Embrace chaos to achieve stability

Imagine this. You have built a website to sell your company's products. After a few months of hard labor, the application finally goes live. Of course, the application has been thoroughly tested. It all started with Unit Tests. First on the local machine and after the engineers filed a Pull Request a whole series of checks were executed. Each quality gate passed successfully and a fully automated pipeline successfully deployed the application on a cloud environment. But you went that extra mile. Performing load tests, security tests, pen tests and smoke tests. And finally, to make sure you have the least downtime possible when the sh*t hits the fan, you created and implemented failover scenarios and disaster recovery plans. Now you are all set. Let the sales begin!

Author René van Osnabrugge

And then... everything goes black. The datacenter is down, and the failover you carefully set up does not work as expected. And, after a few hours of stress, when the data-center has recovered, the way back does not go well.

This scenario is not something that only exists in a fantasy world. It is a real scenario. Things happen and you need to be prepared. And the truth is, you cannot prepare for everything. When you operate a business in the cloud (but also in your own datacenters) you need to embrace the fact that things can go wrong. The question is, how well can you deal with it.

Chaos Engineering

When Netflix moved to the cloud in 2011, they wanted to address the fact they lacked sufficient resiliency tests in production. To make sure they were prepared for unexpected failures in production, they created a tool called Chaos Monkey. This tool caused outages and breakdowns on random servers. By testing these "unexpected" scenarios they could validate and learn if their infrastructure could deal with, and recover from, failure in an elegant manner. Without meaning to, Netflix introduced a whole new practice. Chaos Engineering.

Breaking servers was one way to test this, but quickly other scenarios became relevant. Slow networks, unreliable messaging, corrupt data etc. Not much later, other tech companies, especially those running large scale and complex landscapes in the cloud, also adopted similar practices. This practice, where the mindset shifts from expecting stable production systems to expecting chaos in production, is called Chaos Engineering.

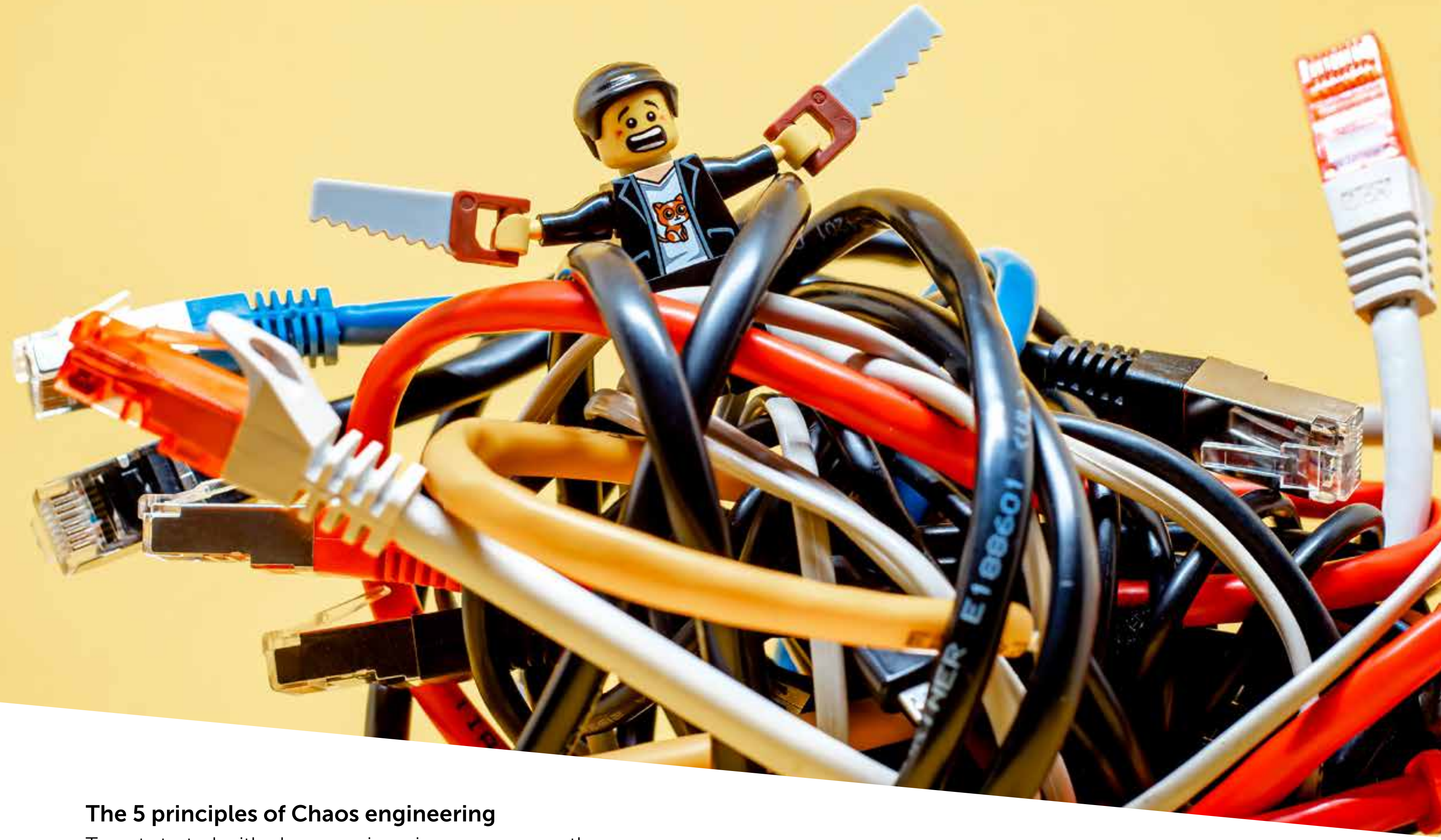
Chaos Engineering is a concept that uses hypotheses and experiments to validate the expected behavior of complex systems. This way you can grow confidence in the reliability and resilience of these systems.

Why Chaos Engineering?

Chaos Engineering lets you compare what you think will happen to what actually happens in your systems. You literally "break stuff " to learn how to build more resilient systems. Therefore you can look at Chaos Engineering as a test practice. But there are important differences. First of all Chaos Engineering, when done right, is also performed on production systems. Secondly, with Chaos Engineering you don't really test for failure. You test beforehand, and by conducting Chaos experiments you try to prove the assumptions you made in your test scenarios and architecture are actually valid and working.

With the rising complexity of our infrastructure, due to s oftware architectures like microservices, but also the "connected" systems we build nowadays, the traditional QA approach is not sufficient anymore. There is simply too much that can go wrong, and with the dynamic nature of the software and infrastructure stack this can be different every day. With Chaos Engineering it all starts with a hypothesis. And based on the hypothesis, you define and conduct experiments to prove that your hypotheses is correct.

Here is an example hypothesis, "When the external payment provider I use is unavailable, my customers get the option to pay afterwards and continue their checkout process".



The 5 principles of Chaos engineering

To get started with chaos engineering you can use the following simple plan. I will explain these steps in detail in the rest of this article.

Before we get started you should understand that Chaos Engineering is not something you can do on a rainy Sunday afternoon. Chaos Engineering needs careful planning and impact analysis. You need to understand what happens if your hypothesis is wrong. You also need to understand "the blast radius". In other words, what breaks if things do not work out as you planned. And linked to that, are there people available during the execution of the experiment, so they can jump in when things go not as planned?

The website [Principles of Chaos Engineering][1] describes 5 principles you should consider when doing Chaos Engineering:

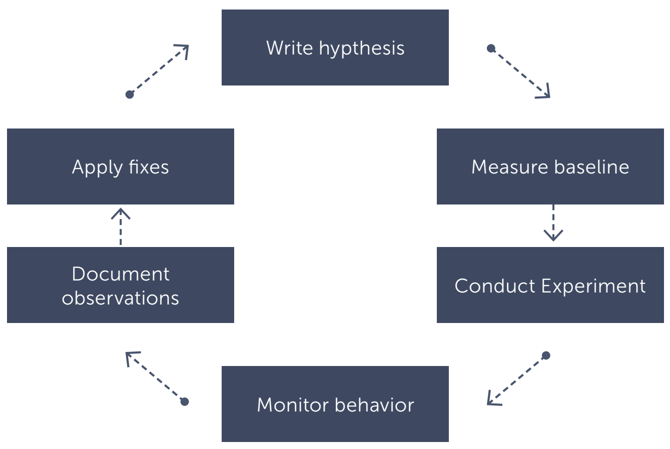
1. Build a hypothesis around steady state behavior
   This means you should focus on what is visible for the customer. Not the internal working of a system or things you can only influence when you know the inner workings. Focus on the steady state and the metrics that belong to a steady state.
2. Vary real world events
   Prioritize events based on expected frequency. Consider everything that can influence the system steady state. For example, disk failure, servers dying, or network outages.
3. Run experiments in production
   Simulation and sampling is great, but running on real world data and metrics is better. Try to run on production whenever possible. Of course, this requires careful planning and involvement of people. Usually this is done on so-called "game days", where people are ready for the "game". When you start with Chaos Engineering, it might be a better idea to validate your hypotheses on non-production systems. Start there, to get an idea what to expect and what you should measure. Production introduces an extra level of complexity and control because you need to make sure your users are not impacted.
4. Automate experiments to run continuously
   As with almost everything in DevOps, automation is key. Running experiments and gathering metrics is time intensive and hard work. Make sure you automate experiments so you can run them repeatedly.
5. Minimize blast radius
   Experimenting in production has the potential to cause unnecessary customer pain. So be mindful of that. Make sure there is room in your error budget or prepare for some issues. There must be an allowance for negative impact but keep the fallout of experiments minimal.

How does it work?

Chaos engineering involves going through a number of steps. These steps are followed for each new experiment. As I described before, it is important to plan this carefully. Because many of the chaos experiments are executed on production systems, you can easily break things that have customer impact. Often companies choose to organize so-called Game days. On these days people know that chaos experiments will be executed and can be on standby or be extra careful to monitor the systems for strange behavior.

[1] PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering

When running chaos experiments you can follow this structure:



Write a hypothesis
With chaos engineering, it starts with a hypothesis. This is important! It is not a test. For example, the hypothesis "The payment service should respond" is not a valid hypothesis. This is something you should already have tackled in your test suite. Chaos Engineering is about making sure your application becomes more resilient. You should already be quite certain your system can deal with unknown situations and your hypothesis should build on that. For example. "When the payment service goes down, we offer our customers an alternative way of payment". Think about the user. How can the user continue its journey with the least impact . A good example that Netflix uses when the login functionality stops working, is they offer services for free, without logging in. That way, users can still utilize the service.

Measure baseline behavior
Before you run any experiment, you should be aware of the baseline behavior. How does your system normally respond? In other words, can you recognize anomalies? You should have a good idea of the baseline because otherwise you may draw the wrong conclusions. For example, if you run an experiment to prove your response times will stay the same as "usual", you should know what usual is. Maybe this varies throughout the day due to traffic on your site. If you run an experiment in that timeframe, you might see strange things that are caused by factors other than your experiment.

When you think about creating the baseline, you should think of metrics and user metrics that are important to look at in the light of the experiment and hypothesis you are working on. Not everything is relevant at the same time.

Conduct experiment
When you created the hypothesis and baseline, you can start running an experiment. Running an experiment is causing the behavior that could disproof your hypothesis. Slowing down traffic, bringing a service down, shutting down or killing containers etc. There are several tools that can help you in running Chaos experiments. Many of them are targeted at virtual machines or a Kubernetes cluster and cause havoc on the infrastructure layer. Of course, you can also write your own scripts or tools to help you with your experiments.

Some examples of tools you can use are:
> Gremlin[2]
> Chaos Toolkit[3]
> Chaos Mesh[4]
> Azure Chaos Studio[5]

Monitor the resulting behavior
When you conduct the experiment, it is time to look at the metrics again. What do you see? Do you see the expected behavior of your system? Is the hypothesis valid? When you see the system does not behave as expected, try to gather as much information as possible why this is the case. Also, make sure you keep the blast radius and real user impact in focus.

Document the process and observations
After the experiment is complete, you have either proved or disproved your hypothesis. Make sure you document the process you executed, especially when you found that your hypotheses failed. Make sure you document your learnings. Consider performing a blameless learning review to find out what happened and document the learning review for future use.

Identify fixes and apply them
When you find your hypotheses did not work, apply the necessary fixes and automate the experiment. Make sure you can run the experiment multiple times, maybe even on a schedule. Systems change, and environments change, and you need to validate your hypotheses over and over again.

How can I get started with Chaos Engineering?

Getting started with Chaos Engineering is something you can do any time, as long as you take the user impact and blast radius into account. A common way to introduce chaos is to deliberately inject faults that cause system components to fail. The goal of Chaos Engineering is to create a more resilient and reliable application. With Chaos Engineering practices, you need to test and validate your application is indeed more resilient. Architectural patterns like circuit breakers, failover, and retry can help to make your application more robust. Then, after you have built your application, you need to observe, monitor, respond to, and improve your system's reliability under adverse circumstances. For example, taking dependencies offline (stopping API apps, shutting down VMs, etc.), restricting access (enabling firewall rules, changing connection strings, etc.), or forcing failover (database level, Front Door, etc.), is a good way to validate that the application can handle faults gracefully.

[2] https://www.gremlin.com/
[3] https://chaostoolkit.org
[4] https://chaos-mesh.org/
[5] https://azure.microsoft.com/en-us/services/chaos-studio/

It is important to start small. Start by defining a hypothesis and a very small experiment and go through the different steps that I described above. To define your first hypothesis, you should look at things you expect to go right but that you never actually look at. A good source of inspiration is a keynote of Adrian Cockroft[6]. In this keynote, he explains some basic things that go wrong. For your convenience, I have listed a number of these categories and things that can go wrong:

Infrastructure Failures	
Device Failures	Disk, power supply, cabling, circuit board, firmware
CPU failures	Cache corruption, Logic bugs
Datacenter failures	Power, Connectivity, cooling, fire, flood, wind, earthquake
Internet Failures	DNS, ISP, internet routes
Software stack Failures	
Time Bombs	Counter wrap round, memory leak
Date bombs	Leap year, leap second, epoch
End of unix time	
Expiration	Certificates timing out
Revocation	License or account shut down by supplier
Exploit	Security failures e.g. Heartbleed
Language bugs	Compiler, interpreter
Runtime bugs	JVM, Docker, Linux, Hypervisor
Protocol problems	Latency dependent or poor error recovery
Application Failures	
Time bombs (in application code)	Counter wrap around, memory leak
Date bombs (on application code)	Leap year, leap second, epoch, Y2K
Content bomb	Data dependent failures
Configuration	Wrong config or bad syntax
Versioning	Incompatible versions
Cascading failures	Error handling bugs
Cascading overload	Excessive logging, lock contention, hysteresis
Retry storms	Too many retries, work amplification, bad timeout strategy
Operations failures	
Poor capacity planning	
Inadequate incident management	
Failure to initiate incident	
Unable to access monitoring dashboards	
Insufficient observability of systems	
Incorrect corrective actions	

Summary

Chaos Engineering is fairly new to many people. Although it has existed for several years, it is not yet embraced by the broad audience. That is a shame because chaos engineering can really help build more resilient systems. By defining hypotheses and conducting experiments to prove your hypotheses you can test your system to deal with unexpected situations. There are many small experiments you can execute on your system, so getting started should be very simple. However, always take the potential user impact and blast radius into account and carefully plan your game day. </>

René van Osnabrugge
ALM, DevOps, Continuous Delivery, Initiator and Inspirator

xpirit.com/rene

[6] https://www.youtube.com/watch?v=cefJd2v037U