ChaosLLM: A Dependability Testing Approach for Tool-calling Agents

Antonio Ken IANNILLO
Interdisciplinary Center of Security, Reliability, and Trust (SnT)
University of Luxembourg
email: antonioken.iannillo@uni.lu

***Abstract*—Large Language Models (LLMs) are increasingly wrapped in *agents* that orchestrate external tools, memories, and planning components to perform mission-critical functions. Despite growing interest in security testing of LLMs, the dependability of such agents remains largely unexplored. Existing robustness evaluations focus on *prompt-level* attacks and overlook non-adversarial *system-level* faults that routinely arise in real deployments (e.g., tool crashes or timeouts). We present ChaosLLM, a lightweight fault injection framework that sits between an LLM agent and its environment, allowing researchers to emulate realistic failures and quantify their impact on task success and recovery behavior. This paper focuses on tool-calling capabilities as a starting point for a broader investigation. We detail our experimental design, define the dependability metrics, and run the experiments on a LangChain ReAct agent. Our preliminary results motivate the need for further research efforts. Finally, we discuss how ChaosLLM can be extended in future work.**

***Index Terms*—Large Language Models, Fault Injection, Dependability, Chaos Engineering, Software Reliability**

I. Introduction

LLM-powered agents have begun to assist developers in code generation, issue triage, and architectural planning [1]–[3]. In contrast to single-shot chatbots, these agents coordinate *multiple* steps, *external* APIs (e.g., web services, repositories, search engines) and *persistent* memories. A silent malfunction in any of these components can lead to incorrect or confusing answers, yet the research community currently lacks tools and benchmarks to expose such vulnerabilities systematically.

Inspired by classical chaos engineering for microservices [4] and fault injection methodologies [5], [6], we introduce *ChaosLLM* - a framework that deliberately perturbs the operational workflow of LLM agents and records how the agents behave under stress. This short paper focuses on the tool calling capabilities and makes three contributions:

- A taxonomy of common, nonadversarial system-level faults relevant to LLM agents;
- A middleware-based injector that requires no change to the source code of agent or tools;
- An experimental protocol and preliminary results for five simple scenarios on a LangChain ReAct agent.

More complex and complete workloads involving specific software engineering tools, memory, multiagent, and MCP instrumentation fall outside the scope of this first study.

Parts of the manuscript were generated or assisted using OpenAI's GPT-4 language model. The content was reviewed and edited by the authors to ensure accuracy, originality, and adherence to academic standards.

II. Background and Fault Model

LLM agents augmented with tools such as LangChain's ReAct, AutoGen's collaborative agents, and GitHub Copilot Agents delegate subtasks (e.g., arithmetic, code execution, or retrieval) to external components. Although misprompting remains a concern, daily reliability depends on whether these components behave correctly and on the agent's ability to detect and recover from faults.

In a tool-calling agent, an LLM is repeatedly called in a loop. At each step, the agent decides which tools to call and what the inputs to those tools should be. Then those tools are run, and the outputs are fed back into the LLM as observations. The loop terminates when the agent decides that it has enough information to solve the user request, and it is not worth calling any more tools.

Based on the previous literature on fault injection and recent LLM studies (cfr. Section VI), we consider four generic failure modes. They are:

1) **Unreachable**: the tool cannot be contacted and immediately rejects the request.
2) **Slow Response**: the tool responds correctly, but only after a significant, unexpected delay.
3) **Non-Responsive (Hang)**: the tool does not respond at all, leaving the request indefinitely pending.
4) **Incorrect Response**: it returns a result syntactically plausible but semantically incorrect or subtly flawed.

These classes map to well-known API failure modes in distributed systems. We do not claim completeness; our focus is on the granularity of the tool call interface. Extending the taxonomy and parameterization is left to future work.

III. ChaosLLM Framework

A. Architecture Overview

ChaosLLM fits between the LLM (reasoner) and every external tool. The agent's own control loop, prompts, and memory stores are left untouched. Only the call site that connects the LLM's Action to the tool's run() method is wrapped in a thin interception layer. Because the hook is a simple decorator design pattern, it works with vanilla LangChain and does not require modifications to the agent or the tools. The engine supports four distinct classes of tool-level failures defined in Section II, each encoded as a *fault effect* that the injector can impose on an otherwise correct call.

First, we have the **unreachable** condition. In this scenario, the injector simulates an immediate refusal of the call, for example, because of a missing library or network error. The moment the agent asks to invoke a tool, ChaosLLM raises a bespoke Python exception that short-circuits the normal execution path. From the agent's point of view, the tool simply never existed or went offline before the call.

The second failure is **slow response**. The tool does eventually reply, but only after an abnormally long pause. ChaosLLM implements this by blocking the thread for a configurable duration. Nothing else is altered, but the wall-clock delay forces the agent to cope with latency spikes that can cascade into timeout logic higher up the stack.

Third, comes the **non-responsive (hang)** failure. A hang offers no answer at all. The injector simply never returns control. Therefore, any fault-tolerant mechanism must come from the agent's own timeout watchdog.

Finally, ChaosLLM supports **incorrect response** faults. These are the subtlest to spot because everything appears normal at the protocol level: the tool responds promptly and with syntactically well-formed data, yet the semantics are corrupted. The injector achieves this by passing the genuine output through a user-defined transformer that perturbs it. By keeping the structure intact, we test the agent's self-validation capacity rather than its ability to detect crashes. Two perturbators keep the same data type. A *delta perturbator* introduces a plausible but wrong answer, small enough that an incautious agent might trust it. The numerical outputs are nudged by a random offset. Non-numeric outputs undergo random microedits (single-character insert, delete, replace, or swap). A *incorrect perturbator* creates an obviously garbled answer that still looks like a legitimate return value. By offering both a *delta* (subtle drift) and an *incorrect* (total scramble) variant, ChaosLLM can test whether an agent possesses (1) fine-grained sanity checks to reject almost correct but wrong values and (2) coarse-grained filters to catch obviously invalid values.

B. Test Automation

All experiments are driven by a plain text campaign file that is parsed by a launcher script. Each nonempty line encodes one experiment. Each run is executed in a fresh Python subprocess to avoid cross-run side effects. We fix the LLM generation parameters (e.g., temperature) and propagate a fixed seed to every package that supports seeding. Because the task set is small, final labels were verified by a human to avoid overfitting task-specific heuristics. If the baseline and the faulty run reproduce the semantically same result, the run is classified as `CORRECT`. Otherwise, the run is classified as:

- `SILENT-DIFF` (when the answers differ),
- `TIMEOUT` (when the run exceed the time budget), or
- `CRASH` (when the agent crashes due to an uncaught exception), or
- `ERROR` (when the agent provides an error message)

IV. Experimental Design

In this short paper, we focus on five single-step decision problems similar to those that are frequently showcased in open-source agent demos. They are intentionally simple, yet they exercise (i) arithmetic reasoning, (ii) tool orchestration, and (iii) temporal knowledge. Each workload is implemented as one LangChain 'task function' that feeds an identical prompt to the ReAct agent. The agent is equipped with the minimal set of tools required to solve the problem (Table I).

For each pair (task, fault), we run 3 independent trials with random seeds controlling the occurrence of the fault. Fault-free baselines use the identical prompt set. We run 3 different campaigns for three models provided by OpenAI:

- gpt-4: The flagship text model with the largest parameter count that is generally accessible, highest quality on public leader boards
- gpt-4-mini: A much smaller and less expensive derivative of GPT-4 produced by the same training recipe (the same tokenizer and the same instruction-following objective) but with a lower parameter budget. It is mainly used for latency- or cost-sensitive applications.
- o3: A model trained with a different pipeline from the GPT-4 family (distinct optimizer hyperparameters, chain-of-though, mixture-of-experts routing, and additional reinforcement learning from human-feedback stages). It roughly has the same GPT-4-class accuracy on many tasks, but with noticeably different error modes and temperature sensitivity.

V. Results

The experimental campaign counted 225 runs (5 tasks, 5 faults, 3 repetitions, 3 models), for which we computed the following dependability metrics:

- **Task Success Rate (TSR)**: fraction of runs that reach a correct final answer;
- **Hallucination Rate (HR)**: responses that contain confident but incorrect claims (operationally defined as number of SILENT-DIFF outcomes over the total number of runs);
- **Timeout Ratio (TR)**: proportion of calls where the agent exceeds the per-run budget we imposed (i.e., 60 seconds).

Table II and Table III compress the campaign results.

The o3 model achieves the highest Task Success Rate (78.7%) and the lowest timeout rate, while gpt-4-mini lags behind both larger models. Surprisingly, gpt-4 and gpt-4-mini share an identical timeout ratio, even though they differ substantially in parameter count; therefore, the difference stems from semantic failures (hallucinations and hard errors) rather than from time-budget violations alone.

Slow faults had virtually no effect: every model answered correctly in the 45 affected runs. The ReAct loop simply waits and proceeds once the tool responds, confirming that the default timeout threshold of 60 seconds is generous for these microbenchmarks.

TABLE I
MICRO-WORKLOAD SUITE USED IN **CHAOSLLM**. EACH ROW IS AN INDEPENDENT SCENARIO GIVEN TO THE LANGCHAIN REACT AGENT, TOGETHER WITH THE TOOLS MADE AVAILABLE TO THE AGENT AND THE GROUND-TRUTH ANSWER RECORDED IN `EXPECTED.JSON`.

ID	**Scenario**	**Natural-language user prompt**	**Tools exposed to the agent**	**Ground-truth answer**
FX	Currency Conversion Math	"How much is 150 € in USD if the exchange rate is 1.1?"	CalculatorTool	165
TIP	Split Bill with Tip	"We spent $120 at a restaurant, added a 15% tip, and we're 4 people. How much does each person pay?"	CalculatorTool	34.5
BMI	Body Mass Index (BMI) Calculation	"What is the BMI for someone who is 70 kg and 1.75 m tall?"	CalculatorTool	22.86
ETA	Travel Distance Estimator	"How long will it take to drive from Los Angeles to San Diego if the average speed is 60 mph?"	SearchTool, CalculatorTool	2 hours
TRIVIA	Trivia Bot	"Who discovered gravity, and how many years ago was that?"	WikipediaTool, DateTimeTool, CalculatorTool	Isaac Newton, 338 years

TABLE II
OUTCOME COUNTS AGGREGATED OVER THE FIVE TASKS (`FX`, `TIP`, `BMI`, `ETA`, `TRIVIA`) AND THREE REPETITIONS.

	CORRECT			**SILENT-DIFF**			**TIMEOUT**			**CRASH**			**ERROR**		
	gpt-4	gpt-4-mini	o3	gpt-4	gpt-4-mini	o3	gpt-4	gpt-4-mini	o3	gpt-4	gpt-4-mini	o3	gpt-4	gpt-4-mini	o3
Unreachable	12	7	12	3	3	3	0	0	0	0	0	0	0	5	0
Slow	15	15	15	0	0	0	0	0	0	0	0	0	0	0	0
Hang	3	3	9	0	0	2	12	12	4	0	0	0	0	0	0
Incorrect	12	12	13	2	3	2	0	0	0	0	0	0	1	0	0
Delta	6	4	10	9	11	4	0	0	1	0	0	0	0	0	0
Total	48	41	59	14	17	11	12	12	5	0	0	0	1	5	0

TABLE III
AGGREGATE OUTCOMES OVER THE FULL CAMPAIGN
(HIGHER **TSR** IS BETTER, LOWER **HR** AND **TO** ARE BETTER).

Model	TSR (%)	HR (%)	TO (%)
gpt-4	64.0	18.7	16.0
gpt-4-mini	54.7	22.7	16.0
o3	78.7	14.7	5.3

Hang faults were the most disruptive. Across models, 52% (12/23) of all noncorrect outcomes are timeouts triggered by this fault. The gpt-4 and gpt-4-mini models time out on 4 of the 5 tasks, while the o3 model manages to recover in 3. A closer inspection of the execution traces reveals that the stronger the backbone, the less it actually relies on tools. In every run that ended with a `CORRECT` label despite an injected hang, the agent never invoked the blocked tool; it solved the task through pure in-context reasoning. Only the TRIVIA task forces every model to call at least the datetime tool, which explains why all three models systematically time out on that task under the hang fault. This observation suggests that adding smarter "tool call only if needed" logic can reduce the surface of denial-of-service style failures, especially for larger LLMs that can often solve simple subtasks without delegation.

For **Unreachable** tools, the agent usually retries with an alternative plan, leading to high pass rates except on the TRIVIA task, where an unreachable datetime tool makes the agent believe it is the year 2023, except for the most recent o3 model. Additionally, the gpt-4-mini model leaks the raw exception string in five runs, hence the larger ERROR count. The blatantly **Incorrect** perturbator is handled well: only one out of 45 runs ends in an ERROR (the model gpt-4 on the TRIVIA task asks the user for help: *"Could you please try asking for the current date and time again?"*). Agents appear to perform a coarse plausibility check and immediately disregard the idea of using the tool if the output is obviously garbled and approach the task only with their own capabilities and knowledge. The subtle **Delta** corruption is the Achilles' heel of every model. It accounts for 42% of all hallucinations (SILENT-DIFF). The agent confidently returns a wrong result with confidence in 26 out of 45 runs, indicating that fine-grained sanity checks are still missing. Although the o3 model tops the aggregate scoreboard, the gap is not uniform across fault types: its advantage comes from its ability to reason without the need for simple tools, as presented in this preliminary work. In contrast, all three models are equally vulnerable to delta perturbations, consistent with the hypothesis that the weakness may lie in the generic prompting pattern of the ReAct tool, and we cannot rule out backbone contributions given our limited scope. Here we highlight three takeaways from this preliminary experimental campaign: (1) *Under our microbenchmarks and a 60s timeout, slow-response faults did not affect outcomes*, but complete hangs require an external watchdog; (2) Agents cope well with overly garbled output, yet *fail silently on plausible but wrong* results; (3) The choice of backbone matters, but *robust prompting and tool-level validation matter more*.

VI. Related Work

LLMs are now routinely evaluated for security and robustness, yet existing efforts concentrate on *prompt–level* attacks rather than on the system context in which agents operate. Prompt fuzzing frameworks, such as PromptFuzz [7] and red-teaming suites like AgentXploit [8] craft adversarial inputs that attempt to elicit policy violations or jailbreak behavior. While invaluable for hardening the natural-language interface, these techniques leave untested the failure modes that stem from the external tooling, memories, and control loops that modern agents rely on. Chaos Monkey [9] and similar approaches [5], [6] deliberately disrupt virtual machines, containers, or network links to check whether cloud services degrade gracefully. However, these approaches operate at infrastructure granularity and are blind to the unique control flow of an LLM agent. In the context of software engineering, research has documented logic bugs [10], unsafe networking patterns [11], and quality regressions when replacing Stack Overflow with model suggestions [12]. Finally, initial attempts at evaluating the quality of agent code have begun to surface [13]. TrustAgent [14], for example, introduces benchmarks that measure factual consistency and fidelity to the role between cooperating agents. However, even these studies assume fault-free access to the underlying tools and services. ChaosLLM represents an initial step toward that goal by injecting realistic tool-level failures and quantifying their effect on agent dependability.

VII. Future Work

The present study should be viewed as a first step toward systematically testing the reliability of LLM tool ecosystems. Several simplifying assumptions helped us keep the prototype small and the campaign affordable, yet they also limit the external validity of the findings. ChaosLLM currently intercepts tool invocations in LangChain. However, *Model Context Protocol (MCP)* has rapidly emerged as the de facto standard for tool calling across closed- and open-source alternatives. MCP adds explicit, strongly typed JSON schemas, function call continuations, and streaming partials. Porting ChaosLLM to the MCP layer would let us inject faults before any framework-specific adaptation occurs. Our five tasks are deliberately simplistic. Real products rely on heterogeneous collections of tools and execute far more complex tasks. Future campaigns should therefore incorporate off-the-shelf services and reproduce real user conversations that include different activities. Currently, all injected failures are *persistent*: If a tool is unreachable or returns a corrupted value, it does so for the whole run. In production, however, most incidents are *transient* (momentary network blips, short overload bursts, flaky third-party APIs). Transient faults interact with an agent's retry policy and self-healing logic in ways that persistent faults cannot expose. Extending ChaosLLM with temporal fault schedules is, therefore, a high-priority item. Today's prototype reasons about a single autonomous agent. However, emerging stacks comprise *swarms* of agents that delegate subtasks to each other through explicit protocols. Although one may *model* consider an *assistant agent* just another tool, this elides other aspects (for example, trust). ChaosLLM should therefore be extended with first-class awareness of agent-to-agent protocols. Addressing these points will transform ChaosLLM from a proof-of-concept into a comprehensive dependability tool for the next generation of tool-using and multiagent LLM applications.

References

[1] K. El Haji, C. Brandt, and A. Zaidman, "Using github copilot for test generation in python: An empirical study," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, 2024, pp. 45–55.

[2] M. Hu, P. Zhao, C. Xu, Q. Sun, J.-G. Lou, Q. Lin, P. Luo, and S. Rajmohan, "Agentgen: Enhancing planning abilities for large language model based agent via environment and task generation," in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, 2025, pp. 496–507.

[3] F. Altiero, D. Cotroneo, R. De Luca, and P. Liguori, "Securing ai code generation through automated pattern-based patching," in *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2025, pp. 282–289.

[4] A. Al-Said Ahmad, L. F. Al-Qora'n, and A. Zayed, "Exploring the impact of chaos engineering with various user loads on cloud native applications: an exploratory empirical study," *Computing*, vol. 106, no. 7, pp. 2389–2425, 2024.

[5] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–55, 2016.

[6] G. Yu, G. Tan, H. Huang, Z. Zhang, P. Chen, R. Natella, Z. Zheng, and M. R. Lyu, "A survey on failure analysis and fault injection in ai systems," *ACM Transactions on Software Engineering and Methodology*, 2024.

[7] J. Yu, Y. Shao, H. Miao, and J. Shi, "Promptfuzz: Harnessing fuzzing techniques for robust testing of prompt injection in llms," *arXiv preprint arXiv:2409.14729*, 2024.

[8] Z. Wang, V. Siu, Z. Ye, T. Shi, Y. Nie, X. Zhao, C. Wang, W. Guo, and D. Song, "Agentxploit: End-to-end redteaming of black-box ai agents," *arXiv e-prints*, pp. arXiv–2505, 2025.

[9] M. A. Chang, B. Tschaen, T. Benson, and L. Vanbever, "Chaos monkey: Increasing sdn reliability through systematic network destruction," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 371–372.

[10] T. Sharma, "Llms for code: The potential, prospects, and problems," in *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2024, pp. 373–374.

[11] M. Dunne, K. Schram, and S. Fischmeister, "Weaknesses in llm-generated code for embedded systems networking," in *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2024, pp. 250–261.

[12] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 38, no. 19, 2024, pp. 21 841–21 849.

[13] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of github copilot's code generation," in *Proceedings of the 18th international conference on predictive models and data analytics in software engineering*, 2022, pp. 62–71.

[14] W. Hua, X. Yang, M. Jin, Z. Li, W. Cheng, R. Tang, and Y. Zhang, "Trustagent: Towards safe and trustworthy llm-based agents through agent constitution," in *Trustworthy Multi-modal Foundation Models and AI Agents (TiFA)*, 2024.