Beyond Fault Injection: Leveraging LLMs for Autonomous Chaos Engineering

Gerard A. Matthew
University of Illinois Urbana-Champaign
gerardm3@illinois.edu

P. Brighten Godfrey
University of Illinois Urbana-Champaign
pbg@illinois.edu

Abstract

Distributed systems fail in unique and unpredictable ways, leading operators to adopt chaos engineering practices for understanding system resilience. Standard practice today is to use automated systems for fault execution, while relying on humans for hypothesis generation (that is, determining what potential faults should be tested). We build and experiment with a prototype that addresses this gap with a multi-agent system that incorporates architectural context to autonomously generate and validate resiliency hypotheses. Using a 24-component microservice application, our prototype generated its own hypotheses and validated four of five sampled hypotheses. Based on these results, we identify open research challenges toward fully autonomous chaos engineering.

1 Introduction

Failures in distributed systems often hide in the interaction between design, implementation and configuration choices spread across components. An experienced site reliability engineer reviewing a microservice deployment may check for single points of failure, verify resource limits, and understand how the system handles varying load conditions. They are less likely to think through complex failure scenarios such as a two-minute gRPC keepalive timeout combined with a cached Consul resolver which creates a two-minute blind spot after a service restart. This type of failure manifests in production with reliability implications.

Chaos engineering [4] addresses this by intentionally injecting faults into production or staging systems and observing whether the system maintains acceptable behavior by following a hypothesis-driven cycle. Tools like LitmusChaos [14] and Chaos Mesh [3] enable automated fault injection in Kubernetes. A key question, however, is what faults to test. Testing the Cartesian product of every component and every fault type involves a huge number of tests, and if we expand to exercising cross-component interactions, exhaustive tests are infeasible. Generally, therefore, humans select the scope of tests.

Recent works have begun to apply LLMs to find faults. Existing LLM-based test generation methods [11, 2] focus on code coverage, but do not deal with operational issues that arise when running complex distributed software. Some recent work has studied root cause analysis [1, 15, 5] but this assumes knowledge of an existing fault, as opposed to discovering what faults might arise. Most relevant to our work, ChaosEater [10] applied LLMs to hypotheses generation based on Kubernetes manifests. But deployment manifests contain only Kubernetes-specific infrastructure-level information and cannot express application behavior.

This highlights what we see as the most important challenge: operational faults in distributed applications do not come from only a narrow, structured set of problems. They can arise from application-specific considerations, hidden dependencies, and mismatched assumptions between interacting distributed components. Comprehensive chaos engineering thus involves what we could call "open world" hypothesis generation.

ACM Conference on AI and Agentic Systems (CAIS 2026). Workshop: AI Agents for Discovery in the Wild.

We investigate whether we can autonomously perform such open-world, system-wide failure reasoning at and above the level of an experienced site reliability engineer. We hypothesize that LLMs processing the full system intermediate representation can surface failures which require cross-component reasoning and predict operational failures.

To realize this, we envision a suite of specialized agents [9, 12, 16] that takes application source code as input and produces validated failure verdicts as output. Extraction agents build a graph [6] of the system from source code, capturing service dependencies and component configuration properties, which can be updated on each application modification. A hypothesis agent generates candidate failure scenarios using multi-round LLM generation, with a critic agent for scenario validation. Accepted scenarios are planned, then an execution agent orchestrates fault injection, workload generation, and observation. Confirmed failures use a remediation agent that accesses the code repository, generates a patch in a sandbox environment, and re-validates the hypothesis. To demonstrate this idea, we have prototyped most of the above; but the extraction agents used manually curated metadata, and the remediation agent is future work.

In this paper, we evaluate hypothesis generation quality through an input ablation and end-to-end execution through live Kubernetes cluster validation. Our preliminary experiments showed very promising results: with appropriate temperature parameter, the system was able to generate many useful, novel hypotheses, outside the scope of functionality of a widely used chaos testing tool (Litmus). Our experience using the system also revealed some of the challenges of applying agents in an "open-world" space. First, although generating novel types of hypotheses was our goal, this also means that the system needs to generate new test execution procedures, which is a challenge especially with respect to safety if we intend to use the system in a production environment. Second, because it is easy for the system to generate hypotheses, it produced many valid but uninteresting failure scenarios that did not expose new risks. In traditional code testing, there are well-defined notions of important faults (e.g., crash) and result quality (e.g., code coverage); in our broader setting, how to define what is "interesting", and to know whether we have found everything "interesting", are open questions.

The remainder of this paper presents the system design (§3), an input ablation showing deployment manifests alone does produce appropriate architectural-level failure scenarios, while early validation (§4) confirmed 4 of 5 hypotheses on a Kubernetes cluster running DeathstarBench Hotel Reservation [7]. These results provide evidence that autonomous chaos engineering is achievable on this class of application, though several open problems remain before such a system is practical.

2 Motivation

Consider a microservice that loads data from a database into an in-memory index at startup without a refresh mechanism. Traditional chaos engineering approach will attempt to kill the database pod, inject network latency, or stress the CPU. While these are useful tests, they target how the system responds when a running component degrades. These tools do not test whether a rolling restart, combined with a data mutation during the rollout, causes replicas to serve different results. These types of hypotheses require knowledge of how the service is written, what it depends on, and how those dependencies behave under operational conditions which are not observed from only the deployment manifest.

Deployment manifests describe how an application is deployed, including replica counts, container images and resource limits. However, they do not describe how the application behaves, and prior work has shown that software configuration [13] and design choices are a dominant source of cloud failures [8] whose complexity grows as systems evolve [17]. These behavioral properties determine how the system fails under operational events like restarts, scaling, and partial outages. We use the term architectural metadata to refer to behavioral properties extracted from application source code that describe how each component works, not just how it is deployed. These includes data access patterns, concurrency models, protocol configurations and error handling behavior.

We define a hypothesis as architecturally non-obvious if it requires connecting metadata fields across components to propose a failure that testing each component with each fault type individually would not surface. These hypotheses can span multiple testing disciplines, while existing chaos engineering tools inject faults and observe degradation. The architectural failure space is broader, therefore surfacing them requires reasoning which crosses testing disciplines.

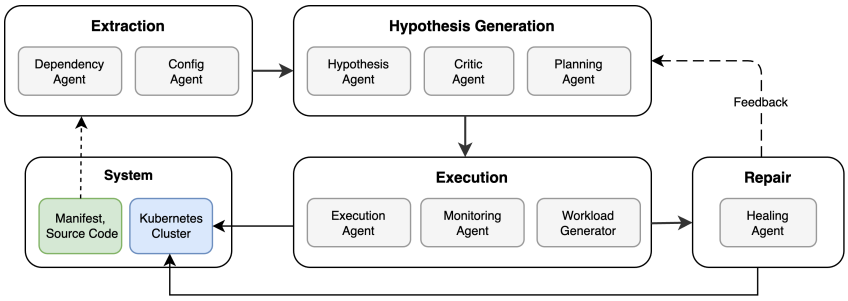

Figure 1: Four agent stage pipeline from system configuration into executable hypothesis

Table 1: Hypotheses generated from architectural metadata and validated on a Kubernetes cluster.

ID	Generated Hypothesis	Verdict	Key Evidence
H1	MongoDB replicas without replica set will silently split writes across pods	**Confirmed**	5 records on 1 pod, 0 on 2 others
H2	After Consul failure, stale cached addresses will prevent connection recovery for up to 120 seconds	**Confirmed**	Error rate rose from 29% to 52% to 100%; no recovery
H3	Absence of request timeouts and connection limits will cause goroutine accumulation under sustained load	Not Validated*	Latency rose from 101 to 244 ms post-recovery
H4	Check then insert pattern without a distributed lock allows concurrent reservations to double book	**Confirmed**	30/30 concurrent writes of the same hotel room on the same day succeeded
H5	In-memory data loaded at startup with no refresh will diverge permanently after rolling restart	**Confirmed**	Queries returned $\{0, 27\}$ across pods

*Partial signal at 84 RPS. The hypothesis predicted out-of-memory failure at 5,000 RPS.

3 System Design

The system takes as input application source code and infrastructure manifests that describes the components, dependencies, and respective metadata. The pipeline consists of four stages, each implemented as one or more specialized agents, connected by a feedback loop (Figure 1).

Extraction agents. These agents infer relationships from source between components that are not explicitly declared, extract insights using techniques from static analysis and produce hypothesis that feed into the next pipeline stage. They build a graph [6] of the system that updates incrementally with each code commit. In this evaluation, the structured representation was manually curated.

Hypothesis agent with critic. The hypothesis agent generates failure scenarios using multiple LLM rounds and reduces duplication by passing output as input into subsequent rounds. A critic agent reviews each scenario to ensure a generated hypothesis does not contain hallucinated output. A planning agent converts scenarios into executable experiments by resolving scenario intent against the system API definitions.

Execution and healing agents. Before each experiment, the execution agent collects baseline metrics prior to fault injection. During the experiment, a monitoring agent collects metrics to detect anomalies. At the end of each experiment, a verdict is produced based on the observed data during the experiment. The system restores the application to a known-good state to continue with experiments. This is implemented and used in our evaluation. In the future, self-healing agents will generate, apply remediation patches and validate fixes to failed experiments.

4 Evaluation

We evaluate components of our design on DeathstarBench Hotel Reservation [7], deployed in a three-node (EC2, 8 vCPUs, 32 GiB) Kubernetes cluster. Our prototype used architectural metadata

manually created from source code. Using Claude Sonnet 4.5, it generated hypotheses and resolved them into executable experiments which ran against the cluster using pipeline described in Section 3.

4.1 Hypothesis Validation

We explored whether architectural reasoning could produce failure scenarios that require cross-component analysis. We confirmed four out of five sampled scenarios without human intervention (Table 1). H1 identified that MongoDB pods deployed without replica set configuration would silently split data. In H2, the hypothesis agent connected a 120-second gRPC keepalive timeout with Consul's cached address resolution to predict a two-minute recovery blind spot after a Consul pod failure. The experiment ran three times with up to 100% failure without recovery. H4 was confirmed by sending 10 concurrent reservation requests, all of which succeeded. H3 produced a partial signal as insufficient load could be generated from our test setup.

4.2 Input Ablation

Each confirmed hypothesis depends on architectural metadata absent from Kubernetes deployment manifests (e.g., replication configuration, timeout behavior, locking patterns). To test whether manifests alone could produce equivalent hypotheses, we ran 10 rounds of generation with only Kubernetes manifests as input. The LLM produced 122 unique scenarios, all targeting infrastructure-level faults. With architectural metadata alone, the system produces 24 seeds, 21 referencing source-code related properties. With both input types, it produces 33 seeds spanning infrastructure and source-code related hypotheses.

5 Open Research Opportunities

Metadata extraction. Our system intermediate representation was manually selected; however, determining which source-code properties are worth extracting remains an open question. Larger systems may exceed the LLM's context window and would require strategies to preserve architectural context at scale.

Hypothesis selection and prioritization. The system generates all hypotheses in a single pass and was limited to execute a selected subset. Practical use requires an approach to rank hypotheses by resiliency risk while balancing diverse failure modes. However, risk alone is insufficient for prioritization as the system needs to distinguish actionable hypothesis from those that are acceptable design tradeoffs. The definition of what coverage means for architectural failure hypotheses, as opposed to code branches, is an open research question.

Execution safety. Autonomous fault injection risks introducing failures beyond the experiment scope, potential data loss from destructive operations, and outages if the system restoration fails. Our evaluation ran on a test cluster, however, extending to a production environment would require guaranteed safety guardrails with human approval for destructive operations.

New fault type generation. The hypothesis agent generates scenarios with fault types that do not map to existing fault primitives. There is the opportunity to evaluate the generation of new composable fault primitives which can be executed safely with human approval.

6 Conclusion

LLM reasoning over architectural metadata surfaces failure conditions that deployment manifest approaches do not produce. Our results show the feasibility of autonomous end-to-end chaos engineering extending beyond infrastructure level metadata. We hope this work motivates further investigation into which architectural metadata are most valuable for autonomous chaos engineering, how to prioritize generated hypotheses, and what execution safety guarantees are needed.

References

[1] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In Proceedings of the 45th International Conference on Software Engineering, ICSE '23, page 1737–1749. IEEE Press, 2023.

[2] Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Effective high coverage test generation for python. Proc. ACM Softw. Eng., 2(FSE), June 2025.

[3] Chaos Mesh Authors. Chaos mesh: A powerful chaos engineering platform for kubernetes.

[4] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos Engineering . IEEE Software, 33(03):35–41, May 2016.

[5] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xue-Chao Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo GHOSH, Xuchao Zhang, Qingwei Lin , Saravan Rajmohan, and Dongmei Zhang. Automatic root cause analysis via large language models for cloud incidents. In EuroSys'24, April 2024.

[6] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization, 2025.

[7] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16, page 1–16, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: a survey of progress and challenges. In Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI '24, 2024.

[10] Daisuke Kikuta, Hiroki Ikeuchi, and Kengo Tajiri. Chaoseater: Fully automating chaos engineering with large language models, 2025.

[11] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pages 919–931, 2023.

[12] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: communicative agents for "mind" exploration of large language model society. In Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[13] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. Large language models as configuration validators. In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering, ICSE '25, page 1704–1716. IEEE Press, 2025.

[14] LitmusChaos Contributors. Litmuschaos. `https://litmuschaos.io`, 2026.

[15] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring llm-based agents for root cause analysis. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, page 208–219, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang (Eric) Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. In COLM 2024, August 2024.

[17] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. An evolutionary study of configuration design and implementation in cloud systems. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 188–200, 2021.