Gremlin

Chaos Engineering Adoption Guide

Your guide to implementing Chaos Engineering within your organization

Introduction

By following this guide, you'll successfully increase your organization's reliability with minimal effort and risk.

Welcome to Chaos Engineering with Gremlin!

This document will serve as your guide to implementing Chaos Engineering and Gremlin within your organization. From educating your team on the principles of Chaos Engineering to running automated experiments, we'll walk through each stage of the adoption process in order to ensure a smooth and successful rollout. We'll explain how to:

- Prepare your organization for Chaos Engineering
- Deploy Gremlin to your systems
- Create and execute chaos experiments safely and confidently
- Integrate Gremlin into the workflow of your engineering teams
- Automate your chaos experiments as part of your CI/CD pipeline

At the end of this guide, Gremlin won't just be another bookmark in your engineers' browsers, but an integral part of your organization's culture and everyday DevOps practices. We also provide a checklist at the end of each chapter to help you track your progress.

This guide assumes that your organization has already purchased Gremlin and understands the fundamentals of Chaos Engineering. If your organization is unaware of Chaos Engineering and the benefits it can provide, please read our guide to championing Chaos Engineering within your organization. If your organization hasn't yet purchased Gremlin, request a demo.

Prerequisites

Before starting with Gremlin, there are steps we need to take in preparation for running chaos experiments.

The overarching goal of Chaos Engineering is to improve the reliability of our applications and systems by testing how they handle failure. To do this, we need to take a structured and well-organized approach with clearly defined objectives and key performance indicators (KPIs). Running random experiments without any direction or oversight won't yield actionable results and will put our systems at unnecessary risk. To achieve this, we'll move through the following steps:

1. Define the objectives and KPIs we want to achieve through chaos experimentation
2. Select and prioritize the system(s) we want to target for initial experimentation
3. Identify and track metrics that will help us track progress towards our objectives
4. Prepare our teams for running chaos experiments

1. Define Your Reliability Objectives and KPIs

To meet the high-level goal of improved reliability, we need to guide our Chaos Engineering adoption process using more granular objectives. These objectives will likely vary between business units.

The **executive team** will be more interested in long-term business-oriented objectives, such as:

- Ensuring successful new product launches
- Avoiding costly and damaging downtime, which can drive down customer satisfaction and product usage
- Reducing the number of delayed releases due to reliability and stability problems
- Reducing the amount of maintenance work, rework and technical debt
- Increasing engineering productivity and change velocity

The **engineering team** will be more interested in short-term operational objectives, such as:

- Limiting the amount of downtime and time spent firefighting
- Reducing the number of on-call incidents and late night pages
- Reducing the rate of failures introduced by changes
- Increasing the speed of application migrations by proactively testing reliability

When starting your Chaos Engineering journey, consider which objectives are important to your organization. These will guide the adoption process and allow you to track your progress towards greater reliability. Note that this list is by no means exhaustive, and should be taken more as a starting point.

Identifying Reliability Objectives Using Past Incidents

If you're not sure which objectives to track, think back to any incidents your organization experienced over the past year. Ask questions such as:

- What was the nature of the incident? Was it a hardware failure? A problem with a downstream dependency? An accident caused by a team member?
- What systems were affected? Did they become temporarily unavailable or go completely offline? How long did it take to restore them to normal service?
- What was the impact on customers? Did they experience downtime or data loss? Did your customer support team see a corresponding increase in tickets?
- How quickly was your response team alerted to the incident? Was it detected by an automated monitoring solution, or did someone notice it? Did you have an on-call crew ready to go, or did you need to scramble a response team?
- How long did it take to resolve the incident? Did you have any automated systems in place that resolved or mitigated the failure? Did you have to restore from a backup?
- How was the problem resolved? Was it an ad-hoc fix that only exists in production, or did you merge the fix back into your codebase? Did you document the fix?
- What organizational changes were made as a result? Was anyone blamed for the incident? Were policies implemented to prevent the problem in the future?

Focus on the questions that were either left unanswered, or that your teams struggled to answer. For example, if we had an automated monitoring system in place that failed to detect the incident and notify the on-call team, one of our objectives might be to test our alerting thresholds by reproducing the conditions leading up to the incident. Not only will this address the problem, but it teaches us more about our monitoring system and how to optimally tune our thresholds.

2. Track Progress Towards Your Objectives

To determine whether our fixes are directing us towards our objectives, we need to measure and track the internal state of our systems. We can do this using observability, which is the measure of a system's internal behaviors based on its outputs. Gathering observability data lets us quantify different attributes of our systems such as performance, capacity, and throughput. This data is useful not just for understanding how our systems are operating in the current moment, but we can compare it against historical data to see how our systems have changed over time.

For example, if we implement a change that affects application performance, we can compare our metrics from before and after the change to quantify the impact on latency, throughput, etc. This helps us stay aligned with our objectives, avoid creating wasted effort, and provide the best value to the business.

> *Without observability, you don't have 'chaos engineering'. You just have chaos."*
>
> **Charity Majors, Closing the Loop on Chaos with Observability, Chaos Conf 2018**

Collecting the right metrics can be challenging. Modern systems are complex, and collecting every bit of observability data can quickly lead to information overload. Even just tracking the four golden signals—latency, traffic, error rate, and resource saturation—can be overwhelming in large systems.

Focus on collecting metrics that relate directly to your objectives. For **business-oriented objectives**, these might include metrics related to business efficiency and customer satisfaction, such as:

- Net Promoter Score (NPS) and Social Promoter Score (SPS): how satisfied customers are with our service, and how likely they are to promote us.
- Meaningful availability: how our customers perceive our service's availability.
- Recovery Time Objective (RTO) and Recovery Point Objective (RPO): how long it takes our IT and business activities to recover after a disaster, and the amount of data we can tolerate losing in a failure.

For **engineering-oriented objectives**, focus on metrics related to application and infrastructure health, such as:

- Service Level Objectives: the target level of availability over time for our systems. These help define the standard of service that our customers should expect, and are therefore important guides for both the engineering team and the business.
- Golden signals: latency, traffic, error rate, and resource saturation.
- Mean Time Between Failures (MTBF): the average amount of time between outages. A low MTBF indicates that our systems experience frequent failure.
- Mean Time to Detection (MTTD): the average gap in time between when an incident actually starts and when we detect it. A low MTTD means we have an effective monitoring strategy in place and are quickly made aware of problems.
- Mean Time to Resolution (MTTR): the average gap in time between the start of an incident and when it's resolved. A low MTTR means our teams are effective at detecting and resolving incidents.

3. Prioritize Your Targets

Now that we know our objectives and KPIs, we need to identify the parts of our infrastructure that directly impact those KPIs. This is where we'll run our chaos experiments using Gremlin. By running experiments on these systems and seeing how our KPIs change, we can plan and implement fixes more effectively.

If your organization experienced an incident within the past year, start by trying to replicate that incident. Identify the systems that were involved (or their test/staging equivalents), and start thinking about the conditions that caused the incident. If you had multiple incidents, choose the one that affected the greatest number of customers, generated the most after-hour alerts, or was the most expensive to resolve.

We also recommend targeting systems that are critical to your business. As an example, an unreplicated database is a critical component since a failure risks bringing down any applications that depend on it. Running chaos experiments on essential infrastructure might seem dangerous, but it will provide far more value than experimenting on non-essential components. Not only will it help us find and fix issues with these systems faster, but it demonstrates the value that Chaos Engineering provides to the business. You can always experiment on non-essential systems to build confidence, but this won't provide as much value.

If you're still not sure where to start, use our reliability calculator to grade the reliability of each of your services, and choose the service that results in the lowest grade.

4. Prepare Your Teams

Much like DevOps, the practice of Chaos Engineering is rooted in the engineering team, but involves all other parts of the organization. When we inject failure into a system, we risk impacting everyone who uses that system, whether it's internal users or customers.

As an example, consider a relatively simple experiment: consuming 1 GB of RAM on a single host in a cluster. This might seem fairly harmless at first, but then we find out that this exceeded an alerting threshold, triggering a wave of notifications that sends our SREs into response mode. And if this is a Kubernetes cluster, our innocuous experiment could prevent applications from deploying onto the node, causing a completely unintended kind of failure.

Before we start performing chaos experiments, we want to help educate our teams on what Chaos Engineering is, its principles, and its risks. We want to involve the owners of the systems that we're testing in helping design and execute experiments, as well as be available in case the systems fail. As our practice expands throughout the organization, we want to inform non-engineering teams who may be affected by experiments including support teams, executive teams (especially the CTO), and developer advocacy teams.

If you're unsure of how to go about this process, read our guide to championing Chaos Engineering within your organization. You'll find additional tips on how to discuss Chaos Engineering with non-technical users and get their involvement.

Checklist

- [ ] List the technical and business objectives you want to achieve with Chaos Engineering
- [ ] Determine which KPIs and metrics you'll need to track progress towards your objectives
- [ ] Identify and prioritize the target(s) you want to attack
- [ ] Create a recovery plan in case of unexpected failure
- [ ] Inform your organization about Chaos Engineering and your planned experiments

Running Your First Experiment

Now that you've laid the groundwork, it's time to start planning your first experiment. As we've established, running a chaos experiment isn't as simple as running a random attack against a random system. Experimentation follows a thoughtful, controlled, scientific process.

When developing an experiment:

1. Start with a *hypothesis* stating the question that you're trying to answer, and what you think the result will be. For example, if your experiment is to test whether your web server can handle increased load, your hypothesis might state that "as CPU usage increases, request throughput remains consistent."

2. Define your *blast radius*. The blast radius includes any and all components affected by the test. A smaller blast radius will limit the potential damage done by the test. We strongly recommend you start with the smallest blast radius possible. Once you are more comfortable running chaos experiments, you can increase the blast radius to include more components.

3. *Monitor* your infrastructure. Determine which metrics will help you reach a conclusion about your hypothesis, take measurements before you test to establish a baseline, and record those metrics throughout the course of the test so that you can watch for changes, both expected and unexpected.

4. *Run the experiment*. Make sure you have a way to stop the experiment and revert any changes it introduced before you begin the experimentation process.

5. Analyze the data and form a *conclusion*. Does it confirm or reject your hypothesis? Use the results to address failure points in your systems and refine your experiment.

Gremlin provides Scenarios, which let you record your hypothesis, metrics, and results in the Gremlin web app. Additionally, you can halt any active attacks in case of a problem.

Structuring Chaos Experiments Around GameDays

A GameDay is a dedicated period of time for teams to focus on running a chaos experiment, observe the results, and determine which actions to take in order to improve reliability. GameDays allow teams to run tests, respond to any issues that arise, and provide insights or feedback in a collaborative setting. GameDays typically run for 2–4 hours and involve the team of engineers responsible for the system being tested, but often include members from both sides of the organization.

GameDays provide a structured approach to running chaos experiments. While they're often used to run large-scale tests impacting production systems, teams that are just starting out should follow this same approach to familiarize themselves with the process. Not only does it allow your teams to practice collaborative Chaos Engineering, but it encourages the use of GameDays as a regular practice.

The basic steps involved in running a GameDay are:

- Identify your target systems (see the Prioritize Your Targets section above)
- Schedule a time and location for the GameDay
- Scope out your chaos experiments
- Execute your experiments
- Recap the experience and convert the results into action items

To learn more about GameDays, read our tutorial on How to Run a GameDay.

Deploy Gremlin

Next, we'll install the Gremlin client onto our target systems. This will let us target these systems for attack using the Gremlin web app, the Gremlin command-line client, and the Gremlin API. See our documentation for information on how to install the Gremlin client onto your infrastructure and applications.

For your first experiment, we recommend only installing the Gremlin client onto the systems that you plan on testing as part of your experiment. While Gremlin gives you the ability to halt any attacks, we want to reduce the risk of running an attack on the wrong systems as much as possible. Once you become more comfortable with using Gremlin, you can deploy the client to the rest of your infrastructure.

To verify that your systems are connected to the Gremlin Control Plane, log into your account at https://app.gremlin.com and navigate to the Clients page. This will list each client in your Gremlin Control Plane. If you are having trouble getting the client to start or connect, see our infrastructure troubleshooting guide or contact Gremlin support through the Gremlin web app.

Run the Experiment

Now that you've outlined your experiment and deployed Gremlin to your systems, we'll walk you through running an experiment. There are four actions we'll focus on: establishing a baseline, running attack(s), analyzing the results, and repeating the experiment after implementing a solution.

1. Establish a Baseline

To measure how our systems change during an experiment, we need to understand how they behave now. This involves collecting relevant metrics from our target systems under normal load, which will provide a baseline for comparison. Using this data, we measure exactly how much our systems change in response to the attack.

If Attack Visualizations is enabled in the Gremlin web app, the Gremlin UI will automatically chart CPU utilization for the target system(s) when running a CPU attack. This will also record utilization metrics immediately before and after the attack. For other metrics, use your monitoring solution of choice.

2. Create and Run the Experiment in Gremlin

The Gremlin web app provides two ways of running an experiment: attacks and scenarios.

An attack is a method of injecting failure into a system, such as consuming compute resources, shutting down a system, or dropping network packets. Attacks can be scheduled or executed ad hoc.

A Scenario is a collection of attacks that can be saved along with a title, description, hypothesis, and detailed results. Scenarios execute attacks in sequence, giving you greater control over how your attacks are executed and allowing you to recreate complex failures. You can use this to repeat the same set of experiments and observe how your systems behave over time. In addition to running Scenarios ad hoc, you can schedule Scenarios to run automatically. After each Scenario run, you can record your observations for comparison with other runs.

Once you've created your attack or Scenario and selected your target(s), click the Run Attack or Run Scenario button to start the attack. Be sure to record your metrics as the attack runs, and if possible, monitor your relevant KPIs in real-time. How is the target responding? Is it matching your hypothesis, or behaving differently than you anticipated?

If the attack is causing unexpected problems with your systems (e.g. they've become unresponsive), stop the test using the Halt button in the top-right corner of the Gremlin web app. It may take several seconds for the agent to receive the signal and stop the attack. Make sure to record this result, since even a failed chaos experiment is an important indicator of reliability. In addition, if a Gremlin client loses connection to the Gremlin Control Plane, it will automatically trigger a safety mechanism that halts all attacks on the system.

3. Analyze the Results

With the data gained from your experiment, begin forming your conclusions by comparing your observations to your hypothesis. Questions you want to ask might include:

- Did your systems behave as expected?
- If you had any failsafe systems, did they operate as intended?
- What new bugs did you uncover as a result of the experiment?
- How quickly did your alerting system detect the issue and notify you? Is this time acceptable according to your service level objectives (SLOs)?
- How well did your observability tools track your key metrics? Are there any changes you would make to reduce noise or collect more meaningful data? If we're reproducing a previous incident, how closely do these results match what we experienced?
- Did your systems automatically return to a normal state after the experiment, or did they require manual intervention?

We want to approach these questions with the goal of not just fixing issues, but preventing them from happening in the future. Depending on the root cause, this might mean fixing defects in our application, adding redundant infrastructure, implementing automated systems and processes to detect and handle failure, etc. Provide your engineers with the observability data you collected and the insights you gained from the experiment, as this will give them the resources to make effective decisions for improving reliability.

4. Repeat Your Experiment

Once you've implemented a fix, repeat your experiment to ensure that the fix addresses the underlying problem. If your systems successfully withstand the attack, consider increasing the *magnitude* of the attack (its severity). For instance, if you used a CPU attack to consume 20% of CPU time, increase the amount by another 20% in the next experiment. You should also consider increasing the *blast radius*, or the number of systems targeted in a single attack. This is especially useful for testing clustered systems, auto-scaling systems, and load-balanced systems. Continue to refine your systems and experiments based on these results.

Automate Your Experiment

Once you're confident in your ability to withstand the attack, start running it on a regular basis as part of your normal testing practices. With Gremlin, you can automate chaos experiments by calling the Gremlin API. This will help ensure that new changes don't introduce regressions or cause new reliability concerns.

Start by exporting your attack or Scenario as an API call. In the Gremlin web app, open the attack or Scenario that you just ran and scroll to the bottom of the screen. Click **Gremlin API Examples** to view the fully formatted API call. You can use this to execute your experiment from any tool capable of making HTTP requests, including your CI/CD solution. You can leverage this functionality to fully integrate Gremlin into your pre-production pipeline, run chaos experiments alongside your normal testing practices, and repeat the experiment after deploying to production to verify reliability.

Repeating attacks also helps identify regressions. For example, imagine we found a bug with our application that caused it to consume all available disk space. We implemented a fix and created alerting rules to notify us if the problem occurs again. Instead of waiting for another incident to trigger the alert, we can use a disk attack to proactively and continuously test that our alerting process works, and that the original problem hasn't resurfaced.

Throughout this process, continuously monitor the results of your experiments and compare your metrics to track your progress. Are your KPIs trending in a positive direction? Are you catching more defects in pre-production? Are your teams becoming more comfortable with running GameDays and responding to FireDrills?

Checklist

- [ ] Set up your observability and monitoring tools to track key metrics
- [ ] Deploy Gremlin to your target systems
- [ ] Familiarize yourself with the Gremlin web app
- [ ] Organize and schedule a GameDay
- [ ] Run your first chaos experiment
- [ ] Make improvements to your applications and systems based on your observations
- [ ] Run an attack or Scenario using the Gremlin API
- [ ] Automate the attack or Scenario as part of your CI/CD pipeline

Scale Up Your Chaos Engineering Practice

Now that we've successfully completed our first chaos experiment, what's next? How do we go from running one-off experiments to fully ingraining Chaos Engineering into our organization?

The key to early Chaos Engineering adoption is to play it safe and start at a small scale. Instead of introducing the entire engineering team to Chaos Engineering at the same time, start with a single team. Ideally, this is a team overseeing mature and reliable production systems. Using the processes detailed in this document, run a low-magnitude chaos experiment in your production environment following the GameDay structure. Use this opportunity to find and fix problems in these systems, as well as develop runbooks that your engineering teams can use to respond to outages.

For example, a good starting point might be the team in charge of a website. This team already understands the importance of having reliable systems, and most likely has processes in place for automatically handling or responding to incidents. An initial chaos experiment might be to consume additional CPU capacity on one web server to test whether it can maintain performance, or if your golden signals (specifically latency and throughput) suffer as a result. A more disruptive test might involve shutting down one or more servers to test whether your website can handle multiple concurrent system failures.

Once you understand how these systems respond to failure, and that the team has a plan in place for responding to these failures, begin running FireDrills. A FireDrill is a staged incident designed to test both your teams and your runbooks against a real-world outage. FireDrills help teams become more familiar with the incident response process, which will reduce your response times and mean time to resolution (MTTR) during a real production incident. They're also an opportunity to train new employees, ensure they have access to the appropriate

Schedule periodic FireDrills to test your incident response processes and prevent teams from becoming complacent.

tools, and that they can follow your runbooks. Schedule monthly or quarterly FireDrills to periodically test your teams and prevent them from becoming complacent.

Introduce Chaos Engineering to Additional Teams

Once you've proven success with one team, repeat the process with other teams. The best place to start is with a team that depends on the systems that you just tested. For example, if you ran your first experiment with your database team, focus on running a GameDay with your backend web team. Leverage the existing line of communication between these two teams by encouraging them to run joint GameDays and run experiments across both functions. This also allows the more experienced team to mentor the less mature team, which helps the entire organization open up to Chaos Engineering.

Run Experiments in Production

If you've only been running chaos experiments in a pre-production environment (such as a test or staging environment), you aren't extracting the full value of Chaos Engineering. Experimenting in production is necessary because:

- Pre-production environments can't accurately replicate production. Even minor differences in architecture and usage patterns can cause your applications to behave drastically differently, and these differences are impossible to fully reproduce.
- Emergent behaviors arise over time, especially in complex systems. Not only can we not replicate these in pre-production, but we can't always predict them either.
- Production is what your customers are using. For this reason alone, finding and fixing defects in production is critical to the success of the business.

It's natural for teams to be risk-averse when it comes to experimenting in production, which is why we must approach it carefully and strategically. There are several strategies for testing in production that allow us to safely run experiments without putting our entire production infrastructure and user base at risk. These include:

- Blue-green deployments, where we run two identical production environments side-by-side with one acting as a failover.
- Canary deployments, where we contain new deployments and experiments to a small subset of users and infrastructure before rolling them out to everyone.
- Dark launches, where we copy user traffic from live systems to separate systems. This lets us experiment using real-world traffic, but without impacting the systems serving that traffic.

To learn more about these strategies and their benefits, read our blog post on Testing Doesn't Stop at Staging.

Continue to Leverage Automation

In the previous section, we explained how to automate your chaos experiments as part of your testing process. Automating your chaos experiments lets you continuously verify the reliability of your applications and systems. As you scale up your Chaos Engineering practice, automation will allow you to:

- Run more experiments in a shorter amount of time
- Repeat experiments consistently for every change
- Immediately raise alerts in case of a failure
- Trigger additional automation, such as a configuration management tool or monitoring solution

Not all experiments should be automated. Those that have unacceptably high-risk or complexity—such as simulating a region failure—will almost always need human oversight. However, automating experiments that are lower-risk and easier to execute allows you to continuously ensure reliability without consuming valuable engineering time.

Final Checklist

- [ ] Introduce Gremlin into the workflow of a team in charge of a mature, stable service
- [ ] Run a low-magnitude test on a production system
- [ ] Run a FireDrill
- [ ] Schedule recurring GameDays and FireDrills
- [ ] Introduce Gremlin to additional teams
- [ ] Reproduce a previous outage in production
- [ ] Integrate an automated attack with another tool in your CI/CD pipeline, such as a monitoring or alerting service
- [ ] Run continuous, automated experiments in production