EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Chaos Engineering in Production: Designing Fault-Injection Experiments with Controlled Blast Radius and Automated Rollback

Navin Chandran

Senior Director, Cognizant Technology Solutions, USA

Article information

Received: 13th February 2026
Received in revised form: 20th March 2026
Accepted: 24th April 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20655900

Abstract

Distributed systems fail in combinations no designer anticipated, and the failures that matter most are precisely the ones that integration tests never exercise: a dependency that slows rather than stops, a region that partitions, a retry storm that turns a small fault into an outage. Chaos engineering confronts this directly by injecting controlled failures into running systems, including production, to discover weaknesses before an uncontrolled failure does. The discipline is often misread as breaking things at random. It is the opposite: a scientific method applied to system resilience, in which an engineer states a hypothesis about steady-state behaviour, injects a specific fault, and measures whether the system holds up. This paper presents chaos engineering as an experimental practice and concentrates on the two mechanisms that make it safe enough to run on live traffic. Blast-radius control limits the fraction of users and infrastructure an experiment can affect, beginning with a single host and escalating only as confidence grows. Automated rollback continuously compares the live system against its expected behaviour and aborts the experiment the instant the deviation exceeds a threshold, bounding the worst case. We trace the practice from its origin in the Netflix Simian Army to today's platforms, describe how to design an experiment, and argue that deliberately injecting failure is now a core technique of reliability engineering rather than a stunt.

Keywords:- Chaos Engineering, Fault Injection, Resilience, Site Reliability Engineering, Blast Radius, Automated Rollback, Distributed Systems, Steady-State Hypothesis, Production Testing.

.

I. INTRODUCTION

A large distributed system is never fully healthy. At any moment some disk is failing, some network link is congested, some dependency is responding slowly, and the system is expected to absorb all of it without the user noticing [1]. The hard truth of operating such systems is that their resilience to these conditions is largely untested, because the conditions are difficult to reproduce in a test environment and emerge only from the interaction of components under real load [1], [4]. A system can pass every unit and integration test and still collapse the first time a single dependency degrades in production.

Chaos engineering was developed at Netflix to close this gap [1], [3]. Its premise is that the only reliable way to learn how a system behaves under failure is to make it fail, deliberately and under observation, while it is doing real work. The most famous early tool, Chaos Monkey, did exactly this by randomly terminating production instances during business hours, forcing engineers to build services that tolerate the loss of any single node as a routine event rather than a crisis [3], [15]. The practice has since matured from a single tool into a formal discipline

with stated principles, supported by commercial and open-source platforms and adopted well beyond its origin [1], [2].

The persistent misunderstanding is that chaos engineering means causing damage at random. It is better understood as the experimental method applied to operations, and its credibility rests on being safe to run where it matters most, in production [1]. This paper develops that view. We frame the practice as hypothesis-driven experimentation, then examine in detail the two safety mechanisms that make production experiments responsible, controlled blast radius and automated rollback, before discussing the platforms that implement them and the organisational conditions the practice requires.

II. CHAOS ENGINEERING AS EXPERIMENT

A. The Steady-State Hypothesis

A chaos experiment begins not with a fault but with a hypothesis, and this is what separates it from merely breaking things [1]. The engineer first defines the system's steady state in terms of a measurable output that indicates normal behaviour, such as the rate of successful requests, the latency distribution, or a business metric like streams started per second. The hypothesis is that this steady state will persist even when a specific fault is introduced. The experiment then injects that fault and tests the hypothesis against reality. If the steady state holds, the system has demonstrated resilience to that condition; if it does not, the experiment has found a weakness cheaply and on the team's terms rather than during an unplanned outage [1], [2]. Figure 1 shows this loop.

B. Realistic Faults, Real Environment

The faults injected are chosen to mirror events that actually occur in production: instances terminating, latency injected into a dependency, errors returned from a service, a network partition between zones, or resource exhaustion on a host [1], [14]. The principle that distinguishes mature chaos engineering is that these are best exercised in the production environment, because staging never faithfully reproduces production's scale, traffic patterns, and configuration, and a resilience result that holds only in staging proves little, which is why practitioners make the explicit case for testing resilience in production itself [1], [4], [5]. Running in production is also what makes the safety mechanisms indispensable, which is the subject of the next two sections. Research has pushed this further toward principled fault selection, using the system's own data-flow lineage to choose the failures most likely to expose a real weakness rather than testing combinations blindly [6], [7].

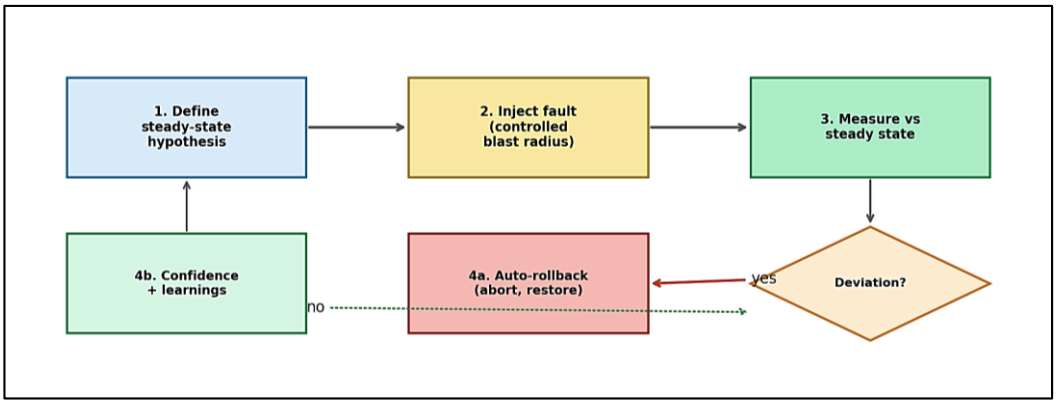

Fig 1: The chaos experiment loop. A steady-state hypothesis is tested by injecting a fault within a controlled blast radius; if measured behaviour deviates beyond threshold, the experiment is automatically rolled back, otherwise it yields confidence and learnings.

III. CONTROLLED BLAST RADIUS

A. Bounding Who Can Be Affected

The first safety mechanism is to limit the scope of any experiment, its blast radius, which is the set of users and infrastructure that a fault can possibly affect [2]. A well-designed experiment never begins at full scale. It starts with the smallest scope that can still produce a meaningful signal, a single host or a tiny fraction of traffic routed to a canary, and confirms the hypothesis there before any expansion [2], [10]. This containment is what makes it acceptable to experiment in production at all: even if the hypothesis is wrong and the system fails, the damage is confined to a deliberately small population, and the experiment has still surfaced the weakness.

B. Escalating With Confidence

Once an experiment passes at small scale, the blast radius is widened in stages, from one host to a single availability zone, to a region, and only eventually to the full system, with the hypothesis re-tested at each level

[2]. Figure 2 illustrates this escalation. The discipline is to increase exposure only as confidence accumulates, so that the experiments most likely to cause harm are run at the scale least able to do so. Targeting controls, routing the fault to specific users, devices, or request classes, let an experiment hold its blast radius precisely while still exercising a realistic slice of production traffic [2], [11]. The combination of small starting scope and gradual, evidence-gated escalation is what turns production fault injection from recklessness into engineering.

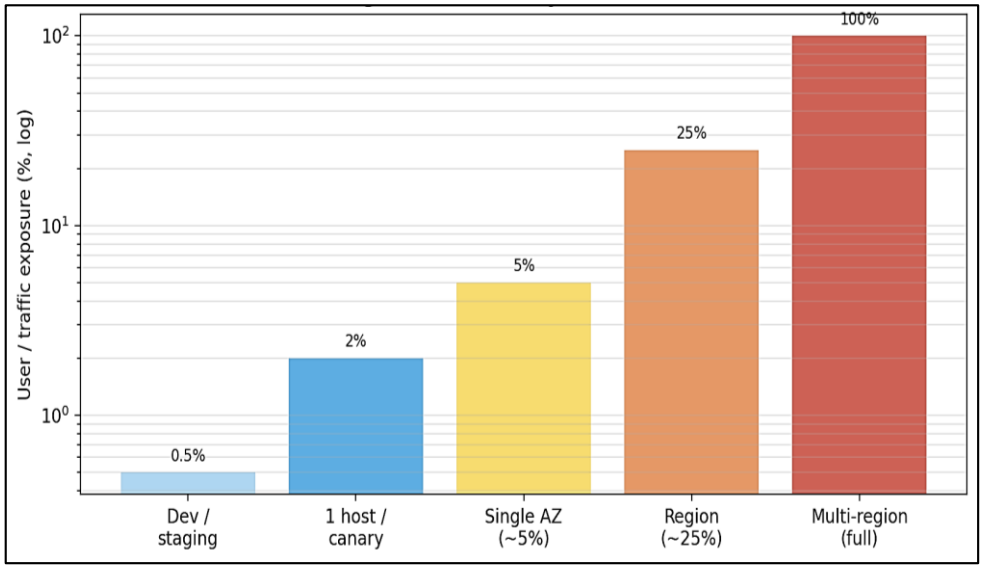

Fig 2: Blast radius is escalated only as each stage confirms the steady-state hypothesis, so the riskiest experiments run at the smallest scale and exposure grows with confidence.

IV. AUTOMATED ROLLBACK

Limiting blast radius bounds how many users an experiment can reach; automated rollback bounds how long and how badly it can affect them. The mechanism continuously monitors the steady-state metrics during the experiment and compares the population exposed to the fault against an unaffected control [1], [2]. The moment the deviation crosses a predefined threshold, the harm to the experimental group exceeding what the team has decided is acceptable, the system automatically aborts: it stops injecting the fault and restores normal conditions, without waiting for a human to notice and react [2], [14]. This automatic kill switch is essential because the value of chaos engineering depends on running real faults against real users, and no responsible team would accept that risk if the worst case were unbounded. With automated rollback the worst case is bounded by the detection threshold and the speed of the abort, both of which are engineered in advance rather than left to on-call reflexes. Together with blast-radius control, it transforms the potential cost of an experiment from open-ended to small and known, which is the precondition for experimenting in production responsibly [1], [2].

Table 1. The Two Safety Mechanisms of Production Chaos Experiments

Mechanism	Bounds	How	Failure if absent
Controlled blast radius	How many can be affected	Start tiny; escalate on confidence	A wrong hypothesis hits everyone
Automated rollback	How long / how badly	Threshold on steady-state deviation	A bad experiment runs until noticed
Steady-state hypothesis	What counts as failure	Measurable normal-behaviour metric	No criterion to abort against
Control group	Attribution of impact	Compare exposed vs unexposed	Cannot tell fault from noise

V. PLATFORMS AND TOOLING

The practice has been productised. Netflix generalised Chaos Monkey into a broader Simian Army and later into platforms that automate experiment design and execution at scale, including the data-driven selection of which failures to test [3], [7]. Commercial offerings package fault injection, blast-radius controls, and automated halting behind a managed interface, lowering the barrier for teams without Netflix-scale engineering [11]. In the cloud-native ecosystem, open-source projects integrate chaos experiments directly into container orchestration, expressing faults and their scope declaratively alongside the rest of a system's configuration [12]. What these tools share is an implementation of the same two safety mechanisms: every credible platform provides a way to bound

the blast radius and a way to halt automatically when steady-state metrics degrade [2], [11], [12]. The tooling, in other words, encodes the discipline rather than replacing it.

VI. THE ORGANISATIONAL DIMENSION

Chaos engineering is as much a cultural practice as a technical one, and it fails without organisational support. Deliberately injecting failure into production requires a blameless culture in which the weaknesses an experiment uncovers are treated as learning rather than fault, and it requires investment in the observability that makes steady state measurable and deviation detectable in the first place [4], [10]. The underlying philosophy is that of antifragility, the idea that a system subjected to controlled stress becomes stronger because it is continually forced to remove its weaknesses, rather than accumulating hidden fragility until a large uncontrolled failure exposes it all at once [4], [13]. Reliability engineering frameworks treat this proactive failure testing as a complement to error budgets and service-level objectives, a way to spend a controlled amount of reliability to buy confidence [10]. The practice connects, finally, to the longer tradition of dependable-systems research, which has always argued that fault tolerance must be validated by fault injection rather than assumed from design [8], [9], [16].

VII. A TAXONOMY OF FAULTS TO INJECT

Designing experiments requires a vocabulary of failures, and mature practice organises the faults worth injecting into a small number of domains, each exercising a different resilience property. Infrastructure faults remove compute: terminating an instance, draining a node, or simulating the loss of an entire availability zone, which tests whether the system tolerates the disappearance of capacity it was assumed to have [3], [14]. Network faults are often the most revealing, because the network is far less reliable than designers assume; injecting latency, packet loss, or a partition between zones probes the timeout, retry, and failover logic that only ever runs under exactly these conditions [17]. Resource faults exhaust CPU, memory, disk, or I/O on a host to test graceful degradation under pressure rather than outright failure. Application faults make a service return errors or respond slowly, and dependency faults degrade a downstream service, which together test the calling system's circuit breakers, fallbacks, and bulkheads, the very mechanisms meant to stop one component's trouble from becoming everyone's [9]. Figure 3 arranges these domains, and Table 2 pairs each with the resilience it is designed to validate. The discipline of the taxonomy is to ensure coverage: a resilience claim is only as trustworthy as the breadth of fault domains it has actually survived, and a program that only ever kills instances has tested just one corner of the space [6], [7].

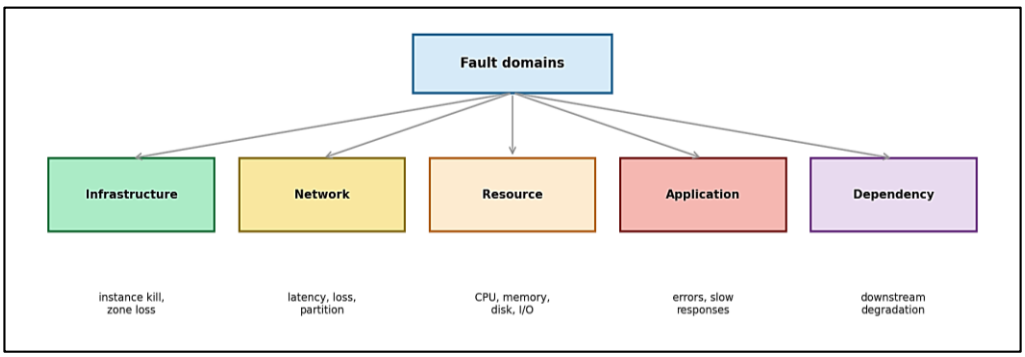

Fig 3: Faults worth injecting fall into a few domains, each validating a distinct resilience property. Broad coverage across domains is what makes a resilience claim credible; testing only one domain proves only one thing.

VIII. FROM GAMEDAYS TO CONTINUOUS CHAOS

Chaos engineering is usually adopted in stages, and understanding the progression helps a team place itself and plan its next step. The entry point is the GameDay, a scheduled, supervised exercise in which engineers gather to run a planned experiment against a system, watch how it and they respond, and capture the learnings, including gaps in tooling, runbooks, and alerting that the exercise exposes [2], [5]. GameDays are valuable precisely because they rehearse the human response alongside the technical one, but they are episodic, and a system changes continuously between them. The next stage automates individual experiments so they can run on demand with the safety mechanisms enforced programmatically, and the mature stage integrates them into the delivery pipeline as continuous verification, so that resilience is re-tested automatically as the system evolves rather than validated once and assumed to persist [2], [18]. Expressing experiments declaratively, as chaos as code checked into version control alongside the system, makes them repeatable, reviewable, and safe to run unattended within their configured blast radius [12], [19]. The trajectory mirrors the broader shift in operations from manual, periodic

checks toward automated, continuous assurance, and it is what lets resilience keep pace with the rate of change in a system that is never finished [10].

IX. MEASURING RESILIENCE

An experiment is only as good as the measurement that decides its outcome, and this is where chaos engineering depends utterly on observability. The steady-state hypothesis must be expressed as a metric the system actually emits, and detecting deviation in time to roll back requires that metric to be available with low latency and compared against an unaffected control to separate the fault's effect from normal variation [1], [2]. Beyond the per-experiment verdict, organisations track aggregate resilience indicators such as the time to detect and the time to recover from injected failures, using improvements in these as evidence that the practice is making the system more robust over time [10]. The connection to reliability engineering is direct: error budgets quantify how much unreliability is acceptable, and a chaos experiment is a controlled way to spend a small, deliberate portion of that budget to buy confidence, rather than discovering the system's limits through an uncontrolled outage that spends the budget all at once [10]. Care is required in the measurement itself, since tail latency and coordinated effects can mislead a naive metric, and the resilience-engineering tradition cautions that resilience is a property of the whole socio-technical system, the people and processes as well as the software, so the human responses an experiment reveals are as much a part of the result as the graphs [16], [20]. Academic work on systematic resilience testing reinforces the same point, showing that principled, repeatable fault injection finds weaknesses that ad hoc testing misses [21].

Table 2. Fault Domains and the Resilience Each Validates

Fault domain	Example fault	Resilience it tests
Infrastructure	Instance / zone termination	Tolerance of lost capacity, failover
Network	Latency, loss, partition	Timeouts, retries, partition handling
Resource	CPU / memory / disk exhaustion	Graceful degradation under pressure
Application	Injected errors or slow responses	Error handling, fallbacks
Dependency	Downstream service degradation	Circuit breakers, bulkheads

X. CONCLUSION

Chaos engineering reframes failure from something to be feared into something to be scheduled. By stating a steady-state hypothesis, injecting a realistic fault, and measuring the result, it applies the experimental method to the resilience of systems that are otherwise tested only by the accidents they suffer [1]. Its legitimacy as a production practice rests entirely on two mechanisms working together: a controlled blast radius that bounds how many users an experiment can affect and escalates only with confidence, and automated rollback that bounds how long and how severely by aborting the instant behaviour deviates [2]. With those guardrails, the worst case of an experiment is small and known in advance, which is what makes deliberately breaking a live system a responsible engineering act. From a single monkey terminating instances to declarative, data-driven platforms, the trajectory has been toward making controlled failure a routine part of building systems that must not fail [3], [7], [12]. The systems that survive their worst days are, increasingly, the ones that have already rehearsed them.

REFERENCES

[1] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, “Chaos engineering,” IEEE Software, vol. 33, no. 3, pp. 35–41, May–Jun. 2016.

[2] C. Rosenthal and N. Jones, Chaos Engineering: System Resiliency in Practice. Sebastopol, CA, USA: O’Reilly Media, 2020.

[3] Y. Izrailevsky and A. Tseitlin, “The Netflix Simian Army,” Netflix Technology Blog, Jul. 2011.

[4] [4] A. Tseitlin, “The antifragile organization,” Communications of the ACM, vol. 56, no. 8, pp. 40–44, Aug. 2013.

[5] J. Allspaw, “Fault injection in production: Making the case for resilience testing,” ACM Queue, vol. 10, no. 8, pp. 30–35, Aug. 2012.

[6] P. Alvaro, J. Rosen, and J. M. Hellerstein, “Lineage-driven fault injection,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, VIC, Australia, 2015, pp. 331–346.

[7] P. Alvaro et al., “Automating failure testing research at internet scale,” in Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC), Santa Clara, CA, USA, 2016, pp. 17–28.

[8] H. S. Gunawi et al., “FATE and DESTINI: A framework for cloud recovery testing,” in Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, USA, 2011, pp. 238–252.

[9] P. Joshi, H. S. Gunawi, and K. Sen, “PREFAIL: A programmable tool for multiple-failure injection,” in Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Portland, OR, USA, 2011, pp. 171–188.

[10] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA, USA: O'Reilly Media, 2016.
[11] Gremlin, Inc., "Gremlin: Enterprise chaos engineering platform," Gremlin Product Documentation, 2023.
[12] Cloud Native Computing Foundation, "Chaos Mesh: A cloud-native chaos engineering platform," CNCF Project Documentation, 2023.
[13] N. N. Taleb, Antifragile: Things That Gain from Disorder. New York, NY, USA: Random House, 2012.
[14] C. Bennett and A. Tseitlin, "Chaos Monkey released into the wild," Netflix Technology Blog, Jul. 2012.
[15] L. Hochstein, "The discipline of chaos engineering," in Seeking SRE, D. Blank-Edelman, Ed. Sebastopol, CA, USA: O'Reilly Media, 2018.
[16] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11–33, Jan.–Mar. 2004.
[17] P. Bailis and K. Kingsbury, "The network is reliable: An informal survey of real-world communications failures," ACM Queue, vol. 12, no. 7, pp. 48–58, Jul. 2014.
[18] Chaos Community, "Principles of chaos engineering," principlesofchaos.org, 2018.
[19] R. Miles, Learning Chaos Engineering: Discovering and Overcoming System Weaknesses Through Experimentation. Sebastopol, CA, USA: O'Reilly Media, 2019.
[20] D. D. Woods, "Four concepts for resilience and the implications for the future of resilience engineering," Reliability Engineering & System Safety, vol. 141, pp. 5–9, Sep. 2015.
[21] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in Proceedings of the IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Nara, Japan, 2016, pp. 57–66.