Engineer

Chaos engineering

controlled experiments that inject failure into production

What it is

Chaos engineering is the practice of injecting real-world failures into a running system on purpose, under control, to find out whether it copes. The Principles of Chaos Engineering, the short document most practitioners point to, defines it as “the discipline of experimenting on a system in order to build confidence” that the system can withstand turbulent conditions in production. The word that matters is experimenting. A chaos experiment does not check a feature against a specification; it asks a question about the whole system and accepts whatever answer comes back.

The case for it starts with distributed systems. Every service in one can be working correctly and the interactions between them can still produce outcomes nobody predicted, and rare real-world events such as a crashed server or a severed network link make that worse. The Principles list the kinds of weakness that hide there: fallback settings that are wrong when a service is unavailable, poor timeout settings that amplify retries, dependencies overwhelmed by their callers, and one failed component taking several others down. None of those shows up when each part is tested on its own.

An experiment has four steps in the Principles’ version:

- Define steady state as a measurable output of the system that indicates normal behaviour, such as throughput, error rate or a latency percentile.
- Hypothesise that the steady state will continue in both a control group and an experimental group.
- Introduce variables that reflect real events: servers that crash, disks that fail, network connections that are cut.
- Compare the groups’ measured outputs and look for evidence that contradicts the prediction.

The harder the steady state is to disturb, the more confidence the team can have in the system. When the hypothesis falls, the team has found a weakness while it is still small and while people are watching, rather than during an outage at night.

The best-known tool is Netflix's Chaos Monkey, which randomly terminates virtual machine instances and containers in production. Its README calls it an example of a tool that follows the Principles, and its stated purpose is to make engineers build services that are resilient to instance failures.

Use the concept in a decision

Put "What is chaos engineering?" to work in a real decision. We can help relate the concept to the system you are building or buying. Monitoring and alerts that speak up while a fault is still small, and the figures that show how often it stays that way.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built the company's first observability dashboard — the company's first view of its own systems

Built the company's first observability dashboard Get in touch →

Why it matters

Redundancy, failover and fallbacks are all promises about what happens when something breaks, and a promise that has never been tested is a hypothesis. A standby that has never taken traffic, a retry policy that has never met a slow dependency, an alert that has never fired: each of these may work, and the first real failure is an expensive way to find out. Chaos engineering moves that discovery to a time the team chooses.

It also tests the system rather than the parts. Unit and integration tests check behaviour someone already thought of. The Principles make the distinction directly: chaos experiments verify that the system does work, instead of trying to validate how it works. That is why they focus on the output a user would notice, such as error rate and latency, and not on the internals of any one component.

Production is where the evidence is. Systems behave differently depending on their environment and traffic patterns, and the Principles argue that sampling real traffic is the only reliable way to capture the request path. Google's Site Reliability Engineering book reaches the same conclusion from the other direction: synthetic load may not show where components really break, so once a

service has protection against overload, it suggests exercising failures against a limited part of the live service, with spare capacity ready in case the protections do not work and someone has to fail over by hand.

Finally, it changes incentives. Chaos Monkey's README puts the idea plainly: exposing engineers to failures more often gives them a reason to build resilient services. A team whose instances can be terminated on any working day designs for that from the start, instead of discovering the gap on the day a machine dies without warning.

What it looks like in practice

A chaos experiment is written down before it is run, much like a test plan. Here is one for an invented online shop, with every number made up for the example:

```
Experiment:   the payment provider slows down
Steady state: checkout success rate and p99 checkout latency, per 5 min
Hypothesis:   with 300 ms added to calls to the payment provider, the
              experimental group's success rate stays within 0.2 points
              of the control group's
Groups:       2% of checkout traffic in each, assigned at random
Variable:     300 ms of latency on the payment client, nothing else
Abort if:     success rate gap above 1 point for 2 minutes in a row
Window:       30 min, working hours, on-call engineer told in advance
Result:       hypothesis disproved; retries tripled the calls to the
              provider and p99 latency passed the 1 s timeout
Follow-up:    a retry budget per request; rerun the experiment after
```

A disproved hypothesis is the useful outcome here. The weakness is found on 2% of traffic for a few minutes, with an abort condition already agreed, and the follow-up is an engineering change rather than an apology.

The variable itself is often injected with ordinary tools. On Linux, the netem queueing discipline emulates the properties of real networks, including delay, loss, duplication and corruption. For example:

```
# experiment 1: add 100 ms of delay to every packet leaving eth0
tc qdisc add dev eth0 root netem delay 100ms
tc qdisc del dev eth0 root
```

```
# experiment 2: drop one packet in a thousand, at random
tc qdisc add dev eth0 root netem loss 0.1%
tc qdisc del dev eth0 root
```

The `del` line is the important one, and it belongs in a cleanup step that runs whether the experiment passed, failed or crashed halfway. Netem also takes a `seed` so that random loss can be reproduced, and its manual warns that for realistic TCP performance results netem has to be placed on the ingress of the receiving host.

At the level of whole machines, Chaos Monkey shows what careful defaults look like. Out of the box `enabled` is false and `leashed` is true, which means terminations are only simulated and logged. When it is switched on, it works through groups of instances (by app, stack or cluster), with a mean time between terminations of two days per group and a minimum of one day, on work days only, between 09:00 and 15:00 in its configured time zone. Accounts, regions and stacks can be excluded, and its configuration has a slot for an outage checker that tells it whether an outage is under way; the shipped implementations of that slot do nothing, so the check is the operator’s to supply. The SRE book’s own list of production failure tests is just as concrete: reduce task counts quickly or slowly, lose a cluster’s worth of capacity at once, and blackhole backends, including the noncritical ones whose silence can still stall a front end.

On this site, the case study of how we proved the intrusion ban works every run, from September 2026, is not a chaos experiment in the full sense, but it has the same shape. Every hardening run bans an address from the block the standards reserve for documentation, reads the resulting rule back from the kernel rather than trusting the service’s own status, and unbans it in a cleanup block that runs whether the check passed or failed. Because the test address belongs to nobody, a stray ban that outlives a failed run cannot lock out a real network, and the page is open about the cost: writing a rule into production on every run is a real intrusion into a live system, accepted because the alternative had already been shown to be worse. The backup and restore-test implementation applies that idea to recovery in its design: a monthly workflow restores a known file to a scratch location and compares its bytes. Its implementation record dated 2026-10-27 says the timers were disabled without a configured repository and records no completed live restore. The workflow exists in code; a successful recovery exercise remains a separate deliverable.

Where it goes wrong

Breaking things without a hypothesis. Killing a random server and watching what happens is a demonstration, not an experiment. Write down the steady state, the hypothesis and the abort condition first. If the team cannot see its steady state on a dashboard in real time, that is the first thing to fix, and the experiment can wait.

Watching internals instead of output. CPU on one host or the length of one queue says little about whether users noticed. The Principles ask for measurable output of the whole system, such as throughput, error rates and latency percentiles, because that is what the hypothesis is about.

An unbounded blast radius. The Principles accept some short-term negative impact but make containing the fallout the engineer's responsibility. Start with one instance or a small percentage of traffic, keep an abort switch that works without the tool that caused the damage, and never run an experiment on top of a live incident.

Never leaving staging. A staging environment is a sensible first step and a poor last one, because its traffic, data and neighbours are not production's. Move to a small slice of real traffic once the experiment has run cleanly before production, rather than treating staging results as the answer.

Running it once. A system that survived an experiment in March has been changed many times since. The Principles call manual experiments labour-intensive and ultimately unsustainable, and ask for experiments automated to run continuously, so that a regression in resilience shows up the week it is introduced.

Only ever killing machines. Clean crashes are the easy case, because something notices them. The Principles also name software failures such as malformed responses and non-failure events such as a traffic spike or a scaling event, and the SRE book adds a backend that accepts requests and never answers, which can still hurt a front end when requests have long deadlines. Prioritise variables by likely impact or by how often they happen.

Findings that go nowhere. A disproved hypothesis is a weakness with a name. Give it an owner and a priority in the same tracker as everything else, and rerun the experiment once the fix ships to confirm that the steady state now holds.

Further reading

- Principles of Chaos Engineering — the definition, the four steps of an experiment and the five advanced principles, last updated in March 2019.
- Chaos Monkey — Netflix's documentation for its instance-termination tool, which requires Spinnaker.
- Chaos Monkey: configuring behavior via Spinnaker — termination frequency, grouping and exceptions, with the default values.
- Chaos Monkey: configuration file format — every setting and its default, including the leashed mode and the working-hours window.
- Addressing Cascading Failures — chapter 22 of Google's Site Reliability Engineering book, including testing until failure and production failure tests.
- tc-netem(8) — the Linux manual page for the network emulator used to inject delay, loss and corruption.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is chaos engineering?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/chaos-engineering/