

When Agents Fail: A Comprehensive Study of Bugs in LLM Agents with Automated Labeling

NIFUL ISLAM, Oakland University, USA

RAGIB SHAHARIAR AYON, Texas State University, USA

DEEPAK GEORGE THOMAS, Tulane University, USA

SHIBBIR AHMED, Texas State University, USA

MOHAMMAD WARDAT, Oakland University, USA

Large Language Models (LLMs) have revolutionized intelligent application development. While standalone LLMs cannot perform any actions, LLM agents address the limitation by integrating tools. However, debugging LLM agents is difficult and costly as the field is still in its early stage and the community is underdeveloped. To understand the bugs encountered during agent development, we present the first comprehensive study of bug types, root causes, and effects in LLM agent-based software. We collected and analyzed 1,187 bug-related posts and code snippets from Stack Overflow, GitHub, and Hugging Face forums, focused on LLM agents built with seven widely used LLM frameworks as well as custom implementations. For a deeper analysis, we have also studied the component where the bug occurred, along with the programming language and framework. This study also investigates the feasibility of automating bug identification. For that, we have built a ReAct agent named *BugReAct*, equipped with adequate external tools to determine whether it can detect and annotate the bugs in our dataset. According to our study, we found that *BugReAct* equipped with Gemini 2.5 Flash achieved a remarkable performance in annotating bug characteristics with an average cost of 0.01 USD per post/code snippet.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Large Language Model, AI Agents, Bug Study

-:

Niful Islam, Ragib Shahariar Ayon, Deepak George Thomas, Shibir Ahmed, and Mohammad Wardat. 2026. When Agents Fail: A Comprehensive Study of Bugs in LLM Agents with Automated Labeling. 1, 1 (January 2026), 31 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

LLM agents are autonomous systems that assist users with tasks, while having human-like abilities such as autonomy, multimodal perception, and reasoning [32]. Due to the recent advancement in large language models, more developers and users have been working on creating and utilizing these systems [8, 27, 35]. They have applications in safety-critical industries such as security [30], banking [34], healthcare [31], and robotics [29]. Moreover, LLM-based agents have the potential to be applied in complete software development [50]. Like any traditional or deep learning-based software,

Authors' Contact Information: Niful Islam, Oakland University, Rochester Hills, USA, islam3@oakland.edu; Ragib Shahariar Ayon, Texas State University, San Marcos, USA, ipd21@txstate.edu; Deepak George Thomas, Tulane University, New Orleans, USA, dthomas23@tulane.edu; Shibir Ahmed, Texas State University, San Marcos, USA, shibir@txstate.edu; Mohammad Wardat, Oakland University, Rochester Hills, USA, wardat@oakland.edu.

© 2026 Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

-

LLM-based agents also have bugs. Analyzing these bug types, their root causes, and effects across forums like Stack Overflow, GitHub, and Hugging Face is essential for understanding their characteristics and helping the software engineering community mitigate them. Moreover, it would be challenging for developers to identify solutions to bugs due to the recent proliferation of LLM Agent frameworks. Therefore, it is imperative that researchers develop approaches to identify and resolve bugs in these systems.

While LLM agents are also considered a type of software, they differ from traditional systems in three major ways. Firstly, the agents rely on LLMs as their key component, which is inherently black-box in nature [46]. Therefore, debugging the system is more complicated. Secondly, unlike traditional machine learning (ML) or deep learning (DL) systems, LLM agents are inherently nondeterministic. While fixed-parameter ML/DL solutions (e.g., KNN or CNNs) provide consistent outputs for a fixed input, LLMs can produce different outputs for the same input when the temperature value is set to a moderate or high level [59]. Therefore, testing these systems becomes more challenging because certain bugs cannot be reproduced consistently. Lastly, the LLM agents community is still relatively new. For example, LangChain, the most widely used framework for building LLM agents, was released at the end of 2022, and many additional libraries with new features continue to appear frequently [45]. Consequently, when a developer encounters issues while building a system, there are fewer experienced developers available on community platforms to provide support. The causes necessitate an extensive study of bugs in LLM agents.

Recent work has examined defects in LLM agents and proposed taxonomies of agent-based bugs and errors [54, 62]. However, one of these efforts focuses only on defect types, while the other primarily analyzes bugs in the frameworks used to build LLM agents rather than in the agents themselves. Additionally, while LLM agents have demonstrated exceptional abilities across multiple fields [23, 48, 104], their effectiveness drops significantly when tasked with fixing issues within an agentic system. The best-performing agent achieves only 4% accuracy [62], highlighting its currently limited self-healing capability.

To fill this gap, we have conducted the first extensive study of bugs in LLM agents across different platforms. Our empirical study primarily follows the approach utilized by Islam *et al.* [39] for characterizing bugs in deep learning models. We focus on characterizing the bug's type, root cause, and effects. Also, it identifies the component (planning, agent core, memory, and tools) of the LLM agent that gets most affected by bugs. Furthermore, we study the distribution of bug types, root causes, and frameworks. Finally, we investigate how bugs in AI agents have been changing over time. Additionally, we built a ReAct agent that can automatically label bugs in LLM agents while scraping data from various AI Agent repositories. One of the novelties of our agent is that it extracts data from relevant sites online. We claim that our benchmark and automated AI agent would assist developers in systematically resolving bugs in this area.

There were various challenges associated with building the empirical study. Due to recent developments in the field of LLM Agents, conventional [36, 39] data sources such as Stack Overflow and GitHub will not provide the same quantity or quality of information. Therefore, we searched for relevant forums such as Stack Overflow and GitHub, and we also studied Hugging Face forums for collecting datasets. Furthermore, while developing *BugReAct*, we addressed potential biases in the underlying foundation models by using three different models, ensuring our findings were not overly dependent on a single source of data. Finally, a challenge common to developing the empirical study and approach was that, since the field is evolving fast, the bugs encountered were new. The annotators (both human and non-human) faced a non-trivial challenge in understanding these bugs. We release a dataset of bugs to assist developers in navigating this dynamic field.

Contributions. This paper makes the following main contributions:

- **Novelty:** To the best of our knowledge, we are the first to systematically explore LLM-based agent bug characteristics across different frameworks. We leverage this study and design *BugReAct* to identify bug types, root causes, and effects, while also providing rationales for these labels. This allows developers to better grasp the nature of bugs in their programs, supporting clearer understanding and more efficient diagnosis.
- **Empirical Study:** We conduct an empirical study to build a taxonomy of LLM-based agent bugs across seven popular frameworks, aiming to understand the common mistakes developers make when building agents. By identifying and categorizing bugs reported in forums such as Stack Overflow, GitHub, and Hugging Face, we provide an in-depth discussion of key findings and their implications for future research and improvements.
- **Agent-Based Approach:** We propose *BugReAct*, an LLM-based agent system designed to annotate posts containing different types of bugs automatically. Particularly, *BugReAct* dynamically analyzes posts, invokes external tools to gather additional information, applies reasoning to characterize bugs, and provides precise annotations, effectively addressing challenges such as handling various program designs, multiple programming languages and frameworks, and online forums. Our study shows the proposed solution achieves substantially superior results relative to the baseline methods.
- **Evaluation:** We extensively evaluate *BugReAct* on 1,187 posts/code snippets using three LLMs. *BugReAct* succeeds in characterizing bugs comparable to manual human labeling, demonstrating its usefulness in automated bug annotation. Moreover, our research has the potential to support downstream program analysis of LLM-based agent systems, such as bug localization and repair, and to guide future research.

The rest of the article is organized as follows. Section 2 presents the background and compares related work in the field. Section 3 then describes the complete empirical study, outlining the data collection process and reporting the findings from the dataset, along with actionable insights for developers, library maintainers, and researchers. Section 4 details *BugReAct*, which is used for labeling bugs, followed by the results obtained from *BugReAct* on the collected dataset in Section 5. Section 6 discusses threats to validity, and Section 7 concludes the paper.

2 Background and Related Work

2.1 Background

Agent. Recent papers have acknowledged that a standard definition of agents is yet to be found. Hu *et al.* [28] referred to it as agentic systems consisting of foundation models as their component. Furthermore, they said that these systems execute objectives using the following: planning, tools, and iterative computation. Whereas Huang *et al.* [32] referred to it as a digital object that can understand, think, and act with human-like independence. Furthermore, Huang *et al.* developed a taxonomy to describe the types of AI agents available; Reactive: agents that map queries directly to outputs, Deliberative: agents that can reason and plan using internal world models, Hybrid: agents that have specific layers for reactive and deliberate behavior, Learning: agents which incorporate feedback and become better with time, Cognitive: agents achieving human level performance in reasoning and problem solving tasks, Collaborative and Competitive: agents which work with others, cooperating or competing based on task specifications, Vertical: domain specialized agents. Also, they attempt to generalize the agents into a layered architecture consisting of seven components: Foundation Models, Data Operations, Agent Frameworks, Deployment and Infrastructure, Evaluation and Observability, Security and Compliance, and Agent Ecosystem.

AI-Agent Bug Analysis. While AI systems have been present since decades, Huang *et al.* [33] claimed that the rise of recent AI agents can be attributed to the following factors: increased compute, innovations in NLP and other

ML fields, Big Data, and progress in interdisciplinary fields such as cognitive science, neurobiology, etc. The authors claimed that these agents will no longer be considered tools but will instead be intellectual collaborators. Once we consider that Agents are used in various safety-critical domains, understanding their failure characteristics becomes necessary.

2.2 Related Work

Empirical Study on Bugs. There has been a plethora of studies done on investigating bugs in a variety of Machine Learning systems. Thung *et al.* [90] claimed to be among the first to study this field. They targeted the following areas that used the tools mentioned: data mining (Apache Mahout), natural language processing (OpenNLP), and information retrieval (Lucene), and labeled 500 artifacts. Furthermore, a notable work on studying deep learning (DL) systems is that of Humbatova *et al.* [36], wherein they built a fault taxonomy. They analyzed GitHub repositories by studying issues, pull requests, and commits. They also studied Stack Overflow posts. Their investigation looked into programs that used top frameworks for DL: Tensorflow, Keras, and PyTorch. This process led to the generation of 1059 artefacts. In addition, they performed interviews and surveys to strengthen and validate their taxonomy. Another set of works close to ours is by Jahan *et al.* [40] and Thomas *et al.* [89], who created fault taxonomies for attention-based and deep reinforcement learning (DRL) models. Since the foundation models (specifically LLMs) for Agents comprise attention-based architectures, this study can be useful in understanding the bugs in the LLMs. Jahan *et al.* investigated 555 real-world faults by mining GitHub, StackOverflow, and Hugging Face. Also, Reinforcement Learning is often used to align agents with user objectives. Thomas *et al.* investigated bugs in DRL models by mining GitHub and StackExchange and studying 2,787 posts [58]. Rahardja *et al.* [62] developed a taxonomy of issues found in LLM agents by analyzing 201 GitHub issues across 18 repositories. They also evaluated state-of-the-art software engineering agents to measure their effectiveness in agentic settings. Their results show that existing LLM agents perform far worse on agentic tasks than on traditional software tasks: for example, Agentless [96] resolved only 3.33% of issues in agentic systems compared to 32.0% in traditional systems. Similar trends were observed for AutoCodeRover [103] (1.33% compared to 30.67%) and SWE-agent [99] (0.67% compared to 18.33%), all powered by GPT-4o. These findings highlight the need for a deeper analysis of bugs in LLM agents. Compared to the taxonomy presented in the study by Rahardja *et al.*, which draws heavily from 11 repositories that function as frameworks or tools for building agents, our work focuses on bugs that originate from the agents themselves rather than from the underlying frameworks. Pan *et al.* [60] analyzed more than 150 code execution traces to investigate failures in Multi Agent Systems (MAS). The researchers created a taxonomy of failures and identified the stage at which each occurred, including pre-execution, execution, and post-execution. The study found that most failures began in the pre-execution stage and continued into the execution stage. The authors also developed an LLM-based method that automatically annotated failures in MAS. Overall, while some studies examine bugs across several domains, there is still a gap in understanding bugs in the development of agentic systems. This gap exists because agentic systems differ from traditional softwares in three key aspects which are their black box nature, their dynamic behavior, and the relatively early stage of their community. This work fills this gap by offering an extensive study of bugs observed in large language model agents and provides practical insights for developers, library maintainers, and researchers that can support the growth of this emerging community. Moreover, the above work done in the above section doesn't delve into understanding bug characteristics. Our work mainly focuses on understanding bug characteristics of agentic systems.

Bugs Characteristics. Islam *et al.* [39] studied DL bugs with the objective of understanding the characteristics of bugs in DL systems. They extracted posts from StackOverflow and GitHub commits related to popular libraries for DL. Their investigation of 3216 artifacts resulted in identifying severe bug types, and also root causes and correlation of bugs among deep learning libraries. Jiang *et al.* [42] claimed to be the first group studying the characteristics of LLM library bugs. They looked at 313 commits from Hugging Face Transformers and vLLMs and arrived at five bug symptoms and 14 root causes. Han *et al.* [24] also performed a similar study on foundation language models by investigating 469 bugs obtained through mining GitHub and StackOverflow. The FLMs they focused on were T5, LLaMa, OPT, GPT-NeoX, GLM, Pythia, and GPT-J. They focused on identifying root causes, fixing patterns, and symptoms. To the best of our ability, we don't know of any prior work that deals with bug characteristics of agentic systems. Therefore, we leverage the above work as inspiration to investigate it.

Agentic AI for SE and SE for Agentic AI. Deshpande *et al.* [13] developed a taxonomy of agent errors. Furthermore, they investigated whether LLMs could diagnose errors in agentic systems. The difference between their approach and ours is that the researchers applied LLMs on agent traces while our approach takes user-based queries as input. While traces may provide more information, collecting them is a non-trivial task. More importantly, feeding entire traces to the LLMs would consume a considerable amount of tokens. Liu *et al.* [50] studied 106 papers to learn about LLM-based agents that are applied to Software Engineering. They found papers wherein Agents addressed the following tasks: "Requirements Engineering", "Code Generation", "Static Checking", "Testing", "Fault Localization", and "Repair". Moreover, researchers have compiled and analyzed large-scale interaction trajectories of LLM-based agents in software repair tasks [9]. Xia *et al.* [97] developed an LLM agent named Agentless for bug localization, repair, and patch validation, which achieved the highest performance among all open-source software agents on the SWE-bench Lite benchmark while maintaining low cost. Ning *et al.* [54] mined Stack Overflow and extracted 6854 posts related to LLM Agents, in order to investigate defects. They grouped the defects into eight categories. Furthermore, they developed an approach to detect these defects using static analysis. This study demonstrated the relevance of AI Agents in software engineering tasks. Furthermore, it also shows how software engineering concepts such as static analysis might be applied to improve agentic systems.

Agent-Based Bug Identification for Agents. Our work follows a similar approach to Cemri *et al.*, [11], who claim to be the first to investigate failure modes in Multi-Agent LLMs (MAS). In addition to developing a taxonomy, they investigated training an LLM to identify the failure modes from their taxonomy. Their work did not involve mining Stack Overflow or GitHub for data collection. Rather, they obtained traces from major benchmarks such as SWE-Bench-Lite [43]. AGDebugger [16] addresses the challenges developers face in debugging multi-agent systems by providing interactive debugging features, real-time steering of agent teams, and visualization of conversation flows. Following earlier studies, we built and tested the ability of LLM agents to identify and label bugs in other LLM agents. In this way, we examined their potential for self-healing. Although our architecture differs from previous work, we developed a ReAct agent equipped with tools that extract information from external sources, which makes our solution more dynamic.

3 Empirical Study

As presented in Figure 1, the proposed approach has two significant steps. The steps include collecting and annotating data for the empirical study and building an LLM agent to check if an agent can classify the bugs automatically. This section discusses the empirical study conducted to assess the bugs that occurred during the developers' design of LLM

Table 1. Descriptions of Bug Types

Bug Type	Description
Logic Bug (LB)	The bug indicates a lack of logical understanding in the pipeline. This bug includes using a function that does not fit the specific task, missing or incorrectly implementing any code segment, or the absence of proper guarding conditions [39] (e.g., [69, 75, 79, 81]).
Configuration Bug (CB)	This bug indicates any configuration errors, including parameter misconfiguration or environment misconfiguration [25] (e.g., [72, 82, 83]).
Initialization Bug (IB)	Any variable/function may require proper initialization before use. Not initializing the variables/functions correctly before using them or initializing them wrongly (e.g., inside a loop in the wrong way) will cause this error [102] (e.g., [21]).
Argument Bug (AB)	This bug indicates issues with the API signature, such as the API expecting specific arguments while the user provides arguments in an incorrect format, or passes extra or missing parameters. It is worth noting that passing an incorrect value to a function that matches its signature will not trigger this bug [64] (e.g., [76]).
Parsing Bug (PRB)	This LLM-agent-specific bug occurs while parsing the LLM output. Often, the LLM-generated output format does not align with the parser's structure or the user's expectations, triggering the bug. It mostly occurs in the parser of the LLM agent (e.g., [70]).
Prompting Bug (PPB)	This bug is related to the prompts in the LLM. This includes missing prompt variables/components or invoking the LLM with incorrect instructions (e.g., [71, 78]).
API Bug (APIB)	This bug is related to the APIs or libraries used to build the agent. Causes include dependency conflicts, incorrect version usage, bugs within the libraries themselves, or trying to use a library without installing it [39] (e.g., [20, 87]).
Reference Bug (RB)	In agentic AI, often different libraries implement modules with similar names. This bug occurs when the user refers to a different, deprecated, or missing module from a specific library. This bug mostly occurs in the import stage (e.g., [74]).
Availability Bug (AVB)	This bug usually comes from the API or server side, indicating that a particular language model or service is unavailable—either because of a server issue or because the feature has not been released yet (e.g., [77]).
Model Bug (MB)	This bug is related to the LLM itself and occurs when a user requests a task the LLM cannot perform, such as asking a chat model to generate an image or using a non-functional model to produce a function call (e.g., [68]).
Resource Limitation Bug (RLB)	This bug is related to the user's local system and its resource usage. It may occur when attempting to run a large LLM with insufficient system resources or limited credits (e.g., [85]).

agents for a specific task. Our research team has manually analyzed 4,341 bug-related posts/code snippets before finally annotating 1,187 instances for the study. The empirical study answers the following research questions.

- **RQ1 (Bug Characterization):** What are the most common bug types, root causes, and effects in developing LLM agents?
- **RQ2 (Bug-Prone Components):** Which component of the LLM agent is more prone to bugs?
- **RQ3 (Bug Mapping):** What is the distribution across bug types, root causes, and frameworks used?

Table 2. Descriptions of Root Causes and Their Subclasses

Root Cause	Subclass	Definition
API Misuse (AM)	Wrong API Context	The API is used in an inappropriate situation, or there is a misunderstanding of its intended purpose. For example, using a translation API to extract sentiment [18] (e.g., [20, 21, 75]).
	Invalid API Arguments	Incorrect types or number of arguments are passed to the API. For instance, passing a string where a JSON object is expected [36] (e.g., [72]).
Incorrect or Missing Parameter (IMP)	Incorrect Value	A valid parameter is passed, but its value is logically wrong, like using a learning rate of 10 instead of 0.01 [61] (e.g., [76, 83]).
	Missing Value	One or more optional parameters are omitted entirely, often leading to unexpected default behavior [61] (e.g., [73]).
Incorrect Data Format (IDF)	Input Data Format Error	An error that occurs when the data from an external resource provided to the LLM is not in the expected type, structure, or schema. For example, the input text contains a stop sequence that causes the LLM to terminate its response early (e.g., [80]).
	Output Data Format Error	The expected output format from the LLM is not met. For example, when prompted for structured output, the LLM-generated response may miss required keys or deviate from the format, causing the parser to fail when processing it (e.g., [69]).
Incorrect or Missing Control Flow (IMCF)	Missing Flow	A required logical code segment is not implemented, such as a missing if condition or function call [41] (e.g., [74, 81]).
	Incorrect Flow	This occurs when the logic is present in the code but implemented incorrectly, such as using an if condition that checks the wrong variable [39] (e.g., [79]).
Incorrect Instruction (II)	Prompt Specification	Errors in constructing the prompt, such as ambiguous language, poor formatting, or missing instruction cues. It includes four issues in the prompt, including missing context, missing specifications, multiple contexts, and unclear instructions [15] (e.g., [71]).
	Prompt Orchestration	Logical flaws in the prompt, such as not passing variables correctly or providing a plain string when a JSON-formatted input is required (e.g., [78]).
API Limitation (AL)		Task failure due to limitations of the API or Library, such as unsupported capabilities or service downtime. Users typically cannot control this [49] (e.g., [77, 84]).
Component Mismatch (CM)		Incorrect usage or selection of components like LLM, tools, or memory modules, resulting in integration failure [92] (e.g., [68]).
Requirement Violation (RV)		Dependency conflicts or missing requirements, such as incompatible library versions or unmet prerequisites [10] (e.g., [87]).
Others		Issues not related to LLM agents, often caused by the user's machine, environment, or external components (e.g., [82, 85]).

- **RQ4 (Bug Evolution):** How do different types of bugs evolve in frequency over time?

3.1 Dataset

For the empirical study, we collected a large dataset and annotated it based on bug type, root cause, effect, the LLM agent component involved in bug occurrence, programming language, and development framework. Following previous works [36, 39], two sources were employed in this study, namely the Stack Overflow forum and GitHub, to collect the datasets. In order to test the ability of the LLM agent, we have collected an additional dataset from Hugging Face forums. The process of collecting data from each site is illustrated below.

3.1.1 Stack Overflow. We selected Stack Overflow, one of the most popular Q&A sites for developers [53], to collect our dataset. As illustrated in Figure 1, the data collection process consists of three main stages. In the first stage, keyword

Table 3. Descriptions of Effects

Effects	Description
Crash	The program throws an error and stops working [39] (e.g., [80]).
Incorrect Output (IO)	LLM generates a complete output, but it does not align with the expected output [39] (e.g., [20, 68, 70, 78]).
Empty Response (ER)	The system does not generate an output at all [39]. (e.g., [71, 76])
Output Dump (OD)	This effect indicates that the system generated the entire output at once instead of streaming it gradually (e.g., [84]).
Stateless Interaction (SI)	The agent does not remember the past conversations and answers only the current question without understanding the full context (e.g., [73]).
Partial Output (PO)	The LLM generates incomplete or truncated output [39] (e.g., [72]).
Tool Ignored (TI)	The effect indicates that the system does not invoke tool(s) during the process [95] (e.g., [69]).
Slow Output (SO)	The output is generated slowly [88](e.g., [77, 81, 85]).
Warning	This means the program does not terminate but instead issues a warning [88] (e.g., [79]).
Hang	The system ceases to respond and requires human intervention to restore functionality [39] (e.g., [82]).
Indeterminate Loop (IL)	The system runs into a loop indefinitely [39](e.g., [86]).
Resource Overuse (RO)	This effect indicates that the system consumed high resources like RAM [88] (e.g., [87]).
Silent Fail (SF)	This effect illustrates that the agent has failed to perform a task, but did not provide any log that the task failed [51] (e.g., [83]).
Unknown	The post does not mention the effect of the problem [88], (e.g., [21, 74]).

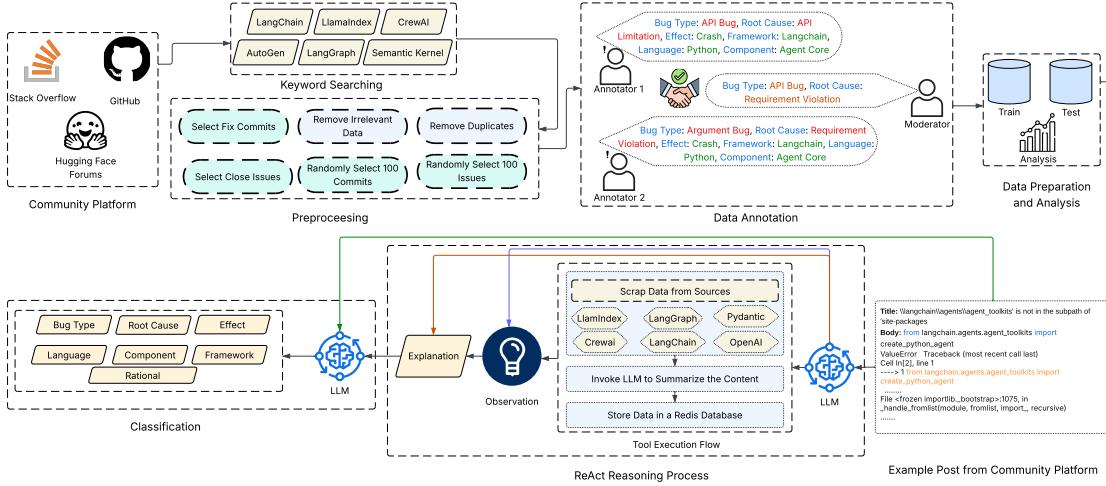


Fig. 1. Workflow of the data-collection and labeling.

searching, we gathered posts related to LLM agents by searching with keywords such as LLM agent, LangChain, LlamaIndex, Semantic Kernel, CrewAI, AutoGen, and OpenAI, collecting a total of 2656 posts. In the second stage, we manually evaluated each post, eliminating those that were irrelevant or consisted only of questions, and posts with insufficient or no code, and removed duplicate entries. We retained only those posts that contained sufficient information about the problem. Finally, we selected 870 posts relevant to our study. In the third step, two PhD students independently analyzed each post and assigned labels. The details of data annotation are mentioned in Section 3.1.4.

3.1.2 Github. We have collected data from two GitHub sources, namely GitHub commits and GitHub Issues, to verify the existence of bugs that we identified in the Stack Overflow dataset. For both GitHub commits and issues, we collected 1891 GitHub repositories utilizing the following libraries: LangChain, LangChain-js, LangGraph, LlamaIndex, CrewAI,

AutoGen, and Semantic Kernel. We then excluded repositories maintained by the original developers of these libraries to ensure our analysis focused exclusively on bugs in the implementation of agents, rather than bugs within the libraries themselves. The subsequent steps differ between GitHub commits and issues.

GitHub Commits: Once all the repositories are collected, we filtered the commits from these repositories, collecting only those explicitly marked as ‘fix’ commits (i.e., commits whose messages contained the keyword ‘fix’). In the second stage, following prior work [39], we randomly selected 100 commits from each library. Since a single GitHub repository can utilize multiple frameworks, we ensured that the randomly selected commits are mutually exclusive to avoid overlap. Each commit was analyzed and labeled according to the classification criteria established previously for the Stack Overflow dataset. As in the Stack Overflow analysis, two annotators independently labeled each commit, and agreement between annotators was evaluated at different stages of the labeling process. In total, we have collected 36, 16, 16, 19, 12, 20, 23 bugs from LangChain, LangChain-js, LangGraph, LlamaIndex, CrewAI, AutoGen, and Semantic Kernel, respectively.

GitHub Issues: To collect GitHub issues, we have iterated through the repositories and collected all the closed issues. After that, similar to GitHub commits, we randomly chose 100 non-overlapping issues, if a library had more than 100 issues. Subsequently, we filtered out the relevant instances and labeled them following the classification steps defined previously.

3.1.3 Hugging Face Forums. Since Hugging Face plays a critical role in building LLM agents, especially with open-source LLMs [38], the Hugging Face forums contain issues related to LLM agents. Therefore, we have selected posts from the forums by searching the keywords: LangChain, LangChain-js, LangGraph, LlamaIndex, CrewAI, AutoGen, and Semantic Kernel. Thereafter, we removed the duplicate posts, and the filtered relevant posts were selected for labeling. Since the dataset is used for evaluation purposes only, unlike Stack Overflow and GitHub, it was not labeled by two annotators. Instead, it was labeled independently by one author and later compared with the predictions of the agent.

3.1.4 Data Annotation. Unlike previous works [55], no predefined taxonomy exists to leverage for annotating the dataset. Therefore, we adopted similar methodologies applied in previous works [36, 54] for annotation, where for a given post, an author independently assigned a card along with a description for the label. Once all the cards were generated, the final label was defined using the open card sorting method [101]. The defined bug types, root causes, and effects are presented in Table 1, 2, and 3, respectively. Although our labeling process did not draw on an existing taxonomy, several of the resulting labels, nonetheless, coincide with categories identified in prior literature. Section 3.6 presents a comparative analysis of the newly defined labels alongside the existing ones, as well as their distributions in agentic and non-agentic systems. In addition to bug types, root causes, and effects, we have also presented the programming language and library used to build the agent. Furthermore, we have also presented the agent component that initiated the bug. To determine the components of LLM agents, we used the four components described in previous literature [54].

Once all the labels are defined, following a previous study [39], two PhD students independently annotated all four labels (i.e., bug type, root cause, effect, and component). For Stack Overflow, the programming language and development framework were identified using tags, while for GitHub, the data was collected based on the frameworks. Therefore, the programming language and framework labels were annotated collaboratively, in contrast to the other four labels, which were annotated independently. The agreement between two annotators was measured at 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100% intervals of the labeling process using Cohen’s Kappa coefficient [47]. The disagreement was addressed after each iteration. In situations where two annotators failed to come to an agreement,

Table 4. Inter-Rater Agreement Between Two Annotators

Data	Bug Type	Root Cause	Effect	Component
5%	0.3468	0.4484	0.7815	0.0000
10%	0.3772	0.2854	0.6625	0.0915
20%	0.4078	0.3521	0.7584	0.3076
30%	0.5612	0.4850	0.8323	0.5637
40%	0.9129	0.8964	1.0000	1.0000
50%	0.8891	0.9094	1.0000	1.0000
60%	0.9174	0.9610	1.0000	1.0000
70%	0.9855	0.9466	0.9366	0.9801
80%	0.9862	0.9871	1.0000	1.0000
90%	0.9861	0.9871	1.0000	1.0000
100%	0.9859	1.0000	0.9805	1.0000

a moderator with expertise in the field adjudicated. The agreement (measured using Cohen’s kappa) between two annotators at different iteration stages in labeling Stack Overflow posts is shared in Table 4. A higher kappa value (0 to 1) reflects stronger agreement [91].

According to the data, at the initial stage of labeling, there was slight agreement between the annotators, which increased to a moderate level across all components by 30% of the data. In the subsequent iterations, the annotators achieved perfect agreement across all labels for the Stack Overflow dataset. Due to the limited number of samples, the agreement for GitHub issues and commits was measured after the annotations were completed. Since the GitHub data was annotated after the Stack Overflow data, the level of disagreement was low. For GitHub issues, the kappa scores for bug type, root cause, and component were 0.9422, 0.7084, and 0.983, respectively, while the effect showed no disagreement. For GitHub commits, no disagreement was recorded. Since some posts may contain multiple bugs, data instances exhibiting multiple bug types or root causes were annotated multiple times. In total, 15 such instances were annotated more than once, resulting in 1,202 annotations. Lastly, since data collected from Hugging Face forums was only used for evaluation, one annotator labeled the instances, and the match was calculated between the human and the best LLM agent developed.

3.2 Most common bug type, root cause and effect (RQ1)

While bug distributions vary across different platforms, the majority of bug types, root causes, and effects are common to all platforms. The primary bug type in both Stack Overflow and GitHub (both commits and issues) is the Logic Bug, accounting for 22.7% and 30.2% of all bugs, respectively. Figure 2 presents the distribution of bug types across two platforms. The distribution of root causes and effects is available in Figure 3 and 4, respectively.

Although logic bugs occur in traditional software engineering systems, existing tools for addressing logic bugs like Pylint [3] cannot be directly applied to LLM agents due to their different reasoning mechanisms. The tools require further adaptation before integrating them into the LLM agent environment. While some bugs, such as parsing bugs and model bugs, are less common in GitHub, they are more frequently observed in Stack Overflow. This can be attributed to the fact that in GitHub, particularly in commits, the user is working with an already built system that has passed through at least one level of development. In contrast, parsing and model bugs typically arise during the initial implementation phase.

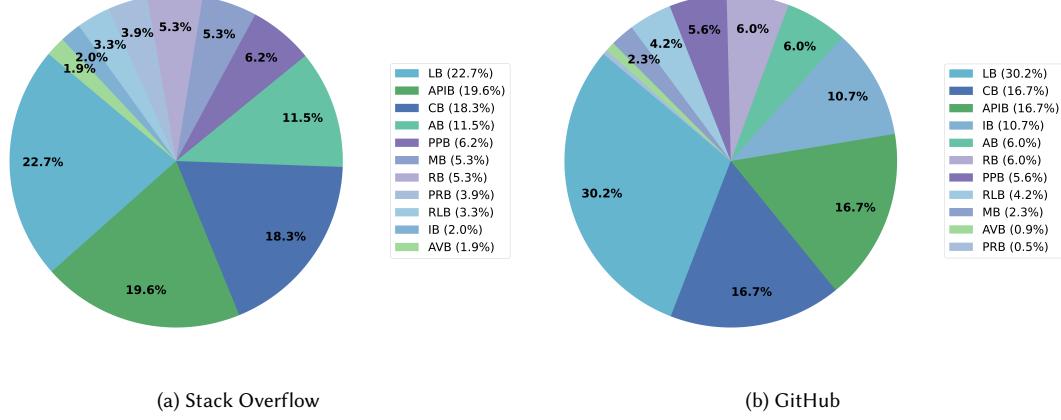


Fig. 2. Distribution of bug types across different sources.

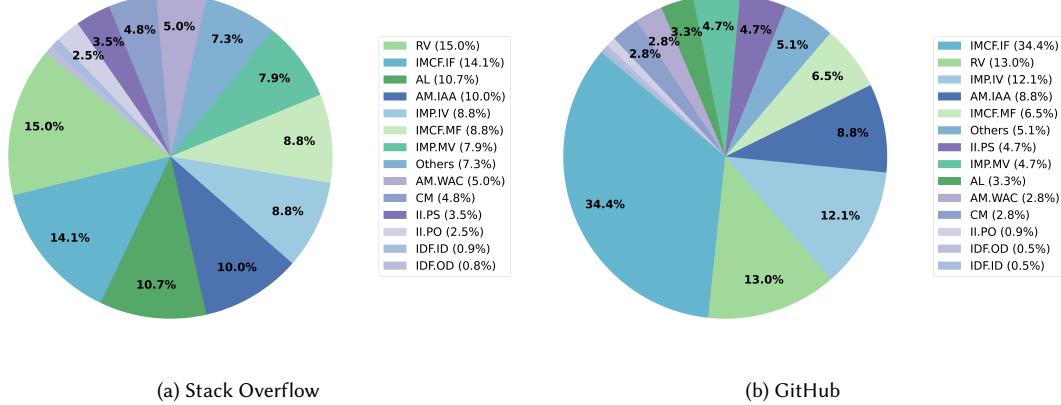


Fig. 3. Distribution of root causes across different sources.

Finding 1: Logic bug is the most common type of bug, with 22.7% and 30.2% of all bugs in Stack Overflow and GitHub.

Unlike bug types, where the top three categories account for over 60% of all bugs, the root causes are more diverse. The primary root cause of bugs in Stack Overflow is Requirement Violation. Building LLM agents often requires multiple frameworks, and since these frameworks are constantly evolving, keeping track of requirements becomes a challenging task. In GitHub, however, this root cause is less common because the platform mostly contains finished projects where library versions are often fixed. To minimize bugs caused by requirement violations, developers can develop tools like Poetry [2] to lock project dependencies, reducing the risk of installing conflicting packages.

Finding 2: Requirement violations and incorrect or missing control flow are the most common root causes reported on Stack Overflow and GitHub, accounting for 15% and 34.4%, respectively.

Crash is the most dominant type of effect across both platforms, accounting for 61% of bugs in Stack Overflow and 47% in GitHub. Since it is often difficult to determine the effect from GitHub commits when it is not explicitly mentioned in the commit description. Furthermore, Unknown emerges as a prominent effect type in GitHub. While some tools exist that localize bugs based on the crash report, their effectiveness in LLM agents is yet to be discovered [52]. Incorrect output is also a common effect observed across all platforms. Hang and Resource Overuse are among the least common effects in LLM agents.

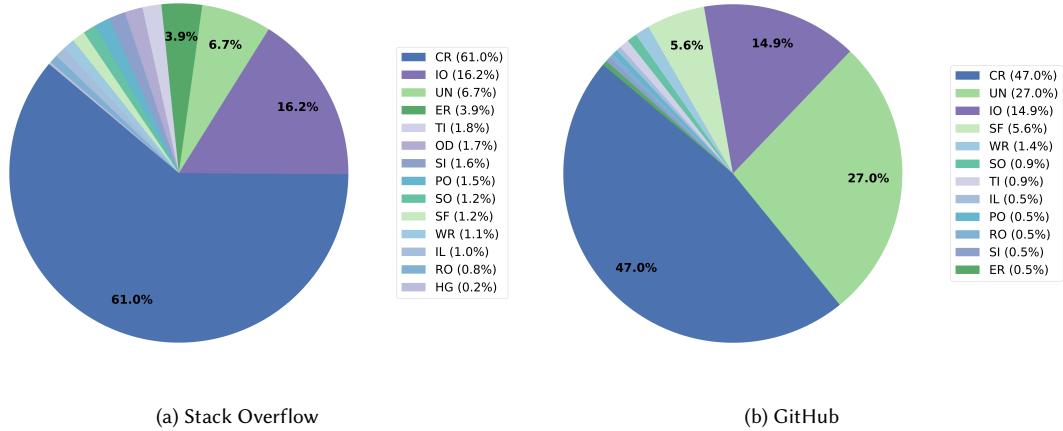


Fig. 4. Distribution of effects across different sources.

Finding 3: The most frequent effect of bugs is a crash, with 61% of cases on Stack Overflow and 47% on GitHub.

3.3 Most bug-prone LLM agent component (RQ2)

Although identifying the exact component affected by a bug in GitHub commits is challenging due to large project sizes, the agent core is the most error-prone among the known components. This is similar to Stack Overflow, where the agent core accounts for 58% of the bugs, followed by tools. This analysis indicates that prioritizing testing, debugging, and monitoring of the agent core can significantly reduce the likelihood of bugs. The distribution of LLM agent components where the bug occurred is provided in Figure 5, 6, and 7. As shown in Figure 5, for all bug types except Initialization and Resource Limitation bugs, those occurring in the agent core account for more than half of the total bugs. Since Resource Limitation bugs do not occur due to LLM agents, they do not fall under any category of the agent components.

The distribution of components across root cause and effect shows a similar pattern as shown in Figure 6 and 7, respectively. According to the data, except for bugs that occurred due to Incorrect Data Flow (Input Data Format Error) and Others, the agent core is the primary component where bugs occurred. While it is difficult to identify the bug type and root cause solely by observing the output, determining its effects is comparatively easier. According to the

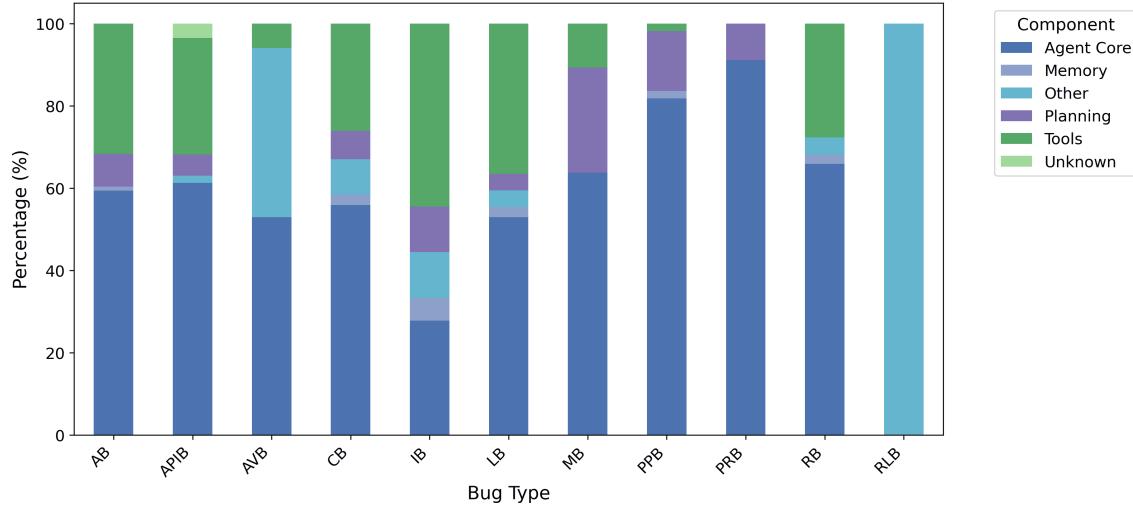


Fig. 5. Component distribution across bug type.

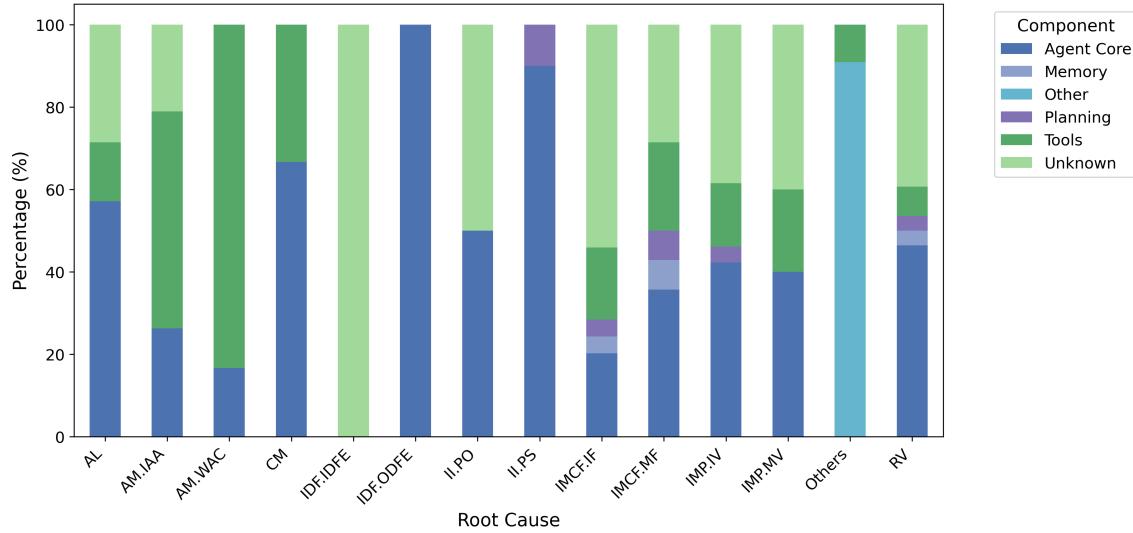


Fig. 6. Component distribution across root causes.

analysis, agents that exhibit indeterminate loops usually encounter issues in the planning stage in 66.6% of the time, whereas stateless interactions are caused by bugs in the memory component in 57.1% of cases. Therefore, when effects such as indeterminate loops or stateless interactions are observed, developers can focus on the planning and memory components, respectively, to speed up bug localization.

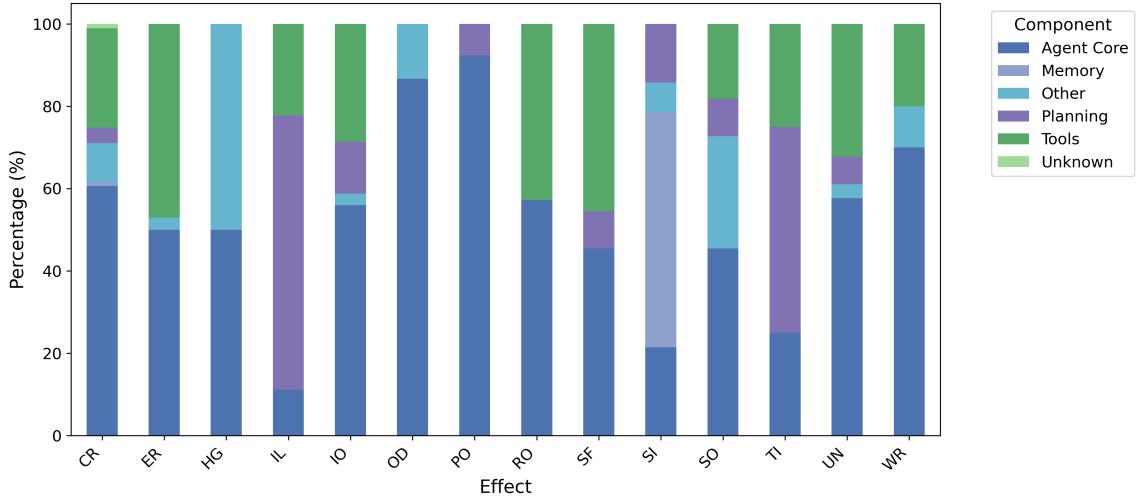


Fig. 7. Component distribution across effects.

Finding 4: The agent core is the most error-prone component of LLMs, with 58% of bugs on Stack Overflow occurring in this component.

Finding 5: Indeterminate loops result from issues in the planning stage, making up 66.6%, while stateless interactions are caused by bugs in memory components, accounting for 57.1%.

3.4 Distribution between bugs (RQ3)

To answer RQ3, we analyzed the correlation between bug types and their root causes. As shown in Figure 8, the majority of API-related bugs occur due to requirement violations. Given the rapid evolution of the field, new versions of libraries are frequently released, making it challenging for developers to manage the required library versions. Since building an agent often involves multiple libraries, developers frequently encounter conflicting version requirements. LangChain, being the most commonly used library [45], is particularly susceptible to requirement violation errors. As discussed in RQ1, tools like Poetry [2] can lock the versions of libraries along with the dependencies. Using virtual environments can also solve some of the conflicts. The second most common cause of API bugs is API limitations, which can also be attributed to the rapid pace of development of the libraries, with limited testing. While Argument Bugs are the fourth most common type of bugs in LLM agents, they predominantly occur due to API misuse. Given that the field is still emerging, developers are often unfamiliar with existing methods and their signatures. The proportion of this bug type has been decreasing each year since 2023, as shown in the analysis for RQ4. The analysis also suggests that most prompt-related bugs arise from incorrect prompt structure (i.e., improperly written prompts), whereas a smaller number of prompt bugs are caused by issues in prompt logic (prompt orchestration), such as incorrectly passing prompt variables. Lastly, since resource limitation bugs are not directly related to the development of the agent, they are categorized under ‘Others’ root causes.

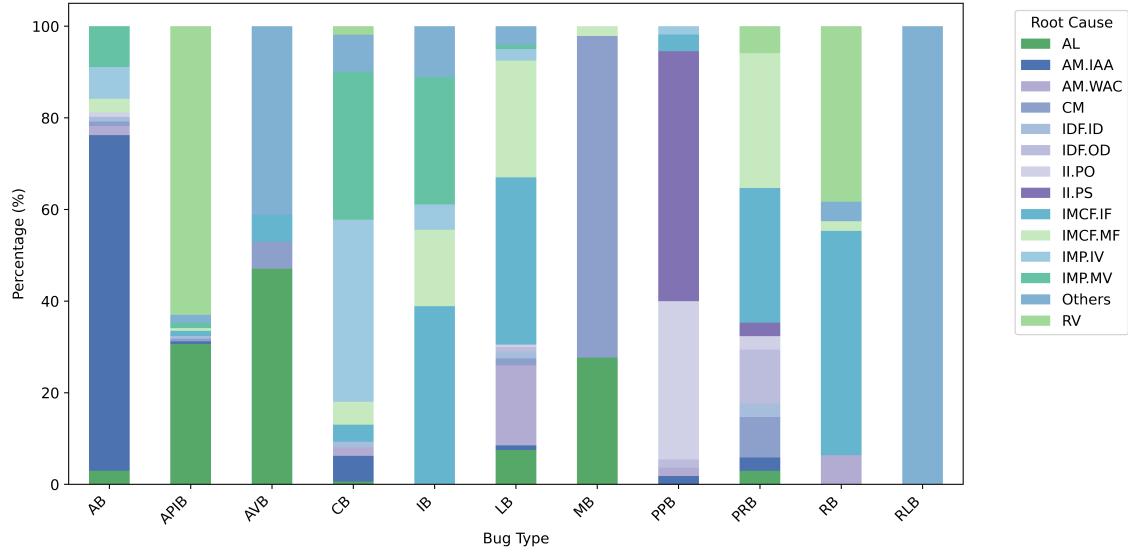


Fig. 8. Distribution of root cause across bug types.

Finding 6: 51.5% of bugs that occur due to requirement violations are associated with LangChain.

3.5 Bug distribution over time (RQ4)

To evaluate the distribution of bugs across different years, we analyzed the six characteristics based on the post date. According to the analysis presented in Figure 9, during 2021 and 2022, which marked the early stage of LLM agents, only a small number of Stack Overflow posts related to bugs in LLM agents were made, totaling fewer than 30 in the collected dataset. At the end of 2022, with the release of libraries for building LLM agents, for instance, LangChain in October 2022, the development of LLM agents surged, resulting in the highest number of bug-related posts on Stack Overflow in 2023. In our dataset, 2024 observed a lower number of posts related to bugs in LLM agents, suggesting that developers either faced fewer challenges in building these systems or that development activity decreased slightly. This trend may also reflect the community becoming more familiar with the area, which leads to fewer issues being reported. The number of posts in 2025 does not reflect the entire year, as data collection stopped after mid-2025.

As shown in Figure 10, the proportion of reported bugs in JavaScript has been decreasing since 2023, with 14.1% of bugs reported in 2023 coming from JavaScript, dropping to 7.4% in 2025. In contrast, the proportion of bugs reported in C# has been increasing during the same period, rising from 2.6% to 7.4%. The trend may suggest that developers are using JavaScript less for building agents, or that fewer bugs are being encountered when developing LLM agents with JavaScript. Further investigation is needed to better understand this trend.

Finding 7: The posts related to JavaScript for building LLM agents have decreased by 6.7% in two years. While the posts of C# are increasing by 4.8% due to frameworks like Semantic Kernel.

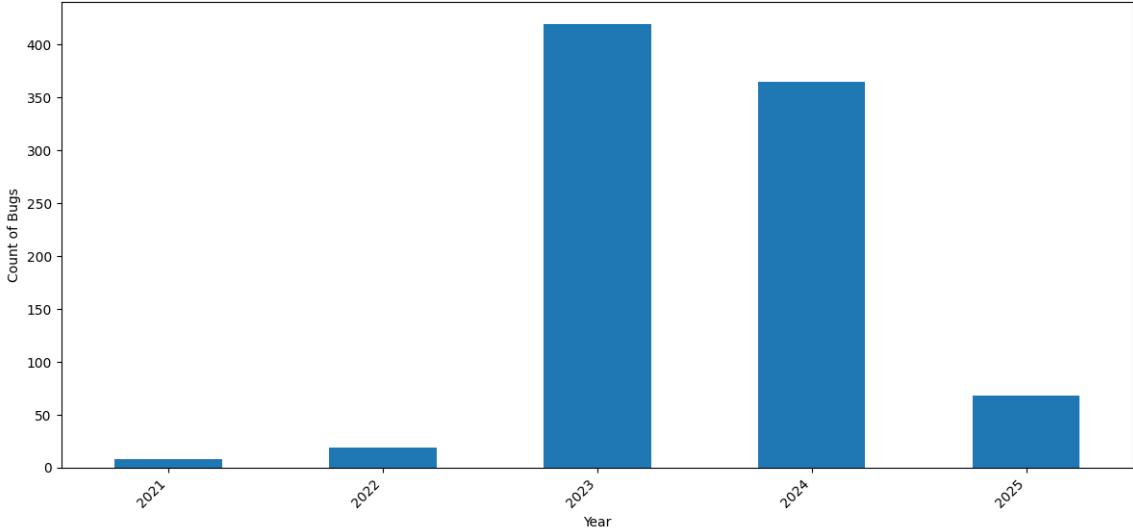


Fig. 9. Yearly distribution of bug-related posts in LLM-agent topics on Stack Overflow.

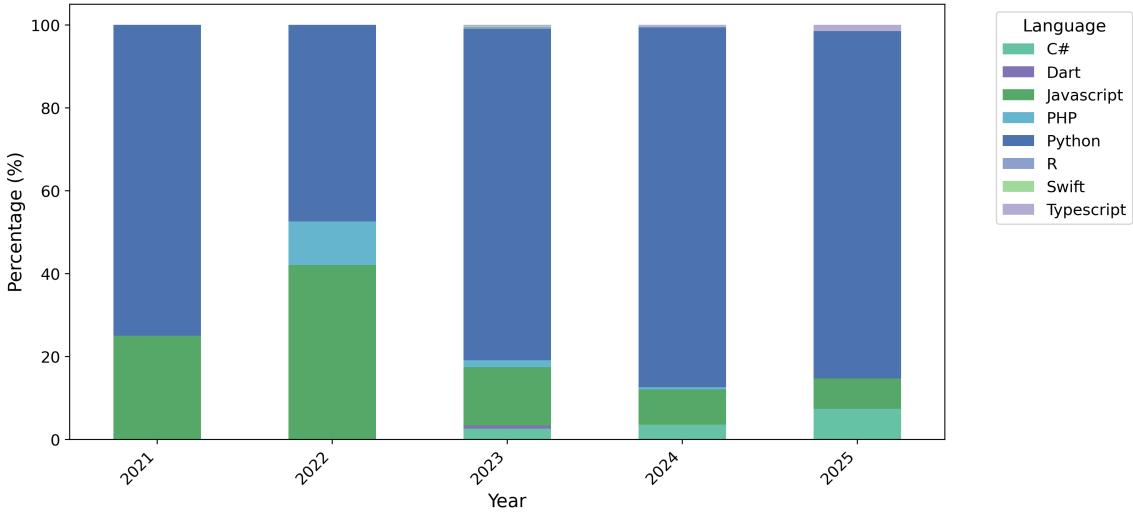


Fig. 10. The yearly distribution of programming languages used for developing LLM agents.

While the majority of libraries, such as LangChain, Autogen, and CrewAI, were released at the end of 2022, developers began using these libraries instead of relying on custom-made agents. According to the distribution of frameworks reported in Figure 11, LangChain remains the most popular framework for building LLM agents. Based on the collected dataset, the use of Semantic Kernel for building agents increased from 1% in 2023 to 8.8% in 2025. As agents have become more complex and multi-agent systems are being developed, LangGraph has gained significant popularity, with its

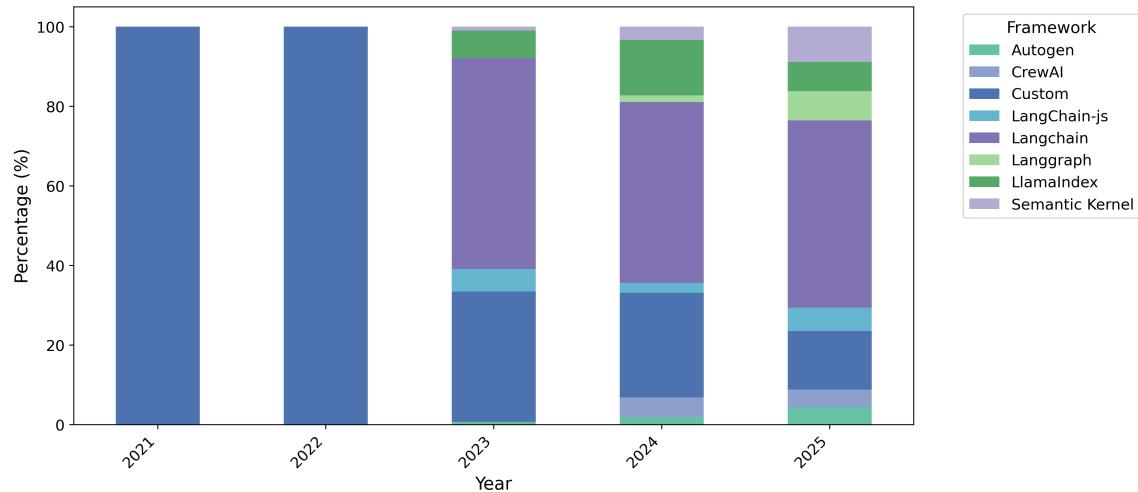


Fig. 11. The yearly distribution of frameworks used for developing LLM agents.

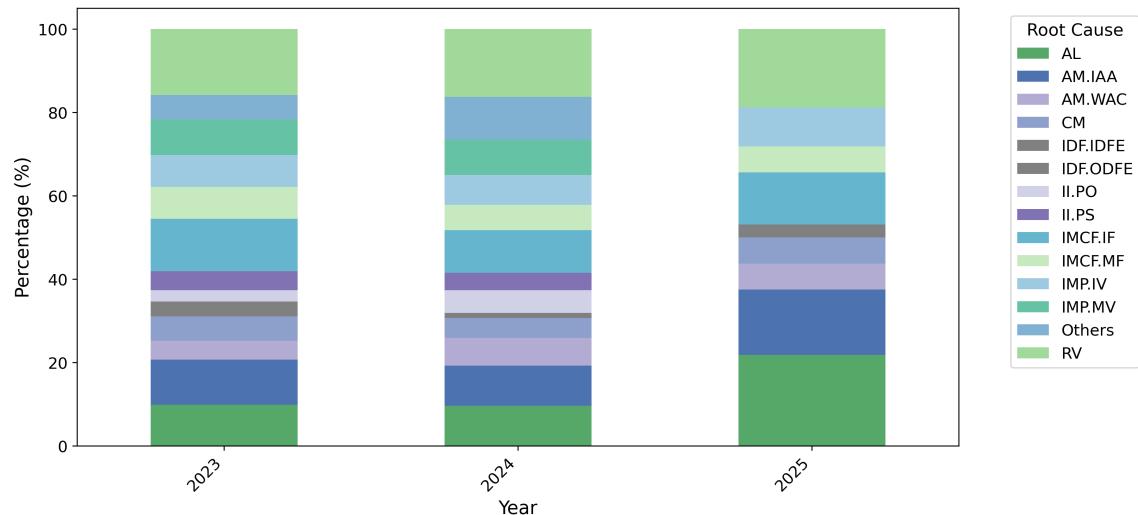


Fig. 12. Distribution of the root causes in LLM agents developed with LangChain.

share of total reported bugs rising from 1.6% in 2024 to 7.4% in 2025. The trend indicates the need for an automatic code generation and optimization tool for LLM agents like Auto-Keras for deep neural networks [44].

Finding 8: The use of custom methods for building LLM agents has been decreasing yearly. In 2021 and 2022, 100% of the agents were built without using any frameworks, which has dropped to 14.7% in 2025.

Since LangChain is the most popular tool for building LLM agents [45], we investigated the common root causes of bugs that occur when developing agents with LangChain. According to the study, with details presented in Figure

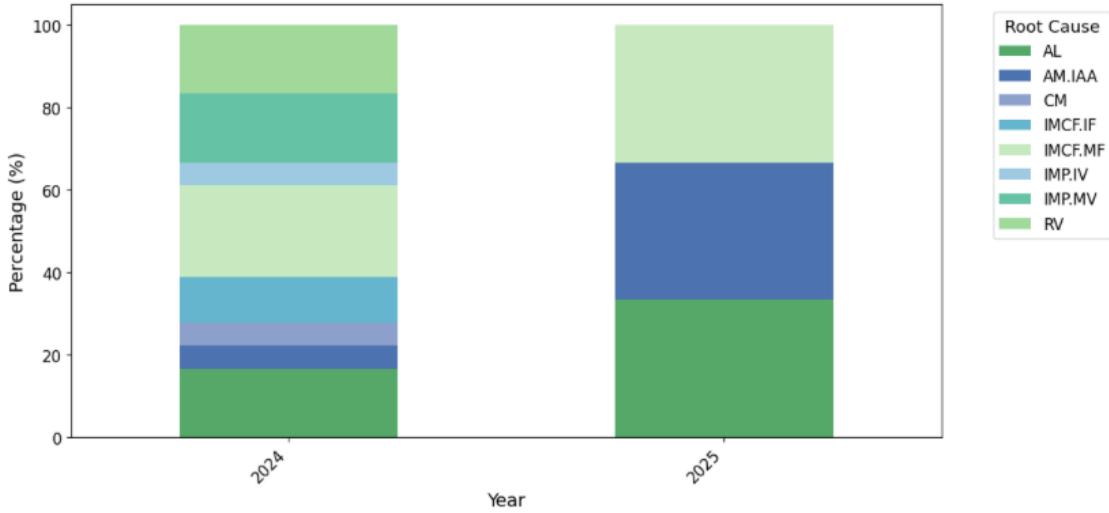


Fig. 13. Distribution of the root causes in LLM agents developed with CrewAI.

[12](#), API limitations and API misuse accounted for 25.2% of bugs in 2023, increasing to 43.7% in 2025. A similar pattern is observed in CrewAI (Figure 13), with the proportion rising from 22.3% in 2024 to 66.6% in 2025. Additionally, an increasing number of bugs are occurring due to API limitations in libraries such as LangChain-js, LlamaIndex, Semantic Kernel, and Autogen is also reported. This trend can be attributed to the rapid development of the field, which forces libraries to evolve quickly, creating opportunities for potential bugs. Library maintainers should prioritize more rigorous documentation before releasing the framework. To address the rise in bugs related to API misuse, similar to issues discussed in the deep learning field [98], researchers should treat agentic libraries and general-purpose libraries as distinct categories when developing methods to prevent API misuse.

Finding 9: In LangChain, API limitation and API misuse accounted for 25.2% of bugs in 2023, increasing to 43.7% in 2025. A similar growing pattern of these root causes has been observed in CrewAI as well.

3.6 Cross-System Bugs Comparison

Our study shows that several of the bug types we identified, such as Logic Bugs and Initialization Bugs, as well as certain root causes such as API Misuse and effects such as Crash, have also been observed in other software systems. However, we also identified several bug types that were not reported in previous studies, including Parsing Bugs, Reference Bugs, Availability Bugs, Model Bugs, and Resource Limitation Bugs. Similarly, this study presents unique root causes that have not been previously documented, such as the Prompt Orchestration subcategory under Incorrect Instruction root causes. These bugs are not frequently observed in other software systems due to the unique characteristics of agentic systems. For instance, while parsing is required in many software systems, LLMs generate dynamic output that necessitates parsers generalized enough to accommodate different output formats. Similarly, as the agentic field is in its evolving stage, modules and functions are frequently relocated to different packages to improve library organization. Users often fail to track these reference changes and continue to refer to outdated paths, which commonly results in

import errors. Regarding availability bugs, since agentic systems rely on external LLMs or tools, service downtime in these dependencies can bring down the entire agent. Likewise, while model bugs may be found in other AI systems, the challenge is amplified in agentic systems because LLMs have different variations; some are designed exclusively for conversation, while others are fine-tuned for instruction-related tasks, making it difficult for users to select the appropriate LLM type for specific tasks. Finally, resource limitation bugs can also appear in other software systems. However, because LLMs require extensive and specialized computational resources, users report these issues more frequently in agentic systems.

Although some bugs have been reported in previous studies across other software fields, their distribution differs significantly from that of agentic systems. For instance, in deep learning compiler systems, 9.78% of bugs have a root cause of API misuse [67], while another study in open-source C programs found that it contributes to 17.05% of all bugs. In comparison, API misuse accounts for 15% of bugs in agentic systems on Stack Overflow and 11.6% on GitHub. Similarly, bugs related to data format contribute to 25% of total bugs in deep learning systems [65], whereas they represent only 1.7% in agentic systems (Stack Overflow dataset). This discrepancy can be attributed to the fact that, unlike traditional deep learning systems, agentic systems typically do not fine-tune models or build them from scratch. Instead, pre-trained models are utilized for building agents, which reduces dependencies on input datasets and associated data formatting issues. However, some similarities in bug distribution are also observed. For instance, API bugs in deep learning systems built with PyTorch account for 16% [39], which aligns closely with the distribution found in this study.

3.7 Actionable Insights

This section discusses the implications obtained from our study (RQ1-RQ4) and shares recommendations on what developers, library maintainers, and researchers can do to minimize the bugs in LLM agents. A summary of the insights is shared in Table 5.

3.7.1 Developers. From RQ3, we observed that the most common cause of API Bugs is requirement violation, affecting 63% of this bug type. It is challenging for developers to fully address this issue, as new library releases often introduce conflicting requirements, and because the field is still emerging, frameworks frequently release newer versions. To address some of the issues, developers can leverage dependency and packaging tools such as Poetry [2], which define a project's dependencies and configuration in a single file, ensuring consistent package versions throughout the project. In addition, most of the argument-related bugs occur due to passing wrongly aligned parameters in the method. Although the proportion of this bug type has been decreasing since 2023, developers can integrate type-checking tools to ensure that methods are invoked with the correct parameters [56]. Lastly, bugs occurred due to API Limitations contribute to 10.7% of the bugs in Stack Overflow. RQ4 indicates that the occurrence of bugs caused by API limitations has been increasing over the past three years across most libraries. While identifying bugs within a framework can be challenging for a small development team, a practical approach is to review the library's open issues before adopting it to gain awareness of some of the potential challenges.

3.7.2 Library Maintainers. As libraries related to LLM agents continue to grow rapidly, new versions often create conflicts with existing dependencies. Therefore, when releasing a new version, library maintainers should not only specify the newly added features but also document any underlying version changes, enabling developers to anticipate potential conflicts when installing the updated library. To further mitigate the increasing number of bugs related to API

Table 5. Summary of key findings and recommended actionable insights

RQs	Finding	Actionable Insights
RQ1	Logic bug is the most common type of bug found in LLM agents.	Researchers: Update existing tools such as Pylint to adapt to bugs in agents.
	Requirement violations and incorrect or missing control flow are the most common root causes.	Developers: Use dependency management tools and virtual environments to prevent some bugs related to requirement violations. Library maintainers: Specify underlying requirements and any version changes in new releases.
	The most frequent effect of bugs is a crash.	Researchers: Test the effectiveness of current crash analysis tools on LLM agents.
RQ2	Indeterminate loops usually arise from problems in the planning stage, whereas stateless interaction usually occurs because of bugs in the memory stage.	Developers: When you encounter bugs that cause the agent to run for an indeterminate amount of time or fail to remember the previous conversation, check the planning stage and the memory stage first, as this usually leads to faster debugging.
RQ3	Majority of the bugs occurred due to requirement violations associated with LangChain.	Library maintainers: Specify underlying requirements and any version changes in new releases.
RQ4	The posts related to JavaScript is decreasing while C# is increasing.	Researchers: Focus more on agentic bugs in C#.
	In LangChain and CrewAI, bugs related to API limitations and API misuse have been increasing.	Library maintainers: Conduct more rigorous testing before releasing a newer version. Researchers: Deal with agentic and general-purpose libraries differently while building tools for API misuse

limitations, as highlighted in RQ4, library maintainers should conduct more extensive testing before releasing new versions.

3.7.3 Researchers. As API limitations are a common issue not limited to LLM agents, multiple Software Composition Analysis (SCA) tools exist that can report known vulnerabilities in libraries, licensing issues, and outdated components [66]. OSV-Scanner [22], developed by Google, is one of the popular tools in this domain. It queries the existing OSV database [1] to identify vulnerabilities; however, while the database includes some known vulnerabilities of LLM agent-related libraries, such as LangChain, OSV-Scanner often fails to capture the potential risks specific to these libraries. To the best of our knowledge, existing SCA tools are not well-suited for LLM agent-related libraries. Researchers could extend existing SCA tools or develop new tools that aggregate known issues from community platforms such as Stack Overflow, Hugging Face forums, OpenAI Discussions, or Reddit, providing developers with a more comprehensive understanding of potential vulnerabilities before using a library.

4 LLM Agent for Labeling Bugs

This section describes the LLM agent designed for annotating bugs. As shown in Figure 1, the agent consists of two main components. The first component is a ReAct agent that leverages different tools to reason about where the bug occurred. The second component takes this reasoning and maps it to the appropriate class labels. The subsequent sections explain the two components of the agent in detail, along with an example explaining the classification process.

4.1 BugReAct Agent

To annotate bugs in LLM agents, we built a ReAct agent [100] equipped with 10 tools. These tools can scrape the documentation of the following libraries: LangChain, LangChain-js, LangGraph, Pydantic, Crewai, LlamaIndex, Semantic Kernel, and Autogen. In addition, we integrated two community forums, the OpenAI developer community and GitHub Discussions. The GitHub Discussions tool searches the discussion tabs of the eight frameworks mentioned above. Since searches in documentation and forums can yield irrelevant results that may overflow the context window and impair the LLM’s decision-making, we introduced a summarization module after each search. This module condenses the retrieved information, preserving only the most critical content. To further improve efficiency, we integrated an in-memory database (Redis [63]) to ensure that once content has been searched and summarized, it does not need to be processed again. The integration of an in-memory database instead of a traditional one makes the agent faster by reducing the delay for retrieving information. The ReAct agent is allowed to invoke any tool multiple times, with each tool’s output treated as an observation. Based on these observations, the agent determines its subsequent actions. Once it has gathered sufficient information to reach a conclusion, the agent produces a final output that includes both an explanation of the bug and the reasoning behind its occurrence.

4.2 Classification

This is the smallest component of the classification agent. Its responsibility is to take the explanation generated by the ReAct agent in the previous step, together with the post or code snippet containing the bug, and pass them through an LLM for classification. The LLM is provided with the set of class labels and their definitions through the prompt and is instructed to assign one label for each of the six categories. In addition, it is asked to provide the rationale for selecting the bug type, root cause, and effect to support further analysis. We use the Pydantic library to enforce structured output from the LLM. For LLMs that do not natively support structured outputs, we rely on strict prompting to ensure the response conforms to the required format.

4.3 Workflow of BugReAct with an Illustrative Example

Figure 14 illustrates the decision-making workflow of a Stack Overflow post [6] processed by *BugReAct*. The process begins when the example post is wrapped within a prompt and passed to the LLM agent (1.a), initiating the planning stage. Here, the agent analyzes the input and formulates an action plan (1.b). Once the plan is established, it transitions to the action stage, where it determines that consulting the LangChain documentation is required (1.c). The tool responsible for documentation search is then invoked with a specific query, and control is passed to the tool. The tool first checks whether the keyword already exists in the database (1.d); if not, it queries the LangChain documentation directly (1.e). Since retrieved content may contain irrelevant details or exceed the model’s context window, it is processed through a summarization step (1.f). The summarized output is stored in the Redis database for future use (1.g.1) and then returned to the agent (1.g.2). The agent interprets this result as an observation and proceeds to the thought stage (1.h). If the output is insufficient, the tool may be re-invoked to gather more information (1.i). Once adequate evidence is collected, the agent produces a final decision that includes an explanation of the underlying cause of the bug, thereby concluding the first stage of *BugReAct*.

The second stage begins when the final output from Stage 1 (2.a.1), along with the original example input (2.a.2), is passed to the LLM. At this stage, the model is also given access to a predefined set of labels and their corresponding definitions. Based on the combined inputs, it selects six labels that describe the characteristics of the fix, accompanied by

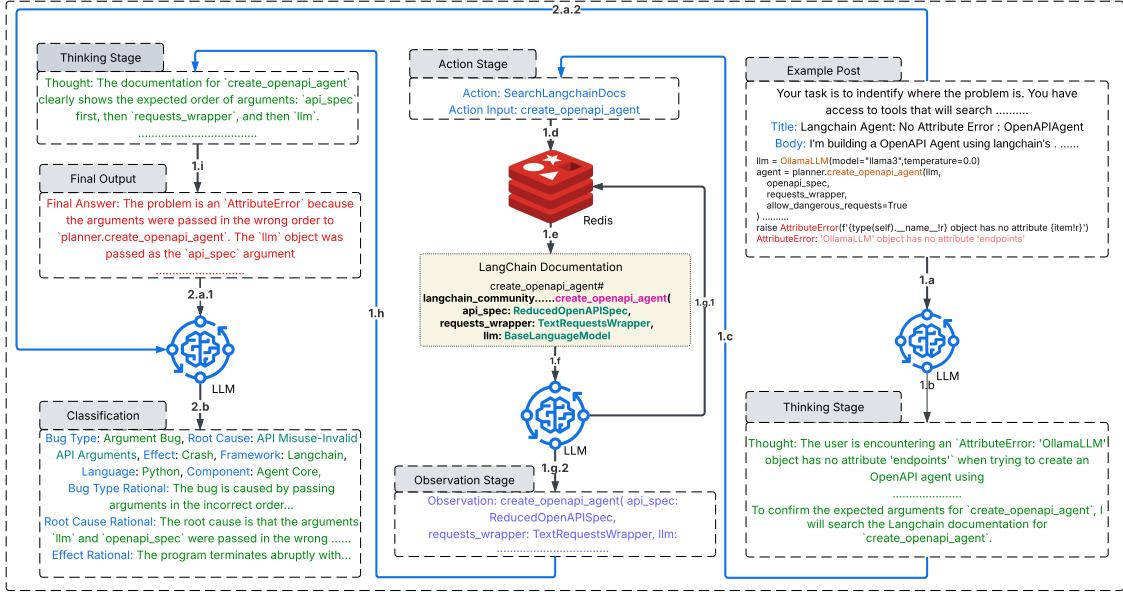


Fig. 14. The flow of automatic annotation for a Stack Overflow post [6] through *BugReAct*, equipped with Gemini 2.5 Flash.

three rationales for bug type, root cause, and effect (2.b). For each label, the model provides an explanation to justify its selection. The predicted set of six labels is then compared against the human-annotated benchmark, enabling systematic evaluation of the model’s performance.

5 Evaluation

We implemented *BugReAct* in Python version 3.13.1. LangChain was used as the primary framework for building the agent, along with some additional frameworks like Pydantic for structured output, Selenium for scraping web pages, and Redis for saving data into an in-memory database. The experiment was conducted on a MacBook equipped with an M3 chip and 16 GB RAM. For invoking LLMs, we have used OpenAI, OpenRouter, and Claude APIs. The implementation relies on modular structures along with proper exception handling, which allows scalability and robustness. We have evaluated *BugReAct* based on the following research questions:

- **RQ5 (Effectiveness)** How effective is *BugReAct* in characterizing bugs in LLM agents?
- **RQ6 (Reliability)** How accurate is *BugReAct* in labeling LLM bugs compared to human annotators?
- **RQ7 (Efficiency & Cost)** How efficient and cost-effective is *BugReAct* in characterizing bug types, root causes, and effects?

5.1 Effectiveness

To evaluate the effectiveness of *BugReAct*, we have evaluated it on the annotated Stack Overflow and GitHub dataset. Furthermore, we have experimented with three highly powerful LLMs, namely Gemini 2.5 Flash [12], GPT o3 mini [57], and Claude Sonnet 4 [5]. The selected LLMs have achieved state-of-the-art performance in reasoning and are compatible with ReAt agents. We ensured that selecting LLMs from different providers for a more diverse evaluation.

Table 6. F1-score of different models in different datasets

Label	Stack Overflow						GitHub					
	Without Answer			With Answer			Issues			Commits		
	Gemini	GPT	Claude	Gemini	GPT	Claude	Gemini	GPT	Claude	Gemini	GPT	Claude
Bug Type	0.57	0.51	0.55	0.66	0.61	0.53	0.54	0.44	0.47	0.64	0.48	0.58
Root Cause	0.45	0.35	0.34	0.57	0.49	0.46	0.46	0.27	0.24	0.64	0.62	0.46
Effect	0.89	0.82	0.84	0.90	0.83	0.83	0.86	0.73	0.80	0.59	0.55	0.34
Language	0.96	0.97	0.98	0.94	0.96	0.96	0.89	0.94	0.91	0.95	0.96	1.00
Component	0.72	0.96	0.57	0.72	0.52	0.51	0.48	0.46	0.45	0.49	0.53	0.37
Framework	0.76	0.55	0.98	0.66	0.96	0.97	0.35	0.37	0.35	0.30	0.28	0.27

Since the collected dataset is highly imbalanced, accuracy alone often provides a biased evaluation. Therefore, we report the F1-score, which combines both precision and recall to offer a more balanced assessment. Table 6 presents the F1-scores of the three LLMs on the Stack Overflow and GitHub datasets. For a deeper analysis of the Stack Overflow dataset, we also evaluate how *BugReAct* performs when provided with the accepted answer compared to without it.

The results show that Gemini 2.5 Flash consistently achieves the best performance across both Stack Overflow (with and without accepted answers) and GitHub (issues and commits). Following that, GPT o3 mini outperforms Claude Sonnet 4. This difference can be attributed to several factors. GPT o3 mini tends to produce shorter and more compact outputs [7]. However, in doing so, it often neglects to call the necessary tools to verify its knowledge, unlike Gemini 2.5 Flash, which actively invokes tools to confirm specific details. Claude Sonnet 4, the latest model in the Claude Sonnet family, frequently invokes tools excessively and struggles to converge on a correct conclusion. Although our pilot study with Claude Opus 4 demonstrated an 8.3% improvement in both accuracy and F1-score over Claude Sonnet 4, the significantly higher cost of this model makes it impractical for real-world applications.

5.2 Reliability

For evaluating the reliability of the proposed *BugReAct*, we compare its predictions with those of a human annotator on the Hugging Face Forums dataset. Figure 15 shows this comparison of Gemini 2.5 Flash and GPT o3 mini. As illustrated, *BugReAct* matches only 39.4% with the best performing Gemini 2.5 Flash of the human annotations when it is provided with just the title and body of the post. However, the match increases to 59.3% when replies, specifically those marked by the human annotator as containing a solution, are also included. A higher level of agreement is observed in critical components such as root cause, effect, and LLM agent component, while less critical aspects such as language and framework show a slight decrease in performance. While the performance dropped significantly for GPT o3 mini and Claude Sonnet 4, *BugReAct* demonstrates strong agreement with the human annotator. This agreement is particularly evident when the model is given the full context of the post, including replies and the conversation between the user and the solution provider. Nonetheless, whether the agent can consistently provide annotations along with clear rationales, comparable to human annotators, remains an open question for future research.

5.3 Efficiency and Cost

To evaluate the effectiveness and cost of *BugReAct*, we recorded the API cost and the time taken to reach a solution. Table 7 presents the average cost and time per post or commit across the three platforms. Our analysis shows that Claude Sonnet 4 consumes the most time and incurs the highest cost to reach a solution. Gemini 2.5 Flash requires slightly more time on average than GPT o3 Mini, but unlike GPT o3 Mini, it invokes multiple tools before arriving at a

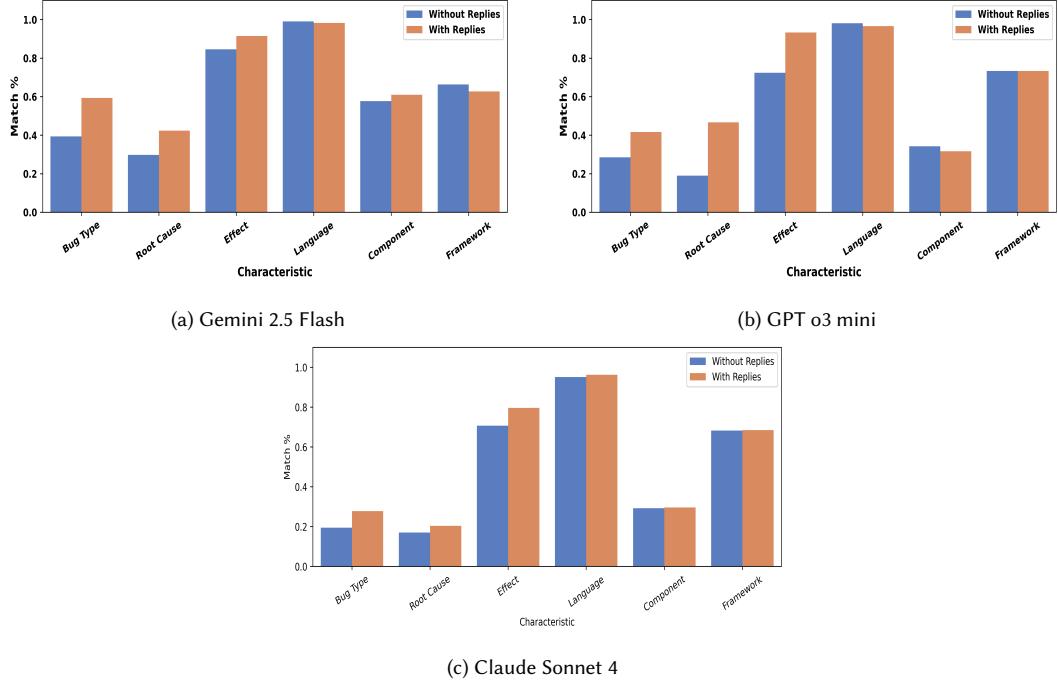
Fig. 15. Comparison of *BugReAct*’s match with human annotators, with and without replies.

Table 7. Effectiveness and cost of different LLMs across different platforms.

LLM	Stack Overflow				GitHub				Hugging Face			
	Without Answer		With Answer		Issues		Commits		Without Answer		With Answer	
	Time(s)	Cost(\$)	Time(s)	Cost(\$)	Time(s)	Cost(\$)	Time(s)	Cost(\$)	Time(s)	Cost(\$)	Time(s)	Cost(\$)
Gemini	33.996	0.016	7.824	0.006	22.306	0.008	18.008	0.013	20.185	0.021	9.394	0.007
GPT	19.090	0.017	11.841	0.005	17.277	0.014	15.407	0.002	25.778	0.024	14.688	0.010
Claude	125.935	0.049	55.901	0.039	118.184	0.025	97.358	0.013	100.380	0.041	75.674	0.015

solution. Additionally, the agent requires fewer resources, both in terms of time and cost, when it is provided with an answer from the community forum. In this scenario, the agent makes fewer tool calls and reaches a solution more quickly, thereby reducing the overall cost. We also conducted a pilot study with Claude Opus 4 on 110 Stack Overflow posts without providing any accepted answers, which took an average of 115.4 seconds to reach a solution and cost an average of 0.455 USD per post.

5.4 Comparison with Baselines

To compare the proposed approach with existing solutions, we conducted an extensive study. The comparison includes both encoder-based and decoder-based LLMs across four different architectural settings. This section illustrates these architectural settings and discusses their performance in detail.

5.5 Encoder

This architectural setting is commonly used for LLM-based classification tasks [14, 26]. In this approach, the input text is fed into an encoder LLM, which generates an encoded representation that is then passed to a classification head for the final prediction. While this method is often fast and robust for specific cases [37], a key limitation is its requirement for a labeled training set for downstream tasks. In our study, we investigated CodeBERT-base [17] and CodeT5-base [94] due to their state-of-the-art performance in coding-related tasks [4, 19]. Notably, although CodeT5 is an encoder-decoder model, only the encoder component was used in our experiments. Because this approach requires a training dataset for fine-tuning, we randomly selected 80% of the dataset for training and used the remaining 20% for fine-tuning.

5.6 Zero-shot

In prompt-based zero-shot classification, the LLM is provided with the input text and a set of possible labels, and it analyzes the text and selects one of the given options for classification, without requiring any additional information [93]. For this study, the LLM receives the input text and, similar to the classification stage of *BugReAct*, the LLM makes a prediction based on the input text only. The zero-shot architecture leverages the same LLMs used for evaluating *BugReAct* (Gemini 2.5 Flash, GPT o3 mini, and Claude Sonnet 4). We employed structured output in Gemini 2.5 Flash and GPT-3 mini to ensure that the LLM generated responses corresponding to the predefined labels. Since Claude Sonnet 4 does not support structured output, we relied on strict prompting to ensure that the LLM generates only the predefined labels.

5.7 One-shot

This architecture is similar to zero-shot, except that one example from each category is provided. Since our classification involves multiple labels, we initially selected the example with the shortest character length to ensure that every example fits within the prompt window. However, for Stack Overflow and GitHub issues, even the shortest example exceeded the window length. To address this, we used the same LLM to generate a concise summary of each example, which was then included in the prompt instead of the full text (e.g., if Gemini 2.5 Flash is used for classification, it is also used to generate the summary).

5.8 ReAct Agent

This architecture is similar to that of *BugReAct*, except it lacks the ability to extract useful information from tools. Since the main contribution of *BugReAct* is the integration of tools and the summarization of the outputs extracted by these tools, this architecture is used to evaluate the effectiveness of a standalone ReAct agent without tool usage. For this task, whenever the agent attempts to invoke a tool, a fixed response of “No results found” is returned. This setup allows us to measure how much the integration of newly developed tools improves overall performance.

5.9 Results Analysis and Discussion

Figure 16 presents the F1-scores of all the baseline architectures discussed above. Among the four non-trivial categories labeled by both human annotators (i.e., bug type, root cause, effect, and component), the best-performing model of *BugReAct* outperformed the baseline solutions in all cases except for the effect category in GitHub commits. For trivial categories, such as language and framework, the proposed agent also outperformed most of the baseline solutions. The

traditional ReAct agent achieved an average F1-score of only 24.65% in the non-trivial categories, while the proposed solution achieved 65.75% for the same labels using the best-performing Gemini 2.5 Flash LLM, which demonstrates the significant impact of the developed tools. The analysis also shows that encoder-based solutions achieved commendable performance. The one-shot approach, however, failed to provide satisfactory results and performed worse than zero-shot. This result can be attributed to the fact that the one-shot architecture relies on the example provided in the prompt. The example, in this study, is selected based on its length to fit within the context window, rather than as the most representative example of the category, which may have misled the LLM.

While *BugReAct* outperformed existing baselines, it has several limitations. The first limitation is its dependence on external sources. If these sources become unavailable, the performance may drop significantly. In addition, since it scrapes websites to obtain information, external factors such as internet speed can introduce a bottleneck and affect the tool's total execution time. Finally, the agent is evaluated on posts and small code snippets; the architecture may require adjustments when applied to a large code base.

Overall, *BugReAct* demonstrated noteworthy performance in annotating bugs related to LLM agents. While previous studies reported the limited self-healing ability of existing LLM agents [62], this study shows that LLM agents can identify and annotate bugs within agentic systems when appropriately designed. Although *BugReAct* is designed to assist annotators in labeling bugs related to LLM agents, it can also be adapted for fixing bugs in agentic systems by replacing the classification head with a fixation head. Future studies can explore *BugReAct*'s potential for repairing bugs in LLM agents.

6 Threats to Validity

6.1 Internal Threat

One potential internal threat to validity is the classification of bugs. While we did not use a predefined taxonomy, some of our labels were adapted from previous literature, and new labels were introduced when no prior work addressed a particular type of bug. To mitigate this threat, two researchers independently annotated the bugs and resolved disagreements through discussion with the assistance of a moderator with rich knowledge in the field that reduces subjective bias and improved the reliability of our classification. In addition, we have provided appropriate rationals behind selecting a particular class label for bug type, root cause and effect.

Another potential threat to validity is the annotation of Stack Overflow and Hugging Face Forums posts without accepted answers. To mitigate this, we marked posts with a verified solution, which includes the accepted answer, a solution provided by the user (via edits, comments, or new answers), user-confirmed solutions in comments, or solutions verified by our annotators. For posts without a verified answer, we referenced external resources that helped us make the decision. However, although both annotators verified the external resources, since these external solutions are not confirmed by the user, a small degree of subjectivity may remain. Another concern with the implementation of *BugReAct* is its potential impact on the evaluation and results. To minimize this threat, we evaluated two different seen datasets (Stack Overflow and GitHub) and one unseen dataset (Hugging Face), and we report the performance of the model's matches compared to human annotators.

6.2 External Threat

One potential external threat is the availability of the dataset. All posts and code snippets used for annotation were collected up to August 2025, and any modifications or deletions made after that date were not included in the study.

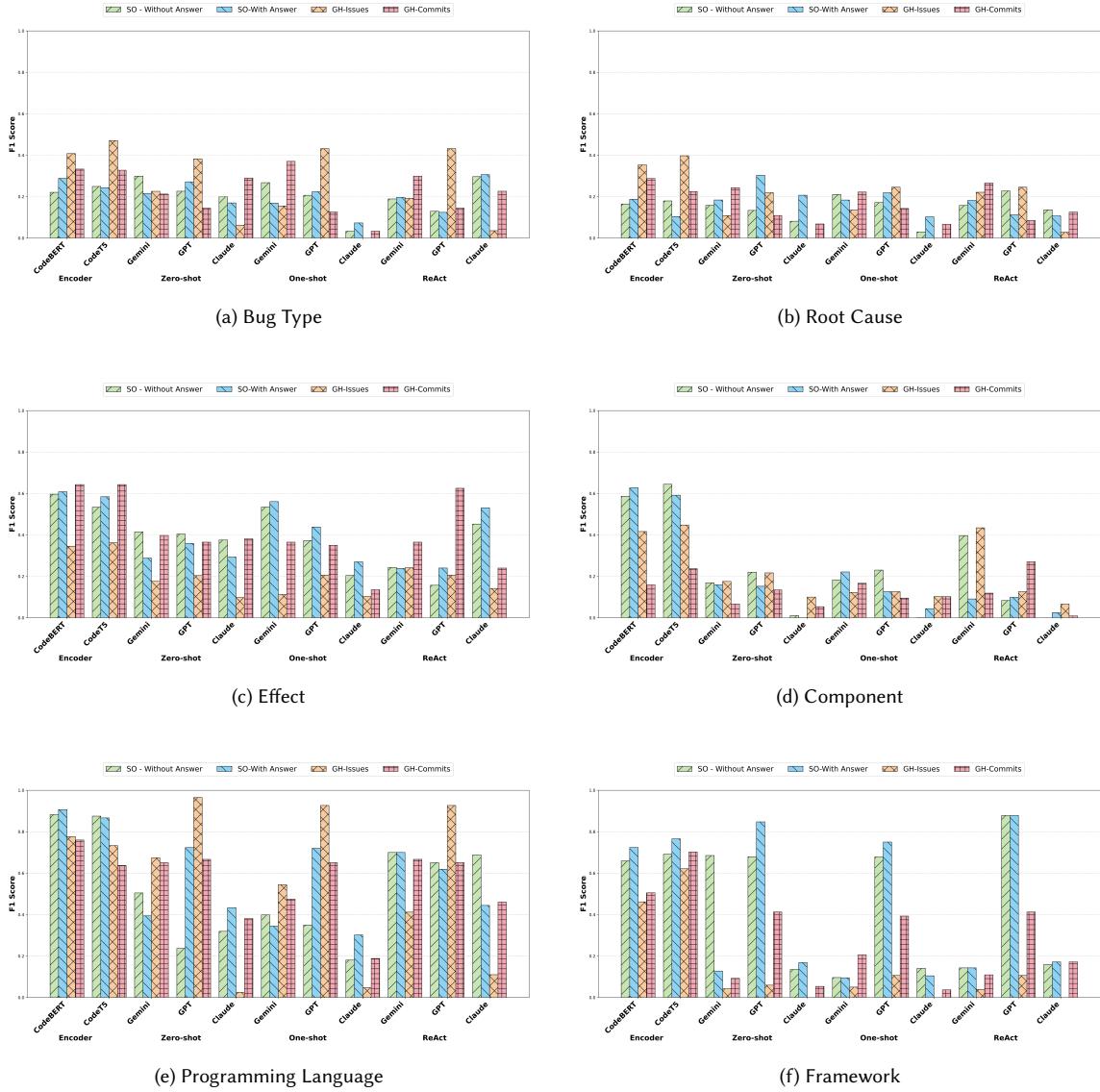


Fig. 16. Comparison of different LLMs and architectures.

Another external threat is the inclusiveness of the dataset. To address this threat, we selected libraries based on their popularity, considering both GitHub stars [45] and the number of related posts on Stack Overflow, focusing on widely used libraries for building LLM agents. However, we acknowledge that not all libraries used for developing LLM agents are represented in this study. Lastly, the proposed *BugReact* invokes 10 tools that scrape the specific web page to get information. Any changes to the web pages after November 2025 may affect tool calls, and the scraping logic might need to be adjusted accordingly.

7 Conclusions and Future Work

In recent years, the use of LLM agents has grown significantly, while developers have faced critical challenges in building and maintaining these systems due to the novelty of the field and limited knowledge about the types of bugs that occur. To address this issue, we conducted the first comprehensive study of characterizing bugs in LLM agents. Our empirical study provides a deeper understanding of bugs across three community forums, namely Stack Overflow, GitHub, and Hugging Face discussions. Based on our findings, we provide actionable takeaways for developers, library maintainers, and researchers to improve practices and accommodate necessary changes. Additionally, we developed an LLM-based ReAct agent named *BugReAct* to automatically annotate bugs from community platforms and evaluate the effectiveness of LLM agents in identifying and annotating bugs. Our study highlights the potential of LLMs, particularly Gemini 2.5 Flash, for this task at low cost. Future work should explore LLMs' capabilities in fixing bugs in LLM agents and perform deeper analyses to achieve human-level performance in both annotating and resolving bugs.

References

- [1] Accessed: 2025-09-10. OSV – Open Source Vulnerabilities Database. <https://osv.dev>.
- [2] Accessed: 2025-09-10. Poetry – Python dependency management and packaging tool. <https://python-poetry.org>.
- [3] Accessed: 2025-09-10. Pylint. <https://pypi.org/project/pylint/>.
- [4] Md Faizul Ibne Amin, Atsushi Shirafuji, Md Mostafizer Rahman, and Yutaka Watanobe. 2024. Multi-label code error classification using CodeT5 and ML-KNN. *IEEE Access* (2024).
- [5] Anthropic. 2025. Claude Sonnet 4. <https://www.anthropic.com/clause/sonnet>. Accessed: 2025-09-10.
- [6] Badhusha. 2025. *Langchain Agent: No Attribute Error : OpenAPIAgent*. <https://stackoverflow.com/questions/79565168> Accessed: 2025-09-10.
- [7] Marthe Ballon, Andres Algabe, and Vincent Ginis. 2025. The Relationship Between Reasoning and Performance in Large Language Models—o3 (mini) Thinks Harder, Not Longer. *arXiv preprint arXiv:2502.15631* (2025).
- [8] Federico Lorenzo Barra, Giovanna Rodella, Alessandro Costa, Antonio Scalagna, Luca Carenzo, Alice Monzani, and Francesco Della Corte. 2025. From prompt to platform: an agentic AI workflow for healthcare simulation scenario design. *Advances in Simulation* 10, 1 (2025), 29.
- [9] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. *arXiv preprint arXiv:2506.18824* (2025).
- [10] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.
- [11] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. 2025. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657* (2025).
- [12] Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).
- [13] Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. 2025. TRAIL: Trace Reasoning and Agentic Issue Localization. *arXiv preprint arXiv:2505.08638* (2025).
- [14] Xiaoting Du, Zhihao Liu, Chenglong Li, Xiangyue Ma, Yingzhuo Li, and Xinyu Wang. 2024. LLM-BRC: A large language model-based bug report classification framework. *Software Quality Journal* 32, 3 (2024), 985–1005.
- [15] Ramtin Ehsani, Sakshi Pathak, and Preetha Chatterjee. 2025. Towards detecting prompt knowledge gaps for improved llm-guided issue resolution. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 699–711.
- [16] Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fournier, Jack Gerrits, Erkang Zhu, and Saleema Amershi. 2025. Interactive debugging and steering of multi-agent ai systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–15.
- [17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
- [18] Kostas Ferles, Jon Stephens, and Isil Dillig. 2021. Verifying correct usage of context-free API protocols. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–30.
- [19] Dhan Prasad Ghale and Mohammad Dabbagh. 2025. Automated Code Comments Generation using Large Language Models: Empirical Evaluation of T5 and BART. *IEEE Access* (2025).
- [20] GitHub. 2023. Commit 36c71abc. <https://github.com/thedigitalworkplace/Autogen/commit/1b8d65df0a54354b5fec152f9aa4162827a7fb2d#diff-5c90ea22e07a2b469f2fa38e46b32d69f19942152caf396628736288971a1ffcR26-R32>. Accessed: 2025-09-11.
- [21] GitHub. 2024. Commit 1b8d65d. <https://github.com/thedigitalworkplace/Autogen/commit/1b8d65df0a54354b5fec152f9aa4162827a7fb2d#diff-5c90ea22e07a2b469f2fa38e46b32d69f19942152caf396628736288971a1ffcR26-R32>

- [22] Google OSV Scanner. 2025. OSV Scanner. <https://github.com/google/osv-scanner>. Accessed: 2025-09-10.
- [23] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594* (2024).
- [24] Junxiao Han, Guanqi Wang, Jiakun Liu, Lingfeng Bao, Xing Hu, Jinling Wei, and Shuguang Deng. 2025. A Comprehensive Study of Bug Characteristics on Foundation Language Models. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 257–268.
- [25] Xue Han and Tingting Yu. 2016. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.
- [26] Chaerim Hong and Taeyeon Oh. 2025. Optimization for threat classification of various data types-based on ML model and LLM. *Scientific Reports* 15, 1 (2025), 22768.
- [27] Soodeh Hosseini and Hossein Seilani. 2025. The role of agentic ai in shaping a smart future: A systematic review. *Array* (2025), 100399.
- [28] Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435* (2024).
- [29] Jerry Huang and Ken Huang. 2025. *AI Agents in Robotics*. Springer Nature Switzerland, Cham, 323–368. doi:10.1007/978-3-031-90026-6_11
- [30] Jerry Huang, Ken Huang, and Chris Hughes. 2025. AI Agents in Offensive Security. In *Agentic AI: Theories and Practices*. Springer, 167–205.
- [31] Ken Huang. 2025. AI Agents in Healthcare. In *Agentic AI: Theories and Practices*. Springer, 303–321.
- [32] Ken Huang. 2025. The Genesis and Evolution of AI Agents. In *Agentic AI: Theories and Practices*. Springer, 1–22.
- [33] Ken Huang and Jerry Huang. 2025. AI Agent Tools and Frameworks. In *Agentic AI: Theories and Practices*. Springer, 23–50.
- [34] Ken Huang, Daniel Wu, Jyoti Ponnappalli, and Grace Huang. 2025. AI Agents in Banking. In *Agentic AI: Theories and Practices*. Springer, 237–277.
- [35] Laurie Hughes, Yogesh K Dwivedi, Tegwen Malik, Mazen Shawosh, Mousa Ahmed Albashrawi, Il Jeon, Vincent Dutot, Mandanna Appanderanda, Tom Crick, Rahul De', et al. 2025. AI agents and agentic systems: A multi-expert analysis. *Journal of Computer Information Systems* (2025), 1–29.
- [36] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1110–1121.
- [37] Rida Ghafoor Hussain, Kin-Choong Yow, and Marco Gori. 2025. Leveraging an enhanced CodeBERT-based model for multiclass software defect prediction via defect classification. *IEEE access* (2025).
- [38] Zak Hussain, Marcel Binz, Rui Mata, and Dirk U Wulff. 2024. A tutorial on open-source large language models for behavioral science. *Behavior Research Methods* 56, 8 (2024), 8214–8237.
- [39] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 510–520.
- [40] Sigma Jahan, Saurabh Singh Rajput, Tushar Sharma, and Mohammad Masudur Rahman. 2025. Taxonomy of Faults in Attention-Based Neural Networks. *arXiv preprint arXiv:2508.04925* (2025).
- [41] Lingxiao Jiang and Zhendong Su. 2007. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. 184–193.
- [42] Weipeng Jiang, Xiaoyu Zhang, Xiaofei Xie, Jiongchi Yu, Yuhan Zhi, Shiqing Ma, and Chao Shen. 2025. The Foundation Cracks: A Comprehensive Study on Bugs and Testing Practices in LLM Libraries. *arXiv preprint arXiv:2506.12320* (2025).
- [43] 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?. In *ICLR*.
- [44] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 1946–1956.
- [45] kaushikb11. 2025. awesome-llm-agents: A curated list of awesome LLM agents frameworks. <https://github.com/kaushikb11/awesome-llm-agents>. GitHub repository, last updated: 2025-09-07.
- [46] Jaehyun Kim, Dongyoung Kim, and Yiming Yang. 2024. Learning to Correct for QA Reasoning with Black-box LLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Miami, Florida, USA, 8916–8937. doi:10.18653/v1/2024.emnlp-main.504
- [47] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.
- [48] Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. 2024. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579* (2024).
- [49] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 241–252. doi:10.1109/ICST49551.2021.00034
- [50] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *CoRR* (2024).
- [51] Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and checking silent semantic violations in large distributed systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 91–107.
- [52] Marcos Medeiros, Uirá Kulesza, Roberta Coelho, Rodrigo Bonifácio, Christoph Treude, and Eiji Adachi Barbosa. 2024. The Impact Of Bug Localization Based on Crash Report Mining: A Developers' Perspective. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 13–24.
- [53] Iraklis Moutidis and Hywel TP Williams. 2021. Community evolution on stack overflow. *Plos one* 16, 6 (2021), e0253010.

- [54] Kaiwen Ning, Jiachi Chen, Jingwen Zhang, Wei Li, Zexu Wang, Yuming Feng, Weizhe Zhang, and Zibin Zheng. 2024. Defining and Detecting the Defects of the Large Language Model-based Autonomous Agents. *CoRR* (2024).
- [55] Razvan Nistor and Leonhard Applis. 2024. What about Haskell Bugs? Adapting bug taxonomies to Haskell's features and community. In *Proceedings of the 36th Symposium on Implementation and Application of Functional Languages*. 38–50.
- [56] Wonseok Oh and Hakjoo Oh. 2024. Towards Effective Static Type-Error Detection for Python. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1808–1820.
- [57] OpenAI. 2025. Introducing OpenAI o3 and o4-mini. <https://openai.com/index/introducing-o3-and-o4-mini/>. Accessed: 2025-09-10.
- [58] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*. 27730–27744.
- [59] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2025. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–28.
- [60] Melissa Z Pan, Mert Cemri, Lakshya A Agrawal, Shuyi Yang, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Kannan Ramchandran, Dan Klein, et al. 2025. Why do multiagent systems fail?. In *ICLR 2025 Workshop on Building Trust in Language Models and Applications*.
- [61] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [62] Alfin Wijaya Rahardja, Junwei Liu, Weitong Chen, Zhenpeng Chen, and Yiling Lou. 2025. Can Agents Fix Agent Issues? *arXiv preprint arXiv:2505.20749* (2025).
- [63] Redis. 2025. Redis – The Real-time Data Platform. <https://redis.io/>. Accessed: 2025-09-10.
- [64] Andrew Rice, Edward Aftandilian, Ciera Jasper, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–22.
- [65] Mehil B Shah, Mohammad Masudur Rahman, and Foutse Khomh. 2025. Towards understanding the impact of data bugs on deep learning models in software engineering. *Empirical Software Engineering* 30, 6 (2025), 168.
- [66] Pranet Sharma, Zhenpeng Shi, Sevval Şimşek, David Starobinski, and David Sastre Medina. 2024. Understanding Similarities and Differences Between Software Composition Analysis Tools. *IEEE Security & Privacy* (2024).
- [67] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.
- [68] Stack Overflow. 2023. Chain of thought prompt using OpenAI to query an order list providing incorrect answer. <https://stackoverflow.com/questions/78389571>. Accessed: 2025-09-11.
- [69] Stack Overflow. 2023. Custom langchain tool not completing agent pipeline. <https://stackoverflow.com/questions/76077652>. Accessed: 2025-09-10.
- [70] Stack Overflow. 2023. how to convert the result from openai call, convert it into json and write to .txt file? <https://stackoverflow.com/questions/78959794>. Accessed: 2025-09-11.
- [71] Stack Overflow. 2023. How to write custom prompt template in LLaMA2 model using LangChain. <https://stackoverflow.com/questions/77536364>. Accessed: 2025-09-11.
- [72] Stack Overflow. 2023. langchain: How to view the context my retriever used when invoke. <https://stackoverflow.com/questions/78322637>. Accessed: 2025-09-10.
- [73] Stack Overflow. 2023. LangChain losing context and timing out when using memory with agent. <https://stackoverflow.com/questions/76146349>. Accessed: 2025-09-10.
- [74] Stack Overflow. 2023. OpenAI API: How do I handle errors in Python? <https://stackoverflow.com/questions/76363168>. Accessed: 2025-09-10.
- [75] Stack Overflow. 2023. OpenAI Chat Completions API: How do I use a function to store conversation memory? <https://stackoverflow.com/questions/76734099>. Accessed: 2025-09-11.
- [76] Stack Overflow. 2023. OpenAI Chat Completions API: Why do I get NULL response? <https://stackoverflow.com/questions/75614444>. Accessed: 2025-09-10.
- [77] Stack Overflow. 2023. OpenAI Chat Completions API: Why does it take so long to get a completion? <https://stackoverflow.com/questions/75987139>. Accessed: 2025-09-10.
- [78] Stack Overflow. 2023. Stop AI from continuing a conversation in a single response. <https://stackoverflow.com/questions/77141533>. Accessed: 2025-09-11.
- [79] Stack Overflow. 2024. How to run async methods in langchain? <https://stackoverflow.com/questions/76621589>. Accessed: 2025-09-10.
- [80] Stack Overflow. 2024. I reinstalled package llmaindex typescript and my document cannot be indexed. <https://stackoverflow.com/questions/7880838>. Accessed: 2025-09-10.
- [81] Stack Overflow. 2024. LangChain failure with long input texts. <https://stackoverflow.com/questions/78885166>. Accessed: 2025-09-10.
- [82] Stack Overflow. 2024. langchain hangs in the middle of execution. <https://stackoverflow.com/questions/79165882>. Accessed: 2025-09-10.
- [83] Stack Overflow. 2024. LLamaindex: How to add new documents to an existing index. <https://stackoverflow.com/questions/79103243>. Accessed: 2025-09-10.

- [84] Stack Overflow. 2024. Streaming function in Microsoft Autogen. <https://stackoverflow.com/questions/77952195>. Accessed: 2025-09-10.
- [85] Stack Overflow. 2024. Why is LlamaCPP freezing during inference? <https://stackoverflow.com/questions/78564699>. Accessed: 2025-09-10.
- [86] Stack Overflow. 2025. Infinite loop in custom ReAct agent using langchain, [Missing 'Action:' after 'Thought:']. <https://stackoverflow.com/questions/79473112>. Accessed: 2025-09-10.
- [87] Stack Overflow. 2025. Memory leak due to accumulating threads after add documents with Pinecone. <https://stackoverflow.com/questions/79577094>. Accessed: 2025-09-10.
- [88] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705.
- [89] Deepak-George Thomas, Matteo Biagiola, Nargiz Humbatova, Mohammad Wardat, Gunel Jahangirova, Hridesh Rajan, and Paolo Tonella. 2025. muPRL: A Mutation Testing Pipeline for Deep Reinforcement Learning Based on Real Faults. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2238–2250.
- [90] Ferdinand Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.
- [91] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.
- [92] Jun Wang, Guanping Xiao, Shuai Zhang, Huashan Lei, Yepang Liu, and Yulei Sui. 2023. Compatibility issues in deep learning systems: Problems and opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 476–488.
- [93] Yuqi Wang, Wei Wang, Qi Chen, Kaizhu Huang, Anh Nguyen, and Suparna De. 2023. Prompt-based zero-shot text classification with conceptual knowledge. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*. 30–38.
- [94] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [95] Cailin Winston and René Just. 2025. A taxonomy of failures in tool-augmented llms. In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 125–135.
- [96] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. doi:10.1145/3715754
- [97] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 801–824.
- [98] Deheng Yang, Kui Liu, Yan Lei, Li Li, Huan Xie, Chunyan Liu, Zhenyu Wang, Xiaoguang Mao, and Tegawendé F Bissyandé. 2024. Demystifying API misuses in deep learning applications. *Empirical Software Engineering* 29, 2 (2024), 45.
- [99] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.
- [100] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.
- [101] Fiorella Zampetti, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta. 2021. Self-admitted technical debt practices: a comparison between industry and open-source. *Empirical Software Engineering* 26, 6 (2021), 131.
- [102] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. 2020. Ubitec: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 221–232.
- [103] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.
- [104] Lianghui Zhu, Xinggang Wang, and Xinlong Wang. [n. d.]. JudgeLM: Fine-tuned Large Language Models are Scalable Judges. In *The Thirteenth International Conference on Learning Representations*.