Chaos Engineering

Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal, Netflix

// *Modern software-based services are implemented as distributed systems with complex behavior and failure modes. Chaos engineering uses experimentation to ensure system availability. Netflix engineers have developed principles of chaos engineering that describe how to design and run experiments.* //

THIRTY YEARS AGO, Jim Gray noted that "A way to improve availability is to install proven hardware and software, and then leave it alone."[1] For companies that provide services over the Internet, "leaving it alone" isn't an option. Such service providers must continually make changes to increase the service's value, such as adding features and improving performance. At Netflix, engineers push new code into production and modify runtime configuration parameters hundreds of times a day. (For a look at Netflix and its system architecture, see the sidebar.) Availability is still important; a customer who can't watch a video because of a service outage might not be a customer for long. But to achieve high availability, we need to apply a different approach than what Gray advocated.

For years, Netflix has been running Chaos Monkey, an internal service that randomly selects virtual-machine instances that host our production services and terminates them.[2] Chaos Monkey aims to encourage Netflix engineers to design software services that can withstand failures of individual instances. It's active only during normal working hours so that engineers can respond quickly if a service fails owing to an instance termination.

Chaos Monkey has proven successful; today all Netflix engineers design their services to handle instance failures as a matter of course. That success encouraged us to extend the approach of injecting failures into the production system to improve reliability. For example, we perform Chaos Kong exercises that simulate the failure of an entire Amazon EC2 (Elastic Compute Cloud) region. We also run Failure Injection Testing (FIT) exercises in which we cause requests between Netflix services to fail and verify that the system degrades gracefully.[3]

Over time, we realized that these activities share underlying themes that are subtler than simply "break things in production." We also noticed that organizations such as Amazon,[4] Google,[4] Microsoft,[5] and Facebook[6] were applying similar techniques to test their systems' resilience. We believe that these activities form part of a discipline that's emerging in our industry; we call this discipline *chaos engineering*. Specifically, chaos engineering involves experimenting on a distributed system to build confidence in its capability to withstand turbulent conditions in production. These conditions could be anything from a hardware failure, to an unexpected surge in client requests, to a malformed value in a runtime configuration parameter. Our experience has led us to determine principles of chaos engineering (for an overview, see http://principlesofchaos.org), which we elaborate on here.

Shifting to a System Perspective

At the heart of chaos engineering are two premises. First, engineers should view a collection of services running in production as a single system. Second, we can better understand this system's behavior by injecting real-world inputs (for example, transient network failures)

NETFLIX AND ITS SYSTEM ARCHITECTURE

Netflix provides customers access to movies and TV shows, which are delivered by streaming over the Internet to devices such as smart TVs, set-top boxes, smartphones, tablets, and desktop computers.

The Netflix UI displays content tailored to the specific user. For example, the movie list on the home screen is customized on the basis of data such as what the user has watched previously. The UI includes features such as search, recommendations, ratings, user profiles, and bookmarks for resuming previously watched videos.

The Netflix infrastructure has two main components. *Open Connect* is Netflix's content delivery network, which is a geographically distributed cache of video data. Servers with large storage capacity are located at sites around the world. When a Netflix user hits the Play button, these servers stream video data to the user.

The *Control Plane* is a distributed set of services that implements all the UI functionality other than video streaming. The services in the Control Plane run inside virtual machines in the Amazon Web Services cloud, replicated across three geographic regions to reduce latency and improve availability. The chaos-engineering examples in the main article refer to services in the Control Plane.

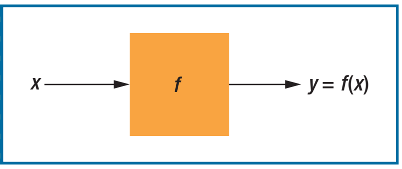

FIGURE 1. A functional perspective of a software system. The specification *f* defines how inputs (*x*) map to outputs (*y*). For distributed systems, such functional specifications are incomplete; they don't fully characterize a system's behavior for all possible inputs.

and observing what happens at the system boundary.

In a traditional software engineering approach, a functional specification describes how a software system should behave (see Figure 1). If the system comprises multiple programs, the functional specification also defines each program's behavior.

In practice, when you're describing distributed systems, functional specifications are incomplete; they don't fully characterize a system's behavior for all possible inputs. This problem is exacerbated by the scale and complexity of user behavior and the underlying infrastructure, which make it difficult to fully enumerate all inputs. In practice, the system's scale makes it impractical to compose specifications of the constituent services to reason about behavior.

From a system perspective, engineers focus on the dynamic view of the software as a set of processes running in production. This perspective assumes that the organization deployed the software into production some time ago and that the system has active users. The engineer views the system as a single entity and observes its behavior by capturing metrics at its boundary. The engineer cares about the typical behavior of metrics over time: the system's steady-state behavior. The behavior can be changed by user interactions (for example, the rate or type of requests), engineer-initiated changes (for example, runtime configuration changes or code changes), and other events such as the transient failure of a third-party service on which the system depends.

From this perspective, we ask questions such as "Does everything seem to be working properly?" and "Are users complaining?" rather than "Does the implementation match the specification?"

Principles of Chaos Engineering

In chaos engineering, as in other experimental disciplines, designing an experiment requires specifying hypotheses, independent variables, dependent variables, and context.[7]

We believe four principles embody the chaos-engineering approach to designing experiments:

- Build a hypothesis around steady-state behavior.
- Vary real-world events.
- Run experiments in production.
- Automate experiments to run continuously.

Build a Hypothesis around Steady-State Behavior

At Netflix, we're using chaos engineering to ensure that the system still works properly under a variety of different conditions. However, "works properly" is too vague a basis for designing experiments. In our context, the quality attribute we focus on most is availability. Each Netflix feature (for examples, see the sidebar) is implemented by a different service (or multiple services), each of which could fail. Yet, even if one of these internal services fails, that failure won't necessarily affect overall system availability.

Netflix services use fallbacks to ensure graceful degradation: failures in noncritical services have a minimal impact on the user experience. For example, consider a service *A* that makes requests against a caching service that sits in front of service *B*. If the caching service has a failure, *A* can fall back to making a request directly against *B*. This behavior isn't desirable in the long term because it increases request latency and the load on *B*, but in the short term the user isn't affected.

Other Netflix services provide personalized content for which the fallback would be to present a reasonable default. An example is the bookmark service. If it fails, the Netflix UI can default to starting videos at the beginning rather than providing a "resume from previous location" option.

Ultimately, what we care about is whether users can find content to watch and successfully watch it. We operationalize this concept by observing how many users start streaming a video each second. We call this metric *SPS*—(stream) starts per second.[8]

SPS is our primary indicator of the system's overall health. Although the term "chaos" evokes a sense of unpredictability, a fundamental assumption of chaos engineering is that complex systems exhibit behaviors regular enough to be predicted. Similarly, the SPS metric varies slowly and predictably throughout a day (see Figure 2). Netflix engineers spend so much time looking at SPS that they've developed an intuition about whether a given fluctuation is within the standard range of variation or is cause for concern. If they observe an unexpected change in SPS, we know the system has a problem.

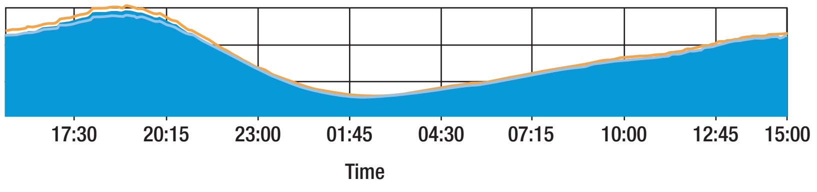

FIGURE 2. A graph of SPS ([stream] starts per second) over a 24-hour period. This metric varies slowly and predictably throughout a day. The orange line shows the trend for the prior week. The *y*-axis isn't labeled because the data is proprietary.

SPS is a good example of a metric that characterizes the system's steady-state behavior. Another such metric at Netflix is new account signups per second. A strong, obvious link exists between these metrics and the system's availability: if the system isn't available, users won't be able to stream video or sign up for the service. Other domains will use different metrics that characterize the system's steady-state behavior. For example, an e-commerce site might use the number of completed purchases per second, whereas an ad-serving service might use the number of ads viewed per second.

When designing chaos-engineering experiments, we form hypotheses around how the treatment will affect the system's steady state. For example, Netflix has software deployments in multiple geographic regions (Northern Virginia, Oregon, and Ireland). If an outage occurs in one region, we can fail over to another region; that is, redirect incoming client requests from the unhealthy region to a healthy one. When we test such a scenario, we hypothesize that failing over from one region to another will have minimal impact on SPS.

When we formulate a hypothesis for a chaos-engineering experiment, the hypothesis is about a particular kind of metric. Compare a metric such as SPS, which characterizes the overall system's steady state, with a finer-grained metric such as CPU load or time to complete a database query. Chaos-engineering experiment designs focus on the former: steady-state characterizations that are visible at the system's boundary and that directly capture an interaction between the users and system. We don't use the finer-grained metrics for the design of chaos-engineering experiments because they don't directly measure system availability.

However, we do observe finer-grained metrics when running chaos-engineering experiments to check whether the system is functioning properly, because these metrics indicate impact at the service level that the user doesn't feel. For example, increased request latency or CPU utilization might be a symptom that a service is operating in degraded mode, even though from the user's viewpoint, the system is working properly. Individual service owners at Netflix set up alerts on these internal metrics to catch problems with their particular services. We might even conclude an experiment early if finer-grained metrics indicate that the system isn't functioning correctly, even though SPS hasn't been impacted.

Vary Real-World Events

The "happy path" is a term of art in software development that refers to an execution trace through a program in which the inputs don't lead to any errors or corner cases. When engineers test their code, the temptation is great to focus on the happy path. This leads to code that works for common cases.

Unfortunately, those error conditions and corner cases will happen in the real world, even if they don't happen in our test cases. Clients of our services send malformed requests; services we consume send malformed responses; our servers die, their hard disks fill up, or their memory is exhausted; network latencies temporarily spike by several orders of magnitude; and traffic from our clients can spike unexpectedly. A recent study reported that 92 percent of catastrophic system failures resulted from incorrect handling of nonfatal errors.[9]

This brings us to the next chaos principle: vary real-world events. When designing chaos experiments, choose your stimulus by sampling from the space of all possible inputs that might occur in the real world. One obvious choice is to look at historical data to see what types of input were involved in previous system outages, to ensure that the kinds of problems that happened previously can't happen again. Providing access to this historical data is yet another reason for doing postmortems on system outages.

However, any input that you think could potentially disrupt the system's steady-state behavior is a good candidate for an experiment.

At Netflix, we've used inputs such as these:

- Terminate virtual-machine instances.
- Inject latency into requests between services.
- Fail requests between services.
- Fail an internal service.
- Make an entire Amazon region unavailable.

Although our experiments have focused on hardware and software failures as inputs, we believe that using other kinds of input is valuable as well. For example, we could imagine input such as varying the rate of requests received by a service, changing the runtime parameters, or changing the metadata that propagates through the system.

In some cases, you might need to simulate the event instead of injecting it. For example, at Netflix, we don't actually take an entire Amazon region offline; we don't have that capability. Instead, we take actions that assume that a region has gone offline—for example, redirecting client requests to other Amazon regions—and we observe the effects. Ultimately, the engineers designing the chaos experiments must use their judgment to make tradeoffs between the events' realism and the risk of harming the system. An interesting illustration of this trade-off is the experiments Kyle Parrish and David Halsey performed on a production financial-trading system.[10]

In other cases, we selectively apply the event to a subset of users. For example, as part of an experiment, we might make an internal Netflix service behave as if it's unavailable from the viewpoint of some users, whereas for others it appears to function properly. This lets us reduce the experiment's scope to mitigate risk.

Run Experiments in Production

The computing power of a single server has increased dramatically in the past several decades. Despite these increases, it's not possible to deploy the entire Netflix system on a single server. No individual server has the computing power, memory, storage, network bandwidth, or reliability to host the system, which has millions of subscribers. The only known solution for implementing an Internet-scale service is to deploy software across multiple servers and have them coordinate over the network, resulting in a distributed system. With the rise in microservices architectures' popularity,[11] more and more software engineers will likely be implementing Internet-based services as distributed systems.

Alas, traditional software-testing approaches are insufficient for identifying distributed systems' potential failures. Consider the following two failure modes based on observed failures at Netflix. In both cases, "client" and "server" refer to internal services, which typically act as both clients and the server.

In the first case, a server is overloaded and takes increasingly longer to respond to requests. One client places outbound requests on an unbounded local queue. Over time, the queue consumes more and more memory, causing the client to fail.

In the second case, a client makes a request to a service fronted by a cache. The service returns a transient error that's incorrectly cached. When other clients make the same request, they're served an error response from the cache.

Failure modes such as these require integration testing because they involve interactions among services. In some contexts, full integration testing might be possible only in production. At Netflix, fully reproducing the entire architecture and running an end-to-end test simply aren't possible.

However, when we can reproduce the entire system in a test context, we still believe in running experiments in production. This is because it's never possible to fully reproduce all aspects of the system in a test context. Differences will always exist, such as how synthetic clients behave compared to real clients, or DNS configuration issues.

Automate Experiments to Run Continuously

The final principle is to leverage automation to maintain confidence in results over time.

Our system at Netflix changes continuously. Engineers modify existing services' behavior, add services, and change runtime configuration parameters. In addition, new metadata continually flows through the system as Netflix's video catalog changes. Any one of these changes could contribute to a service interruption.

Because of these changes, our confidence in past experiments' results decreases over time. We expect that new experiments will first be run manually. However, for the experiments' results to be useful over time, we must automate the experiments to ensure they run repeatedly as the system evolves. The rate at which they run depends on the context. For example, at Netflix, Chaos Monkey runs continuously during weekdays, and we run Chaos Kong exercises monthly.

The idea of automating a chaos experiment can be intimidating because you're introducing some kind of stimulus that could lead to a system failure. However, Netflix's experience with Chaos Monkey suggests that such automation is feasible, and keeping it running ensures that new production services continue to be designed to withstand these failures.

Running a Chaos Experiment

On the basis of our four principles, here's how we define an experiment:

1. Define steady state as some measurable output of a system that indicates normal behavior.
2. Hypothesize that this steady state will continue in both the control group and experimental group.
3. Introduce variables that reflect real-world events such as servers crashing, hard drives malfunctioning, and network connections being severed.
4. Try to disprove the hypothesis by looking for a difference in steady state between the control group and experimental group.

Consider a Netflix service that provides some value to the user but isn't critical for video streaming—for example, the bookmark service. Imagine we're designing an experiment to verify that this service's failure won't significantly affect streaming.

For this experiment, we use SPS as the measurable output and check that a failure in the bookmark service has only a minor impact on SPS. We identify a small percentage of Netflix users who'll be involved in this experiment and divide them into the control group and experimental group. For the control group, we don't introduce any failures. For the experimental group, we simulate the service's failure by selectively failing all requests against that service associated with the experimental group's users. We can selectively fail requests for specific users through our FIT service. We hypothesize that the SPS values will be approximately equal for these two groups. We run the experiment for some period of time and then compare the SPS values.

At the experiment's end, either we have more confidence in the system's ability to maintain behavior in the presence of our variables, or we've uncovered a weakness that suggests a path for improvement.

The Future of Chaos Engineering

Although the concepts in this article aren't new, their application to improve software systems is still in its infancy. This article aims to explicitly define the underlying concepts to help advance the state of the practice. Software systems' increasing complexity will continue to drive the need for empirical approaches to achieving system availability. We hope that the practitioner and research communities will come to recognize chaos engineering as a discipline and continue to move it forward, particularly in the following areas.

Case Studies from Other Domains

We know from informal discussions with other Internet-scale organizations that they're applying similar approaches. We hope more organizations will document how they're applying chaos engineering, to demonstrate that these techniques aren't unique to Netflix.

Adoption

What approaches are successful for getting an organization to buy into

ABOUT THE AUTHORS

ALI BASIRI is a senior software engineer at Netflix and the first ever chaos engineer. He focuses primarily on automating the detection of weaknesses in distributed systems through fault injection, and is a coauthor of *Principles of Chaos*. Basiri received a BMath in computer science from the University of Waterloo. Contact him at abasiri@netflix.com.

NIOSHA BEHNAM is a senior software engineer and a founding member of the Traffic and Chaos Team at Netflix. He's responsible for the software that enables control-plane failover between Amazon Web Services regions and tools providing visibility into service resiliency. Behnam received a combined MS in engineering management and MBA from California Polytechnic State University, San Luis Obispo and an MS in network engineering from the University of California, Santa Cruz. Contact him at nbehnam@netflix.com.

RUUD DE ROOIJ is a software engineer at a startup company. He previously was a software engineer in Netflix's Traffic and Chaos Team, where he worked on traffic routing in the edge gateway software and the tools used for regional failover. His interests are global traffic routing, DNS, and building reliable systems out of unreliable components. De Rooij received an MSc in technical informatics from the Delft University of Technology. Contact him at ieee@ruud.org.

LORIN HOCHSTEIN is a senior software engineer in Netflix's Traffic and Chaos Team, where he works to ensure that Netflix remains available. His interests include the failure modes of distributed systems, operations engineering, and empirical software engineering. Hochstein received a PhD in computer science from the University of Maryland. Contact him at lorin@netflix.com.

LUKE KOSEWSKI is a former site reliability engineer and a founding member of Netflix's Traffic and Chaos Team, where he devises novel ways to balance and manipulate edge traffic hitting the Netflix stack. Kosewski received a BMath (specializing in computer science) from the University of Waterloo. Contact him at lkosewski@netflix.com.

JUSTIN REYNOLDS is a senior software engineer in Netflix's Traffic and Chaos Team. He focuses on intuition engineering and developing novel ways to build intuition around the health of systems too complex to grasp through traditional means. Reynolds received a BS in computer science from San Jose State University. Contact him at jreynolds@netflix.com.

CASEY ROSENTHAL is an engineering manager at Netflix. He manages teams to tackle big data, architect solutions to difficult problems, and train others to do the same. Rosenthal received a BA in philosophy from Ohio University. Contact him at crosenthal@netflix.com.

chaos engineering and getting that organization's engineering teams to adopt it?

Tooling

At Netflix, we're using tools built in-house that work with the infrastructure we've built. It's not clear how much of the tooling of chaos experiments will be specific to a particular organization's infrastructure and how much will be reusable. Is it possible to build a set of tools that are reusable across organizations?

Event Injection Models

The space of possible events to inject into a system is potentially large, especially when you consider combinations of events. Research shows that many failures are triggered by combinations of events rather than single events.[9] How do you decide what set of experiments to run?

If you'd like to reach out to discuss chaos engineering, contact us at chaos@netflix.com.

References

1. J. Gray, *Why Do Computers Stop and What Can Be Done about It?*, tech. report 85.7, Tandem Computers, 1985.
2. C. Bennett and A. Tseitlin, "Chaos Monkey Released into the Wild," *Netflix Tech Blog*, 30 July 2012; http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.
3. K. Andrus, N. Gopalani, and B. Schmaus, "FIT: Failure Injection Testing," *Netflix Tech Blog*, 23 Oct. 2014; http://techblog.netflix.com/2014/10/fit-failure-injection-testing.html.
4. J. Robbins et al., "Resilience Engineering: Learning to Embrace Failure," *ACM Queue*, vol. 10, no. 9, 2012; http://queue.acm.org/detail.cfm?id=2371297.
5. H. Nakama, "Inside Azure Search: Chaos Engineering," blog, Microsoft, 1 July 2015; https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering.
6. Y. Sverdlik, "Facebook Turned Off Entire Data Center to Test Resiliency," *Data Center Knowledge*, 15 Sept. 2014; www.datacenterknowledge.com/archives/2014/09/15/facebook-turned-off-entire-data-center-to-test-resiliency.
7. W.R. Shadish, T.D. Cook, and D.T. Campbell, *Experimental and Quasi-experimental Designs for Generalized Causal Inference*, 2nd ed., Wadsworth, 2001.
8. P. Fisher-Ogden, C. Sanden, and C. Rioux, "SPS: The Pulse of Netflix Streaming," *Netflix Tech Blog*, 2 Feb. 2015; http://techblog.netflix.com/2015/02/sps-pulse-of-netflix-streaming.html.
9. D. Yuan et al., "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems," *Proc. 11th USENIX Symp. Operating Systems Design and Implementation* (OSDI 14), 2014; www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf.
10. K. Parrish and D. Halsey, "Too Big to Test: Breaking a Production Brokerage Platform without Causing Financial Devastation," presentation at Velocity 2015 Conf., 2015; http://conferences.oreilly.com/velocity/devops-web-performance-ny-2015/public/schedule/detail/45012.
11. S. Newman, *Building Microservices*, O'Reilly Media, 2015.