Design based-research for streamlining the integration of text-generative AI into socially-assistive robots

Anna Lekova[1], Detelina Vitanova[2]
[1]Institute of Robotics
Bulgarian Academy of Sciences, Acad. Georgi Bonchev str., 1113 Sofia
a.lekova@ir.bas.bg
[2] Computer Science Department,
ULSIT, 119 Tsarigradsko Shose blvd., Sofia, Bulgaria
d.vitanova@unibit.bg

ABSTRACT: ***Integrating text-generative AI through generative pre-trained transformers (GPTs) into socially-assistive robots (SARs) could significantly enhance their ability to perform natural language processing (NLP) tasks. Well-known implementations of GPTs are OpenAI ChatGPT, Google Gemini, MS Azure AI services, BgGPT. A universal approach for streamlining this integration would allow people without technical expertise to enhance conversations with their robots. This is particularly relevant, given that INSAIT has developed BgGPT, the first free and open Bulgarian-specific language model, designed for Bulgarian users, institutions and businesses. Improving the efficiency of voice-based and text-based queries to robots is essential for enhancing front-end services, as it facilitates more natural interactions with users. On the back-end, text generation plays a key role in interpreting and responding to these queries. Therefore, the study explores design-based research focused on streamlining the integration of BgGPT endpoints into various SARs, with a specific focus on evaluating response times. The main concept involves developing an Express-based web server as the backend infrastructure that facilitates access to GPTs and SARs local modules using standard TCP and HTTP protocols. In the front end, the server's GET and POST endpoints are accessed using Blockly, simplifying application design by offering a visual programming environment that allows users to customize conversation flows without any programming skills. The conclusions regarding the rationale are drawn from the implementation of the proposed integration for three different GPT models and two SARs—NAO and Furhat.***

Keywords: Socially-assistive robots, Conversational Artificial Intelligence, chat GPT models, visual programing, APIs.

I. INTRODUCTION

As Artificial Intelligence (AI) advances, AI-powered physical robots have become new tools for improving human well-being in everyday life. Recently, the integration of text-generative AI through Generative Pre-trained Transformers (GPTs) into Socially-Assistive Robots (SARs) has introduced new possibilities for enhancing human-robot interactions. However, this integration requires programming skills and technical knowledge, particularly in areas like text-to-speech services, speech recognition, text generation, user-robot interfaces and the robot's sensor and actuator subsystems. In this context, there is currently no user-friendly approach how to integrate cloud GPT models, such as OpenAI ChatGPT, Google Gemini and MS Azure AI services, into SARs. On the back-end, text generation plays a key role in interpreting and responding to queries. On the other hand, streamlining voice-based and text-based queries to robots is important for enhancing front-end services, allowing for more natural interface user-robot.

Human-like interactions with robots using Conversational Artificial Intelligence (ConvAI) enable natural communication in various contexts, enhanced by the robot's physical presence and hardware sensors. ConvAI combines Natural Language Processing (NLP) with machine or deep learning and although ConvAI can be virtual, the diverse sensory systems and motion control of the robots can provide context for the surrounding environment. ConvAI integrated into robots should actively generate responses by analyzing user utterances and conversation context, enhanced by GPT model capabilities. Studies on human-robot interaction increasingly focus on integrating cloud-based services for speech recognition systems (SRS) and text generation [1-7]. Most of these efforts aim to extend conversational dialogue and convert voice commands into machine-readable code, creating a more natural and intuitive interface for communication by integrating chat bots to enhance responsiveness. However, few address the technological limitations. Authors in [4] conducted a study on the accuracy and delay of cloud-based speech recognition systems (SRS) in human-robot interaction. Authors' findings suggest that the precision and latency of cloud-based SRS are significantly influenced by the network connection's quality and the computational capabilities of the cloud server and can vary from a few hundred milliseconds to several seconds. These results highlight that, while cloud-based technologies offer great promise for improving human-robot interactions, they also present certain technical challenges, particularly with latency and real-time processing. In summary, technical expertise is required to improve AI-driven conversations with robots, emphasizing the need for a more universal approach to streamline the integration of GPTs in SARs. Streamlining voice and text queries, along with text generation on the back end, is essential for enhancing user interactions and overall system efficiency.

ACKNOWLEDGE THE FINANCIAL SUPPORT OF THE PROJECT WITH ADMINISTRATIVE CONTRACT № КР-06-Н57/8 FROM 16.11.2021. - FUNDED BY THE "COMPETITION FOR FUNDING BASIC RESEARCH - 2021." FROM THE RESEARCH SCIENCES FUND, BULGARIA.

The study explores Design-Based Research (DBR) focused on streamlining the integration of chat GPT endpoints into various SARs. The previous iterations in this DBR are summarized in [8]. The novelty in the ongoing iteration in the building-testing cycles of a software architecture for convAI in two SARs, lies in the integration of Bulgarian cloud services for NLP into the robots' native software. INSAIT has developed BgGPT, the first free and open Bulgarian-specific language model, designed for Bulgarian users, institutions and businesses. Additionally, the architecture has been optimized to address technical challenges, identified in previous iterations, especially regarding response time issues with cloud services for NLU on the Node-RED platform [9]. This study successfully addressed these challenges through solutions implemented using a web server developed with Express.js [10] and the integration of Blockly [11], thus enhancing response times and streamlining programming process.

The contribution of the proposed study is a design-based research how to streamline the integration of text- generative AI, such as OpenAI ChatGPT, NLPcloud GPT and BgGPT endpoints into two SARs - NAO and Furhat, and to evaluate the time of GPT APIs response. The main concept involves creating an Express-based server that provides seamless access to different SARs through standard TCP and HTTP protocols. In the front end, the server's GET and POST endpoints are accessed using Blockly, simplifying application design by offering a visual programming environment that allows users to customize conversation flows without any programming skills. The conclusions regarding the rationale are drawn from the implementation of the proposed integration, which involved an analysis of its effectiveness in real-world scenarios.

II. SOME SPECIFICS OF THE INTEGRATION OF SARS WITH CHAT GPT MODELS

Design-based research in the integration of GPT models into SARs focuses on iterative development and testing within real-world SARs settings. The modular architecture proposed in Fig. 1 offers middleware solutions to simplify the integration of various NLP services, such as ASR, TTS, text generation, into the native software of robots that may have different capabilities and varying levels of embedded AI components. Narration can occur through voice, QR code, or text. Different SARs manage voice interactions and QR code scanning, while text chatting is facilitated through Blockly blocks that interface with the Express server frontend (Fig.2).

A. NLP capabilities of SARs

While the Furhat robot [12], known as one of the most advanced conversational robots, possesses many AI capabilities, the humanoid NAO robot [13] offers engaging animations but has limited speech recognition and dialogue options based on a predefined lexicon, resulting in a restricted vocabulary and a limited number of dialog scenarios. Integrating ConvAI into NAO can significantly enhance its capabilities, particularly for intensive speech and listening exercises for individuals with language difficulties.

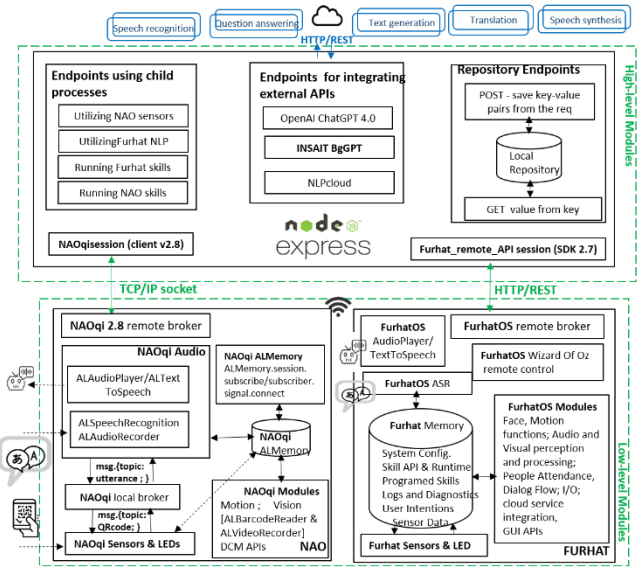

Fig 1. Modular architecture for integration of various chat GPT models in SARs

B. Designing API endpoints for chat GPT models

A universal approach would be to implement a RESTful API server, like an Express-based server, that bridges different ChatGPT models with SARs. This solution would enable developers to easily switch between or combine various NLP models, while providing uniform access for speech recognition, text-to-speech and text generation without requiring wide programming skills.

An Express-based server refers to a web server built using Express.js, a lightweight and flexible web application framework for Node.js. Express server simplifies the process of building web servers and APIs by providing a set of tools and features for handling HTTP requests, routing, middleware integration, etc. It offers minimal core functionality, designed to be extended, while still maintaining the flexibility of Node.js. The developed server for integrating ChatGPT models into robots, has various types of endpoints that use child processes either for handling the robots' remote API sessions, or external APIs for cloud-based natural language processing models like ChatGPT or BgGPT to enhance robot interactions. Figure 1 illustrate different types of endpoints with child process. The first box on the left shows endpoints for executing Python 2.7 child processes in Node.js using the built-in child_process module. This allows to run Python scripts from the Node.js. To use different Python interpreters, a virtual environment (venv) endpoint was developed that executes shell commands to start the venv activation scripts. This allows for the flexible management of project-specific dependencies and interpreter versions. Similar endpoints are presented for running, exec commands for: java -jar , SSH connection, PSCP, the PuTTY Secure Copy client for transferring files securely, etc.

In line with the principle of modularity, access to OpenAI ChatGPT and NLP cloud services was established as described in [8]. Similarly, the newly designed access to BgGPT was implemented as a child process in JavaScript. Using child processes provides some benefits in performance, maintainability, and modularity, since it isolates the API logic from the main application, resulting in Express server code focused on HTTP requests. Child

processes enable non-blocking execution, allowing the server to handle multiple requests simultaneously without delays. Additionally, they facilitate the use of Python scripts without rewriting them in Node.js, ensuring compatibility with existing systems.

Local endpoints were established for accessing the internal repository for data posting and retrieval, integrating services through the use of `const repository = {};` , which initializes an empty object that stores key-value pairs in json format.

C. *Integration of Blockly with Express server*

While Express executes backend commands using child processes, the frontend is interfaced through Blockly blocks. When the user interacts with the blocks for NAO in Blockly (Fig.2), this triggers an API request to the server, which then runs a Node.js script (utilizing a child process for the Python NAOqi session) to control the NAO robot. This setup enables streamlining the integration between the visual programming environment and the server's backend operations. Through the NAO blocks, users can perform actions such as reading QR codes, uploading audio files to the NAO's internal memory, playing MP3 files, creating animations, and more.

Fig. 2. Interface the Express server frontend using Blockly blocks

III. IMPLEMENTATION AND EVALUATION OF THE PROPOSED INTEGRATION

A. *Streamlining the integration of BgGPT endpoints into NAO and Furhat robots*

The proposed implementation of the integration of BgGPT endpoints into NAO and Furhat robots illustrates that this can be done without significant programming skills. Data processing, storage and online display were conducted on a laptop (11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz, 8.00 GB RAM, MS Windows 11 Prof. 64-bit).

The Express server operates locally, with requests handled through the endpoint accessible via the URL: `*http://localhost:3000/bgGPT?question=<text>&context=<text>*`. In this format, the `<text>` placeholders are replaced with the actual `question` and `context` values provided by the user in the query string. When a request is made to this endpoint, the server triggers the `exec` command, which spawns a child process. Since INSAIT didn't have a client for using BgGPT at the time of this study, we utilized the Axios library in a similar manner. The pseudocode how to access BgGPT APIs for the JS child process for the BgGPT endpoint is shown in Fig. 3, where the `question` and `context` parameters are passed as command-line arguments. The output and any errors from the script are managed within this process. It also executes and retrieves the output of child processes, cloud services, or shell commands in Node.js. By the NAO Event Listener: the data output from the Python script is logged in stdout.on('data'). Accordingly, the output of the command (stored in `stdout`) is then sent to the server using the fetch API and saved as a key-value pair in the internal repository accessible at `*http://localhost:3000/repository*`. The data is sent in JSON format with the following structure:

- key: 'answer', representing the name of the output value.
- value: `stdout`, which contains the command's output.

```
1. Import necessary libraries (axios for HTTP requests).

2. Parse command line arguments and initialize variables:
  - Extract `question` and `context` from command line arguments.
  - Store the API key.

3. Define the request payload:
  -Model: "INSAIT-Institute/bggpt-stage3-RFLC-Bigbalance-Duolingual-v2"
  - Prompt: "<s>[INST] {question} [/INST] {context}</s> [INST] {question} [/INST]"
  - Set max tokens (1024).
  - Set temperature to 0.1 (controls randomness).
  - Set top_k to 20 (limits how many words are considered).
  - Set repetition penalty to 1.1 (prevents repeated text).
  - Set stop condition to "</s>".
  - Stream is set to false.

4. Send a POST request to the BgGPT API:
  - Use axios to send the request to 'https://api.bggpt.ai/completions'.
  - Attach the request payload with headers
   (data, { 'apikey': apiKey,  'accept': 'application/json',
   'content-type': 'application/json'}).

5. If the request is successful:
  - Extract and log the response data:
   response =>
  - Specifically, log the generated text result:
   text = response.data.choices[0].text;

6. If the request fails:
  - Log the error message or error details.
```

Fig. 3 Pseudocode for the flow and logic how to access BgGPT APIs

B. *Evaluating the response times of BgGPT endpoints*

To set-up logging in an Express server to capture incoming requests, request parameters and response details, we utilized a logging middleware - Morgan for Node.js. We logged and monitored the request durations within Express, which records details of HTTP requests and aids in debugging and monitoring server activity. The `'dev'` format provides a concise, color-coded log showing the request method, URL, status code, response time, and response size. In the example console output, we can observe two logged HTTP requests:

```
Server is successfully running, ATlog is listening on port 3000
`POST /repository 200 7.881 ms - 48`
`GET /venv 404 14.271 ms - 143`
 `GET /QA?question= Котките и мишките (and so on) 882 tokens
  200 3898.670 ms - 882
```

The first line indicates that a POST request to `/repository` was successful with a status code of 200, responding in just 7.881 milliseconds (ms) and sending 48 bytes of data. The second line shows a GET request to `/venv` that returned a 404 status (resource not found) in 14.271 ms, with a response size of 143 bytes. These logs demonstrate that the Blockly to API server is fast, handling requests efficiently even in error cases like the 404 response. The third line displays a GET request to NLPCloudClient with the parameters `question='обичат ли се котка и мишка?'` and `context='обясни по детски'`. These values are sourced from `Blockly.JavaScript.quote_(block.getFieldValue('QUESTION') and 'CONTEXT')`. The status code of 200 indicates a successful response, along with the response time in milliseconds and the number of received tokens.

After establishing the connection between Blockly and the Express server, we found that the connection time is minimal and can be neglected. Unfortunately, the API responses are often quite slow. Three Wi-Fi network configurations were analyzed (all with Protocol: Wi-Fi 4 (802.11n); Network band 2.4 GHz and Link speed (Receive/Transmit): (1) less than 90/90 Mbps, (2)135/135 Mbps, (3) 5G 150/150 Mbps).

The delays coming from the cloud-based BgGPT API responses are illustrated in Figures 4-7. The graph in Figure 4 shows the relationship between the API response time (in seconds) and the number of tokens received. The plot illustrates that, as the number of completion tokens increases, the response time generally increases as well. The trend is almost linear,

We also tested whether some variations caused by factors such as network bandwidth, server load and latency within the cloud infrastructure, might cause different levels of delay. We analyzed whether a network congestion during peak usage times can result in slower response times, i.e. during cloud resources manage a higher volume of requests. We varied also the type of the request content: question explanation (tell in a childish way about the planet Saturn) and fairy tale generation (tell me a fairy tale about: a cat on a tree). From Figures 5 and 6, we can conclude that the relationship between received tokens and response time is not significantly affected by the type of query content, however it does slightly depend on network congestion. Low link speeds result in delays of about 1 to 2 seconds, although this is not always the case when fewer tokens are received.

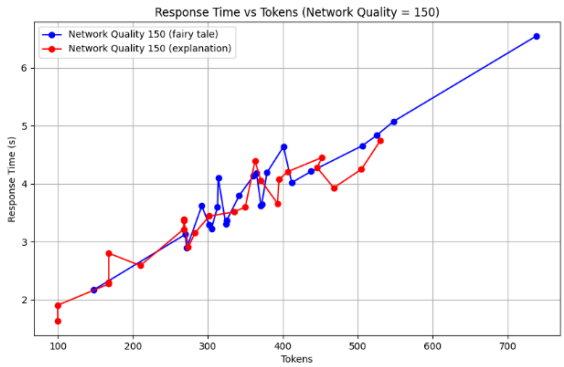

Fig. 4 Relationship between the API response time (in seconds) and the number of tokens received

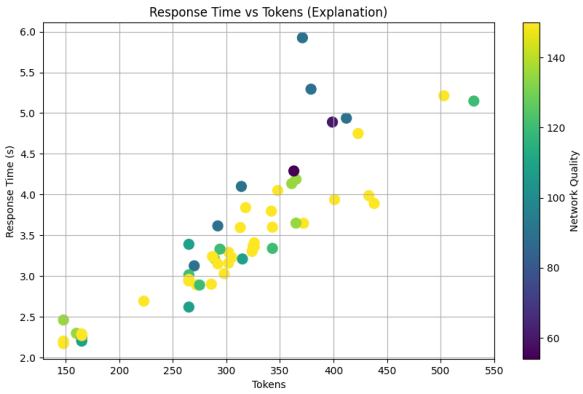

Fig. 5 Relationship between the API response time (in seconds) and the number of tokens received according to the network congestion for question explanation

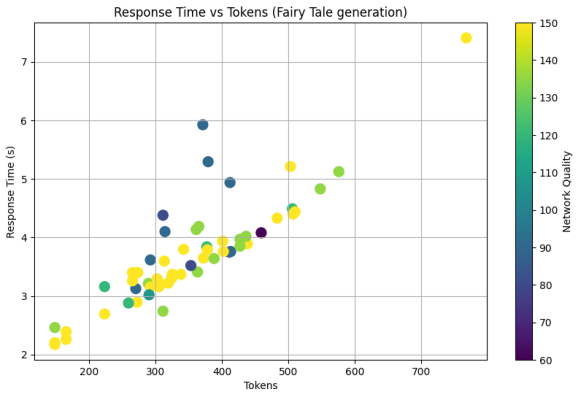

Fig. 6 Relationship between the API response time (in seconds) and the number of tokens received according to the network congestion for fairy tale generation

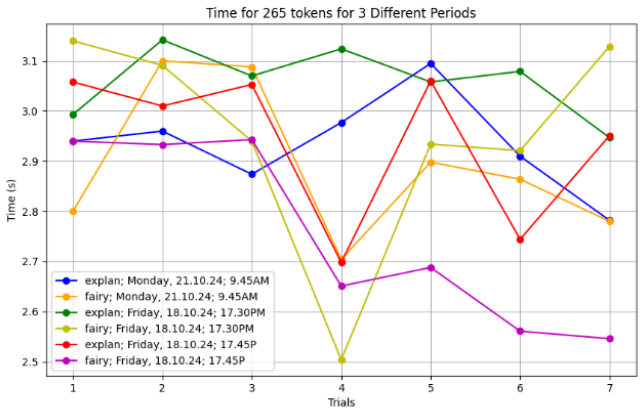

Fig. 7 Relationship between the API response time (in seconds) and the server congestion

C. *Discussion*

In general, our results reflect a difference in response time between the chat BgGPT APIs and web-based chat BgGPT. We explain this by several factors, with the main one being network latency. When using APIs, requests and responses must travel over the internet, which can introduce delays. In difference, web-based chat GPT is optimized to reduce this latency, particularly if it is hosted on servers with some optimizations for the user. Additionally, API requests often

incur overhead for authentication, data formatting, and other protocol-specific requirements, further increasing response times. The server load handling API requests can also vary, leading to slower response times during peak usage periods, although we didn't observe this (Fig. 7).

The results presented in Figures 4 to 6 align with the findings of Patil and Gudivada [14]. This comprehensive survey discusses the relationship between Large Language Models (LLMs) latency and output token count. A detailed analysis explaining the linear relationship between LLM latency and the number of output tokens, outlining a formula for total response time that includes a constant factor plus a term proportional to the output token count, can be seen in papers [15], [16].

IV. CONCLUSIONS

The study proposed design-based research aimed at optimizing the integration of chat GPT models, particularly BgGPT, into various SARs without programming skills. The main concept involves developing an Express-based web server as the backend infrastructure, which enables access to GPTs APIs and local modules of SARs through standard TCP and HTTP protocols. On the front end, users can access the server GET and POST endpoints via Blockly, providing a visual programming environment that simplifies application design and allows for customization of conversation flows. After evaluating the response times, it was determined that the delays are not attributable to network or cloud server congestion. Factors such as network speed, specific days of the week, times of day, and the types of request content were considered, however the latency primarily arise from using APIs and their slow responses due to the overhead for authentication, data formatting, and other protocol-specific requirements. Further research is necessary to enhance response times.

ACKNOLEDGEMENTS

The research findings were supported by the National Scientific Research Fund, Project № KP-06-H67/1. We express our gratitude to Emiliyan Pavlov from INSAIT, for assisting us in obtaining the API key and facilitating the use of the BgGPT API.

REFERENCES

[1]. T. Belpaeme, J. Kennedy, A. Ramachandran, B. Scassellati, and F. Tanaka, "Social robots for education: A review," Science Robotics, vol. 3, no. 21, p. eaat5954, Aug. 2018, DOI: https://doi.org/10.1126/scirobotics.aat5954.

[2]. O. Elfaki et al., "Revolutionizing social robotics: A cloud-based framework for enhancing the intelligence and autonomy of social robots," Robotics, vol. 12, no. 2, p. 48, Apr. 2023, DOI: https://doi.org/10.3390/robotics12020048.

[3]. F. Kaptein et al., "A cloud-based robot system for long-term interaction: Principles, implementation, lessons learned," ACM Transactions on Human-Robot Interaction, vol. 11, no. 1, pp. 1–27, Mar. 2022, DOI: https://doi.org/10.1145/3481585.

[4]. Deuerlein, M. Langer, J. Seßner, P. Heß, and J. Franke, "Human-robot-interaction using cloud-based speech recognition systems," Procedia CIRP, vol. 97, pp. 130–135, 2021, DOI: https://doi.org/10.1016/j.procir.2020.05.214.

[5]. L. Grassi, C.T. Recchiuto, and A. Sgorbissa, "Sustainable cloud services for verbal interaction with embodied agents," Intelligent Service Robotics, vol. 16, pp. 599–618, 2023, DOI: https://doi.org/10.1007/s11370-023-00485-3.

[6]. S. Kaszuba, J. Caposiena, S.R. Sabella, F. Leotta, and D. Nardi, "Empowering collaboration: A pipeline for human-robot spoken interaction in collaborative scenarios," in: A.A. Ali et al., Social Robotics. ICSR 2023, Lecture Notes in Computer Science, vol. 14454, Springer, Singapore, 2024, DOI: https://doi.org/10.1007/978-981-99-8718-4_9.

[7]. Y. Lai et al., "Intuitive multi-modal human-robot interaction via posture and voice," in: J. Filipe, J. Röning (eds), Robotics, Computer Vision and Intelligent Systems. ROBOVIS 2024, Communications in Computer and Information Science, vol. 2077, Springer, Cham, 2024, DOI: https://doi.org/10.1007/978-3-031-59057-3_28.

[8]. Lekova, P. Tsvetkova, A. Andreeva, M. Simonska, and A. Kremenska, "System software architecture for advancing human-robot interaction by cloud services and multi-robot cooperation," International Journal on Information Technologies and Security, vol. 16, no. 1, pp. 65–76, 2024, DOI: https://doi.org/10.59035/fmfz4017.

[9]. ExpressJS. Online: Retrieved October, 2024 from https://expressjs.com/

[10]. Google Developers, Blockly. Online: Retrieved October, 2024 from https://developers.google.com/blockly

[11]. Furhat Robotics. Online: Retrieved October, 2024 from https://furhatrobotics.com/

[12]. Aldebaran Robotics, NAO. Online: Retrieved October, 2024 from https://www.aldebaran.com/en/nao

[13]. R. Patil and V. Gudivada, "A review of current trends, techniques, and challenges in large language models (LLMs)," Applied Sciences, vol. 14, no. 5, p. 2074, 2024, DOI: https://doi.org/10.3390/app14052074.

[14]. C. Han, M. Xu, and H. Wang, "Adapting large language models for embodied agents," arXiv preprint arXiv:2407.05347, 2024, DOI: https://doi.org/10.48550/arXiv.2407.05347.

[15]. Sharma and K. Patil, "Embodied agent interactions: Recent developments," arXiv preprint arXiv:2410.10819, 2024, DOI: https://doi.org/10.48550/arXiv.2410.10819.