiveness of the assistant.

5 Discussion

5.1 Cost

Hosting or using a large language model, such as OpenAI's GPT-4 or similar models, can entail significant costs due to various factors. Firstly, the computational power required for training and fine-tuning these models is immense, necessitating specialized hardware like GPUs or TPUs. This results in high electricity consumption and infrastructure costs. Secondly, the storage requirements for such models are also substantial due to the vast amounts of data they need to process and store. This leads to increased data storage, backup, and management expenses.

These models' ongoing maintenance and updates require a dedicated team of experts, which adds to the labor costs. In addition, there are often licensing fees associated with using pre-trained models or accessing APIs from service providers, which can be a recurring expense. Lastly, the environmental costs of running these models cannot be overlooked, as the energy consumption contributes to a larger carbon footprint. Overall, the costs of hosting or using a large language model can be substantial and should be carefully weighed against the potential benefits before committing to their implementation.

5.2 Context Length

Utilizing a large language model has its own set of limitations regarding context length. One major constraint is the model's maximum token limit, which determines the maximum number of tokens (words and punctuation) that can be processed in a single input. Exceeding this limit may result in truncation or omission of parts of the text, leading to incomplete or inadequate responses. Furthermore, as the context length increases, the model's ability to retain and process information can diminish, causing it to lose track of relevant details and potentially generate less accurate or coherent outputs.

Longer contexts also increase the computational requirements for processing, leading to higher latency and increased costs. Large language models may need help maintaining coherence and consistency over extended context lengths, as they are

more likely to generate irrelevant or repetitive content. In summary, the limitations of context length when using a large language model can impact the quality of generated responses, computational efficiency, and overall usability of the model in various applications.

5.3 Hallucinations

Hallucinations in large language models occur when the model fabricates information not present in the input or training data, which can present notable challenges and risks. This phenomenon can propagate inaccurate or misleading information, creating an output that seems plausible but is unfounded, potentially leading to user confusion or misinterpretation. These hallucinations can also prove hazardous, particularly in delicate contexts such as health or legal advice. Users may act upon this fabricated information, unaware of its inauthenticity, resulting in significant real-world consequences.

The unpredictability of hallucinations adds a layer of complexity to ensuring the responsible and ethical utilization of AI technologies. The difficulty in controlling or predicting when a model will hallucinate complicates the implementation of safeguards, necessitating both users and developers to approach the model's output with heightened caution. Moreover, hallucinations can engender trust issues. Frequent encounters with hallucinated information may lead users to question the model's reliability, potentially undermining its utility and credibility. Hence, AI developers must address this issue to maintain user trust and confidence in AI technologies.

5.4 Ethics

The utilization of large language models introduces an array of potential limitations and hazards, particularly those pertaining to uncertainties. A primary constraint is the unpredictability inherent in the model's outputs, resulting from its deficiency in comprehending the world and its complexities. Large language models do not possess 'consciousness' or 'understanding' like human cognition. Instead, they generate responses predicated on patterns derived from the data on which they were trained. This can potentially yield nonsensical, inappropriate, or even offensive outputs, contingent upon the input and the model's interpretation thereof.

The risk emanates from the fact that these models, despite their advanced sophistication, can unintentionally generate detrimental or misleading information. For example, inaccurate information could precipitate harmful real-world repercussions in sensitive fields such as medical or legal advice. Another uncertainty is how these models would respond to unprecedented situations or inputs outside their training data. Their performance under such circumstances could be more predictable and could lead to inaccuracies.

The issue of bias in AI constitutes another critical concern. If the training data is imbued with biased information, the model could inadvertently perpetuate or amplify these biases. This risk is particularly acute given that these models often operate as

'black boxes,' rendering their internal operations and decision-making processes largely inscrutable to users. The potential misuse of these models represents a significant hazard. In the wrong hands, such potent tools could be exploited to disseminate misinformation, generate deepfake content, or for other nefarious purposes. Consequently, managing the uncertainties associated with large language models is an essential task that necessitates ongoing diligence and vigilance from AI developers, users, and policymakers.

5 Conclusion

This comprehensive study into Large Language Models illuminates their transformative potential in application development, particularly for efficient AI agents. It offers crucial insights into the creation of specialized agents, their role in collaborative problem-solving, and their efficacy in human-centric tasks.

The study further explores the development of tools enhancing LLM agents' capabilities while highlighting potential challenges and risks, such as hallucinations, that could compromise their effectiveness. It underscores the vast application spectrum of LLMs, including natural language understanding, language translation, sentiment analysis, and task assistance through conversational interfaces.

Moreover, the research emphasizes the need for vigilance and continuous improvements to navigate uncertainties and biases in AI technologies. It contributes significantly to the evolving AI discourse, spotlighting LLMs' critical role in the AI landscape across various sectors.

This study successfully leveraged Microsoft's open-source AutoGen framework to develop an LLM agent serving as an effective personal assistant. This agent proficiently performed various tasks, including calendar management, email handling, and task planning. Its ability to execute complex tasks initiated by user commands exemplifies the potential of LLMs in automating routine tasks.

The assistant's skillful management of intricate tasks involving multiple functions demonstrates the potential of LLM Agents in automating tasks and engaging effectively with their environment. However, the study also reinforces the need for ongoing research and development to address AI technologies' uncertainties and biases. The insights derived from this research are pivotal in guiding the responsible and effective use of AI technologies.

References

1. Arian Askari, Mohammad Aliannejadi, Evangelos Kanoulas, and Suzan Verberne. (2023). A Test Collection of Synthetic Documents for Training Rankers: ChatGPT vs. Human Experts. In Proceedings of the 32nd ACM International Conference on Information and Knowledge Management.
2. Bo, Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, Minghua Ma, Pu Zhao, Si Qin, Xiaoting Qin, Chao Du, Yong Xu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Taskweaver: A code-first agent framework. *arXiv preprint*, 2023.
3. Chang, Edward & Chang, Emily. (2023). Discovering Insights Beyond the Known: A Dialogue Between GPT-4 Agents from Adam and Eve to the Nexus of Ecology, AI, and the Brain.
4. Chen, G., Dong, S., Shu, Y., Zhang, G., Sesay, J., Karlsson, B. F., ... & Shi, Y. (2023). Autoagents: A framework for automatic agent generation. arXiv preprint arXiv:2309.17288.
5. Furuta, Hiroki, et al. "Exposing Limitations of Language Model Agents in Sequential-Task Compositions on the Web." ICLR 2024 Workshop on Large Language Model (LLM) Agents.
6. Gan, C., Zhang, Q., & Mori, T. (2024). Application of llm agents in recruitment: A novel framework for resume screening. arXiv preprint arXiv:2401.08315.
7. Hong, S., Lin, Y., Liu, B., Wu, B., Li, D., Chen, J., ... & Zheng, X. (2024). Data Interpreter: An LLM Agent For Data Science. arXiv preprint arXiv:2402.18679.
8. Horton, J. J. (2023). Large language models as simulated economic agents: What can we learn from homo silicus? (No. w31122). National Bureau of Economic Research.
9. Kinoshita, R., & Shiramatsu, S. (2022, November). Agent for Recommending Information Relevant to Web-based Discussion by Generating Query Terms using GPT-3. In 2022 IEEE International Conference on Agents (ICA) (pp. 24-29). IEEE.
10. Li, Y., Wen, H., Wang, W., Li, X., Yuan, Y., Liu, G., ... & Liu, Y. (2024). Personal llm agents: Insights and survey about the capability, efficiency and security. arXiv preprint arXiv:2401.05459.
11. Liu, Z., Hu, H., Zhang, S., Guo, H., Ke, S., Liu, B., & Wang, Z. (2023). Reason for future, act for now: A principled framework for autonomous llm agents with provable sample efficiency. arXiv preprint arXiv:2309.17382.
12. Nascimento, N., Alencar, P., & Cowan, D. (2023). Gpt-in-the-loop: Adaptive decision-making for multiagent systems. arXiv preprint arXiv:2308.10435.
13. Suri, G., Slater, L. R., Ziaee, A., & Nguyen, M. (2024). Do large language models show decision heuristics similar to humans? A case study using GPT-3.5. Journal of Experimental Psychology: General.
14. Tsai, Y. D., Tsai, Y. C., Huang, B. W., Yang, C. P., & Lin, S. D. (2023). Automl-gpt: Large language model for automl. arXiv preprint arXiv:2309.01125.
15. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., ... & Wang, C. (2023). Autogen: Enabling next-gen llm applications via multi-agent conversation framework. arXiv preprint arXiv:2308.08155.
16. Wu, Y., Min, S. Y., Prabhumoye, S., Bisk, Y., Salakhutdinov, R., Azaria, A., ... & Li, Y. (2023). SPRING: GPT-4 Out-performs RL Algorithms by Studying Papers and Reasoning. arXiv preprint arXiv:2305.15486.
17. Wu, Z., Zheng, S., Liu, Q., Han, X., Kwon, B. I., Onizuka, M., ... & Xiao, C. (2024). Shall We Talk: Exploring Spontaneous Collaborations of Competing LLM Agents. arXiv preprint arXiv:2402.12327.
18. Yang, H., Yue, S., & He, Y. (2023). Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions. arXiv preprint arXiv:2306.02224.

19. Yuan, S., Song, K., Chen, J., Tan, X., Shen, Y., Kan, R., ... & Yang, D. (2024). Easytool: Enhancing llm-based agents with concise tool instruction. arXiv preprint arXiv:2401.06201.
20. Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., ... & Neubig, G. (2023). Webarena: A realistic web environment for building autonomous agents. arXiv preprint arXiv:2307.13854.
21. Zhou, W., Jiang, Y. E., Li, L., Wu, J., Wang, T., Qiu, S., ... & Sachan, M. (2023). Agents: An open-source framework for autonomous language agents. arXiv preprint arXiv:2309.07870.

Appendix.

Code:

```
import os
from typing import Optional, List, Dict
from pydantic import BaseModel
import datetime
import json
import uuid

config_list = autogen.config_list_from_json(env_or_file="OAI_CONFIG_LIST")
planner = autogen.AssistantAgent(
    name="planner",
    llm_config={"config_list": config_list},
    system_message="You are a helpful AI assistant. You suggest coding and reasoning steps for another AI assistant to accomplish a task. Do not suggest concrete code. For any action beyond writing code or reasoning, convert it to a step that can be implemented by writing code. For example, browsing the web can be implemented by writing code that reads and prints the content of a web page. Finally, inspect the execution result. If the plan is not good, suggest a better plan. If the execution is wrong, analyze the error and suggest a fix.",
    )
planner_user = autogen.UserProxyAgent(
    name="planner_user",
    max_consecutive_auto_reply=0,
    human_input_mode="TERMINATE",
    code_execution_config={"use_docker": False},
    )

def is_termination_msg(content) -> bool:
    have_content = content.get("content", None) is not None
    if have_content and "TERMINATE" in content["content"]:
        return True
    return False

assistant = autogen.AssistantAgent(
    name="assistant",
    is_termination_msg=is_termination_msg,
    llm_config={"temperature": 0, "timeout": 600,  "cache_seed": 42,  "config_list": config_list},
    )
user_proxy = autogen.UserProxyAgent(
    name="user_proxy",
```

```
    human_input_mode="TERMINATE",
    max_consecutive_auto_reply=10,
    code_execution_config={"work_dir": "planning","use_docker": False,},
    )

import os
@user_proxy.register_for_execution()
@assistant.register_for_llm(
    name="ask_planner",
    description="ask planner to: 1. get a plan for finishing a task, 2. verify the execution result of the plan and potentially suggest new plan. Takes in message (str): question to ask planner. Make sure the question include enough context, such as the code and the execution result. The planner does not know the conversation between you and the user, unless you share the conversation with the planner."
    )
def ask_planner(message:str)->str:
    planner_user.initiate_chat(planner, message=message)
    return planner_user.last_message()["content"]

@user_proxy.register_for_execution()
@assistant.register_for_llm(name="add_event", description="Add an event to the calendar, returns the id of the event added")
def add_event(title: str, description:str, start_time: datetime.datetime, end_time: datetime.datetime)-> str:
    with open('events.json', 'w') as f:
        json.dump({'cal':str(uuid.uuid4()),'title': title,'start_time': start_time,'end_time': end_time,'details': description,}, f)
    return str(id)

@user_proxy.register_for_execution()
@assistant.register_for_llm(name="get_events", description="Takes in a start and end datetime and returns json of event between those times")
def get_events(starting: datetime.datetime, ending:datetime.datetime)-> Dict:
    with open('events.json', 'r') as f:
        calendar = json.load(f)
    sorted_events = sorted(calendar.items(), key=lambda x: x[1]['start_time'], reverse=False)
    events_between = []
    for id, event in sorted_events:
        if datetime.datetime.strptime(starting, '%Y-%m-%d %H:%M:%S') <= datetime.datetime.strptime(event['start_time'], "%Y-%m-%d %H:%M:%S") <= datetime.datetime.strptime(ending, '%Y-%m-%d %H:%M:%S'):
            events_between.append(event)
```

```
    return json.dumps(events_between)

@user_proxy.register_for_execution()
@assistant.register_for_llm(name="get_emails", description="Returns json of emails in inbox")
def get_emails()-> Dict:
    with open('inbox.json', 'r') as f:
        inbox = json.load(f)
    return inbox

@user_proxy.register_for_execution()
@assistant.register_for_llm(name="draft_email", description="Returns json of a saved email draft")
def draft_email(subject:str, body:str, recipients:str)-> Dict:
    id = str(uuid.uuid4())
    with open('outbox.json', 'w') as f:
        json.dump({'id':id,'subject': subject,'body': body,'recipients': recipients,}, f)
    return {'id':id,'subject': subject,'body': body,'recipients': recipients,}

@user_proxy.register_for_execution()
@assistant.register_for_llm(name="add_todo", description="Takes in a task title, description, and due date;
and returns the id of the task added")
def draft_email(title:str, description:str, due_date:datetime.datetime, completed:bool)-> str:
    id = str(uuid.uuid4())
    with open('taskList.json', 'w') as f:
        json.dump({'id':id,'title': title,'description': description,'due_date': due_date,'completed':completed}, f)
    return str(id)

@user_proxy.register_for_execution()
@assistant.register_for_llm(name="get_todos", description="Returns a dictionary of task list")
def get_todos()-> Dict:
    with open('taskList.json', 'r') as f:
        taskList = json.load(f)
    return taskList

@user_proxy.register_for_execution()
@assistant.register_for_llm(name="get_files", description="Returns a list of file names in my documents
folder")
def get_files()-> List:
    file_list = os.listdir('/localFilePath/')
    return file_list
```

```
@user_proxy.register_for_execution()
@assistant.register_for_llm(name="read_in_file", description="Takes in file name (str) and returns str content of the file")
def read_file(filename:str)-> str:
    with open(os.path.join('/localFilePath/', filename), 'r') as file:
        content = file.read()
    return content

user_proxy.initiate_chat(assistant, message="can you check my recent emails and draft responses accordingly and update calendar or todos accordingly")
```