AgentChaos: Chaos Engineering for Agent Systems via Programmatic Fault Injection

Gou Tan
Sun Yat-sen University
Guangzhou, China
tang29@mail2.sysu.edu.cn

Zhensu Sun
Singapore Management University
Singapore
zssun@smu.edu.sg

Jieke Shi
Singapore Management University
Singapore
jiekeshi@smu.edu.sg

Ting Zhang
Monash University
Australia
ting.zhang@monash.edu

Zilong He
Sun Yat-sen University
Guangzhou, China
hezlong@mail2.sysu.edu.cn

Qingfu Wu
Sun Yat-sen University
Guangzhou, China
wuqf23@mail2.sysu.edu.cn

Shuai Liang
Sun Yat-sen University
Guangzhou, China
China Unicom
Beijing, China
liangsh76@mail2.sysu.edu.cn

Weifeng Sun
Singapore Management University
Singapore
wfsun@smu.edu.sg

Junda He
Singapore Management University
Singapore
jundahe.2022@phdcs.smu.edu.sg

Pengfei Chen†
Sun Yat-sen University
Guangzhou, China
chenpf7@mail.sysu.edu.cn

Chuanfu Zhang
Sun Yat-sen University
Guangzhou, China
zhangchf9@mail.sysu.edu.cn

Lwin Khin Shar
Singapore Management University
Singapore
lkshar@smu.edu.sg

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Abstract

Agent systems rely on LLM APIs for every response, but these APIs can return server errors, truncated responses, or corrupted content that propagates through downstream agents and causes task failure. Evaluating robustness under these faults is crucial for reliable deployment. Existing fault injection methods are offline, require source code modification, or cannot modify specific response fields. A comprehensive evaluation also requires a systematic fault taxonomy because different fault types affect downstream agents differently. We propose AgentChaos, a chaos engineering framework for controlled, runtime, non-intrusive LLM API fault injection. Since all agent systems access LLMs through the same HTTP interface, we inject faults at this shared layer without modifying source code. We define crash, omission, and value faults on content and tool call fields, intercept and modify LLM API responses at runtime, and verify whether each fault is triggered to filter untriggered tasks and avoid underestimating fault impact. Evaluations across agent systems, benchmarks, and backbone LLMs under 65 fault configurations show that all systems degrade under fault injection, with pass@1 dropping by up to 50 percentage points. The ranking is consistent across models, suggesting that robustness depends on system implementation rather than model capability. Existing fault diagnosis methods achieve below 53% accuracy on fault type and below 56% on fault step, leaving room for improvement. We further reveal practical findings for agent system developers.

CCS Concepts

• **Software and its engineering** → **Software reliability**; **Software performance**.

Keywords

Agent Systems, LLMs, Fault Injection, Chaos Engineering

ACM Reference Format:
Gou Tan, Zhensu Sun, Jieke Shi, Ting Zhang, Zilong He, Qingfu Wu, Shuai Liang, Weifeng Sun, Junda He, Pengfei Chen, Chuanfu Zhang, Lwin Khin Shar, and David Lo. 2026. AgentChaos: Chaos Engineering for Agent Systems via Programmatic Fault Injection. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837437

1 Introduction

Agent systems built on Large Language Models (LLMs) have been widely adopted for complex tasks, such as question answering [24],

†Pengfei Chen is the corresponding author.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837437

reasoning [42, 43], and software engineering [19, 22, 38, 46, 47]. The majority of these systems, such as Claude Code, rely on LLM APIs to generate every response. To fulfill a single task, an agent system typically issues multiple LLM API calls, making the reliability of the underlying APIs a critical factor for the entire system.

However, LLM APIs in practice can return various errors, such as server errors (e.g., HTTP 5xx), truncated responses (e.g., token limit cutoff), or corrupted content (e.g., garbled characters). The more LLM API calls a task requires, the higher the chance of encountering such a fault. As shown in Fig. 1, once a faulty response occurs, it can propagate through downstream agents and cause task failure. Quantifying the impact of LLM API faults on agent systems is therefore crucial for reliable deployment. Chaos engineering provides a principled approach to this problem by injecting controlled faults into running systems and observing their behavior, discovering weaknesses before real failures occur in production [51].

Existing fault injection methods cannot support controlled, runtime, non-intrusive LLM API fault injection. *Agent-oriented fault injection methods* either operate offline or at runtime target non-API faults. Offline methods (e.g., AgenTracer [53]) use an LLM to perturb completed execution traces after tasks finish, producing fault labels for downstream diagnosis rather than injecting faults into live systems. Because faults are never injected into running systems, they cannot trigger real runtime behaviors nor reveal actual software weaknesses. Runtime methods (e.g., MAS-FIRE [23]) target semantic failures such as hallucination and role ambiguity by hijacking system prompts, rewriting agent outputs, and rerouting messages. The injection logic needs to be re-implemented for each target system, and systems that do not expose the required interfaces cannot be injected. Since these methods do not target LLM API faults, their findings on fault impact do not generalize to LLM API faults such as response truncation. In contrast, *traditional API fault injection methods* (e.g., Rainmaker [11], ChaosBlade [6]) operate at the HTTP or OS layer without code modification, but treat responses as opaque and cannot modify specific fields within the response body, such as `message.content` or `tool_calls`. They can simulate server errors and timeouts, but cannot inject faults that require parsing and modifying the response content, such as response truncation or encoding corruption.

Beyond the injection method itself, a comprehensive robustness evaluation also requires a systematic fault taxonomy. LLM API faults take many forms, such as server errors, truncated responses, and corrupted encoding, and each form affects downstream agents differently. Without a systematic taxonomy, evaluators may test only a few obvious types and miss the most damaging ones. For example, a server error is easy to detect and retry, but a truncated response still looks like valid output, so the agent accepts it and passes the incomplete result to the next step.

To address these problems, we propose AgentChaos, a chaos engineering framework that evaluates agent system robustness through controlled, runtime, non-intrusive LLM API fault injection. Our key insight is that all agent systems access LLMs through the same HTTP request-response interface, so we can inject faults at this shared transport layer without modifying any source code. We first define a fault taxonomy by adapting the classical fault classification from distributed systems [4] to LLM API responses (§3), covering crash, omission, and value faults on both content and tool call fields. Combined with injection strategies and compound scenarios, this taxonomy yields 65 fault configurations. We then design two components for the injection framework. (1) An *HTTP-layer injection mechanism* (§4.3) that patches the HTTP client at runtime to intercept and modify LLM API responses according to the fault configuration, requiring no changes to any agent system. (2) A *trigger verification procedure* (§4.4) that records execution traces and checks whether each fault was actually triggered, filtering untriggered tasks from evaluation. During evaluation, each task receives one fault configuration at runtime, and we compare pass@1 before and after injection to quantify robustness degradation.

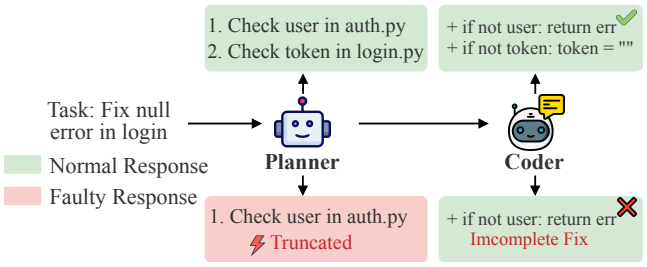

Figure 1: A truncated LLM API response at the Planner produces an incomplete plan, leading to an incomplete fix.

We evaluate across multiple agent systems [19, 22, 24, 46, 47], benchmarks [3, 9, 13, 26, 42, 43], and backbone LLMs [1, 12, 30, 32] under all 65 fault configurations. All systems show performance degradation, with pass@1 dropping up to 50 percentage points (MapCoder on HumanEval+), as shown in Table 3. We also evaluate fault diagnosis on failed tasks, where the goal is to identify which type of fault was injected and at which LLM call the fault occurred. Both rule-based pattern matching and LLM-based diagnosis [55] achieve below 53% accuracy on fault type and below 56% on fault step (Table 9). Beyond these results, our evaluation reveals several new findings (discussed in §6). (1) *The most severe faults are not the most harmful.* (2) *The most harmful faults are also the hardest to diagnose.* (3) *Robustness depends on system implementation.*

In summary, we make the following contributions.

- We define a fault taxonomy for LLM API responses by adapting the classical fault classification from distributed systems [4]. The taxonomy covers crash, omission, and value faults on both content and tool call fields. Combined with injection strategies and compound scenarios, it yields 65 fault configurations (§3, §4.2).
- We design a controlled, runtime, non-intrusive fault injection framework for agent systems. The framework installs a wrapper at the HTTP layer to intercept and modify LLM API responses at runtime without modifying any agent system source code (§4.3), and verifies whether each fault was actually triggered to filter untriggered tasks from evaluation (§4.4).
- We evaluate across multiple agent systems, benchmarks, backbone LLMs, and fault configurations. All systems degrade under fault injection (pass@1 drops up to 50 percentage points), and existing fault diagnosis methods achieve below 56% accuracy.

2 Background

2.1 Agent Systems and LLM API Dependence

An agent system uses one or more LLM-powered agents to solve complex tasks [8, 44–46, 50]. Each agent has a role (e.g., coder,

```
// (a) Text generation
{"choices": [{"message": {
    "content": "def add(a, b):\n return a + b" ,
    "tool_calls": null},
  "finish_reason": "stop"}]}

// (b) Tool invocation
{"choices": [{"message": {
    "content": null,
    "tool_calls": [{"function": {"name": "execute",
     "arguments": "{\"code\":\"print(add(1,2))\"}"}}] },
  "finish_reason": "tool_calls"}]}
```

Figure 2: OpenAI Chat Completions API response. `content` and `tool_calls` are mutually exclusive. Inactive fields are gray. `finish_reason`: finish metadata.

Table 1: Fault taxonomy for LLM API responses. ✓ marks a valid target field.

Category	Fault type	Content	Tool call	Real-world scenario
Crash	Error	✓	✓	server overload, HTTP 5xx, rate limiting [54, 58]
	Timeout	✓	✓	network congestion, backend delay, API latency [34, 54]
Omission	Empty	✓	✓	safety filter, content policy rejection [28]
	Truncate	✓	✓	token limit, TCP interruption, incomplete completion [54, 58]
Value	Corrupt	✓	✓	encoding error, garbled characters [52]
	Schema	✓	✓	parsing error, schema mismatch [35, 37, 58]

reviewer, planner) and calls the LLM API to generate text, invoke tools (e.g., code executors, file editors, web search), or communicate with other agents through message passing. Each activated agent makes at least one LLM API call, so a task involves a chain of LLM API calls and tool invocations across agents and rounds.

Agent systems access LLMs through HTTP-based APIs. Providers expose endpoints such as OpenAI `/v1/chat/completions` and Anthropic `/v1/messages`, but all follow the same HTTP request-response pattern. An agent sends conversation history and receives the LLM output. This shared communication layer enables our non-intrusive fault injection. Although providers use JSON schemas, their responses contain generated text, tool call arguments, and completion metadata. Most agent frameworks use OpenAI-compatible APIs, so we use the OpenAI Chat Completions format throughout this paper. As shown in Figure 2, its `message` object contains two mutually exclusive fields. The `content` field stores generated text, while `tool_calls` stores structured tool arguments. The `finish_reason` field indicates normal completion (`stop`), token-limit truncation (`length`), or tool invocation (`tool_calls`).

2.2 Fault Injection and Chaos Engineering

Fault injection is a technique that introduces controlled faults into a system to evaluate its robustness. Chaos engineering systematizes this idea into a discipline that proactively injects faults into running systems to discover weaknesses before they cause failures in production [7, 20, 27, 51]. The core principle is that a system's fault tolerance cannot be verified under normal conditions alone. Only by injecting realistic faults into a running system can developers observe how the system responds, such as retries, path switches, or silent error propagation, and identify weaknesses that would otherwise remain hidden. In traditional software systems, tools such as ChaosBlade [6] inject infrastructure-level faults (e.g., instance termination, CPU exhaustion, network delay) to test robustness.

3 Fault Taxonomy for LLM API Responses

To address the lack of a systematic fault taxonomy for LLM API responses, we make a deep investigation on fault types at the LLM API response layer. Using a deductive classification based on dependability theory [4], several authors take the three fault categories (i.e., crash, omission, value), apply each to the response fields in §2.1, and then discuss and split each into finer types: a crash fault yields Error and Timeout, an omission fault yields Empty and Truncate, a value fault yields Corrupt and Schema. Since an LLM API response has a fixed set of fields, this enumeration covers all fault types rather than selecting them by hand. Existing fault taxonomies for agent systems and ours both describe the faults that can affect agent systems, but differ in sources and layers. They group observed faults through developer surveys [29], incident analysis [49], or literature review [23] at the prompt or application layer, while ours is fixed and targets the response layer. Thus, these two taxonomies are complementary. An LLM API response has a fixed structure with a known set of fields (§2.1). Therefore, this investigation produces a complete fault list that covers every way a response can deviate for each category without relying on human experience, and the same process applies when new API fields appear in the future. Faults in our taxonomy are not hypothetical. Empirical studies on LLM-based software in production report that API errors, connection timeouts, rate limiting, and response format issues are among the most common failures in agent frameworks [58], AI coding tools [54], and LLM-integrated applications [33, 34], and our taxonomy systematically covers all of them.

Target fields. The first dimension of our investigation is the target field. As described in §2.1, `content` holds generated text and `tool_calls` holds structured tool arguments. Faults on `content` corrupt the text that the next agent reads. Faults on `tool_calls` corrupt the arguments passed to tools. Because `content` is a plain text string while `tool_calls` contains structured JSON, the same fault type affects each field differently. We apply each fault type to both fields to produce the configurations in Table 1. This distinction matters in practice because tool invocation and parameter generation are known sources of failures in agent systems that rely on LLM APIs [35, 37, 58].

Fault categories. The second dimension is the fault category. In distributed systems [4], faults are classified by the severity of service deviation. A *crash fault* returns no valid response, an *omission fault* returns an incomplete response, and a *value fault* returns a response with incorrect content. For traditional software services, faults on structured outputs are easy to detect by checking error codes or field values. LLM API responses are different because they contain free-form text such as reasoning, code, or tool call arguments, and the expected output is not predefined. A fault on such content, such as truncated code or garbled characters, cannot be detected by checking any single field. This makes LLM API faults harder to detect and more likely to propagate silently. A faulty LLM response still looks normal, so the receiving agent uses it as normal input and passes it to the next agent. This differs from a traditional distributed system, where a fault surfaces through an error code and is caught where it occurs. We adapt each fault category to the LLM API response structure as described below.

Fault types. We define six fault types across the three categories. Table 1 maps each type to its real deployment scenario: Error to server overload and HTTP 5xx, Timeout to network delay, Empty

to a safety filter, Truncate to a token-limit cutoff, Corrupt to an encoding error, and Schema to a tool-call format mismatch.

Crash faults make the response unusable for the receiving agent.

- **Error.** The LLM API returns an error message instead of the expected output, as happens during server overload, deployment rollouts, or rate limiting [54, 58].
- **Timeout.** The LLM API does not respond within the allowed time, as happens during network congestion or backend processing delays [34, 54].

Omission faults return a valid response structure with missing content.

- **Empty.** The LLM API returns a response with no content, as happens when a safety filter or content policy blocks the output [28]. The response parses but contains no useful information.
- **Truncate.** The LLM API returns a response that is cut off partway through, as happens when the token limit is reached or a TCP connection drops during streaming [54, 58]. The agent receives a partial response that may look like a short but complete answer, making truncation harder to detect than an empty response.

Value faults return a response that is structurally valid and complete, but with wrong content. These are the hardest faults to detect because the response looks normal from the outside. Semantic faults such as hallucination and semantic drift also fall in this category, since the model returns a complete response whose content is wrong. They originate in the model rather than in the API layer, so our framework classifies them here but does not inject them.

- **Corrupt.** The response content is damaged by encoding errors, as happens when proxy servers or caching layers apply incorrect character encoding. The output parses but is semantically meaningless.
- **Schema.** The response contains valid JSON that does not follow the expected structure, as happens when the LLM generates malformed output or a parsing error corrupts the response [35, 37, 58]. No parsing error is raised, and the fault propagates silently.

This taxonomy defines what faults to inject when we evaluate how robust agent systems are under LLM API faults. Combined with injection strategies and compound scenarios (§4.2), it expands into the fault configurations used in the experiments, letting us measure the degradation each fault causes and identify which fault type and configuration degrade each architecture most.

4 Methodology

4.1 Overview

We present AgentChaos, a chaos engineering framework that evaluates the robustness of agent systems through controlled, runtime, non-intrusive fault injection at the LLM API layer. All agent systems communicate with LLM services through HTTP requests, regardless of their internal architecture (§2.1). We exploit this shared layer by installing a fault injection wrapper on the HTTP client at runtime. The wrapper intercepts every LLM API response, decides whether to inject a fault based on the configured policy, modifies the response if the policy fires, and returns the result to the agent system. No source code of the agent system is changed.

As illustrated in Fig. 3, the framework takes an agent system, tasks, and a fault configuration as input. It outputs task results with execution traces that record every LLM API call. The framework has the following components: (1) *fault configuration* (§4.2) combines fault types, target fields, injection strategies, and compound scenarios into complete configurations, (2) *fault injection* (§4.3) implements the HTTP layer wrapper that intercepts and modifies LLM API responses at runtime, and (3) *trigger verification* (§4.4) checks execution traces after task completion and filters tasks where the configured fault was not triggered.

4.2 Fault Configuration

The fault taxonomy (§3) defines the fault types and target fields. To form complete configurations, we further specify injection frequency, compound combinations, and injection position.

Injection strategies. Different real failures affect different numbers of LLM calls within a single task. A network glitch may corrupt only one call, while an expired API key causes every call to fail. We define the following strategies to cover these patterns.

- *Single*. Inject the fault once at the first matching LLM call, then stop. This models a transient failure such as a network glitch.
- *Persistent*. Inject the fault at every matching LLM call throughout the entire task. This models a sustained failure such as an expired API key or a misconfigured proxy.
- *Intermittent*. Inject the fault at each matching LLM call independently with probability 0.3. This models a flaky connection where roughly 30% of requests fail.
- *Burst*. Inject the fault at the first 3 consecutive matching LLM calls, then stop. This models a temporary overload where the server rejects requests briefly and then recovers.

Compound scenarios. Real deployment failures can change the response in one or more ways. For example, server overload may first add a delay and then return an error response. A content safety filter may strip tool calls and replace the content with a rejection message simultaneously. To model such cases, we define compound scenarios that reproduce a real deployment failure by modifying the same LLM API response within a single interception. A scenario is grouped by the failure it reproduces, not by the number of fault types, so it stays compound even when it maps to a single type. We take these failures from empirical studies on LLM-based software [33, 34, 54, 58] and vendor documentation [28], selecting those we can reproduce at the LLM API response layer. The fault types column maps each failure to the base types from our taxonomy (§3) that its response changes most resemble. Table 2 lists all compound scenarios. Each scenario applies the changes its failure produces, which may affect one or several fields of the response.

Configuration space. A complete fault configuration consists of four elements: (1) the fault type (from §3), (2) the target field (`content` or `tool_calls`), (3) the injection strategy (single, persistent, intermittent, or burst), and (4) the injection position (which LLM call to start injecting at). We construct the full set of configurations in three groups. *Basic experiments* combine each of the 6 fault types with each of the 2 target fields and each of the 4 injection strategies, yielding $6 \times 2 \times 4 = 48$ configurations. *Position experiments* inject selected fault types (error, timeout, schema) at different call positions (1st, 2nd, 3rd call), yielding $3 \times 3 = 9$ configurations.

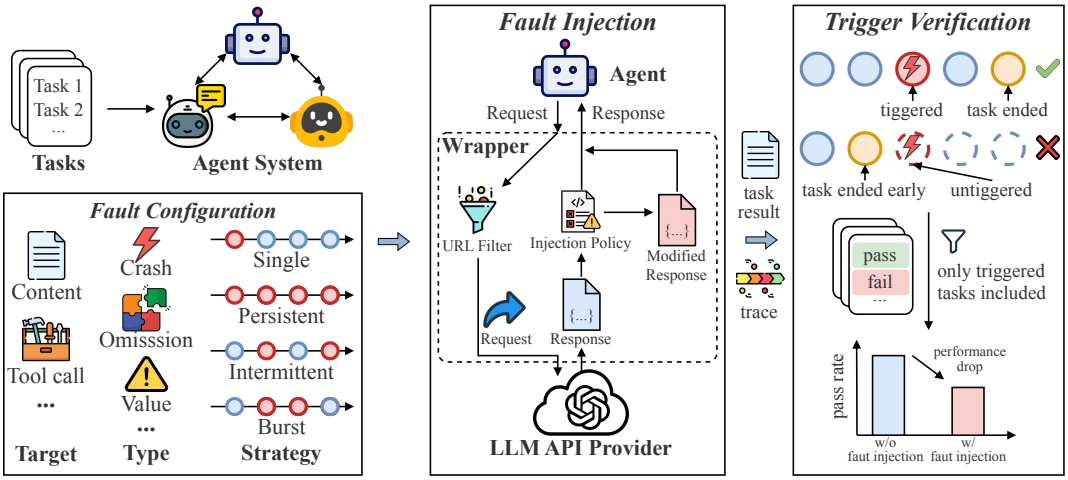

Figure 3: Overview of AgentChaos. The framework takes an agent system, tasks, and a fault configuration as input, injects faults at the HTTP layer, and outputs task results with execution traces for robustness evaluation.

Table 2: Compound fault scenarios. Each reproduces one real deployment failure.

Scenario	Fault Types	Description
API degradation	Timeout, Error	Delay then return error response
Content filter	Empty, Schema	Remove tool calls and replace content with filter message
Max tokens	Truncate	Truncate content and set finish reason to length
Proxy HTML	Corrupt	Replace content with an HTML error page
Stale cache	Corrupt	Replay previous response on the next call
Stale data	Schema	Replace tool call arguments with wrong values
Wrong entity	Schema	Replace tool call arguments with ambiguous values
Slow response	Timeout	Add delay with no content change

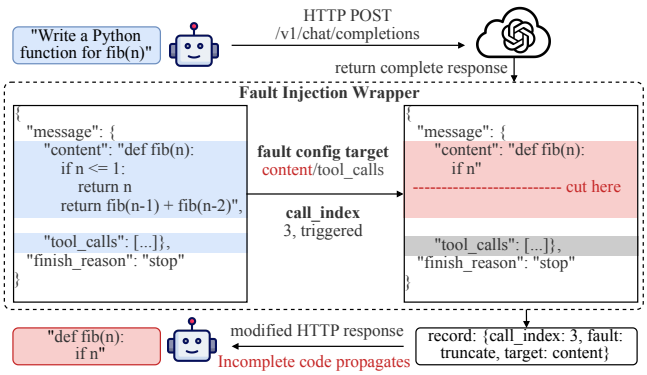

Figure 4: Fault injection wrapper. It intercepts LLM API responses, injects the configured fault, records the change in the execution trace, and returns the response to the agent.

Compound experiments use the 8 scenarios in Table 2. Adding up the three groups gives 48 + 9 + 8 = 65 fault configurations.

4.3 Non-Intrusive Fault Injection

To address the offline and intrusive limitations of existing injection methods, we install a fault injection wrapper at the HTTP layer shared by all agent systems (§2.1). The wrapper provides controlled fault injection at runtime and requires no modification to any agent system source code.

Wrapper installation. Before execution starts, we replace the HTTP client's request method with the wrapper, so that every outgoing request passes through the wrapper before reaching the LLM provider. To support concurrent execution in which multiple tasks run in parallel with different fault configurations, each task is bound to a separate injection state, ensuring that fault sequences are isolated across tasks. In our implementation, we use monkey patching on `httpx.AsyncClient` and Python's `contextvars` for per-task state isolation.

Request interception. At each HTTP request, the wrapper performs the following steps (Fig. 4). First, it checks whether the request URL matches an LLM API endpoint (e.g., `/chat/completions`). Requests to other URLs pass through unchanged. Second, the wrapper ensures that the full response body is available before modification. Third, the wrapper forwards the request to the real LLM provider and receives the complete response. Fourth, it evaluates the injection policy (described below) to decide whether to inject a fault at this call, and applies the modification if the policy fires. Finally, the wrapper records the call in the execution trace and returns the (possibly modified) response to the agent system.

Injection policy. The injection policy decides whether to inject a fault at a given LLM call. The wrapper maintains a call counter and a fire counter per task. The call counter increments each time the wrapper intercepts an LLM API request. The fire counter tracks how many times the fault has actually been injected so far.

Before checking the injection strategy, the wrapper applies a target field guard. Since `content` and `tool_calls` are mutually exclusive (§2.1), the wrapper checks whether the response contains the configured target field. If not, it skips this call and lets the response pass through without incrementing the fire counter. This guard ensures that faults are applied only to responses that contain the intended target field.

If the target field matches, the wrapper evaluates the injection strategy using the call counter and fire counter. For *single*, the policy fires once (when the fire counter is 0 and the call counter has reached the configured start position), then deactivates. For *persistent*, the policy fires on every matching call from the start position onward. For *intermittent*, the policy fires on each matching call with the configured probability (default 0.3), which is evaluated

independently per call for reproducibility. For *burst*, the policy fires on consecutive matching calls while the fire counter is below the configured burst count (default 3), then deactivates.

Modification functions. Each fault type maps to a deterministic modification function that operates on the response JSON at `choices[0].message`. We describe how each function modifies the `content` field (plain text) and the `tool_calls` field (structured JSON parsed, modified recursively, and serialized back) respectively.

For *crash faults*, **error** replaces the `content` field with an error like "HTTP 500 Internal Server Error", and replaces `tool_calls` arguments with an empty object. **Timeout** replaces the `content` field with a timeout notification and clears `tool_calls`.

For *omission faults*, **empty** sets the `content` field to an empty string, or the `tool_calls` field to an empty list. **Truncate** keeps the first 30% of the target field (configurable). For `content`, it also sets `finish_reason` to "`length`" to match a token limit cutoff. For `tool_calls`, it parses the JSON arguments, truncates all string values to 30% of their length, and serializes the result to valid JSON.

For *value faults*, **corrupt** applies UTF-8 to Latin-1 misinterpretation on the `content` field, producing visually garbled but structurally intact text. For `tool_calls`, it replaces approximately 20% of characters in all string values with random Unicode symbols. **Schema** replaces the `content` field with a JSON error object (e.g., {"`error`", "`content_policy_violation`"}), and replaces `tool_calls` arguments with a JSON object containing unexpected keys (e.g., {"`wrong_param`", "`unexpected_value`"}). In real deployments an LLM can return valid JSON whose keys do not match the tool's expected schema, a failure that tool-call studies report as common [35, 37, 58]. We reproduce this failure with a fixed placeholder key, since the mismatch is what matters, not the key name.

All modification functions are deterministic given the same configuration and response, and each keeps the response body parseable as valid JSON. The wrapper also caches each original response to support the replay action used in compound scenarios such as stale cache (Table 2).

Trace recording. After each interception, the wrapper records the call as a span in the execution trace, covering every LLM API call in the task rather than only the injected one. Each span contains the call index, the request content, the original response, and the modified response (if a fault was applied). When the wrapper applies a fault, it additionally logs a fault event that records the call index, fault type, and target field. This trace provides the complete record needed for trigger verification (§4.4) and fault diagnosis (§5.4).

4.4 Trigger Verification

An agent system decides how many LLM calls to make at runtime, and this number varies across tasks. A fault configured for a specific call position may not fire if the task finishes before reaching that position. For example, if a fault is configured for the 3rd LLM call but the task finishes in 2 calls, the fault is never applied. The intermittent strategy may also not select any call during a short task. The target field guard (§4.3) may cause additional skips if the LLM consistently returns tool calls when a content fault is configured. Including such untriggered tasks in the evaluation would mix faulted and unfaulted results, making the system appear more robust than it actually is.

We therefore check after each task whether the execution trace contains at least one fault event (as recorded by the wrapper in §4.3). If no fault event exists, the task is marked as untriggered and excluded from the evaluation. Only triggered tasks are used to compute pass@$1_{\text{w/ FI}}$ and Δpass@1. This filtering ensures that the measured degradation reflects the true impact of the injected fault.

5 Experimental Evaluation

In this section, we evaluate AgentChaos by answering the following research questions:

- **RQ1:** How robust are agent systems under LLM API faults?
- **RQ2:** How do different fault configurations affect robustness?
- **RQ3:** How effectively can existing methods diagnose the injected fault type and step?

5.1 Experiment Setup

Agent Systems. We evaluate agent systems covering diverse architectural patterns. We select one public system for each structurally distinct pattern: AutoGen for conversation, MAD for debate, MapCoder for pipeline, EvoMAC for evolutionary, and Mini-SE for single-agent. Production agents build on the same patterns [2, 31], so these systems cover the main agent styles. To ensure fair comparison and uniform fault injection, we reimplement all systems on Google ADK [17] with unified tool interfaces, preserving each system's original interaction logic while standardizing the underlying LLM API layer.

- **AutoGen** [46] uses a conversation pattern in which an assistant and a user proxy iteratively generate and execute code.
- **MAD** [24] uses a debate pattern in which agents argue across rounds under a moderator before a judge selects the final answer.
- **MapCoder** [22] uses a pipeline pattern in which a retriever, planner, coder, verifier, and debugger run in fixed stages.
- **EvoMAC** [19] uses an evolutionary pattern in which agents decompose tasks and refine code across generations.
- **Mini-SE** [47] uses an agent to search, view, and edit files and submit patches for multi-file software issues.

Datasets. We use benchmarks for code generation, knowledge reasoning, mathematical reasoning, and repository-level bug fixing. AutoGen, MAD, MapCoder, and EvoMAC are evaluated on all benchmarks except SWE-bench Pro, which requires specialized software engineering tools unavailable to these systems. Mini-SE is evaluated only on SWE-bench Pro. To reduce computational cost, we randomly sample 300 examples from large datasets and use all examples from smaller datasets.

- **HumanEval** [9] contains manually crafted Python problems with test cases for correctness.
- **MBPP** [3] contains crowd-sourced Python problems, each with a requirement, function signature, and test cases.
- **HumanEval+ / MBPP+** [26] are EvalPlus extensions of HumanEval and MBPP with more tests for stricter evaluation.
- **MMLU-Pro** [43] contains curated reasoning questions from textbooks and exams across domains.
- **MATH-500** [42] contains challenging math problems that require multi-step reasoning.
- **SWE-bench Pro** [13] contains long-horizon software engineering tasks that often require patches across files.

Fault assignment. We construct 65 fault configurations from the taxonomy. (1) *Basic experiments* combine fault types, target fields,

Table 3: Pass@1 without fault injection (w/o) and with fault injection (w/), and Δpass@1 (Δ) for each model, system, and dataset. Mini-SE is evaluated only on SWE-bench Pro. Bold: highest Δpass@1 per dataset within each model.

Model	System	HumanEval			HumanEval+			MBPP			MBPP+			MMLU-Pro			MATH-500			SWE-bench Pro		
		w/o	w/	Δ	w/o	w/	Δ	w/o	w/	Δ	w/o	w/	Δ	w/o	w/	Δ	w/o	w/	Δ	w/o	w/	Δ
Claude-Sonnet-4.5	AutoGen	98.61	79.17	19.44	98.59	77.46	21.13	91.35	74.04	17.31	74.19	62.58	11.61	58.97	51.92	7.05	29.32	20.94	8.38	–	–	–
	MAD	83.44	59.24	24.2	82.8	57.96	24.84	83.67	59.18	24.49	58.66	43.58	15.08	65.84	45.2	20.64	68.42	47.72	20.7	–	–	–
	MapCoder	96.53	47.92	**48.61**	97.22	47.92	**49.3**	99.55	58.48	**41.07**	75.61	34.76	**40.85**	76.96	38.71	**38.25**	60.48	26.21	**34.27**	–	–	–
	EvoMAC	98.73	80.25	18.48	98.7	80.52	18.18	97.5	80.83	16.67	78.19	63.46	14.73	75.17	61.54	13.63	64.79	48.94	15.85	–	–	–
	Mini-SE	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	11.3	10.43	**0.87**
GPT-5.2	AutoGen	95.71	74.29	21.42	100	77.14	22.86	92.23	73.79	18.44	77.48	60.93	16.55	48.78	37.4	11.38	68.03	55.74	12.29	–	–	–
	MAD	87.83	69.57	18.26	90.16	68.85	21.31	86.6	68.04	18.56	63.06	54.48	8.58	41.36	30.37	10.99	50.93	40.28	10.65	–	–	–
	MapCoder	91.89	42.57	**49.32**	93.96	44.3	**49.66**	99.57	56.65	**42.92**	69.76	26.95	**42.81**	33.88	14.69	**19.19**	31.56	11.03	**20.53**	–	–	–
	EvoMAC	89.22	65.69	23.53	87.38	61.17	26.21	84.42	64.94	19.48	67.26	50.67	16.59	52.33	43.01	9.32	48.45	40.72	7.73	–	–	–
	Mini-SE	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	3.47	6.18	**−2.71**
DeepSeek-V3.2	AutoGen	89.42	74.04	15.38	91.26	77.67	13.59	89.22	74.25	14.97	66.06	57.3	8.76	55.8	58.04	−2.24	57.89	39.04	18.85	–	–	–
	MAD	41.13	23.4	17.73	25.17	14.97	10.2	24.77	15.77	9	40.06	28.39	11.67	29.39	18.7	10.69	30.35	20.62	9.73	–	–	–
	MapCoder	92.96	46.48	**46.48**	93.71	45.45	**48.26**	100	59.38	**40.62**	71.21	28.79	**42.42**	50.37	24.81	**25.56**	41.83	17.87	**23.96**	–	–	–
	EvoMAC	96.69	76.82	19.87	97.42	78.71	18.71	95.34	78.81	16.53	68.56	53.89	14.67	54.22	42.57	11.65	47.62	38.1	9.52	–	–	–
	Mini-SE	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	5.96	6.88	**−0.92**
Seed-1.8	AutoGen	94.67	74.67	20.0	98.63	73.97	24.66	93.4	76.42	16.98	71.24	58.17	13.07	35.61	29.55	6.06	24.46	24.46	0	–	–	–
	MAD	75.16	56.05	19.11	75.0	57.69	17.31	73.58	59.76	13.82	69.41	60.06	9.35	29.93	27.82	2.11	50.35	39.79	10.56	–	–	–
	MapCoder	89.93	43.17	**46.76**	92.09	46.04	**46.05**	99.08	58.06	**41.02**	66.47	29	**37.47**	38.22	16.99	**21.23**	30.65	14.18	**16.47**	–	–	–
	EvoMAC	96.58	76.03	20.55	95.92	77.55	18.37	94.82	75.65	19.17	70.79	53.26	17.53	47.24	36.21	11.03	45.49	35.02	10.47	–	–	–
	Mini-SE	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	5.7	4.39	**1.31**

and injection strategies. (2) *Position experiments* inject faults at different call positions. (3) *Compound experiments* combine multiple fault types in a single execution. We uniformly distribute configurations across tasks, so each task receives one configuration.

Metrics. We measure task success using pass@1 [9], the percentage of tasks solved correctly with one attempt. We run each system under two conditions, without fault injection (w/o FI) and with fault injection (w/ FI). We quantify robustness degradation by Δpass@1, the difference in pass@1 before and after fault injection. A higher Δpass@1 indicates lower robustness, where Δpass@1 = $\text{pass@1}_{\text{w/o FI}} - \text{pass@1}_{\text{w/ FI}}$.

We compute $\text{pass@1}_{\text{w/ FI}}$ only over tasks where the fault was actually triggered, as described in §4.4. For robustness evaluation, we report pass@1 and Δpass@1. For fault diagnosis, we report **type accuracy** and **step accuracy**, where the ground truth is the injected fault type and the step index of the first injection. Type accuracy is the percentage of failed tasks where the predicted type matches the ground truth. Step accuracy is the percentage where the predicted step matches.

Implementation. We implement AgentChaos in Python 3.12 and test with LLMs from different providers: `Claude-Sonnet-4.5` [1], `GPT-5.2` [30], `DeepSeek-V3.2` [12], and `Seed-1.8` [32], all with temperature 0.7, because it is a common default in existing studies [10, 41]. Code execution tools run in sandboxed subprocesses with a 30-second timeout. Each task allows up to 50 LLM calls for multi-agent systems and 100 for Mini-SE. Experiments run on Ubuntu 24.04 with Intel Xeon Gold 6326 CPU and 128GB RAM.

5.2 RQ1: Robustness of Agent Systems

Table 3 presents pass@1 and Δpass@1 for all models, systems, and datasets, while Table 4 reports resource consumption. Unless otherwise noted, analyses use `Claude-Sonnet-4.5`.

Overall degradation. Table 3 shows that fault injection lowers pass@1 for all systems on most datasets. Δpass@1 ranges from 0.87% (Mini-SE on SWE-bench Pro) to 49.66% (MapCoder on HumanEval+ under `GPT-5.2`). Even AutoGen, with the lowest Δpass@1 among these systems, drops from 98.61% to 79.17% on HumanEval (Δ=19.44%). These results confirm that LLM API faults degrade performance across systems and datasets. These numbers are measured only on triggered tasks (§4.4). Table 5 reports the trigger rate, which stays high on most systems but drops on AutoGen, whose short call chains often finish before reaching the fault's injection position. While this degradation is expected, its patterns are not, as shown below. Mini-SE shows the smallest Δpass@1 because its pass@1 without fault injection is already low (3.47% to 11.3% across models), leaving limited room for further degradation. Some cells show negative Δpass@1 (e.g., AutoGen −2.24% on MMLU-Pro under `DeepSeek-V3.2`, Mini-SE −2.71% on SWE-bench Pro), because each fault configuration applies to only a few triggered tasks (e.g., 300 / 65 ≈ 4.6 per configuration) and LLM outputs vary across independent runs at temperature 0.7. We repeat the full evaluation three times with independent seeds and find the robustness ranking stays similar, so the temperature-induced variation has little effect on our conclusion.

Table 4: Average LLM and tool call counts per task without fault injection (w/o) and with fault injection (w/), and their ratio. Backbone LLM is `Claude-Sonnet-4.5`.

System	LLM Calls			Tool Calls		
	w/o	w/	Ratio	w/o	w/	Ratio
AutoGen	1.72	7.3	4.24×	0.47	3.07	6.55×
MAD	11.74	13.32	1.14×	7	7.02	1×
MapCoder	7.34	5.2	0.71×	1.29	1.2	0.93×
EvoMAC	10.76	9.31	0.87×	3.95	3.58	0.91×
Mini-SE	21.73	36.07	1.66×	20.2	29.02	1.44×

Degradation by system implementation. Degradation depends on how each system propagates or recovers from faults. MapCoder is the least robust, dropping 48.61% on HumanEval, 41.07%

Table 5: Trigger rate per system and target field. Backbone LLM is `Claude-Sonnet-4.5`.

System	Overall	Content	Tool_calls	Compound
AutoGen	48.30%	69.63%	12.41%	69.01%
MAD	99.49%	100.00%	98.79%	99.42%
MapCoder	86.03%	100.00%	63.79%	95.32%
EvoMAC	98.65%	100.00%	96.38%	100.00%
Mini-SE	87.67%	76.73%	100.00%	100.00%

Table 6: Failure mode breakdown per agent system. Backbone LLM is `Claude-Sonnet-4.5`.

System	Aborted	Reported	Silent
AutoGen	1.44%	32.85%	65.71%
MAD	8.70%	61.12%	30.19%
MapCoder	4.69%	73.49%	21.82%
EvoMAC	11.85%	48.29%	39.86%
Mini-SE	13.81%	25.10%	61.09%

on MBPP, and 38.25% on MMLU-Pro, because its pipeline feeds each agent's output into the next stage, so a fault propagates through all downstream stages without recovery. MAD drops 24.2% on HumanEval and 20.7% on MATH-500, because a fault on its single moderator propagates directly to the final answer despite aggregation across debaters. EvoMAC drops 18.48% on HumanEval and 15.85% on MATH-500, because its iterative refinement allows later generations to recover from earlier faults, making it the most robust system in our evaluation. AutoGen drops 19.44% on HumanEval on triggered tasks, but its low LLM call count (μ=1.72 per task) means most configured faults do not trigger, so its effective degradation across all tasks is lower than EvoMAC. Mini-SE drops only 0.87% on SWE-bench Pro, because its baseline pass@1 is already low (3.47% to 11.3%) and its high LLM call count (μ=21.73) dilutes the impact of a single fault.

Consistency across models and task domains. The degradation pattern is consistent across models. MapCoder, for example, shows the highest Δpass@1 under every model on every dataset. Our faults modify the LLM response in the same way for all models, so the resulting degradation appears to depend on how the system implementation handles the faulty response rather than on which model produced it. Across task domains, all systems show higher Δpass@1 on code generation tasks (HumanEval, HumanEval+, MBPP) than on reasoning tasks (MMLU-Pro, MATH-500). For example, under `Claude-Sonnet-4.5`, MapCoder drops 48.61% on HumanEval but 38.25% on MMLU-Pro. Faults on generated code directly produce incorrect programs that fail test cases, while faults on reasoning text may still yield a correct answer if the core logic remains complete.

Resource overhead. Faults affect LLM and tool call counts per task (Table 4). AutoGen increases from 1.72 to 7.3 LLM calls (4.24×) and from 0.47 to 3.07 tool calls (6.55×), because faulty responses trigger repeated retries. MapCoder decreases from 7.34 to 5.2 LLM calls (0.71×), because faults cause its pipeline to terminate before reaching later stages. Mini-SE increases from 21.73 to 36.07 LLM calls (1.66×), because faults trigger additional debugging steps. The direction depends on each system's fault handling. Systems that retry after faults consume more, while systems that fail early consume fewer. The wrapper adds less than 1ms per intercepted call, which is negligible compared to LLM API response times that range from 1s to 30s.

Table 7: Δpass@1 (%) by fault configuration and system. Backbone LLM is `Claude-Sonnet-4.5`. Results aggregated over all datasets. Bold: highest drop per system within each section.

Config	Target	AutoGen	MAD	MapCoder	EvoMAC	Mini-SE
Fault Type						
Error	Content	23.75	37.5	59.62	42.31	0
Error	Tool call	−12.5	0	0	4.05	0
Timeout	Content	**28.21**	22.33	**64.42**	**45.19**	−20
Empty	Content	22.5	**38.46**	56.73	44.23	9.09
Truncate	Content	21.25	28.85	58.65	40.38	**14.29**
Truncate	Tool call	0	2.74	52.63	−3.95	0
Corrupt	Content	−2.38	−1.92	2.88	0	5.88
Schema	Content	21.69	**38.46**	57.69	43.27	−30
Schema	Tool call	11.11	0	−2.63	4.76	7.69
Injection Strategy						
Single		1.66	22.33	48.23	3.64	3.12
Burst		0	13.33	46.81	34.78	−1.69
Intermittent		−3.57	8.47	−1.27	1.77	8.7
Persistent		**57.41**	**54.74**	**62.39**	**47.64**	**10**
Injection Position						
1st call		−4.84	−8.06	**83.87**	**6.45**	0
2nd call		11.11	**24.19**	1.61	−1.61	0
3rd call		**16.67**	4.84	−4.84	3.23	0
Compound Scenario						
Single-fault avg.		17.15	22.98	43.1	20.68	3.01
API degradation		0	68.18	81.82	4.55	0
Content filter		0	45.45	**86.36**	0	−33.33
Max tokens		−18.18	9.09	**86.36**	−9.09	0
Proxy HTML		−4.76	**76.19**	80.95	0	−25
Slow response		9.52	−9.52	9.52	**4.76**	0
Stale cache		–	0	0	−4.76	0
Stale data		−25	−14.29	−5.88	−9.52	0
Wrong entity		**33.33**	−10	−5.88	0	−25

Failure mode breakdown. Beyond the size of the drop, we also examine how each task fails, since an error shown to the user is a safe reaction while a wrong answer with no warning is not. We classify each failed task into Aborted (the run stops), Reported (the output tells the user something went wrong), and Silent (plausible but wrong output with no warning). Table 6 reports the breakdown per system. Reported is a safe reaction and covers most failures on MAD and MapCoder, and Aborted runs stay low across all systems, while the share of Silent failures varies widely, from 21.82% on MapCoder to 65.71% on AutoGen.

> **Answer to RQ1:** All agent systems degrade under LLM API faults, reaching Δpass@1 of 49.66% (MapCoder). The robustness ranking is MapCoder (highest drop), MAD, EvoMAC, AutoGen (lowest drop), and this ranking is similar across all backbone LLMs. Mini-SE shows only 0.87% drop because its pass@1 without fault injection is already below 12% and each task makes over 20 LLM calls, so one faulty call has limited effect. Faults also change resource consumption unpredictably (0.71× to 4.24×).

5.3 RQ2: Impact of Fault Configurations

Beyond overall degradation, we analyze how degradation varies across fault configurations. Table 7 presents Δpass@1 by fault type, injection strategy, injection position, and compound scenario, aggregated over all datasets.

Fault type and target field. Faults targeting the content field cause higher Δpass@1 than faults targeting the tool call field. Timeout, empty, truncate, schema, and error content faults all cause Δpass@1 above 15% on at least one system. The most damaging

type differs across systems: timeout for AutoGen (28.21%), MapCoder (64.42%), and EvoMAC (45.19%); empty and schema for MAD (38.46% each); and truncate for Mini-SE (14.29%). Among content faults, only corrupt remains below 7% across all systems (−2.38% to 5.88%), because agents can recognize and ignore garbled characters. Most tool call faults cause Δpass@1 below 5% (e.g., error ranges from −12.5% to 4.05%), because tool error handling catches them before they propagate to subsequent agents. However, truncate causes 52.63% on MapCoder because its pipeline passes corrupted tool call arguments to the next stage without validation. Since the most damaging type cannot be predicted in advance, evaluating only obvious fault types risks missing the most harmful one. Mini-SE shows low Δpass@1 for most types because its baseline pass@1 is already low, although truncate content (14.29%) and empty content (9.09%) still cause notable drops. Extreme negative values on Mini-SE (e.g., schema −30%) reflect the few triggered SWE-bench Pro tasks, where one task flip causes large swings.

Injection strategy. Persistent injection on every LLM call causes the highest Δpass@1 across systems: MapCoder (62.39%), AutoGen (57.41%), MAD (54.74%), EvoMAC (47.64%), and Mini-SE (10%). AutoGen ranks second despite its lowest Δpass@1 in §5.2, because faults on each call remove its advantage of making few calls per task. Burst injection ranks second for MapCoder (46.81%) and EvoMAC (34.78%), as consecutive faults reduce recovery between calls. Single injection causes high Δpass@1 for MapCoder (48.23%) and MAD (22.33%) because a critical fault propagates through downstream pipeline and debate. AutoGen and EvoMAC remain low (1.66% and 3.64%) because subsequent rounds recover from a single fault. Intermittent injection has the least impact for most systems because probabilistic triggering produces few injections per task.

Injection position. Injection-position sensitivity differs across architectures. We inject at the first, second, and third calls because later calls rarely fire on systems with short call chains (Table 4). Pipeline systems are most sensitive because early faults propagate to all downstream steps. MapCoder ranges from −4.84% to 83.87% across positions. Debate systems are moderately sensitive, with MAD ranging from −8.06% to 24.19%. Iterative systems are least sensitive, with EvoMAC ranging from −1.61% to 6.45%, because later rounds can recover from faults at any position. AutoGen ranges from −4.84% to 16.67%. Most AutoGen tasks complete in few LLM calls (μ=1.72), so its retry mechanism absorbs a fault at the 1st call, and the −4.84% reflects noise from the few triggered tasks.

Compound scenario. Beyond single faults, compound faults that modify response content cause higher Δpass@1 than those affecting only latency or metadata. MapCoder shows Δpass@1 of 86.36% under max tokens and content filter, 81.82% under API degradation, and 80.95% under proxy HTML. MAD shows high Δpass@1 under proxy HTML (76.19%) and API degradation (68.18%). AutoGen shows the highest Δpass@1 under wrong entity (33.33%), which replaces named entities in responses. Stale cache shows "–" for AutoGen because it replays a previous response, but most AutoGen tasks make one LLM call and have no previous response to replay. Compound faults affecting only latency or metadata cause low Δpass@1. Slow response, stale cache, and stale data show Δpass@1 below 10% for most systems, confirming that latency alone does not affect task correctness. EvoMAC shows low or negative Δpass@1 across most compound scenarios (−9.52% to 4.76%), because its iterative refinement recovers from compound faults like single faults. Mini-SE shows 0% on most compound scenarios because it is evaluated on SWE-bench Pro alone, so each scenario applies to few triggered tasks, and its low baseline pass@1 means most tasks fail regardless of fault injection.

Table 8: Sensitivity check on intermittent probability (prob) and burst count (burst). Mini-SE on SWE-bench Pro, others on HumanEval+. Backbone LLM is `Claude-Sonnet-4.5`.

System	prob=0.1	prob=0.3	prob=0.5	burst=1	burst=3	burst=5
AutoGen	0.00	0.00	0.00	0.00	0.00	0.00
MAD	15.50	13.30	17.20	45.00	41.70	46.70
MapCoder	0.00	0.00	0.00	42.40	42.40	42.40
EvoMAC	0.00	0.00	0.00	0.00	43.10	41.70
Mini-SE	0.00	0.00	0.00	0.00	0.00	0.00

Table 9: Fault diagnosis accuracy (%) on Mini-SE (SWE-bench Pro). We evaluate all failed cases under fault injection across all backbone models. Rule: rule-based diagnosis. LLM: LLM-based diagnosis (`Claude-Sonnet-4.5`). κ: Cohen's kappa between the two methods.

	Type Accuracy			Step Accuracy		
Fault Type	Rule	LLM	κ	Rule	LLM	κ
Error	**47.57**	**47.57**	0.813	**69.90**	66.02	0.862
Timeout	**91.04**	87.31	0.411	**89.55**	88.81	0.907
Empty	**96.74**	77.17	0.098	**38.04**	31.52	0.629
Truncate	4.3	**34.41**	0.203	31.18	**40.86**	0.381
Corrupt	**13.64**	6.36	0.163	**26.36**	20	0.322
Schema	**52.46**	27.05	0.372	**63.93**	60.66	0.762
Overall	**52.45**	47.25	0.567	**55.5**	53.52	0.661

Sensitivity to parameters. We vary the intermittent probability and the burst count on a set of tasks from HumanEval+ and SWE-bench Pro and measure their effect on Δpass@1 (Table 8). When the probability moves across 10%, 30%, and 50%, each Δpass@1 changes by less than 5 percentage points, and 30% falls in the middle of this stable range. The burst count is stable for most systems as well, and does not change beyond a burst of 3.

> **Answer to RQ2:** Fault impact depends on configuration. Content faults cause higher drops than tool call faults, and the most damaging type varies across systems. Persistent injection causes the highest Δpass@1 (up to 62.39%). Injection position matters most for pipeline systems, where a fault at the first stage drops pass@1 by up to 83.87%. Compound faults that modify content amplify degradation (up to 86.36%).

5.4 RQ3: Fault Diagnosis

We evaluate how well existing methods diagnose the injected fault type and step from execution traces. We focus on Mini-SE on SWE-bench Pro, because Mini-SE produces the longest traces among all tested systems (μ=21.73 LLM calls per task), making fault diagnosis most challenging. We select this setting because it represents the hardest case for diagnosis. If methods already struggle on long traces, there is even more reason to improve diagnosis before applying it to production systems. We run Mini-SE with all backbone models under fault injection and collect all failed cases (654 in total across all models and fault configurations). We apply two diagnosis methods to each case. (1) *Rule-based diagnosis* is a baseline we

design for this evaluation. It scans each step in the execution trace and matches the LLM API response against common fault signatures in order, such as HTTP error codes, timeout messages, empty response fields, encoding corruption patterns, and incomplete text. It returns the first match as the predicted fault type and step. These patterns match general LLM API error formats that developers encounter in production [28]. We write them from public API error documentation rather than from the content we inject, so they are not tailored to our injection implementation. (2) *LLM-based diagnosis* [55] feeds the full execution trace, the task query, and the system architecture into an LLM in a single prompt and asks it to predict the fault type and the step where the fault first occurred. The prompt lists the fault-type definitions, but does not tell the model which fault we injected or at which step. This method can detect faults that lack fixed patterns, such as truncated or corrupted content, but may struggle to locate the exact step in long traces. Both methods receive the same trace information.

Overall diagnosis accuracy. As shown in Table 9, rule-based diagnosis achieves 52.45% type accuracy and 55.5% step accuracy. LLM-based diagnosis achieves 47.25% type accuracy and 53.52% step accuracy. Rule-based diagnosis achieves 5.2% higher type accuracy than LLM-based diagnosis, while both methods achieve comparable step accuracy (within 2%). Neither method exceeds 56% on any metric, indicating that current diagnosis methods cannot reliably identify fault type or step from agent execution traces.

Accuracy by fault type. Diagnosis accuracy varies by fault category. On crash faults, both methods achieve high accuracy. For type accuracy, rule-based diagnosis achieves 91.04% on timeout and 47.57% on error, and LLM-based diagnosis achieves 87.31% and 47.57%, because crash faults produce distinctive patterns such as HTTP status codes and connection failures. For step accuracy, both methods remain high on timeout (rule-based 89.55%, LLM-based 88.81%) and error (rule-based 69.9%, LLM-based 66.02%). On omission faults, the two methods diverge. For empty, rule-based diagnosis achieves 96.74% type accuracy, while LLM-based diagnosis achieves 77.17%, because empty content fields are easy to match by fixed patterns but LLM-based diagnosis can confuse them with normal short responses. For truncate, rule-based diagnosis achieves only 4.3% type accuracy, while LLM-based diagnosis achieves 34.41%, because truncated output does not match any fixed pattern but LLM-based diagnosis can detect incomplete content. Step accuracy is low for both methods on empty (rule-based 38.04%, LLM-based 31.52%) and truncate (rule-based 31.18%, LLM-based 40.86%), because omission faults do not leave clear markers at the exact fault step. On value faults, the two methods both achieve low accuracy on corrupt (rule-based 13.64% type, LLM-based 6.36%), because corrupted encoding resembles normal text. Schema faults are easier to detect within this category, with rule-based achieving 52.46% type accuracy and LLM-based achieving 27.05%, because schema violations produce recognizable structural errors. Step accuracy is moderate on schema (rule-based 63.93%, LLM-based 60.66%) but low on corrupt (rule-based 26.36%, LLM-based 20%).

Complementarity. Cohen's κ between the two methods is 0.567 for type and 0.661 for step overall, indicating moderate agreement. On crash faults, κ is high (κ=0.813 for error type, κ=0.907 for timeout step), because crash faults produce clear signals that both methods can detect. On omission and value faults, κ is low (κ=0.098 for empty type, κ=0.203 for truncate type), because rule-based diagnosis relies on structural patterns while LLM-based diagnosis relies on semantic content. Rule-based diagnosis captures empty faults that LLM-based diagnosis misses, while LLM-based diagnosis captures truncate faults that rule-based diagnosis misses. Combining both approaches, for example by using rule-based diagnosis for crash and empty faults and LLM-based diagnosis for truncate faults, can improve overall accuracy.

> **Answer to RQ3:** Neither rule-based nor LLM-based diagnosis exceeds 56% overall accuracy. Crash faults are easy to diagnose (up to 91.04%), but omission faults such as truncation, which cause significant damage on most systems (§5.3), are among the hardest to diagnose (4.3% type accuracy by rule-based, 34.41% by LLM-based). Value faults such as corruption are also hard to diagnose (6.36% type accuracy) and hard to detect at runtime. The two methods are complementary (κ=0.567), so routing each fault category to the better method can improve accuracy.

6 Discussion

6.1 Implications for Practice

Our results provide practical guidance for agent system developers and framework designers. We use two terms throughout this section. A fault is *severe* if it causes the response to return a clear, visible error, such as a server error or a timeout [4]. A fault is *harmful* if it causes a large drop in task success (Δpass@1), regardless of how visible it is [21]. These two properties are independent: a fault can look alarming yet cause little degradation, or look normal yet degrade the task success rate sharply.

The most severe faults are not the most harmful. Omission faults such as truncation and empty responses cause performance drops comparable to crash faults such as error and timeout on most systems (§5.3), despite appearing less severe. For example, on MAD, empty content causes a 38.46% drop, close to error content (37.5%) and higher than timeout content (22.33%). Crash faults trigger errors and automatic retries, while omission faults return valid HTTP 200 responses and bypass error handling. Developers should add output validation after each LLM API call, such as checking `finish_reason`, verifying code syntax completeness, and confirming that tool call arguments match the expected schema.

The most harmful faults are also the hardest to diagnose. Omission faults such as truncation and empty responses cause large performance drops across most systems yet remain hard to diagnose. Rule-based diagnosis identifies truncation with only 4.3% accuracy (§5.4). Truncated output looks like weak model output in execution traces, so developers may misattribute the failure to model capability and upgrade the model instead of fixing fault handling. Agent frameworks should log structured metadata for each LLM API call, including token usage relative to the limit, `finish_reason`, and response length, to make truncation detectable during later analysis.

Robustness depends on system implementation. The robustness ranking across systems is similar under all backbone LLMs (§5.2). The reason appears to be that fault handling depends on how the system processes faulty responses rather than on which model produced them. The pipeline system we evaluate (MapCoder) is

the most vulnerable: a single fault at its first stage drops pass@1 by up to 83.87% (§5.3), because each stage consumes the previous output and propagates the fault to all downstream stages. The iterative system is the most robust, as later rounds can observe and correct errors from earlier ones. These mechanisms are tied to how each system structures its LLM calls, so we expect them to carry over to other systems of the same pattern, though confirming this requires evaluating more systems per pattern (§6.2). Replacing the model alone is unlikely to fix these weaknesses. Developers should add stage-level output validation to pipeline systems and consider iterative refinement to recover from faults.

6.2 Threats to Validity

For internal validity, we reimplement all agent systems on Google ADK with unified tool interfaces. Although we preserve each system's original interaction logic, behavioral differences from the original implementations may affect absolute pass@1 values. Because the same reimplementation runs with and without fault injection, Δpass@1 remains a valid robustness measure. We uniformly distribute 65 fault configurations across tasks, so each configuration applies to only a few triggered tasks (e.g., 300 / 65 ≈ 4.6 per configuration). This causes Δpass@1 to fluctuate, as shown by occasional negative values in Table 3. Temperature 0.7 also introduces variance across runs. We reduce this variance by computing Δpass@1 between two runs on the same task set with deterministic fault injection.

For external validity, our evaluation spans five architectural patterns, seven benchmarks across four task domains, and four backbone LLMs from different providers. Two limitations remain. First, new architectures (e.g., RAG-based agents) may exhibit different fault propagation patterns and are not covered. Second, and more importantly, we evaluate only one agent system per architectural pattern (e.g., MapCoder stands for the pipeline pattern), so within each pattern we cannot separate the effect of the pattern from that of the specific system, which is a significant threat to validity. We therefore read our robustness ranking as a pattern that holds consistently across the systems and models we test, not as a proven property of the architectural patterns themselves. Establishing the latter would require evaluating additional agent systems within each pattern. For construct validity, we measure robustness using Δpass@1, which captures overall performance degradation but not partial correctness or output quality. We adopt pass@1 because it is the standard metric across our benchmarks. This supports comparison with existing studies and makes the threat minimal. Future work can complement Δpass@1 with finer-grained metrics such as partial test case pass rates or output similarity scores.

7 Related Work

7.1 Fault Injection for Agent Systems

Existing fault injection methods can be grouped into agent-oriented offline methods, agent-oriented runtime methods, and infrastructure tools. AgenTracer [53] and AutoInject [21] operate offline. AgenTracer uses an LLM to perturb completed execution traces and replays them to construct fault datasets. AutoInject analyzes how fault propagation patterns depend on organizational structures at the trace level. Because these methods operate on static traces rather than running systems, they cannot capture runtime behaviors such as retries or early termination. MAS-FIRE [23] operates at runtime but is intrusive. It modifies prompts, rewrites responses, and routes messages to inject semantic faults such as hallucination and role ambiguity. However, it requires adapting the injection logic to each system and does not target LLM API faults such as response truncation or encoding corruption. Chaos engineering tools inject infrastructure faults but cannot modify fields within LLM API response bodies. Rainmaker [11] injects transient HTTP faults including 503 errors and timeouts at the REST layer. CrashFuzz [15] performs coverage-guided crash and reboot injection for cloud systems. ChaosBlade [6] operates at the OS, container, or network layer. These tools can cause API requests to fail, but cannot truncate `message.content` or corrupt `tool_calls` arguments.

In summary, existing methods cannot provide runtime, non-intrusive injection that modifies LLM API response content. Offline methods [21, 53] cannot inject faults into running systems. Runtime methods [23] are intrusive and target semantic rather than LLM API faults. Traditional tools [6, 11, 15] cannot modify response content. We provide this capability at the LLM API layer. This allows us to inject field-level faults such as truncation and corruption across agent frameworks without modifying their source code.

7.2 Failure Analysis of Agent Systems

Existing studies on agent system reliability analyze failure traces after execution through failure mode identification [5, 51], anomaly detection [36], fault localization [16, 55, 57], and error classification [14, 56]. Empirical studies on agent workflow frameworks [48], AI coding tools [54], LLM-integrated applications [33, 34, 38, 39], and open-source LLM deployments [52] further show that failures at LLM API and integration boundaries are common and diverse. However, these studies are observation-based. They neither control which fault occurs or when nor inject faults into running agent systems, so they cannot isolate the effect of a specific LLM API fault type from causes such as model capability limitations or measure its impact on task completion. Security studies address a different problem. AgentFuzz [25] and TaintP2X [18] detect injection vulnerabilities through fuzzing and taint analysis, but they focus on malicious inputs rather than infrastructure faults.

8 Conclusion and Future Work

We present AgentChaos, a chaos engineering framework for controlled, runtime, non-intrusive LLM API fault injection in agent systems. It defines crash, omission, and value faults on content and tool call fields, injects them through an HTTP layer wrapper without modifying source code, and verifies whether each fault is triggered to filter untriggered tasks. Across agent systems, benchmarks, and backbone models, all systems degrade under fault injection, with Δpass@1 ranging from 0.87% (Mini-SE) to 49.66% (MapCoder). The robustness ranking is consistent across models, suggesting that robustness depends on system implementation. Existing fault diagnosis methods remain below 56% accuracy, leaving room for improvement. Future work will extend the fault taxonomy and diagnosis methods to cover faults that are hard to detect. We will also design and evaluate defense strategies that validate LLM API responses before downstream use and retry incomplete responses.

9 Data Availability

The data and implementation are available on Zenodo [40] and GitHub at https://github.com/IntelligentDDS/AgentChaos.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (Grant Number: 2024YFB4505904). The corresponding author is Pengfei Chen. This research is supported by the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] 2025. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5. Accessed: 2026-08-03.

[2] Anthropic. 2025. Claude Code GitHub Actions. https://code.claude.com/docs/en/github-actions. Accessed: 2026-08-03.

[3] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.* 1, 1 (2004), 11–33. doi:10.1109/TDSC.2004.2

[5] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya G. Parameswaran, Dan Klein, Kannan Ramchandran, Matei A. Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why Do Multi-Agent LLM Systems Fail?. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (Eds.). http://papers.nips.cc/paper_files/paper/2025/hash/b1041e52d3be19f0a9bc491657488e4a-Abstract-Datasets_and_Benchmarks_Track.html

[6] ChaosBlade. 2026. Chaosblade: An Easy to Use and Powerful Chaos Engineering Toolkit. https://github.com/chaosblade-io/chaosblade. Accessed: 2026-08-03.

[7] Hongyang Chen, Pengfei Chen, Guangba Yu, Xiaoyun Li, and Zilong He. 2024. MicroFI: Non-Intrusive and Prioritized Request-Level Fault Injection for Microservice Applications. *IEEE Trans. Dependable Secur. Comput.* 21, 5 (2024), 4921–4938. doi:10.1109/TDSC.2024.3363902

[8] Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, Xin Zhou, Xing Hu, and David Lo. 2025. SecureAgentBench: Benchmarking Secure Code Generation under Realistic Vulnerability Scenarios. *CoRR* abs/2509.22097 (2025). arXiv:2509.22097 doi:10.48550/ARXIV.2509.22097

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[10] Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. 2025. SWE-Exp: Experience-Driven Software Issue Resolution. *CoRR* abs/2507.23361 (2025). arXiv:2507.23361 doi:10.48550/ARXIV.2507.23361

[11] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. 2023. Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 1701–1716. https://www.usenix.org/conference/nsdi23/presentation/chen-yinfang

[12] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, and et al. 2025. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. arXiv:2512.02556 [cs.CL] https://arxiv.org/abs/2512.02556

[13] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. 2025. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? *CoRR* abs/2509.16941 (2025). arXiv:2509.16941 doi:10.48550/ARXIV.2509.16941

[14] Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. 2025. TRAIL: Trace Reasoning and Agentic Issue Localization. *CoRR* abs/2505.08638 (2025). arXiv:2505.08638 doi:10.48550/ARXIV.2505.08638

[15] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage Guided Fault Injection for Cloud Systems. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2211–2223. doi:10.1109/ICSE48619.2023.00186

[16] Yu Ge, Linna Xie, Zhong Li, Yu Pei, and Tian Zhang. 2025. Who is Introducing the Failure? Automatically Attributing Failures of Multi-Agent Systems via Spectrum Analysis. *CoRR* abs/2509.13782 (2025). arXiv:2509.13782 doi:10.48550/ARXIV.2509.13782

[17] Google. 2026. Agent Development Kit (ADK). https://github.com/google/adk-python. Accessed: 2026-08-03.

[18] Junjie He, Shenao Wang, Yanjie Zhao, Xinyi Hou, Zhao Liu, Quanchen Zou, and Haoyu Wang. 2026. TaintP2X: Detecting Taint-Style Prompt-to-Anything Injection Vulnerabilities in LLM-Integrated Applications. (2026).

[19] Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2025. Self-Evolving Multi-Agent Collaboration Networks for Software Development. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=4R71pdPBZp

[20] Haojia Huang, Pengfei Chen, Guangba Yu, Haiyu Huang, Jia Chang, Jun Li, and Jian Han. 2025. Conan: Uncover Consensus Issues in Distributed Databases Using Fuzzing-Driven Fault Injection. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2025, Montreal, QC, Canada, March 4-7, 2025*. IEEE, 716–727. doi:10.1109/SANER64311.2025.00073

[21] Jen-tse Huang, Jiaxu Zhou, Tailin Jin, Xuhui Zhou, Zixi Chen, Wenxuan Wang, Youliang Yuan, Michael R. Lyu, and Maarten Sap. 2025. On the Resilience of LLM-Based Multi-Agent Collaboration with Faulty Agents. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025 (Proceedings of Machine Learning Research, Vol. 267)*, Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (Eds.). PMLR / OpenReview.net. https://proceedings.mlr.press/v267/huang25ay.html

[22] Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 4912–4944. doi:10.18653/V1/2024.ACL-LONG.269

[23] Jin Jia, Zhiling Deng, Zhuangbin Chen, Yingqi Wang, and Zibin Zheng. 2026. MAS-FIRE: Fault Injection and Reliability Evaluation for LLM-Based Multi-Agent Systems. arXiv:2602.19843 [cs.SE] https://arxiv.org/abs/2602.19843

[24] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2024. Encouraging Divergent Thinking in Large Language Models through Multi-Agent Debate. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, 17889–17904. doi:10.18653/V1/2024.EMNLP-MAIN.992

[25] Fengyu Liu, Yuan Zhang, Jiaqi Luo, Jiarun Dai, Tian Chen, Letian Yuan, Zhengmin Yu, Youkun Shi, Ke Li, Chengyuan Zhou, Hao Chen, and Min Yang. 2025. Make Agent Defeat Agent: Automatic Detection of Taint-Style Vulnerabilities in LLM-based Agents. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 3767–3786. https://www.usenix.org/conference/usenixsecurity25/presentation/liu-fengyu

[26] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[27] Minghua Ma, Jackson Clark, and Shenglin Zhang. 2025. AIOpsLab in Action: An Open Platform for AIOps Research. In *Proceedings of the 33rd ACM International*

Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025, Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang (Eds.). ACM, 1223–1227. doi:10.1145/3696630.3728619

[28] Microsoft. 2026. Content streaming. https://learn.microsoft.com/en-us/azure/foundry/openai/concepts/content-streaming. Accessed: 2026-08-03.

[29] Yuqing Niu, Jieke Shi, Ruidong Han, Ye Liu, Chengyan Ma, Yunbo Lyu, and David Lo. 2026. What You Trust is Insecure: Demystifying How Developers (Mis)Use Trusted Execution Environments in Practice. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2026, Limassol, Cyprus, March 17-20, 2026*. IEEE, 228–239. doi:10.1109/SANER67736.2026.00034

[30] OpenAI. 2025. ChatGPT. https://chatgpt.com. Accessed: 2026-08-03.

[31] OpenAI. 2025. Introducing ChatGPT agent: bridging research and action. https://openai.com/index/introducing-chatgpt-agent/. Accessed: 2026-08-03.

[32] Bytedance Seed. 2025. Seed1.8 Model Card: Towards Generalized Real-World Agency. https://seed.bytedance.com/en/seed1_8. Accessed: 2026-08-03.

[33] Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. 2025. Are LLMs Correctly Integrated into Software Systems?. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 1178–1190. doi:10.1109/ICSE55347.2025.00204

[34] Yuchen Shao, Yuheng Huang, Jiazhen Zou, Yuling Shi, Long Yang, Lei Ma, Ting Su, and Chengcheng Wan. 2026. Comfrey: Mitigating Integration Failures in LLM-enabled Software at Run-Time. (2026).

[35] Akshey Sigdel and Rista Baral. 2026. Schema First Tool APIs for LLM Agents: A Controlled Study of Tool Misuse, Recovery, and Budgeted Performance. *CoRR* abs/2603.13404 (2026). arXiv:2603.13404 doi:10.48550/ARXIV.2603.13404

[36] Ron Solomon, Yarin Yerushalmi Levi, Lior Vaknin, Eran Aizikovich, Amit Baras, Etai Ohana, Amit Giloni, Shamik Bose, Chiara Picardi, Yuval Elovici, and Asaf Shabtai. 2025. LumiMAS: A Comprehensive Framework for Real-Time Monitoring and Enhanced Observability in Multi-Agent Systems. *CoRR* abs/2508.12412 (2025). arXiv:2508.12412 doi:10.48550/ARXIV.2508.12412

[37] Yewei Song, Xunzhu Tang, Cedric Lothritz, Saad Ezzini, Jacques Klein, Tegawendé F. Bissyandé, Andrey Boytsov, Ulrick Ble, and Anne Goujon. 2025. CallNavi, A challenge and empirical study on LLM function calling and routing. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering, EASE 2025, Istanbul, Turkey, June 17-20, 2025*, Muhammad Ali Babar, Ayse Tosun, Stefan Wagner, and Viktoria Stray (Eds.). ACM, 114–125. doi:10.1145/3756681.3756975

[38] Gou Tan, Zilong He, Min Li, Pengfei Chen, Jieke Shi, Zhensu Sun, Ting Zhang, Danwen Chen, Lwin Khin Shar, Chuanfu Zhang, and David Lo. 2026. LIDL: LLM Integration Defect Localization via Knowledge Graph-Enhanced Multi-Agent Analysis. *CoRR* abs/2601.05539 (2026). arXiv:2601.05539 doi:10.48550/ARXIV.2601.05539

[39] Gou Tan, Zilong He, Min Li, Haiyu Huang, Yilun Wang, Pengfei Chen, Giuliano Casale, and Chuanfu Zhang. 2026. LLMRCA: Multilevel Root Cause Analysis for LLM Applications Using Multimodal Observability Data. *ACM Trans. Softw. Eng. Methodol.* (April 2026). Just Accepted. doi:10.1145/3806200

[40] Gou Tan, Zhensu Sun, Jieke Shi, Ting Zhang, Zilong He, Qingfu Wu, Shuai Liang, Weifeng Sun, Junda He, Pengfei Chen, Chuanfu Zhang, Lwin Khin Shar, and David Lo. 2026. Artifact of Paper "AgentChaos: Chaos Engineering for Agent Systems via Programmatic Fault Injection". doi:10.5281/zenodo.21823973

[41] Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. 2025. Mixture-of-Agents Enhances Large Language Model Capabilities. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=h0ZfDIrj7T

[42] Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 9426–9439. doi:10.18653/V1/2024.ACL-LONG.510

[43] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. 2024. MMLU-Pro: A More Robust and Challenging Multi-Task Language Understanding Benchmark. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/ad236edc564f3e3156e1b2feafb99a24-Abstract-Datasets_and_Benchmarks_Track.html

[44] Yilun Wang, Guangba Yu, Haiyu Huang, Zirui Wang, Yujie Huang, Pengfei Chen, and Michael R. Lyu. 2026. Cloud-OpsBench: A Reproducible Benchmark for Agentic Root Cause Analysis in Cloud Systems. *CoRR* abs/2603.00468 (2026). arXiv:2603.00468 doi:10.48550/ARXIV.2603.00468

[45] Zirui Wang, Guangba Yu, and Michael R. Lyu. 2026. AI-NativeBench: An Open-Source White-Box Agentic Benchmark Suite for AI-Native Systems. *CoRR* abs/2601.09393 (2026). arXiv:2601.09393 doi:10.48550/ARXIV.2601.09393

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

[47] Junjielong Xu, Boyin Tan, Xiaoyuan Liu, Chao Peng, Pengfei Gao, and Pinjia He. 2025. Scalable Supervising Software Agents with Patch Reasoner. *CoRR* abs/2510.22775 (2025). arXiv:2510.22775 doi:10.48550/ARXIV.2510.22775

[48] Ziluo Xue, Yanjie Zhao, Shenao Wang, Kai Chen, and Haoyu Wang. 2025. A Characterization Study of Bugs in LLM Agent Workflow Orchestration Frameworks. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 3369–3380. doi:10.1109/ASE63991.2025.00278

[49] Zhou Yang, Chenyu Wang, Jieke Shi, Thong Hoang, Pavneet Singh Kochhar, Qinghua Lu, Zhenchang Xing, and David Lo. 2023. What Do Users Ask in Open-Source AI Repositories? An Empirical Study of GitHub Issues. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 79–91. doi:10.1109/MSR59073.2023.00024

[50] Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. 2026. A Survey on Evaluation of LLM-based Agents. In *Findings of the Association for Computational Linguistics, ACL 2026, San Diego, California, United States, July 2-7, 2026*, Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (Eds.). Association for Computational Linguistics, 26690–26714. doi:10.18653/V1/2026.FINDINGS-ACL.1330

[51] Guangba Yu, Gou Tan, Haojia Huang, Zhenyu Zhang, Pengfei Chen, Roberto Natella, Zibin Zheng, and Michael R. Lyu. 2026. A Survey on Failure Analysis and Fault Injection in AI Systems. *ACM Trans. Softw. Eng. Methodol.* 35, 1 (2026), 28:1–28:42. doi:10.1145/3732777

[52] Guangba Yu, Zirui Wang, Yujie Huang, Renyi Zhong, Yuedong Zhong, Yilun Wang, and Michael R. Lyu. 2026. Why Does the LLM Stop Computing: An Empirical Study of User-Reported Failures in Open-Source LLMs. *CoRR* abs/2601.13655 (2026). arXiv:2601.13655 doi:10.48550/ARXIV.2601.13655

[53] Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. 2025. AgenTracer: Who Is Inducing Failure in the LLM Agentic Systems? *CoRR* abs/2509.03312 (2025). arXiv:2509.03312 doi:10.48550/ARXIV.2509.03312

[54] Ruixin Zhang, Wuyang Dai, Hung Viet Pham, Gias Uddin, Jinqiu Yang, and Song Wang. 2026. Engineering Pitfalls in AI Coding Tools: An Empirical Study of Bugs in Claude Code, Codex, and Gemini CLI. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering, FSE Companion 2026, Concordia University, Montreal, QC, Canada, July 5-9, 2026*. ACM, 393–403. doi:10.1145/3803437.3805213

[55] Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025. Which Agent Causes Task Failures and When? On Automated Failure Attribution of LLM Multi-Agent Systems. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net. https://openreview.net/forum?id=GazlTYxZss

[56] Chenyang Zhu, Spencer Hong, Jingyu Wu, Kushal Chawla, Yuhui Tang, Youbing Yin, Nathan Wolfe, Erin Babinsky, and Daben Liu. 2026. RAFFLES: Reasoning-based Attribution of Faults for LLM Systems. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2026 - Volume 1: Long Papers, Rabat, Morocco, March 24-29, 2026*, Vera Demberg, Kentaro Inui, and Lluís Marquez (Eds.). Association for Computational Linguistics, 7659–7688. doi:10.18653/V1/2026.EACL-LONG.359

[57] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. 2025. Where LLM Agents Fail and How They can Learn From Failures. *CoRR* abs/2509.25370 (2025). arXiv:2509.25370 doi:10.48550/ARXIV.2509.25370

[58] Xinxue Zhu, Jiacong Wu, Xiaoyu Zhang, Tianlin Li, Yanzhou Mu, Juan Zhai, Chao Shen, Chunrong Fang, and Yang Liu. 2026. An Empirical Study of Bugs in Modern LLM Agent Frameworks. *CoRR* abs/2602.21806 (2026). arXiv:2602.21806 doi:10.48550/ARXIV.2602.21806

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