Let's Fix this Together: Conversational Debugging with GitHub Copilot

Yasharth Bajpai[†], Bhavya Chopra[†], Param Biyani[†], Cagri Aslan[‡],
Dustin Coleman[‡], Sumit Gulwani[‡], Chris Parnin[¶], Arjun Radhakrishna[‡], and Gustavo Soares[‡]
[†]Microsoft, Bengaluru, India; [‡]Microsoft, Redmond, USA; [¶]Microsoft, Raleigh, USA
{ybajpai, t-bhchopra, t-pbiyani, caslan, dcoleman, sumitg, cparnin, arradha, gsoares}@microsoft.com

***Abstract*—Despite advancements in IDE tooling, code understanding, generation, and automated repair, debugging continues to present significant challenges. Existing debugging strategies available to developers in literature are often too mechanical and rigid for day-to-day issues. Recent advances in Large Language Models (LLMs) promise practical solutions that allow for more free-form debugging strategies. While LLMs offer satisfactory assistance in some cases, they often leap to action without sufficient context, making implicit assumptions and providing inaccurate responses. Moreover, the dialogue between developers and LLMs predominantly takes the form of question-answer pairs, placing the burden of formulating the correct questions and sustaining multi-turn conversations on the developer.**

We introduce ROBIN, a novel multi-agent conversational AI-assistant within GitHub Copilot Chat, specifically designed for debugging. ROBIN moves beyond the question-answer pairs by introducing the *investigate & respond* pattern, that focuses on using information gathered automatically from the IDE or gathered interactively from the developer before responding. ROBIN incorporates a general debugging strategy to systematically analyze bugs to sustain collaborative interactions while ensuring that the conversation does not deviate from the debugging task at hand. Through a within-subjects user study with 16 industry professionals, we find that equipping ROBIN to—(1) leverage the *insert expansion* interaction pattern, (2) facilitate *turn-taking*, and (3) utilize *debugging strategies*—leads to lowered conversation barriers, a 2.5x improvement in bug localization and a substantial 3.5x improvement in bug resolution compared to AI-assisted debugging in Visual Studio prior to ROBIN.

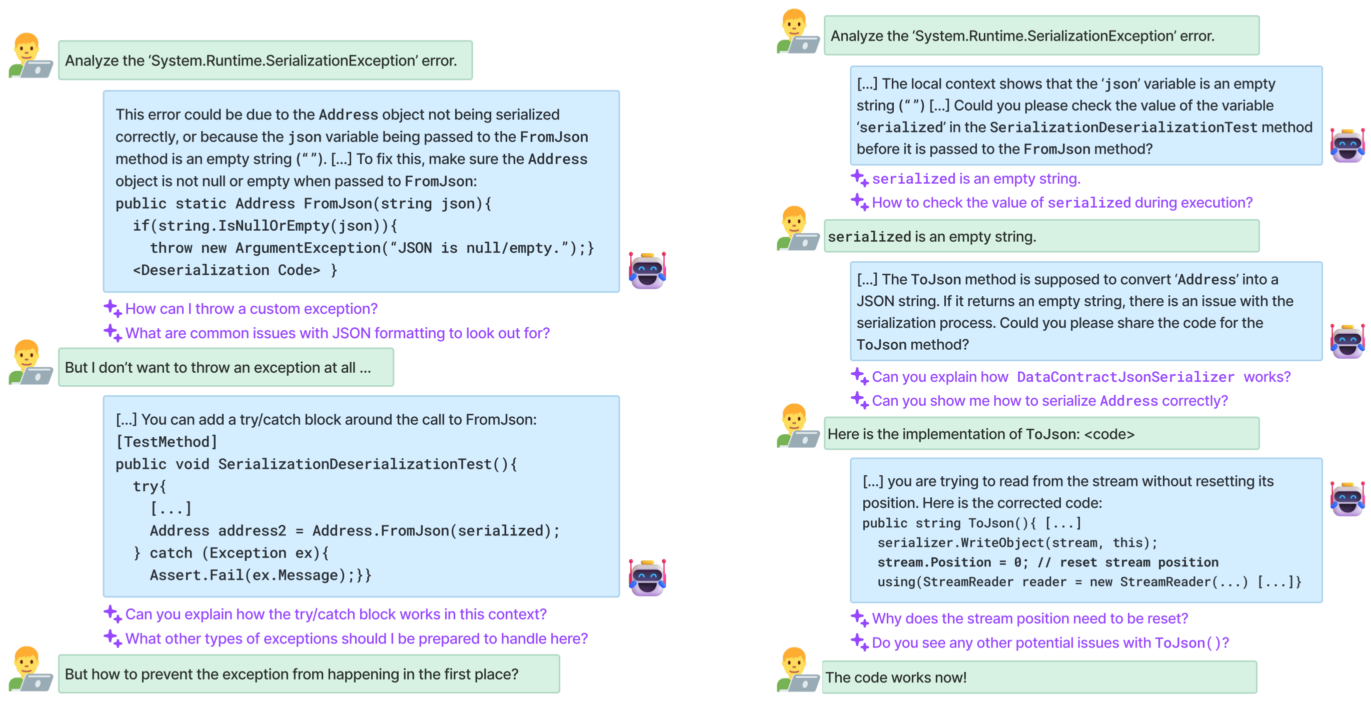
I. INTRODUCTION

Debugging has traditionally been a challenging and time-consuming component of software development. Unlike other programming tasks, debugging is often more free-form, which, if left unchecked, can consume a significant portion of developers' time [1]. Prior work suggests that developers may spend over 35% of their development time on debugging their code [2]. This has prompted the development of a wealth of debugging tools available in modern Integrated Development Environments (IDEs) like Visual Studio [3] and IntelliJ [4]. These IDEs offer dedicated debugging modes with traditional tools like *watch expressions* and *breakpoints* to sophisticated ones like *hot reloading* and *time-travel debugging*. Despite these tools, the process of identifying, localizing, analyzing, and resolving bugs in live environments with multiple signals can overwhelm developers, particularly non experts who end up underutilizing available tools [5]–[7].

Recently, with the advent of conversational large language models (LLMs) like ChatGPT [8] there has been an emergence of conversational assistants designed to aid developers in various aspects of the software development life-cycle. [9]–[16]. AI programming assistants, such as GitHub Copilot Chat [9], are becoming increasingly integrated into popular IDEs as a widely used tooling category by the software community for day-to-day tasks, and debugging is no exception. However, despite their potential and the improvements they have brought, they have not yet fully resolved the complexities and challenges in debugging. These challenges with AI debugging assistants can be divided into two major categories: their *limitations in conversationality*, and their dependency on the user for *direction and strategy*.

LLM-powered conversational AI assistants provide a pragmatic approach to enable interactive implementations of the debugging journey. However, the current generation of these tools typically operate on the *question-answer adjacency pair interaction pattern* [17], where the user initiates the conversation with a query, and the AI tool responds to *close the conversation*. In order to close the conversation, the assistant may make assumptions without sufficient information or may skip essential debugging steps such as localizing the root cause, leading to inaccurate or unhelpful responses [18]. For a knowledge and process-intensive domain like debugging, this approach often leads to sub-optimal results without the user heavily guiding the conversation and providing the necessary context. Experts experientially acquire a collection of strategies for performing programming tasks and in the process learn the usage of IDE tools [19]. Hence for non-experts, an efficient and successful debugging conversation is often dependant on the AI assistant directing them towards the right investigation strategies using available debugging tools.

The conversation in Figure 1a highlights issues arising from both these aspects through a pre-mature closing of the conversation. Here, the user is attempting to fix a test case failure and the assistant's first response is a premature effort to provide a solution—suggesting to throw an exception with a custom message. This response attempts to close the conversation sub-optimally—the assistant has not taken any steps to diagnose the problem or interact with the user to gather more information. The user rejects the fix and reopens the conversation, attempting to redirect the conversation towards



Analyze the 'System.Runtime.SerializationException' error.

This error could be due to the Address object not being serialized correctly, or because the json variable being passed to the FromJson method is an empty string (""). [...] To fix this, make sure the Address object is not null or empty when passed to FromJson:

```
public static Address FromJson(string json){
  if(string.IsNullOrEmpty(json)){
    throw new ArgumentException("JSON is null/empty.");}
  <Deserialization Code> }
```

How can I throw a custom exception?

What are common issues with JSON formatting to look out for?

But I don't want to throw an exception at all ...

[...] You can add a try/catch block around the call to FromJson:

```
[TestMethod]
public void SerializationDeserializationTest(){
  try{
    [...]
    Address address2 = Address.FromJson(serialized);
  } catch (Exception ex){
    Assert.Fail(ex.Message);}}
```

Can you explain how the try/catch block works in this context?

What other types of exceptions should I be prepared to handle here?

But how to prevent the exception from happening in the first place?

(a) Conversation with baseline assistant, demonstrating sub-optimal fixes

Analyze the 'System.Runtime.SerializationException' error.

[...] The local context shows that the 'json' variable is an empty string ("") [...] Could you please check the value of the variable 'serialized' in the SerializationDeserializationTest method before it is passed to the FromJson method?

serialized is an empty string.

How to check the value of serialized during execution?

serialized is an empty string.

[...] The ToJson method is supposed to convert 'Address' into a JSON string. If it returns an empty string, there is an issue with the serialization process. Could you please share the code for the ToJson method?

Can you explain how DataContractJsonSerializer works?

Can you show me how to serialize Address correctly?

Here is the implementation of ToJson: <code>

[...] you are trying to read from the stream without resetting its position. Here is the corrected code:

```
public string ToJson(){ [...]
  serializer.WriteObject(stream, this);
  stream.Position = 0; // reset stream position
  using(StreamReader reader = new StreamReader(...) [...]}
```

Why does the stream position need to be reset?

Do you see any other potential issues with ToJson()?

The code works now!

(b) Conversation with ROBIN, demonstrating guided localization

Fig. 1: Contrasting responses for the same task across AI-assistants. The purple text represents generated follow-ups.

a deeper analysis and fix. However, the assistant jumps to a superficial fix again, suggesting catching and re-throwing the exception inside the test method. This frustrates the user due to lack of constructive suggestions.

To this end, we present ROBIN[1], an innovative multi-agent workflow-based debugging system integrated into the Visual Studio IDE as part of the GitHub Copilot Chat extension[2]. ROBIN introduces the *investigate & respond agentic pattern*, that aids users in systematic debugging through streamlined conversations, while minimizing user intervention required to repair conversations and provide debugging strategies. ROBIN leverages LLMs as reasoning engines to provide interactive and collaborative debugging assistance through a chat-based interface. It incorporates a general debugging strategy that includes interaction patterns to optimize debugging conversations. ROBIN analyzes exception information, code context, and user queries, and guides developers through a series of steps to explore potential hypotheses, gather more information, and utilize IDE debugging tools to fix issues. Figure 1b depicts a developer's conversation with ROBIN as they attempt to fix error in Figure 1a—note how ROBIN proactively investigates the error and asks the user for additional information when necessary, leading to a successful fix.

We evaluate ROBIN through a task-based study with 16 industry professionals comparing its performance and user feedback with existing approaches. Our results demonstrate that ROBIN leads to 1.5x faster problem resolution, 3.5x higher success rates, and higher engagement with ROBIN. Furthermore, we witness an increased use of follow-ups and debugger tooling inside the IDE towards bug resolution.

Our contributions in this paper are as follows:

- We propose ROBIN, an LLM-powered *investigate & respond* multi-agentic debugging assistant inside IDEs.
- We implement conversational debugging strategies to facilitate effective interactions.
- We conduct and analyze a within-subjects study using ROBIN and GitHub Copilot Chat (as our baseline) for debugging real-world bugs with 16 industry professionals.
- We discuss insights and recommendations derived from the user study for builders of automated debugging tools.

II. DESIGN FOR AI-ASSISTED DEBUGGING

A. Design Principles

Our design of ROBIN, considers the following four principles:

1) **D1. Balanced strategy:** The debugging process should balance reaching a resolution (fix) as soon as possible while avoiding reaching an incorrect or premature cause of the problem.
2) **D2. Debugging tools:** A debugging agent should have access to the same tools and context as a developer, enabling both shared context, such as knowledge of runtime values of variables, and also allowing a partnership in using the debugging tools effectively with the developer.

[1]The name, ROBIN, is inspired by Batman's supporting character; a resourceful, determined, and skilled investigator. We identify the developer to be the protagonist (Batman) in the debugging process.

[2]GitHub Copilot Chat: https://learn.microsoft.com/en-us/visualstudio/ide/visual-studio-github-copilot-chat

3) **D3. Human-in-the-loop:** Completely automating the debugging process without any visibility or partnership with the developer is undesirable, resulting in latency, increased costs due to numerous calls to LLMs, inaccurate or incomplete results, and a lack of transparency.
4) **D4. Conversational capabilities:** A debugging agent should facilitate effective multi-turn conversations, allowing both the agent and the user to clarify, guide, and recover the conversation.

To implement D1 (Balanced Strategy), we take inspiration from explicit debugging strategies from Latoza et al. [20], where a strategy language encodes the debugging process [21]. In designing prompts for a debugging agent, we provide a high-level strategy for the agent, letting the reasoning capabilities of large language models adapt it as needed. To implement D2 (Debugging Tools), we ensure that ROBIN has knowledge and access to debugging features, including but not limited to breakpoints, local variable windows, and stack trace windows. To implement D3 (Human-in-the-loop), we focus on a conversational interaction model where the debugging agent requests inputs from the developer. We describe D4 (Conversational Capabilities) in the following subsection.

B. Conversational Capabilities

Facilitating effective conversations between AI agents and developers presents a unique set of challenges. The existing interaction model used by conversational agents, including tools like Copilot Chat, is primarily based on a *question-answer adjacency pair* structure [17]. This interaction pattern puts the responsibility on the developer to not only initiate the conversation but also to frame questions accurately at the right time. Meanwhile, the AI agent's role is to generate a response that aims to conclude the conversation. However, this model assumes that developers deeply understand the debugging process so that they can ask the right questions. Additionally, Large Language Models (LLMs) may prematurely attempt to resolve tasks, even when the information provided is insufficient. This propensity to act without adequate information can lead to assumptions that result in inaccurate or misleading responses. To overcome these limitations, we aim to enhance ROBIN with three critical conversational capabilities:

- Guide the Conversation Through Diverse Interaction Patterns: ROBIN will be capable of initiating the dialogue, asking clarifying questions, and suggesting the next steps, thus actively guiding the debugging process. This is referred to as ***insert expansion***, which allows the user to contribute interactionally relevant information, requested by ROBIN, to their conversational turn.
- Inducing Principles for effective communication: ***Gricean maxims*** of conversation propose that effective communication is guided by the right amount and orderly presentation of information, its truthfulness, and relevance to the context of discussion [18]. With ROBIN, we propose to softly imbibe these four maxims through behavioural adaptations in various system components, enhancing conversation quality.
- Generate Relevant Follow-up Suggestions: Many existing AI conversational assistants offer follow-up suggestions, i.e., potential utterances the user may want to continue the conversation with (see, for example, GitHub Copilot Chat and Bing Copilot [9], [22]). However, the usability of these follow-ups is questionable due to their generic nature. Irrelevant or generic follow-ups can be distracting and can also misdirect the conversation towards a non-fruitful direction. In contrast, ROBIN will maximize the relevance of debugging conversations by integrating domain understanding and conversation context into the generation of follow-up suggestions. This ensures that the suggestions naturally converge towards the goal of effective debugging.

III. ROBIN: AN AI AGENT FOR DEBUGGING EXCEPTIONS

This section outlines details the multi-agent workflow approach with ROBIN, our proposed AI debugging assistant, to enable considerations we discuss in II.

A. Agent Descriptions

ROBIN employs four agents, as seen in Figure 2: Responder agent, Collaborative agent, Follow-up agent, and Context Retrieval agent. The first three are LLM-based agents. Table I provides the prompt instruction highlights for these agents. Next, we describe these agents and explain how their goals and implementation align with our design considerations, using the conversation from Fig 1b as an example:

- ***Responder Agent***: This agent is designed to produce a response with a solution for the debugging task. The prompt instructions for this agent guide the model to follow the debugging process outlined in Section II-A. Additionally, the agent adheres to a *question-answer interaction pattern*, attempting to close the conversation with a comprehensive and structured response, in line with D3 (Human-in-the-loop). In Fig 1b, the Responder agent is responsible for the last message from the assistant—it contains the fix and is the answer to the developer's first message.
- ***Context Retrieval Agent***: In accordance with D2 (Debugging Tools), this agent exploits IDE features to collect the necessary context for the debugging task. It contributes to the conversation by replying with this context. We developed the Context Retrieval Agent using the APIs available in the Visual Studio IDE infrastructure. This allows us to access the Visual Studio Debugger and File programmatically Viewer and extract various types of information: Exception Information, Local State Information and Stack Information. Exception information includes the exception message, type, stack trace, and the exact line of code where the exception is thrown. The local state Information consists of open-source files in the IDE, code selections, and local variable context. While the stack information includes logical code snippets from the current stack and the corresponding active line at the exception time. In Fig 1b, the first message from the assistant is aware of the value of the `json` variable as a message from the Context Retrieval agent was added to the internal messages before the message was generated.

- ***Collaborative Agent***: This agent assists in acquiring more context and information from the user while isolating the source of the issue. It generates messages encouraging user participation in the conversation, fostering a collaborative debugging process per design consideration D3 (Human-in-the-loop). Its prompt instructions also promote the use of debugging tools by directing the user towards adding breakpoints, stepping through code, and handling exceptions. This approach creates domain-specific interaction patterns in line with the design considerations D3 (Human-in-the-loop) and D4 (Conversational Capabilities; Section II). Going back to Fig 1b, the Collaborative agent is responsible for the first and second responses from the assistant. These messages ask for additional information and context from the developer. Further, depending on the context and prior interaction, these messages may have included instructions on using debugging tools to gather the required information.
- ***Follow-up Agent***: Concentrating on the generation of follow-up suggestions pertinent to the conversation, this agent ensures a coherent and comprehensive interaction, in line with D4 (Conversational Capabilities). Its prompt instructions direct the agent to produce prompt suggestions that align with the debugging process, guiding the user towards the goal of fixing the bug. In Fig 1b, the purple text below each response from the assistant are follow-ups generated by the Follow-up agent. Note how these follow-ups are specific to this conversation. For example, the follow-up "`serialized` is an empty string" is a likely answer to the assistant's original question, and "How to check the value of `serialized` during execution?" is a follow-up question on how to acquire the information needed to answer the assistant's question.

TABLE I: Prompt instructions for LLM-based agents

Agent	Prompt Instruction Highlights
Responder	Structure your response as follows: • Explain the exception message • Explain possible causes for the exception • Localize the error with the available code context • Suggest a fix to the user's code
Collaborative	• Proactively ask for any additional portions of source code, or values of variables from the user IF it can help you reason better about the exception/error. • The user can access debugger features such as breakpoints, step over, step in, watch window, locals window, immediate window, call stack, and so on.
Follow-Up	• Propose two very pertinent follow-up prompts that the user could make use of to either seek a clarification, or to answer a question asked by ROBIN. • These prompts should guide the conversation towards bug localization and resolution. Do not digress from the goal of localizing and fixing the error

Prompt Optimizations. Practical constraints of limited context windows and information relevance warrant some crude filtering of context when it exceeds a certain maximum length set within the system, particularly in two instances. First, `inner_messages` are pruned by removing messages from the top of the conversation (barring the initial context setting messages). Second, if the context retrieved by Context Retrieval Agent is more than its individual limit, we limit the stack frames to the first N frames and reduce the open document to fit within the context.

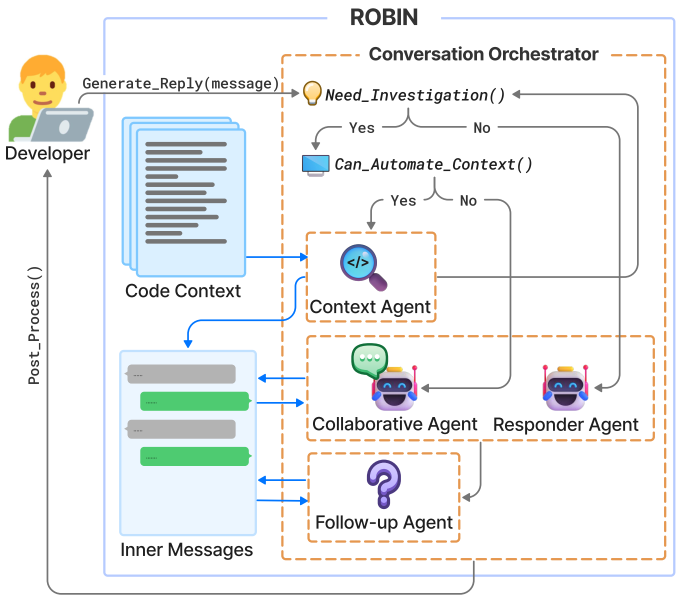


Fig. 2: ROBIN: *Multi-agent workflow.* Gray arrows indicate agent workflow, blue arrows indicate read/write operations.

B. ROBIN*'s Workflow*

As depicted in Figure 2, ROBIN operates as a *conversable agent* [23], interacting with the developer via conversational messages. ROBIN's workflow introduces the *investigate & respond* agentic pattern. ROBIN iteratively and interactively investigates the problem, gathering relevant information to reason about the correct solution. Once ROBIN is confident it has enough information, it produces a response. This approach distinguishes ROBIN's responses from prompt-based techniques such as chain-of-thought (CoT) [24], which relies on pre-trained knowledge, and retrieval-augmented generation (RAG) [25] methods that ground the agent's response in a single step of context retrieval.

When the developer posts a message, the `Generate_Reply` function is triggered, prompting ROBIN to generate a response. ROBIN maintains an `inner_messages` state, which serves as an internal chat history. This state is initially empty and gets updated each time an agent replies to the conversation. To generate a response, ROBIN coordinates a discussion with other agents within the system, each playing their respective roles in the multi-agent workflow. Each agent is equipped with intelligent processing capabilities, powered by either LLMs or deterministic business logic. These agents contribute to the conversation at different stages, as dictated by the workflow. ROBIN's workflow is structured around decision points, known as *functions*, that determine which agent will participate next in the conversation.

The initial workflow step utilizes an LLM-powered function, `Need_Investigation`, to assess the progress of the debugging process, as outlined in D1 (Balanced Strategy). Given the available information and messages in the conversation, the function determines whether the bug's root cause can be correctly identified and fixed, or if the bug requires to be investigated further. If the model is not confident about the root cause, and can not propose an immediate fix, it will respond with "YES". Otherwise, it will return "NO".

If the cause and location and of the bug has been accurately identified, a "conversation closing" response containing the suggested fix is generated by the Responder agent. If not, ROBIN concentrates on the fault localization and investigates further. To facilitate this, ROBIN uses the `Can_Automate_Context` function to decide whether additional useful information can be automatically retrieved. If the fault localization step remains incomplete even after this, the conversation is then directed to the Collaborative agent to gather more information from the developer. Currently, `Can_Automate_Context` permits the Context Retrieval agent to contribute once at the start of the conversation.

Regardless of the path taken during the debugging process, ROBIN consistently concludes by generating follow-ups, ensuring a coherent and comprehensive interaction. Subsequently, the `Post_Process` function synthesizes a user-facing response using `inner_messages` posted by agents.

IV. USER STUDY

We conducted a within-subjects study with 16 software developers to assess the impact of ROBIN's debugging approach and use of conversation patterns on developers' AI-assisted debugging experiences. Alongside considering task completion metrics, we aim to understand participant perception of ROBIN by prompting them to think aloud while solving the tasks and through semi-structured questions following the tasks.

TABLE II: Overview of participants, their experience, and expertise in C#, alongside the AI-assistants used per task.

ID	Exp. (yrs)	C# expertise	Warm-up	Task 1	Task 2
P1	2.5	Intermediate	ROBIN	Baseline	ROBIN
P2	1	Beginner	ROBIN	Baseline	ROBIN
P3	5	Intermediate	ROBIN	Baseline	ROBIN
P4	6	Intermediate	ROBIN	Baseline	ROBIN
P5	10	Expert	ROBIN	Baseline	ROBIN
P6	21	Expert	ROBIN	Baseline	ROBIN
P7	1	Beginner	ROBIN	Baseline	ROBIN
P8	13	Expert	ROBIN	Baseline	ROBIN
P9	4	Intermediate	Baseline	ROBIN	Baseline
P10	7	Expert	Baseline	ROBIN	Baseline
P11	2.5	Intermediate	Baseline	ROBIN	Baseline
P12	2.5	Intermediate	Baseline	ROBIN	Baseline
P13	7	Expert	Baseline	ROBIN	Baseline
P14	8	Expert	Baseline	ROBIN	Baseline
P15	2	Beginner	Baseline	ROBIN	Baseline
P16	1	Beginner	Baseline	ROBIN	Baseline

Participants: We recruited participants with at least 1 year of experience in C# development from our organization through snowball sampling (Table II). Prior consent was obtained from all participants to audio and screen record our interactions and use anonymized quotes for research purposes.

Study protocol: We conducted a within-subjects study with two conversational debugging AI-assistants integrated within Visual Studio, both of which have access to the exception, call stack and local variables at the point of exception — (A) ***Baseline:*** GitHub Copilot Chat Preview extension, and (B) **ROBIN:** as described in Section III; both leveraging OpenAI's GPT-4[3] for all study sessions. We conducted one-hour-long video conferencing sessions with each of the participants. The study began with demographic questions about their programming background in C#, use of Visual Studio and debugger tools, and experiences with using LLMs for debugging. Next, participants were presented with a warm-up task for familiarization with the AI-assistant & the IDE. They were randomly assigned to start with one of the AI-assistants for the first task and the other for the second task. Participants were prompted to think out loud as they solved the tasks to report any thoughts and feedback about the AI-assistants. Lastly, we asked semi-structured questions for qualitative feedback.

Study tasks: We selected one warm-up task and two study tasks that involved comprehending, localizing and fixing runtime exceptions with increasing difficulty. Figure 3 shows the task setup for the study. The tasks comprise bugs mined from two open-source repositories and adapted for increased tractability. We sought bugs that had committed fixes for comparison with participant solutions and that varied in complexity and localization effort. We observe that the ROBIN outright invokes the Responder agent for the warm-up task and invokes the Collaborative agent for tasks 1 and 2 (indicating greater difficulty). Based on observations from pilot studies, we added time bounds of 15 & 25 minutes to each task, respectively.

(Warm-up Task) Index Out Of Range: Encountered in accessing a list element. A beginner-friendly debugging task to bring participants up-to-speed with the integrated AI-assistants and available debugging tools in Visual Studio.

(Task 1) Serialization Exception[4]: Moderately difficult task, exposing participants to a `Serialization Exception`, arising through the `System.Text.JSON` library where it encounters a null string while trying to deserialize a string. Participants must *localize* the bug to the `ToJson` method (not present in the exception call stack) and identify a subtle logical error: failure to reset the stream position to 0.

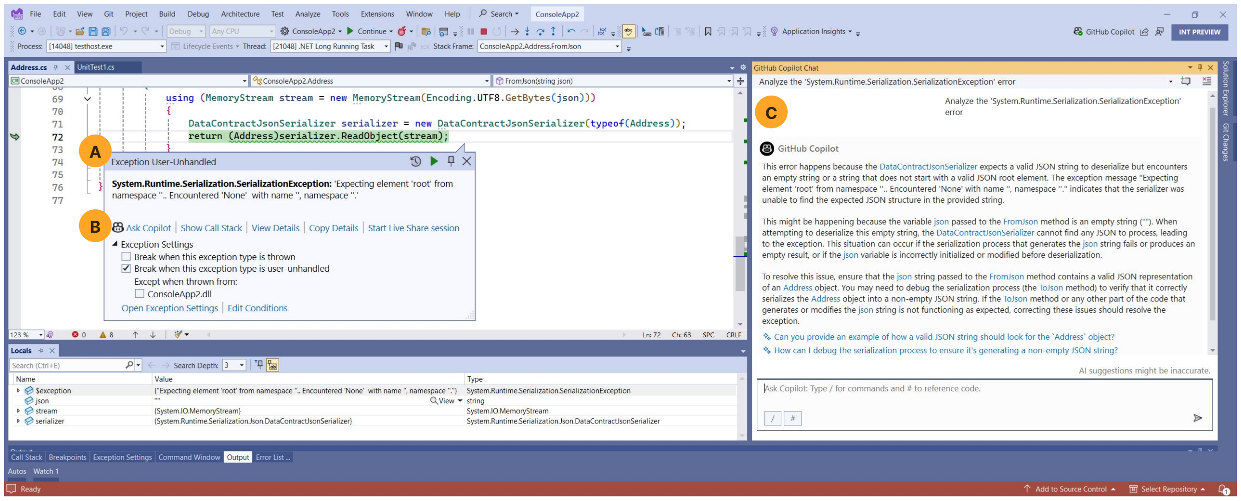
(Task 2) Arithmetic Overflow Exception[5]: An obscure `YAMLException` wrapped around an `Arithmetic Overflow`. Bug localization involves finding the correct branch of exploration and navigating several call stacks deep to locate the `DeserializeIntegerHelper` method. The problem is then reduced to a flaw in the logical handling of a lower-edge integer value manipulation.

Analysis: We assess the performance of each AI-assistant based on the time spent, successful identification of the root

[3]GPT-4: https://openai.com/index/gpt-4/

[4]*protobuf-net*, issue#191: github.com/protobuf-net/protobuf-net/issues/191

[5]*YAMLDotNet*, issue#673: github.com/aaubry/YamlDotNet/issues/673


A
B
C
Exception User-Unhandled
Ask Copilot | Show Call Stack | View Details | Copy Details | Start Live Share session
Exception Settings
Open Exception Settings | Edit Conditions
GitHub Copilot Chat
GitHub Copilot
Locals
Call Stack | Breakpoints | Exception Settings | Command Window | Output | Error List ...
Ready
Add to Source Control
Select Repository

```
using (MemoryStream stream = new MemoryStream(Encoding.UTF8.GetBytes(json)))
{
    DataContractJsonSerializer serializer = new DataContractJsonSerializer(typeof(Address));
    return (Address)serializer.ReadObject(stream);
}
```

System.Runtime.Serialization.SerializationException: 'Expecting element 'root' from namespace ''.. Encountered 'None' with name '', namespace ''.'

- [ ] Break when this exception type is thrown
- [x] Break when this exception type is user-unhandled

Except when thrown from:

- [ ] ConsoleApp2.dll

Name	Value	Type
$exception	{"Expecting element 'root' from namespace ''.. Encountered 'None' with name '', namespace ''."}	System.Runtime.Serialization.SerializationException
json	""	string
stream	{System.IO.MemoryStream}	System.IO.MemoryStream
serializer	{System.Runtime.Serialization.Json.DataContractJsonSerializer}	System.Runtime.Serialization.Json.DataContractJsonSerializer

Analyze the 'System.Runtime.Serialization.SerializationException' error

This error happens because the DataContractJsonSerializer expects a valid JSON string to deserialize but encounters an empty string or a string that does not start with a valid JSON root element. The exception message "Expecting element 'root' from namespace ''.. Encountered 'None' with name '', namespace ''." indicates that the serializer was unable to find the expected JSON structure in the provided string.

This might be happening because the variable json passed to the FromJson method is an empty string (""). When attempting to deserialize this empty string, the DataContractJsonSerializer cannot find any JSON to process, leading to the exception. This situation can occur if the serialization process that generates the json string fails or produces an empty result, or if the json variable is incorrectly initialized or modified before deserialization.

To resolve this issue, ensure that the json string passed to the FromJson method contains a valid JSON representation of an Address object. You may need to debug the serialization process (the ToJson method) to verify that it correctly serializes the Address object into a non-empty JSON string. If the ToJson method or any other part of the code that generates or modifies the json string is not functioning as expected, correcting these issues should resolve the exception.

Can you provide an example of how a valid JSON string should look for the `Address` object?

How can I debug the serialization process to ensure it's generating a non-empty JSON string?

AI suggestions might be inaccurate.

Ask Copilot: Type / for commands and # to reference code.

Fig. 3: Study Setup. Participants are exposed to the exception window (A), which has a “Ask Copilot” button to invoke the AI-assistant (B). The assistant is available as a chat panel on the right side (C).

TABLE III: Success rates & engagement with AI-agents

Task	# localizations	# fixes	avg time (tool)	avg time (self)
T1-ROBIN	8 *of* 8	8 *of* 8	8.4 mins	2.6 mins
T1-Baseline	4 *of* 8	3 *of* 8	4.6 mins	9.7 mins
T2-ROBIN	7 *of* 8	6 *of* 8	10.2 mins	3.5 mins
T2—Baseline	2 *of* 8	1 *of* 8	8 mins	15.1 mins

cause of bugs, and patterns in developer-assistant conversations. We performed inductive thematic analysis with two annotators to gather qualitative insights while collecting timestamps of activities performed by participants. For the purpose of analysing study results, a bug is considered to be successfully ‘localized’ when the participant has narrowed down the root of the issue to the method requiring a fix. Subsequently, a ‘fix’ is considered successful if it is semantically similar to the fix made by the maintainers of the open-source repositories.

Additionally, to determine differences in the nature of developer-AI conversations across the baseline and ROBIN, three authors open-coded all 32 conversation logs obtained from participant interactions for tasks 1 and 2. Each user prompt is labeled with one or more codes from Table IV. To find repeating occurrences of conversation patterns, we mine all possible sub-patterns and draw insights using the top-16 patterns (having individual counts > 6) in Section V-B.

V. RESULTS

In our study, participants’ use of the assistants for the warm-up task reveals that both the baseline and ROBIN are equally effective in improving participants’ understanding of bugs and suggesting fixes for bugs that require minimal localization effort. However, ROBIN significantly boosts productivity through its *Collaborative Agent* workflow for accurately localizing bugs and supports learning experiences for novice developers owing to its *collaborative* human-in-the-loop behaviour. We summarize participants’ task success metrics in section V-A,

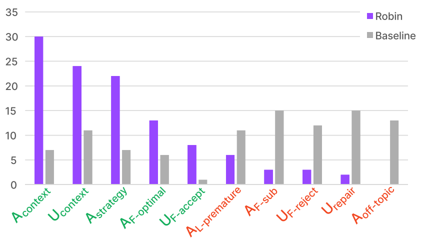
TABLE IV: Labels for user & AI-assistant utterances; with conducive traits in green, non-conducive traits in red, & neutral traits in blue text.

Label	Description
User	
U_{invoke}	Invoke AI-assistant through UI
$U_{question}$	Ask question
$U_{context}$	Provide code context or variable values
$U_{F\text{-}accept}$	Accept AI generated fix
$U_{F\text{-}reject}$	Reject AI generated fix
U_{repair}	Request correction to repair conversation course
AI-Assistant	
$A_{explain}$	Provide explanation for code/exception/fix
$A_{L\text{-}context}$	Request IDE context to guide localization
$A_{L\text{-}strategy}$	Provide debugging steps to guide localization
$A_{L\text{-}pre}$	Perform shallow/premature localization
$A_{F\text{-}optimal}$	Fix generation addressing the root cause
$A_{F\text{-}sub}$	Fix generation based on incomplete localization
$A_{off\text{-}topic}$	Derailed from the debugging objective

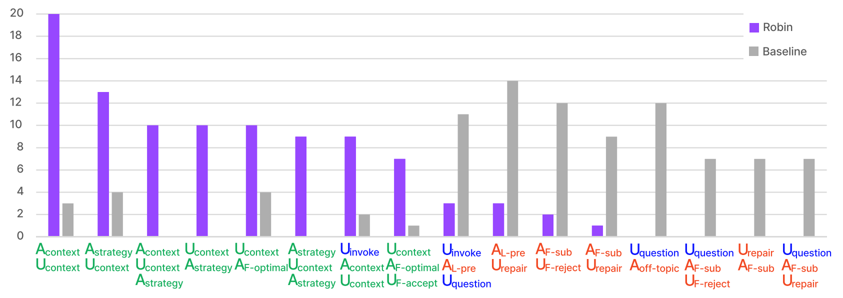
and their interactions with the baseline and ROBIN in section V-B, followed by qualitative insights in sections V-C–V-E.

A. *Task Performance and Debugging Behaviors*

We logged 80 prompts made by 16 participants in total. Table III provides a summary of task completion metrics. Notably, utilizing ROBIN, we observe enhanced task-completion success rates, including a **2.5x improvement in bug localization** and a substantial **3.5x improvement in bug resolution** ($\chi^2 = 9.3932$, $p < 0.01$). Figure 5 illustrates a prolonged self-effort in bug localization with the baseline AI-assistant, while ROBIN hand-holds developers through bug localization. Table III also indicates an increased engagement with ROBIN, supported by participants spending more time in debugging with ROBIN than on their own.



(a) Frequencies of conversation labels



(b) Most frequent conversation-pattern snippets

Fig. 4: Total occurrences of conversation labels across all user study sessions for ROBIN and the baseline, indicating increased conducive traits and sub-conversation patterns with ROBIN.

B. *Drift in conversation patterns with* ROBIN

To assess the quality of the underlying conversations between developers and the AI-assistants, we derive 13 open-codes, defined in Table IV. We also categorize these conversational traits as conducive, non-conducive, or neutral for debugging scenarios. Upon comparing the frequencies of individual labels (Figure 4a) and those of the mined sub-conversation patterns (Figure 4b), we observe an increase in non-conducive conversational elements with the baseline assistant, while recording multiple conducive conversation chains for ROBIN.

Notably, in Figure 4b, we see high occurrence of ROBIN's sub-patterns involving $A_{context}$, $A_{strategy}$, and $U_{context}$, which facilitate bug localization by requesting code context or specific variable values, while also detailing ways to obtain this information. Section V-D discusses instances where ROBIN shares strategies to gather the required context. Further, we also see a sub-pattern involving the generation of an optimal fix and its subsequent acceptance ($A_{F\text{-}optimal}$–$U_{F\text{-}accept}$).

On the other hand, the baseline assistant shows high occurences of non-conducive sub-conversation patterns, involving frequent repairs being made by users (Collaborative debugging equips developers with actionable plans) before and after sub-optimal fix generations ($A_{F\text{-}sub}$). The baseline assistant's responses also go off-topic at times ($A_{off\text{-}topic}$). Another set of contrasting patterns shows how ROBIN tries to engage in seeking additional context post invocation (U_{invoke}–$A_{context}$), whereas the baseline often performs premature localization as its first response (U_{invoke}–$A_{L\text{-}pre}$).

These above-mentioned drifts in conversational patterns translate into observable improved task completion rates and increased engagement, which we discuss in detail in the following sections (V-C–V-E).

C. *Single-turn responses lead to premature localization and sub-optimal fixes*

Most participants did not accept the fixes suggested by the baseline AI-assistant (P1, P5, P10–P13, P16). Across all sessions, participants believed that the baseline assistant only helped them handle the errors better by throwing custom exceptions or handling case-by-case values using conditional logic (also seen in Figure 1a. P11 said, *"I would likely not use this fix. I could throw a custom* `ArgumentException` *to handle this better, but I still want to find the deeper issue as to why the string is empty in the first place."*

Few participants further reflected on this characteristic of the baseline assistant, and hypothesized that these sub-optimal fixes are a result of *"pre-mature bug localization"*. P12 said that fixes generated by the baseline are trying to *"cure the symptoms instead of the actual disease,"* and this sentiment was shared by P9–P11 as well. P9 said, *"this is trying to localize the bug a bit too soon and just hide it, instead of finding what is giving rise to the issue."* Figure 5 shows several instances (P1–P3, P5, P7 in Task 1; P10–P13, P15, P16 in Task 2) where the baseline assistant generates a fix (dark green bars) before succeeding or, in most cases, even attempting to localize the bug. As we conducted the user study sessions, we observed that these brief attempts at generating fixes often distracted beginner and intermediate participants from their initial debugging plans, sometimes leading them to accept sub-optimal fixes (P1–P3, P7 in Task 1; P15 in Task 2).

On the other hand, ROBIN enables participants to follow a streamlined debugging workflow and generates fixes upon successful localization, as also seen in Figure 5. Participants notice the investigative step-by-step approach followed by ROBIN in locating the root cause of the issue. P5 commented that *"This is providing me thoughtful explanations and steps. The other version gave me code to solve the bug upfront, which confused me even more."*

D. *Collaborative approach enables guided localization*

All participants followed steps provided by ROBIN to add breakpoints and step-throughs to report variable values. They appreciated ROBIN's collaborative approach, wherein it guides on how/where to add breakpoints, and *"telling which variables to keep track of,"* as seen in the following instances:

> [...] To confirm this, you can add a breakpoint at this line and check the value of `result`. If `result` is −92233720368[...], then this is where the overflow happens. Please perform this step and let me know what you find.
>
> [...] Set a breakpoint at the `foreach` loop in `DeserializeValue`. [...] When the execution

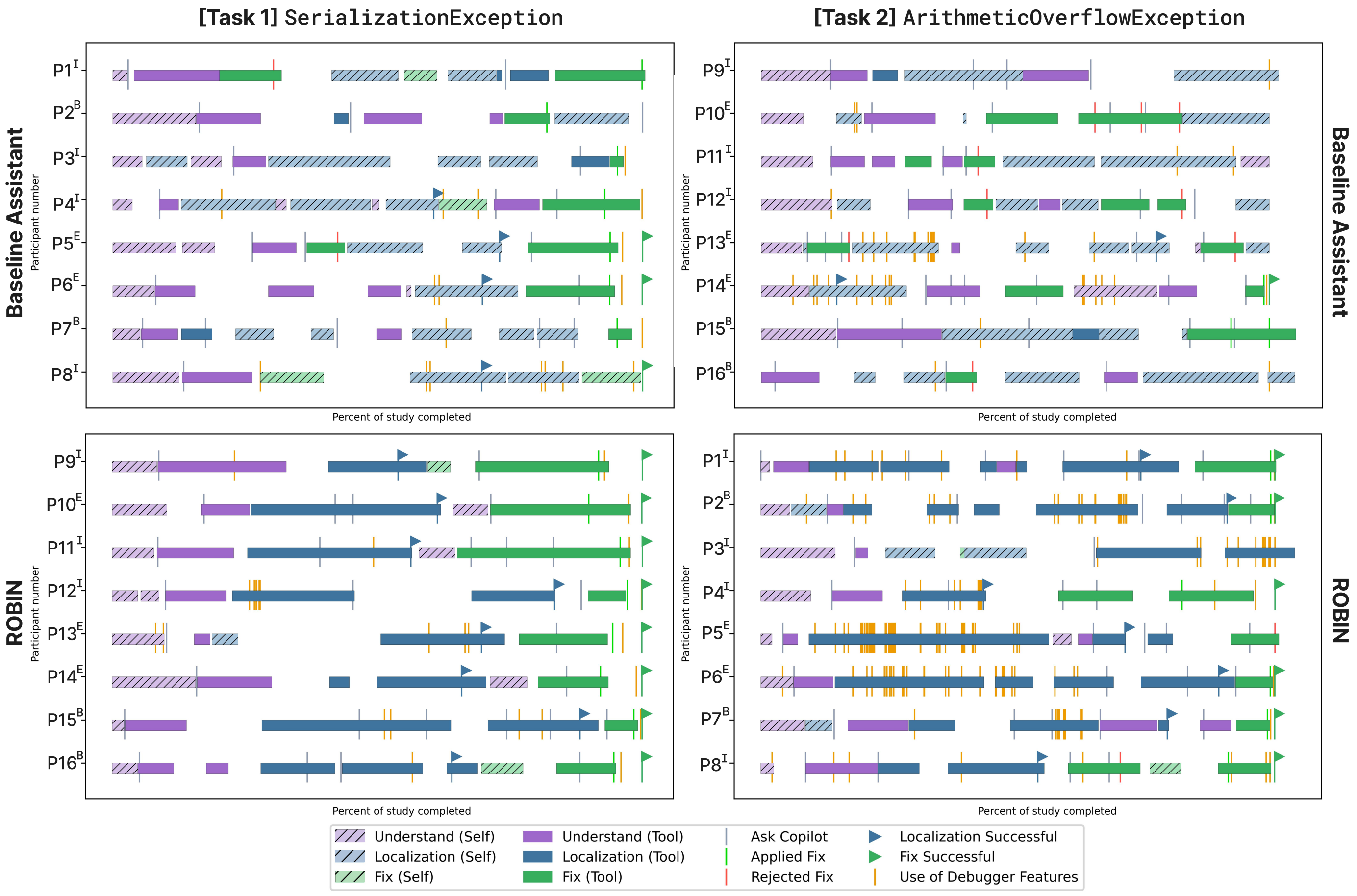


Fig. 5: ***Timelines of Participant Activity.*** The debugging workflow is interrupted by periods of *sub-optimal* bug fixes for the baseline AI-assistant (seen as intermittent dark green bars before localization). ROBIN presents streamlined debugging workflows, progressing to bug resolution only after successful localization. We also observe prolonged self-localization effort (long slashed blue bars) with the baseline, and sparing use of debugger features; while ROBIN guides participants' localization effort (long dark blue bars) with increased use of debugger features.

hits the breakpoint, use "Step Over" to execute the code line-by-line. Monitor the values of `parser` and `nodeType` in each iteration and please provide their values when the exception is thrown.

Figure 1b also shows a snapshot from a conversation where ROBIN requests additional debugging information. This characteristic often led beginner participants to draw parallels between ROBIN and senior software developers. P2 said, *"I often get stuck with the most pointless bugs while working, and you don't feel like going to ask a senior SDE. This chat assistant would help a lot in these cases."* Few participants also viewed their interactions with ROBIN as learning experiences. The exploratory nature of responses aided participants' navigation of unfamiliar source code as they solved the tasks. P10, an expert developer, expressed such debugging assistance would be *"invaluable to new contributors"*. Further, performing the tasks proposed by ROBIN helped beginner participants learn IDE features better, while *"feeling better equipped and knowing what to do the next time I see a similar bug"* (P1).

We observed higher use of debugger features by participants of all expertise levels while using ROBIN. This is clearly seen in Figure 5, with the increased occurrence of vertical yellow bars with guided localization efforts (dark blue bars). Additionally, we find this increased use to be statistically significant for Task 2 ($t = 2.7496$, $df = 14$, $p < 0.05$), which required higher localization effort than Task 1.

We also note P16's experience, where they expected an actionable plan on which variable values to track from the baseline debugging assistant after having been exposed to ROBIN, and expressed how they felt *"stuck,"* not knowing *"how to get more help"* from the assistant.

E. Aligned follow-up suggestions lower conversation barriers

P14 appreciated how ROBIN's follow-up questions lie in the *"vicinity"* of their query and task at hand. P7 was pleasantly surprised when they saw a follow-up suggesting an answer to ROBIN's question and said, *"this tells me what the model is expecting to hear, and I can directly click this to move ahead."* Most participants expressed that ROBIN provided them with questions they would want to ask next (P1, P2, P4–P7, P9–P15). These prompts for follow-up with ROBIN facilitated turn-taking with minimal time spent on formulating queries to continue conversing. Figure 1b shows follow-ups generated by ROBIN for Task 1 for a participant. Follow-ups generated by the baseline assistant often weren't specific to the participants' local code and conversational context and led to off-topic responses, derailing from the objective of localization and fix generation. For instance, P2 said, *"this went off-track because I used the follow-up"* when they received a follow-up suggestion to understand serialization using the `JsonSerializer` library, instead of `DataContractJsonSerializer` which was used in their code. We also observe several instances of participants hovering over the baseline assistant's follow-ups and ultimately deciding not to choose them.

F. Threats to Validity

Among threats to internal validity, since we placed time limits of 15 and 25 minutes for Tasks 1 and 2 respectively, several participants timed out for the more complex Task 2. However, we note that it would have taken participants several more steps to reach a preferred fix. Further, we presented the tasks to all participants through verbal descriptions of the objectives. This may have inevitably led to priming, wherein the participants' prompt formulations could have been influenced by our description of the bug. Lastly, we encouraged participants to think aloud as they go about the task, adding to the time taken to perform various steps. To address this, we only track time when participants actively work on the tasks.

Among threats to external validity, our task selection may not be representative of the breadth of real-world software bugs faced by developers. We tried to mitigate this by adapting three real-world bugs found on GitHub, varying in complexity, for our tasks. Since the participants did not write the code themselves, their use of AI-assistants for debugging self-authored code will add to the current set of findings. Another threat includes greater participant base having intermediate–advanced experience due to snowball sampling. We attempted to recruit a balanced proportion of beginner participants to understand their preferences and identify any additional barriers they might face in using the AI-assistants. Our study also does not consider long-term usage and potential habituation effects of using an AI-assistant like ROBIN for beginner participants.

VI. DISCUSSION

A. Personalization based on developer expertise

Developers may have different levels of experience with debugging strategies [1], [5], [26] or debugging tools (e.g., breakpoints, watch expressions, etc.) [6]—less experienced developers may not even know of their existence or underutilize them [7]. Our study participants belonged to diverse expertise levels (1–21 years), having varying expectations from debugging AI assistants. *Beginner* and *intermediate* users often relied on the understanding of the exception and hypothesis provided by the assistant; however, for an *expert*, both these were often clear from initial exploration of the exception. *Beginners* in our study required extensive hand-holding for localizing bugs, whereas *experts* could achieve the same outcomes without step-by-step instructions for routine tasks (like setting breakpoints and code stepping). Section V-D describes how beginners in the study often perceived ROBIN as an educative tool, comparing the experience to receiving guidance from senior team members. As a next step, we aim to explore ways to align ROBIN's responses with developer experience and familiarity with the code base.

B. Need for deeper integration with IDE

Participants also expressed a need for both ROBIN and the baseline assistant to have greater awareness of the source code to avoid copy-pasting code into the chat. In particular, expert developers desired a deeper integration within the Visual Studio IDE, such that ROBIN could automatically perform UI-actions like setting breakpoints and stepping through the code on their behalf. P6 said "if it already knows what the next step is, why doesn't it automatically perform it, and then point me to the conclusion?" Further, enabling AI-assistants to perform visible UI-actions may also facilitate learning experiences for beginners like P12, who expressed that, *"watching a video tutorial is better than reading chat responses to figure out how to use debugger features in Visual Studio because there are so many menus and elements in this IDE."*

C. Towards effective human-AI debugging

"All problems lie in absence of a good conversation" – John Niland — This holds true not only for human-human interactions but human-AI interactions as well. Effective communication often guided by Grice's maxims of ***quantity***, ***quantity***, ***relevance*** and ***manner*** [18]. Here, we reflect on the study insights in this context.

ROBIN's behaviour in coming back to the user for clarifications or additional information, rather than providing a *sub-optimal fix*, embodies the ***maxim of quality***, in contrast with the baseline's tendency to speculate and act on insufficient information (Section V-C). Towards the ***maxim of quantity***, ROBIN picks the Responder agent to address the warm-up task while picking the Collaborative agent for more involved tasks. This demonstrates ROBIN's adaptability to debugging scenarios with varying severity, unlike the baseline, which tends to always provide a single-turn response. Some participants pointed out differences in the quality of follow-ups as the AI-assistants were swapped across tasks (Section V-E). In conjunction with the 3.5x higher success rate with ROBIN, this showcases the importance of aligned follow-ups in preventing conversation derailments, enforcing the ***maxim of relevance***.

VII. RELATED WORK

Automated debugging has a rich history driven by rules encoded by human reasoning over syntactic and static analysis based signals. Understanding the bug and localization is often the application of a bunch of strategies that literature has been rich in [5], [26]–[28]. Often these strategies are good for a certain category of bugs, but not one-size fits; thus, different strategies can drive faster resolution in different cases. There have been attempts around strategizing the entire debugging life-cycle, introducing concepts like hypothesis debugging [29], [30], delta debugging, and program slicing [28].

The practical application of these strategies often proves cumbersome, as developers' debugging processes need to blend multiple methodologies [31], resulting in mechanical and infeasible implementations of individual strategies in real-world scenarios. ROBIN attempts to incorporate the essence of some of these strategies through the *understand-localize-fix* flow while narrowing down towards the fault by requesting specific pieces of additional context and clarifications from the user and providing clear instructions, allowing them not to lose focus [32], [33]. This further reduces debugging efforts, which often result in redundant sessions of Edit-Runs, particularly in codebases where the developer lacks awareness of third-party APIs and different pieces of scattered code [34]. Alaboudi et al. [1] emphasize the inadequacies of traditional debugging methods, advocating for a comprehensive approach beyond mere fault localization. ROBIN aligns with this philosophy by incorporating hypothesis-based debugging into its *investigate & respond* workflow [30]. Similar to scientific debugging [35], ROBIN leverages (LLMs) to propose hypotheses and reasoning towards the issue. However, it depends on environmental cues and the developer to accept or discard the current hypothesis.

Long before the advent of LLMs and neural debugging approaches, the vision for a "Programmer's Apprentice" was proposed with the aim of developing conversational agents for aiding software development tasks [36]. With the emergence of LLMs, this vision has become a practical reality. LLMs are now increasingly employed for code generation within auto-completion interfaces [16] and natural language explanations in chat-based interfaces. Recent work on "Programmer's Assistant" facilitates meaningful conversations with LLMs for software tasks by leveraging code contexts and maintaining transcripts of prior conversations [37]. Developers have exhibited favourable attitudes towards AI tools like GitHub Copilot for their utility in everyday programming tasks, despite not necessarily enhancing task completion or success rates [38]. Recent bug fix benchmark performances indicate LLMs hold considerable potential in debugging. In their studies, Prenner et al. [39], Sobania et al. [40] utilized OpenAI's CodeX [10], and ChatGPT [8] respectively to debug QuixBugs [41] with improvements over state-of-the-art.

The recent rise of agent frameworks allows for creating more complex software specific workflows [23], [42], [43]. Inspired by these frameworks, ROBIN breaks down debugging tasks into a conversational workflow. Each state within our workflow represents a conversational agent, fostering a richer and extendable collaboration between developers and ROBIN.

In parallel, software comprehension, an important subtask for automated software engineering tools has also seen improvements, from using longer contexts for LLMs [44], to improved code retrieval strategies [45], [46] and code execution feedback [47]–[49]. Previously, before LLMs could reliably work with code, Li et al. [50] used a human-in-the-loop query-feedback procedure to indicate the validity of variable values to facilitate bug localization and reported faster debugging fixes over traditional debugging methods. Recently, automated software engineering tools, such as ChatDBG [45], AutoCodeRover [46], and Devin [51], have emerged as notable advancements in solving issues and bugs in codebases in a completely automated manner. We believe in the supervised assistance approach to strike a balance between automation and manual intervention for realistic deployment.

VIII. LIMITATIONS

User Responses. Our current design places users as first-class citizens, with their responses significantly steering the conversation. However, this approach inherently assumes user proficiency in providing systematic and standard responses. In future, we could potentially explore the development of guidelines that help users towards more structured responses, thereby enhancing the effectiveness of collaboration.

Dependence on the backing LLM. ROBIN employs LLMs to power reasoning and generation capabilities of the agents and functions described in the Sec III-B. The experience and effectiveness can change significantly with any changes to the underlying model based on its ability in various dimensions.

Automated Context Retrieval. Can_Automate_Context() function defaults to false beyond the limited scope of the Context Retrieval agent (described in III-A). Future improvements to ROBIN will include support for wider automated on-demand retrieval. This shall abstract away the burden of fetching and sharing additional context on the developer end, reducing their cognitive load to focus on hypothesis and decision making.

IX. CONCLUSION AND FUTURE WORK

In this paper we introduce ROBIN, an *investigate & respond* agentic workflow towards pragmatic and collaborative debugging within IDEs. In designing ROBIN, we explore design principles to keep the human-in-the-loop through the use of IDE and debugger features, approaching a balance between complete automation and manual exploration, by shifting towards effective conversational traits. Through a within-subjects study with 16 industry professionals, we observe (3.5 times) higher bug resolution rates and increased engagement with debugger tools, owing to the drift towards conducive conversation patterns for debugging. ROBIN enables guided localization, while participants engage in more streamlined debugging experiences following the *understand–localize–fix* workflow. We hope to improve ROBIN, envisioning an increasingly cohesive debugging experience for developers. As discussed in Section VI, we aim to personalize ROBIN based on developer expertise, and enable deeper IDE integration through automated UI actions. We plan to investigate domain-specific interaction patterns for project scaffolding, code migration and re-platforming as potential long-running and time intensive software engineering tasks, just like debugging.

ROBIN, now a part of GitHub Copilot Chat, is a checkpoint in an evolving journey which allows for us to re-calibrate and understand developer needs and expectations towards intelligent debugging assistance.

References

[1] A. Alaboudi and T. D. LaToza, "Rethinking Debugging and Debuggers," *12th Annual Workshop at the Intersection of PL and HCI*, 6 2022. [Online]. Available: https://kilthub.cmu.edu/articles/conference_contribution/Rethinking_Debugging_and_Debuggers/19852006

[2] D. H. O'Dell, "The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills." *Queue*, vol. 15, no. 1, p. 71–90, feb 2017. [Online]. Available: https://doi.org/10.1145/3055301.3068754

[3] Microsoft, "Visual studio," 2023. [Online]. Available: https://visualstudio.microsoft.com/

[4] J. Brains, "Intellij idea," 2023. [Online]. Available: https://www.jetbrains.com/idea/

[5] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020737385800547

[6] M. Ko, D. B. Bose, H. A. Chowdhury, M. Seyam, and C. Brown, "Exploring the barriers and factors that influence debugger usage for students," in *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2023, pp. 168–172.

[7] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681.

[8] OpenAI, "Chatgpt," 2022. [Online]. Available: https://chat.openai.com

[9] GitHub, "Github copilot chat," 2023. [Online]. Available: https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[11] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2023.

[12] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2023.

[13] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," in *NeurIPS*, 2022.

[14] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint*, 2023.

[15] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[16] P. Gupta, A. Khare, Y. Bajpai, S. Chakraborty, S. Gulwani, A. Kanade, A. Radhakrishna, G. Soares, and A. Tiwari, "Grace: Language models meet code edits," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1483–1495. [Online]. Available: https://doi.org/10.1145/3611643.3616253

[17] R. Dale, R. Fusaroli, N. D. Duran, and D. C. Richardson, "Chapter two - the self-organization of human interaction," in *The Self-Organization of Human Interaction*, ser. Psychology of Learning and Motivation, B. H. Ross, Ed. Academic Press, 2013, vol. 59, pp. 43–95. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124071872000022

[18] I. Shatz, "Grice's maxims of conversation: The principles of effective communication." [Online]. Available: https://effectiviology.com/principles-of-effective-communication/

[19] M. Arab, T. D. LaToza, J. Liang, and A. J. Ko, "An exploratory study of sharing strategic programming knowledge," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491102.3502070

[20] T. D. LaToza, M. Arab, D. Loksa, and A. J. Ko, "Explicit programming strategies," *Empirical Software Engineering*, vol. 25, no. 4, p. 2416–2449, Mar. 2020. [Online]. Available: http://dx.doi.org/10.1007/s10664-020-09810-1

[21] M. Arab, T. D. LaToza, J. Liang, and A. J. Ko, "An exploratory study of sharing strategic programming knowledge," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491102.3502070

[22] Microsoft, "Bing copilot," 2023. [Online]. Available: https://www.bing.com/chat

[23] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation," 2023.

[24] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[25] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.

[26] M. Arab, T. D. LaToza, J. Liang, and A. J. Ko, "An exploratory study of sharing strategic programming knowledge," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491102.3502070

[27] J. T. Liang, M. Arab, M. Ko, A. J. Ko, and T. D. LaToza, "A qualitative study on the implementation design decisions of developers," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 435–447. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00047

[28] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, p. 446–452, jul 1982. [Online]. Available: https://doi.org/10.1145/358557.358577

[29] Q. Ma, H. Shen, K. Koedinger, and T. Wu, "How to teach programming in the ai era? using llms as a teachable agent for debugging," 2024.

[30] A. Alaboudi and T. D. LaToza, "Using hypotheses as a debugging aid," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2020, pp. 1–9.

[31] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 199–209. [Online]. Available: https://doi.org/10.1145/2001420.2001445

[32] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 301–310. [Online]. Available: https://doi.org/10.1145/1368088.1368130

[33] ——, "Finding causes of program output with the java whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1569–1578. [Online]. Available: https://doi.org/10.1145/1518701.1518942

[34] A. Alaboudi and T. D. LaToza, "Edit - run behavior in programming and debugging," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2021, pp. 1–10. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/VL/HCC51201.2021.9576170

[35] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable automated debugging via large language model-driven scientific debugging," 2023.

[36] C. Rich and R. C. Waters, *The programmer's apprentice*. New York, NY, USA: Association for Computing Machinery, 1990.

[37] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The programmer's assistant: Conversational interaction with a large language model for software development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, ser. IUI '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 491–514. [Online]. Available: https://doi.org/10.1145/3581641.3584037

[38] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491101.3519665

[39] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, ser. APR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 69–75. [Online]. Available: https://doi.org/10.1145/3524459.3527351

[40] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 23–30. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/APR59189.2023.00012

[41] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 55–56. [Online]. Available: https://doi.org/10.1145/3135932.3135941

[42] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J.-R. Wen, "A survey on large language model based autonomous agents," 2023.

[43] W. Chen, Y. Su, J. Zuo, C. Yang, C. Yuan, C.-M. Chan, H. Yu, Y. Lu, Y.-H. Hung, C. Qian, Y. Qin, X. Cong, R. Xie, Z. Liu, M. Sun, and J. Zhou, "Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors," 2023.

[44] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259360665

[45] K. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, "Chatdbg: An ai-powered debugging assistant," 2024.

[46] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," 2024.

[47] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," 2023.

[48] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, Y. Wu, Z. Liu, and M. Sun, "Debugbench: Evaluating debugging capability of large language models," 2024.

[49] L. Zhong, Z. Wang, and J. Shang, "Ldb: A large language model debugger via verifying runtime execution step-by-step," 2024.

[50] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 82–92. [Online]. Available: https://doi.org/10.1145/3180155.3180242

[51] C. Labs, "Devin, ai software engineer," 2024. [Online]. Available: https://www.cognition-labs.com/introducing-devin