**Test for Resilience**

Resiliency is the ability of a system to recover from infrastructure or service disruptions and continue to function as normal.

[TEST FOR RESILIENCE](https://dta-images.discoverfinancial.com/articles/test-for-resilience/static/Test_for_Resilience.pptx)

**Overview**

In a distributed environment, systems failure is unexpected and unavoidable. Responding to unexpected failures is the hardest problems to solve. The problem is more involved than writing code to handle failures. What happens when the machine where the application is running fails? How do you detect this application failure and how soon recover your application.

Resiliency is the ability of a system to recover from infrastructure or service disruptions and continue to function as normal. Resilience is *not* about avoiding failures but accepting the fact that failures will happen and responding to them in a way that avoids downtime or data loss. The goal of resiliency is to return the application to a fully functioning state after a failure.

This article discuses resilience, and, specifically, how to:

* Build confidence in your system by identifying unknown failures during turbulent conditions and fixing them before they happen in production
* Measure the failure during experimentation and improve and validate your monitoring and observability
* Improve skills by practicing handling unexpected problems, which, in turn, improves recovery time

**Read this first:**

Understanding the concepts in the following articles will help you understand the topics covered in this article.

* [Site Reliability Engineering](https://dta.discoverfinancial.com/articles/dojo_operate_site-reliability-engineering)
* [Observability](https://dta.discoverfinancial.com/articles/sw-observability)
* [Circuit Breaker](https://dta.discoverfinancial.com/code-patterns/circuit-breaker-design-pattern)

**Why test resilience?**

Resilience isn't something that is achieved and then never considered again. It must be tested, and ideally not just when things happen to break.

Testing resilience helps you:

* Proactively identify the system weaknesses that could potentially lead to outages that harm customers
* Fix broken systems whose ripple effects would spread, ultimately even causing a [Denial of Service (DoS)](https://en.wikipedia.org/wiki/Denial-of-service_attack)
* Prepare and educate the engineering team for actual failures and how to avoid happening at midnight
* Create a cost-efficient failover plan that saves money in the event of a failure
* Prepare better incident reporting and application performance monitoring after chaotic experiments

**Test resilience with chaos engineering**

Chaos testing or chaos engineering is the systematic way to test the resilience of the system.

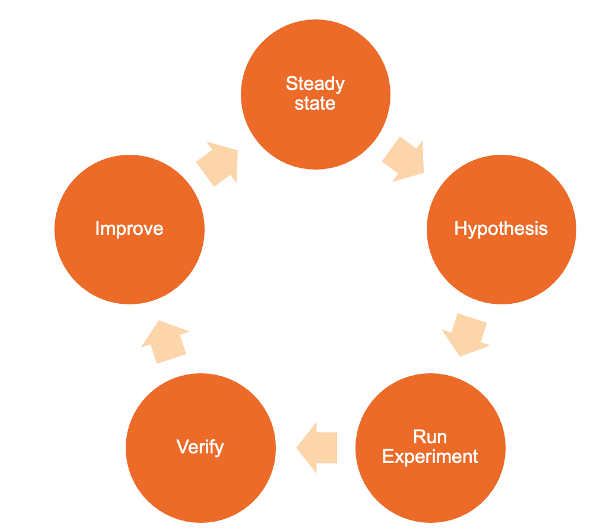
**"Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production."** -[principlesofchaos](http://principlesofchaos.org/)

It is important to understand that chaos engineering is not about doing random tests and breaking things without a purpose in production. Chaos engineering is a systematic and well-planned, structured experiment to build confidence in the system's capability to recover quickly from failures.

**Principles of chaos engineering**

As the image below shows, chaos engineering is a continual circle of:

* having your application or system in a steady state
* creating a hypothesis about what would happen if you changed something
* running experiments to see what is working and what is breaking
* verifying changes in behavior
* improving the application or system



**Reach a steady state for your system or application**

Ensuring your application or system in a steady state lets you understand the behavior of the system in normal conditions. The key here is to measure operational metrics and customer experience rather than focusing on the internal attributes of the system like CPU, memory, etc.

The system behavior should have a predictable pattern but vary significantly when failure in the system is introduced.

If you can't properly measure your system, you can't monitor drifts in the steady state. Invest in measuring everything, from the network, machine, application, and people's level.

**Create a hypothesis**

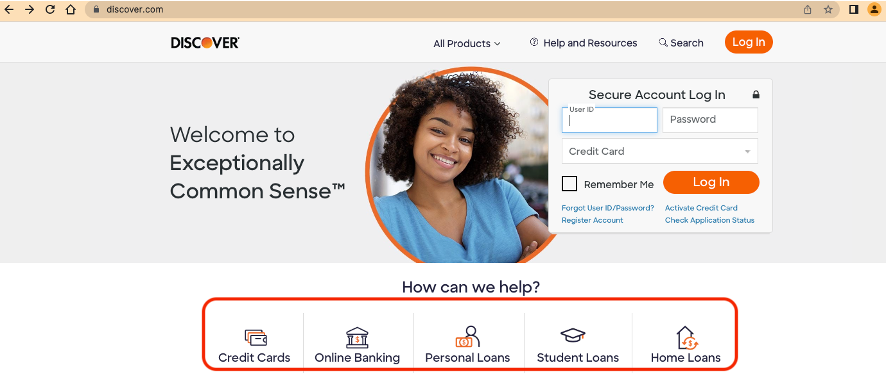
Create a hypothesis about your steady state by asking "What if?" Gather your entire team, including the product owner, lead, dev and test engineers, and architects to craft your hypothesis.

Ask everyone to write their own answer to the "What if?" on a piece of paper, in private. What you'll see is that, most of the time, everyone will have a different hypothetical answer, and often, some of the team will not have thought about the hypothesis at all.

Spend time understanding how the system should behave during the "What if...?"

Experiment on parts of the system you believe are resilient, not the ones you know will break you. After all, that's the whole point of the experiment.

For example, on the ABC.com site, what if the Product catalog fails to load on the main page?



What should happen on the API? Should the failing dependency continue to receive requests every time a user goes to the home page? Or should the API circuit break the failing dependency?

Make hypothesis on parts of the system you believe are resilient. Start with a small hypothesis like the above example and build more complex hypothesess.

Some additional "What ifs" to consider:

* What if caching fails?
* What if database query takes time to execute?
* What if server crashes?
* What if entire region out?

**Run experiments**

Chaos engineering is not breaking things randomly in production. Instead, it's a journey of learning by breaking things in a structured, systematic way. You can use chaos testing through well planned experiments to build confidence in your system to withstand turbulent conditions. Building confidence is the key to success.

An important aspect of the experiment phase is understanding the potential blast radius of the experiment and the failure you're injecting - and minimizing it.

One of the better ways to conduct experiments is to use canary deployments that reduce the risk of failure. In canary deployments, when new versions of the applications enter production, you gradually roll out the change to a small subset of users, see what happens, make chagnes, and then slowly roll it out to everybody.

Perform your experiment in the following order

1. Known Knowns
2. Known Unknowns
3. Unknown Knowns
4. Unknown Unknowns

The following image illustrates this concept:



So, in this phase:

* Pick one hypothesis.
* Scope your experiment carefully.
* Identify the relevant metrics to measure.
* Notify the organization.
* Run your experiment.

**Verify and document the results**

Monitor the experiment to record any nuances in the system behavior.

Document the analysis of every single experiment. Invest time to deep dive on the failures, understand the reason(s) why failure happened, and to prevent a similar failure in the future.

One of the most important guidelines for writing good documentation is to be blameless and avoid identifying individuals by name. This cab be challenging in an environment that doesn't encourage such behavior and that doesn't embrace failure.

Document by answering the following questions:

* What happened? (timeline)
* What was the impact to our customers?
* Why did the error occur?
* What did you learn?
* How will you prevent in the future?

**Improve your system**

Prioritize to fix the findings of your experiments over new feature development. Fixing current issues is more important than continuing the development of new features. Automate the experiments to run your experiment continuously.

**Maturity model**

Before assessing your maturity, try to answer the following questions:

* Do you understand the behavior of your system in "normal" condition?
* Have great observability of your system/application? (Able to measure operational metrics)
* Have a list of all possible hypothetical failure scenarios captured?
* Have your risk tolerance for your production environment?
* Know your blast radius and know how to minimize it?
* Have mechanisms to stop your experiment and roll back?

**Where does your team fit in the model?**

Read through the following levels of the maturity model to understand where your team fits with regards to building resilience into your systems.

**Walk**

* Team does not understand the behavior of the system in "normal" condition.
* Team does not have great observability of the system/application? (Able to measure operational metrics)
* Team does not have list of all possible hypothetical failure scenarios.
* Team does not have risk tolerance for production environment.
* Team does not know blast radius and how to minimize it.
* Team does not have mechanism to stop experiment and roll back.

**Run**

* Team understands the behavior of your system in "normal" condition.
* Team understands to have observability of the system/application? (Able to measure operational metrics)
* Team understands to create all possible hypothetical failure scenarios.
* Team understands to have risk tolerance for production environment.
* Team understands the blast radius and know how to minimize it.
* Team understands to create mechanism to stop experiment and roll back.

**Fly**

* Team understands the behavior of the system in "normal" condition.
* Team has great observability of the system/application? (Able to measure operational metrics)
* Team has a list of all possible hypothetical failure scenarios.
* Team has risk tolerance for production environment.
* Team knows blast radius and knows how to minimize it.
* Team has mechanisms to stop an experiment and roll back.

**Additional resources**

* [Chaos engineering @ ABC](https://dta.discoverfinancial.com/articles/chaos-gremlin)
* [AWS Fault Injection Simulator](https://aws.amazon.com/fis/) is a toolkit that helps when conducting chaos experiments on applications deployed in AWS.
* [Chaos engineering Wikipedia](https://en.wikipedia.org/wiki/Chaos_engineering)