A Characterization Study of Bugs in LLM Agent Workflow Orchestration Frameworks

Ziluo Xue*†, Yanjie Zhao*†, Shenao Wang†, Kai Chen‡†, and Haoyu Wang‡†
† Huazhong University of Science and Technology, Wuhan, China
{xzl, yanjie_zhao, shenaowang, kchen, haoyuwang}@hust.edu.cn

***Abstract*—Large Language Models (LLMs) have rapidly gained popularity, transforming research and industry. To support their adoption, LLM agent workflow orchestration frameworks (hereinafter referred to as LLM agent frameworks) like LangChain have become essential for building advanced applications. However, their complexity makes bugs inevitable, and these bugs can propagate to downstream applications, causing severe failures or unintended behaviors. In this paper, we first present an abstraction of the structure of mainstream LLM agent frameworks, identifying four key architectural components: data preprocessing, core schema, agent construction, and featured modules. Building on this abstraction, we conduct the first empirical study on LLM agent framework bugs, analyzing 1,026 bug instances extracted from 1,577 real-world bug-related GitHub pull requests (PRs) from three popular LLM agent frameworks: LangChain, LlamaIndex, and Haystack. For each bug, we examine its root cause, symptom, and structural component, providing a systematic taxonomy of nine root causes and six symptom categories. Finally, leveraging the framework structure abstraction and the large-scale empirical study, we perform detailed statistical analysis in terms of the distribution of bugs in different frameworks, the distribution across different framework components, and the relationship between root cause and symptom. The analysis reveals unique challenge patterns compared to traditional software, providing actionable guidance for practitioners on quality assurance.**

***Index Terms*—LLM frameworks, Bug characterization, Empirical study**

I. Introduction

Recent years have witnessed a great popularity of Large Language Models (LLMs) in both industrial and academic fields, such as OpenAI's GPT-4 [1], Meta's Llama [2], and DeepSeek [3]. Applications of LLMs have expanded rapidly, powering developments in areas such as agents [4], [5], content generation [6], [7], knowledge retrieval [8], and decision-making systems [9], [10], [11]. To support this growing ecosystem, the LLM agent workflow orchestration frameworks have emerged (also called LLM agent frameworks [12], [13], [14]), designed to simplify the integration and utilization of LLMs in various workflows. Frameworks like LangChain [15], LlamaIndex [16], Haystack [17] and so on have risen to prominence, offering a wide array of features such as context management, data pipelines, external API integration, and prompt engineering. These tools have become mainstream solutions and foundational components in the LLM ecosystem due to their ease of use and robust capabilities.

However, as these LLM agent frameworks become more sophisticated and encompass a wider range of functionalities, they inevitably introduce bugs and inconsistencies, posing significant challenges for developers striving to build reliable systems. These bugs can manifest in various forms, such as incorrect task execution, failures in external API integrations, or breakdowns in context management, often propagating through the system and causing downstream errors or even system crashes. Such issues not only increase the complexity of debugging and maintenance but also undermine the trust and reliability of applications built on these frameworks, making it difficult to ensure consistent performance.

Faced with the growing complexity and challenges, **there is an urgent need for a systematic understanding of the bugs and inconsistencies inherent to these frameworks**. However, existing research primarily focuses on analyzing bugs in agent applications built on top of these frameworks rather than examining the bugs in the frameworks themselves. For instance, Lin et al. [18] conducted an empirical study on ChatGPT-related projects from GitHub, categorizing them into three types and analyzing user-reported issues such as installation and usage problems. Similarly, Ning et al. [19] examined defects in LLM-based agents, identifying eight types of workflow issues and developing a tool to detect them. While these studies provide valuable insights, they primarily address application-level defects, leaving the bugs and inconsistencies inherent to the framework themselves largely unexplored.

To address these practical challenges, we conduct the first empirical study to understand the bugs in LLM agent frameworks. We first propose a structural abstraction of mainstream frameworks, identifying their key components and workflows. To achieve this, we focus on three widely adopted and representative frameworks: LangChain [20], LlamaIndex [21], and Haystack [22]. These frameworks were chosen for their prominence in the LLM ecosystem, complete design, and active communities. Currently, there is no unified standard for LLM agent frameworks architecture. Nevertheless, by analyzing these frameworks, we identify common patterns and abstract them into four main components: data preprocessing, core schema, agent construction, and featured modules.

* Ziluo Xue and Yanjie Zhao contributed equally to this work.
† The full name of the author's affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.
‡ Kai Chen and Haoyu Wang are the corresponding authors.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00278

Building on the abstraction of LLM agent frameworks, we are able to conduct a systematic bug characterization. To achieve this, we focus on analyzing GitHub pull requests (PRs) from the repositories of the three selected frameworks. These PRs often include detailed descriptions of issues, fixes, and discussions among developers, providing a rich dataset for understanding the nature of bugs in these frameworks. By manually annotating 1,577 bug-related PRs, we classify bugs based on their root causes, identifying the underlying technical factors leading to the defects, and their symptoms, describing how these bugs observably manifest in the framework. Through this process, we develop a detailed taxonomy of nine root cause categories and six symptom categories, enabling us to identify patterns in bug distribution and gain insights into the unique characteristics of each framework. To comprehensively understand the nature of bugs in LLM agent frameworks, it is essential to explore not only their root causes and symptoms but also how these issues manifest across different frameworks and structural components. By mapping bugs to specific architectural elements and identifying recurring patterns, we aim to uncover insights into the vulnerabilities and challenges inherent in these frameworks. In line with this goal, we propose several research questions (RQs) to guide our investigation:

- **RQ1: Root Cause.** What is the root cause of bugs in LLM agent frameworks, and what is the distribution of them across different frameworks and components?
- **RQ2: Symptom.** What is the symptom of bugs in LLM agent frameworks, and what is the distribution of them across different frameworks and components?
- **RQ3: Cause-Symptom Relationship.** How do root causes correspond to symptoms in LLM agent frameworks, and what patterns emerge in their distribution?
- **RQ4: Unique Challenge.** How do LLM agent frameworks exhibit unique problem patterns compared to traditional software systems, and what specialized quality assurance approaches might effectively address novel challenges?

In this paper, we focus on the four RQs to comprehensively characterize bugs in LLM agent frameworks. Our work and findings not only address the critical need for a comprehensive bug characterization in these frameworks, but also provide valuable insights for developers and researchers, helping them anticipate common issues, accelerate debugging, and improve productivity and reduce maintenance costs, leading to more reliable LLM-powered downstream applications.

To summarize, the key contributions are listed as follows:

- **Framework Structural Abstraction.** We propose the first unified structural abstraction for LLM agent frameworks, identifying four core components based on the analysis of three widely used frameworks.
- **Comprehensive Bug Characterization.** We conduct the first large-scale characterization study of 1,026 bugs in three LLM agent frameworks and design a systematic taxonomy to classify them, in terms of nine root cause categories and six symptom categories.
- **Practical Insights.** Through a comprehensive statistical analysis of bug patterns, we derive several challenges in LLM agent frameworks, offering actionable insights to improve framework design, enhance usability, and reduce vulnerabilities.

Artifact Availability. The replication artifact is available at https://github.com/security-pride/Bugs_in_LLM_Frameworks.

II. Background

A. LLM Agent Frameworks

Integrating LLMs into real-world applications requires more than just access to models via APIs like those provided by OpenAI or Hugging Face. Building mature solutions often involves addressing challenges such as handling complex workflows, managing context effectively, and integrating data pipelines seamlessly. To tackle these challenges, the community has developed a range of LLM agent frameworks. These frameworks are designed to simplify and accelerate the development process by providing essential features such as context memory management, workflow orchestration, and data pipeline integration. Furthermore, many frameworks incorporate advanced capabilities like dynamic prompt engineering and indexing strategies, enabling developers to create robust, feature-rich applications that cater to diverse and evolving user needs. By offering these tools and functionalities, LLM agent frameworks serve as a crucial bridge between foundational model APIs and the demands of real-world applications.

To develop an abstraction of LLM framework structure, we conducted extensive research on popular frameworks in the current ecosystem. Table I presents key metrics from their GitHub repositories, highlighting their widespread adoption and community engagement. This analysis involved examining official documentation [20], [21], [22], [23], [24], [25], [26], [27] and GitHub repositories to evaluate their functionality, architectural design, and relevance to current development practices. Among these frameworks, we identified LangChain, LlamaIndex, and Haystack as particularly representative due to their comprehensive architecture, prominence in the ecosystem, and active development communities. These three frameworks demonstrate well-defined and complete architectural designs, covering essential components such as data preprocessing, agent construction, and workflow orchestration, making them ideal for deriving a generalized structural abstraction. They are also widely adopted by developers and organizations, as evidenced by their high GitHub star counts, frequent updates, and extensive use in real-world applications.

Based on our analysis of LangChain, LlamaIndex, and Haystack, we derived a generalized architectural abstraction, as illustrated in Figure 1. This abstraction captures the core design patterns shared across these frameworks, addressing the lack of unified construction standards in the ecosystem. By focusing on commonalities, our abstraction provides a structured perspective for understanding the functionality and addressing challenges in LLM agent frameworks. At its center lies the

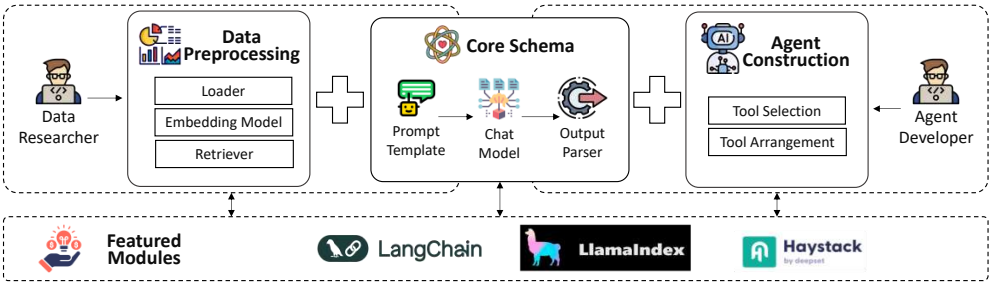

Fig. 1: Abstraction of LLM framework structure.

TABLE I: Statistics of LLM agent frameworks.

Agent Framework	#Star	#Fork	#PR[1]	#Issue	#PR(B)[2]	#Issue(B)
LangChain [15]	103k	16.7k	17,012	7,947	1,455	4,886
LlamaIndex [16]	40k	5.7k	6,793	5,307	-[3]	1916
Haystack [17]	19.8k	2.1k	4,308	3,657	209	722
AutoGPT [28]	173k	45.3k	5,194	2,994	81	364
AutoGen [29]	41.4k	6.2k	2,708	2,108	-	-
outlines [30]	11k	571	602	524	48	207
FlowiseAI [31]	36.2k	18.9k	1,396	1,290	-	197
crewAI [32]	28.4k	3.8k	899	1,089	-	288

[1] PR refers to closed pull request, and the same applies to issue.
[2] PR(B) refers to bug-tagged closed PR, and the same applies to issue.
[3] - indicates either no bug-tagged records exist or the count is negligible.

core schema, which serves as the backbone for coordinating key tasks such as prompt management, model interactions, and result parsing. Surrounding this core are two critical components: the **data preprocessing** component, responsible for transforming raw data into structured formats, and the **agent construction** component, which facilitates the creation of complex workflows for dynamic applications. Additionally, specialized modules such as memory management and advanced indexing extend the architecture to address specific needs. This abstraction offers a foundation for systematic analysis and improvement of LLM agent frameworks, which we detail further in § II-B.

B. Architecture Abstraction of LLM agent frameworks

1) Core Schema: The core schema serves as the foundation for the entire framework, defining the structural and functional elements required for deployment and development of downstream applications. It can be divided into three main parts:

- **Prompt Template:** A structured format for crafting and customizing prompts based on prompt engineering principles [33], [34]. These templates enable dynamic generation of prompts tailored to specific tasks, optimizing accuracy and relevance.
- **Chat Model:** The central engine enabling natural language interactions. Frameworks construct rich interfaces to various external models from specific enterprises and popular open source platforms to support diverse conversational and task-based scenarios.
- **Output Parser:** A module for interpreting and structuring the outputs generated by the LLM, ensuring compatibility with expected formats and downstream applications.

2) Data Preprocessing: This stage is used for managing and transforming raw data into a structured format suitable for downstream processes under the specific framework. It includes the following sub-modules:

- **Loader:** A compatibility module that imports raw data from diverse sources including APIs, databases, and various file formats into the framework.
- **Embedding Model:** A transformation component that converts text data into numerical vectors representing semantic meaning. Frameworks provide interfaces to external embedding models from platforms such as OpenAI and HuggingFace.
- **Retriever:** A selection mechanism that extracts relevant data portions from massive resources based on specific queries or requirements, utilizing built-in algorithms or external support.

Additionally, different frameworks have their own approaches to different stages of data preprocessing, particularly in content processing after loading and vector storage after embedding. LlamaIndex offers unique index scheme so as to organize processed vectors and tokens and enable fast and precise retrieval. LangChain has customized text splitter to empower a strict and standardized data workflow format as well as rich vector store backend support.

3) Agent Construction: As agent [35], [36] becoming one of the most popular applications of LLMs, agent construction is typically the primary goal and core functionality of LLM agent frameworks, requiring comprehensive management of the framework's entire toolchain. The focus lies in enabling intelligent and dynamic utilization of the tools and utilities provided by the framework. This often entails selecting and integrating various algorithms or external tool-selection models tailored to specific tasks or contexts. Through a well-designed decision-making process, agents can intelligently determine which tools or models to invoke, seamlessly adapting to the unique demands of each interaction and application.

4) Featured Module: The featured module emphasizes the current lack of a unified design standard for LLM agent

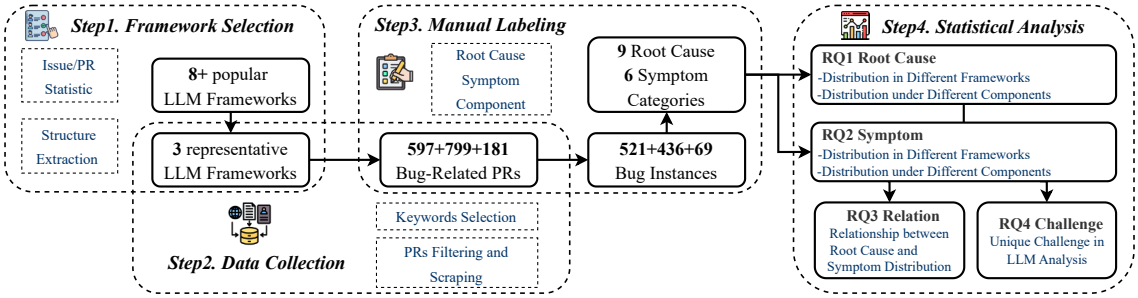

Fig. 2: Overall methodology.

frameworks. Each framework has tailored its unique modules to address specific user needs, enhance developer convenience, and support various use cases. For instance, LangChain's Chain module [37] and Haystack's Pipeline mechanism [38] help standardize and engineer the development process, making it more structured and efficient. LangChain's Memory module [39], on the other hand, provides critical contextual management, supporting the development of advanced chatbots or agents by maintaining coherent, persistent conversation states. Meanwhile, LlamaIndex stands out for its focus on data linking and retrieval [40], providing robust support for applications dealing with source data. Its innovative Node and Index structure efficiently organizes and queries data, catering to the demands of complex data-driven applications.

III. Methodology

After abstracting and understanding the structure of agent frameworks, we designed a comprehensive methodology (as shown in Figure 2) to systematically explore the four RQs.

A. Framework Selection

The framework selection process prioritized frameworks with rich, accessible bug histories and comprehensive architectures. We analyzed community activity metrics from eight popular frameworks listed in Table I, focusing particularly on bug-related issues and PRs as primary sources of authentic bug information. LangChain leads with 1,455 bug-tagged PRs and 4,886 bug-related issues, followed by LlamaIndex with 1,916 bug-related issues, and Haystack with 209 bug-tagged PRs and 722 bug-related issues. While frameworks like AutoGPT have higher star counts (173k), they exhibit significantly fewer bug-related activities (81 PRs, 364 issues), limiting their suitability for comprehensive bug analysis. Based on both quantitative bug data and architectural completeness from our abstraction work, we selected **LangChain**, **LlamaIndex**, and **Haystack** as representative target frameworks for our study.

B. Data Collection

Our methodology is grounded in the analysis of real-world industrial artifacts. The closed PRs from these popular open-source frameworks represent how bugs are actually fixed in practice by the global developer community, making our dataset highly relevant to industrial contexts. To gather relevant PRs, we utilized GitHub's Search API [41] with targeted filtering queries. We constructed queries combining official bug labels (e.g., bug, type:bug) and repository specifications with bug-related keywords (e.g., fix, error, crash) in PR titles and descriptions. For example, to collect data from LangChain, we used the following query:

```
type:pr is:merged repo:langchain
label:bug fix in:title status:success
```

For repositories without consistent bug labeling (e.g., LlamaIndex), we employed keyword-based filtering. This systematic collection process yielded 597 PRs from LangChain, 799 from LlamaIndex, and 181 from Haystack.

C. Manual Labeling

The PRs were analyzed through a collaborative manual annotation process. This was performed by two annotators, both with over three years of professional experience in LLM application development and having actively used at least two of the three selected frameworks in projects. The process focused on three critical dimensions: **Root Cause**, which identifies the underlying technical factors that led to the bug; **Symptom**, which describes how the bug manifests from a user perspective; and **Framework Component**, which specifies which part of the framework architecture contained the bug.

The labeling process followed a rigorous open coding procedure [42]. We first randomly sampled 30% of the collected PRs for pilot labeling to establish initial taxonomies. During this phase, both researchers independently examined each PR's title, description, code changes, and associated discussions to understand the context and nature of the bug. They iteratively refined the taxonomies by grouping similar bugs into categories, with an experienced third researcher serving as arbitrator to resolve any disagreements. For instance, in cases of conflicting classifications, the arbitrator would first determine the bug's primary manifestation to assign it to a most fitting category. If a PR was found to contain genuinely inseparable multiple bugs or was too ambiguous for reliable categorization, it could be excluded from the final analysis to maintain data integrity and the validity of our findings.

Following the pilot phase, the researchers independently labeled the remaining 70% of PRs based on the established taxonomies. This process was conducted in four rounds, with 17-18% of the remaining PRs labeled in each round. We measured inter-rater agreement using Cohen's Kappa coefficient [43] after each round. The initial round yielded a coefficient of approximately 45%, indicating moderate agreement. After thorough discussion of inconsistencies with the arbitrator, the second round showed significant improvement with a coefficient of 78%. By the third and fourth rounds, the coefficient exceeded 92%, demonstrating high reliability in our labeling approach.

Through this rigorous labeling process, we identified and classified 1,026 distinct bug instances across the three frameworks, with 521 from LangChain, 436 from LlamaIndex, and 69 from Haystack. These numbers represent the validated bug instances that remained after careful filtering and analysis of the initially collected 597, 799, and 181 bug-related PRs respectively. Each bug instance was thoroughly characterized according to our iteratively established taxonomies of nine root cause categories and six symptom categories.

D. Statistical Analysis

Building on our dataset of 1,026 bug instances (521 from LangChain, 436 from LlamaIndex, and 69 from Haystack) categorized according to nine root cause types and six symptom categories, we conducted statistical analysis to address our research questions. We analyzed root cause distributions across different frameworks and their components (**RQ1**), investigated symptom manifestation patterns within various framework contexts (**RQ2**), explored the relationships between specific root causes and their corresponding symptoms (**RQ3**), and identified unique challenges in LLM framework bug analysis compared to traditional software systems (**RQ4**).

IV. RQ1: Root Cause

A. Root Cause Taxonomy

In our study, we classified 1,026 bug instances from three frameworks into nine root cause categories as shown in Figure 3. Below, each category is briefly described, with its aggregated bug count and its percentage share of the total. We first focus on the sum data of the bug instances from three frameworks and will discuss their distribution across the frameworks in § IV-B.

R1: API Misuse (24.7%). This category covers cases where APIs are used incorrectly, including missing required API calls, invoking the wrong API, or making redundant calls. Errors often stem from misunderstandings of the framework's interfaces or misinterpreting API documentation. In LangChain, LlamaIndex, and Haystack, the numbers are 139, 112, and 3 respectively, yielding a total of 254 bug instances. This accounts for approximately 24.7% of all bugs, making API Misuse one of the most prevalent issues that directly affects performance and system stability.

R2: Incompatibility (5.5%). Incompatibility issues include both functional incompatibility—where internal modules or external libraries have conflicting behaviors—and version incompatibility, which arises when dependency updates are not synchronized with framework support. With 36, 15, and 5 instances in the three frameworks respectively, the total comes to 56 instances, accounting for about 5.5% of the total. Although not the largest share, these issues underscore the difficulties in maintaining consistent dependency management in rapidly evolving software ecosystems.

R3: Assignment Issue (4.0%). Assignment Issues occur when variables, parameters, or data structures are assigned incorrect values or types, often due to logic flaws or improper initialization. The counts for this category are 22, 19, and 0 respectively, summing to 41 instances (roughly 4.0% of all bugs). While fewer in number, these errors indicate potential pitfalls in data initialization and variable management that can have cascading effects on program execution.

R4: Parameter/Argument Issue (15.6%). This category addresses problems caused by incorrect, incomplete, or conflicting parameters when invoking APIs. Errors here may be due to values outside acceptable ranges or failure to supply necessary arguments. With 96, 59, and 5 instances respectively, Parameter/Argument issues total 160 bugs, representing about 15.6% of all instances. This demonstrates the need for better input validations and clearer API documentation.

R5: Code Logic Issue (38.0%). Code Logic Issues arise from deviations in the intended design or workflow, such as mistakes in the order of method calls, missing conditional branches, or flawed algorithm design. The three frameworks contribute 163, 182, and 45 instances, respectively, amounting to 390 instances in total. Representing 38.0% of all bugs, Code Logic Issues are the most significant category, highlighting that ensuring logical correctness and robust algorithmic design is crucial for overall framework stability.

R6: Import Error (3.5%). Problems during the importation of external or internal modules—including missing, incorrect, or redundant imports—fall into the Import Error category. Recorded instances are 19, 13, and 4, which sums to 36 bugs (about 3.5% of all instances). Despite the lower frequency, these errors can critically undermine functionality if dependencies are not managed successfully.

R7: Typo (4.7%). Typographical errors, such as mistakes in syntax, variable names, or function calls, are cataloged under the Typo category. The counts of this category are 27, 19, and 2, respectively, leading to 48 instances (approximately 4.7% of the total). While seemingly minor, typos can cause unexpected behavior or immediate failures that disrupt development.

R8: Incorrect Exception Handling (3.0%). This category includes errors where exceptions are either not caught when necessary or are mishandled, resulting in misleading error messages or silent failures. The three frameworks record 14, 14, and 3 instances respectively, totaling 31 instances (about 3.0%). Correctly managing exceptions is critical, as these issues can lead to system crashes or obscure bugs that are difficult to diagnose.

R9: Incorrect Numerical Computation (1.0%). Errors in mathematical operations, such as rounding mistakes or miscal-

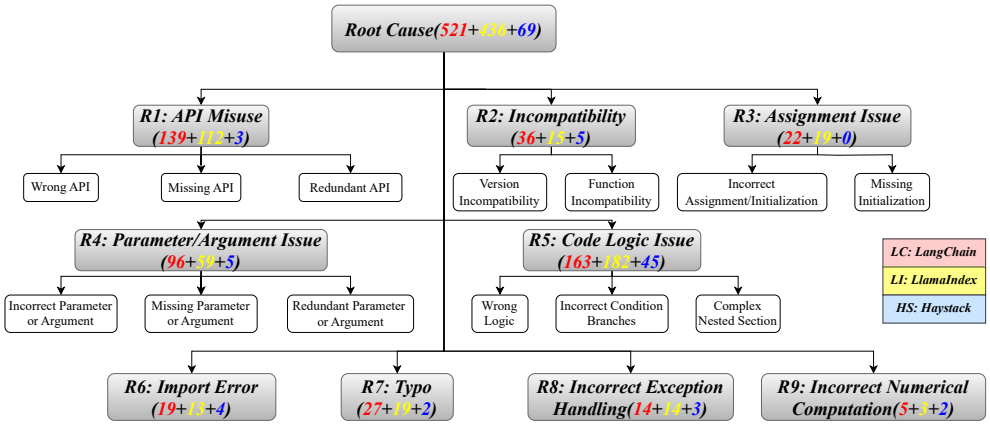

Fig. 3: Root cause taxonomy (R1–R9) of bugs in LLM agent frameworks, with data collected from LangChain (LC), LlamaIndex (LI), and Haystack (HS).

culations due to precision issues, are covered by this category. With counts of 5, 3, and 2 across the frameworks, this yields a total of 10 instances, representing just about 1.0% of all bugs. Although less frequent, these errors can severely affect model accuracy and application reliability when they occur.

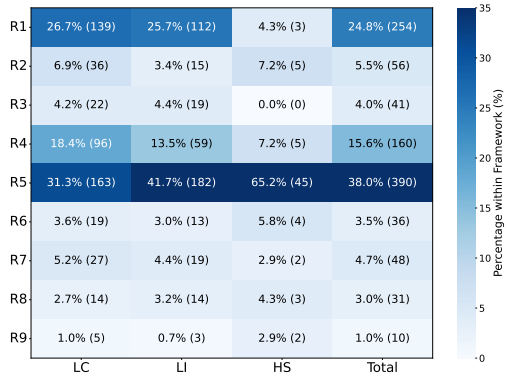

Fig. 4: Root cause distribution in different frameworks.

B. Root Cause Distribution in Different Frameworks

We analyze the distribution of bug instances by root cause within each framework, breaking down instances per category and their percentages relative to each framework's total bug count. This approach summarizes underlying causes of observed differences based on each framework's unique features and offers insights into potential areas for enhancing design robustness. As shown in Figure 4, the first three columns represent root cause distribution for each framework while the last column shows the total distribution.

In **LangChain**, a total of 521 bug instances were recorded. It shows a lower percentage of internal logic issues (31.3%) compared to the overall average (38%). However, its API misuse (26.7%) and parameter errors (18.4%) are slightly elevated. This points to a framework with a modular design that, while adaptable, increases the likelihood of user misconfigurations.

LlamaIndex records 436 bug instances in total. It places a heavier burden on internal logic (41.7%) compared to the overall 38%. Its share of API misuse (25.7%) is nearly on par with the overall distribution, while parameter errors fall slightly below (13.5%). This distribution highlights that the sophisticated design of its indexing mechanisms contributes to a greater prevalence of internal logic errors. Therefore, ensuring the robustness and maintainability of these complex internal systems, rather than mitigating user-facing errors, seems to be the central concern for LlamaIndex's development.

Haystack, with a total of 69 bug instances, shows a markedly different data distribution. Haystack is characterized by a dominant share of internal logic errors, which account for 65.2%—a figure that far exceeds the overall average. In contrast, both API misuse (4.3%) and parameter errors (7.2%) are significantly lower than the aggregated percentages. This pattern aligns with Haystack's design focus on maintaining a simple user interface that minimizes user-induced errors while its core logic still accounts for a large portion of the vulnerabilities. However, note that Haystack's smaller sample size (69 bugs) may contribute to statistical variations, potentially affecting the high percentage of internal logic issues.

C. Root Cause Distribution across Components

Based on the heatmap analysis shown in Figure 5, we focus on comparing each component's root cause distribution (target component column) against the overall distribution (Total column) to identify significant deviations. These variations often reveal component-specific challenges aligned with their architectural roles.

The **Core Schema** component shows significantly higher Parameter/Argument Issues (20.6%) compared to the overall distribution, suggesting schema definition activities face particular challenges with parameter handling and validation.

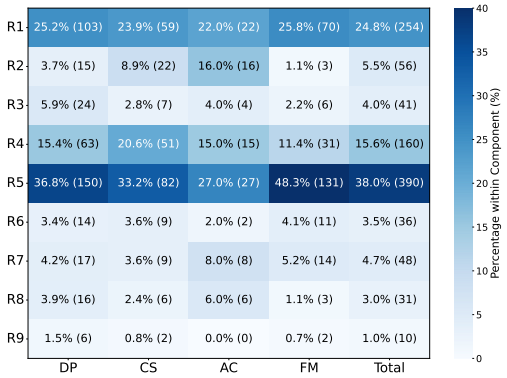

Fig. 5: Root cause distribution across components.

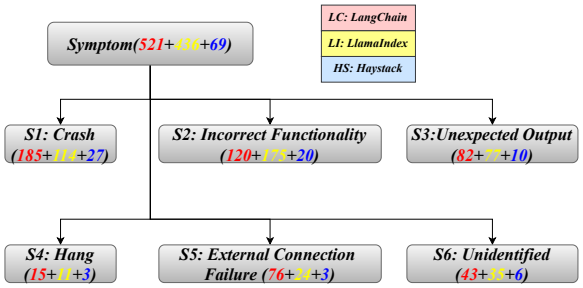

Fig. 6: Symptom categorization.

This aligns with the component's fundamental role in defining structural interfaces across the framework. The **Agent Construction** component demonstrates a dramatically higher occurrence of Incompatibility issues (16.0%) than found overall, likely reflecting the integration challenges inherent when composing agents from multiple components. The component also shows elevated Incorrect Exception Handling (6.0%), pointing to difficulties managing error conditions during agent assembly. The **Featured Module** component exhibits Code Logic Issues (48.3%) at rates notably higher than the framework average, indicating that implementing specialized functionality introduces more complex logical challenges. This suggests that extending the framework with custom features requires particularly careful attention to algorithmic correctness. The **Data Preprocessing** component's error distribution largely mirrors the overall framework pattern, with only minor deviations, suggesting these issues manifest consistently across preprocessing operations and don't represent component-specific challenges.

Answer to RQ1: *Code logic errors (38.0%) and API misuse (24.7%) are the primary root causes of bugs in LLM agent frameworks. LlamaIndex has more internal logic issues, while Haystack is dominated by core processing errors. Schema components often face parameter validation issues, and Featured Modules are prone to logical complexity.*

V. RQ2: Symptom

A. Symptom Categories

In our analysis of 1,026 bugs across three frameworks, we identified six distinct symptom categories that characterize how bugs manifest in LLM agent frameworks. Each category represents a specific way in which framework failures become observable to developers and users. The categories and instance counts are shown in Figure 6.

S1: Crash (31.8%). Crash is one of the most prevalent and diverse symptoms observed in LLM agent frameworks. Broadly speaking, any interruption of execution during runtime, whether accompanied by explicit exception throwing or a sudden termination of the process, falls under this category. In the three frameworks, we observed 185, 114, and 27 instances respectively, totaling 326 cases (31.8% of all bugs). This high percentage underscores how framework instability frequently manifests as complete execution failure.

S2: Incorrect Functionality (30.7%). This category refers to situations where the program runs without crashing or hanging but fails to perform as intended. This includes cases where behaviors deviate from expected results, or the system violates predefined constraints. With 120 instances in LangChain, 175 in LlamaIndex, and 20 in Haystack, this accounts for 315 cases (30.7%). Such issues often arise from logic errors, improper handling of edge cases, or unintended side effects of code changes, requiring careful debugging to identify root causes.

S3: Unexpected Output (16.5%). While technically a subset of incorrect functionality, unexpected output is treated as a distinct classification due to its significance in LLM agent framework bug detection. This refers to cases where the program produces results that deviate from expectations but without a predictable cause tied to incorrect functionality. We identified 82, 77, and 10 instances across the frameworks, totaling 169 cases (16.5%). These outputs often serve as initial indicators of bugs in complex systems, preceding the identification of specific functional errors.

S4: Hang (2.8%). Hanging symptoms occur when a program fails to produce any output or proceed as expected, yet does not terminate. The program remains in a stalled or idle state, often without clear feedback or indication of progress. With only 15, 11, and 3 instances across the frameworks, this is the least common symptom (2.8%). Hanging can be caused by insufficient exception handling, unresponsive external dependencies, or algorithmic issues like deadlocks or infinite loops, typically requiring manual intervention to resolve.

S5: External Connection Failure (10.0%). This symptom often occurs during interactions with external libraries, services, or resources. It includes cases where frameworks successfully connect to external components but trigger exception handling mechanisms, as well as failures to establish connections due to version incompatibilities or misconfigurations. LangChain exhibited 76 such cases, LlamaIndex had 24, and Haystack had 3, summing to 103 instances (10.0%). This significant percentage highlights the integration challenges these frameworks face with external dependencies.

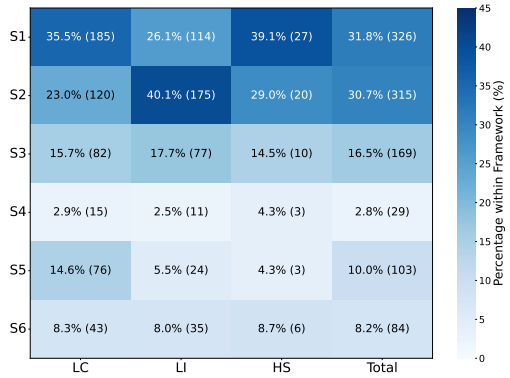

Fig. 7: Symptom distribution in different frameworks.

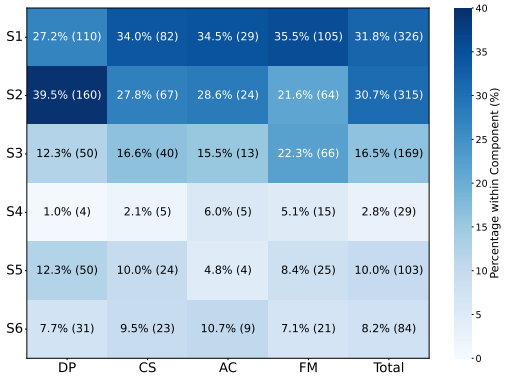

Fig. 8: Symptom distribution across structural components.

S6: Unidentified (8.2%). PRs without related issues sometimes make it difficult to predict the symptoms caused by the introduced bugs. This often happens because changes are either too abstract or have a broad scope of application. We encountered 43 such cases in LangChain, 35 in LlamaIndex, and 6 in Haystack, totaling 84 instances (8.2%). This category reveals areas where bug reporting practices could improve to better capture failure characteristics.

B. Symptom Distribution in Different Frameworks

This section examines how bug symptoms manifest differently across the three different frameworks. The distribution, instances number, and the percentage are shown in Figure 7. By analyzing the symptom patterns unique to each framework, we gain insights into their architectural strengths and weaknesses, as well as potential areas for improvement.

LangChain exhibits a higher proportion of Crashes (35.5% compared to the overall 31.8%) and External Connection Failures (14.6% compared to the overall 10.0%), while showing fewer Incorrect Functionality issues (23.0% compared to the overall 30.7%). This distinct pattern reflects LangChain's architecture as an integration framework that orchestrates numerous external services and APIs, creating more opportunities for connection failures and execution interruptions when component communications break down.

LlamaIndex's significant deviation from overall patterns lies in its much higher rate of Incorrect Functionality issues (40.1% while the overall average is 30.7%) and substantially lower incidence of External Connection Failures (5.5% against the overall 10.0%). This distribution aligns with LlamaIndex's focus on complex document indexing and retrieval operations, where intricate data processing and query handling mechanisms often continue execution despite errors, producing incorrect results rather than terminating outright.

Haystack shows the highest rate of Crashes among all frameworks (39.1% whereas the overall average is 31.8%) and the lowest rate of External Connection Failures (4.3% compared to the overall 10.0%). This symptom profile corresponds to Haystack's pipeline-based architecture, which processes data through sequential components. When errors occur in this linear workflow, they more frequently result in execution termination rather than continued processing with incorrect results, explaining both the higher crash rate and lower incorrect functionality compared to other frameworks.

C. Symptom Distribution across Components

Based on the heatmap shown in Figure 8, we compare each component's column against the Total column to identify significant deviations that reveal component-specific patterns.

The **Data Preprocessing** component shows notably higher Incorrect Functionality (39.5%) compared to the overall average (30.7%), but lower Crash rates (27.2% compared to 31.8% overall). This suggests that data preprocessing operations tend to complete execution but produce incorrect results rather than failing completely, highlighting challenges in data transformation logic. The **Core Schema** component exhibits a balanced distribution across symptom types, with slightly elevated Crash rates (34.0%) and reduced Incorrect Functionality issues (27.8%). This pattern aligns with its role as a foundational framework element where structural failures are more likely to cause complete crashes. The **Agent Construction** component demonstrates distinctly higher Hang issues (6.0% compared to the framework average of 2.8%) and lower External Connection Failure rates (4.8% compared to 10.0% overall). This indicates that agent assembly processes are particularly susceptible to deadlocks or infinite loops during component integration. The **Featured Module** component shows the highest Crash rate (35.5%) and Unexpected Output issues (22.3% while the overall is 16.5%), coupled with the lowest Incorrect Functionality (21.6%). This suggests that specialized modules either fail completely or produce surprising results rather than subtly malfunctioning, possibly due to their complex feature implementations.

> ***Answer to RQ2:*** *Crashes (31.8%) and incorrect functionality (30.7%) are the most common symptoms, with LangChain prone to connection failures, LlamaIndex to functionality issues, and Haystack to crashes. Data preprocessing often yields incorrect results, while featured modules are prone to failures and unexpected outputs.*

VI. RQ3: Cause-Symptom Relationship

The relationship between root causes and symptoms, as illustrated by Figure 9, highlights how specific root causes contribute to a wide range of symptoms while certain symptoms are concentrated around particular causes.

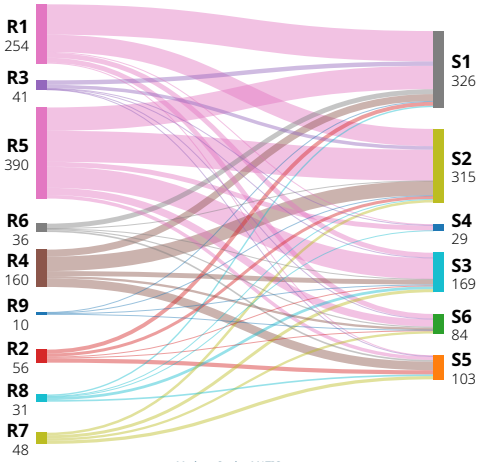

Fig. 9: Relationship between root cause and symptom.

From the root cause perspective, Code Logic Issue (390 occurrences, 38.0% of all issues) emerges as the most pervasive problem, primarily manifesting as Incorrect Functionality (133) and Crash (100). This dominant cause affects all symptom categories, demonstrating how logical flaws can compromise system reliability at multiple levels. API Misuse (254 occurrences) shows a concentrated impact pattern, strongly associated with Crash (130) - accounting for nearly 40% of all system crashes. This suggests improper API usage frequently leads to catastrophic system failures rather than subtle malfunctions. Parameter/Argument Issue (160 occurrences) displays a specialized pattern, predominantly linked to Incorrect Functionality (62) and External Connection Failure (37), demonstrating how parameter configuration errors primarily disrupt system integration and expected behavior.

From the symptom perspective, Crash (326 occurrences) is overwhelmingly caused by two issues: API Misuse (130) and Code Logic Issue (100), which together account for over 70% of all crashes. This clear attribution pattern can make crash diagnostics relatively straightforward. Incorrect Functionality (315 occurrences) shows a more distributed cause pattern, with Code Logic Issue (133), API Misuse (74), and Parameter/Argument Issue (62) as major contributors. This diversity explains why functional issues are often challenging to diagnose. External Connection Failure (103 occurrences) has a distinctive cause profile, with Parameter/Argument Issue (37) as its leading cause, followed by Incompatibility (18). This specific pattern indicates that integration problems frequently stem from argument handling and compatibility issues.

The data reveals that while some symptoms (like Crash) have clear primary causes, others (like Incorrect Functionality) arise from multiple root issues. Similarly, certain root causes have widespread impacts (Code Logic Issue), while others produce more targeted effects (Parameter/Argument Issue). These insights suggest that debugging strategies should be tailored to address both systemic issues affecting overall stability and specialized problems disrupting specific functionalities.

> ***Answer to RQ3:*** *Code logic issues (40%) impact all symptom categories, while API misuse shows concentrated effects. Crashes have clear primary causes, whereas incorrect functionality arises from diverse causes, making it harder to diagnose. This cause-symptom map can serve as a practical debugging "cheat sheet" for development teams.*

VII. RQ4: Unique Challenges

Our empirical investigation reveals distinctive patterns and challenges in LLM agent frameworks that distinguish them from traditional software systems. These unique characteristics not only influence how bugs manifest but also suggest the need for specialized quality assurance approaches tailored to the probabilistic and interconnected nature of these frameworks.

Novel Bug Patterns in LLM agent frameworks. The symptom distribution shows a notable departure from conventional patterns—while crashes typically dominate traditional software studies, our analysis reveals that Incorrect Functionality (315 occurrences) nearly matches Crash (326 occurrences) in frequency. Moreover, we identified Unexpected Output as a significant new symptom category (169 occurrences, 16.5%) uniquely relevant to LLM agent frameworks, reflecting their probabilistic and generative nature. The root cause analysis further highlights framework-specific challenges. Incompatibility issues (56 occurrences) and their frequent connection to External Connection Failure (103 occurrences) underscore the highly interconnected nature of LLM agent frameworks. Unlike monolithic systems, these frameworks extensively integrate with diverse third-party components, APIs, and services, creating a complex dependency network that introduces unique failure points. As frameworks attempt to abstract away the complexity of underlying LLM infrastructure, developers often encounter unexpected behaviors at integration boundaries. The prevalence of Parameter/Argument Issues (160 occurrences) leading to external connection failures further illustrates how the intricate parameterization required by LLM operations creates distinctive challenges for developers.

Challenges and Opportunities in Adapting Quality Assurance Approaches. These unique characteristics necessitate a fundamental reconceptualization of verification paradigms when applied to LLM orchestration frameworks. Our root cause analysis reveals emergent failure modes that transcend traditional quality assurance boundaries. **Conventional testing techniques [44], [45] are inherently ill-equipped for LLM agent frameworks as they presuppose deterministic outcomes within discrete state spaces, whereas LLM-driven systems operate in continuous probability distributions with non-linear compositional effects.** This knowledge mismatch becomes particularly acute in frameworks where stochastic foundation model outputs propagate through com-

positional architectures, creating complex interaction spaces resistant to boundary-case analysis.

The disproportionate occurrence of API misuse and logic vulnerabilities in our dataset signifies an imperative for metamorphic testing oracles capable of verifying semantic invariants across heterogeneous component interfaces. Traditional unit testing paradigms cannot capture emergent properties at integration boundaries, such as semantic drift between retrieval systems and prompt engineering, resulting in coherent but functionally divergent behaviors. Future verification frameworks could focus on detecting cross-component logical inconsistencies, particularly information flow anomalies and subtle compositional incompatibilities that manifest only during full-system execution. Static analysis approaches [46], [47] conceived for deterministic software ecosystems lack the semantic awareness to identify orchestration-specific vulnerabilities, particularly concerning parameter sensitivity cascades across component boundaries. Our findings underscore the necessity for framework-cognizant analysis systems that can reason about semantic equivalence across components and model propagation effects through orchestration pathways.

Answer to RQ4: *LLM agent frameworks face unique challenges, with Unexpected Output as a new symptom and Incorrect Functionality nearly as frequent as Crashes, reflecting their probabilistic nature and complex integrations. These traits render traditional testing methods insufficient, requiring specialized approaches for probabilistic outputs and cross-component interactions.*

VIII. Discussion

Implications. Our study provides valuable insights for diverse stakeholders. For developers, prevalent issues like code logic errors and API misuse highlight the need for robust validation, better error handling, and targeted testing. Component-specific bug mapping suggests allocating resources to critical bottlenecks, while improvements in API documentation, input validation, and automated dependency checks can mitigate parameter and compatibility issues. For practitioners, our findings offer guidance on framework selection and workflow design, helping users address known vulnerabilities—such as parameter configuration in LangChain or workflow optimization in LlamaIndex. For academic researchers, this work establishes a methodology for analyzing LLM framework bugs and exploring architecture-failure relationships. The taxonomy of root causes and symptoms lays a foundation for research into debugging strategies and automated detection, fostering collaboration between software engineering and natural language processing to develop robust, user-friendly frameworks.

Threats to Validity. Several threats to validity should be acknowledged in this study. First, the dataset's timeliness may affect the relevance of the findings, as the collected GitHub PRs reflect the state of the selected frameworks at a specific point in time. Framework updates or newer versions may introduce different issues or resolve identified bugs. Second, the reliability of manual labeling poses a potential risk, as the categorization of root causes and symptoms depends on subjective interpretation, which could introduce biases or errors. Third, the limited diversity of frameworks analyzed may restrict the generalizability of the findings to other LLM agent frameworks with different architectures or use cases. Future work could address these limitations by incorporating a broader range of frameworks and automating parts of the labeling process to improve consistency and scalability.

IX. Related Works

Empirical Study of Bugs. While no direct research exists on bug analysis in LLM agent frameworks, extensive studies on various systems provide valuable insights. The most related works focus on bug characterization in deep learning (DL) frameworks [48], [49], [50], [51]. These studies analyzed bugs in TensorFlow, JavaScript-based DL systems, and multiple DL frameworks, identifying root causes, symptoms, and debugging strategies while constructing taxonomies and fix patterns. Additional empirical studies examined bugs in broader DL environments and applications [52], [53], [54], [55], while machine learning systems [56], [57], [58] and general software systems [59], [60], [61] have also provided valuable references for categorizing bugs by root causes and symptoms.

LLM Agent Defects. Recent studies have begun to explore the unique defects and vulnerabilities in LLM-based agent systems [62], [63]. Ning et al. [19] conducted the first systematic analysis of LLM-based agents, identifying eight defect types, including tool invocation failures and task execution errors. They developed Agentable to detect these defects, revealing prevalent issues like service interruptions and incorrect outputs in real-world projects. Cemri et al. [64] introduced MAST (Multi-Agent System Failure Taxonomy), which categorizes failures in multi-agent systems into specification issues, inter-agent misalignments, and task verification errors. Through their analysis of seven frameworks and over 200 tasks, they highlighted 14 failure modes, emphasizing the challenges of designing reliable multi-agent systems. Additionally, Zhang et al. [65] proposed automated failure attribution methods for multi-agent systems, introducing the Who&When dataset to analyze failures at the agent and step levels. While these works focus on identifying defects at the application level of LLM-based agents, our study shifts the focus to the underlying frameworks that support these agents.

X. Conclusion

In this paper, we conducted the first comprehensive characterization of bugs in LLM agent frameworks. We proposed a structural abstraction identifying key architectural components across popular frameworks and investigated bugs through PRs from three widely-used systems. We established taxonomies for root causes and symptoms, mapped their distributions, and identified relationships between causes and manifestations. Our findings revealed unique challenge patterns in LLM agent frameworks differing from traditional software systems, highlighting the need for specialized quality assurance.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (grants No. 62572209, 62502168) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

REFERENCES

[1] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774

[2] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," *CoRR*, vol. abs/2302.13971, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.13971

[3] DeepSeek-AI, "Deepseek-v3 technical report," *CoRR*, vol. abs/2412.19437, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2412.19437

[4] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 2024, pp. 8048–8057. [Online]. Available: https://www.ijcai.org/proceedings/2024/890

[5] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, "A survey on large language model based autonomous agents," *Frontiers Comput. Sci.*, vol. 18, no. 6, p. 186345, 2024. [Online]. Available: https://doi.org/10.1007/s11704-024-40231-1

[6] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Trans. Software Eng.*, vol. 50, no. 9, pp. 2254–2268, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3428972

[7] J. Y. Koh, D. Fried, and R. Salakhutdinov, "Generating images with multimodal language models," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/43a69d143273bd8215578bde887bb552-Abstract-Conference.html

[8] X. Long, J. Zeng, F. Meng, Z. Ma, K. Zhang, B. Zhou, and J. Zhou, "Generative multi-modal knowledge retrieval with large language models," in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, M. J. Wooldridge, J. G. Dy, and S. Natarajan, Eds. AAAI Press, 2024, pp. 18 733–18 741. [Online]. Available: https://doi.org/10.1609/aaai.v38i17.29837

[9] Y. Yu, Z. Yao, H. Li, Z. Deng, Y. Jiang, Y. Cao, Z. Chen, J. W. Suchow, Z. Cui, R. Liu, Z. Xu, D. Zhang, K. Subbalakshmi, G. Xiong, Y. He, J. Huang, D. Li, and Q. Xie, "Fincon: A synthesized LLM multi-agent system with conceptual verbal reinforcement for enhanced financial decision making," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/f7ae4fe91d96f50abc2211f09b6a7e49-Abstract-Conference.html

[10] M. Mozikov, N. Severin, V. Bodishtianu, M. Glushanina, I. Nasonov, D. Orekhov, P. Vladislav, I. Makovetskiy, M. Baklashkin, V. Lavrentyev, A. Tsvigun, D. Turdakov, T. Shavrina, A. Savchenko, and I. Makarov, "EAI: emotional decision-making of llms in strategic games and ethical dilemmas," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/611e84703eac7cc03f78339df8aae2ed-Abstract-Conference.html

[11] Y. Kim, C. Park, H. Jeong, Y. S. Chan, X. Xu, D. McDuff, H. Lee, M. Ghassemi, C. Breazeal, and H. W. Park, "Mdagents: An adaptive collaboration of llms for medical decision-making," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/90d1fc07f46e31387978b88e7e057a31-Abstract-Conference.html

[12] kaushikb11, "Awesome LLM Agent Frameworks," https://github.com/kaushikb11/awesome-llm-agents, Accessed May 31, 2025.

[13] S. Wang, Y. Zhao, X. Hou, and H. Wang, "Large language model supply chain: A research agenda," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, May 2025. [Online]. Available: https://doi.org/10.1145/3708531

[14] S. Wang, Y. Zhao, Z. Liu, Q. Zou, and H. Wang, "Sok: Understanding vulnerabilities in the large language model supply chain," *CoRR*, vol. abs/2502.12497, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2502.12497

[15] H. Chase, "LangChain," Oct. 2022. [Online]. Available: https://github.com/langchain-ai/langchain

[16] J. Liu, "LlamaIndex," 11 2022. [Online]. Available: https://github.com/jerryjliu/llama_index

[17] M. Pietsch, T. Möller, B. Kostic, J. Risch, M. Pippi, M. Jobanputra, S. Zanzottera, S. Cerza, V. Blagojevic, T. Stadelmann, T. Soni, and S. Lee, "Haystack: the end-to-end NLP framework for pragmatic builders," Nov. 2019. [Online]. Available: https://github.com/deepset-ai/haystack

[18] Z. Lin, N. Zhang, C. Liu, and Z. Zheng, "An empirical study of chatgpt-related projects and their issues on github," *Expert Syst. Appl.*, vol. 267, p. 126113, 2025. [Online]. Available: https://doi.org/10.1016/j.eswa.2024.126113

[19] K. Ning, J. Chen, J. Zhang, W. Li, Z. Wang, Y. Feng, W. Zhang, and Z. Zheng, "Defining and detecting the defects of the large language model-based autonomous agents," *CoRR*, vol. abs/2412.18371, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2412.18371

[20] H. Chase, "LangChain Documentation," https://python.langchain.com/docs/introduction/, Oct. 2022.

[21] J. Liu, "LlamaIndex Documentation," https://docs.llamaindex.ai/en/latest/, 11 2022.

[22] M. Pietsch, T. Möller, B. Kostic, J. Risch, M. Pippi, M. Jobanputra, S. Zanzottera, S. Cerza, V. Blagojevic, T. Stadelmann, T. Soni, and S. Lee, "Haystack Documentation," https://docs.haystack.deepset.ai/docs/intro, Nov. 2019.

[23] S. Gravitas, "AutoGPT Documentation," https://docs.agpt.co/, Accessed March 14, 2025.

[24] Microsoft, "AutoGen Documentation," https://microsoft.github.io/autogen/stable/, Accessed March 14, 2025.

[25] dottxt ai, "outlines Documentation," https://dottxt-ai.github.io/outlines/latest/reference/chat_templating/, Accessed March 14, 2025.

[26] FlowiseAI, "FlowiseAI Documentation," https://docs.flowiseai.com/, Accessed March 14, 2025.

[27] crewAI, "crewAI Documentation," https://docs.flowiseai.com/, Accessed March 14, 2025.

[28] Significant Gravitas, "AutoGPT," 2023. [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT

[29] Microsoft, "AutoGen," 2023. [Online]. Available: https://github.com/microsoft/autogen

[30] dottxt-ai, "outlines," 2023. [Online]. Available: https://github.com/dottxt-ai/outlines

[31] FlowiseAI, "FlowiseAI," 2023. [Online]. Available: https://github.com/FlowiseAI/Flowise

[32] crewAI, "crewAI," 2023. [Online]. Available: https://github.com/crewAIInc/crewAI

[33] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *CoRR*, vol. abs/2402.07927, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.07927

[34] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *Data Intelligence and Cognitive Informatics*, I. J. Jacob, S. Piramuthu, and P. Falkowski-Gilski, Eds. Singapore: Springer Nature Singapore, 2024, pp. 387–402.

[35] X. Huang, W. Liu, X. Chen, X. Wang, H. Wang, D. Lian, Y. Wang, R. Tang, and E. Chen, "Understanding the planning of llm agents: A survey," *CoRR*, vol. abs/2402.02716, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.02716

[36] X. Li, S. Wang, S. Zeng, Y. Wu, and Y. Yang, "A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges," *Vicinagearth*, vol. 1, no. 1, p. 9, Oct. 2024.

[37] H. Chase, "LangChain Documentation of Chain Mechanism," https://python.langchain.com/docs/how_to/sequence/, Oct. 2022.

[38] M. Pietsch, T. Möller, B. Kostic, J. Risch, M. Pippi, M. Jobanputra, S. Zanzottera, S. Cerza, V. Blagojevic, T. Stadelmann, T. Soni, and S. Lee, "Haystack Documentation of Pipeline," https://docs.haystack.deepset.ai/docs/pipelines, Nov. 2019.

[39] H. Chase, "LangChain Documentation of Memory Module," https://python.langchain.com/docs/how_to/chatbots_memory/, Oct. 2022.

[40] R. K. Malviya, V. Javalkar, and R. Malviya, "Scalability and Performance Benchmarking of LangChain, LlamaIndex, and Haystack for Enterprise AI Customer Support Systems," in *IJGIS Fall of 2024 Conference*. The New World Foundation, sep 8 2024, https://ijgis.pubpub.org/pub/6yecqicl.

[41] GitHub, "GitHub Search API Documentation," https://docs.github.com/en/search-github/searching-on-github/searching-issues-and-pull-requests#search-only-issues-or-pull-requests, Accessed March 14, 2025.

[42] C. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[43] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. [Online]. Available: https://doi.org/10.1177/001316446002000104

[44] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 474–494, 2010.

[45] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3363562

[46] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, 2008. [Online]. Available: https://doi.org/10.1109/MS.2008.130

[47] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[48] Y. Zhang, Y. Chen, S. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 129–140. [Online]. Available: https://doi.org/10.1145/3213846.3213866

[49] L. Quan, Q. Guo, X. Xie, S. Chen, X. Li, and Y. Liu, "Towards understanding the faults of javascript-based deep learning systems," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 105:1–105:13. [Online]. Available: https://doi.org/10.1145/3551349.3560427

[50] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, pp. 135:1–135:31, 2023. [Online]. Available: https://doi.org/10.1145/3587155

[51] Y. Yang, T. He, Z. Xia, and Y. Feng, "A comprehensive empirical study on bug characteristics of deep learning frameworks," *Inf. Softw. Technol.*, vol. 151, p. 107004, 2022. [Online]. Available: https://doi.org/10.1016/j.infsof.2022.107004

[52] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 510–520. [Online]. Available: https://doi.org/10.1145/3338906.3338955

[53] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1110–1121. [Online]. Available: https://doi.org/10.1145/3377811.3380395

[54] Q. Shen, H. Ma, J. Chen, Y. Tian, S. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 968–980. [Online]. Available: https://doi.org/10.1145/3468264.3468591

[55] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *J. Syst. Softw.*, vol. 177, p. 110935, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2021.110935

[56] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*. IEEE Computer Society, 2012, pp. 271–280. [Online]. Available: https://doi.org/10.1109/ISSRE.2012.22

[57] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, "An empirical study on real bugs for machine learning programs," in *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, J. Lv, H. J. Zhang, M. Hinchey, and X. Liu, Eds. IEEE Computer Society, 2017, pp. 348–357. [Online]. Available: https://doi.org/10.1109/APSEC.2017.41

[58] M. M. Morovati, A. Nikanjam, F. Tambon, F. Khomh, and Z. M. J. Jiang, "Bug characterization in machine learning-based systems," *Empir. Softw. Eng.*, vol. 29, no. 1, p. 14, 2024. [Online]. Available: https://doi.org/10.1007/s10664-023-10400-0

[59] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: a large-scale empirical study," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, J. M. González-Barahona, A. Hindle, and L. Tan, Eds. IEEE Computer Society, 2017, pp. 413–424. [Online]. Available: https://doi.org/10.1109/MSR.2017.59

[60] Y. Zhang, S. Cao, H. Wang, Z. Chen, X. Luo, D. Mu, Y. Ma, G. Huang, and X. Liu, "Characterizing and detecting webassembly runtime bugs," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 37:1–37:29, 2024. [Online]. Available: https://doi.org/10.1145/3624743

[61] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 385–396. [Online]. Available: https://doi.org/10.1145/3377811.3380397

[62] B. Zhang, Y. Tan, Y. Shen, A. Salem, M. Backes, S. Zannettou, and Y. Zhang, "Breaking agents: Compromising autonomous LLM agents through malfunction amplification," *CoRR*, vol. abs/2407.20859, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.20859

[63] M. Yu, F. Meng, X. Zhou, S. Wang, J. Mao, L. Pang, T. Chen, K. Wang, X. Li, Y. Zhang, B. An, and Q. Wen, "A survey on trustworthy LLM agents: Threats and countermeasures," *CoRR*, vol. abs/2503.09648, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.09648

[64] M. Cemri, M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. G. Parameswaran, D. Klein, K. Ramchandran, M. Zaharia, J. E. Gonzalez, and I. Stoica, "Why do multi-agent LLM systems fail?" *CoRR*, vol. abs/2503.13657, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.13657

[65] S. Zhang, M. Yin, J. Zhang, J. Liu, Z. Han, J. Zhang, B. Li, C. Wang, H. Wang, Y. Chen, and Q. Wu, "Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems," 2025. [Online]. Available: https://arxiv.org/abs/2505.00212