Interactive Debugging and Steering of Multi-Agent AI Systems

Will Epperson
willepp@cmu.edu
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA, USA

Gagan Bansal
gaganbansal@microsoft.com
Microsoft Research
Redmond, WA, USA

Victor Dibia
victordibia@microsoft.com
Microsoft Research
Redmond, WA, USA

Adam Fourney
adam.fourney@microsoft.com
Microsoft Research
Redmond, WA, USA

Jack Gerrits
jagerrit@microsoft.com
Microsoft Research
Redmond, WA, USA

Erkang Zhu
erkang.zhu@microsoft.com
Microsoft Research
Redmond, WA, USA

Saleema Amershi
samershi@microsoft.com
Microsoft Research
Redmond, WA, USA

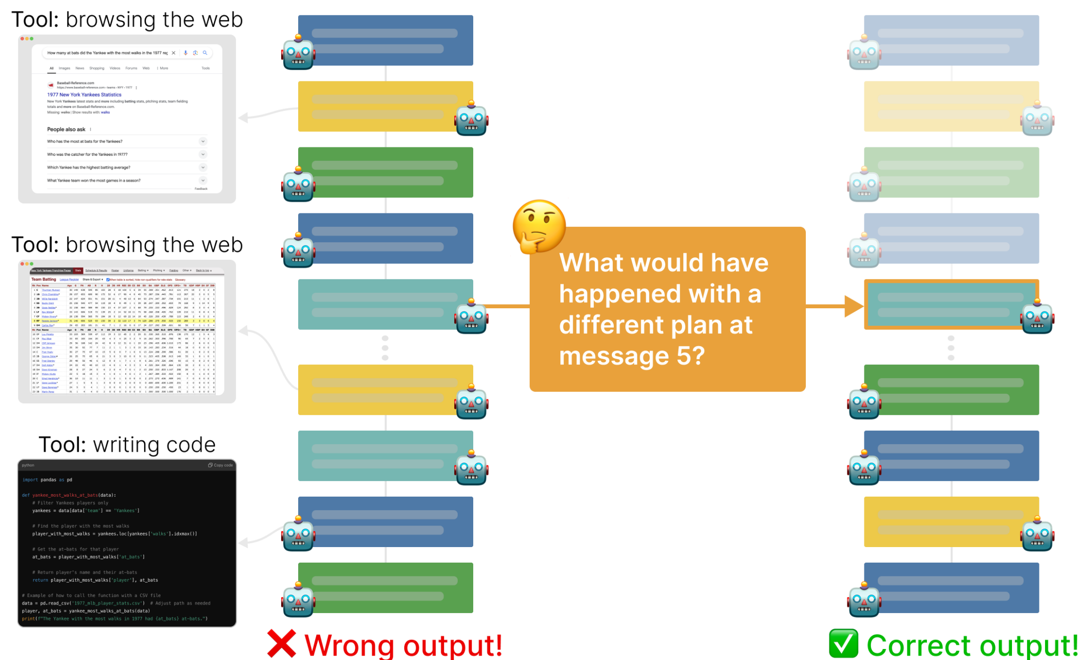

Figure 1: Debugging multi-agent AI systems involves reasoning over long multi-turn conversations where specialized agents use tools like web browsing and writing code with LLMs. AGDebugger allows users to interactively debug and steer multi-agent teams by resetting the agents to earlier points in the workflow then editing messages to interactively test hypotheses about their behavior.

This work is licensed under a Creative Commons Attribution 4.0 International License.
CHI '25, Yokohama, Japan
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3713581

Abstract

Fully autonomous teams of LLM-powered AI agents are emerging that collaborate to perform complex tasks for users. *What challenges do developers face when trying to build and debug these AI agent teams?* In formative interviews with five AI agent developers, we identify core challenges: difficulty reviewing long agent conversations to localize errors, lack of support in current tools

for *interactive* debugging, and the need for tool support to iterate on agent configuration. Based on these needs, we developed an interactive multi-agent debugging tool, AGDebugger, with a UI for browsing and sending messages, the ability to edit and reset prior agent messages, and an overview visualization for navigating complex message histories. In a two-part user study with 14 participants, we identify common user strategies for steering agents and highlight the importance of interactive message resets for debugging. Our studies deepen understanding of interfaces for debugging increasingly important agentic workflows.

CCS Concepts

• **Human-centered computing → Interactive systems and tools**; • **Computing methodologies → Multi-agent systems**.

Keywords

AI agents, ai debugging, interactive debugging systems, language models

ACM Reference Format:
Will Epperson, Gagan Bansal, Victor Dibia, Adam Fourney, Jack Gerrits, Erkang Zhu, and Saleema Amershi. 2025. Interactive Debugging and Steering of Multi-Agent AI Systems. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26-May 1, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3706598.3713581

1 Introduction

Multi-agent AI systems are emerging as a powerful paradigm for addressing tasks beyond the scope of single large language models (LLMs) [9, 22, 43, 47]. By combining LLMs with multi-turn state tracking, external tool use, and collaborative interactions, multi-agent systems can perform complex, real-world tasks such as accessing up-to-date information or executing actions in dynamic environments [15]. This capability has allowed multi-agent AI systems to excel in challenging benchmarks that require reasoning about complex tasks and using tools to interact with the world like web browsing [15], reasoning over complex files [27], and writing and executing code [17, 41].

Despite these advances, existing AI development tools fall short when it comes to understanding and debugging the complex, multi-turn behaviors of agent teams. Traditional AI debugging practices, which focus on model training or correcting datasets, are inadequate in this new paradigm. Much of the work in debugging LLM systems centers around crafting effective text *prompts* that instruct the LLM how to accomplish a task through in-context learning [5, 48]. Prior research has developed tools for crafting effective prompts for tasks involving individual LLM invocations [16, 29, 32, 39] or chains of LLM calls [3, 44], but has not addressed the challenges associated with debugging teams of fully-autonomous AI agents.

Debugging multi-agent teams introduces new debugging challenges since agent teams require first crafting individual prompts and tools for each agent, then understanding how the team works together to accomplish a task by making numerous LLM calls over a multi-turn conversation. Agent conversations are complex and dynamic, where agents formulate a plan on the fly for a task and then execute the plan by using tools to interact with the world as needed [9, 43]. Debugging multi-agent systems requires simultaneously understanding both the individual behaviors of agents and the emergent interactions between them. This multifaceted debugging challenge demands new tools that can integrate insight across the entire agent team to avoid "cascading errors" that fix one component while breaking another [28, 36, 44].

While recent systems like AutoGen Studio [10] and OpenDevin [41] enable developers to interact with multi-agent AI teams, they primarily focus on task execution and agent construction. These platforms lack robust debugging features, particularly for multi-turn interactions. As agent conversations grow longer, existing tools do not provide the ability to pause, rewind, or edit agent behaviors in real time. Additionally, they lack comprehensive visualizations to help developers track and understand the evolving dynamics between agents. These gaps in literature makes debugging multi-agent systems particularly challenging, as developers need to iteratively diagnose and adjust interactions across the team.

To address the limitations of current tools and better support the debugging of multi-agent AI systems, we explore two key research questions: (1) How can we design systems that enable developers to effectively debug multi-agent AI teams? (2) How do developers use such a system to debug and improve agent workflows in practice?

To investigate these research questions, we first conducted formative interviews with five expert developers who have extensive experience building multi-agent AI systems. These interviews revealed several key challenges in the debugging process, such as difficulties in understanding long, multi-turn agent conversations, the lack of interactive debugging support in existing tools, and the need for better tooling to iterate on agent configurations.

Using these findings, we designed an interactive debugging system, AGDebugger to address these challenges. AGDebugger has three primary features to facilitate the debugging process. The first builds off prior interfaces for agent configuration [10, 21, 41] and allows users to interactively send messages to the agents and inspect the history of messages sent with fine-grained control on the execution of messages. An extension to chatting with agents is the ability to interactively control a conversation by resetting to previous points and editing previously sent agent messages. AGDebugger enables such an interaction by check-pointing agent state (including state that might be impacted by agent actions and tool use) before each message is sent to enable resetting to earlier points in a conversation and editing agent messages. Finally, as agent conversations grow longer and users edit the conversation history they can become difficult to track. AGDebugger also includes an interactive overview visualization for summarizing the agent conversations and edits.

We conducted a two-part user study with 14 participants to evaluate the effectiveness of AGDebugger for debugging multi-agent workflows. In the first part, participants diagnosed errors in agent workflows by using AGDebugger to inspect and understand where the agents failed. In the second part, they used AGDebugger to edit agent messages and steer the agents toward successful outcomes. Our findings reveal that participants frequently made three types of modifications: specifying more detailed instructions, simplifying agent tasks, and altering the agents' plans.

These patterns reflect common failure modes in multi-agent systems, and AGDebugger effectively supported participants in iteratively steering the agents towards correct behavior. Finally, we conclude by presenting remaining open challenges in agent debugging found from our study. Such challenges include decoupling steering from the agent implementation and determining if edits had an effect. AGDebugger is available as an open source tool at https://github.com/microsoft/agdebugger. In summary, this paper makes the following contributions:

1. Formative interviews with agent developers that reveal common challenges developers encounters while developing multi-agent AI systems including understanding long, multi-turn agent conversations, the lack of interactive debugging support, and the need for better tooling to iterate on agent configurations.
2. A prototype agent debugging system, AGDebugger, with three key features for facilitating agent debugging: interactively sending and stepping through agent messages, the ability to interactively edit and steer agent teams, and an overview visualization to summarize agent conversations and edits.
3. Results from a user study where participants use AGDebugger to diagnose and then experiment with fixes for errors in agent workflows. We identify common patterns of steering across users around adding instructions to agent messages, simplifying messages, and modifying the agents' plan.

2 Related Work

2.1 Multi-Agent AI Systems

In recent years, general purpose Large Language Models (LLMs) leveraging in-context learning (e.g., few-shot prompting), have been applied to various domains that previously required training specialized models [5]. When combined with the ability to use tools and store state, these models can function as agents that interact with the world to accomplish more complex tasks. AI agents have been developed for tasks ranging from browsing the web [15, 49] to advanced coding tasks that involve looking up documentation and combining code authoring with execution [17, 41].

For more intricate problems, performance improvements can often be achieved by *multi-agent systems*, where multiple specialized agents work together to plan and perform tasks [14, 22, 43, 47]. These agents use LLMs for decision making and planning by breaking down tasks into smaller sub-components, leverage tools to interact with the world, and memory to keep track of past actions and provide context for subsequent actions [14]. Multi-agent systems typically combine multiple *specialized* agents, each with different roles, prompts, tools, and memory that collaborate together to solve tasks by planning and delegating to the appropriate agent. Such systems demonstrate strong performance on complex tasks and make it easier for developers to design and develop reusable agents for different tasks, similar to object oriented programming [8, 14]. Creating multi-agent systems involves configuring each individual agent's capabilities and tools as well as how agents communicate with one another. Frameworks such as AutoGen [43], CAMEL [22], OS-Copilot [47], Crew AI [9], and LangGraph [21] facilitate the development of such systems, enabling the creation of LLM-powered agents with distinct tools and roles.

Despite these advances, LLMs exhibit various failure modes, such as losing track of critical information over long contexts [25] and hallucinating facts that lack contextual grounding [26]. These underlying model issues remain in multi-agent settings and can be exacerbated when agents interact through chained model calls, further complicating error tracking and resolution [44, 46].

2.2 LLM and Agent Debugging

Debugging a single LLM component or team of agents builds on prior research on debugging traditional machine learning and AI models. Researchers have previously recognized the difficulty of diagnosing model failure modes [1, 2]. In classic ML workflows, interactive debugging tools have been developed to assist developers at various stages of model development, from tracking model changes [2] to identifying poorly performing data subsets during evaluation [6]. One key insight from these tools is the value of providing developers with direct access to the raw data driving model decisions, enabling them to better understand and intervene when failures occur [6, 31].

Prior research also discusses the value of *counterfactual* explanations for understanding ML models [40]. Such methods allow users to explore "what if" scenarios by modifying model inputs and observing how predictions shift [20, 23, 42]. Interactive systems support what if analysis of NLP models, aiding users in generating variations of input sentences to test how model outputs change [7, 35, 45].

For LLMs, debugging primarily involves crafting effective prompts to specify a task, rather than selecting model training data or architectures. Prompt engineering has become central to eliciting desired LLM behaviors, but studies show that users often struggle to express their design goals in prompts, and face challenges debugging incorrect model outputs [16, 48]. Interactive tools allow users to experiment with different system prompts and models [29, 38]. Tools like PromptMaker [16] and Sequences Salience [39] aim to assist users in refining their prompts, while other systems allow for iterative feedback-driven adjustments [32]. Other systems help users compare the outputs from multiple prompts and use a "LLM as a judge" to grade which outputs are better [18, 19, 37] or unit test LLM components with code or model-based assertions [4, 33]. Recent research like ChainForge [3] or PromptChainer [44] have contributed tools for creating and debugging pipelines of LLM calls, however do not support LLM based agents.

Tools focused on debugging a single LLM component or predefined pipeline of LLM calls do not fully address the challenges associated with debugging multi-agent systems. Muti-agent debugging requires understanding both individual agent behavior, tool use, and memory as well as agent interactions over long multi-turn conversations including appropriate delegation or task termination. Improvements in one component can introduce errors in others, underscoring the need for testing the entire agent team simultaneously on a task—a problem previously identified for pipelines of traditional AI systems or chains of LLM calls [2, 28, 36, 44].

Recent systems like AutoGen Studio [10], OpenDevin [41], or Crew AI [9] provide interfaces for creating and interacting with multi-agent AI teams for a task, allowing developers to communicate with agents via a chat-style UI. While these systems support agent construction, they focus less on the interactions required for interactive debugging. There remains limited HCI research on the design of tools for debugging multi-agent systems and how developers use such tools.

Prior research has demonstrated the power of pause and reset mechanisms for debugging, such as the crash-and-rerun programming model of TurKit that allows programmers to re-run programs that make expensive function calls to crowdworkers [24]. Interactive tools for LLM debugging like the OpenAI Playground provide the ability to edit earlier LLM messages, but only support interaction with a single LLM in a chat rather than a multi-agent team [29]. LangGraph offers a UI for creating multi-agent systems based on a graph communication model and offers support for breakpoints to inspect agent state when called in the graph [21]. However their implementation is not public, requires developers to pre-specify breakpoints, and only works for graph-based agents. The pause and reset interactions presented in AGDebugger are applicable to any multi-agent system that interacts by passing messages.

3 Background: Agent Framework and Tasks

This section describes three key background concepts: the implementation of the agent framework we used, reviews the GAIA benchmark dataset for evaluating AI agents [27], and details an example of a specific multi-agent team (implemented using this framework) that achieves high performance on GAIA.

3.1 Agent Implementation Framework

Our debugging system is built on the open-source AutoGen framework [43]. In this framework, agents are implemented as Python classes that communicate by sending *messages* through a shared *runtime*. These messages exchanged are also typed Python objects containing data. Agents implement *message handlers* that respond to particular types of messages. When an agent receives a message, the framework triggers the appropriate handler, which might make LLM calls, use tools, and send a new message in a response. Furthermore, while processing a new message, agents often update their state such as navigating to a new page in a web browser the agent controls. All messages are sent through the central runtime which manages a *message queue*. When messages are processed, they are moved off the queue and sent to the appropriate agents. This design allows for flexible patterns of communication between agents, where agents can communicate directly with each other or send messages to all other agents.

In addition to messages, agents can have internal "thoughts" that are simply log messages. These thoughts are not sent to other agents but can be helpful for debugging. Each agent can keep track of its message history to provide context for future model calls.

3.2 GAIA Benchmark Tasks

An agent framework by itself is only half the story: it needs to be applied to tasks or work to be of value. In this paper, we apply the framework to problems from the GAIA agent benchmark, allowing us to measure the proficiency of our agents and teams. The GAIA benchmark is a collection of challenging AI assistant tasks that require diverse skills such as coding, using the internet, and parsing files [27]. It serves as a standardized way to evaluate the capabilities of AI agents across a range of complex tasks. GAIA tasks are divided into three levels of difficulty, with level 1 the easiest and level 3 the hardest. As of writing, these tasks remain very challenging for AI assistants, with the top-performing team on the GAIA leaderboard scoring an average of around 35% on the test set [11].

In our studies and examples, we focus on two specific tasks from the validation set of GAIA Level-1, shown in Table 1. These tasks are complex as they involve searching for and synthesizing information from multiple websites to generate the final answer. While our agent team performs near the state-of-the-art on the GAIA leaderboard, it consistently fails to complete these specific tasks correctly, making them ideal candidates for debugging. On both tasks, our agent team outputs an answer, but the answers are incorrect. We investigate these two tasks specifically in our user study since both tasks are similar (e.g. web-focused and same difficulty level) with prompts that are easy for participants to understand without extra background.

ID	Benchmark question	Answer
T1	How many at bats did the Yankee with the most walks in the 1977 regular season have that same season?	519
T2	Of the cities within the United States where U.S. presidents were born, which two are the farthest apart from the westernmost to the easternmost going east, giving the city names only? Give them to me in alphabetical order, in a comma-separated list	Braintree, Honolulu

Table 1: Example agent tasks for debugging from the GAIA Level-1 validation set [27].

3.3 Agent Team for GAIA Tasks

To address the GAIA benchmark tasks, we use the Magentic-One generalist AI agent team [12]. This team, implemented using the framework described above, consists of five agents that collaborate to solve tasks:

(1) An *Orchestrator* who plans and controls the conversation
(2) A *Coder* who authors Python code to solve subproblems
(3) An *Executor* who executes code locally and returns results
(4) A *File Surfer* who can parse and interact with local files of various formats (e.g., PDF, PowerPoint, etc.)
(5) A *Web Surfer* who can access and interact with web pages in a browser, and can perform search queries, in a manner comparable to prior web agents [15]

Each of these agents maintains their own state, has access to different tools to fulfill user requests, and can make their own calls to LLMs. To complete a task, the agents are given an input prompt, then collaboratively develop and execute a plan to solve the task and produce a final result. These agent conversations can become quite

lengthy. For example, in the runs we analyze, it took the agents 71 messages to produce an answer for task T1, and 90 messages for task T2. The raw log files with the messages contain 6,368 words for T1 and 7,230 words for T2. This complexity in agent interactions highlights the need for effective debugging tools, which is the focus of our research.

4 Formative Interviews on Agent Debugging

To better understand current developer pain-points around developing multi-agent AI systems, we interviewed five developers at Microsoft with experience building multi-agent applications. We recruited these participants within a large technology corporation. All five had prior experience using the AutoGen multi-agent framework [43]: two were core contributors to the open source project and the other three had experience developing multi-agent prototypes with the framework. Our participants were three research scientists, one software engineer, and one engineering manager.

We conduced semi-structured interviews in a one hour session to ask each participant about their development experience building multi-agent systems. In our interviews, we asked each participant questions about their prior experience developing agents, challenges they ran into, and desired features from agent debugging tools. Our exact interview questions are included in Appendix A. We took detailed notes during each interview and then did thematic analysis of interview notes to synthesize common themes. Our interviewees described three primary pain points in developing multi-agent apps, detailed below.

4.1 Understanding Long Agent Conversations is Cumbersome

The first pain-point that our participants discussed is the difficulty of understanding long agent conversations. To understand the results of a workflow, participants currently write all the messages exchanged between agents to the system console. The console is then saved as a single output text file and then reviewed post-hoc.

For a single task, on the order of 50-100+ text-heavy messages might be exchanged between agents. Each message has metadata information like the sender of the message, its recipient, and any text generated from LLM calls or tool invocations. In some cases, individual messages can themselves be difficult to interpret (e.g., the output of Python scripts written by agents). However, interpretation challenges compound as conversations grow and more agents are involved. Participants must read lengthy histories to understand both how the agents act and where things might be going wrong.

4.2 Lack of Support for Interactive Debugging

The next pain point described by participants is the current lack of support for an interactive debugging experience. Participants desired the ability to have fine-grained control over the agents as they progress through a task. This includes interrupting the agents if they seem to be stuck or going down the wrong path, resetting agents to earlier points in the conversation, and editing messages to steer agents towards the desired goal. To this end, both P2 and P4 mentioned their desire to use "breakpoints" for agent debugging. Once reached, breakpoints would interrupt agent execution, and allow developers to understand and manipulate the state of each agent before continuing. For example, PDB (python debugger) provides a comparable experience for debugging traditional python scripts [34]. Likewise, P5 drew parallels with the difficulties of debugging distributed systems, and expressed a desire for tools that capture, replay, and step through agent messages one at a time, step by step. This style of interaction would be helpful because issues often occur midway through a workflow. For instance, participants described how the AI agents often suggest reasonable plans, execute the first few steps of the plans correctly, then get stuck on a particular step, throwing off the rest of the workflow. In such cases, participants expressed a desire to reset the workflow to the last point where progress was being made, and then retry from this location—perhaps with a newly-corrected plan.

4.3 Iterating on Agent Configuration

Finally, four participants noted that iterating on agent configurations is currently a slow and arduous process. While debugging, developers are continuously tweaking their agent configurations by changing the system prompts, adding or removing agents from the team, or altering the selection of available tools. At present, developers must restart the workflows from the beginning to test the effectiveness of any given change. In cases where errors arise later in the conversation, developers must then wait considerable time to observe any impacts. Moreover, due to the stochastic nature of LLMs, the same errors might not always occur, requiring multiple run-throughs to gain confidence in a remediation. All of this slows down the debugging process considerably. To this end, participants expressed a desire to "freeze" the conversations at critical points and then iterate on potential fixes while the problematic context is isolated and in memory.

4.4 Design Goals

Based on our formative interviews and the reviewed studies on AI debugging, we synthesized the following design goals for an interactive debugging tool for AI agents. Users with such a tool should be able to:

- G1. **Understand messages exchanged between agents.** An agent debugging tool needs to expose the messages sent between agents so that users can understand the details of the conversation and how the agents are progressing through tasks. This is important for identifying *where* errors are happening in the workflow.
- G2. **Interrupt the conversation and send new messages.** Users should be able to pause/interrupt the workflow at any point, and send new messages to the agents.
- G3. **Reset back to a previous point in the workflow** Once a failure point is identified, users need the ability to reset to an earlier point in the workflow in order to experiment with steering agents to alternate paths.
- G4. **Change agent configurations.** An agent debugging tool should let users change agent configurations, such as the prompts or models used, in order to experiment with fixes.

These design goals are aimed at supporting the debugging loop presented in Figure 2, which is inspired by traditional software and AI debugging loops. An AI agent debugging tool should be able to help users both identify errors and experiment with fixes. **G1** deals

with error identification, **G4** deals with experimenting with fixes, and **G2** and **G3** support both identification and experimentation. We view this debugging process as a self-reinforcing loop where users identify errors, then experiment with fixes which allows them to refine their understanding of the error.

Figure 2: The agent debugging loop where developers iteratively identify errors and experiment with fixes that shape their understanding of the issue.

5 AGDebugger: Interactive Agent Debugging

Based on our design goals, we developed AGDebugger, an interactive debugging tool for AI agents. AGDebugger has three core features that enable interactive agent debugging: (1) it presents the messages exchanged between agents in an interactive message viewer that also lets users send new messages, (2) it lets users reset the agents workflow to earlier points in the conversation to edit, and (3) it has an overview visualization for helping users navigate long conversation histories.

5.1 Message Sending and History

The first feature of AGDebugger is the ability to send new messages to agents and view the messages exchanged between agents (design goals **G1** and **G2**). Users can send messages to start the agents working on a new task or pause agent execution to send new messages during the middle of a run. This type of interaction draws design inspiration from standard AI chat apps and prior agent creation and chat interfaces [9, 10, 21]. Our message sending feature extends these designs by providing fine grained conversation control with the ability to pause execution and send new messages to particular agents in the middle of a conversation.

Figure 3 shows the AGDebugger interface, with message sending controls appearing in panel A. This feature lets users send a new message to all other agents (broadcast) or to any one agent in particular. When a message is sent it is appended to the message queue, where all messages are processed in the order in which they arrived. Once processed, the message is recorded in the message history, along with an execution timestamp. The message queue can be run automatically with the play button or can be stepped through one message at a time by the user. This step-by-step debugging is similar to line-by-line execution provided by python debuggers like PDB [34].

In the example in Figure 3 we can see the history of the current conversation in the message history panel with the most recent messages at the bottom. Here, the Orchestrator has asked the Web Surfer to sort a table in the current web page, and the Web Surfer has initiated an internal monologue (thought) to determine which low-level action(s) will accomplish the Orchestrator's instruction. The next message in the queue is the result of the Web Surfer's action where it reports on clicking on a particular part of the page.

5.2 Message Resetting and Edits

While the ability to send messages to agents and view the conversation history is included in prior agent configuration tools, AGDebugger provides a novel interaction with the ability to reset agents to previous points in the conversation and make edits to messages (design goal **G3**). Since agents are stateful, resets and edits require more than a simple transformation of the message history (e.g. truncating the transcript) that might work for non-agentic LLM applications. AGDebugger offers robust checkpointing support to reset the states of the agents themselves (e.g., having the Web Surfer return to the web page it was visiting at that time).

Users can reset to an earlier point in a workflow in two ways. They can directly edit a historical message inline then save the edit to reset the conversation back to this timestamp. Or users can click the reset button on a message if they wish to restore the conversation back to that point without any changes to the message (e.g., to simply retry the flow). This resetting interaction provides an affordance for users to ask two core agent debugging questions:

(1) What happens if I retry the workflow from this point?
(2) What *would have happened* if this alternative message had been produced?

Figure 4 shows an example where one agent is requesting information from another agent, and the user is leveraging the edit and reset capabilities to refine the request by providing more specific instructions. This gives the user the ability to see *what would have happened* if the Orchestrator had produced a different plan. If the workflow now succeeds, they know to focus their efforts on making the plans more precise rather than perhaps tweaking how the web agent executes the plans.

5.2.1 Technical details: checkpoints and sessions. To support edit and reset, AGDebugger checkpoints each agent's state before every new message is processed (Figure 5). For some agents, these state checkpoints might be simple (e.g., in the case of stateless agents like the code Executor); for others like the Web Surfer the checkpoints are more complex and contain information like the current URL and position of the web browser viewport on the page. Nevertheless, all agents must implement two methods, `save_state` and `load_state`, that are called when the state is check-pointed or restored.

When a user requests a message reset, AGDebugger forks the conversation and creates a new *session*: the system retrieves the checkpoint corresponding to the moment of the reset target message, and restores the internal states of each agent accordingly. Messages and checkpoints before the reset are preserved and shared between sessions, while new messages and checkpoints are added only to the newly forked session. When the user is ready to resume, the target message is added back to the queue, triggering the continuation of the workflow.

For debugging and reproducibility purposes, checkpoints should capture as much information as necessary in order to restore the agents' state with high fidelity. However, it is not always feasible, desirable, or possible to *perfectly* restore state. For example, the Web Surfer relies on a web browser to access pages, and the browser's internal state arguably includes the state of any running JavaScript (not to mention the back-end state of the remote web application

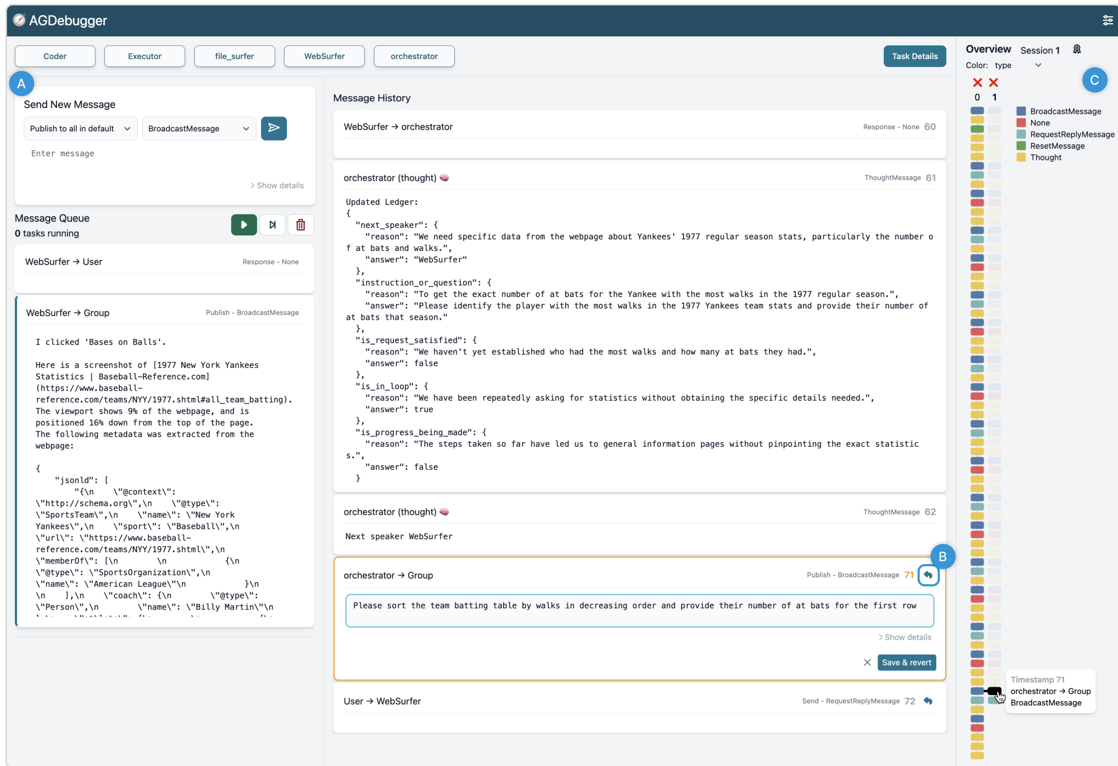

Figure 3: AGDebugger helps users interactively debug and steer their agent teams. (A) Users can interactively send new messages, control the flow of messages, and see the history of agent messages (Section 5.1). (B) Users can revert to earlier points in the workflow by resetting and editing messages (Section 5.2). (C) The overview visualization helps users make sense of long conversations and the history of edits in an interactive visualization (Section 5.3).

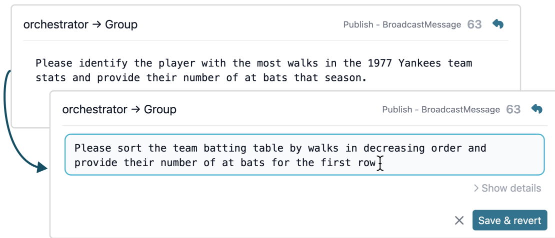

Figure 4: Users debug agent workflows by directly editing prior agent messages then restarting the workflow from that point, such as adding more specific instructions to a message to steer the agents towards the correct outcome.

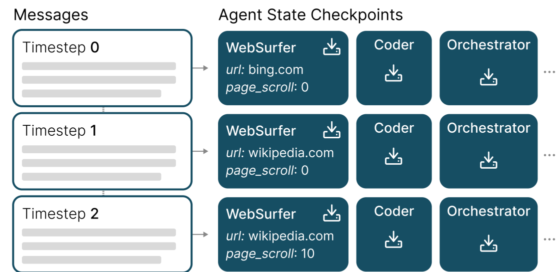

Figure 5: Agent state is captured in a checkpoint before each new message is processed to enable future message resets.

itself). Checkpointing all this internal information would be cumbersome and likely impossible to restore, and offers diminishing returns over simply recording the URL and viewport location. To this end, AGDebugger adopts a *good enough* checkpoint policy—when a user resets, AGDebugger will put the agents close to where they were at the time of the given checkpoint. From there, the agents will naturally re-consider their state before continuing their progress. The precise checkpoint fidelity needed for a given agent is left as an implementation detail to the agent developer.

5.3 Conversation Overview Visualization

To help users navigate the agent conversation and edits, we designed an overview visualization to summarize the messages in the conversation (design goals **G1** and **G3**). We draw design inspiration from code commit graph visualizations that visualize code commit history and branches [13] and unit visualizations [30]. Each message is encoded as a rectangle where a conversation is a vertical line of messages, the most recent at the bottom. As agents send new messages, they are appended to the bottom of the current session. Users can toggle the color of the rectangle to encode the message

type, sender, or recipient. The conversation overview also links to the full messages in the history view to facilitate navigating from the overview to the full messages. Clicking on a message scrolls the history view to the target message, and a message's full metadata is shown on hover

When a user resets to an earlier message and creates a new *session*, we annotate the visualization with a horizontal dash at the reset point. Messages after the fork point are displayed normally, however the prior messages have less opacity to indicate they are the same as the previous session before the fork point. Aligning the conversation forks helps users compare how the conversations differ after each edit, for example if different agents are invoked or different message types exchanged. The linear structure of our visualization facilitates understanding agents communicate over time and changes after edits. This is complimentary to other visualization approaches that show how agents communicate as a graph at a single point in time [21].

The example shown in Figure 6 shows the overview visualization with two resets. For benchmark tasks like the ones in our example, the check or X characters denote if the corresponding session is passing or failing. The example shows how the first two conversations were not outputting the correct answer, whereas the user made an edit that produced the correct answer in the final one.

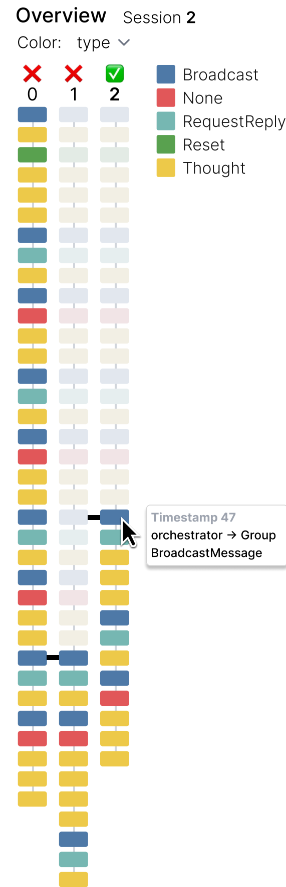

Figure 6: The interactive overview visualization summarizes the agent conversation. Each reset forks the current conversation and creates a new conversation session, represented as a new column. Users can toggle the message color to represent the message type, sender, or receiver. Message details are shown on hover and clicking navigates to the full message in the Message History view.

5.4 Agent Configuration

AGDebugger also supports configuring the agents used in a workflow (design goal **G4**). We do not consider this a primary debugging feature of the system as AGDebugger only supports basic agent configuration, however still provides an interactive way to tweak agent behavior while experimenting with edits to agent messages during a workflow. Future debugging tools can integrate the design from systems like AutoGen Studio [10] or Crew AI [9] which provide deeper customization of agents and tools in the UI.

The cards at the top of the interface show the current agents in the debugging session. For example, in Figure 3 AGDebugger shows there are five agents in the current session: the Coder, Executor, File Surfer, Web Surfer, and Orchestrator. Clicking on one of the cards shows the configurable details of the agent such as the model it uses, the system prompt, or other configuration details.

In the underlying agent framework, agents are defined through code and are very flexible. Agents can expose editable configuration options through two methods to load_config and save_config. These functions return dictionaries with values such as the system prompt, model name, or temperature which can then be edited in the agent panel in AGDebugger. After an agent's configuration is changed, any future messages will use this updated configuration.

6 User Study

To evaluate our system, we ran a two-part user study. In the first part of the study, six participants used AGDebugger to summarize the errors they found in two agent runs that had failed. In the second part, we provided a compiled list of the agents' errors to eight new participants and asked them to edit and steer the agents using AGDebugger.

6.1 Study Design

Part 1: Error Identification. Part 1 of our study serves as a preliminary user study to gather data on the agent errors identified by participants in the two tasks, assess how long it takes participants to identify messages to edit, and measure participant preference in having the ability to edit and reset messages. We recruited six participants from Microsoft all with backgrounds in computer science and experience working with LLMs. Four participants were graduate students, and two were research scientists, with the majority having experience developing AI agents and working with the GAIA benchmark for agent evaluation.

Each participant analyzed logs from two agent runs on different tasks: one using AGDebugger and another using a reduced version of the system that lacked the ability to reset messages or the overview visualization that shows differences between sessions. This reduced version represented a baseline developer workflow

where logs could be read but not interactively explored. Participants were instructed to identify errors and propose fixes, entering their responses in an online form. Participants debugged runs from the five-agent team on the two tasks described in Table 1.

For both tasks (T1 and T2), participants were given a task description, the expected output, and the incorrect agent output. The study included a demo and a 15-minute session with each system, followed by a post-task survey. The order of log reviews and system conditions was randomized and counterbalanced.

Part 2: Interactive Steering of Agents. Part 2 of our study serves as our main evaluation of AGDebugger, to gather data on how developers use the system to debug and steer multi-agent systems. To gain deeper insights, participants were assigned just one task, allowing us to closely observe their editing strategies. We recruited eight additional participants from the same tech company, all experienced with LLMs. Three participants were research scientists, and five were graduate students. Their experience with developing AI agents varied: two had no prior experience, two had limited experience, two had moderate experience, and two had extensive experience. Additionally, two participants had worked with the agent framework and the GAIA benchmark.

Participants were asked to debug a single failing agent run (T1 or T2) using AGDebugger with all features enabled. They were instructed to understand the agents' errors and steer them toward the correct output by editing messages or agent configurations. We provided a summary of errors identified in Part 1, helping to reduce onboarding time. Each study session lasted one hour, including a demo, 30 minutes for debugging, and an exit interview.

During the debugging process, we observed the types of edits participants made to the agents and how they approached the task. We also asked participants to think aloud as they used the tool. Afterward, participants completed a post-task questionnaire where they rated the usability of AGDebugger and the helpfulness of its individual features. We also asked open-ended questions to gather insights into their strategies for steering agents and their overall impressions of the system. Questions are included in Appendix B.

For analysis, we aggregated the Likert scale ratings to assess system usability and feature usefulness. We also reviewed task recordings to annotate the edits made by participants and qualitatively coded interview transcripts to identify themes related to their open-ended responses.

6.2 Part 1 Findings: Error Identification

For each task, we analyzed the six error descriptions produced by participants, which revealed four primary errors for T1 and three primary errors for T2. For example, for task T1, which involves looking up the statistics for a Yankees player from the 1977 team (see Table 1 for exact benchmark question phrasing), participants identified several key issues.

First, the Web Surfer agent failed to correctly parse the table containing the statistics once it navigated to the correct page. Additionally, the Orchestrator's instructions to the Web Surfer were often too high-level, omitting crucial steps like sorting the table first. Finally, the agents frequently defaulted to returning statistics for a famous player from the 1977 Yankees instead of the correct player. The errors identified by participants in part 1 were shared with participants in part 2 when they began debugging.

We observed that participants spent significant time reading through messages to trace the agents' progress and pinpoint where errors occurred. The error descriptions produced in both conditions (while using AGDebugger and the baseline system) were equally high-quality. Every participant also edited a message at least once while they were reading the log in the AGDebugger condition. Reading agent messages for debugging took considerable time before participants started to edit any messages, with participants taking about 10 minutes on average (out of the total 15 minutes allocated per task) before starting to experiment with edits.

Even just for identifying errors, participants found the extra features in AGDebugger helpful, with five out of six preferring AGDebugger to the baseline version. The ability to interactively edit and reset messages was the primary reason for their preference. As one participant noted:

> *"Message editing is very useful. It's like you want to change some ground truth in the middle of the messages and then see how the agents behave from there but with code that's very hard to do" — P2*

This interactive capability helped streamline debugging, making AGDebugger more favorable compared to the baseline, which lacked editing features.

6.3 Part 2 Findings: AGDebugger Facilitates Interactive Steering and Debugging

In the second part of our study, our participants used the interactive debugging features of AGDebugger to try to steer the agents towards outputting the correct answer for the task. While steering the agents towards outputting the exact correct answer proved quite difficult (two out of eight participants were able to steer agents towards exact right answer), interacting with the agent teams helped participants refine their understanding of how the agents operated and why they were making errors.

Participants found AGDebugger helpful to facilitate this debugging and overall rated the system as helpful and that they would use it again for debugging in the future (Figure 7 Left). We also collected ratings for the three primary debugging features in AGDebugger (Figure 7 Right) to better understand what contributed to the overall system ratings. Participants rated the message resetting feature most highly with a mean rating of 4.9/5. Over the 30 minute debugging session, every participant edited messages at least once, with several participants making five separate edits (Figure 8).

Participants described how the ability to reset to earlier messages and steer the conversation helped them understand what is going on in the workflow, generating insights that would not have been possible otherwise:

> *"When I'm trying to develop what a language model is doing, what I want to do is see what's the result when I create slight variations on a given point in a given exchange. And so being able to edit the message was I think the core insight of this entire system. And I think that it is a necessary insight when developing any sort of language models that interact with each other." — P7*

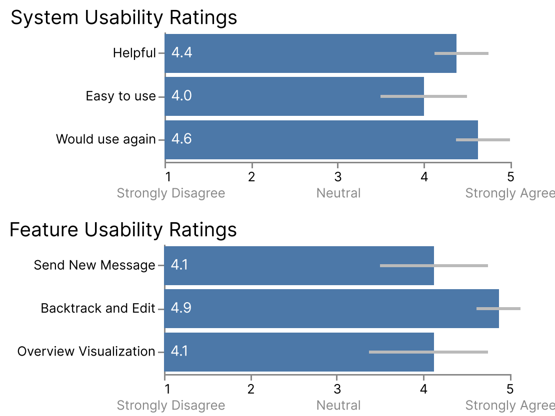

Figure 7: System and feature-level ratings scores from part 2 of our user study. Participants found AGDebugger helpful for debugging with the ability to backtrack and edit as the most highly rated feature. Mean scores are plotted along with a 95% confidence interval.

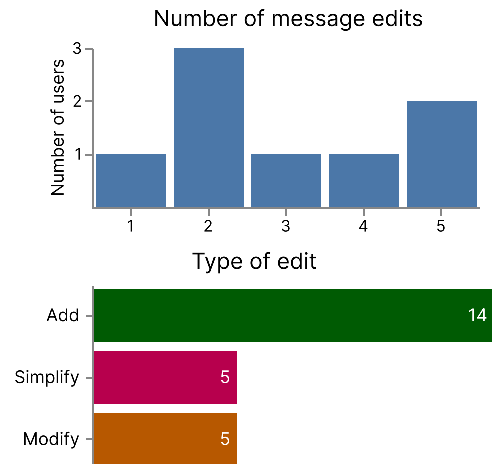

Figure 8: Each participant in part 2 of our user study used the message editing feature to help them debug, with some participants editing messages five separate times. The most common edit was to add more specific instructions to the message, followed by equal rates of simplification of instructions and modifying the goal of the plan.

By editing messages, participants could see *what would have happened* later on in the task. This gives them the ability to do lightweight counterfactual testing for agent workflows and pinpoint where exactly the errors are coming from in the workflow.

Sending new messages and the overview visualization were also both highly rated features with average scores of 4.1/5. Especially for long conversation histories or after many edits, the visualization helped participants navigate where they were in the workflow.

Although AGDebugger allows users to update basic agent configuration like the agent's system prompt or the model used for LLM calls, no participants used this feature. Participant behavior and comments indicated they were worried about accidentally breaking the agent behavior without more knowledge of its implementation. Our conjecture is that editing the messages exchanged between agents is a more lightweight and faster first step to test agent behavior than updating the agent configuration and that updating agent configuration might occur over a longer debugging period.

6.4 Three User Approaches to Steer Agents

Across the eight study sessions, participants edited messages a total of 24 times. We analyzed the changes made to the messages and categorized them into three high level categories:

(1) **Add** more specific instructions
(2) **Simplify** instructions by removing text
(3) **Modify** the goal of the plan

Examples of each edit type from the study are included in Figure 9. The first type of edit was the most common: **adding instructions that are more specific and concrete**. More than half of all the message edits fell into this category (14/24). This edit type closely corresponds to a commonly observed failure mode of AI agent teams: that the plans created are often not at the right level of granularity. Therefore, when another agent is told to execute this plan it is open to interpretation and might fail. The example on the left of Figure 9 demonstrates this plan refinement where the participant changed a message from a high-level instruction from the Orchestrator to the Web Surfer to something more directly actionable like telling the web surfer to first sort the table and then return the result in the first row. By making these types of edits, users are able to answer questions like: *With a more actionable plan, would the agents have made progress?*

The next type of edit is almost the inverse of the first: **making instructions simpler**. LLMs have a tendency to be verbose and struggle to attend to all parts of long instructions [25], so concise instructions are key. With this type of edit, users would simplify the task to see if the LLMs could first succeed on a sub-task before moving on to the next component. In the example in the middle of Figure 9, the participant noticed that the agent was struggling with the compound instruction to first identify the player with the most walks and then provide their number of at bats. Therefore, they removed the second part of the instruction to nudge the agent towards completing a simpler sub-task first. This type of edit enables users to ask: *With an easier plan, would agents have made progress?*

The final category of edit is the most drastic and involves **modifying the plan** generated by the agents. For example, in Figure 9 right, we see an example where a participant nudges the agents towards using a code-based approach to solve the task since they were previously failing with looking up the information on the web. The results of this new execution inform users' decisions about how the agents might approach the task more successfully. Another instance of this type of edit occurred when a participant changed the URL the agents chose to visit, hoping the new URL might help them succeed in the task by providing better information. Like the previous two types of edits, this type of edit is a form of counterfactual testing where a user is investigating: *What would have happened if the agents came up with a different plan?*

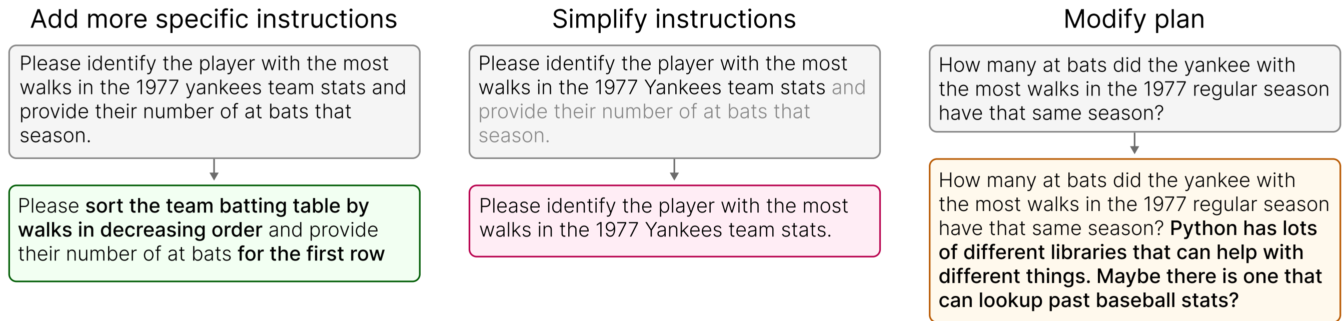

Figure 9: Examples of the three types of edits participants made to steer models. Add edits occurred when users add extra instructions or make instructions more specific. Simplify edits involved removing unnecessary details from messages. Modify edits involve totally changing the instruction or result.

6.5 User Study Limitations

Our current study design is subject to several limitations. First, the 30-minute debugging session provided to participants in the second part of the study may have been insufficient for them to engage with all the issues in a failing agent run. Second, we tested the system with users on only two benchmark tasks from GAIA. While we believe these tasks are representative of general agent development tasks, it remains to be seen how the tool assists developers working on other types of tasks. Future studies could explore longer-term usage of tools like AGDebugger and investigate how developers utilize them for developing and debugging multi-agent teams over extended periods. Additionally, we acknowledge that tools like AGDebugger might be even more beneficial when used concurrently during the active development or tweaking of agents, an aspect not fully captured in our study.

7 Discussion

AGDebugger is designed to help developers debug multi-agent AI teams, where multiple agents with distinct roles and capabilities collaborate to solve complex tasks. Our system and user study build on prior work in interactive probing of AI and Large Language Models (LLMs) [2, 16, 31, 38, 39], extending the concept of interactive debugging to multi-agent AI teams. These teams pose unique challenges due to the autonomous nature of the agents, their use of external tools to interact with the world, and the long, multi-turn conversations that occur between agents. Understanding such complex behaviors requires new interactive tools for analyzing and probing agent interactions.

AGDebugger introduces several key mechanisms to enable this interactivity, allowing developers to modify messages exchanged between agents and explore counterfactual scenarios by altering these intermediate communications. This allows users to probe how changes to agent messages affect their collective behavior. In the following section, we discuss the current limitations of our approach (e.g., handling non-resettable actions, verifying edits) and outline the challenges and opportunities for advancing interactive, steerable multi-agent systems in the future.

7.1 Open Challenges for AI Agent Steering

The development and study of AGDebugger revealed several open challenges in developing interactive systems for debugging multi-agent teams.

Dealing with non-resettable agent actions. One challenge with resets (i.e., rolling back the agent's state) is managing actions that affect the external world beyond the agent. For instance, if an agent sends an email, it is nearly impossible to *un*-send it. As a result, AGDebugger's checkpoint and reset mechanisms are limited to controlling the agent's internal state or handling situations where an undo operation is possible. This limitation underscores a critical safety-related design consideration for rollback strategies. When agents perform reversible actions, their behavior can be more flexible, as these actions can be reset or modified by tools like AGDebugger. However, recognizing that some actions are irreversible encourages developers to implement safeguards for AI agents, such as monitoring, pre-execution validation, or stricter constraints on actions that cannot be undone.

Steering requires deep knowledge of implementation. A challenge that our user study revealed was that steering agents was relatively easier with a deeper technical understanding of the agent's implementation. In particular, knowledge on how each agent processes instructions and uses its tools. For example, the Web Surfer agent, which participants debugged, is designed to perform one task at a time, such as navigating to a specific URL or clicking a button on a web page. However, some participants attempted to modify the Web Surfer's input plan with reasonable but overly complex tasks, such as instructing it to visit three websites to gather information before responding. These changes were unsuccessful because the Web Surfer is designed to handle only one task per instruction, and could not perform multiple steps in sequence. Additionally, participants did not update the agent's configuration, as they lacked in-depth knowledge of how the agent utilized its tools. Since the input to AI agents is text, it might be less clear exactly what kind of text they expect whereas in traditional programming APIs these input constraints are often made more explicit through types and input checks. This challenge highlights the need for better communication of an agent's capabilities, enabling end users to better understand and influence the agent's behavior.

Did my edit actually work? A similar challenge participants faced when interacting with the AI agents was tracing the effect of edits. Whenever an agent makes an inference call to an LLM, the response could be non-deterministic (depending on model temperature settings). As agent conversations progress, more and more messages are saved and then injected into the context for the next model call. Therefore, when a message is changed, particularly if it is later on in the conversation, the LLM response might attend more to the earlier messages rather than the one that was updated. This reflects a known limitation of LLMs where they do not always attend to all information in long contexts equally well [25]. From a debugging perspective, this means that even after a message update it can be hard to immediately understand if the new run is different from before. Sometimes the edit may not have an obvious effect until after several conversation turns. This, combined with the non-deterministic nature of LLM responses, often left participants unsure if their interventions were having the intended effect, echoing the challenges identified in prior literature on the difficulty of debugging cascading errors in model pipelines [36, 44].

We saw this challenge materialize for several of our participants when they wanted to test a hypothesis by changing a message but the agents still did not obviously abide by the new plan. This led to frustration when the agents were not responding to changes. For example, after several rounds of editing one of our participants changed a message to very directly tell the agents what *not* to do by adding "DO NOT GIVE ME A SUMMARY OF THE WHOLE PAGE, I JUST NEED THE LIST OF CITIES." to the end of the message.

Participants that made edits to messages *earlier* in the agent conversation seemed to have more success steering. Both of the participants who successfully steered the agents towards producing the correct answer editing messages towards the beginning of the conversation rather than at the end. One changed the agent plan (a *modify plan* edit) and told the agents to use a code-based approach rather than searching for the result on the web. The other simplified the plan the agents were executing on a web page (a *simplify* edit), telling them to first sort a table before returning results. Since these were edits to earlier messages, the agents' behavior actually changed and they returned the correct result.

7.2 Future Directions for Multi-Agent Debugging

Future work can build on the design and findings of AGDebugger to further improve the debugging experience for AI agents along the following dimensions.

Direct performance feedback to agents. In AGDebugger, users guide agents by directly modifying their messages, in essence simulating agent behavior as the edited message is delivered as a normal agent message. This approach gives users greater control over agent outputs but requires a deep understanding of the agents' mechanisms to craft effective edits. Alternatively, users can provide real-time natural language feedback or rewards, simply indicating when agents are off-course and need to adjust. Prior research has developed interactive systems that incorporate this type of feedback for single model calls or prompts, but not for the more complex, multi-agent configurations that we examined [32]. Future research could explore methods for integrating user feedback into multi-agent AI workflows, allowing agents to dynamically adapt to feedback.

Ensuring Robustness and Generalizing Fixes. While building complex AI agent teams, developers need different ways to test for robustness and generalize fixes for recurring error patterns. When users edit a message and get a different outcome in AGDebugger, it is not immediately obvious if this was a reliable edit or a fortunate outcome based on stochastic model responses. Running the same edit multiple times can help developers determine if the change produces consistent results across different scenarios, which is key to identifying and generalizing solutions that prevent similar errors from occurring in the future. Prior systems that help users compare LLM responses across prompt iterations or model changes might be extended to help facilitate this comparison in multi-agent settings [3, 18, 19]. Exploring how repeated edits contribute to identifying recurring error patterns and their systematic resolution presents a valuable opportunity for future research, as it helps to ensure that agents improve their behavior holistically rather than merely correcting isolated issues.

Automatic error identification. Finally, our user study indicated that reading and pinpointing errors in agent conversations is a time-consuming part of debugging before users can experiment with changes. Future research might investigate automatic methods of error identification to help users spot issues in long multi-turn agent conversations more quickly and identify points for intervention, building on recent research that uses LLMs as a judge to more easily parse if model responses are high quality [33, 37].

8 Conclusion

In conclusion, we present the design and evaluation of an interactive multi-agent debugging tool, AGDebugger. AGDebugger enables users to steer multi-agent AI teams by editing the messages sent between agents and reverting them back to earlier checkpoints. The findings from our user study reveal common user strategies towards steering agent teams, some of the open difficulties of this task, and highlight the need for fine-grained interactive control of multi-agent AI teams. Future research can focus on refining feedback mechanisms from users to agents and how to design AI agent interactions to be safe, easy to debug, and easy to *reset.*

Acknowledgments

Many thanks to Grace Proebsting, Omar Shaikh, Steve Drucker, Gonzalo Ramos, and the MSR AI Frontiers team for their feedback on this project. We also thank our user study participants for their participation and reviewers for their feedback.

References

[1] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: a case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, New York, NY, USA, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[2] Saleema Amershi, Max Chickering, Steven M. Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis Tools for Machine Learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*.

Association for Computing Machinery, New York, NY, USA, 337–346. https://doi.org/10.1145/2702123.2702509
[3] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena Glassman. 2023. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. arXiv:2309.09128 [cs.HC]
[4] Autoblocks AI. 2024. Autoblocks. https://www.autoblocks.ai. Accessed: 2024-12-02.
[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., Red Hook, NY, USA, 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
[6] Ángel Alexander Cabrera, Erica Fu, Donald Bertucci, Kenneth Holstein, Ameet Talwalkar, Jason I. Hong, and Adam Perer. 2023. Zeno: An Interactive Framework for Behavioral Evaluation of Machine Learning. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 419, 14 pages. https://doi.org/10.1145/3544548.3581268
[7] Furui Cheng, Vilém Zouhar, Robin Shing Moon Chan, Daniel Fürst, Hendrik Strobelt, and Mennatallah El-Assady. 2024. Interactive Analysis of LLMs using Meaningful Counterfactuals. arXiv:2405.00708 [cs.CL] https://arxiv.org/abs/2405.00708
[8] Yuheng Cheng, Ceyao Zhang, Zhengwen Zhang, Xiangrui Meng, Sirui Hong, Wenhao Li, Zihao Wang, Zekai Wang, Feng Yin, Junhua Zhao, and Xiuqiang He. 2024. Exploring Large Language Model based Intelligent Agents: Definitions, Methods, and Prospects. arXiv:2401.03428 [cs.AI] https://arxiv.org/abs/2401.03428
[9] CrewAI. 2024. CrewAI. https://www.crewai.com/. Accessed: 2024-12-02.
[10] Victor Dibia, Jingya Chen, Gagan Bansal, Suff Syed, Adam Fourney, Erkang Zhu, Chi Wang, and Saleema Amershi. 2024. AutoGen Studio: A No-Code Developer Tool for Building and Debugging Multi-Agent Systems. arXiv:2408.15247 [cs.SE] https://arxiv.org/abs/2408.15247
[11] Hugging Face. 2024. GAIA Benchmark Leaderboard. https://huggingface.co/spaces/gaia-benchmark/leaderboard Accessed 08-2024.
[12] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang (Eric) Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. 2024. *Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks*. Technical Report MSR-TR-2024-47. Microsoft. https://www.microsoft.com/en-us/research/publication/magentic-one-a-generalist-multi-agent-system-for-solving-complex-tasks/
[13] GitKraken. 2024. GitKraken Commit Graph: Bring color & clarity to your commit history. https://www.gitkraken.com/solutions/commit-graph. Accessed: 2024-09.
[14] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model Based Multi-agents: A Survey of Progress and Challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, Kate Larson (Ed.). International Joint Conferences on Artificial Intelligence Organization, Online, 8048–8057. https://doi.org/10.24963/ijcai.2024/890 Survey Track.
[15] Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. WebVoyager: Building an End-to-End Web Agent with Large Multimodal Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 6864–6890. https://aclanthology.org/2024.acl-long.371
[16] Ellen Jiang, Kristen Olson, Edwin Toh, Alejandra Molina, Aaron Donsbach, Michael Terry, and Carrie J Cai. 2022. PromptMaker: Prompt-based Prototyping with Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 35, 8 pages. https://doi.org/10.1145/3491101.3503564
[17] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues? https://openreview.net/forum?id=VTF8yNQM66
[18] Minsuk Kahng, Ian Tenney, Mahima Pushkarna, Michael Xieyang Liu, James Wexler, Emily Reif, Krystal Kallarackal, Minsuk Chang, Michael Terry, and Lucas Dixon. 2025. LLM Comparator: Interactive Analysis of Side-by-Side Evaluation of Large Language Models. *IEEE Transactions on Visualization and Computer Graphics* 31, 1 (2025), 503–513. https://doi.org/10.1109/TVCG.2024.3456354
[19] Tae Soo Kim, Yoonjoo Lee, Jamin Shin, Young-Ho Kim, and Juho Kim. 2024. EvalLM: Interactive Evaluation of Large Language Model Prompts on User-Defined Criteria. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 306, 21 pages. https://doi.org/10.1145/3613904.3642216
[20] Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. 2015. Principles of Explanatory Debugging to Personalize Interactive Machine Learning. In *Proceedings of the 20th International Conference on Intelligent User Interfaces* (Atlanta, Georgia, USA) *(IUI '15)*. Association for Computing Machinery, New York, NY, USA, 126–137. https://doi.org/10.1145/2678025.2701399
[21] LangChain. 2024. LangGraph Studio. https://github.com/langchain-ai/langgraph-studio. Accessed: 2024-12-02.
[22] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. arXiv:2303.17760 [cs.AI] https://arxiv.org/abs/2303.17760
[23] Brian Y. Lim, Anind K. Dey, and Daniel Avrahami. 2009. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 2119–2128. https://doi.org/10.1145/1518701.1519023
[24] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. 2010. TurKit: human computation algorithms on mechanical turk. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology* (New York, New York, USA) *(UIST '10)*. Association for Computing Machinery, New York, NY, USA, 57–66. https://doi.org/10.1145/1866029.1866040
[25] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. https://doi.org/10.1162/tacl_a_00638
[26] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. 2020. On Faithfulness and Factuality in Abstractive Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 1906–1919. https://doi.org/10.18653/v1/2020.acl-main.173
[27] Gregoire Mialon, Thomas Scialom, Clémentine Fourrier, Thomas Wolf, and Yann LeCun. 2024. GAIA: A Benchmark for General AI Assistants. https://ai.meta.com/research/publications/gaia-a-benchmark-for-general-ai-assistants/
[28] Besmira Nushi, Ece Kamar, Eric Horvitz, and Donald Kossmann. 2017. On human intellect and machine failures: troubleshooting integrative machine learning systems. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, New York, NY, USA, 1017–1025.
[29] OpenAI. 2024. Chat Playground. https://platform.openai.com/playground/. Accessed: 2024-09-05.
[30] Deokgun Park, Steven M. Drucker, Roland Fernandez, and Niklas Elmqvist. 2018. Atom: A Grammar for Unit Visualizations. *IEEE Transactions on Visualization and Computer Graphics* 24, 12 (2018), 3032–3043. https://doi.org/10.1109/TVCG.2017.2785807
[31] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Amy J. Ko, and James Landay. 2010. Gestalt: integrated support for implementation and analysis in machine learning. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology* (New York, New York, USA) *(UIST '10)*. Association for Computing Machinery, New York, NY, USA, 37–46. https://doi.org/10.1145/1866029.1866038
[32] Savvas Petridis, Benjamin D Wedin, James Wexler, Mahima Pushkarna, Aaron Donsbach, Nitesh Goyal, Carrie J Cai, and Michael Terry. 2024. ConstitutionMaker: Interactively Critiquing Large Language Models by Converting Feedback into Principles. In *Proceedings of the 29th International Conference on Intelligent User Interfaces* (Greenville, SC, USA) *(IUI '24)*. Association for Computing Machinery, New York, NY, USA, 853–868. https://doi.org/10.1145/3640543.3645144
[33] Promptfoo. 2024. Promptfoo. https://www.promptfoo.dev/. Accessed: 2024-12-02.
[34] Python Software Foundation. 2024. The Python Debugger (pdb). https://docs.python.org/3/library/pdb.html. Accessed: 2024-09-05.
[35] Marco Tulio Ribeiro and Scott Lundberg. 2022. Adaptive Testing and Debugging of NLP Models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 3253–3267. https://doi.org/10.18653/v1/2022.acl-long.230
[36] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt.
[37] Shreya Shankar, J.D. Zamfirescu-Pereira, Bjoern Hartmann, Aditya Parameswaran, and Ian Arawjo. 2024. Who Validates the Validators? Aligning LLM-Assisted Evaluation of LLM Outputs with Human Preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*

(Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 131, 14 pages. https://doi.org/10.1145/3654777.3676450
[38] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M. Rush. 2022. Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models. arXiv:2208.07852 [cs.CL] https://arxiv.org/abs/2208.07852
[39] Ian Tenney, Ryan Mullins, Bin Du, Shree Pandya, Minsuk Kahng, and Lucas Dixon. 2024. Interactive Prompt Debugging with Sequence Salience. arXiv:2404.07498 [cs.CL] https://arxiv.org/abs/2404.07498
[40] Sandra Wachter, Brent D. Mittelstadt, and Chris Russell. 2017. Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR. arXiv:1711.00399 http://arxiv.org/abs/1711.00399
[41] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs.SE] https://arxiv.org/abs/2407.16741
[42] James Wexler, Mahima Pushkarna, Tolga Bolukbasi, Martin Wattenberg, Fernanda B. Viégas, and Jimbo Wilson. 2020. The What-If Tool: Interactive Probing of Machine Learning Models. *IEEE Trans. Vis. Comput. Graph.* 26, 1 (2020), 56–65. https://doi.org/10.1109/TVCG.2019.2934619
[43] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang (Eric) Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. https://www.microsoft.com/en-us/research/publication/autogen-enabling-next-gen-llm-applications-via-multi-agent-conversation-framework/
[44] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. PromptChainer: Chaining Large Language Model Prompts through Visual Programming. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 359, 10 pages. https://doi.org/10.1145/3491101.3519729
[45] Tongshuang Wu, Marco Tulio Ribeiro, Jeffrey Heer, and Daniel Weld. 2021. Polyjuice: Generating Counterfactuals for Explaining, Evaluating, and Improving Models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, Online, 6707–6723. https://doi.org/10.18653/v1/2021.acl-long.523
[46] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 385, 22 pages. https://doi.org/10.1145/3491102.3517582
[47] Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. 2024. OS-Copilot: Towards Generalist Computer Agents with Self-Improvement. arXiv:2402.07456 [cs.AI] https://arxiv.org/abs/2402.07456
[48] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 437, 21 pages. https://doi.org/10.1145/3544548.3581388
[49] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. https://openreview.net/forum?id=oKn9c6ytLx

A Formative interview details

In our formative study, we conducted semi-structured interviews where we asked participants the following questions about their experience developing multi-agent systems:

(1) What task have you been using AutoGen for and why?
(2) How were you solving this task before?
(3) What obstacles and difficulties did you run into while using AutoGen and authoring multi-agent applications?
(4) How did you solve these difficulties?
(5) Are there any pain points you were unable to solve?

B User study part 2 questions

In the second part of our user study, we asked participants quantitative ratings questions along with open-response questions. The ratings questions collected via a survey were:

(1) How much experience do you have developing AI agents? *(5 point Likert)*
(2) Do you have prior experience using autogen or agnext? *(Yes/No)*
(3) Do you have prior experience with the GAIA benchmark? *(Yes/No)*
(4) I found this system helpful *(5 point Likert)*
(5) I found this system easy to use *(5 point Likert)*
(6) I would like to use this system again in the future *(5 point Likert)*
(7) Rate how much you found this feature helpful: Sending new messages *(5 point Likert)*
(8) Rate how much you found this feature helpful: Backtracking to previously sent messages and editing *(5 point Likert)*
(9) Rate how much you found this feature helpful: Conversation overview visualization *(5 point Likert)*

After completing the task, the participants in study 2 were asked the following open-response questions in the interview:

(1) *(If did not steer to right answer)* Given more time, do you think you would have been able to steer the models to output the correct answer?
(2) Can you describe your approach to the task, how did you try to intervene?
(3) After intervention, how has your opinion about which issues are the most actionable changed?
(4) How did you feel that having the ability to edit messages and configuration influenced your understanding of the agents' errors?
(5) What other ways would you have liked to steer the models?