Understanding Bugs in Modern Agentic Frameworks: A Study of Symptoms, Root Causes, and Triggering Conditions

Xiaowen Zhang
Concordia University
Montreal, Canada
xiaowen.zhang@mail.concordia.ca

Hannuo Zhang
Concordia University
Montreal, Canada
hannuo.zhang@mail.concordia.ca

Shin Hwei Tan*
Concordia University
Montreal, Canada
shinhwei.tan@concordia.ca

Abstract

Modern agentic frameworks such as CrewAI and AutoGen have evolved into complex, autonomous multi-agent systems, introducing reliability challenges that go beyond earlier pipeline-based LLM libraries. However, existing empirical studies focus on earlier LLM libraries or task-level bugs, leaving the unique complexities of these agentic frameworks unexplored. We present a comprehensive study of 409 fixed bugs across five representative agentic frameworks, proposing a five-layer architectural abstraction. Our taxonomy identifies previously unreported symptom categories—Unexpected Execution Sequence, User Configuration Ignored, and Incomplete/Incorrect Trace—and isolates agent-specific root causes including Model-Related Fault, Cognitive Context Mismanagement, and Orchestration Fault. Notably, the model integration layer is the most bug-prone yet receives disproportionately low test inclusion rate during bug fixing (47%), revealing a critical validation gap. Despite varying design paradigms, bug symptoms, root causes, and bug-prone components show substantial cross-framework consistency (JS similarity 0.62–0.88). Finally, we present the first systematic study of bug-triggering conditions, identifying error-prone factor combinations across element configurations, input patterns, and operations, and demonstrate their transferability across frameworks, providing a foundation for test oracle design and cross-framework benchmark.

CCS Concepts

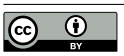
• **Software and its engineering → Software defect analysis**; **Software reliability**.

Keywords

Agentic Frameworks, Bug Triggers, Software Reliability, Failure Modes, Bug Taxonomy, Empirical Study

ACM Reference Format:
Xiaowen Zhang, Hannuo Zhang, and Shin Hwei Tan. 2026. Understanding Bugs in Modern Agentic Frameworks: A Study of Symptoms, Root Causes, and Triggering Conditions. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834378

*Corresponding Author



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834378

1 Introduction

With the recent advancement of large language models (LLM), several modern agent orchestration frameworks such as LangChain, LangGraph, CrewAI, AutoGen, and SmolAgents have emerged, each with distinct characteristics. These frameworks enable developers to compose autonomous agents, coordinate multi-agent workflows, manage contextual reasoning, and invoke external tools at scale. As they form the backbone of a growing ecosystem of downstream applications, defects within these frameworks can propagate widely, causing failures that range from silent semantic errors to system crashes. Therefore, it is important for developers and researchers to have a systematic understanding of the nature, causes, and triggering inputs of these defects.

A recent empirical study of defects in three LLM agent frameworks (i.e., LangChain, LlamaIndex, and Haystack) [54] proposed a taxonomy of nine root cause categories and six symptom categories, along with a four-component architectural abstraction: core schema, data preprocessing, agent construction, and featured modules. The study established a baseline by identifying Code Logic Issues (38%) and API Misuse (24.7%) as dominant root causes, Crash (31.8%) and Incorrect Functionality (30.7%) as the most prevalent symptoms. It further highlighted Unexpected Output (16.5%) as a characteristic symptom, reflecting the probabilistic and generative nature of LLM-based systems. Its abstractions center on data pipelines, retrieval-augmented generation (RAG), and prompt engineering. Despite these contributions, the prior study has three key limitations that motivate our work. **First, it targets an earlier generation of LLM frameworks** whose abstractions center on four components: i) data preprocessing, ii) core schema, iii) agent construction, and iv) featured modules — a design philosophy that no longer reflects the current state of the ecosystem. Modern agentic architectures have shifted toward autonomous multi-agent execution, long-running stateful workflows, and direct model orchestration. Frameworks such as AutoGen, CrewAI, and SmolAgents introduce qualitatively distinct paradigms such as event-driven messaging, role-based agent teams, and code-executing agents, that lead to qualitatively different failure modes [39]. Consequently, the previous four-component model, which is designed based on traditional software library structures, fails to capture the complexities of modern agent coordination, dynamic task delegation, and emergent runtime behaviors that characterize modern agentic systems.

Second, the study does not analyze bug-triggering scenarios, a critical gap for advancing automated testing of agentic frameworks. Knowing not only *what* bugs occur but *under what conditions* they manifest is essential for constructing fault-revealing test inputs, designing test oracles, and supporting fault localization in

multi-agent settings where non-deterministic interactions make reproduction inherently challenging.

Third, the study does not examine the cross-framework transferability of bug characteristics, leaving open whether the identified taxonomy and patterns generalize across frameworks with fundamentally different design philosophies. A transferability analysis is essential to distinguish framework-specific anomalies from universal bug patterns, and to inform the development of framework-agnostic repair and testing techniques.

To address these gaps, we present an empirical study of 409 fixed bugs from five representative frameworks: LangChain, LangGraph, CrewAI, AutoGen, and SmolAgents. To better reflect modern agentic architectures, we propose a refined five-layer abstraction comprising the Orchestration, Intelligence, Knowledge, Action, and Infrastructure layers. Our study aims to answer the following six Research Questions (RQs):

RQ1 (Symptoms): *What are the prevalent symptoms of bugs in modern agentic frameworks?* Analyzing symptoms identifies new failure modes in modern agentic frameworks to aid the development of effective test oracles.
RQ2 (Root Causes): *What are the common root causes of these bugs?* Uncovering root causes reveals fundamental design flaws and challenges stemming from autonomous orchestration and LLM integration that compromise framework reliability.
RQ3 (Bug-prone Components): *Which architectural layers are most susceptible to defects, and which layers do developer testing efforts primarily focus on?* This analysis identifies bug-prone components and reveals potential misalignment between developer testing focus and the actual distribution of bugs.
RQ4 (Commonality & Association): *Do symptoms, root causes, and components exhibit commonality across frameworks, and what are the relationships between these dimensions?* This analysis reveals systemic failures and guides developers in prioritizing testing for critical failure patterns.
RQ5 (Triggering Scenarios): *What user-side scenarios are most likely to trigger bugs in agentic frameworks?* Understanding these scenarios helps guide the design of automated test generation tools that effectively target common failure-inducing inputs.
RQ6 (Transferability): *To what extent are bug-triggering scenarios transferable across different framework designs?* Exploring transferability facilitates the generalization of localized bugs as framework-agnostic benchmarks for systematic cross-platform validation.

Our analysis reveals a misalignment in agentic framework reliability, where the model integration layer exhibits the highest bug density (25%) but comparatively low test inclusion rate during bug fixing (47%). Furthermore, despite varying design paradigms across the selected frameworks, their bug symptoms, root causes, and bug-prone components show high distributional similarity (JS similarity 0.62–0.88), revealing a shared reliability landscape.

Specifically, this paper makes the following contributions:

Architectural Abstraction for Modern Agentic Frameworks. We propose a five-layer architectural abstraction grounded in a systematic analysis of 409 fixed bugs across five representative modern agentic frameworks, providing a structured foundation for component-level testing and cross-framework comparison.

Taxonomies of Agentic Failures. We develop a taxonomy to capture failures characteristic of modern agentic frameworks, identifying new symptoms such as Unexpected Execution Sequence, User Configuration Ignored, and Incomplete/Incorrect Trace, and isolating agent-specific root causes including Model-Related Fault, Cognitive Context Mismanagement, and Orchestration Fault.
Bug-Triggering Conditions and Transferability. We present the first systematic study of bug-triggering conditions in agentic frameworks, identifying error-prone factors across element configurations, input patterns, and operations. We further explore the cross-framework transferability of these conditions, providing actionable insights for proactive defense and robust benchmarking.

2 Background and Related Work

2.1 Evolution of Agentic Frameworks

LLM application development has transitioned from using simple utility libraries to relying on complex autonomous orchestration infrastructures [39]. Early frameworks [16, 29, 40, 55] served as stateless intermediaries, providing standardized interfaces for prompt construction, retrieval-augmented generation pipelines, and basic API wrapping. In this paradigm, frameworks served as modular tool collections rather than cohesive execution environments, leaving developers to manually manage conversation state and multi-step workflows. In contrast, modern agentic frameworks such as LangGraph [18], CrewAI [13], AutoGen [52], and SmolAgents [46] have introduced integrated runtime environments characterized by persistent state management and autonomous decision loops. Unlike earlier approaches, they support cyclic control flows and coordination, realized through patterns such as event-driven messaging, role-based agent teams, and sandboxed code execution. This evolution shifts the role of frameworks from a passive library to an active orchestrator managing context persistence and dynamic tool invocation. Consequently, system correctness becomes tightly coupled to the framework's coordination logic and state transition semantics.

2.2 Study Scope and Framework Selection

To ensure a rigorous empirical analysis, we classify bugs into three levels: framework-level, application-level bugs, and task-level failures. We define **framework-level bugs** as defects within the source code of agentic frameworks, such as recovery errors, that occur independently of model capabilities. In contrast, **application-level bugs** stem from the misuse of framework APIs or errors in user-defined logic during system construction. Distinct from software defects, **task-level failures** refer to behavioral issues (e.g., hallucinations) where the agent fails an objective despite the code operating as intended, often representing the inherent uncertainty of model outputs. This three-tier distinction is essential for isolating the orchestration infrastructure from both application logic and model performance.

We selected five agentic frameworks from a recent survey [5] to represent diverse architectures and abstraction levels. **LangChain** typifies linear and directed acyclic graph (DAG) orchestration, while **LangGraph** introduces stateful cyclic graphs for iterative reasoning. To represent multi-agent collaboration, we include CrewAI,

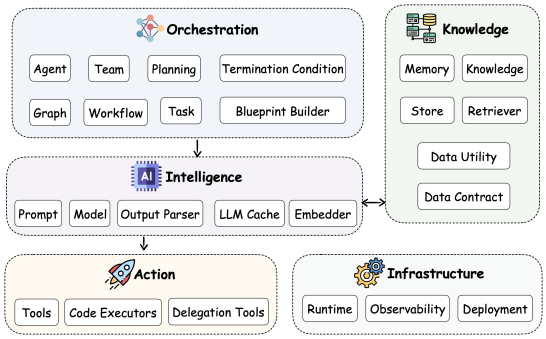


Figure 1: Conceptual architecture of agentic frameworks.

which prioritizes structured process-oriented workflows, and AutoGen, which facilitates autonomous conversation-driven interactions. Finally, **SmolAgents** represents a code-centric paradigm where agents interact primarily through sandboxed code executors.

2.3 Modern Agentic Framework Abstraction

To systematically analyze diverse agentic frameworks, we propose a five-layer architectural model (Figure 1), spanning from low-level environment management to high-level multi-agent coordination. We derived this abstraction by synthesizing conceptual components from recent literature [33] and functional modules in the five studied frameworks. Unlike the prior study that groups framework-specific modules into a flat "Featured Modules" category [54], our abstraction redistributes them into functional layers, capturing the integrated nature of agentic systems.

Infrastructure provides the runtime backbone for execution loops, inter-agent communication, lifecycle management, and monitoring, ensuring reliable operation and visibility into agent workflows.

Action interfaces agents with the external environment via tools, code executors, and delegation mechanisms that translate agent decisions into concrete operations.

Knowledge manages context, state, and data through memory stores, retrievers, and data contracts, supporting state persistence and consistent information flow across agents and workflows.

Intelligence serves as the cognitive engine, managing inference, semantic mapping, and protocol adaptation through models, adapters, prompts, embedders, and output parsers that convert raw LLM responses into structured, actionable representations.

Orchestration defines agents, task structures, and collaboration topologies, governing dynamic execution through workflow and graph-based connections, planning, termination conditions, and team management, with construction tooling to instantiate these structures from configurations or blueprints.

3 Methodology

3.1 Data Collection and Annotation

We collected bug-related issues with confirmed fixes from the official GitHub repositories of the selected agentic frameworks using a systematic pipeline to identify traceable issue–fix pairs. We mined all closed issues labeled as bugs, sorted by creation time from newest to oldest. Then, we filtered out issues without a "completed" `state_reason`, excluding those unlikely to have fixes, such as not-planned issues. To identify corresponding fixes, we retrieved timeline events for each issue and collected cross-referenced pull request (PR) links. Due to GitHub API rate limits, we examined only the first 100 events per issue and retained the latest 500 issues with linked PRs per framework. By our collection cutoff date (August 11, 2025), this process yielded 820 candidate issues: LangChain 419, LangGraph 25, CrewAI 96, AutoGen 183, and SmolAgents 97. Given the large number of issues in LangChain, we randomly sampled 100 LangChain issues to prevent one framework from dominating the dataset while preserving internal representativeness, and combined them with all others, resulting in 501 issues for manual verification. During manual verification, we kept only issues with PRs that fixed bugs in the core framework implementation, excluding duplicates, multi-bug reports, issues without fixes, or fixes only affecting peripheral tools (e.g., CrewAI CLI and AutoGen Studio) or test code. Multi-bug reports were excluded because a single report may contain multiple independent bugs, complicating one-to-one bug annotation. After filtering, 409 issues remained: LangChain 88, LangGraph 17, CrewAI 77, AutoGen 146, and SmolAgents 81. As LangGraph is built on LangChain components and they are often used together, we collectively refer to both frameworks as LC/LG.

We adopt precise definitions for two key terms [1, 37, 51]: a *fault* is a static defect in code or configuration, and a *failure* is an externally observable incorrect behavior. These terms correspond to our root cause and symptom dimensions, respectively. Other terms (*bug, defect, error, issue*) are used in their general sense.

For each bug report, we systematically annotated four dimensions: symptoms, root causes, buggy components, and triggering patterns. Following prior work [31, 50, 57], we derived our taxonomy using an open-coding approach. One author first reviewed all issues and associated fixes to establish an initial set of categories. Two authors independently annotated the collected issues across 11 batches, covering 5% of the data in the first two batches and 10% in each subsequent iteration. After each batch, we measured inter-annotator agreement using Cohen's kappa (κ) [49] and resolved disagreements through discussion, iteratively refining the categories and involving a third author when consensus could not be reached. The average κ was 0.68 in the first iteration, improved to 0.96 in the second, and consistently exceeded 0.89 in all remaining iterations. This manual analysis spanned six person-months as it required extensive documentation review to gain a deeper understanding of agentic frameworks.

3.2 Analysis Methods

To address RQ1, RQ2, and RQ3, our study developed taxonomies for bug symptoms, root causes, and components by analyzing the 409 bug reports and their associated fixes. For RQ3, we determined the affected conceptual components based on the functionality of the modified code and checked if each bug fix had corresponding test cases. Then, we further analyzed the commonality and association among these dimensions (RQ4).

Commonality analysis: To assess cross-framework consistency of bug characteristics, we evaluated the distributional similarity of symptoms, root causes, and components across the selected frameworks. While prior studies [31, 41, 48] used Spearman or Pearson correlation, these metrics can be misleading. Specifically,

rank-based Spearman is sensitive to minor permutations in low-frequency categories, while linear Pearson is skewed by dominant categories. To obtain a more robust measure, we employed Jensen-Shannon Divergence (JSD) [43], which measures the overlap between two probability distributions by comparing each to their mean. We then define *Jensen-Shannon (JS) similarity* as $1 - \sqrt{JSD}$, ranging from 0 (completely different) to 1 (identical).

Association analysis: To investigate relationships among symptoms, root causes, and components, we followed the statistical approach in [50]. First, we applied the Chi-squared test of independence to identify significant associations between dimension pairs (e.g., symptoms and root causes). We then used Cramér's V to measure association strength. Following Cohen's guidelines [32], we interpreted V values as weak (0.1–0.2), moderate (0.2–0.3), or strong (>0.3). To identify significant category pairs, we conducted a post-hoc analysis with a Bonferroni correction [47], ensuring only associations unlikely by chance were marked significant.

To answer RQ5, we analyzed bug-triggering scenarios for 273 bugs with reproduction code or descriptive configuration. Unlike a prior study on refactoring engines [50] where bugs often arise from simple input code, agentic frameworks exhibit complex interactions between element configurations, task types, and runtime behaviors. We characterized these via three sub-dimensions: Element Configurations (251 bugs), Input Patterns (56), and Operations (101). We define a *factor* as a bug-triggering characteristic, where a single bug may involve multiple factors per sub-dimension. Specifically, **Element Configurations** involve static element settings and environment variables, typically represented as element-attribute pairs (e.g., `model:modelId`). **Input Patterns** characterize runtime data such as task semantics or LLM responses. **Operations** denote functional behaviors, such as explicit tool calls, that exceed standard routines (i.e., basic inference passes).

Automated Frequent Pattern Mining: To identify recurring bug-triggering patterns, we applied FP-Growth [34], a technique proven effective for frequent pattern mining in log analysis [45, 56]. Each bug is treated as a transaction of factors, and an FP-tree is constructed to compactly encode shared structure. Mining this tree reveals frequent co-occurring factor sets above a support threshold, allowing us to characterize common multi-factor bug-triggering conditions. For factor combinations, we adopted a mixed-granularity approach to capture patterns across hierarchical levels. For any factor in the form $a : b$, we augmented the set by injecting its parent factor a. We set FP-Growth `min_support` to 0.05, requiring a pattern to appear in at least 5% of cases, and excluded patterns with fewer than five occurrences.

Transferability: To answer RQ6, we investigated whether bugs from one agentic framework recur in others. Given their shared analogous abstractions and coordination logic, we hypothesized cross-framework manifestations and sampled 35 representative bugs from the three triggering dimensions. Specifically, we randomly sampled bugs for the Element Configurations, while for Input Characteristics and Operations, we prioritized bugs whose triggering patterns are defined by *shared agentic interaction surfaces* (i.e., common abstractions such as models and agents, including their configurations and operations)—specifically message triggers and serialization interfaces—rather than framework-specific internals, as these provide structural preconditions for cross-framework reproduction. We evaluated these scenarios against the latest versions of LangChain (1.2.7), LangGraph (1.0.7), CrewAI (1.9.2), AutoGen (0.7.5), SmolAgents (1.24.0). For each source bug, we evaluated its presence across the four remaining frameworks. For each target framework, we first searched its GitHub repository for existing reports. A target issue was categorized as *Duplicate* if its reproducible code was functionally equivalent to the source. If no duplicate existed, we manually adapted the source bug code to the target framework. Upon successfully triggering the bug, we submitted a report and recorded it as an *Adapted* issue. If adaptation failed, we searched for a *Variant* issue sharing the same failure pattern based on the original bug-triggering factors but differing in component or context. For each source bug, only the highest-priority relation was recorded per target framework. A source bug is considered *transferable* if at least one target framework contains a Duplicate, Adapted, or Variant issue.

4 RQ1: Symptoms

4.1 Symptom Categories

Our analysis identified nine distinct symptom categories below:

Crash/Error. The framework or its internal components throw an exception, as indicated by error messages or stack traces.

Initialization Failure. Failures preventing framework initialization, involving module import, installation, or syntax errors.

Unexpected Intermediate State. A transient internal state produced during execution deviates from expectations. Users typically observe such deviations in logs, debug outputs, or code, and report them without mentioning any resulting consequences.

Unexpected Output. There is a mismatch between the expected and actual results, including responses from LLMs, agent decisions, workflow outcomes, tool and utility results, or serialized exports. For example, SmolAgents-1481 illustrates an unexpected agent decision, where the manager prematurely returned an incomplete answer before collecting all parallel subtask results [9].

Unexpected Execution Sequence. The framework produces an interaction sequence that deviates from the intended agent interaction order or task progression, such as order mismatch, early exit, unexpected continuation, or missing tool calls. For instance, in AutoGen-6710, the workflow manager mishandled loops, preventing any agents from activating and causing an early exit [27].

Incomplete/Incorrect Trace. The framework generates partial or misformatted execution traces or logs, hindering analysis and debugging. A case is AutoGen-6531, where the log of an `LLMCallEvent` omitted available tools, losing crucial execution context [19].

Performance Anomaly. The framework exhibits degraded performance or unresponsiveness, such as slow execution or hangs.

User Configuration Ignored. The framework silently ignores user-provided configurations, rendering them ineffective. For example, in LangChain-32059 [15], setting `num_gpu=4` for `OllamaEmbeddings` had no effect, with all computations falling back to CPU.

Others. This includes rare or miscellaneous symptoms that do not fit into other categories (e.g., broken documentation links).

4.2 Distribution and Comparison with Baseline

Table 1 presents the frequency and percentage of bug symptoms across studied frameworks. **Crash/Error** (54%) is the most common

Table 1: Bug distribution by symptoms and frameworks.

Symptom	Total		LC/LG		CrewAI		AutoGen		Smol	
	#	%	#	%	#	%	#	%	#	%
Crash/Error	**222**	**54.28**	**54**	**13.20**	**34**	**8.31**	**78**	**19.07**	**56**	**13.69**
Initialization Failure	**47**	**11.49**	**7**	**1.71**	**17**	**4.16**	**16**	**3.91**	**7**	**1.71**
Unexpected Output	**41**	**10.02**	**21**	**5.13**	3	0.73	10	2.44	**7**	**1.71**
Unexp. State	29	7.09	5	1.22	**6**	**1.47**	**16**	**3.91**	2	0.49
Unexp. Sequence	20	4.89	4	0.98	5	1.22	8	1.96	3	0.73
User Configuration Ignored	14	3.42	5	1.22	4	0.98	3	0.73	2	0.49
Incomplete/Incorrect Trace	12	2.93	2	0.49	3	0.73	5	1.22	2	0.49
Performance Anomaly	9	2.20	3	0.73	2	0.49	3	0.73	1	0.24
Others	15	3.67	4	0.98	3	0.73	7	1.71	1	0.24

Abbreviations: *Smol* denotes SmolAgents. *Unexp. State* and *Unexp. Sequence* denote Unexpected Intermediate State and Unexpected Execution Sequence, respectively.

symptom, reflecting persistent challenges in execution stability, consistent with prior studies on framework bugs [31, 54]. Automated techniques like fuzzing remain effective for detecting these runtime failures. **Initialization Failure** (11%) ranks second, with most cases (30/47) arising from module import errors caused by missing dependencies, version conflicts, or incomplete framework module exports. Due to Python's interpreted nature and dynamic imports, these issues appear only at runtime, revealing gaps between configuration correctness and execution-time validation. **Unexpected Output** (10%) is the third most common symptom, particularly in LC/LG (5%). Most cases (18/41) originate from core components, including model, agent, or workflow outputs. Such deviations may stem from the non-deterministic behavior of underlying models or orchestration defects within agents and workflows, making output validation particularly challenging as correctness often cannot be determined by simple assertions.

Compared to the baseline study [54], we identify four isolated symptoms in modern agentic frameworks. Import errors, which the baseline study primarily links to crashes as root causes, are treated as symptoms of **Initialization Failure**. These account for 7.3% of all bugs in our dataset, compared to 3.5% in the baseline study, indicating a higher prevalence of dependency and environment setup issues in agentic frameworks. **Unexpected Execution Sequence**, **User Configuration Ignored**, and **Incomplete/Incorrect Trace** are newly identified in modern frameworks, capturing failures in multi-agent orchestration, configuration enforcement, and trace recording (concerns that were absent or conflated with application-layer issues in earlier frameworks). These symptoms are treated as distinct categories (rather than refinements of the broader incorrect-functionality category as in [54]) because they capture failures in the execution process rather than the final output, representing concerns that were absent in earlier framework studies. These findings reveal gaps in validating execution paths, configuration effects, and trace integrity in modern agentic systems.

> ***Finding 1:*** *Crash/Error (54%), Initialization Failure (11%), and Unexpected Output (10%) are the most prevalent symptom categories. Unexpected Execution Sequence, User Configuration Ignored, and Incomplete/Incorrect Trace are newly identified categories not reported in the baseline study.*

5 RQ2: Root Causes

Our analysis grouped the root causes of 409 bugs into five categories: Agent-Specific Fault, General Programming Fault, Dependency/Environment Fault, Documentation Fault, and User Misuse.

5.1 Agent-Specific Fault

These faults reflect incorrect design decisions, assumptions, or execution policies in agentic frameworks, potentially disrupting agents' reasoning, perception, or orchestration. Fixing them requires domain knowledge, including interaction specifications and context propagation principles. We divide these faults into three categories: Model-Related Fault, Cognitive Context Mismanagement, and Orchestration Fault.

Model-Related Fault. These faults arise from misalignments between framework logic and LLM service chain, including the model, its API endpoint, and its client SDK (e.g., LiteLLM [30]):

- **Model Request Incompatibility (25/48):** The framework constructs inputs violating LLM-specific constraints (e.g., unsupported parameters or message formats), causing invalid request errors. In AutoGen-6116 (Figure 2a), Gemini-2.0-Flash responded only to the final system message when multiple were provided. This is fixed by merging all system messages.
- **Model Response Incompatibility (19/48):** The framework fails to parse or merge model responses due to nondeterministic behavior or uncommon response schema. In LangChain-31511 (Figure 2b), the framework failed to merge tool call chunks from Qwen3 when subsequent chunks lacked `name` and `id`. The fix added merging logic to handle such omissions.
- **Other causes (4/48):** The framework fails due to incorrect model metadata, or unsupported features.

Cognitive Context Mismanagement. We define *cognitive context* as runtime information shaping agents' situational awareness (e.g., conversation history, knowledge, tasks, tools, and collaborators). Mismanaging this context disrupts agent cognition, leading to degraded understanding, inconsistent decisions, or incorrect actions.

- **Prompt Fault (8/24):** Predefined prompts may contain faults (e.g., incorrect task specifications, insufficient instructions), leading to unexpected outputs. In SmolAgents-606 (Figure 2d), a manager agent failed to delegate tasks because its system prompt had the wrong parameter (`request` instead of `task`) .
- **Message Fault (6/24):** Errors in managing conversation history (e.g., missing messages, wrong roles, history truncation). For example, in SmolAgents-1097 (Figure 2f), exporting a planning step as an assistant message caused some LLMs to continue planning rather than execute tasks, as the missing role-change signal disrupted the agent's transition between task phases.
- **Context Propagation Fault (10/24):** The framework fails to propagate/isolate cognitive context, resulting in inconsistent or stale context across steps, agents, or components. For instance, LangChain-5700 (Figure 2e) reported that a subgraph could not access the runtime context of its parent graph, because the framework did not merge the parent context into the subgraph.

Orchestration Fault. These faults reflect incorrect coordination of agentic workflows, spanning navigation (workflow routing, speaker

```
 def _process_create_args(self, messages): ...
+ # Gemini only accepts a single system message
+ if create_args.get("model", "").startswith("gemini-"):
+   messages = create_args["messages"] # list of messages
+   messages = self._merge_system_messages(create\_args["messages"])
  ...
```

(a) Model Request Incompatibility: AutoGen-6116 [12]

```
 # chunk1: {"name":"search", "args":'{"query":', "id":"1", "index":1}
 # chunk2: {"name":None, "args":' "test"}', "id":None, "index":0}
 # [Fix] Handle chunks where model omits name/id
+ elif (e.get("type") == "tool_call_chunk" and merged
+   and e.get("name") is None and e.get("id") is None):
+   merge\_special\_chunk(e, merged)
```

(b) Model Response Incompatibility: LangChain-31511 [26]

```
 def tool(tool_function: Callable) -> Tool:
   class SimpleTool(Tool):
-    def __init__(self, name:str, ..., function: Callable):
-      self.name = name
-      ...
-      self.forward = function
+    def __init__(self):
       self.is_initialized = True

   ...
- simple_tool=SimpleTool(name=tool_json_schema["name"], ...,
↪  function=tool_function)
- ...
+ SimpleTool.name = tool_json_schema["name"]
+ ...
+ SimpleTool.forward = staticmethod(tool_function)
+ simple_tool = SimpleTool()
  return simple_tool
```

(c) Class Design Defect: SmolAgents-913 [11]

```
# prompts/toolcalling_agent.yaml
system_prompt: |-
- You can give requests to team members...The only argument you can give is 'request'.
+ You can give tasks to team members...The only argument you can give is 'task'.
```

(d) Prompt Fault: SmolAgents-606 [8]

```
 runtime = Runtime(context=context, store=store, stream_writer=stream_writer, previous=None)
+parent_runtime = config[CONF].get(CONFIG_KEY_RUNTIME, DEFAULT_RUNTIME)
+runtime = parent_runtime.merge(runtime)
```

(e) Context Propagation Fault: LangChain-5700 [22]

```
   def to_messages(self, summary_mode: bool, **kwargs) -> List[Message]: ...
     return [Message(role=MessageRole.ASSISTANT,content=[{"type":"text","text":self.plan}]),
+      Message(role=MessageRole.USER, content=[{"type":"text","text":"Execute the plan."}]),
    ]
```

(f) Message Fault: SmolAgents-1097 [21]

Figure 2: Simplified code snippets illustrating representative root causes.

selection), termination (signal and condition handling), and component coordination (concurrency, mode transitions, action alignment). For instance, AutoGen-6710 [27] exhibited premature termination due to incorrect cycle tracking, leaving prerequisites unsatisfied and blocking agent execution.

5.2 General Programming Fault

This encompasses common coding mistakes, divided into 12 subtypes. Following prior work [38, 53], we organize these according to descending scope, ranging from broad feature-level faults to narrow statement-level errors.

Missing Case Handling. The framework does not implement specific features or edge cases. For example, in AutoGen-5518, the framework did not support PowerShell code, producing an incorrect file extension, causing the local executor to reject the code [20].

Class Design Defect. Framework classes may violate architectural constraints (e.g., incorrect inheritance or invalid protocols). In SmolAgents-913, `SimpleTool` broke validation rules by defining `__init__` parameters without defaults, causing errors when exported for sandbox execution. The fix (Figure 2c) removed these parameters and assigned them externally to ensure compliance.

State/Resource Mismanagement. This involves mishandled state updates or resource lifecycles, causing inconsistencies or failed operations. In AutoGen-6308 [6], a `SearchClient` was closed prematurely by an `async with` block, causing later tool calls to fail.

Concurrency Fault. The framework mishandles concurrent execution (e.g., asynchronous tasks, thread-unsafe scheduling, or race conditions). In AutoGen-4490 [4], unawaited `WriteStateAsync()` caused out-of-order writes and `InconsistentStateException`.

Serialization Fault. The framework mishandles (de)serialization, violating high-level data contract assumptions (e.g., schema consistency, completeness, and idempotence).

API Misuse. The framework violates API contracts by passing invalid or missing parameters, leading to invocation failures or ineffective calls.

Missing Parameter Forwarding. These faults occur when optional parameters fail to reach their targets due to omission or absence in the receiving interface. Such faults rarely trigger explicit error messages but may silently affect downstream processing.

Wrong Control Flow. The framework executes in incorrect order or under unintended conditions due to incorrect control flow (e.g., evaluating the wrong branch [14]).

Missing Check. The framework executes existing logic without validating preconditions (e.g., nullity, state, or existence checks), allowing invalid inputs or states to propagate.

Incorrect Variable/Value. The framework uses incorrect variables or values, mismanages local variable updates, or constructs values with incorrect structures.

Type Fault. The framework incorrectly uses or converts data types, resulting in type mismatches, invalid casting, or improper type conversions that violate expected type constraints.

5.3 Dependency/Environment Fault

This category captures problems related to dependency management and execution environment compatibility, which affect framework integration, deployment, or extensibility.

Dependency Fault. These faults stem from missing or incompatible dependencies, third-party library bugs, or structural issues (e.g.,, improper module exports and rigid coupling) that hinder integration and extensibility.

Environment Incompatibility. These occur when the framework is incompatible with the execution environment (e.g.,, operating system, Python version, container image, hardware, or drivers).

Table 2: The number of bugs due to different root cause in each agentic framework.

Root Cause	Subcategory	Total		LC/LG		CrewAI		AutoGen		Smol	
		#	%	#	%	#	%	#	%	#	%
General Programming Fault	Missing Case Handling	**51**	**12.47**	17	4.16	9	2.20	7	1.71	18	4.40
	Incorrect Variable/Value	26	6.36	12	2.93	6	1.47	3	0.73	5	1.22
	Wrong Control Flow	23	5.62	8	1.96	6	1.47	7	1.71	2	0.49
	Class Design Defect	19	4.65	6	1.47	2	0.49	6	1.47	5	1.22
	Type Fault	18	4.40	7	1.71	5	1.22	4	0.98	2	0.49
	State/Resource Mismanagement	17	4.16	5	1.22	1	0.24	10	2.44	1	0.24
	Missing Check	17	4.16	9	2.20	1	0.24	6	1.47	1	0.24
	API Misuse	16	3.91	2	0.49	4	0.98	8	1.96	2	0.49
	Missing Parameter Forwarding	10	2.44	3	0.73	3	0.73	3	0.73	1	0.24
	Serialization Fault	9	2.20	1	0.24	2	0.49	5	1.22	1	0.24
	Concurrency Fault	6	1.47	0	0.00	1	0.24	5	1.22	0	0.00
	Subtotal	**212**	**51.83**	**70**	**17.11**	**40**	**9.78**	**64**	**15.65**	**38**	**9.29**
Agent-Specific Fault	Model-Related Fault	**48**	**11.74**	11	2.69	1	0.24	28	6.85	8	1.96
	Cognitive Context Mismanagement	24	5.87	2	0.49	6	1.47	7	1.71	9	2.20
	Orchestration Fault	16	3.91	2	0.49	5	1.22	8	1.96	1	0.24
	Subtotal	**88**	**21.52**	**15**	**3.67**	**12**	**2.93**	**43**	**10.51**	**18**	**4.40**
Documentation Fault		**51**	**12.47**	8	1.96	8	1.96	16	3.91	19	4.65
Dependency/Environment Fault	Dependency Fault	35	8.56	10	2.44	9	2.20	12	2.93	4	0.98
	Environment Incompatibility	12	2.93	0	0.00	5	1.22	5	1.22	2	0.49
	Subtotal	**47**	**11.49**	**10**	**2.44**	**14**	**3.42**	**17**	**4.16**	**6**	**1.47**
User Misuse		11	2.69	2	0.49	3	0.73	6	1.47	0	0.00

5.4 Documentation Fault

This arises when documentation or tutorials misrepresent framework interfaces, behavior, or usage, encompassing incorrect examples (32/51), missing constraints (10/51), and outdated descriptions (9/51). For instance, CrewAI-2647 [7] reported an authentication error when enabling planning for a `Crew` using non-OpenAI models, as the framework silently invoked `gpt-4o-mini` for planning without documenting the implicit OpenAI API key requirement.

5.5 User Misuse

This category includes issues caused by incorrect interactions or misconfigurations by users rather than internal framework faults.

5.6 Distribution and Comparison with Baseline

Table 2 summarizes root-cause distributions across frameworks, including raw counts (#) and proportions (%). **General Programming Fault** accounts for the largest share (51.83%), with Missing Case Handling (12.47%) as the most frequent subcategory. Its consistent presence across frameworks suggests a systemic lack of defensive programming practices rather than framework-specific design issues, making it largely amenable to conventional debugging techniques. **Agent-Specific Fault** accounts for 21.52% of all bugs, comprising Model-Related Fault (11.74%), Cognitive Context Mismanagement (5.87%), and Orchestration Fault (3.91%).These subcategories reflect challenges unique to agentic frameworks, spanning model integration, context management, and multi-agent coordination. *Model-Related Fault are nearly as prevalent as Missing Case Handling*. Notably, AutoGen accounts for the majority of Model-Related Fault (6.85% out of 11.74%), likely due to its support for heterogeneous model backends, which increases compatibility risks.

Table 3: Bug distribution by components and frameworks.

Component	Total		LC/LG		CrewAI		AutoGen		Smol		Tested	
	#	%	#	%	#	%	#	%	#	%	#	%
Intelligence	**101**	**24.69**	**38**	**9.29**	6	1.47	**42**	**10.27**	15	3.67	47	46.53
Orchestration	**82**	**20.05**	7	1.71	**23**	**5.62**	**36**	**8.80**	16	3.91	**54**	**65.85**
Action	62	15.16	7	1.71	9	2.20	22	5.38	**24**	**5.87**	**38**	**61.29**
Knowledge	56	13.69	**30**	**7.33**	13	3.18	10	2.44	3	0.73	25	44.64
Infrastructure	54	13.20	14	3.42	**16**	**3.91**	19	4.65	5	1.22	17	31.48
Documentation	54	13.20	9	2.20	10	2.44	17	4.16	**18**	**4.40**	0	0.00
Total	409	100.00	105	25.70	77	18.80	146	35.70	81	19.80	181	44.30

Documentation Fault (12.47%) ranks third, primarily caused by incorrect or outdated example code, reflecting challenges in keeping documentation synchronized with rapidly evolving agentic APIs.

Compared to the baseline [54], which identifies Code Logic Issues (38%) and API Misuse (25%) as dominant in early LLM pipelines, our analysis reveals a structural shift in agentic frameworks. We decouple **Model-Related Fault** from generic API Misuse to capture adaptation gaps between framework logic and heterogeneous model protocols. Meanwhile, coordination-related issues previously subsumed under code logic are distinguished into **Cognitive Context Mismanagement** and **Orchestration Fault**, both absent from the baseline taxonomy.

> ***Finding 2:*** *While General Programming Fault dominates (52%), Agent-Specific Fault (22%) represents root cause categories absent from the baseline study, comprising Model-Related Fault, Cognitive Context Mismanagement, and Orchestration Fault.*

6 RQ3: Bug-prone Components

Table 3 shows the bug distribution across framework components. While our analysis primarily focuses on functional components, we also include "Documentation" to provide a comprehensive view of all reported issues. We included a *Tested* column to measure the percentage of bug fixes accompanied by test cases. Bugs are predominantly found in core functional components, with Intelligence accounting for the largest share (25%), followed by Orchestration (20%) and Action (15%). These results indicate that model adaptation and coordination logic are the primary sources of fault. This distribution further varies across frameworks according to their design priorities. For instance, SmolAgents skews toward Action (6%) due to its action-in-code design, and LC/LG shows a notably higher proportion in Knowledge (7%) due to its RAG emphasis.

While the baseline study [54] identified Data Processing (40%) and Core Schema (24%) as the key bug-prone components in early pipelines, our results show the reliability bottleneck has shifted to Intelligence and Orchestration in modern agentic frameworks, reflecting the increasing complexity of multi-agent coordination. Our five-layer abstraction pinpoints bugs in model adaptation and coordination logic that are collapsed together in prior study.

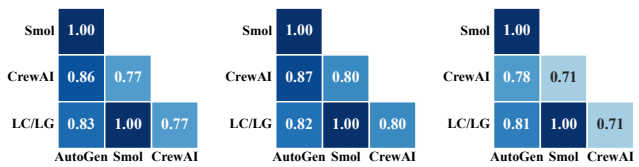
Our analysis also reveals a significant misalignment between empirical risk and testing effort during the bug-fixing process. While Intelligence is the most bug-prone, it receives a disproportionately low test inclusion rate (47%), whereas Orchestration and Action see much higher rates (66% and 61%).

> ***Finding 3:*** *Intelligence (25%) and Orchestration (20%) are the most bug-prone layers overall, though the dominant components differ across frameworks. Developers include tests more often for Orchestration than Intelligence (66% vs. 47%), leaving the most bug-prone layer undervalidated.*

7 RQ4: Commonality & Association

7.1 Cross-framework Commonality

All studied frameworks show high cross-framework commonality in symptoms (0.76–0.86) and root causes (0.80–0.88). Component-level similarity (0.62–0.81) is lower, largely due to LangChain's disproportionate concentration of bugs in the Knowledge components (Section 6). Nevertheless, the lower bound of 0.62 still indicates substantial commonality overall. These patterns suggest a layered evaluation framework: universal black-box oracles for common symptoms, complemented by white-box testing targeting framework-specific bug-prone components to improve diagnosis and coverage.



(a) Symptoms **(b) Root Causes** **(c) Components**

Figure 3: Jensen-Shannon similarity between frameworks.

Table 4: Significant co-occurring category pairs.

Relationship	Category A	Category B	*N*
Root Cause × Symptom	Agent-specific Fault	Unexp. State	14
	Dependency/Environment Fault	Init. Failure	30
Root Cause × Component	Agent-specific Fault	Intelligence	44
		Orchestration	32
	Dependency/Environment Fault	Infrastructure	21
	General Programming Fault	Action	52
		Knowledge	44
Symptom × Component	Init. Failure	Infrastructure	20
	Incomplete/Incorrect Trace		6
	Unexp. Sequence	Orchestration	12
	Unexp. State		14

Abbreviations: *Init. Failure*, *Unexp. State* and *Unexp. Sequence* denote Initialization Failure, Unexpected Intermediate State and Unexpected Execution Sequence, respectively.

7.2 Inter-dimensional Relationships

Chi-squared tests confirm significant associations ($p < 0.001$) across all analyzed dimension pairs, indicating that bug characteristics are interdependent. Strongest associations involve Root Cause with Symptom ($\chi^2 = 241.19, V = 0.384$) and Component ($\chi^2 = 472.56, V = 0.537$), while the association between Symptom and Component is also substantial ($\chi^2 = 171.16, V = 0.289$).

Table 4 lists significant category pairs from post-hoc analysis (N denotes frequency). We omit pairs involving Others symptom or documentation-only issues. These associations reveal two bug patterns newly observed in agentic frameworks. First, faults in Orchestration components frequently manifest as Unexpected Intermediate State and Unexpected Execution Sequence, indicating that frameworks often fail to maintain state and sequence integrity across complex topologies. For instance, in CrewAI-1463 [3], the workflow failed to handle multi-start configurations, leading to untracked execution paths [3]. Second, the association between Action/Knowledge components and General Programming Fault highlights a lack of defensive logic in deterministic components when handling edge cases triggered by LLM interactions. These relationships enable the design of symptom-based test oracles and targeted defensive mechanisms to mitigate the specific root causes.

> ***Finding 4:*** *Symptoms, root causes, and components show substantial cross-framework commonality (0.62–0.88) and significant inter-dimensional associations. Category-level analysis reveals two core bug patterns: (1) Orchestration components frequently compromise state and sequence integrity, and (2) Action/Knowledge components often lack defensive logic against LLM-driven interactions.*

8 RQ5: Triggering Scenarios

This section analyzes frequent bug-triggering factors among 273 bugs across three dimensions: element configurations, input characteristics, and operations. Factors are not mutually exclusive, as a single bug may involve multiple triggers. Detailed definitions of these factors are provided in Appendix A.

```python
# Element Config. Labels: "Model:Backend" and "Model:ModelID"
client = OpenAIClient(model="gemini-1.5-flash")
# Input Characteristic Label: "InputMessages"
messages = [
    SystemMessage(content="Say anything Start with 'FOO'"),
    SystemMessage(content="Say anything End with 'BAR'"),
    UserMessage(content="Just say '.'", source="user")]
result = await client.create(messages=messages)
```

(a) AutoGen-6116 [12]

```python
@tool # Element Configuration Label: "Tool:Signature"
def calculator(args: Dict[str, str]):
    pass

agent = create_react_agent(model, tools=[calculator])
executor = AgentExecutor(agent, tools=[calculator])
# Operation Label: TaskIntent:CalculationTask
result = executor.invoke({"input": "10+25=?"})
# Input Characteristic Label: "LLMResponse:ToolCallSchema"
# Non-standard JSON formatting: "{'a':'10', 'b':'25', 'operator':'+'}"
```

(b) LangChain-30910 [25]

Figure 4: Simplified triggering scenarios with labels.

***Element Configurations*(251, 92%).** Element configurations are the most pervasive dimension, involved in 92% of bug-triggering scenarios. Model (88, 35%) settings are the most frequent trigger, with Backend (55, 22%) and ModelID (39, 16%) governing communication protocols and LLM identity, respectively. Tool (64, 25%) definitions shape how agents interact with external services, where BuiltinType (18, 7%) and Signature (16, 6%) misconfigurations directly corrupt tool-calling prompts. Beyond these, Agent (38, 15%), Team (30, 12%), CodeExecutor (17, 7%), and Store (16, 6%) cover coordination, execution, and persistence concerns, respectively. Notably, pattern mining identifies a high-risk combination where bugs emerge exclusively from specific Backend and ModelID co-configurations (8%), exemplified by AutoGen-6116 triggered by the `OpenAIClient` and "gemini-1.5-flash" pairing (Figure 4a).

***Input Characteristics* (56, 21%).** TaskIntent (19, 34%) and LLMResponse (12, 21%) are the two dominant triggers. Complex task intents—such as MultiToolCallTask (5, 9%) and MultimodalTask (5, 9%)—frequently expose flaws in orchestration logic; for instance, in SmolAgents-1481, a prompt to "launch two web search agents in parallel" exposed greedy behavior in the manager agent, causing incomplete answers [9]. LLMResponse-related triggers arise when frameworks fail to handle atypical model outputs, namely ToolCallSchema (6, 11%) and RegularSchema (5, 9%) violations; as shown in Figure 4b, a non-standard JSON string in the arguments field causes parsing failures. InputMessages (11, 20%) captures structural or formatting anomalies in message history, frequently arising from model-specific constraints—such as Anthropic models rejecting multiple system messages (Figure 4a)—that frameworks fail to uniformly enforce. Code (8, 14%) covers executable code inputs whose specific patterns expose implementation limitations of code execution components; for example, in SmolAgents-839, chained assignments crashed the `LocalPythonExecutor`.

***Operations* (101, 37%).** CallTool (24, 24%) is the most frequent operation trigger, reflecting that the interface between agents and external services is a high-risk boundary. ExportComponent (16, 16%) and LoadComponent (7, 7%) highlight serialization and deserialization as risk-prone operations, where corruption in the dump or load process can compromise framework state. ExecCode (11, 11%) involves dynamic execution of generated code within framework-managed sandboxes. ConsecutiveExec (8, 8%) exposes residual state risks from repeated execution of the same operation—as shown in AutoGen-6746 [10], residual state from previous runs corrupts the context of subsequent calls. ProcessDoc (7, 7%) covers utility operations for handling external documents.

Cross-dimension pattern mining yields no combinations meeting the support threshold (≥ 5%), suggesting that triggering conditions across dimensions tend to occur independently.

> ***Finding 5:*** *Element configurations are the most pervasive trigger dimension (92%), with pattern mining identifying specific Backend and ModelID co-configurations as a high-risk combination (8%). Complex TaskIntent (34%) and atypical LLMResponse (21%) dominate input-related triggers, with InputMessages bugs frequently tied to model-specific message formatting constraints. Among operations, CallTool (24%) is the most frequent trigger, while serialization and consecutive execution further highlight state management as a recurring risk.*

9 RQ6: Transferability

We found 16 of 35 source bugs (46%) involving shared agentic interaction surfaces are transferable, yielding 29 issues across all studied frameworks: 4 Duplicates, 11 Adapted, and 14 Variants. Duplicates and Adapted cases account for over half of transfers, indicating many functional counterparts across frameworks. Variants further show that core triggering patterns generalize across components to uncover latent faults, supporting cross-framework bug transfer.

We submitted 11 bug reports, all acknowledged by maintainers or community contributors. One was fixed, six received community acknowledgment via reproductions or fixes, and four were recognized by maintainers as design boundaries or addressed with alternative solutions. These varied resolutions reflect genuine design trade-offs rather than dismissals—for example, we adapted a serialization bug from AutoGen-6855 [24] to LangChain-34925 [28], where AI messages in chat history lost type-specific fields like tool call details; maintainers acknowledged the issue but suggested migrating to newer memory abstractions in LangGraph rather than fixing it directly. Such resolutions reflect genuine design trade-offs rather than dismissals, showing that our cross-framework transfers effectively uncover community-recognized faults across diverse implementations.

> ***Finding 6:*** *Nearly half of sampled bugs involving shared agentic interaction surfaces are transferable across frameworks as adapted functional counterparts or generalized variants, highlighting that existing bug-triggering patterns provide effective templates for cross-framework validation.*

10 Discussion

From Architectural Evolution to Interaction-Level Reliability. Modern agentic frameworks have evolved from passive libraries to active orchestrators governing state, control flow, and agent coordination, expanding the correctness concerns from individual function outputs to include multi-step interactions among agents, models, and external services. Consistent with Findings 2 and 3,

the most bug-prone layers and root causes are concentrated around model integration and workflow coordination, indicating that reliability challenges increasingly arise at interaction boundaries rather than within isolated implementations. Model-Related Fault occurs nearly as frequently as Missing Case Handling, while the growing prevalence of Initialization Failure (Section 4.2) further highlights increasing dependence on external services and runtime ecosystems. Accordingly, our Agent-Specific Fault categories provide structured diagnostic paths for localizing interaction-related faults.

Validation Gaps: Testing Bias and Oracle Limits. Despite being the most bug-prone layer, Intelligence exhibits a notable mismatch between bug density and test inclusion rate (Finding 3). This aligns with prior observations that developers preferentially test deterministic components [35], and our data extend this by showing the bias persists during bug-fixing itself—suggesting that testing toolchains remain inadequately equipped to mock or simulate complex external model dependencies. The newly identified symptom categories further expose structural oracle limitations (Finding 1): Unexpected Execution Sequence and User Configuration Ignored indicate that conventional fuzzing and assertion-based testing are insufficient to validate execution order deviations or configuration enforcement, respectively.

Cross-Framework Benchmarking and Transferability. The substantial cross-framework commonality in symptoms and root causes (Finding 4), together with the transferability of nearly half of sampled bugs and developer engagement with all submitted reports (Finding 6), provides strong evidence for reusable validation artifacts across agentic frameworks. This evidence supports a layered evaluation framework in which shared benchmarks capture common interaction-level correctness concerns, while framework-specific extensions address distinctive architectural mechanisms. Such a framework could enable community-maintained benchmark suites for systematic cross-framework validation.

Design, Testing, and Research Implications. Collectively, our findings suggest that reliability in agentic frameworks should be treated as an interaction-contract problem. For designers, this calls for defining explicit interaction assumptions through typed interfaces and compatibility contracts, while making execution-state transitions and termination criteria explicit at the interaction boundary. For testers, the observed validation gaps and configuration-dependent faults motivate interaction-aware testing strategies that systematically exercise execution sequences, configuration spaces, and atypical LLM behaviors. For researchers, the persistent oracle challenges and cross-framework bug transferability highlight opportunities for automated oracle construction, configuration-aware testing techniques, and community-driven validation benchmarks.

11 Threats to Validity

We identify three threats to validity. First, subjective bias in manual annotation was mitigated through multi-author coding, achieving high inter-annotator agreement (Cohen's $\kappa > 0.89$). Second, to ensure generalizability, we analyzed 409 fixed bugs across five frameworks representing diverse architectural paradigms, covering a broad ecosystem. While LangChain issues were randomly downsampled for cross-framework balance, our goal is taxonomy construction rather than prevalence estimation; the consistent patterns observed across frameworks support the robustness of the taxonomy. Finally, to prevent misclassification of framework-level bugs, we applied a three-tier definition (task, application, and framework bugs) and analyzed only confirmed fixes from official repositories.

12 Related Works

Task- and Application-level Failures. Recent studies examine agent failure modes at the task level, focusing on inter-agent misalignment [44] or planning and execution errors [42]. Others explore the application layer, identifying bugs in prompt engineering or end-user logic [36]. While these studies offer valuable insights into agent behavior, they primarily analyze model-driven symptoms or application-specific faults. Our study targets the framework layer, investigating internal source code and architectural defects of the orchestration engines that underpin these systems.

Framework-level Bug Characterization. While Xue et al. characterize bugs in early LLM pipelines [54], our work addresses the complexities of modern multi-agent systems through a more granular five-layer architectural model. We move beyond general programming errors to identify specialized agent-specific symptoms and root causes (e.g., Cognitive Context Mismanagement). We further evaluate bug-triggering scenarios and their cross-framework transferability, offering a predictive perspective on structural vulnerabilities that are absent in prior literature.

13 Conclusions

This paper presents an empirical study of 409 fixed bugs across five modern agentic frameworks, analyzing their symptoms, root causes, bug-prone components, triggering conditions, and cross-framework transferability under a five-layer architectural model. The results indicate a paradigm mismatch in reliability assessment: correctness in agentic systems depends on interaction contracts across agents, models, and external services, whereas prevailing testing techniques (e.g., fuzzing and assertion-based oracles) are designed for deterministic, function-level behaviors and therefore fail to capture interaction-driven failures. Cross-framework commonality in symptoms and root causes (0.62–0.88), together with substantial bug transferability, further shows that these issues stem from shared architectural interaction patterns rather than framework-specific defects, motivating unified testing infrastructure. A key limitation is that our dataset, derived from resolved GitHub issues, excludes silent and unreported failures, which may affect distributional generality. We identify three directions for future work: oracles for execution-order and configuration constraints, mock infrastructure for heterogeneous LLM behaviors, and cross-framework benchmarks based on transferable triggering patterns.

Data Availability Statement. Data, code, and the annotation cookbook are available at [58].

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada under Discovery Grant No. RGPIN-2024-04301.

A Detailed Definitions of Triggering Factors

A.1 Element Configurations (251, 92%).

Model (88, 35%): Model clients used to interface with LLMs.

- **Backend (55, 22%):** The internal client class managing communication protocols (e.g., `OpenAIClient` in Figure 4a).
- **ModelID (39, 16%):** The specific LLM identifier executing the reasoning tasks (e.g., `gemini-1.5-flash`).

Tool (64, 25%): This enables agents to take actions.

- **BuiltinType (18, 7%):** Pre-implemented tool classes providing ready-to-use actions (e.g., `AzureAISearchTool` in AutoGen).
- **Signature (16, 6%):** The parameter types and return schema used by the framework to structure tool-calling prompts.

Agent (38, 15%): Autonomous entities orchestrating reasoning and tool usage, characterized by role and planning logic.

Team (30, 12%): High-level entities coordinating multi-agent collaboration via hierarchical or flat organizational structures.

CodeExecutor (17, 7%): It manages the execution environment for generated code snippets. For example, `ExecutorType` includes runtime backend for execution (e.g., local or remote executors).

Store (15, 6%): Data persistence and indexing, with a frequent attribute `Backend` representing the underlying database or file system.

A.2 Input Characteristics (56, 21%).

Characteristics of user requests or model outputs that trigger faults.

TaskIntent (19, 34%): Abstract patterns representing the functional goal of a request.

- **MultiToolCallTask (5, 9%):** Requests for orchestrating multiple tool or agent calls. In SmolAgents-1481, a prompt to "launch two web search agents in parallel" exposed greedy behavior in the manager agent, causing incomplete answers [9].
- **MultimodalTask (5, 9%):** Tasks involving non-textual data (e.g., images), provided either as input or generated as tool outputs.

LLMResponse (12, 21%): Patterns in model outputs that deviate from framework expectations.

- **ToolCallSchema (6, 11%):** Atypical or malformed tool call messages that bypass parsers. In Figure 4b (Line 10), the LLM generates a non-standard JSON string for the `arguments` field, leading to parsing failures and subsequent invocation errors.
- **RegularSchema (5, 9%):** Flaws or non-determinism in standard text responses that break downstream logic (e.g., non-deterministic text formats in LangChain-8716 [2]).

InputMessages (11, 20%): Structural or formatting features of the message history, such as multiple system messages in Figure 4a.

Code (8, 14%): Specific patterns in code snippets that expose implementation limitations of executors. In SmolAgents-839, chained assignments crashed the `LocalPythonExecutor`.

A.3 Operations (101, 37%).

Bug-triggering runtime actions.

CallTool (24, 24%): Framework invocations of external tools.

ExportComponent (16, 16%), LoadComponent (7, 7%): Operations for serializing and restoring framework entities, such as state corruption in the `dump` or `load_component` [23].

ExecCode (11, 11%): Dynamic execution of generated code within framework-managed sandboxes.

ConsecutiveExec (8, 8%): Repeated execution of the same task or instance. In AutoGen-6746 [10] residual state from previous runs corrupts the context, causing subsequent calls to fail.

ProcessDoc (7, 7%): Utility operations for handling external documents, such as batch indexing (e.g., LangChain-32272 [17]).

References

[1] 1990. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990* (1990), 1–84. doi:10.1109/IEEESTD.1990.101064

[2] 2023. *Add improved sources splitting in BaseQAWithSourcesChain.* https://github.com/langchain-ai/langchain/pull/8716

[3] 2024. *[BUG] Flow @listen with _and error.* https://github.com/crewAIInc/crewAI/issues/1463

[4] 2024. *Exceptions in Orleans storage.* https://github.com/microsoft/autogen/issues/4490

[5] 2025. *Agentic AI: Comparing New Open-Source Frameworks.* https://www.ilsilfverskiold.com/articles/agentic-aI-comparing-new-open-source-frameworks

[6] 2025. *AzureAISearchTool Not working as expected.* https://github.com/microsoft/autogen/issues/6308

[7] 2025. *[BUG] Authentication Error When Using OpenAI Compatible LLMs.* https://github.com/crewAIInc/crewAI/issues/2647

[8] 2025. *[BUG] Error when using ToolCallingAgent as manager.* https://github.com/huggingface/smolagents/issues/606

[9] 2025. *[Bug] Final answer is too greedy in ToolCallingAgent parallel calls.* https://github.com/huggingface/smolagents/issues/1481

[10] 2025. *Bug: GraphFlow with termination condition automatically ends after first query.* https://github.com/microsoft/autogen/issues/6746

[11] 2025. *[BUG] Tool validation with executor type defined.* https://github.com/huggingface/smolagents/issues/913

[12] 2025. *Could not send Multiple system message at autogen.* https://github.com/microsoft/autogen/issues/6116

[13] 2025. *CrewAI.* https://github.com/crewAIInc/crewAI

[14] 2025. *Customized ChatAgentContainer not support.* https://github.com/microsoft/autogen/issues/6730

[15] 2025. *It seems that num_gpu does not work with ollama models.* https://github.com/langchain-ai/langchain/issues/32059

[16] 2025. *LangChain.* https://github.com/langchain-ai/langchain

[17] 2025. *LangChain Indexing API: Incorrect num_skipped Count Due to Missing Within-Batch Deduplication Tracking.* https://github.com/langchain-ai/langchain/issues/32272

[18] 2025. *LangGraph.* https://github.com/langchain-ai/langgraph

[19] 2025. *LLMCallEvent fails to log "tools" to be chosen from by the LLM in BaseOpenAIChatCompletionClient.* https://github.com/microsoft/autogen/issues/6531

[20] 2025. *LocalCommandLineCodeExecutor to support PowerShell.* https://github.com/microsoft/autogen/issues/5518

[21] 2025. *Planning step can make next ActionStep miss its start.* https://github.com/huggingface/smolagents/issues/1097

[22] 2025. *Runtime context is not being passed to the subgraph.* https://github.com/langchain-ai/langgraph/issues/5700

[23] 2025. *Some agents do not deserialize model_context (set to None or Do not serialize too).* https://github.com/microsoft/autogen/issues/6336

[24] 2025. *Structured logging is missing data from Trace logging (0.6.4).* https://github.com/microsoft/autogen/issues/6855

[25] 2025. *Tool Call Fails: Constructs Invalid Arguments When The Function Accepts Arguments of Type Dictionary.* https://github.com/langchain-ai/langchain/issues/30910

[26] 2025. *Unable to handle inconsistent tool call indices with streaming output when using Qwen3.* https://github.com/langchain-ai/langchain/issues/31511

[27] 2025. *Workflow with multiple cycles stop execute.* https://github.com/microsoft/autogen/issues/6710

[28] 2026. *AIMessage.tool_calls lost during serialization in InMemoryChatMessageHistory.* https://github.com/langchain-ai/langchain/issues/34925

[29] 2026. *Haystack.* https://github.com/deepset-ai/haystack

[30] 2026. *LiteLLM.* https://pypi.org/project/litellm

[31] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 135 (Sept. 2023), 31 pages. doi:10.1145/3587155

[32] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences.* routledge. doi:10.4324/9780203771587

[33] Hana Derouiche, Zaki Brahmi, and Haithem Mazeni. 2025. Agentic AI frameworks: Architectures, protocols, and design challenges. *arXiv preprint*

arXiv:2508.10146 (2025). doi:10.48550/arXiv.2508.10146

[34] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. 2004. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery* 8, 1 (2004), 53–87. doi:10.1023/B:DAMI.0000005258.31418.83

[35] Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E. Hassan. 2026. An empirical study of testing practices in open source AI agent frameworks and agentic applications. *Empirical Software Engineering* 31, 5 (apr 2026), 124. doi:10.1007/s10664-026-10857-9

[36] Niful Islam, Ragib Shahriar Ayon, Deepak George Thomas, Shibbir Ahmed, and Mohammad Wardat. 2026. When Agents Fail: A Comprehensive Study of Bugs in LLM Agents with Automated Labeling. *arXiv preprint arXiv:2601.15232* (2026). doi:10.48550/arXiv.2601.15232

[37] Nan Li and Jeff Offutt. 2017. Test Oracle Strategies for Model-Based Testing. *IEEE Transactions on Software Engineering* 43, 4 (2017), 372–395. doi:10.1109/TSE.2016.2597136

[38] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (San Jose, California) *(ASID '06)*. Association for Computing Machinery, New York, NY, USA, 25–33. doi:10.1145/1181309.1181314

[39] Daniel Liu, Krishna Upadhyay, Vinaik Chhetri, A.B. Siddique, and Umar Farooq. 2025. A Large-Scale Study on the Development and Issues of Multi-Agent AI Systems. In *2025 IEEE International Conference on Big Data (BigData)*. 7785–7792. doi:10.1109/BigData66926.2025.11402104

[40] Jerry Liu. 2022. *LlamaIndex*. https://github.com/run-llama/llama_index

[41] Mugeng Liu, Siqi Zhong, Weichen Bi, Yixuan Zhang, Zhiyang Chen, Zhenpeng Chen, Xuanzhe Liu, and Yun Ma. 2026. A First Look at Bugs in LLM Inference Engines. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2026). Just Accepted. doi:10.1145/3788873

[42] Ruofan Lu, Yichen Li, and Yintong Huo. 2025. Exploring Autonomous Agents: A Closer Look at Why They Fail When Completing Tasks. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 3856–3860. doi:10.1109/ASE63991.2025.00330

[43] M.L. Menéndez, J.A. Pardo, L. Pardo, and M.C. Pardo. 1997. The Jensen-Shannon divergence. *Journal of the Franklin Institute* 334, 2 (1997), 307–318. doi:10.1016/S0016-0032(96)00063-4

[44] Melissa Z Pan, Mert Cemri, Lakshya A Agrawal, Shuyi Yang, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Kannan Ramchandran, Dan Klein, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. 2025. Why Do Multiagent Systems Fail?. In *ICLR 2025 Workshop on Building Trust in Language Models and Applications*. https://openreview.net/forum?id=wM521FqPvI

[45] Rui Ren, Jinheng Li, Yan Yin, and Shuai Tian. 2021. Failure Prediction for Large-Scale Clusters Logs via Mining Frequent Patterns. In *Intelligent Computing and Block Chain*, Wanling Gao, Kai Hwang, Changyun Wang, Weiping Li, Zhigang Qiu, Lei Wang, Aoying Zhou, Weining Qian, Cheqing Jin, and Zhifei Zhang (Eds.). Springer Singapore, Singapore, 147–165. doi:10.1007/978-981-16-1160-5_13

[46] Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 2025. Smolagents: a smol library to build great agentic systems. https://github.com/huggingface/smolagents.

[47] Guogen Shan and Shawn Gerstenberger. 2017. Fisher's exact approach for post hoc analysis of a chi-squared test. *PloS one* 12, 12 (2017), e0188709. doi:10.1371/journal.pone.0188709

[48] 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. doi:10.1145/3468264.3468591

[49] Susana M. Vieira, Uzay Kaymak, and João M. C. Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *International Conference on Fuzzy Systems*. 1–8. doi:10.1109/FUZZY.2010.5584447

[50] Haibo Wang, Zhuolin Xu, Huaien Zhang, Nikolaos Tsantalis, and Shin Hwei Tan. 2025. Towards Understanding Refactoring Engine Bugs. *ACM Trans. Softw. Eng. Methodol.* (July 2025). Just Accepted. doi:10.1145/3747289

[51] Hironori Washizaki (Ed.). 2024. *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0*. IEEE Computer Society. https://www.swebok.org

[52] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *First Conference on Language Modeling*. https://openreview.net/forum?id=BAakY1hNKS

[53] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1319–1331. doi:10.1145/3597926.3598138

[54] Ziluo Xue, Yanjie Zhao, Shenao Wang, Kai Chen, and Haoyu Wang. 2025. A Characterization Study of Bugs in LLM Agent Workflow Orchestration Frameworks. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 3369–3380. doi:10.1109/ASE63991.2025.00278

[55] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=WE_vluYUL-X

[56] Daojuan Zhang, Tianqi Wu, Xiaoming Zhou, Bo Hu, and Wenjie Zhang. 2024. Multi-source System Log Behavior Pattern Mining Method Based on FP-Growth. In *Proceedings of the 2023 International Conference on Communication Network and Machine Learning* (Zhengzhou, China) *(CNML '23)*. Association for Computing Machinery, New York, NY, USA, 248–254. doi:10.1145/3640912.3640961

[57] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *Proc. ACM Softw. Eng.* 1, FSE, Article 33 (July 2024), 23 pages. doi:10.1145/3643759

[58] Zhang, Xiaowen and Zhang, Hannuo. 2026. Understanding Bugs in Modern Agentic Frameworks (Artifact). Zenodo. doi:10.5281/zenodo.21364480

Received 2026-03-26; accepted 2026-06-18