1 Into the world of chaos engineering

This chapter covers

- What chaos engineering is and is not
- Motivations for doing chaos engineering
- Anatomy of a chaos experiment
- A simple example of chaos engineering in practice

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 life-saving 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 book, you'll look at the why, when, and how of applying chaos engineering to improve your computer systems. And perhaps, who knows, save some lives in the process. What's a better place to start than a nuclear power plant?

1.1 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, or 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 as a whole 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 book 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" (Principles of Chaos Engineering, http://principlesofchaos.org/). In other words, it's a software testing method focusing on finding evidence of problems before they are experienced by users.

You want your systems to be reliable (we'll look into that), 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 like floods and earthquakes, which can take down entire datacenters, through power supply cuts, hardware failures, networking problems, resource starvation, race conditions, unexpected peaks of traffic, 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 (http://mng.bz/Xd7v) and the Israeli Beresheet (http://mng.bz/yYgB), both lost on lunar descent. And remember that even if you do everything right, more often than not, you still depend on other systems, and these systems can

fail. For example, Google Cloud,[1] Cloudflare, Facebook (WhatsApp), and Apple all had major outages within about a month in the summer of 2019 (http://mng.bz/d42X). If your software ran on Google Cloud or relied on Cloudflare for routing, you were potentially affected. That's just reality.

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 (more on that later), it's about much more than that—anything that helps us 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.

1.2 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, objectives, and agreements
- Testing a system (often complex and distributed) as a whole
- Finding emergent properties you were unaware of

Let's take a closer look at these motivations.

1.2.1 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, called Bookface, for sharing photos of cats and toddlers and checking on your high-school ex. Your business model might be to serve your users targeted ads, in which case 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 of these 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

[1] You can see the official, detailed Google Cloud report at http://mng.bz/BRMg.

unit of time. That cost includes the direct loss of business as well as various other, less tangible things like public image damage, that might be equally important. As a real-life example, Forbes estimated that Amazon lost $66,240 per minute of its website being down in 2013.[2]

Now, to quantify these risks, the industry uses *service-level indicators* (*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 a *service-level objective* (*SLO*), a tangible target that the engineering team can work toward. SLOs, in turn, can be legally enforced as a *service-level agreement* (*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: "the ratio of success responses to errors from our servers > 99.95% on average monthly"

To give you 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. We could imagine a different set of constraints, set on the trading API:

- SLI: 99th percentile response time
- SLO: 99th percentile response time < 25 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 milliseconds (ms) response time. Building a system like that might be difficult and expensive.

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 like *three nines five* or *three and a half nines* to mean 99.95%, although the latter is not technically correct (going from

[2] See "Amazon.com Goes Down, Loses $66,240 per Minute," by Kelly Clay, Forbes, August 2013, http://mng.bz/ryJZ.

99.9% to 99.95% is a factor of 2, but 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 milliseconds per day

How does chaos engineering help with these? To satisfy the SLOs, you'll engineer the system in a certain way. You will need to take into account the various sinister scenarios, and the best way to see whether the system works fine in these conditions is to go and create them—which is exactly what chaos engineering is about! You're effectively working backward from the business goals, to an engineering-friendly defined 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.

1.2.2 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, micro-benchmarking 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, but during which we rig the conditions to introduce the type of failure we expect to see, and measure that we still get the correct answer within the expected time frame. It's also worth noting, as you'll see in part 2, that even a single-process system can be tested using chaos engineering techniques, and sometimes that comes in really handy.

1.2.3 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. So it might fail again, and get stuck in an infinite loop restarting, 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 system as a whole 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 like 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 in order to try to come up with an error that your purposely written tests might be missing. The randomness definitely 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 had your curiosity and now I've got your attention. Let's see how to do chaos engineering!

1.3 Four steps to chaos engineering

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 datacenter. You need your website to survive power cuts, so you make sure two independent power sources are installed in the datacenter. 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 datacenter, 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 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 datacenter) 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 withstanding a power problem. You used science for the good, and it took only a minute to set up. *That's one small step for man, one giant leap for mankind.*

Before you pat yourself on the back, though, it's worth asking what would happen if the experiment failed and the datacenter went down. In this overly-crude-for-demonstration-purposes case, 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 to execute them. More on that later.

Take a look at figure 1.1, which summarizes the process you just went through. When you're back, let me anticipate your first question: What if you are dealing with more-complex problems?

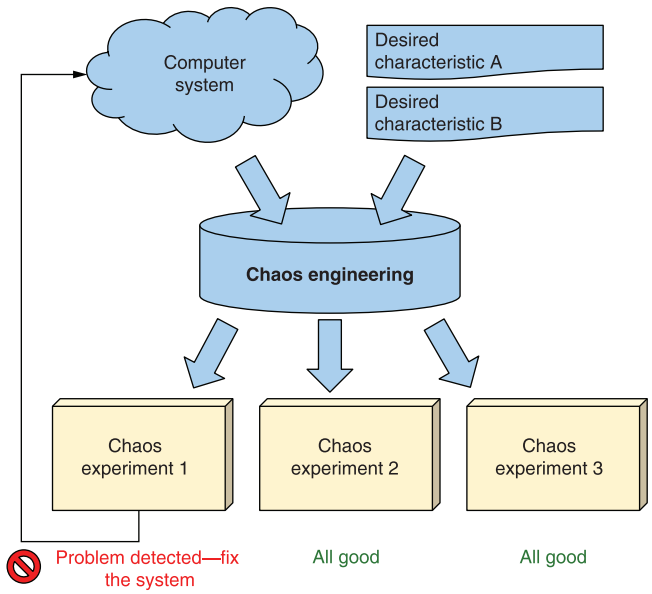

Figure 1.1 The process of doing chaos engineering through a series of chaos experiments

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.[3] We will be drawing inspiration from his experiments throughout the book, but before we get into the details of good craftsmanship in designing our experiments, let's plant a seed of an idea about what we're looking for.

Let's zoom in on one of these chaos experiment boxes from figure 1.1, and see what it's made of. Let me guide you through figure 1.2, which describes the simple, four-step process to design an experiment like that:

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 for these

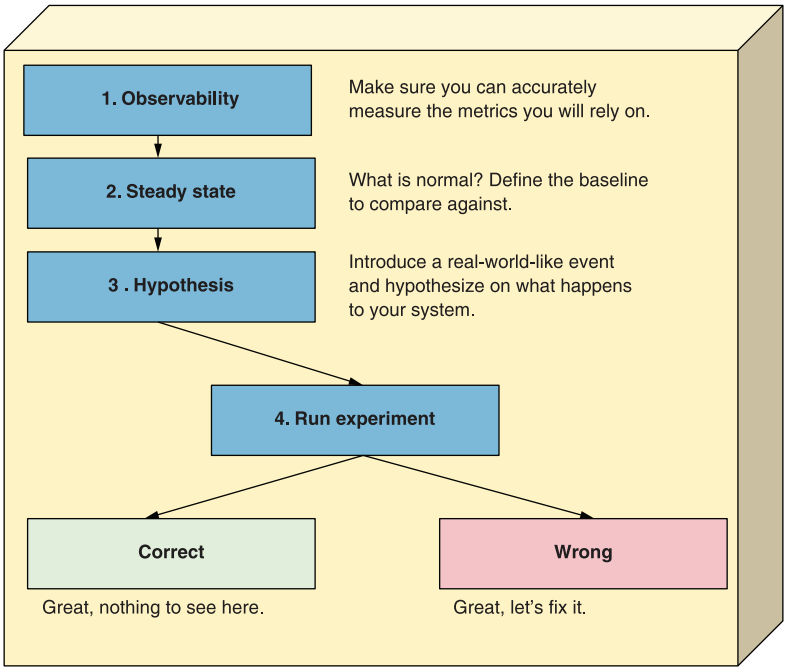

Figure 1.2 The four steps of a chaos experiment

[3] 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." But let's ignore that for now.

variables. We're lucky to be dealing with computers in the sense that we can often produce very accurate and very detailed data easily. We will 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 1.2 again, and let's dive just a little bit deeper, starting with *observability*.

1.3.1 Ensure observability

I quite 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 keyword here is *reliably*. Working with computers, we are often spoiled—the hardware producer or the operating system (OS) 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. But 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 take into account. 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 a crash test on television, you will know it's both frightening and mesmerizing at the same time. Watching a 3000-pound machine accelerate to a carefully controlled speed and then fold like an origami swan on 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 their white coat (and 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.

In this book, we're going to focus on Linux and the system metrics that it offers to us (CPU load, RAM usage, I/O speeds) as well as go through examples of higher-level metrics from the applications we'll be experimenting on.

Observability in the quantum realm

If your youth was as filled with wild parties as mine, you might be familiar with the double-slit experiment (http://mng.bz/MX4W). It's one of my favorite experiments in physics, and one that displays the probabilistic nature of quantum mechanics. It's also one that has been perfected over the last 200 years by generations of physicists.

The experiment in its modern form consists of shooting photons (or matter particles such as electrons) at a barrier that has two parallel slits, and then observing what landed on the screen on the other side. The fun part is that if you don't observe which slit the particles go through, they behave like a wave and interfere with each other, forming a pattern on the screen. But if you try to detect (observe) which slit each particle went through, the particles will not behave like a wave. So much for reliable observability in quantum mechanics!

1.3.2 Define a steady state

Armed with reliable data from the previous step (observability), you need to define what's normal so that 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 our users can access our API in under 200ms." Often, this 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 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, 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.

1.3.3 Form a hypothesis

Now, for the really fun part. In step 3, 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 it carry on working? Will it slow down? 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. Earlier in the chapter, I listed a few examples of what tends to go wrong. 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 (Java Virtual Machine, V8, others)
- Hardware bugs
- Human error (pushing the wrong button, sending the wrong config, pulling the wrong cable, and so forth)

We will look into how to simulate these problems as we go through the concrete examples in part 2 of the book. Some of them are 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 take into account 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 we take 30% of our servers down, the API continues to serve the 99th percentile of requests in under 200 ms.
- If one of our database servers goes down, we continue meeting our SLO.

Now, it's time to run the experiment.

1.3.4 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!

We'll spend some time on the good craftsmanship rules in the following chapters, including automation, managing the blast radius, and testing in production. For now, just remember that as long as this is good science, you learn something from each experiment.

1.4 What chaos engineering is not

If you're just skimming this book in a store, hopefully you've already gotten some value out of it. More information is coming, so don't put it away! As is often the case, the devil is in the details, and in the coming chapters you'll see in greater depth how

to execute the preceding four steps. I hope that by now you can clearly see the benefits of what chaos engineering has to offer, and roughly what's involved in getting to it.

But before we proceed, I'd like to make sure that you also understand what *not* to expect from these pages. Chaos engineering is not a silver bullet, and 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 (https://netflix.github.io/chaosmonkey/), 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 sometimes overlaps with fuzzing, you want to control the variables you are interacting with as closely as possible. More often than not, 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 (https://chaostoolkit.org/), PowerfulSeal (https://github.com/bloomberg/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 system as a whole.

This book will not give you ready-made answers on how to fix your systems. Instead, it will teach you how to find problems by yourself and where to look for them. Every system is different, and although we'll look at common scenarios and gotchas together, 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: bummer. Might be an awkward question to answer at a family reunion, so better be prepared.

With these caveats out of the way, let's get a taste of what chaos engineering is like with a small case study.

1.5 A taste of chaos engineering

Before things get technical, let's close our eyes and take a quick detour to Glanden, a fictional island country in northern Europe. Life is enjoyable for Glandeners. The geographical position provides a mild climate and a prosperous economy for its hard-working people. At the heart of Glanden is Donlon, the capital with a large population of about 8 million people, all with a rich heritage from all over the world—a true cultural melting pot. It's in Donlon that our fictitious startup FizzBuzzAAS tries really hard to *make the world a better place.*

1.5.1 FizzBuzz as a service

FizzBuzzAAS Ltd. is a rising star in Donlon's booming tech scene. Started just a year ago, it has already established itself as a clear leader in the market of FizzBuzz as a Service. Recently supported by serious venture capital (VC) dollars, the company is looking to expand its market reach and scale its operations. The competition, exemplified by FizzBuzzEnterpriseEdition (https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition) is fierce and unforgiving. The FizzBuzzAAS business model is straightforward: clients pay a flat monthly subscription fee to access the cutting-edge APIs.

Betty, head of sales at FizzBuzzAAS, is a natural. She's about to land a big contract that could make or break the ambitious startup. Everyone has been talking about that contract at the water cooler for weeks. The tension is sky-high.

Suddenly, the phone rings, and everyone goes silent. It's the Big Company calling. Betty picks up. "Mhm . . . Yes. I understand." It's so quiet you can hear the birds chirping outside. "Yes ma'am. Yes, I'll call you back. Thank you."

Betty stands up, realizing everyone is holding their breath. "Our biggest client can't access the API."

1.5.2 A long, dark night

It was the first time in the history of the company that the entire engineering team (Alice and Bob) pulled an all-nighter. Initially, nothing made sense. They could successfully connect to each of the servers, the servers were reporting as healthy, and the expected processes were running and responding—so where did the errors come from?

Moreover, their architecture really wasn't that sophisticated. An external request would hit a load balancer, which would route to one of the two instances of the API server, which would consult a cache to either serve a precomputed response, if it was fresh enough, or compute a new one and store it in cache. You can see this simple architecture in figure 1.3.

Finally, a couple of gallons of coffee into the night, Alice found the first piece of the puzzle. "It's kinda weird," she said as she was browsing through the logs of one of the API server instances, "I don't see any errors, but all of these requests seem to stop at the cache lookup." Eureka! It wasn't long after that moment that she found the problem: their code gracefully handled the cache being down (connection refused, no host, and so on), but didn't have any time-outs in case of no response. It was downhill

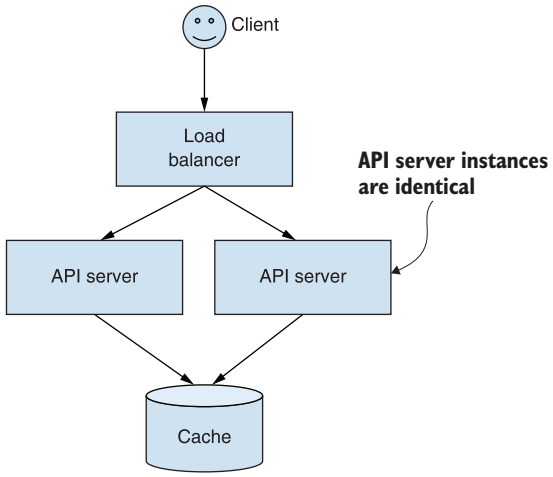

Figure 1.3 FizzBuzz as a Service technical architecture

from there—a quick session of pair programming, a rapid build and deploy, and it was time for a nap.

The order of the world was restored; people could continue requesting FizzBuzz as a Service, and the VC dollars were being well spent. The Big Company acknowledged the fix and didn't even mention cancelling its contract. The sun shone again. Later, it turned out that the API server's inability to connect to the cache was a result of a badly rolled-out firewall policy, in which someone forgot to whitelist the cache. Human error.

1.5.3 Postmortem

"How can we make sure that we're immune to this kind of issue the next time?" Alice asked, in what was destined to be a crucial meeting for the company's future.

Silence.

"Well, I guess we could preemptively set some of our servers on fire once in a while" answered Bob to lift up the mood just a little bit.

Everyone started laughing. Everyone, apart from Alice, that is.

"Bob, you're a genius!" Alice acclaimed and then took a moment to appreciate the size of everyone's eyeballs. "Let's do exactly that! If we could *simulate* a broken firewall rule like this, then we could add this to our integration tests."

"You're right!" Bob jumped out of his chair. "It's easy! I do it all the time to block my teenager's Counter Strike servers on the router at home! All you need to do is this," he said and proceeded to write on the whiteboard:

```
iptables -A ${CACHE_SERVER_IP} -j DROP
```

"And then after the test, we can undo that with this," he carried on, sensing the growing respect his colleagues were about to kindle in themselves:

```
iptables -D ${CACHE_SERVER_IP} -j DROP
```

Alice and Bob implemented these fixes as part of the setup and teardown of their integration tests, and then confirmed that the older version wasn't working, but the newer one including the fix worked like a charm. Both Alice and Bob changed their job titles to site reliability engineer (SRE) on LinkedIn the same night, and made a pact to never tell anyone they hot-fixed the issue in production.

1.5.4 Chaos engineering in a nutshell

If you've ever worked for a startup, long, coffee-fueled nights like this are probably no stranger to you. Raise your hand if you can relate! Although simplistic, this scenario shows all four of the previously covered steps in action:

- The *observability* metric is whether or not we can successfully call the API.
- The *steady state* is that the API responds successfully.
- The *hypothesis* is that if we drop connectivity to the cache, we continue getting a successful response.
- After *running the experiment*, we can confirm that the old version breaks and the new one works.

Well done, team: you've just increased confidence in the system surviving difficult conditions! In this scenario, the team was reactive; Alice and Bob came up with this new test only to account for an error their users already noticed. That made for a more dramatic effect on the plot. In real life, and in this book, we're going to do our best to predict and proactively detect this kind of issue without the external stimulus of becoming jobless overnight! And I promise that we'll have some serious fun in the process (see appendix D for a taste).

Summary

- Chaos engineering is a discipline of experimenting on a computer system in order to uncover problems, often undetected by other testing techniques.
- Much as the crash tests done in the automotive industry try to ensure that the car as a whole 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.
- Chaos engineering isn't about randomly breaking things (although that has its place, too), but about adding a controlled amount of failure you understand.
- Chaos engineering doesn't need to be complicated. The four steps we just covered, along with some good craftsmanship, should take you far before things get any more complex. As you will see, computer systems of any size and shape can benefit from chaos engineering.