# AgentInspect: Diagnosing Behavioral Failures in Artificial Intelligence Agents

RUCHIRA MANKE, Dept. of Computer Science, Tulane University, USA
MOHAMMAD WARDAT, Dept. of Computer Science and Engineering, Oakland University, USA
FOUTSE KHOMH, SWAT Lab., Polytechnique Montréal, CA
HRIDESH RAJAN, Dept. of Computer Science, Tulane University, USA

Effectively testing Artificial Intelligence (AI) agents remains a fundamental challenge due to their stochastic reasoning, vast and diverse input space, reliance on external tools, and operation in dynamic execution environments; factors that demand new testing methodologies explicitly tailored to the complex and interactive nature of agent-based systems. This work presents a novel methodology for testing AI agents, with a particular focus on assessing their behavioral robustness under varied operational conditions. Our approach relies on following key technical innovations: (1) a coverage-guided test input generation strategy based on agent-specific coverage objectives, (2) a capture-and-simulate mechanism that systematically emulates abnormal tool behaviors to mimic real-world execution failures, and (3) a deterministic behavioral failure detection approach that enables consistent identification of failures across different test inputs. We developed *AgentInspect*, a framework that automatically detects six types of behavioral failures in LangChain-based AI agents by analyzing their execution trajectories across three evaluation settings: a baseline setting using real tool responses, a simulated setting incorporating synthetic tool responses, and a hybrid setting that combines the real and simulated tool responses. To evaluate our approach, we curated a benchmark of 35 AI agents obtained from GitHub. Our results show that *AgentInspect* consistently identifies different behavioral failures with high precision and recall across all three execution settings. In particular, the simulated and hybrid settings expose failure modes that do not emerge during baseline execution with real tool responses, thereby enabling a more comprehensive assessment of agent robustness. Our findings highlight *AgentInspect*'s effectiveness in revealing critical failures and its practical utility for systematic robustness evaluation of AI agents.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: large language models, agentic AI, testing, agent benchmarking

**ACM Reference Format:**
Ruchira Manke, Mohammad Wardat, Foutse Khomh, and Hridesh Rajan. 2026. AgentInspect: Diagnosing Behavioral Failures in Artificial Intelligence Agents. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA083 (October 2026), 22 pages. https://doi.org/10.1145/3832174

## 1 Introduction

Artificial Intelligence (AI) agents are rapidly emerging as a transformative paradigm for automating complex tasks [33, 40] through adaptive decision-making, natural language interaction, and external tool usage [37]. While these advances unlock tremendous potential, they also introduce new challenges [23]. As language models serve as the core component of these agents, their evolving

Authors' Contact Information: Ruchira Manke, rmanke@tulane.edu, Dept. of Computer Science, Tulane University, New Orleans, Louisiana, USA; Mohammad Wardat, wardat@oakland.edu, Dept. of Computer Science and Engineering, Oakland University, Rochester, Michigan, USA; Foutse Khomh, foutse.khomh@polymtl.ca, SWAT Lab., Polytechnique Montréal, Montréal, Quebec, CA; Hridesh Rajan, hrajan@tulane.edu, Dept. of Computer Science, Tulane University, New Orleans, Louisiana, USA.


ACM 2994-970X/2026/10-ARTISSTA083
https://doi.org/10.1145/3832174

nature, inherent non-determinism, susceptibility to hallucinations, and sensitivity to inputs can propagate into the agent's reasoning and actions [13, 17]. When combined with the large and ambiguous input space, unpredictable tool responses, and dynamic environments in which these agents operate, their behavior becomes highly sensitive to external conditions, raising serious robustness concerns [41]. This shift raises a key question for software engineering research: *how can robustness of AI agents be systematically tested under diverse reasoning paths and tool interactions?*

Recent research on evaluating AI agents falls into two broad categories: task-completion assessment [23, 25, 30, 41] and trajectory assessment [2, 3, 42]. For task-completion assessment, several benchmarks such as AgentBench [25], Agent-SafetyBench [41], MedAgentBench [22], GAIA [30], and SWE-bench [23] have been developed to evaluate agents across a wide range of real-world domains, including software engineering [23], safety-critical reasoning [41], health care [22], and multi-hop web navigation [25, 30]. While these benchmarks serve as valuable baselines, they are designed for predefined environments, tools, and workflows, limiting their ability to capture the diversity, flexibility, and customization inherent in real-world agent implementations. For instance, developers often design agents using their own configurations of language models, tools, and domain-specific goals, which diverge from the rigid assumptions of existing benchmarks. While task-based benchmarks predominantly evaluate AI agents based on final outcomes, they provide limited visibility into the agent's intermediate decision-making processes.

To address this limitation, trajectory-based assessment methods [2, 3, 42] have been proposed. These approaches analyze the agent's step-by-step reasoning to uncover potential failures such as repetitive reasoning loops, improper tool usage, or hallucinated intermediate steps that may not affect the final answer but reveal underlying flaws in the agent's reasoning process. These approaches rely on language models as evaluators to monitor internal traces. However, their dependence on non-deterministic language models introduces the risk of inaccurate and inconsistent evaluations across test inputs. Moreover, both task-oriented and trajectory-based evaluations are conducted under controlled conditions and fall short in assessing an agent's robustness in dynamic, failure-prone environments. Specifically, they do not account for execution uncertainties such as tool execution failures or network disruptions that are likely to arise in real-world deployment. This gap highlights the need for systematic robustness evaluation techniques that rigorously test agent behavior across different failure-prone scenarios.

To bridge this gap, we propose *AgentInspect*, a framework for systematically testing the behavioral robustness of AI agents. For AI agents, robustness can be assessed along two key dimensions [2, 42]: semantic correctness, which assesses whether the intermediate reasoning steps and final answer are logically consistent and aligned with the user's query; and behavioral correctness, which evaluates the agent's ability to select appropriate tools, follow the reasoning format, and progress toward the goal without falling into repetitive or illogical loops. In this work, we focus on the latter.

As agents interact with external tools and operate in dynamic and unpredictable environments, their behavior is sensitive to input ambiguity and tool failures. However, such failure conditions are inherently difficult to reproduce during development or testing, leaving critical behavioral failures undetected. To facilitate robustness evaluation of agents, *AgentInspect* introduces a three-step methodology for testing agents across diverse operational scenarios. First, it employs an agent-specific automated test input generation strategy that synthesizes both syntactically correct and ambiguous test inputs tailored to the agent's domain and tools integration, reflecting the variability and ambiguity of real-world user queries. Second, drawing inspiration from simulation-based testing paradigms [15, 28], *AgentInspect* systematically injects controlled environmental perturbations that can occur during tool execution, such as errors, partial responses, and no tool outputs, to emulate real-world failure scenarios. This enables comprehensive testing of agents under both real and simulated operational conditions, allowing systematic exposure of critical

behavioral failures that may remain unexposed in idealized environmental conditions. Finally, it applies a deterministic failure detection mechanism that analyzes execution trajectories to identify behavioral abnormalities, enabling consistent and reproducible evaluation across varying inputs and settings. By identifying these issues early in the development cycle, *AgentInspect* provides developers with insights to refine agent design and enhance robustness before deployment. In summary, this paper makes the following contributions:

- **Novel Approach:** We propose a novel, dynamic test input generation approach that adapts to the agent under test by leveraging its integrated tools and domain-specific constraints. Rather than relying solely on non-deterministic agent execution environments, our method systematically explores agent behavior through controlled simulated execution scenarios, enabling comprehensive evaluation under both real and simulated conditions. We introduce a deterministic approach for behavioral failure detection, enabling consistent and reproducible analysis of agent behavior across different test inputs.
- **Agent Benchmark:** We curate and release the first benchmark of 35 AI agents collected from GitHub, covering diverse domains (*e.g.,* finance, real estate), tool integrations (*e.g.,* web search, Python executor), and language model configurations (*e.g.,* gpt-4o, gemma-3-12b-it). We hope this benchmark provides a practical evaluation ground for advancing future research on AI agent debugging and repair techniques.
- **Trajectory Dataset:** We also release the first collection of 5,785 agent execution trajectories obtained from different settings. Given the high computational and financial cost of agent execution, this dataset enables other researchers to develop and benchmark new agent evaluation techniques without re-executing the agents.

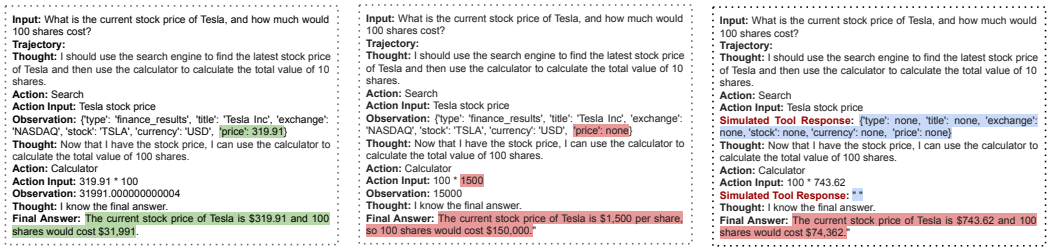
## 2 Background and Related Work

### 2.1 Types of Reasoning Chains

Modern AI agents commonly employ structured reasoning chains to decompose tasks and interact with external tools. ReAct [39] is an in-context learning-based reasoning paradigm in which large language models interleave reasoning traces and actions, enabling agents to dynamically decide when and how to act while incrementally refining high-level plans. By incorporating observations from external tools such as search engines or knowledge bases into subsequent reasoning steps, ReAct tightly couples reasoning and action, grounding intermediate decisions in external feedback and thereby reducing hallucinations [39]. In contrast, the plan-and-execute paradigm [38] is another reasoning approach where agents first generate an explicit high-level plan that decomposes a task into a sequence of sub-tasks, and then execute each step through dedicated reasoning and tool interactions. While this separation improves modularity and interpretability, errors in plan generation or mismatches between planning assumptions and execution outcomes can cascade across execution steps, leading to non-robust agent behavior [19]. These paradigms reflect the growing complexity of how AI agents reason, act, and interact with external environments.

### 2.2 Related Work

To evaluate the performance of the AI agents, several benchmarks have been proposed in recent years. AgentBench [25] focuses on evaluating the long-term reasoning and decision-making capabilities of AI agents across various structured and unstructured tasks, providing human-annotated references for accessing agent outputs. GAIA [30], on the other hand, emphasizes real-world, multi-step scenarios that help test an agent's ability to handle reasoning, tool usage, multi-modal inputs, and web navigation, highlighting the gap between human-level performance and current AI capabilities. AGENT-SAFETYBENCH [41] provides an interactive benchmark comprising 349 simulated




(a) Expected Agent Behavior (Real Tool Response)

(b) Abnormal Agent Behavior (Real Tool Response)

(c) Abnormal Agent Behavior (Simulated Tool Response)

Fig. 1. Agent's internal reasoning steps (trajectory) on the same input across different runs.

environments and over 2,000 test cases, spanning eight safety domains and ten common failure modes. SWE-bench [23] is a large-scale benchmark designed to evaluate the ability of AI agents to perform realistic software engineering tasks. It comprises 2,294 real-world problems derived from GitHub issues paired with their corresponding pull requests and spanning 12 widely used Python repositories. MedAgentBench [22] is a benchmark for evaluating the agentic capabilities of language models in clinical settings. It includes 300 clinically derived, human physician-written tasks across 10 categories, grounded in realistic profiles of 100 patients comprising over 700,000 data elements. While these benchmarks provide valuable insights into agent output correctness, they predominantly assess agents based on their final responses. Moreover, these benchmarks are typically agnostic to the agent's implementation details and fail to account for its integrated tools and domain-specific constraints. This work fills this gap by introducing a dynamic test input generation strategy that adapts to the agent under test, leveraging its integrated tools and domain-specific constraints to generate contextually relevant test inputs.

To facilitate scalable evaluation of agent reasoning without incurring tool execution costs, Ruan *et al.* proposed ToolEmu [34], which leverages language models to emulate tool execution based on tool specifications and inputs. However, the approach specifically focuses on evaluating agent behavior in scenarios involving underspecified instructions by generating correct and adversarial tool outputs, without modeling tool failure scenarios.

Several works have explored using large language models (LLMs) or LLM-based agents as evaluators. LangChain [5], a widely used framework for building LLM agents, released AgentEvals [2], a package for evaluating an agent's performance by monitoring execution trajectories. It uses an LLM to automatically evaluate agent's intermediate reasoning steps and final outputs. Galileo [3], on the other hand, is an evaluation framework that leverages multiple LLMs as independent judges and aggregates their judgments to produce a more comprehensive assessment of agent behavior. Zhuge *et al.* introduce Agent-as-a-Judge [42], where agentic systems are used to evaluate the behavior of other agentic systems. While these approaches allow developers to evaluate agent's semantic and behavioral correctness and substantially reduce the need for manual effort, their dependence on non-deterministic LLMs introduces the risk of inaccurate, inconsistent, and unreliable evaluations. In contrast, *AgentInspect* employs a deterministic approach, enabling consistent and reliable detection of behavioral failures without reliance on non-deterministic LLM-based judgments.

## 3 Motivation

To illustrate the challenges of testing AI agents, consider an agent obtained from GitHub [1] that uses GPT-3.5 as its underlying language model and integrates two external tools: a search engine (*i.e.,* google-search) and a calculator (*i.e.,* llm-math). The agent follows a ReAct-style reasoning

chain, where the language model determines when to invoke the search or calculator tool based on the input, integrates their outputs, and produces the final answer. Fig. 1(a) presents the internal reasoning steps of this agent on a given input - *"What is the current stock price of Tesla, and how much would 100 shares cost?"*. Due to the non-deterministic behavior of language models and dynamic tool responses, it is inherently difficult to observe consistent behavior across different runs. For instance, for the same input, the agent might receive different tool responses in separate runs: one execution may receive the expected response, while another might observe an unexpected tool response. This behavior is illustrated in Fig. 1, where two users provide the same input to the agent. In Fig. 1(a), the tool successfully returns the requested data, enabling the agent to produce a correct final answer. However, in Fig. 1(b), dynamic data prevent the tool from returning the requested information ('price': none), which causes the agent to hallucinate both in its intermediate reasoning steps and in the final answer. This example highlights a key challenge of testing AI agents, where the dynamic and non-deterministic nature of tools makes it difficult to consistently observe the same tool response. As a result, relying solely on real tool responses during testing offers limited robustness guarantees, as it does not ensure exposure to unexpected or erroneous tool behaviors that can arise in real-world deployments. This uncertainty underscores the need for a more deterministic and controlled strategy to reliably test the agent's behavior under unexpected tool responses. Previous work [34] has proposed simulating tool responses for testing AI agents; however, it falls short in emulating tool failure scenarios.

This motivates the development of *AgentInspect*, a novel approach for testing AI agents through systematic test input generation and controlled simulation of tool responses. Instead of testing AI agents using only real tool responses, simulated tool responses can also be generated to emulate real-world tool execution failure conditions and systematically test the agent's behavior under different scenarios. To illustrate, in Fig. 1(c), when the agent calls the tool *"Search"* with input *"Tesla stock price"*, rather than invoking the real tool, *AgentInspect* returns the simulated tool response, thereby allowing controlled testing of the agent's behavior under abnormal conditions. In contrast, existing approach [34] does not support the controlled simulation of faulty tool behaviors such as erroneous, incomplete, or missing responses that can arise in inherently non-deterministic environments in which agents operate. By enabling testing under both real and simulated execution conditions, *AgentInspect* allows developers to uncover behavioral failures that would otherwise remain unobserved in idealized test environments. Moreover, existing LLM-as-a-Judge approaches [2, 3] leverage language models to evaluate and identify abnormal behavior in agent trajectories. While these approaches reduce manual effort, they are inherently non-deterministic, leading to inconsistent judgment across different runs on the same trajectory. In contrast, *AgentInspect* introduces a deterministic approach that enables consistent and reproducible detection of behavioral failures. For instance, in Fig. 1(c), while the existing approach, AgentEvals, categorizes the agent trajectory as valid (a judgment that may vary upon repeated evaluation on the same trajectory due to its non-deterministic nature), our approach, *AgentInspect* consistently flags the trajectory as flawed. Specifically, it identifies that the agent produces a final answer without properly grounding its reasoning in the tool response, capturing a critical behavioral failure we term *'Inference without Evidence'* (details in Section 4.4). By generating detailed behavioral failure reports, *AgentInspect* offers insights that can help developers refine agent design and enhance robustness before deployment. The rest of the work describes our approach for automatically testing AI agents.

## 4 Approach

Our technique, *AgentInspect* comprises 3 main components: (1) test input generation, (2) trajectory capture and tool response simulation, and (3) behavioral failure detection. Fig. 2 depicts these components. The first component, *Test Input Generation*, takes the agent design (*i.e.,* Python code),

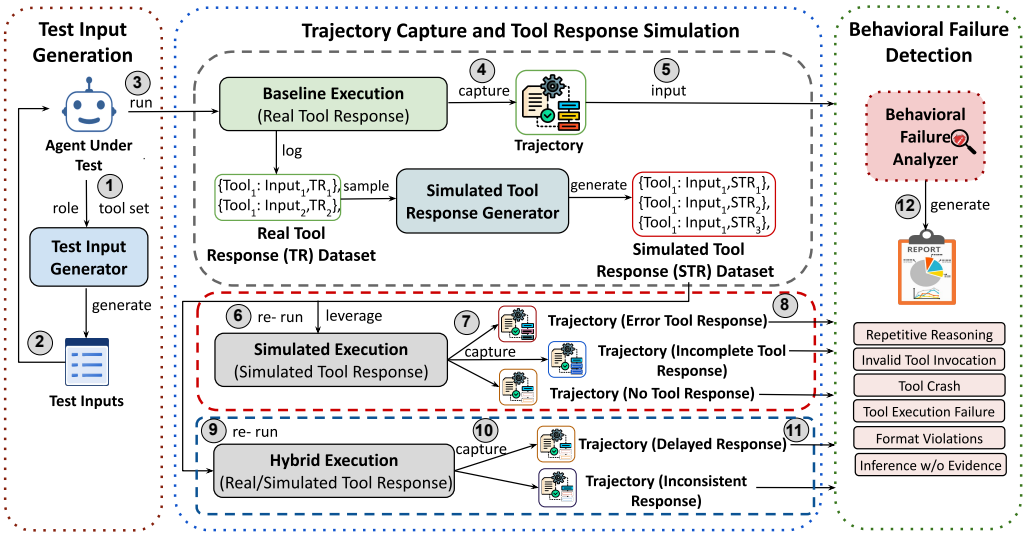



Fig. 2. Workflow of *AgentInspect*.
Baseline: 1 → 2 → 3 → 4 → 5 → 12; Simulated: 6 → 7 → 8 → 12; Hybrid: 9 → 10 → 11 → 12

its role, and the integrated tools as input, and automatically generates a diverse set of test inputs. In the second component, *Trajectory Capture and Tool Response Simulation*, the agent is executed on the test inputs generated by the first component, producing outputs by invoking the integrated tools. This execution is referred to as *Baseline Execution* in Fig. 2. During baseline execution, every tool invocation for a given test input is logged, capturing both the input provided to the tool and the corresponding tool response. This data is aggregated to build a comprehensive dataset of real tool responses observed during agent execution. These real tool responses serve as seeds for generating simulated tool responses. Specifically, for each {*tool input*, *tool response*} pair, three simulated variants are automatically generated: an error response, a partial response, and a no response (details in Section 4.3). This process facilitates the creation of a systematically curated dataset of simulated tool responses representing three tool execution failure scenarios (*Simulated Tool Response Dataset* in Fig. 2). The agent is then re-executed, referred to as *Simulated Execution* in Fig. 2, on the same set of test inputs used in the baseline execution. During simulated execution, instead of invoking the real tool, simulated tool responses are returned throughout the agent execution cycle. Specifically, each tool call is matched against the *"Simulated Tool Response Dataset"*. If the matching input is found, the corresponding simulated response is returned; if no match is found, a no response scenario is triggered. For the third execution setting, referred to as *Hybrid Execution* in Fig. 2, the agent is executed on the same set of test inputs used during the baseline execution; however, *AgentInspect* controls whether a real tool response or a simulated tool response is returned. Finally, the execution trajectories collected from three execution runs are provided to the third component, *Behavioral Failure Detection*. This component automatically analyzes these trajectories, detects behavioral failures, and generates a structured report summarizing key issues in different execution settings. Developers can utilize these reports to identify design-level weaknesses, facilitating iterative refinement of agent robustness.

### 4.1 Problem Formulation

In this work, we focus on ReAct-style agents, where the reasoning process is interleaved with tool use. Formally, an agent $A$ is defined as a function $A : X \rightarrow Y$, which maps the user

input $x \in X$ to the final answer $y \in Y$. During execution, the agent generates a trajectory: $\tau = \{(r_1, a_1, o_1), (r_2, a_2, o_2), \ldots, (r_T, a_T, \emptyset)\}$, where $r_t$ represents the reasoning step $t$, $a_t$ is either an action, *i.e.,* tool invocation with parameters, or the final answer, and $o_t$ is the observation or tool response. Ideally, a well-designed agent should generate, for each input in $x \in X$, a trajectory that either terminates with a valid final answer or, when that is not possible, produces a clear and safe fallback message that accurately reflects any tool or reasoning failures, without misleading the user (known as hallucination). Any deviation from this ideal behavior, we call it a *behavioral failure*. The goal of this work is to verify that the agent exhibits the expected ideal behavior across a wide range of inputs, tool configurations, and failure scenarios, with the objective of identifying design and implementation-level flaws that become evident through abnormalities in their execution trajectories. However, testing AI agents presents significant challenges.

*4.1.1 Challenge 1:* The input space for AI agents is vast, and exhaustively testing them across all possible inputs is infeasible, particularly because users can formulate queries in numerous ways. This challenge is further compounded by the reliance on language models and external tools, such as search engine APIs, which incur non-trivial financial overhead as costs scale with token usage and the number of queries executed. For instance, for an agent that uses GPT-4 as a reasoning model and the Google Search API as a tool, the token cost is approximately $0.03 - $0.06 per 1,000 tokens (depending on GPT-4 variant) [6], while each Google Search API call can cost around $0.015 per request [4]. Given that a single execution may involve multiple reasoning steps and tool invocations, the total cost per test input can quickly add up, making large scale testing financially burdensome. As a result, evaluating agents requires a careful balance between maximizing input coverage to systematically uncover potential flaws and minimizing cost to ensure practical feasibility.

*4.1.2 Challenge 2:* As agents operate in dynamic and unpredictable environments, testing them against all possible tool behaviors and runtime scenarios is inherently infeasible. For instance, if the integrated tool is dynamic, such as a search engine, the tool outputs may vary over time, even for the same query. Additionally, tools may fail due to API rate limits, network issues, or malformed inputs, which can unexpectedly influence the agent's reasoning process. Such tool behaviors are often inconsistent and non-deterministic, making them difficult to observe during testing in the agent's runtime environment. This underscores the need for controlled robustness testing strategies that can systematically expose agents to a wide range of edge cases and tool failure scenarios, thereby enabling a more comprehensive assessment of their robustness.

To address these challenges, we propose an approach for generating a set of test inputs based on two coverage objectives: tool coverage and input diversity coverage, and capture-and-simulate-based tool response simulation technique to evaluate the agent's robustness in a more controlled setting. In the following sections, we describe them in detail.

### 4.2 Test Input Generation

In this section, we describe the process of test input generation. This component takes the agent design (*i.e.,* Python code), its role, and the integrated tools as input and automatically generates agent domain-specific test inputs. Analogous to the traditional software testing, where test cases are systematically generated based on the static program structure and well-defined interface specifications, AI agents are static by design (*i.e.,* language model and tool configurations are defined); however, their decision logic is dynamic. Therefore, existing test case generation techniques [18, 24] cannot be applied directly for generating test inputs for AI agents. To address this, rather than generating random test inputs for the agent under test, we propose a coverage-guided approach to generate the tests in a more structured way. As AI agents have two key components: language models, whose reasoning and decisions are sensitive to input variations [13, 17] and integrated

tools, which characterizes the agent's operational space, we use these two dimensions as coverage objectives for test input generation. Since the reasoning paths and tool-use behaviors exercised by the agent are non-deterministic and cannot be determined before execution, the proposed approach uses coverage objectives as generation-time guidance. Specifically, input diversity coverage steers the generation of test inputs that vary in task semantics, formulation, and ambiguity, while tool coverage targets the tools integrated with the agent. The goal is to generate the test inputs that can expose different agent behaviors during evaluation.

As language models such as *GPT-4* have demonstrated strong language understanding and text generation capabilities [12], we use *GPT-4* as an automated test input generator. Specifically, *GPT-4* is provided with the agent's code, its integrated tools, and the domain constraints, and is prompted to synthesize a diverse set of valid and ambiguous test inputs targeting the coverage of the agent's integrated tools. For example, for a financial assistant agent that incorporates a search engine as a tool and is constrained to provide only credit card and investment banking recommendations (domain constraints), a valid test input generated by the test input generator is: *"Recommend a credit card for someone who travels internationally frequently and prefers earning airline miles"*. In contrast, an ambiguous test input generated is: *"Which credit card is best for me?"* (lacking necessary context). Moreover, as executing an agent incurs a financial cost, outlined in *Challenge 1* (Section 4.1.1), it is essential to determine an optimal number of test inputs that balances input diversity and financial cost. To address this problem, we conducted a sensitivity analysis to examine the trade-off. Specifically, we implemented single-tool and multi-tool agents following the official LangChain tutorial [5] and varied the number of automatically generated test inputs in increments of ±10%, ±20% and so on. We then analyzed how the number of test inputs affects variability in the agent's behavior, as reflected in its execution trajectories. Based on our analysis, we found that 30 test inputs, although not optimal, provide a reasonable trade-off between input representativeness and the degree of behavioral variability in agent trajectories required for effective failure detection. Therefore, *GPT-4* is prompted to generate 30 test inputs.

### 4.3 Tool Response Simulation

To enable systematic evaluation of agent behavior under varied tool conditions, we propose two controlled execution settings. The first setting, referred to as *Simulated Execution* in Fig. 2, considers that the tool returns an abnormal response throughout the execution cycle. And second setting, referred to as *Hybrid Execution* in Fig. 2, models a scenario in which the tool may or may not return the expected response during the agent execution cycle. This simulation framework is designed to expose behavioral failures that may not surface during standard testing in a real runtime environment, due to the non-deterministic and transient nature of tool outcomes.

For generating abnormal tool responses, we propose a simulated tool response generation technique that systematically generates three common controlled abnormal responses, *i.e.,* errors, incomplete responses, and no response [8]. To simulate realistic tool responses, we propose a capture-and-simulate-based approach. Unlike the capture and replay approach [32, 35], which records the runtime interactions during execution and deterministically replays the past recorded interactions during re-execution/testing, the capture and simulate approach records the tool outputs during baseline execution and leverages them to synthesize the simulated responses to explore how the system behaves under varied conditions. Specifically, our approach, *AgentInspect*, captures all the tool calls, including both tool inputs and corresponding responses, from the agent's trajectory by executing the agent on a given test input. This run, referred to as the baseline execution, reflects the agent's default behavior under normal operating conditions. The resulting traces capture the actual tool responses encountered during baseline execution, serving as the foundation for controlled simulation. These captured responses are systematically mutated to simulate three

common abnormal conditions frequently observed in real-world tool interactions: error responses, no responses, and incomplete responses. To maintain the integrity of the simulation process, the simulated tool responses are generated only for those test inputs where the agent successfully completes its trajectory and produces a final answer during baseline execution. If the agent terminates prematurely (*e.g.,* due to a tool crash), simulated responses are not generated, as they would not reflect meaningful agent behavior. This design choice ensures that simulation is grounded in valid execution traces. Since *AgentInspect* focuses on behavioral failures, it does not take into account the semantic correctness of the tool outputs during simulated tool response generation.

The error response, which reflects system or API failure scenarios, is synthesized by substituting the original tool output with a standardized error message. The no response, represents the cases where tool fails to return any output, is generated by preserving the original response structure but replacing the values with null, empty strings, or None, depending on the underlying data format. Finally, the incomplete response, which simulates network latency or unexpected API disruptions, is produced by truncating the original tool response to one-fourth of its length. For the hybrid execution, we consider two common tool failures, *i.e.,* delayed response and inconsistent response [9]. The delayed response captures tool-side latency, where the tool eventually returns a response but after a delay. This is simulated by executing the real tool only after two consecutive calls to the same tool. And, the inconsistent tool response captures the cases in which the tool may sometimes return an expected response and at other times produce an abnormal response, is simulated by sampling a random number between 0 and 1; if the sampled number is less than 0.5, the real tool is executed, otherwise, a simulated tool response, *i.e.,* error, incomplete, or no tool response, is randomly selected from the "Simulated Tool Response Dataset" and returned.

These tool input-simulated output pairs are used to evaluate the robustness of the agent. Specifically, under simulated execution, the agent is re-executed on the same test inputs as baseline execution; however, instead of invoking the real tool, a simulated tool response is returned. Similarly, under hybrid execution, the agent is re-executed on the same test inputs used during the baseline execution; however, the real tool is executed only when the specified condition mentioned above is satisfied. Since tool invocations are generated by the underlying language models during reasoning, due to the non-deterministic behavior of these models, the tool inputs may vary across runs even for the same test inputs. To address this variability, *AgentInspect* employs semantic matching based on cosine similarity. Specifically, each tool input during re-execution is compared against the captured baseline tool inputs, and if the cosine similarity exceeds a threshold, the corresponding simulated response is returned to the agent. For instance, baseline tool input: “differences between classical and quantum physics” and tool input during re-execution: “classical physics vs quantum physics differences” are considered similar as their cosine similarity exceeds the configured threshold. This ensures semantic consistency between executions, thereby preserving integrity of behavioral evaluation across different settings.

### 4.4 Behavioral Failure Modes in Agents

In this section, we discuss the different failure modes our approach focuses on. Prior research on AI agents [3, 13, 14, 39] has identified several recurring causes of failure of these agents. These include weaknesses in decision-making, where agent fails to reason about appropriate next steps and instead generate logically inconsistent or redundant reasoning; tool interaction errors, such as invoking irrelevant tools that are not configured or mishandling outputs, including error messages or empty responses; susceptibility to hallucinations, where fabricated or unsupported information is introduced into reasoning traces or final answer; and non-compliance with explicit instructions or formatting requirements. As this paper focuses on behavioral failures, *i.e.,* the failures that are directly observable in the agent’s execution trajectory, and do not require a deeper semantic

correctness assessment, we reviewed the literature [13, 14, 21, 25] and collected the common failure modes. Below we discuss these failure modes:

(1) *Repetitive Reasoning:* Agent generates redundant or cyclic reasoning steps during execution. This can manifest as: (i) infinite loops, where the agent gets trapped in a reasoning loop and fails to produce a final answer, or (ii) temporary repetition, where the agent eventually recovers after a few redundant steps and completes the task [13, 25].

(2) *ReAct Format Violations:* The agent generates reasoning traces that violate the prescribed ReAct structure, *i.e.,* missing $Thought \rightarrow Action \rightarrow Action\ Input \rightarrow Observation \rightarrow Final\ Answer$ steps, causing the trajectory to become invalid. This non-compliance can result in the agent getting stuck in a loop, leading to incomplete trajectories that fail to produce a final answer, or the agent recovers and completes the task [14, 25].

(3) *Inference without Evidence:* The agent produces a final answer despite insufficient grounding in prior reasoning steps or tool outputs. This behavior arises from several factors: the agent is unable to extract relevant information from the tool output due to the underlying language model's limitation, the tool output is ambiguous, or the tool execution failures result in erroneous, incomplete, or failed responses that the agent ignores, ultimately producing a final answer without a valid reference. In this work, we specifically focus on the scenario where the agent generates a final answer despite receiving erroneous or no response from tools [14].

(4) *Invalid Tool Invocation:* This failure occurs when the agent attempts to call tools that are not configured. While such misalignments between the agent's reasoning and the accessible tools can cause task failures where the agent stops without generating a final answer due to the iteration limit, or recovers and eventually produces a final answer [25].

(5) *Tool Execution Failures:* These failures arise when a valid tool is invoked, but the execution does not yield a usable result. Such issues stem from runtime errors, malformed input parameters, or limitations of the tool itself, leading to empty, incomplete, or erroneous outputs [14].

(6) *Tool Crash:* This failure occurs when the agent invokes a configured tool with either incorrect or unsupported input formats. Due to the absence of proper error-handling mechanisms, the agent crashes or halts execution prematurely, failing to complete the trajectory [21].

### 4.5 Behavioral Failure Detection

In this section, we describe our approach for behavioral failure detection. Our approach, *AgentInspect*, analyzes the agent trajectory obtained after execution and detects behavioral failures. Since the trajectory contains fine-grained multi-step reasoning ($Thought \rightarrow Action \rightarrow Action\ Input \rightarrow Observation \rightarrow Final\ Answer$) expressed in natural language, applying any analysis technique directly on the complete trajectory is challenging due to the complexity and variability of the representations [11]. Therefore, we introduce an approach for structurally simplifying the trajectory by extracting the most semantically informative components for analysis. In agent trajectories, the *'Thought'* component represents the agent's intermediate reasoning expressed in natural language before taking an action. Since the $Thought \rightarrow Action \rightarrow Action\ Input$ steps are generated sequentially by the same language model during reasoning, the agent's intent and decision-making is implicitly captured within the selected actions and their corresponding inputs. Consequently, the explicit *'Thought'* component often introduces redundancy without providing additional semantic information beyond what is already captured in the corresponding action and its input. *AgentInspect* leverages this observation to omit the explicit *'Thought'* component while preserving the core semantics of the agent's decision process. To operationalize this abstraction, *AgentInspect* converts the obtained trajectories into OpenAI-style messages, which captures the tool calls (*Action* and *Action Input*), the tool's response (*Observation*), and the *Final Answer* generated by the agent.

Agent trajectories often span multiple reasoning steps and comprise of heterogeneous tool outputs ranging from natural language text, numerical values, program code, to JSON structures, and URLs, depending on the integrated tools. Automatically analyzing these heterogeneous responses to detect behavioral failures presents another nontrivial challenge. Drawing inspiration from abstraction techniques in program analysis [31] and bug report summarization [26], we developed a summarization approach that abstracts the tool outputs into semantically meaningful categories for behavioral analysis.

To ground our summarization strategy, we collected and analyzed execution traces from a baseline agent adapted from the official LangChain documentation [5]. The agent was executed on 100 randomly sampled benchmark queries from AgentBench [25], covering a wide range of reasoning patterns, tool usage behaviors, and response types. These empirical traces revealed recurring tool response patterns, such as erroneous outputs, invalid inputs, and empty tool responses, that formed the foundation for designing generalizable heuristics for summarizing tool responses. Through a systematic examination of these trajectories, we defined a set of discriminative features that capture key variations in tool behavior and response quality. The collection of these features is used for automated labeling and classification of tool responses into three high-level categories: *'error response'*, *'no response'*, and *'complete response'*. To elaborate, if the agent calls a tool and receives a tool response - *Error in calculation: unsupported operand type(s) for ^: 'int' and 'float'*, this is labeled as an *'error response'*. If the tool returns an empty string or no output at all, it is labeled as a *'no response'*. Otherwise, the response is classified as a *'complete response'*.

The obtained trajectories also helped us to identify the discriminative features that effectively distinguish failure cases from successful agent behaviors. For instance, a notable discriminative feature observed in the intermediate reasoning step, *'Action'* is the invocation of a tool that is not part of the agent's configured toolset. As an example, if an agent is provided access only to 'Search' and 'Calculator' tools but issues a call to 'Compare' tool, this constitutes invalid tool usage. Such behavior typically stems from hallucinations during tool selection. Another common discriminative feature is the agent's inability to follow the ReAct structure, leading to formatting errors such as *Invalid Format: Missing 'Action:' after 'Thought'* in the intermediate reasoning steps. This reflects a breakdown in the agent's ability to translate its reasoning into well-structured actions. These discriminative features are systematically extracted and leveraged by *AgentInspect* to detect behavioral failures in agent trajectories.

Specifically, after abstracting the trajectory, *AgentInspect* identifies each of the behavioral failures as follows: To identify *'Repetitive Reasoning'*, *AgentInspect* utilizes a semantic matching strategy based on cosine similarity. It compares the 'Action' and 'Action Input' pairs across consecutive steps in the trajectory. If the agent continues to reason using highly similar steps (cosine similarity $>=\delta$) for more than two consecutive steps or agent stops without providing a final answer, the behavior is flagged as repetitive. For *'ReAct Format Violations'* and *'Invalid Tool Invocation'*, *AgentInspect* leverages a set of discriminative features that highlight structural abnormalities in the agent's trajectory, such as missing action step or invocation of unconfigured tools. Additionally, when tool responses are labeled as 'Error' or 'No Response', the trajectory is flagged as exhibiting a *'Tool Execution Failure'*. Moreover, if all tool responses in a trajectory fall into 'Error' or 'No Response' categories, *AgentInspect* reports *Inference without Evidence* failure, indicating the agent produced an answer without meaningful tool-supported reasoning. Finally, *'Tool Crash'* failures are detected when the agent abruptly terminates its reasoning process following a tool invocation without reaching to the final answer.

The agent under test is executed on 30 test inputs obtained from the *'Test Input Generation'* component under baseline and simulated tool response settings. *AgentInspect* monitors the agent trajectories for behavioral failures across each run and provides a summary of the observed issues,

including failure types and their frequency for different settings. This report provides developers with insights to identify design flaws and optimize agent behavior before deployment.

## 5 Evaluation

In this section, we aim to answer the following research questions:

- **RQ1 (Effectiveness):** How effective is our approach in automatically diagnosing and characterizing behavioral failures in AI Agents?
- **RQ2 (Comparison):** How does our approach compare to state-of-the-art AI agent evaluation techniques?
- **RQ3 (Impact):** What is the impact of different test generation strategies and execution settings on *AgentInspect*'s ability to identify distinct failure types?

### 5.1 Experimental Setup

In this section, we describe the benchmark used for empirical evaluation and explain the labeling procedure used to establish ground-truth for evaluation.

*5.1.1 Benchmark.* To evaluate our approach, we collected AI agents developed using LangChain framework from GitHub. Specifically, we followed [20] and filtered repositories containing the keyword "LangChain Agents" in their titles. This process yielded 435 repositories as of July, 2025. Upon further manual analysis, we observed that in most repositories, AI agents were implemented as part of larger applications. To ensure correct setup and reproducibility, we exclude repositories in which the agent's logic is not provided in a separate, self-contained file. Additionally, we removed repositories in which agents interacted with tools requiring personal credentials (*e.g.,* Gmail, Slack, or similar APIs) to avoid privacy or configuration issues. As our work focuses on ReAct-style single-agent systems, we further filter out repositories containing agents following other paradigms, such as function-calling or structured-chat agents, since their reasoning chains differ fundamentally from ReAct. Similarly, we filter out multi-agent systems that involve inter-agent communication, which lies outside the scope of this work. After this multi-stage filtering process, we obtained 35 executable ReAct-style agents [29], which we used for empirical evaluation. To the best of our knowledge, this is the first benchmark of real-world AI agents developed by practitioners using their own configurations of language models, tools, and domain-specific constraints. This benchmark includes agents built using diverse language models (*e.g.,* gpt-4o, gemma-3-12b-it, llama-3.1-8b-instant), integrated tools (*e.g.,* calculator, search engines, python code executor), and spanning multiple domains such as banking, real estate, and restaurant lookup.

Each agent is executed on 30 test inputs generated following the procedure described in Section 4.2. This resulted in 1,050 trajectories in the baseline setting across 35 agents. In the simulated and hybrid setting, agents were re-executed only when their baseline runs completed successfully. Given that 103 crashes occurred during baseline execution, these two execution settings yielded a total of 4,735 trajectories. As the agent execution is resource-intensive, incurring costs from language model usage and external API calls, the total cost of generating these trajectories was approximately $550. This highlights the computational and financial overhead associated with evaluating large-scale agents. To support reproducibility and adoption, we release these trajectories as a benchmark. Other researchers can leverage them without re-executing the agents to develop and benchmark agent evaluation techniques.

### 5.2 Labeling

As AI agents have gained popularity recently, finding mature repositories with rich commit histories that highlight design changes over time remains a challenge. Consequently, for the 35 agents in our

benchmark, we lack ground truth labels to objectively evaluate the correctness of our approach. To overcome this challenge, we adopted a strategy commonly used in software engineering literature: manual labeling [16, 27], where domain experts inspect program behaviors or outputs to generate ground truth annotations. Following this practice, for labeling, we used the six behavioral failure categories discussed in Section 4.4 as a reference framework. Specifically, two authors with expertise in AI agents manually inspected and annotated each trajectory based on whether it exhibited one or more of these failure types. The labeling was done in six iterations, starting with 5% of the total agents, *i.e.,* 35 agents, and increasing by 5% increments per iteration. The raters first independently labeled the trajectories and used Cohen's Kappa coefficient ($\kappa$) [36] to measure inter-rater agreement after each iteration. The detailed agreement statistics from each iteration are available in [29]. We observed an increasing trend in agreement across iterations for trajectories of simulated setting with error in the tool response and incomplete tool response, and for trajectories of the hybrid setting, indicating that the raters were progressively converging on a shared understanding of the failure categories and labeling criteria. However, for trajectories of baseline setting and a simulated setting with no tool response, we observed a slight decreasing trend in agreement across iterations. This is because different reasoning patterns began to emerge in later iterations, introducing new edge cases that made consistent categorization more challenging for raters. Overall, we observed that $\kappa$ was always in the perfect agreement range ($\kappa \geq 0.81$) [10], indicating consistent high inter-rater agreement across all iterations. Finally, disagreements were subsequently resolved through discussion following the prior work [10] until a consensus was reached. This refinement ensured the reliability of the final annotations used to evaluate our approach *AgentInspect*.

### 5.3 Results

In this section, we report on the effectiveness of our technique and answer the research questions.

*5.3.1 RQ1 (Effectiveness):* To answer this research question, we evaluate our approach, *AgentInspect* on 35 agents in our benchmark. In Table 1, we present the behavioral failures detected by *AgentInspect* in each agent for three settings: baseline setting (with real tool responses), simulated setting (with simulated tool responses), and hybrid setting (with a mix of real and simulated tool responses). The reported numbers indicate the count of test inputs (out of 30) where specific behavioral failure is detected by *AgentInspect*. Please note that multiple behavioral failures may co-occur within a single test input. To evaluate the effectiveness of *AgentInspect*, we use manually annotated labels as ground truth and measure standard metrics, including true positives (TP), false positives (FP), and false negatives (FN). Due to space limitations, we present the results of 8 agents in Table 1. The results for the remaining 27 agents are available in [29].

In the baseline setting, as shown in Table 1, *AgentInspect* effectively identifies behavioral failures spanning all six categories across different agents. Among the six failure categories, most false positives and negatives occurred in the *'Tool Execution Failure'* category. This is primarily due to the heterogeneous nature of tool outputs, which can vary depending on the integrated tools and their response structure. Since *AgentInspect* uses discriminative features to automatically categorize tool outputs for detecting this failure, such variability in the tool outputs introduces ambiguity, thereby affecting the accuracy of automated classification. However, the ambiguity introduced by heterogeneous tool outputs has limited impact on overall performance. In contrast, in the simulated and hybrid settings, since agent behavior is evaluated under predefined tool execution failure scenarios, *AgentInspect* does not report *'Tool Execution Failure'*. As discussed in Section 4.3, the simulated and hybrid settings apply only when the baseline execution completes successfully; thus, tool crashes that prevent full trajectory generation are excluded from these settings. Consequently, *AgentInspect* does not report *'Tool Crash'* failures in these settings. In

Table 1. Behavioral Failures detected by *AgentInspect*.

| Agent # | Execution Setting | Tool Response Type | Behavioral Failure Categories | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | RR | | | TC | | | ITI | | | RFV | | | IWE | | | TEF | | |
| | | | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN | TP | FP | FN |
| Agent 7 | Baseline | Real | - | - | - | 3 | 0 | 0 | 5 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 1 | 3 | 0 |
| | Simulated | Error | **23** | 0 | 0 | x | x | x | 5 | 0 | 0 | 1 | 0 | 0 | **1** | 1 | 0 | x | x | x |
| | | Incomplete | **4** | 0 | 0 | x | x | x | 6 | 0 | 0 | - | - | - | **21** | 0 | 0 | x | x | x |
| | | No Response | **3** | 0 | 0 | x | x | x | 4 | 0 | 0 | - | - | - | **19** | 0 | 3 | x | x | x |
| | Hybrid | Delayed | - | - | - | x | x | x | 3 | 0 | 0 | - | - | - | **12** | 1 | 0 | x | x | x |
| | | Inconsistent | **3** | 0 | 0 | x | x | x | 6 | 0 | 0 | 1 | 0 | 0 | **1** | 2 | 0 | x | x | x |
| Agent 8 | Baseline | Real | 2 | 0 | 0 | - | - | - | 4 | 0 | 0 | 1 | 0 | 0 | - | - | - | 0 | 0 | 14 |
| | Simulated | Error | 25 | 0 | 1 | x | x | x | - | - | - | - | - | - | **3** | 0 | 0 | x | x | x |
| | | Incomplete | 8 | 0 | 0 | x | x | x | - | - | - | - | - | - | **19** | 0 | 3 | x | x | x |
| | | No Response | - | - | - | x | x | x | 3 | 0 | 0 | - | - | - | **28** | 0 | 2 | x | x | x |
| | Hybrid | Delayed | 4 | 0 | 0 | x | x | x | 2 | 0 | 0 | 2 | 0 | 0 | **2** | 0 | 0 | x | x | x |
| | | Inconsistent | 13 | 0 | 0 | x | x | x | - | - | - | - | - | - | - | - | - | x | x | x |
| Agent 12 | Baseline | Real | 1 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | 8 | 1 | 0 |
| | Simulated | Error | 14 | 0 | 0 | x | x | x | **7** | 0 | 0 | **5** | 0 | 0 | **7** | 0 | 0 | x | x | x |
| | | Incomplete | - | - | - | x | x | x | - | - | - | - | - | - | **29** | 0 | 1 | x | x | x |
| | | No Response | - | - | - | x | x | x | - | - | - | - | - | - | **29** | 1 | 0 | x | x | x |
| | Hybrid | Delayed | - | - | - | x | x | x | **1** | 0 | 0 | **1** | 0 | 0 | **27** | 0 | 3 | x | x | x |
| | | Inconsistent | 1 | 0 | 0 | x | x | x | - | - | - | - | - | - | - | - | - | x | x | x |
| Agent 13 | Baseline | Real | - | - | - | - | - | - | 5 | 0 | 0 | - | - | - | - | - | - | - | - | - |
| | Simulated | Error | **10** | 0 | 0 | x | x | x | 7 | 0 | 0 | **4** | 0 | 0 | **2** | 0 | 0 | x | x | x |
| | | Incomplete | - | - | - | x | x | x | 4 | 0 | 0 | - | - | - | **13** | 0 | 0 | x | x | x |
| | | No Response | - | - | - | x | x | x | - | - | - | - | - | - | **9** | 1 | 0 | x | x | x |
| | Hybrid | Delayed | **6** | 0 | 0 | x | x | x | 19 | 0 | 0 | 19 | 0 | 0 | **1** | 0 | 0 | x | x | x |
| | | Inconsistent | **1** | 0 | 0 | x | x | x | 6 | 0 | 0 | 2 | 0 | 0 | - | - | - | x | x | x |
| Agent 15 | Baseline | Real | 9 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | 1 | 2 | 0 |
| | Simulated | Error | 30 | 0 | 0 | x | x | x | **3** | 0 | 0 | **1** | 0 | 0 | - | - | - | x | x | x |
| | | Incomplete | 19 | 0 | 0 | x | x | x | - | - | - | - | - | - | **9** | 2 | 0 | x | x | x |
| | | No Response | 30 | 0 | 0 | x | x | x | - | - | - | - | - | - | - | - | - | x | x | x |
| | Hybrid | Delayed | - | - | - | x | x | x | 7 | 0 | 0 | 7 | 0 | 0 | 18 | 2 | 0 | x | x | x |
| | | Inconsistent | 6 | 0 | 0 | x | x | x | - | - | - | - | - | - | - | - | - | x | x | x |
| Agent 16 | Baseline | Real | - | - | - | 11 | 0 | 0 | - | - | - | - | - | - | 2 | 0 | 0 | 0 | 7 | 0 |
| | Simulated | Error | **6** | 0 | 0 | x | x | x | **5** | 0 | 0 | **5** | 0 | 0 | 5 | 1 | 0 | x | x | x |
| | | Incomplete | - | - | - | x | x | x | - | - | - | - | - | - | 19 | 0 | 0 | x | x | x |
| | | No Response | **8** | 0 | 0 | x | x | x | - | - | - | - | - | - | 11 | 0 | 0 | x | x | x |
| | Hybrid | Delayed | - | - | - | x | x | x | - | - | - | - | - | - | 16 | 0 | 2 | x | x | x |
| | | Inconsistent | - | - | - | x | x | x | - | - | - | - | - | - | 1 | 0 | 0 | x | x | x |
| Agent 24 | Baseline | Real | 4 | 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | 3 | 3 | 0 |
| | Simulated | Error | 22 | 0 | 0 | x | x | x | **2** | 0 | 0 | **2** | 0 | 0 | **6** | 0 | 1 | x | x | x |
| | | Incomplete | 11 | 0 | 0 | x | x | x | **5** | 0 | 0 | **5** | 0 | 0 | **17** | 0 | 2 | x | x | x |
| | | No Response | 1 | 0 | 0 | x | x | x | **4** | 0 | 0 | **4** | 0 | 0 | **29** | 0 | 0 | x | x | x |
| | Hybrid | Delayed | 2 | 0 | 0 | x | x | x | **2** | 0 | 0 | **2** | 0 | 0 | **24** | 0 | 0 | x | x | x |
| | | Inconsistent | 9 | 0 | 0 | x | x | x | **2** | 0 | 0 | **2** | 0 | 0 | - | - | - | x | x | x |
| Agent 25 | Baseline | Real | 4 | 0 | 0 | 20 | 0 | 0 | 4 | 0 | 0 | 4 | 0 | 0 | - | - | - | 0 | 4 | 0 |
| | Simulated | Error | 10 | 0 | 0 | x | x | x | 4 | 0 | 0 | 4 | 0 | 0 | - | - | - | x | x | x |
| | | Incomplete | 8 | 0 | 0 | x | x | x | 1 | 0 | 0 | - | - | - | **2** | 0 | 0 | x | x | x |
| | | No Response | 10 | 0 | 0 | x | x | x | 1 | 0 | 0 | 0 | 1 | 0 | - | - | - | x | x | x |
| | Hybrid | Delayed | 5 | 0 | 0 | x | x | x | 5 | 0 | 0 | 3 | 0 | 0 | - | - | - | x | x | x |
| | | Inconsistent | 1 | 0 | 0 | x | x | x | 1 | 0 | 0 | 1 | 0 | 0 | - | - | - | x | x | x |

**'-'**: indicates no bug detected by *AgentInspect* in this category; **'x'**: represents bug category is not applicable to this setting.
**RR**: Repetitive Reasoning, **TC**: Tool Crash, **ITI**: Invalid Tool Invocation, **RFV**: ReAct Format Violations, **IWE**: Inference without Evidence, **TEF**: Tool Execution Failure.

Table 1, the results from the simulated settings demonstrate the *AgentInspect*'s ability to uncover additional failure modes that are not observed during baseline execution. For instance, for Agent 12, the baseline execution revealed failures such as *'Repetitive Reasoning'* and *'Tool Execution Failure'*. However, when evaluated under simulated setting, the agent exhibited additional failures, including *'Inference without Evidence'*, *'Invalid Tool Invocation'*, and *'ReAct Format Violations'*. Similarly, for Agent 7, *'Repetitive Reasoning'* was not observed during baseline execution; however, this behavioral failure became prominent under the simulated setting, *'Error in Tool Response'*.

Overall, *AgentInspect* demonstrates promising performance, achieving a precision of 94.6%, a recall of 97.05%, and an F1-score of 95.82% across 35 agents in our benchmark. The results highlight the *AgentInspect*'s ability in detecting behavioral failures across varied runtime conditions. In particular, evaluating agents under different settings allows *AgentInspect* to uncover failure modes that do not emerge through testing in the baseline setting alone, thereby offering a more comprehensive assessment of agent behavior.

Table 2. Comparison of Failures detected by *AgentInspect* with *AgentEvals* [2] and *Galileo* [3].

| Failures Types | #Agents (All) | #Agents (Only by AE) | #Agents (Only by GA) | #Agents (Only AI) | #Agents (AE and GA) | #Agents (AE and AI) | #Agents (GA and AI) |
|---|---|---|---|---|---|---|---|
| Repetitive Reasoning | 22 | – | – | 5 | – | 5 | 1 |
| Invalid Tool Invocation | 7 | – | – | 6 | – | 1 | 5 |
| ReAct Format Violations | 6 | – | – | 5 | – | – | 9 |
| Inference without Evidence | – | – | – | 13 | – | – | – |
| Tool Execution Failure | 3 | 1 | 4 | 10 | 2 | 2 | 10 |
| Tool Crash | 15 | – | – | – | – | – | – |

**AE** denotes *AgentEvals*, **GA** stands for *Galileo*, and **AI** refers to *AgentInspect*.
'-': indicates no bug detected by the approach in this category;
Note: Value represents the number of agents (out of 35) where the corresponding failure is detected by each approach.

*Summary: AgentInspect demonstrates consistent performance in systematically detecting a wide range of behavioral failures across diverse runtime environments and agent configurations. By evaluating agents across different settings, AgentInspect reveals failures that remain unobserved under baseline setting alone, enabling a more comprehensive assessment of agent behavior.*

*5.3.2 RQ2 (Comparison):* To answer this research question, we compare our approach with two state-of-the-art techniques, AgentEvals [2] and Galileo [3]. Specifically, all three approaches are evaluated on the same benchmark of 35 AI agents. To account for the inherent randomness in agent's reasoning process, all approaches are evaluated on the same run and test inputs to ensure a fair comparison. The goal of the evaluation is not to compare precision metrics directly, but to highlight fundamental differences between deterministic and non-deterministic evaluation approaches. Below, we first describe the evaluation setup used for AgentEvals, Galileo, and *AgentInspect*, followed by a comparative analysis of the results produced by these approaches.

**Evaluation Setup:** AgentEvals supports two evaluation modes: (i) supervised, where agent trajectories are compared against expected reference trajectories, and (ii) unsupervised, where an LLM acts as a judge without access to ground-truth references. Since our approach operates in an unsupervised setting, we adopt the unsupervised mode for a fair comparison. We use the default evaluation configuration provided by AgentEvals, where the 'openai:o3-mini' model serves as the judge, and the predefined prompt: 'TRAJECTORY_ACCURACY_PROMPT' serves as the scoring guideline during our experiments. In this mode, AgentEvals assigns a binary score (true or false) to each trajectory, accompanied by a brief comment explaining the decision. A true score indicates that the trajectory is logically consistent, with coherent reasoning steps leading to successful task completion. In contrast, a false score signifies that the trajectory contains reasoning flaws or execution errors that prevent the agent from completing the task correctly. In our experiments, each agent is executed on a set of 30 test inputs. The resulting trajectories from these executions are provided as input to AgentEvals for evaluation. For the trajectories evaluated as false by AgentEvals, we analyzed the accompanying comments to report the specific reasoning flaws detected by the framework. A detailed report for each agent's trajectories is available in [29].

Galileo, on the other hand, works in an unsupervised setting, and uses multiple LLMs-as-a-judge (3 or 5) and their aggregated responses to determine the final evaluation outcome. For capturing trajectories during agent execution, Galileo offers several logging mechanisms, such as using the wrapper: GalileoLogger() or using callbacks: GalileoCallback() within the AgentExecutor(). In our experiments, we employed the GalileoCallback() to log trajectories, which were recorded and visualized through the Galileo console. When abnormal behavior is detected in the agent's trajectory, the platform provides insights in the Galileo console that explain the underlying reasoning error and its potential cause. Since these insights are currently viewable only within the Galileo console and cannot be downloaded directly, we have manually recorded and included the complete insight report in our GitHub repository. However, Galileo generates a report that provides a step-wise

evaluation of the agent's workflow, assigning a probability score between 0 and 1 to each step. A score of 1 indicates successful task completion, while scores between 0 and < 1 reflect the degree to which the agent partially accomplished the step. For transparency, we have included these results in our GitHub repository. We note that for the simulated and hybrid settings, Galileo consistently reported 'Tool calls not executing' as a failure due to the use of simulated tool responses across all agents. Since our approach generates failure reports for each setting independently, to ensure a fair comparison, we compare Galileo with *AgentInspect* in the baseline setting only.

*AgentInspect* is implemented as a lightweight wrapper around the agent to facilitate trajectory logging during execution and enable subsequent analysis. After the agent completes execution on a given test input, *AgentInspect* captures the resulting trajectory and processes it for evaluation using the method outlined in Section 4.4. It first operates in a baseline setting, where it records the real tool responses during the execution and constructs a set of simulated tool responses. The agent is automatically re-executed in the simulated and hybrid settings, and the corresponding trajectories are captured. The collected trajectories from the three settings are automatically analyzed to detect the behavioral failure, and a detailed failure report is generated.

In Table 2, we report how many agents (out of the 35) were flagged with the same failure by all three approaches in the baseline setting. We also present a breakdown of detection outcomes across different combinations of approaches (*e.g.,* failures detected by both AgentEvals and Galileo, by both Galileo and *AgentInspect*, *etc.*) to highlight the relative strengths and limitations of each method. The results for the simulated and hybrid settings are available in [29].

For the *'Repetitive Reasoning'* and *'Tool Crash'* categories, we observed the highest agreement among the three approaches, AgentEvals, Galileo, and *AgentInspect*, identifying these failures in 22 and 15 agents, respectively. Notably, in the *'Repetitive Reasoning'* category, *AgentInspect* detected this failure in 5 additional agents that are not flagged by AgentEvals or Galileo. This difference arises because *AgentInspect* is designed to detect repetitive search behaviors, even in cases where the agent eventually recovers after multiple attempts. In contrast, AgentEvals and Galileo do not flag it as a failure if the agent recovers and generates a final answer. Capturing this subtle form of reasoning inefficiency can help developers impose stricter constraints on an agent's reasoning process, potentially leading to more efficient and reliable behavior.

For the *'Invalid Tool Invocation'* and *'ReAct Format Violations'* categories, when an agent recovers from these failures and ultimately produces a final answer, AgentEvals and Galileo often do not flag the trajectory as faulty. However, this behavior is not consistent and appears to depend on the underlying language model used as the judge. For example, in Agent 2, Galileo did not identify the *'Invalid Tool Invocation'* failure, whereas AgentEvals did. In contrast, when similar behavior occurred in other agents, such as Agent 10 and Agent 19, Galileo did report the failure. This inconsistency highlights a fundamental limitation of LLM-based evaluation, due to the inherent variability in LLM-based judgment, both AgentEvals and Galileo may fail to report failures consistently across agents and test inputs, even when the underlying behavior is similar. However, a deterministic approach employed by *AgentInspect* enabled it to consistently detect *'Invalid Tool Invocation'* failure and *'ReAct Format Violations'* failure in 6 and 5 additional agents, respectively. Similarly, for *'Inference without Evidence'* category, while neither AgentEvals nor Galileo identified this failure, *AgentInspect* identified it in 13 agents.

For the *'Tool Execution Failure'* category, AgentEvals, Galileo, and *AgentInspect* identified failures in three agents. Unlike AgentEvals, which relies on a single language model as the judge, Galileo aggregates judgments from multiple models, enabling it to capture a broader range of failures. However, both approaches share a key limitation: their reliance on language models makes them prone to overlooking certain failure cases. For example, when the tool returns no response (as in Agent 35) or when the tool output exceeds the maximum character limit, leading to a hallucinated

response (as in Agent 11), these issues often go undetected. Likewise, when an agent produces a final answer without invoking any tools (as in Agent 35), neither AgentEvals nor Galileo flags it as a failure, despite the absence of tool-grounded reasoning. While the performance of *AgentInspect* is comparable to that of AgentEvals and Galileo, we found that *AgentInspect* exhibits more consistent performance across agents due to its deterministic analysis pipeline, which does not rely on language model-based judgment. This ensures that similar agent behaviors are identified uniformly, reducing variability and improving reliability in failure detection.

*Summary: The results demonstrate that the deterministic failure detection strategy employed by AgentInspect enables consistent and reliable detection of behavioral failures across agents and test inputs, outperforming LLM-as-a-judge methods.*

*5.3.3 RQ3 (Impact):* To analyze the impact of different test generation strategies on *AgentInspect*'s performance, we studied two key dimensions: the effect of test-suite size on failure categories coverage and the effect of prompt variants on the diversity of generated inputs and the failure categories they expose. Since executing the agents incurs a monetary cost, we selected 12 representative agents from our benchmark for this analysis, ensuring that the subset preserves variation along three key dimensions: tool setting (single-tool and multi-tool agents), underlying model (models from different families), and integrated tools. To study the effect of test-suite size on the coverage of different failure categories, we generated four test suites of varying sizes: 10, 20, 30, and 40 samples, and measured the failure categories covered by *AgentInspect* and compared it with AgentEvals and Galileo (results are available in [29]). Specifically, the test input generator (GPT-4) is prompted to produce test suites of varying sizes, with each suite generated in a single generation step. For AgentEvals and *AgentInspect*, the failure categories detected by each approach are aggregated across all execution settings for each agent. However, for Galileo, we only consider the failure categories detected in the baseline setting (returns 'Tool calls not executing' due to simulation for other settings) and compare its results with *AgentInspect*. We grouped the agents based on single-tool and multi-tool and computed, for each test suite, the average number of failure categories covered across agents within the same group [29]. Our analysis shows that, for single-tool agents, the average number of failure categories detected remains relatively stable across test suites of different sizes. In contrast, multi-tool agents exhibit more noticeable variation across different test suites, indicating that their behavior is more sensitive to input diversity. For both single-tool and multi-tool agents, our analysis suggests that, while not optimal, a test suite with 30 samples offers a reasonable balance between the testing cost and failure category coverage for both *AgentInspect* and baselines. Developers can use it as a default choice; however, since our approach is open-source, they can adjust the test suite size based on the testing budget and configuration of the agent under test.

To study the impact of different prompts on the diversity of the generated test input and failure categories they expose, we designed four prompt variants. Prompt 1 serves as a baseline and includes agent code and role description. Prompt 2 extends Prompt 1 with tool constraints. Prompt 3 extends Prompt 1 with ambiguity constraints. And, Prompt 4 combines both tool and ambiguity constraints with Prompt 1 (full prompt templates are available in [29]). For each of the 12 selected agents, we used these prompts to guide the test generation model to generate 30 test inputs and executed the agent on the corresponding test suite. We manually analyzed the resulting trajectories obtained from the baseline setting and computed the number of inputs that led the agent to exhibit one-step or multi-step reasoning and summarize the variety of tool use patterns observed in the resulting execution trajectories [29]. Our analysis shows that the test inputs generated using Prompt 2, Prompt 3, and Prompt 4 resulted in broadly comparable tool-use coverage during agent execution,

Table 3. Number of Faulty trajectories detected by *AgentInspect* for each agent.

| Agent # | Baseline | # of Faulty Trajectories | Baseline + Simulated (ER) | # of Faulty Trajectories | Baseline + Simulated (ICR) | # of Faulty Trajectories | Baseline + Simulated (NR) | # of Faulty Trajectories | Baseline + Hybrid (DR) | # of Faulty Trajectories | Baseline + Hybrid (INR) | # of Faulty Trajectories |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Agent 4 | – | 0 | RR,ITI,RFV | 0+11 | IWE | 0+21 | IWE | 0+27 | IWE | 0+23 | RR | 0+1 |
| Agent 9 | TC,ITI | 13 | TC,ITI,RR | 13+21 | TC,ITI,RR,IWE | 13+21 | TC,ITI,IWE | 13+18 | TC, ITI,IWE | 13+17 | TC,ITI,RR,RFV | 13+3 |
| Agent 12 | RR,TEF | 15 | RR,TEF,ITI,RFV,IWE | 15+26 | RR,TEF,IWE | 15+29 | RR,TEF,IWE | 15+30 | RR,TEF, ITI,RFV,IWE | 15+27 | RR,TEF | 15+6 |
| Agent 13 | ITI | 20 | ITI,RR,RFV,IWE | 20+28 | ITI,IWE | 20+16 | ITI,IWE | 20+24 | ITI, RR,RFV,IWE | 20+20 | ITI,RR,RFV | 20+27 |
| Agent 14 | TC | 7 | TC,RR | 7+26 | TC,IWE | 7+26 | TC,IWE | 7+26 | TC, ITI,RFV,IWE | 7+20 | TC | 7+5 |
| Agent 15 | RR,TEF | 20 | RR,TEF,ITI,RFV | 20+30 | RR,TEF,IWE | 20+30 | RR,TEF | 20+30 | RR,TEF, ITI,RFV,IWE | 20+20 | RR,TEF | 20+11 |
| Agent 19 | ITI,RFV | 11 | ITI,RFV,RR,IWE | 11+17 | ITI,RFV,IWE | 11+14 | ITI,RFV,IWE | 11+19 | ITI,RFV, RR,IWE | 11+12 | ITI,RFV | 11+19 |
| Agent 29 | TC | 13 | TC,RR,IWE | 13+21 | TC,RR,IWE | 13+21 | TC,ITI,IWE | 13+21 | TC, IWE | 13+14 | TC | 13+0 |

**RR**: Repetitive Reasoning, **TC**: Tool Crash, **ITI**: Invalid Tool Invocation, **RFV**: ReAct Format Violations, **IWE**: Inference without Evidence, **TEF**: Tool Execution Failure, **ER**: Error in Tool Response, **ICR**: Incomplete Tool Response, **NR**: No Tool Response, **DR**: Delayed Response, **INR**: Inconsistent Tool Response.

with each outperforming Prompt 1. However, Prompt 4 helps generate a test suite that engages the agent more in multi-step reasoning than simple one-step execution. From a testing perspective, this is important because such inputs exercise deeper execution behaviors and therefore increase the likelihood of exposing failures that may remain hidden under simpler tests. We further analyzed the impact of test inputs generated using different prompts on the failure categories exposed by the agents. For this, we executed each agent on the 30 test inputs generated using different prompts and compared the failure categories detected by *AgentInspect* across three execution settings (results are available in [29]). Our analysis suggests that, while test inputs generated using different prompts can reveal the same failure categories for some agents and additional failure categories for others, Prompt 4 helps generate test inputs that reveal more variation within each failure category, making it easier to identify and map the conditions under which an agent exhibits a particular failure behavior. This is especially useful for debugging, since the same failures may arise from different underlying root causes; developers can use this variation to better distinguish those root causes during repair. Overall, the results suggest that Prompt 4 can be used as a preferred choice for guiding test generation.

We also examine the impact of each execution setting on the types of behavioral failures detected. Specifically, we conduct a study by enabling baseline execution setting and incrementally combining it with different simulated and hybrid settings. This allows us to isolate and quantify the contribution of each setting to *AgentInspect*'s overall failure detection capability. We further quantified the agent's behavior by counting the number of faulty trajectories detected by *AgentInspect*. A higher number of faulty trajectories reflects limitations in the agent's reasoning, tool usage, or edge-case handling capabilities. In Table 3, we present the results for 8 representative agents; results for the remaining agents are available in [29]. As shown in Table 3, Agent 4 and Agent 14 demonstrate the fewest faulty trajectories in the baseline setting, indicating relatively robust performance under ideal conditions. However, when exposed to the simulated settings, the category of behavioral failures and the number of faulty trajectories substantially increase, highlighting the weakness of the agent in handling unexpected or abnormal tool responses. While for some agents, such as Agent 15, the number of faulty trajectories is high in the baseline setting and further increases when tested under simulated settings, indicating persistent and compounded weaknesses in the agent's reasoning or tool interaction under varied runtime conditions. Moreover, the number of faulty trajectories does not substantially increase for some agents in different settings, such as Agent 19; however, under simulated and hybrid settings, we observe a shift in the types of failures exhibited, indicating that while the overall performance remains consistent, the agent's failure modes vary depending on runtime conditions.

*Summary: The results demonstrate that test suite size, prompt variants, and evaluation settings are key factors in exercising diverse agent behaviors and exposing different failure modes, highlighting their complementary roles in assessing agent robustness.*

### 5.4 Discussion and Limitations

To emphasize the practical utility of our findings, we now illustrate how the results in Table 1 can directly inform and guide agent refinement during development. The primary goal of this study is to uncover design flaws in AI agents by analyzing their behavior on different tool responses across varied inputs. This goal is illustrated in Table 1 through concrete agent behaviors observed in our evaluation. For instance, consider Agent 7 in Table 1. In the baseline setting, the agent resulted in different behavioral failures including *'Tool Crash'*, *'Invalid Tool Invocation'*, and *'ReAct Format Violations'*. These failures reveal underlying design flaws in the agent. For instance, a *'Tool Crash'*, where the agent terminates unexpectedly during execution, indicates the absence of a robust error-handling mechanism. Although multiple behavioral failures are identified in the baseline setting, the failure type, *'Repetitive Reasoning'*, in which the agent enters a reasoning loop and fails to terminate, was not observed under normal execution conditions. However, in the simulated setting, when the tool returns an error, this failure mode becomes prominent. This highlights the need for more robust termination criteria in agent design to prevent infinite or redundant reasoning loops triggered by unexpected tool responses.

Similarly, for Agents: 12, 15, 16, and 24, the behavioral failure, *'Invalid Tool Invocation'*, was not observed in baseline setting. However, in the simulated setting, this behavior becomes evident as the agent attempts to invoke non-configured tools. This indicates that the tools are either incorrectly registered or insufficiently integrated with the agent, leading it to invoke non-existent or incompatible tools. Such failures expose a fundamental flaw in tool-agent integration, highlighting the need for robust integration mechanisms to prevent invocation errors during execution and to ensure reliable, consistent agent performance. Likewise, in Agents: 8, 12, 13, and 24, the behavioral failure, *'Inference without Evidence'* was not observed in the baseline setting but emerged in the simulated settings. This suggests that, in the absence of abnormal tool responses, the agent tends to generate hallucinated outputs. This behavior underscores the importance of incorporating fallback mechanisms within the agent's design to prevent the generation of unsupported or fabricated answers. These examples illustrate the practical utility of *AgentInspect* in diagnosing agent design flaws. Consequently, the identified behavioral failures provide insight for improving agent robustness, enabling developers to strengthen the agent's reliability during development.

Currently, *AgentInspect* supports the failure modes that are directly observable in the agent's execution trajectory and does not require a deeper semantic correctness assessment. To assess the feasibility of extending our framework for other semantic-level failure modes that require deeper semantic judgment, we implemented a semantic checker to detect one of the common failure modes: "Goal Deviation" [14]. It adopts a natural language processing-based semantic analysis approach grounded in Natural Language Inference (NLI) and entity matching and checks whether the actions taken by the agent during execution remain aligned with the intended goal. We evaluated this approach on two agents (Agents 7 and 28) from our benchmark and verified the results through manual inspection (results are available in [29]). The preliminary results suggest that the framework can be extended to support semantic-level failures in future work by incorporating semantic analysis in addition to the structural analysis used in the current approach.

One of the limitations of *AgentInspect* is that it currently supports single-agent systems developed using the LangChain framework, where the agent operates independently without requiring communication or coordination with other agents to complete a task. Despite this limitation, our approach remains impactful due to the popularity of ReAct agents [7, 39], underscoring the immediate applicability and impact of our findings. In the future, we plan to extend it for multi-agent systems and other agent development frameworks. Also, our current test input generation approach generates a test suite in a single generation step using prompt-specified constraints,

which limits the language model's ability to produce more diverse inputs that help expose new failures as the test suite size grows. In the future, we plan to investigate more adaptive test input generation techniques. Another limitation is that our current simulation approach is designed for tools that return textual or structured outputs, rather than tools that perform external actions. In the future, we plan to extend it to support action-oriented tools. Also, *AgentInspect* currently simulates three common tool failures scenarios. Future work will extend the simulation to more subtle tool failure cases, such as misleading but structurally complete responses or stale responses.

## 6 Threats to Validity

We acknowledge several potential threats to validity: (1) The internal threat stems from the lack of an established ground truth for evaluating the behavioral failures in agent trajectories. Therefore, we relied on manual labeling to evaluate the correctness of detected failures. To mitigate subjectivity and potential bias, two authors independently reviewed and labeled agent trajectories, with any disagreements resolved through discussion to reach consensus. (2) Another potential threat concerns the scope of behavioral failure categories supported by *AgentInspect*. While these categories may not capture all possible failure modes, we mitigate this threat by grounding the selected failure categories in prior work on agent reasoning and evaluation [3, 13, 14, 39]. (3) We curated a benchmark of 35 AI agents obtained from GitHub for evaluation; one potential threat to external validity is the representativeness of this benchmark. To mitigate this, we manually verified variability across agent configurations, integrated tools, and domains to ensure broad coverage. Also, the benchmark is publicly available to facilitate replication, extension, and independent validation by the research community. (4) Due to the non-deterministic nature of agent executions, another potential threat arises when comparing *AgentInspect* with existing approaches [2, 3]. To ensure a fair and consistent comparison, we ran all approaches on the same agent execution traces and test inputs, thereby eliminating variability introduced by multiple runs. (5) For test input generation, *AgentInspect* leverages a language model (*i.e.,* GPT-4). A potential threat arises from the possibility that some generated inputs may be misaligned with the intended task objectives. To mitigate this, two authors manually reviewed and validated the generated test inputs to ensure their correctness and relevance. (6) A potential threat arises from the choice of similarity threshold used for comparing tool inputs during simulated execution and identifying *'Repetitive Reasoning'* failure. The selected threshold value may influence which inputs are considered semantically equivalent, thereby affecting the consistency of failure detection. To mitigate this, we empirically selected a reasonable threshold based on preliminary experiments but acknowledge that different thresholds may impact failure detection outcomes. For transparency, the source code with the selected thresholds is publicly available. Future work could explore adaptive or learned thresholding strategies.

## 7 Conclusion

In this work, we present *AgentInspect*, a systematic framework for testing the behavioral robustness of AI agents by analyzing their reasoning traces and interactions with external tools. By leveraging simulated tool responses and discriminative trajectory features, our approach facilitates deterministic controlled testing across a range of runtime conditions, exposing subtle yet critical behavioral flaws that otherwise remain unexposed during baseline execution. We also release the first curated dataset of real-world agents and their execution traces, enabling reproducible evaluation and supporting the development of future agent testing tools. Empirical results demonstrate that *AgentInspect* consistently outperforms prior methods in detecting different behavioral failures, providing developers with insights to improve agent design.

## 8 Data Availability

The source code and evaluation results are available in this repository [29] to support reproducibility and facilitate future research.

## 9 Acknowledgments

The authors thank the anonymous ISSTA 2026 reviewers for their valuable feedback. This work is supported by the US National Science Foundation (NSF) under grants CCF-2223812, CNS-2120448, IIS-2419882, and IIS-2419883, the Fonds de Recherche du Quebec (FRQ), the Canadian Institute for Advanced Research (CIFAR), the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Canada Research Chairs Program. All opinions are those of the authors and do not reflect the views of the sponsors. AI-assisted tools were used to support software development and improve the clarity of the manuscript.

## References

[1] 2023. langchain-agents . https://github.com/DonGuillotine/langchain-agents/blob/main/reAct_math.py.
[2] 2025. *AgentEvals*. https://github.com/langchain-ai/agentevals
[3] 2025. *Galileo: AI Evaluation and Observability Platform*. https://galileo.ai/
[4] 2025. Google Search API . https://serpapi.com/pricing.
[5] 2025. LangChain: Build an Agent. https://docs.langchain.com/oss/python/langchain/agents.
[6] 2025. OpenAI Pricing . https://openai.com/api/pricing/.
[7] Jesse Anglen. 2025. *What is ReAct in AI Agents? Understanding Planning + Action Framework*. https://www.ruh.ai/blogs/react-ai-agents-framework Last updated Dec 24, 2025.
[8] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. 2018. An exploratory study on faults in web API integration in a large-scale payment company. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 13–22.
[9] David Bermbach and Erik Wittern. 2016. Benchmarking web api quality. In *International Conference on Web Engineering*. Springer, 188–206.
[10] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *Proceedings of the 44th International Conference on Software Engineering*. 2091–2103.
[11] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 2846–2857.
[12] Tony Busker, Sunil Choenni, and Mortaza S Bargh. 2025. Exploiting GPT for synthetic data generation: An empirical study. *Government Information Quarterly* 42, 1 (2025), 101988.
[13] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. 2025. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657* (2025).
[14] Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. 2025. TRAIL: Trace Reasoning and Agentic Issue Localization. *arXiv preprint arXiv:2505.08638* (2025).
[15] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. PMLR, 1–16.
[16] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*. Springer, 285–311.
[17] Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fourney, Jack Gerrits, Erkang Zhu, and Saleema Amershi. 2025. Interactive debugging and steering of multi-agent ai systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–15.
[18] Gordon Fraser and Andrea Arcuri. 2013. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE sixth international conference on software testing, verification and validation*. IEEE, 362–369.
[19] Gaole He, Gianluca Demartini, and Ujwal Gadiraju. 2025. Plan-then-execute: An empirical study of user trust and team performance when using llm agents as a daily assistant. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–22.
[20] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and*

*symposium on the foundations of software engineering*. 510–520.

[21] 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).

[22] Yixing Jiang, Kameron C Black, Gloria Geng, Danny Park, James Zou, Andrew Y Ng, and Jonathan H Chen. 2025. MedAgentBench: a virtual EHR environment to benchmark medical LLM agents. *NEJM AI* 2, 9 (2025), AIdbp2500144.

[23] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues?. In *International Conference on Learning Representations*, Vol. 2024. 54107–54157.

[24] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485.

[25] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2024. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, Vol. 2024. 52989–53046.

[26] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. 2012. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[27] Ruchira Manke, Mohammad Wardat, Foutse Khomh, and Hridesh Rajan. 2025. Leveraging data characteristics for bug localization in deep learning programs. *ACM Transactions on Software Engineering and Methodology* 34, 6 (2025), 1–29.

[28] Ruchira Manke, Mohammad Wardat, Foutse Khomh, and Hridesh Rajan. 2025. Mock Deep Testing: Toward Separate Development of Data and Models for Deep Learning. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 756–756.

[29] Ruchira Manke, Mohammad Wardat, Foutse Khomh, and Hridesh Rajan. 2026. *Replication Package for Research Paper Titled "AgentInspect: Diagnosing Behavioral Failures in Artificial Intelligence Agents*. doi:10.5281/zenodo.21482543

[30] Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations*, Vol. 2024. 9025–9049.

[31] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2004. *Principles of program analysis*. Springer Science & Business Media.

[32] Alessandro Orso and Bryan Kennedy. 2005. Selective capture and replay of program executions. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–7.

[33] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*. 15174–15186.

[34] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris Maddison, and Tatsunori Hashimoto. 2024. Identifying the risks of lm agents with an lm-emulated sandbox. In *International Conference on Learning Representations*, Vol. 2024. 27031–27098.

[35] David Saff, Shay Artzi, Jeff H Perkins, and Michael D Ernst. 2005. Automatic test factoring for Java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 114–123.

[36] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.

[37] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.

[38] Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. 2023. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323* (2023).

[39] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *11th International Conference on Learning Representations, ICLR 2023*.

[40] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.

[41] Zhexin Zhang, Shiyao Cui, Yida Lu, Jingzhuo Zhou, Junxiao Yang, Hongning Wang, and Minlie Huang. 2024. Agent-safetybench: Evaluating the safety of llm agents. *arXiv preprint arXiv:2412.14470* (2024).

[42] Mingchen Zhuge, Changsheng Zhao, Dylan R Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, et al. 2025. Agent-as-a-Judge: Evaluate Agents with Agents. In *International Conference on Machine Learning*. PMLR, 80569–80611.

Received 2026-01-30; accepted 2026-06-25