The Business Case for Chaos Engineering

While Chaos Engineering has gained currency in the Site Reliability Engineering community, service and business owners are often nervous about experimenting in production. Proving the benefits of Chaos Engineering to these stakeholders before implementing a program can be challenging. We present the business case for Chaos Engineering, through both qualitative and quantitative tactics, as well as the benefits and tools to convince stakeholders this is necessary and cost-efficient.

THE DISCIPLINE OF CHAOS ENGINEERING

Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production. By running chaos experiments directly on a production system in a controlled manner, you can improve your system’s availability by identifying and eliminating problems before they manifest as outages.

Consider a chaos experiment on a microservice architecture that injects latency between calls from service A to service B. This experiment might reveal that a spike in the response time of service B triggers a retry storm that overloads essential service C, a weakness that would not be detected by unit or integration testing (see sidebar: how chaos relates to traditional testing).

As an example, at Netflix there is an internal microservice that acts as a generic key-value store, and is used by other microservices. This store is not considered a critical service. We ran a chaos experiment where we intentionally injected latency for a small fraction of traffic so that calls to this service would exceed their timeouts and trigger fallback behavior. This experiment led to an unexpected increase in errors in a service that determines which content delivery network URLs to return to client devices, which is critical to video streaming. The exposed vulnerability was eliminated by adjusting timeout thresholds.
This type of experimentation requires a technical shift within the organization. If applications were not originally designed to support chaos experiments, engineers must incorporate new tooling such as fault injection and guard rails to minimize blast radius. More importantly, Chaos Engineering may require a cultural shift. In doing so, a successful program can change the way software engineers build systems by creating incentives for resilient design.

Chaos Engineering sounds dangerous (“you want to intentionally cause problems in the production system?”), so you’ll need to make the case to your organization that a chaos program is worthwhile.

GETTING BUY-IN

The “It Doesn’t Apply to Me” Fallacy

In some domains, such as finance, system failures can be extremely costly. The Knight Capital Group, a U.S. trading firm, famously lost over $400 million because of a software configuration issue. In other domains, such as transportation, a system failure can lead to injury or loss of life. It is not surprising that stakeholders in these domains would be reluctant to run experiments on production systems.

Yet, it is because of the increased cost of impact that we believe Chaos Engineering should have a higher focus in these types of organizations. Regardless of the level of impact, it’s important to have safety features with chaos experiments that minimize blast radius. Experimenting in production has the potential to cause unnecessary customer pain. While there must be an allowance for some short-term negative impact, it is the responsibility and obligation of the Chaos Engineering team to ensure the fallout from experiments is minimized and contained. If Chaos Engineering is well-scoped, the return on investment (ROI) of the practice outweighs the short-term pain it can cause along the way. In forest management, controlled burning is used in cooler months to prevent devastating forest fires in hotter and dryer times of the year. In the same way, controlled chaos experiments can uncover vulnerabilities that would otherwise cause significant damage.

Several high-cost impact industries and companies have already started practicing Chaos Engineering, including PagerDuty (incident response), the U.S. Air Force (aerospace), ING (finance), and Polysync and Uber (automotive).

Quantitative Benefits: ROI of Chaos

To estimate the ROI for chaos, you need to estimate the benefits and costs of a Chaos Engineering program. First-order benefits and costs are directly observable, and hence the simplest to estimate and reason about.

The main first-order benefit of a Chaos Engineering program is a reduction in the number of preventable outages. If the Chaos Engineering program is successful, there should be fewer outages that occur than if the program had not been introduced, all other things being equal.

A Chaos Engineering program has two first-order costs. The first cost is the engineering effort required to implement the program. As an example, Netflix’s Chaos Engineering team is made up of four full-time software engineers.

The second cost is the increase in the amount of chaos-induced harm, and the cost of mitigating that harm. Some induced harm will be intentional, and some will be accidental. Intentionally inducing harm may sound odd, but that’s part of what Chaos Engineering is about. Every outage preventable by chaos engineering will have corresponding chaos-induced harm that is uncovered by an experiment. The difference is that the chaos-induced harm (if done correctly) should be orders of magnitude smaller than the outage it prevented.

Chaos experiments identify vulnerabilities in the system that would otherwise manifest as outages. By minimizing blast radius during experiments, we work to make the impact as small as
possible while still being detectable. As a result, the impact of chaos-induced outages is not zero. Some outages induced by Chaos Engineering will be, alas, accidental. Any new failure injection program brings with it the risks of accidentally causing problems.

**ROI Equation**

To calculate the ROI for Chaos Engineering, we need to know:

- \( C(U) \) – cost of all outages, not using Chaos Engineering
- \( C(U') \) – cost of all outages, using Chaos Engineering
- \( C(U_c') \) – cost of chaos-induced harm
- \( E \) – effort to implement chaos

The ROI equation then becomes

\[
\text{ROI} = \frac{\text{benefits} - \text{costs}}{\text{benefits}} = \frac{C(U) - C(U') - E}{C(U_c') + E}
\]

Given that:

\[
C(U) = C(U_p) + C(U_n)
\]

and

\[
C(U') = C(U_p) + C(U_c')
\]

where

- \( C(U_p) \) – cost of outages preventable by chaos
- \( C(U_n) \) – cost of outages not preventable by chaos

To do an ROI calculation, we need to estimate:

- the cost of preventable outages, \( C(U_p) \)
- the cost of harm induced by the Chaos Engineering program, \( C(U_c') \)
- the effort to implement the program, \( E \)

Estimating the effort to implement the program is straightforward, because you’ll choose the staffing levels. Computing the cost of the outages requires some additional data.

**Modeling outages using historical data**

In an ideal world, we could split the universe into two timelines: in one timeline we institute a Chaos Engineering program, and in another one we don’t, then we collect the outage information from each timeline. Unfortunately, we can’t do an experiment like that.

To approximate peering into an alternate reality, you can estimate the cost of preventable outages by looking at historical data. If you have a good incident management system in place, you already have a historical record of previous incidents. You’ll need to identify which of these could have been prevented if a Chaos Engineering program detected the vulnerability.

At Netflix, our chaos program focuses on the risk of vulnerability to failures in non-critical services (see the sidebar “Surprise! Your Service is Critical”). We look for vulnerabilities where the failure of a non-critical service could put the system in a dangerous state that would lead to an outage.
By identifying which past outages were caused by the failure of a non-critical service, and the impact of these outages, we can estimate how much money we would have saved if we could have avoided them.

Modeling the cost of outages

For the purpose of ROI, once you have a set of historical outages that could have been prevented with chaos, you need to put a dollar value on the cost of the outages.

Depending on your business model, putting a cost on individual outages may be straightforward or difficult. For example, for an online store you can estimate the cost of the outage by the difference between the amount of sales that you would normally process versus the number that were processed during the outage.

We use the term “outage” at Netflix to refer to an incident where some of our customers are unable to stream video. Internally, we assess the impact of an outage by the number of missed streams, the difference between our estimate of how many video streams should be playing (given historical trends) and the observed count of how many video streams are playing.

For Netflix, putting a dollar value on an outage is more complex. Because Netflix uses a subscription model, the relationship between revenue and missed streams are indirect. We believe that an increase in large-scale outages would lead to more people canceling their subscriptions, but coming up with a model that relates outages and cancellation rates is non-trivial. In addition, the effects of outages may be non-linear: one outage that affects 10X users may be more costly to the organization than ten outages that each affect X users, and clusters of outages may be more costly than outages that are evenly distributed throughout the year.

Ultimately, keep in mind that the ROI calculation is based on a simplified model, and it’s there to help you make a decision. Your model that estimates costs won’t fully capture reality, but the exercise of developing it will help you think more about how outages impact your organization’s bottom line.

You’ll also need to estimate the cost of the harm that is deliberately induced by Chaos Engineering. For example, if an injected fault intentionally degrades the user experience (e.g., by serving a non-personalized response), try to quantify the cost the organization of a customer having a degraded experience. This is also a useful exercise because it forces you to make explicit how large of an impact you will allow a chaos experiment to have. You could define an explicit budget for the deliberate impact you will allow the chaos team to have. At Netflix, we automatically detect and stop chaos experiments if they cross a predefined threshold of missed streams. The threshold was chosen to be small enough that it would not trigger existing alerts.

Example scenario

Consider the following (fictional) scenario:

- In the past 12 months there were 2 major outages and 8 minor outages that could have been detected by Chaos Engineering that cost the organization an estimated $700,000 in lost revenue
- A chaos team of three members that cost $150,000 each.
- A budget of $10,000 in missed revenue due to Chaos Engineering experiments

\[
\text{ROI} = \frac{\$700,000 - \$10,000 - 3 \times \$150,000}{\$10,000 + 3 \times \$150,000} = 0.5217
\]

The estimated ROI would be about 52%.
Qualitative benefits: Influencing Better Engineering Practices

Prof. Daniel Kahneman observed that “no one ever made a decision because of a number – they needed a story.” Even if you can demonstrate positive ROI, you’ll need to make a qualitative argument, tell a good story, to make an effective case for chaos. The second-order benefits of Chaos Engineering, described below, can help build this narrative.

Improving design decisions

Teams which leverage chaos practices while building systems begin to think about failure in a first-class way. Design discussions which used to start with “if component X fails…” become “when component X fails.” This is a very subtle, yet powerful, change to strive for because it encourages teams to talk about fallbacks and trade-offs up front and will lead to more hardened and resilient systems in the long run.

When Netflix moved out of the data center and into the cloud, engineers quickly learned about the perilous properties of the cloud: commodity-grade hardware combined with a scale-out approach where scale is achieved by adding more servers. The result was that, in the cloud, servers were more likely to disappear, leaving subscribers without service and our engineers scrambling in the middle of the night to restore the service. That’s when we created Chaos Monkey. Chaos Monkey randomly terminates production server instances during business hours, when engineers are available to track and fix issues. This quickly uncovered many of our vulnerabilities to instance terminations, making our service resilient to this class of failures. For example, there was a hand-patched “snowflake” server that was responsible for synchronizing some DNS configuration. When Chaos Monkey terminated the server, Amazon replaced it with a server that did not have the hand-patched settings, which resulted in an incident.

More importantly, Chaos Monkey created a cultural shift at Netflix, where engineers now design services to be resilient to instance terminations, knowing that Chaos Monkey could be lurking around the corner.

Understanding service criticality

Because distributed systems frequently suffer from partial failures, it is important to understand which services are the critical ones within your system. For instance, at Netflix the ultimate business goal is to serve video playback to customers. Thus, services that directly impact whether or not playback is available are critical to achieving that goal and we label them Tier 1. For example, a service that returns a digital rights management license, required to decrypt a video, is a Tier 1 service. However, a service which provides a personalized recommended list of movies is not critical to video playback, and we’ll refer to such services as Tier 2.

This distinction provides a useful mechanism for defining service tiers (see sidebar: Surprise! Your service is critical). Netflix Chaos Engineering focuses on two tiers, but you could define as many as you need for your testing.
Teams which own services should (1) know which type of service they are operating and (2) be able to demonstrate that they are not more critical than their designated tier. Chaos experiments enable teams to do this in a contained manner.

Assuming each tier has different requirements for on-call and production-readiness, this can also be a motivating factor in prioritizing work on resiliency efforts. For example, teams may choose to spend additional engineering resources to build in fallbacks if it allows them to move their service into a lower tier with less stringent operational requirements.

Validating reliability measures

There are many tools which can be leveraged to make distributed systems resilient to system faults. A few examples of scalability patterns include timeouts, fallbacks, circuit breakers, and bulkheads.9 Like all software, these mechanisms can be independently tested in the small via unit tests to ensure they function as desired. However, the true test of these strategies is to see how they react in a production environment under load, particularly when you consider that each one requires tuning and configuration to set boundary conditions.

One example of a reliability measure we use at Netflix is Hystrix. Hystrix implements the circuit breaker pattern for wrapping RPC calls. This enables teams to define a fallback that is served in the case of a service outage. If the Hystrix error percentage crosses a configured threshold, the circuit opens and serves a fallback instead of making calls to downstream services. It then slowly retries the RPC calls at a lower rate until the failing service recovers. This is a useful tool for protecting the calling service from backing up threads, but it only works if the error threshold is configured correctly and the fallback functions as expected. We have seen outages as a result of failures in this code and configuration, so in order to ensure Hystrix barriers continue to function, we are increasing chaos experiments to validate them on a regular, on-going basis.

AFTER BUY-IN: BEGINNING THE CHAOS

The Role of Chaos in Organizations

Once you’ve received approval from your organization to start a chaos program, the first step is to specify the success criteria for your business. Three items should be analyzed prior to beginning chaos experimentation: business goals, key performance indicators (KPIs), and the cost of impact to service availability.

Acquiring and retaining customers are ubiquitous business goals across many industries; both of which are severely impacted if functionality of your core product is unavailable.
To measure progress against business goals, engineering teams monitor key performance indicators (KPIs), or metrics, which directly impact the success of the business. Technology teams and business partners should align on KPIs before experimentation begins. For example:

- For a home security company, a KPI may be the time between a security panel being armed and doors being locked.
- For an ecommerce company, this might be the number of customers checking out, searching for an item, or adding an item to a shopping basket.
- For a ride-sharing company, it may be the volume of rides hailed in a given minute.
- For a video streaming company, this might be how many streams are started in a given second.

In order to monitor Chaos experiments, a baseline for these metrics should be established which can be based on historical data. The baseline is then compared to the experiment KPIs to understand whether an experiment is causing impact.

The cost of impact varies depending on your business; it could be the cost of acquiring a customer, a safety impact, the engineering hours spent resolving the issue, phone calls to your Customer Service team, or a combination of the aforementioned.

The relationship between business goals, cost of impact, and KPIs will give you a strong indication of whether Chaos Engineering can, and should, be applied to your organization. A high cost of impact, associated with a given KPI, associated with a key business goal should be regularly experimented on through Chaos Engineering to ensure a higher likelihood of success. This can be done by monitoring KPIs while failing various components that fuel them in order for engineers to have a better understanding of the effect and potential blast radius of a failure.

**Running Chaos Experiments**

Ideally, you would run chaos experiments any time your system changes, just as you would with any other type of testing. However; there is risk of impacting customers with these experiments, so there is a balance to strike between minimizing impact to customers and finding the vulnerabilities quickly before it can turn into a larger problem. This becomes more difficult the more complex your environment is. For instance, if you have mechanisms for people to dynamically push configuration updates into the environment, that is a change to the system.

At Netflix, there are tools like Chaos Monkey that run all the time in production and they have become part of the engineering culture. As we dive into more large-scale types of testing, we are more cautious, yet the frequency has increased over the last few years. Originally, there was a push to run experiments manually once or twice leading up to a major release or holiday code freeze. This enabled teams to bolster their systems for that particular event; however, as soon as we resumed changes the systems were vulnerable. As a result, we are moving toward more automation for these experiments. Many service owners are now running chaos experiments weekly or monthly depending on the risk associated with running those experiments. As the tooling improves to catch problems (and kill bad experiments quickly), more experiments will be possible.

For more details on how to design and run Chaos Engineering experiments, see the work of Rosenthal and colleagues.\(^{10}\)

**Adoption Evolution**

The true mark of success for introducing new technology into an organization is whether or not it changes the process, and if that process change is for the better. While it’s important to monitor the quantitative aspects of Chaos Engineering, it’s equally important to measure qualitative aspects. Throughout the adoption process, measure your successes and how things are changing within the organization.
Different cultures and industries approach Chaos Engineering adoption a bit differently. You likely aren’t going to be able to convince everyone of its potential success immediately. We suggest starting with a critical or “Tier 1” service. Getting a critical service to experiment with Chaos Engineering is useful in many ways:

- Downstream services are more likely to adopt if a leader service adopts first
- Higher ROI with Chaos Engineering
- Less time spent mitigating outages

For your first pass, start with a previous outage and go backwards. Before beginning chaos, you should be able to answer the following questions about the outage you’re recreating: which services were impacted, what calls failed, and where the issue started from.

Most service owners won’t object to continuous experimentation on something that has the potential to be a high-cost outage to the organization in order to prevent it from happening again. Once this is in place, gradually expand the chaos experiments towards other services that behave similarly and continue doing this until you have full “basic” adoption. After basic adoption is achieved, you can increase granularity with your experiments in terms of latency and failure based scenarios.

As your adoption evolves, Chaos Engineering should become more automated. However, it’s recommended to start with a consultancy model when introducing chaos in order for service owners and business owners to be kept in the loop with regards to potential risk and reward. Once you feel comfortable teams have an understanding and involvement in the Chaos Engineering practice, begin to remove yourself from the equation and allow the tool to handle more and more of the responsibilities.

**CONCLUSION**

Because Chaos Engineering is still a young discipline, attempts to introduce a chaos program are likely to be met with skepticism. By telling a story around the benefits of chaos realized at organizations such as Netflix, and by making quantitative arguments based on your historical data, you should be able to make a strong case that introducing Chaos Engineering will yield real, tangible benefits to the business.

**SIDEBAR: HOW CHAOS RELATES TO TRADITIONAL TESTING**

To understand the goals of Chaos Engineering, it is important to see where it fits into the larger development life cycle.

Unit tests are critical for validating expected results at a very granular level and unit tests often mock dependencies to run as consistently and quickly as possible. The goal of unit testing is to validate what an individual component is doing—in other words, given certain inputs, determine the expected outputs.

Integration tests have a similar goal – however, they attempt to verify that components work together to achieve the desired result. As microservice architectures have grown more popular, integration testing encompasses not only the interaction of components within a single service, but also across services.

While integration and unit testing are valuable for identifying defects, they don’t cover everything. In addition to ensuring a system is functionally correct, chaos experiments validate assumptions about how systems react to different types of faults between components. Distributed systems fail often. They also fail differently under production load than they do by running a single failure scenario as part of a unit or integration test. By showing how each component within the system behaves under failure conditions, Chaos Engineering exposes system vulnerabilities.
This discipline enables platform and service owners to build confidence in resilience mechanisms such as retries, timeouts, circuit breakers, load balancers, and back pressure.

Figure 2. Unit Testing, Integration Testing, and Chaos Experiments

A single service may have thousands of unit and integration tests validating expectations as defined by the business, but without chaos experiments, it’s impossible to know that the system as a whole will be fault tolerant and available for your customers.

SIDEBAR: SURPRISE! YOUR SERVICE IS CRITICAL

In a complex distributed system, a microservice architecture can easily become a distributed monolith, where a fault in any part of the system can take down the entire system. At Netflix, we use fallbacks and circuit breakers to protect our critical services from faults in non-critical systems. We define a critical service to be any service that is required for selecting a video and playing it back. For example, we have a non-critical service that tracks where to resume a movie for a customer that left halfway through. If that service goes down, the fallback would be to play from the beginning of the movie during the downtime. Another service displays a personalized list of movies and shows to users based on their past viewing preferences. If that service goes down, the fallback would be to display a list of top movies and shows to our customers. Sometimes our non-critical services accidentally become critical. In order to find these types of problems faster, we’ve built a tool called Chaos Automation Platform (ChAP) to inject failure and latency into non-critical services in production in order to observe their impact on critical services in a contained manner.11
REFERENCES


ABOUT THE AUTHORS

**Haley Tucker** is a senior software engineer on the Chaos Engineering team at Netflix, where she is responsible for verifying the resiliency of Netflix services to ensure that customers always enjoy their favorite shows. Tucker received a BS in Computer Science from Texas A&M University. Contact her at haley@netflix.com.

**Lorin Hochstein** is a senior software engineer on the Chaos Team at Netflix, where he works on ensuring that Netflix remains available in the face of continual production changes. Hochstein received a PhD in Computer Science from the University of Maryland. Contact him at lhochstein@netflix.com.

**Nora Jones** is a senior software engineer on the Chaos Engineering team at Netflix, where she works on ensuring that Netflix remains resilient in the face of uncertain conditions. Jones received a BS in Computer Engineering from Virginia Tech and is currently pursuing an advanced degree in Human Factors and Systems Safety at Lund University. Contact her at noraj@netflix.com.

**Ali Basiri** is the founder of the Chaos Engineering team at Netflix where, he works on ensuring Netflix’s availability through a series of resiliency initiatives such as Chaos Automation Platform (ChAP), Chaos Monkey, and FIT. Basiri has a BMath in Computer Science from the University of Waterloo, Canada. Contact him at abasiri@netflix.com.

**Casey Rosenthal** formalized the practice of Chaos Engineering by co-writing and publishing the definition principlesofchaos.org with the Chaos Team at Netflix, which he managed for three years. Rosenthal is currently CTO at Backplane.io, a company that provides reliable and resilient infrastructure. Contact him at casey@backplane.io.