Introduction to chaos engineering, Part 1: Crash test your application

One way to test a complex software system is to carefully break things and see what happens.

by Mikolaj Pawlikowski

June 25, 2021

[This two-part excerpt from the book Chaos Engineering: Site Reliability Through Controlled Disruption is published with the kind permission of Manning Publications. You can download a longer excerpt at no cost, courtesy of the team behind Oracle Linux. —Ed.]

What would you do to make absolutely sure the car you’re designing is safe? A typical vehicle today is a real wonder of engineering. A plethora of subsystems, operating everything from rain-detecting wipers to lifesaving airbags, all come together to not only go from A to B but to protect passengers during an accident. Isn’t it moving when your loyal car gives up the ghost to save yours through the strategic use of crumple zones, from which it will never recover?

Because passenger safety is the highest priority, all these parts go through rigorous testing. But even assuming they all work as designed, does that guarantee you’ll survive in a real-world accident? If your business card reads “New Car Assessment Program,” you demonstrably don’t think so. Presumably, that’s why every new car making it to the market goes through crash tests.
Picture this: a production car, heading at a controlled speed, closely observed with high-speed cameras, in a lifelike scenario—crashing into an obstacle to test the system as a whole. In many ways, chaos engineering is to software systems what crash tests are to the car industry: a deliberate practice of experimentation designed to uncover systemic problems. In this article, you'll look at the why, when, and how of applying chaos engineering to improve your computer systems. And perhaps, who knows, you'll save some lives in the process. What's a better place to start than a nuclear power plant?

**What is chaos engineering?**

Imagine you're responsible for designing the software operating a nuclear power plant. Your job description, among other things, is to prevent radioactive fallout. The stakes are high: A failure of your code can produce a disaster leaving people dead and rendering vast lands uninhabitable. You need to be ready for anything, from earthquakes, power cuts, floods, and hardware failures to terrorist attacks. What do you do?

You hire the best programmers, set in place a rigorous review process, test coverage targets, and walk around the hall reminding everyone that we're doing serious business here. But, “Yes; we have 100% test coverage, Mr. President!” will not fly at the next meeting. You need contingency plans; you need to be able to demonstrate that when bad things happen, the system can withstand them and the name of your power plant stays out of the news headlines. You need to go looking for problems before they find you. That's what this article is about.

*Chaos engineering* is defined as “the discipline of experimenting on a system in order to build confidence in the system’s capability to withstand turbulent conditions in production” (see *Principles of Chaos Engineering*). In other words, it's a software testing method focused on finding evidence of problems before they are experienced by users.

You want your systems to be reliable, and that's why you work hard to produce good-quality code and good test coverage. Yet, even if your code works as intended, in the real world plenty of things can (and will) go wrong.

The list of things that can break is longer than a list of the possible side effects of painkillers: starting with sinister-sounding events such as floods and earthquakes, which can take down entire data centers, to power supply cuts, hardware failures, networking problems, resource starvation, race conditions, unexpected peaks of traffic, and complex and unaccounted-for interactions between elements in your system, all the way to the evergreen operator (human) error.

And the more sophisticated and complex your system, the more opportunities for problems to appear.
It’s tempting to discard these as rare events, but they just keep happening. In 2019, for example, two crash landings occurred on the surface of the Moon: The Indian Chandrayaan-2 mission and the Israeli Beresheet both were lost on lunar descent. And remember that even if you do everything right, you still depend on other systems, and these systems can fail.

It’s a common misconception that chaos engineering is only about randomly breaking things in production. It’s not. Although running experiments in production is a unique part of chaos engineering, it’s about much more than that—it’s about anything that helps you be confident the system can withstand turbulence. It interfaces with site reliability engineering (SRE), application and systems performance analysis, and other forms of testing. Practicing chaos engineering can help you prepare for failure and, by doing that, learn to build better systems, improve existing ones, and make the world a safer place.

**Motivations for chaos engineering**

At the risk of sounding like an infomercial, there are at least three good reasons to implement chaos engineering.

- Determining risk and cost, and setting service-level indicators (SLIs), service-level objectives (SLOs), and service-level agreements (SLAs)
- Testing a system (often complex and distributed) as a whole
- Finding emergent properties that you were unaware of

Let’s take a closer look at these motivations.

**Estimating risk and cost, and setting SLIs, SLOs, and SLAs**

You want your computer systems to run well, and the subjective definition of what well means depends on the nature of the system and your goals regarding it. Most of the time, the primary motivation for companies is to create profit for the owners and shareholders. The definition of running well will, therefore, be a derivative of the business model objectives.

Let’s say you’re working on a planet-scale website designed for sharing photos of cats and toddlers while also checking on your high-school ex. Your business model might be to serve your users targeted advertisements, so you will want to balance the total cost of running the system with the amount of money you can earn from selling these ads. From an engineering perspective, one of the main risks is that the entire site could go down, and you wouldn’t be able to present ads and bring home the revenue. Conversely, not being able to display a particular cat picture in the rare event of a problem with the cat picture server is probably not a deal breaker and will affect your bottom line in only a small way.
For both risks (users can’t use the website, and users can’t access a cat photo momentarily), you can estimate the associated cost, expressed in dollars per unit of time. That cost includes the direct loss of business as well as various other, less tangible things, such as public-image damage, that might be equally important.

Now, to quantify these risks, the industry uses SLIs. In our example, the percentage of time that your users can access the website could be an SLI. And so could the ratio of requests that are successfully served by the cat photos service within a certain time window. The SLIs are there to put a number to an event, and picking the right SLI is important.

Two parties agreeing on a certain range of an SLI can form an SLO, a tangible target that the engineering team can work toward. SLOs, in turn, can be legally enforced as an SLA, in which one party agrees to guarantee a certain SLO or otherwise pay some form of penalty if they fail to do so.

Going back to our cat- and toddler-photo-sharing website, one possible way to work out the risk, SLI, and SLO could look like this.

- The main risk is “People can’t access the website,” or simply the *downtime*.
- A corresponding SLI could be “the ratio of success responses to errors from our servers.”
- An SLO for the engineering team to work toward could be “the ratio of success responses to errors from our servers is greater than 99.95% on average, monthly.”

Here’s a different example: Imagine a financial trading platform, where people query an API when their algorithms want to buy or sell commodities on the global markets. Speed is critical. You could imagine a different set of constraints, set on the trading API.

- SLI: 99th percentile response time
- SLO: 99th percentile response time is less than 25 milliseconds (ms), 99.999% of the time

From the perspective of the engineering team, that sounds like mission impossible: we allow ourselves about only 5 minutes a year when the top 1% of the slowest requests average over 25 ms response time. Building a system like that might be difficult and expensive.

To satisfy the SLOs, you’ll engineer the system in a certain way. You will need to consider the various sinister scenarios, and the best way to see whether the system works fine in these conditions is to create them—which is exactly what chaos engineering is about! You’re effectively working backward from the business goals to an engineering friendly SLO that you can, in turn, continuously test against by using chaos engineering.
Notice that in all of the preceding examples, I am talking in terms of entire systems.

**Number of nines.** In the context of SLOs, we often talk about the number of nines to mean specific percentages. For example, 99% is two nines, 99.9% is three nines, 99.999% is five nines, and so on.

Sometimes, we also use phrases such as three nines five or three and a half nines to mean 99.95%, although the latter is not technically correct (going from 99.9% to 99.95% is a factor of 2, while going from 99.9% to 99.99% is a factor of 5).

The following are a few of the most common values and their corresponding downtimes per year and per day:

- 90% (one nine)—36.53 days per year, or 2.4 hours per day
- 99% (two nines)—3.65 days per year, or 14.40 minutes per day
- 99.95% (three and a half nines)—4.38 hours per year, or 43.20 seconds per day
- 99.999% (five nines)—5.26 minutes per year, or 840 ms per day

**Testing a system as a whole**

Various testing techniques approach software at different levels.

- **Unit tests** typically cover single functions or smaller modules in isolation.
- End-to-end (e2e) tests and integration tests work on a higher level; whole components are put together to mimic a real system, and verification is done to ensure that the system does what it should.
- **Benchmarking** is yet another form of testing, focused on the performance of a piece of code, which can be lower level (for example, microbenchmarking a single function) or a whole system (for example, simulating client calls).

I like to think of chaos engineering as the next logical step: a little bit like e2e testing, except we rig the conditions to introduce the type of failure we expect to see, and we measure that we still get the correct answer within the expected time frame. Even a single-process system can be tested using chaos engineering techniques, and sometimes that comes in really handy.

**Finding emergent properties**

Our complex systems often exhibit emergent properties that we didn’t initially intend. A real-world example of an emergent property is a human heart: Its single cells don’t have the property of pumping blood, but the right configuration of cells produces a heart that keeps us alive. In the same way, our neurons don’t think, but their interconnected collection that we call a brain does, as you’re illustrating by reading these lines.
In computer systems, properties often emerge from the interactions among the moving parts that the system comprises. Let's consider an example. Imagine that you run a system with many services, all using a Domain Name System (DNS) server to find one another. Each service is designed to handle DNS errors by retrying up to 10 times. Similarly, the external users of the systems are told to retry if their requests ever fail.

Now, imagine that, for whatever reason, the DNS server fails and restarts. When it comes back up, it sees an amount of traffic amplified by the layers of retries—an amount that it wasn’t set up to handle. Thus, it might fail again and get stuck in an infinite loop of restarts, while the system as a whole is down. No component of the system has the property of creating infinite downtime, but with the components together and the right timing of events, the whole system might go into that state.

Although certainly less exciting than the example of consciousness I mentioned before, this property emerging from the interactions among the parts of the system is a real problem to deal with. This kind of unexpected behavior can have serious consequences on any system, especially a large one. The good news is that chaos engineering excels at finding issues such as this. By experimenting on real systems, often you can discover how simple, predictable failures can cascade into large problems. And once you know about them, you can fix them.

**Chaos engineering and randomness**

When doing chaos engineering, you can often use the element of randomness and borrow from the practice of *fuzzing*—feeding pseudorandom payloads to a piece of software to try to come up with an error that your purposely written tests might be missing. The randomness can be helpful, but once again, I would like to stress that controlling the experiments is necessary to be able to understand the results; chaos engineering is not just about randomly breaking things.

Hopefully, I’ve held your curiosity and now I’ve got your attention. In part 2, we will see how to do chaos engineering in four easy steps.

**Dig deeper**

Here are some resources from the Groundbreakers Developer Community.

- Deadlocks
- Simulating CPU spike
- Stackoverflow error
- Thread leak
Mikolaj Pawlikowski (@mikopawlikowski) is a software engineering team lead at Bloomberg LP in the United Kingdom and is author of *Chaos Engineering* (Manning Publications, 2021). He is also creator of the PowerfulSeal and Goldpinger open source projects.