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,” as I explored in “Introduction to chaos engineering, Part 1: Crash test your application.”

Chaos engineering experiments (chaos experiments, for short) are the basic units of chaos engineering. You do chaos engineering through a series of chaos experiments. Given a computer system and a certain number of characteristics you are interested in, you design experiments to see how the system fares when bad things happen. In each experiment, you focus on proving or refuting your assumptions about how the system will be affected by a certain condition.

For example, imagine you are running a popular website and you own an entire data center. You need your website to survive power cuts, so you make sure two independent power sources are installed in the data center. In theory, you are covered—but in practice, a lot can still go wrong. Perhaps the automatic switching between power sources doesn’t work. Or maybe your website has grown since the launch of the data center, and a single power source no longer provides enough electricity for all the servers. Did you remember to pay an electrician for a regular checkup of the machines every three months?
If you feel worried, you should. Fortunately, chaos engineering can help you sleep better. You can design a simple chaos experiment that will scientifically tell you what happens when one of the power supplies goes down. (For more dramatic effect, always pick the newest intern to run these steps.)

Repeat the following process for all power sources, one at a time:

1. Check that the website is up.
2. Open the electrical panel and turn the power source off.
3. Check that the website is still up.
4. Turn the power source back on.

This process is crude and sounds obvious, but let’s review these steps. Given a computer system (a data center) and a characteristic (survives a single power source failure), you designed an experiment (switch a power source off and eyeball whether the website is still up) that increases your confidence in the system withstand a power problem. You used science for the good, and it took only a minute to set up.

Before you pat yourself on the back, though, it’s worth asking what would happen if the experiment failed and the data center went down. In this overly crude demonstration, you would create an outage of your own. A big part of your job will be about minimizing the risks coming from your experiments and choosing the right environment in which to execute them. I’ll explain more about that later.

Look at Figure 1, which summarizes the process you just went through. Let me anticipate your first question: What if you are dealing with more-complex problems?

As with any experiment, you start by forming a hypothesis that you want to prove or disprove, and then you design the entire experience around that idea. When Gregor Mendel had an intuition about the laws of heredity, he designed a series of experiments on yellow and green peas, proving the existence of dominant and recessive traits. His
results didn’t follow the expectations, and that’s perfectly fine; in fact, that’s how his breakthrough in genetics was made. (He did have to wait a couple of decades for anyone to reproduce his findings and for mainstream science to appreciate it and mark it “a breakthrough.”)

Before I get into the details of good craftsmanship in designing chaos experiments, here’s an idea about what to look for.

Let’s zoom in on one of these chaos-experiment boxes from Figure 1 and see what it’s made of. Let me guide you through Figure 2, which describes the simple, four-step process to design an experiment like that.

1. **Observability**
   Make sure you can accurately measure the metrics you will rely on.

2. **Steady state**
   What is normal? Define the baseline to compare against.

3. **Hypothesis**
   Introduce a real-world-like event and hypothesize on what happens to your system.

4. **Run experiment**

   - **Correct**
     Great, nothing to see here.
   - **Wrong**
     Great, let’s fix it.

**Figure 2. The four steps of a chaos experiment**

Here’s a more detailed look at the steps.

1. You need to be able to observe your results. Whether it’s the color of the resulting peas, the crash test dummy having all limbs in place, your website being up, the CPU load, the number of requests per second, or the latency of successful requests, the first step is to ensure that you can accurately read the value of these variables.

   It’s great to be dealing with computers in the sense that you can often produce very accurate and very detailed data easily. I call this **observability**.

2. Using the data you observe, you need to define what’s **normal**. This is so that you can understand when things are out of the expected range.

   For instance, you might expect the CPU load on a 15-minute average to be below 20% for your application servers during the working week. Or you might expect 500 to 700 requests per second per instance of your application server running with four cores on your reference hardware specification. This normal range is often referred to as the **steady state**.

3. You shape your intuition into a hypothesis that can be proved or refuted, using the data you can reliably gather (observability). A simple example could be “Killing one of the machines doesn’t affect the average service latency.”
4. You execute the experiment, making your measurements to conclude whether you were right. And funnily enough, you like being wrong, because that's what you learn more from. Rinse and repeat.

The simpler your experiment, usually the better. You earn no bonus points for elaborate designs, unless that’s the best way of proving the hypothesis. Look at Figure 2 again, and let’s dive just a little bit deeper, starting with observability.

**Step 1: Ensure observability**

I like the word *observability* because it’s straight to the point. It means being able to reliably see whatever metric you are interested in. The key word here is *reliably*. Working with computers can spoil you; the hardware producer or the operating system already provides mechanisms for reading various metrics, from the temperature of CPUs to the fan’s RPMs to memory usage and hooks to use for various kernel events.

At the same time, it’s often easy to forget that these metrics are subject to bugs and caveats that the end user needs to consider. If the process you’re using to measure CPU load ends up using more CPU than your application, that’s probably a problem.

If you’ve ever seen an automobile crash test on television, you will know it’s both frightening and mesmerizing at the same time. Watching a 3,000-pound machine accelerate to a carefully controlled speed and then fold like an origami swan upon impact with a massive block of concrete is humbling.

But the high-definition, slow-motion footage of shattered glass flying around and seemingly unharmed (and unfazed) dummies sitting in what used to be a car just seconds before is not just for entertainment. Like any scientist who earned a white coat (and gray hair), both crash test specialists and chaos engineering practitioners alike need reliable data to conclude whether an experiment worked. That’s why observability, or reliably harvesting data about a live system, is paramount.

**Step 2: Define a steady state**

Armed with reliable data from the previous step (observability), you need to define what's normal so you can measure abnormalities. A fancier way of saying that is to define a **steady state**, which works much better at dinner parties.

What you measure will depend on the system and your goals about it. It could be “undamaged car going straight at 60 mph” or perhaps “99% of users can access the API in under 200 ms.” Often, what you measure will be driven directly by the business strategy.

It’s important to mention that on a modern Linux server, a lot of things will be going on, and you’re going to try your best to isolate as many variables as possible. Let’s take the example of the CPU usage of your process. It sounds simple, but in practice, a lot of things can affect your reading. Is your process getting enough CPU bandwidth, or is it being stolen by other processes (perhaps it’s a shared machine, or maybe a cron job updating the system kicked in during your experiment)? Did the kernel schedule allocate cycles to another process with higher
priority? Are you in a virtual machine, and perhaps the hypervisor decided something else needed the CPU more?

You can go deep down the rabbit hole. The good news is that often you are going to repeat your experiments many times, and some of the other variables will be brought to light, but remembering that all these other factors can affect your experiments is something you should keep in the back of your mind.

**Step 3: Form a hypothesis**

Now, here's the really fun part. This is where you shape your intuitions into a testable hypothesis—an educated guess of what will happen to your system in the presence of a well-defined problem. Will the system carry on working? Will it slow down and, if so, by how much?

In real life, these questions will often be prompted by incidents (unprompted problems you discover when things stop working), but the better you are at this game, the more you can (and should) preempt. These events can be broadly categorized as follows:

- External events (earthquakes, floods, fires, power cuts, and so on)
- Hardware failures (disks, CPUs, switches, cables, power supplies, and so on)
- Resource starvation (CPU, RAM, swap, disk, network)
- Software bugs (infinite loops, crashes, hacks)
- Unsupervised bottlenecks
- Unpredicted emergent properties of the system
- Virtual machine failure (such as the Java Virtual Machine)
- Hardware bugs
- Human error (pushing the wrong button, sending the wrong configuration, pulling the wrong cable, and so forth)

Simulating some of these incidents is easy (switch off a machine to simulate machine failure or take out the Ethernet cable to simulate network issues), while others will be much more advanced (add latency to a system call). The choice of failures to consider requires a good understanding of the system you are working on.

Here are a few examples of what a hypothesis could look like.

- On frontal collision at 60 mph, no dummies will be squashed.
- If both parent peas are yellow, all the offspring will be yellow.
- If 30% of the servers are taken down, the API continues to serve the 99th% of requests in under 200 ms.
- If one of the database servers goes down, the service-level objective (SLO) will still be met.

Now, it's time to run the experiment.

**Step 4: Run the experiment and prove (or refute) your hypothesis**

Finally, you run the experiment, measure the results, and conclude whether you were right. Remember, being wrong is fine—and much more exciting at this stage!
Everybody gets a medal in the following conditions:

- If you were right, congratulations! You just gained more confidence in your system withstanding a stormy day.
- If you were wrong, congratulations! You just found a problem in your system before your clients did, and you can still fix it before anyone gets hurt!

Remember that as long as this is good science, you learn something from each experiment.

**What chaos engineering is not**

Chaos engineering is not a silver bullet, and it doesn’t automatically fix your system, cure cancer, or guarantee weight loss. In fact, it might not even be applicable to your use case or project.

A common misconception is that chaos engineering is about randomly destroying stuff. I guess the name kind of hints at it, and Chaos Monkey, the first tool to gain internet fame in the domain, relies on randomness quite a lot. But although randomness can be a powerful tool, and it sometimes overlaps with fuzzing, you want to control the variables you are interacting with as closely as possible. Often, adding failure is the easy part; the hard part is to know where to inject it and why.

Chaos engineering is not just Chaos Monkey, Chaos Toolkit, PowerfulSeal, or any of the numerous projects available on GitHub. These are tools making it easier to implement certain types of experiments, but the real difficulty is in learning how to look critically at systems and predict where the fragile points might be.

It’s important to understand that chaos engineering doesn’t replace other testing methods, such as unit or integration tests. Instead, it complements them: Just as airbags are tested in isolation and then again with the rest of the car during a crash test, chaos experiments operate on a different level and test the entire system.

Every system is different, and you’ll need a deep understanding of your system’s weak spots to come up with useful chaos experiments. In other words, the value you get out of the chaos experiments is going to depend on your system, how well you understand it, how deep you want to go testing it, and how well you set up your observability shop.

Although chaos engineering is unique in that it can be applied to production systems, that’s not the only scenario that it caters to. A lot of content on the internet appears to be centered around “breaking things in production,” quite possibly because it’s the most radical thing you can do but, again, that’s not all chaos engineering is about—or even its main focus. A lot of value can be derived from applying chaos engineering principles and running experiments in other environments too.

Finally, although some overlap exists, chaos engineering doesn’t stem from chaos theory in mathematics and physics. I know: That’s a bummer. What chaos engineering is might be an awkward question to answer at a family reunion, so you better be prepared.

**Conclusion**
Chaos engineering is a discipline of experimenting on a computer system to uncover problems, often undetected by other testing techniques.

Much as the crash tests done in the automotive industry try to ensure that the entire car behaves in a certain way during a well-defined, real-life-like event, chaos engineering experiments aim to confirm or refute your hypotheses about the behavior of the system during a real-life-like problem.

Chaos engineering doesn’t automatically solve your issues, and coming up with meaningful hypotheses requires a certain level of expertise in the way your system works. Also, chaos engineering isn’t about randomly breaking things (although that has its place, too) but about adding a controlled amount of failure you understand.

Finally, chaos engineering doesn’t need to be complicated. The four steps I just covered, along with some good craftsmanship, should take you far before things get any more complex. Computer systems of any size and shape can benefit from chaos engineering.

Dig deeper

Here are some resources from the Groundbreakers Developer Community.

- Deadlocks
- Simulating CPU spike
- Stackoverflow error
- Thread leak

Mikolaj Pawlikowski

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.

Share this Page

[Facebook] [Twitter] [LinkedIn] [Email]