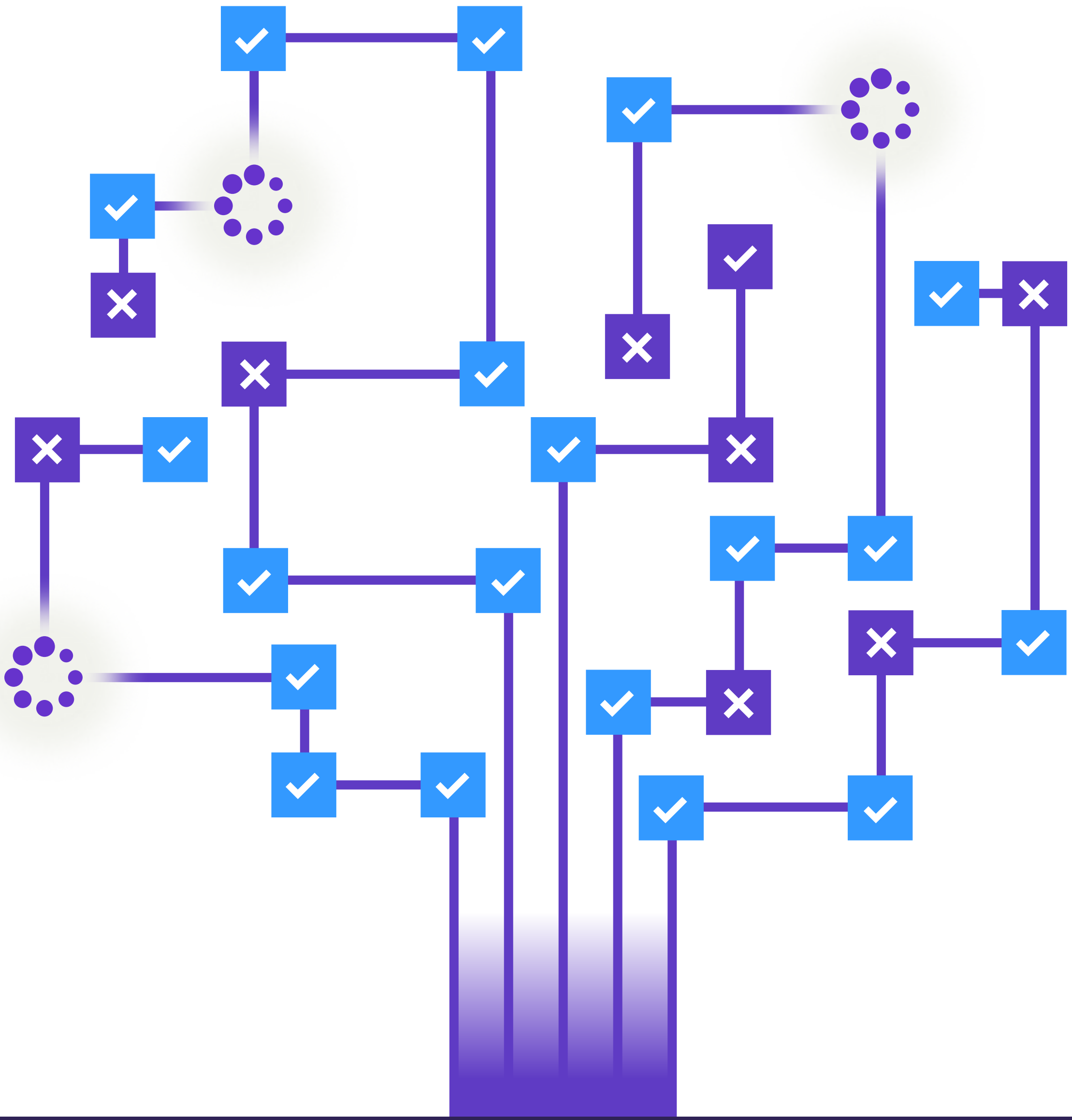
Gremlin

Guide to Enterprise Resiliency Testing

HOW TO FIND AND FIX AVAILABILITY RISKS WITH GREMLIN



MASTERING RELIABILITY

TOOL GUIDE

Table of contents

Table of contents

Intended Audience

This ebook is intended for practitioners who will be directly involved in creating and running experiments within Gremlin. This includes:

- **Site reliability engineers (SREs).**
- **Technical engineers responsible for the reliability of an application, service, or infrastructure component.**
- **Any stakeholders involved in running GameDays.**

Introduction

Systems can fail in myriad ways with significant impacts on day-to-day life, from **outages causing flight cancellations** to **major bank outages** impacting millions of customers. To avoid this, Gremlin provides a range of tools designed to help proactively identify reliability risks in software systems and infrastructure. This ebook explains how these tools work, and how you can most effectively use these tools to improve the reliability of your own systems.

Gremlin is built on the concept of *fault injection*, which is intentionally creating controlled failures in a computing component, such as consuming CPU on a host or adding latency to a Kubernetes container. Simply creating failures isn't enough to improve reliability: you need to be able to observe the impact these failures have on your system, and use those observations to make improvements. This process of using fault injection to answer questions about your infrastructure is called an *experiment*. Gremlin gives you full control over your experiments by letting you fine-tune faults to answer specific questions about your infrastructure, such as terminating specific processes to test recoverability, adding packet loss to specific network routes to test error-handling, and consuming memory in a container to test scalability.

Additionally, Gremlin provides *reliability tests*, which are pre-built experiments that you can run on your services in a single click and include a clear pass/fail result. By running reliability tests and experiments on our systems, we can uncover unexpected behaviors in our systems, validate resilience mechanisms, and improve our overall reliability.

The goal of using these tools isn't to cause our systems to fail, but to help us identify and fix failure modes.

Because Gremlin offers such a wide range of testing methods, each with their own unique behaviors and options, it helps to understand how each one works and when you would want to use it. In this ebook, we'll explain each of Gremlin's experiment and reliability test types organized by use case. We'll present common use cases teams run into when running modern distributed systems, explain which experiments can be used to address each use case, and provide best practices for running those experiments. We present each experiment in the context of the technical and business initiatives it helps solve to make it easier to relate your reliability work to your organization's goals.

By reading this ebook, you'll learn:

- What fault injection is.
- How to run each of Gremlin's unique experiments and reliability tests.
- What each experiment does, the failure modes it helps identify, and the resilience mechanisms it helps validate.
- When you'd want to use one experiment type over another.
- How to interpret the results of an experiment and use this knowledge to improve reliability.

What is fault injection?

Gremlin offers two different (but related) ways to understand reliability risks in complex architectures: experiments, and reliability tests. While both take a slightly different approach to testing, they share a single underlying concept called fault injection. Fault injection is a technique for creating controlled failure in a computing component, such as a host, container, or service.

Fault injection started as a technique for simulating hardware-level failures, such as shorting electrical circuits, creating electromagnetic interference, and disrupting power supplies. The goal was to see how stressors like these affected a device's operations, find the point where the device failed, then redesign the hardware to be more resilient. This same concept applies to software fault injection.

Gremlin provides software fault injection on the infrastructure level using an agent running on a host. When you initiate an experiment or reliability test in Gremlin, the agent injects the fault onto the target system(s), tracks and manages the fault for the length of the experiment, then safely stops it once the experiment is finished (or if you stop the experiment early). The injected fault is then reverted and the system returns to its normal state.

At this point, you might think "why should I inject fault into my systems? We have enough 'chaos' already!" While fault injection might seem counterintuitive, it's the best way of testing and validating a system's resiliency. When teams build complex systems, it's impossible to anticipate all of the various ways these systems can fail in high-stress real-world situations. This is where fault injection fits in. Teams use fault injection to simulate a wide range of stressors acting on their systems. This helps catch and uncover vulnerabilities *before* they turn into production outages and cause the serious technical and business consequences shown below.

COST OF DOWNTIME IN REVENUE/MIN

- **Walmart:** $1,153,000/min
- **Amazon:** $826,000/min
- **Target:** $207,300/min

Assuming 99% uptime (~7 hours of downtime per month), the lost revenue caused by reliability-related outages costs: **$6.1B/year**

COST OF RESOLUTION

- Assuming 100 hours of downtime per year (~98% uptime), $200,000 revenue lost per hour of downtime, and 10 employees involved in resolution, resolving reliability-related outages costs: **$189.2M/year**

IMPROVING UPTIME

If downtime costs $200,000/hour, improving uptime results in the following revenue gains:

- **90% → 99%:** $158M gained per year
- **99% → 99.9%:** $15.8M gained per year
- **99.9% → 99.99%:** $1.7M gained per year
- **90% 99.99%** → $194,625,305

IMPACTS ON CUSTOMER EXPERIENCE

- 20% of consumers will never shop with a brand again after encountering just one error.
- 41% of customers think it’s not possible for a company with a poor user experience to deliver a quality product.

Based on 2022 reported retail sales revenue.

What fault injection methods does Gremlin provide?

Gremlin provides 11 primary types of fault injection, which power each of Gremlin's 11 experiments. Gremlin organizes these experiments into three categories:

STATE	RESOURCE	NETWORK
Shutdown: Shuts down (and optionally reboots) the host operating system.	**CPU:** Generates high load for one or more CPU cores.	**Blackhole:** Drops all matching network traffic.
Time Travel: Changes the host's system time.	**Memory:** Allocates a specific amount of RAM.	**Latency:** Injects latency into all matching outbound traffic.
Process Killer: Kills the specified process(es) on the host.	**IO:** Puts read/write pressure on I/O devices such as hard disks.	**Packet Loss:** Induces packet loss into all matching outbound traffic.
	Disk: Fills storage to a specific percentage by writing temporary files.	**DNS:** Blocks access to DNS servers.

Each experiment is fully customizable with a wide array of options and parameters. For example:

- Resource experiments let you specify the exact amount of each resource to consume, both in discrete units and percentages.
- Network experiments let you target specific kinds of traffic based on IP address, hostname, or port.
- Process Killer lets you specify which processes to terminate using their names, IDs, or a range.

Gremlin also provides *reliability tests*, which are pre-built experiment workflows designed to test your services under specific conditions. The key difference between reliability tests and experiments is that experiments are fully customizable, whereas reliability tests are pre-defined, have a clear pass/fail result, and are based on Gremlin's years of experience and expertise implementing reliability best practices at large organizations. They guide teams who might not be sure where to start, want to make reliability testing easier, or want to automate their reliability testing.

For example, our Latency experiment lets you drop a customizable percentage of network traffic to a specific network interface, while our Dependency Latency reliability test increases latency by a specific amount (100ms) for 5 minutes while monitoring the health of the target service. You could easily recreate the Latency reliability test using experiments, but the reliability test is already defined and can be run or scheduled with a single click. Teams will often define their own experiments to test for specific failure modes, while also using reliability tests to cover more common use cases.

SCALABILITY

CPU: Tests that the service scales as expected when CPU capacity is limited.

Memory: Tests that the service scales as expected when memory is limited.

REDUNDANCY

Host: Tests resilience to host failures by immediately shutting down a random host or container.

Zone: Tests the service's availability when a randomly selected zone is unreachable from the other zones.

DEPENDENCY

Failure: Drops all network traffic to the dependency.

Latency: Delays all network traffic to the dependency by 100ms.

Certificate Expiry: Opens a secure connection to the dependency, retrieves the certificate chain, and validates that no certificates expire in the next 30 days. If there is no secure connection available, and therefore no certificates, this test will pass.

Use cases: when to use experiments and reliability tests

Each experiment and reliability test Gremlin provides can answer a lot of questions about your systems and processes. When starting, it's difficult to know which experiment to use to answer these questions. For this reason, we structured this white paper by use case first, then by the experiment(s) that best fit the use case. You can view each use case below, or use the table of contents to find the one that best applies to your needs.

- Migrating to a new environment
- Preparing for peak traffic events
- Optimizing for cost savings
- Optimizing the user experience
- Meeting security and compliance requirements
- Planning for business continuity

Migrating to a new environment

Moving applications and services to a new environment like the cloud is a major undertaking. Teams must not only ensure their services are compatible with the new environment, but also that they're properly configured for the environment. This doesn't just mean taking advantage of the new platform's capabilities, but understanding the environment's unique behaviors and how those behaviors can impact your services.

Migrating to a new environment can include:

- Moving from an on-premises data center to a cloud environment like AWS, Azure, or GCP.
- Moving from a traditional monolithic architecture to a cloud-native or service-based architecture like Kubernetes.
- Integrating third-party services—managed databases, cloud-hosted caches, etc. —into your applications.

The challenges migration teams face include (but aren't limited to):

- Knowing how to right-size the deployment to balance capacity and scalability with cost.
- Accounting for latency introduced by a cloud environment, especially between zones or regions.
- Properly configuring load balancing, redundancy, and replication, so failovers are automatic and instantaneous.

Right-sizing and optimizing your target environment with resource experiments

In a cloud environment, resource allocation is one of the most important factors as it affects cost, performance, scalability, and more. With Gremlin's resource experiments, you can ensure your new environment has enough capacity to handle your largest workloads, or that it can scale effectively, without adding significant costs.

CPU experiments are relatively simple: they consume the amount of CPU that you specify for the length of time specified. This helps you ensure that your applications behave as expected even when CPU capacity is limited or exhausted. While modern servers have multiple cores per CPU (or even multiple CPUs), they still have limits, especially when running multiple concurrent workloads. In cloud environments, especially, these workloads often compete for CPU time, creating what's called the "noisy neighbor" problem. In addition to simulating noisy neighbors, CPU experiments can help test and validate automatic remediation processes, such as auto-scaling and load balancing.

Memory experiments perform a similar function as CPU experiments, but for memory (RAM). Memory management is often more strict than CPU management, since memory is an even more finite resource. Systems that are low on RAM might terminate processes; or, in the case of containers and serverless workloads, move them to a completely different host. Also, if you have a workload that requires a significant amount of memory (e.g. a machine learning model), memory experiments let you simulate that workload so you can proactively determine the impact on your environment and optimize for capacity.

Disk and I/O experiments do for persistent storage what CPU and memory experiments do for their respective resources. Disk experiments temporarily consume space on a storage device like a hard disk, while I/O experiments consume throughput on a device. You can use I/O experiments to simulate lower-throughput storage devices like network-attached storage (NAS) and spinning disks (HDDs), and use disk experiments to simulate workloads that require large amounts of persistent storage.

For example, database or caching services often require lots of storage space and are constantly reading and writing to storage, requiring high throughput *and* high capacity. This can get expensive unless you optimize, and disk and I/O experiments can help find that middle ground.

Testing network fault tolerance and dependency resilience with network experiments

Network experiments simulate a wide variety of network conditions, such as low throughput, dropped connections, and failed routes.

Blackhole experiments drop network connections to and from a service, a host, or even an entire availability zone or region. This is ideal for testing redundant systems and automatic failover mechanisms. For example, if we have an Amazon RDS database that's configured for replication across two zones, using a blackhole experiment to drop traffic to one RDS zone should cause traffic to automatically redirect to another zone. This can be used on-premises or in single-zone environments to simulate controlled outages and test various fault tolerance mechanisms.

Latency and packet loss experiments let you shape network traffic regarding how slow it is, and how much data arrives at its destination. This lets you simulate slow and unstable network conditions, which is a potential when moving to busy cloud environments where bandwidth is shared between countless workloads. These experiments can test whether your services can handle poor network conditions without timing out, returning errors, or crashing. They're also valuable for testing the behaviors of load balancers and other tools that route traffic based on network quality.

For example, imagine we have multiple AWS Lambda instances running and we're using **Amazon Elastic Load Balancing** (ELB) to automatically distribute incoming traffic across them. If one of our instances suddenly becomes unresponsive, or is returning errors, or is dropping packets, we want to know that ELB will redirect traffic to a faster, more reliable instance. We can run a latency or packet loss experiment to create this unreliable connection ourselves, then watch our ELB to ensure traffic gets redirected quickly and correctly. Once we've verified the behavior, we can stop the experiment using Gremlin's halt button and immediately revert to normal operations.

Testing redundancy with state experiments

In an on-premises environment, organizations have full control over their environments. However, cloud environments shift that control to the provider. This means the cloud provider can migrate workloads, terminate hosts, and change functionality on their terms. Providers will work to maintain service while this is happening, but **sometimes these changes can result in outages**, so it's best to be prepared.

State experiments like shutdown and process killer can reproduce these kinds of unexpected infrastructure changes. The most well-known is the shutdown experiment, which shuts down or restarts a host or container. This is the mechanism behind **Chaos Monkey**, a popular early Chaos Engineering tool that randomly shuts down a host. While Gremlin's shutdown experiment supports randomized targets, we recommend intentionally choosing a host to target so the test is more controlled.

Shutdown experiments are also useful for container-based workloads like Kubernetes. In Kubernetes, containers constantly start, stop, restart, and move between hosts as resource requirements and container counts change. Shutdown lets you verify that your hosts and containers can automatically recover if they're deliberately or accidentally shut down or restarted, without interrupting service.

Preparing for peak traffic events

Peak traffic events like Black Friday and Cyber Monday can push any organization's infrastructure to its limits. It's difficult enough to build large-scale applications that are stable during day-to-day usage, and while some peak traffic events can be anticipated (like sales holidays), some can be completely unpredictable (like a sudden endorsement from a celebrity). Operations teams must strike a balance between having enough capacity to handle surprise peaks, while not spending too much money on leftover unused infrastructure.

Validating scalability and resource quotas

CPU, memory, and disk experiments are ideal for testing system scalability. Cloud computing has made it much easier to automatically scale systems up or down based on changes in demand. However, this process isn't instantaneous, and it's not configured out of the box. After you enable and configure autoscaling, use resource experiments to trigger your scaling thresholds and track your services to see how quickly and how effectively they scale. You might find that the scaling process isn't as fast as expected, or that scaling didn't address the problem. You might also find that your systems don't scale back down once the experiment is finished, resulting in wasted resources.

Equally important are resource quotas, which are limits set in place to prevent services from using too many resources. This is especially common in container-based and serverless environments like AWS Lambda, Amazon ECS, and Kubernetes. In these environments, individual workloads can have limits on the resources they can consume, and the platform enforces those limits by rescheduling the workloads to higher capacity nodes, adding replicas to spread out the work, or simply stopping the workloads. We can use resource experiments to make sure these processes work as expected, so we don't have to worry about unexpected behaviors when running these environments in production. This also helps with capacity planning and identifying noisy neighbors.

Stress testing using resource experiments

Resource experiments can recreate the effects of increased demand on a system. As more users access your system, resource usage is more likely to increase. While simulating real users typically falls under load testing, resource experiments reproduce the impact of this load: high CPU usage, low memory, saturated networks, etc. Experiments and reliability tests can **be combined with load tests** to better simulate production environments. This lets you determine how resilient your systems are when under load and with limited resources.

Testing your monitoring and alerting systems

This ebook focuses primarily on your application infrastructure, but it's important to remember the ancillary systems that support these workloads, especially observability systems. Observability is how engineering teams identify and address problems with the environment, and if you don't have visibility into your systems, troubleshooting and fixing issues will be much harder.

As you run experiments on your services, make sure your monitoring solution is detecting changes in resource usage and availability. If you have alerts set up to notify you, e.g., when a host becomes unavailable or CPU usage passes a certain threshold, run experiments to make sure they trigger when those thresholds are reached and that the notifications are successfully sent.

Optimizing for cost savings

Cloud costs increase based on usage, creating a need for engineers to optimize their applications to run as efficiently as possible. However, there's a balance. Allocating too many resources results in paying for resources you're not using, but allocating too few resources can lead to poor performance and instability. Engineering teams must strike a balance between capacity and cost, and the best time to do this is before our customers notice.

Validating that you can autoscale

Modern architectures like Kubernetes and serverless platforms are designed to scale. Kubernetes supports both horizontal autoscaling (deploying multiple replicas of a container across several nodes) and vertical autoscaling (deploying new nodes to increase total cluster capacity). This works in both directions: we should be able to add new nodes and container replicas as needed to handle increased demand, but we must also be able to remove these as demand slows, *without* impacting our workloads.

Since scaling is predominantly based on resource consumption, **resource experiments like CPU, memory, and disk** are essential for testing scalability. As an example, you can configure the Kubernetes horizontal Pod autoscaler (HPA) to **scale on a metric** such as CPU or memory. If we want to scale a specific container deployment once it consumes a certain percentage of its available CPU—say, 50%—we could run this command:

```
kubectl autoscale deployment frontend --cpu-percent=50 --min=1 --max=10
```

This creates a policy for a deployment named “frontend” to add a new instance if the deployment as a whole is using more than 50% of its available capacity, up to a maximum of 10 replicas. We can now run a CPU experiment to increase the deployment’s CPU usage to 51%, monitor our deployment to make sure it scales successfully, and also monitor our application to see how it behaves during this process. Likewise, when the experiment ends and CPU usage returns to normal, we should watch to make sure our application can tolerate losing replicas. This is less about making sure the HPA feature works, and more about making sure we can deploy new replicas on-demand quickly and without impacting our application’s stability or operations.

> Learn how to **validate horizontal Pod autoscaling on EKS using Gremlin.**

Optimizing the user experience (UX)

Modern users have come to expect services to be fast, stable, and always available. When services become slow or fail, even temporarily, customers become frustrated and are more likely to **abandon you for a competitor**. It’s important to avoid outages, since those are the most visible kinds of failures, but other issues can impact the user experience, including:

- Slow application response times.
- High latency or packet loss resulting in timeouts and invalid data.
- Poor quality of service (QoS) for streaming applications.

Ensuring you can fall back to redundant systems

Having redundant systems in place can ensure that you can keep providing service to your users, even when you have a host, zone, or region outage. Redundancy is difficult to set up, since teams must consider how to replicate systems, services, and data, while also creating a mechanism to fail over from their primary systems to the replicas. This requires several different processes that work together to ensure a smooth and seamless transition from one system to another, without any loss in data or availability.

For example, imagine you have an Amazon EC2 instance in one zone, and a replica in another zone. There's a load balancer in front of these two instances that checks to see if the primary is responsive, and if it's not, automatically fails over to the replica. Using a **blackhole experiment**, you could drop all network traffic to the primary instance to ensure this failover process works as expected.

Of course, not all failures are a binary "on" or "off." A network might not fail entirely, but might perform more slowly than before. What happens if you run a **latency experiment** and add 100ms of latency to the primary? Is this enough to trigger the load balancer to redirect traffic to the replica? If not, how does this impact the user experience? The infrastructure is technically working correctly, but the outcome is less than ideal.

Providing a smooth experience for real-time and streaming applications

Applications like multiplayer gaming, music streaming, and voice and videoconferencing demand high throughput and fast network access. If a network becomes slow or saturated, these applications can experience higher latency or dropped packets. Quality of Service (QoS) policies can prevent this by prioritizing certain types of traffic when bandwidth becomes limited, but it's up to engineers to make sure their applications can remain responsive even when network conditions are poor.

Packet loss and latency experiments can simulate these saturated conditions to help you validate and fine-tune your QoS policies. While you can't always prevent networks from degrading, you can use techniques like **content buffering**, in-app notifications, or fallback networks to work around the issue.

Meeting security and compliance requirements

For organizations in industries with strict security and compliance requirements, fault injection is an additional way of checking adherence. Organizations in finance, healthcare, government, and similar industries face strict requirements for availability, performance, and data integrity.

A core part of both security and compliance is accurate time tracking. If a system's clock becomes desynchronized, it can have a significant impact on system stability and usability, especially when that system must communicate with others. Encryption, data integrity checking, and cluster management are all dependent on a universal time, synchronized via the **Network Time Protocol** (NTP). Gremlin's **time travel experiment** works by temporarily changing the target's system time while simultaneously blocking NTP network traffic. This is valuable for testing time-based scenarios like:

- How systems behave during the transition to or from **Daylight Savings Time** (DST).
- How applications handle leap years.
- Whether your systems are prepared for "end of epoch" problems like the **Year 2038 problem.**
- How databases handle timestamp discrepancies in data, and whether this impacts replication between primary and secondary database nodes.

Preparing for Daylight Savings Time (DST) and leap years

Clock changes caused by **Daylight Savings Time** and leap years can cause communication problems between systems and applications, resulting in discarded messages or incorrect timestamps. These events often aren't thoroughly tested due to their infrequency, but can nonetheless have significant **effects on time-sensitive systems.** You can use a Time Travel experiment to move the system clock forward or backward one hour, test your application, and verify the integrity of your data.

Ensuring security by testing TLS certificate expiration

TLS certificates are a cornerstone of modern Internet security and depend on accurate timekeeping. Certificates expire after a certain amount of time and must be renewed and replaced, or else communication will be disrupted. Time Travel experiments let you test the impact of skewed system clocks on encryption and make sure that you stay ahead of expiring certificates by testing notifications and automatic renewal processes.

Preparing for ″end of epoch″ problems

The systems that computers use to track time can have their own unexpected behaviors. For example, the **Year 2000 problem** (Y2K) was caused by computers storing the current year as a two-digit number, which meant that they couldn't differentiate between January 1, 2000, and January 1, 1900, as they were both stored as "00." A similar problem—called the **2038 problem**—is caused by a limitation of how 32-bit Unix-based systems store date and time values. On January 19, 2038, this value will overflow and roll the date back to December 13, 1901. If you're not sure whether your systems are protected, you can use the Time Travel experiment to test them by moving the system clock forward past the 2038 date and checking your system's clocks to see what they display. You can use this same strategy to prepare for similar **"end of epoch"** events.

Testing clock sync (NTP) between nodes

Systems that share data often require synchronized system clocks, which are best managed using a service like **NTP.** Mismatched timestamps can lead to problems like discarded messages and data conflicts. It's possible that NTP may be unavailable and your clocks will drift. Using Time Travel lets you simulate clock drift and NTP outages simultaneously so that you can proactively prepare for an outage.

Planning for business continuity and disaster recovery

Businesses must be prepared for sudden, unexpected losses in infrastructure. Systems fail, datacenters experience outages, and even the largest cloud providers can't keep their systems running 100% of the time. When your customers demand high availability, replication and redundancy must become a priority.

The best approach to testing replication and redundancy is by simulating sudden infrastructure failures. This means using **network experiments like blackhole and DNS failure.** Blackhole experiments are commonly used to simulate a sudden loss of a system or service. They offer a controlled way of simulating infrastructure-wide outages, without causing infrastructure to fail. This lets teams practice their incident response and recovery plans without the added stress of a real-world disaster.

Proactively testing region failover

Many network gateway and load balancing tools can automatically reroute traffic between nodes based on latency. For example, **Amazon Route 53** can route individual DNS queries to the AWS Region that gives a user the lowest latency. You can ensure this is configured correctly by running a Latency attack on a specific node, availability zone, or even an entire region, and validating that traffic is rerouted to a faster resource.

For example, imagine that you have a Kubernetes cluster running in one region (e.g. New York) and a replica cluster running in another region (e.g. California). The New York cluster is your primary cluster that handles most traffic, while the California cluster is a secondary cluster that handles some or no user traffic. If a major incident like a power outage takes your New York cluster offline, you want to ensure that you can automatically failover to the California cluster. Using a blackhole experiment, you can test this by dropping all network traffic to your New York cluster and monitoring your application to ensure the failover process works as expected. Likewise, you can drop all traffic to the California cluster to ensure you don't have a misconfiguration or dependency that impacts availability.

Avoiding large-scale DNS outages

DNS errors are also a common cause of large-scale outages. The Domain Name System (DNS) is a system for associating domain names with IP addresses so you can refer to computers and services by name rather than address. For example, in July 2021, an **Akamai outage** caused many top websites to go down across airlines, retail, finance, and other industries.

DNS experiments simulate a DNS outage by blocking network access to DNS servers. This helps prepare for DNS failure, test fallback DNS servers, and validate DNS resolver configurations, which answers questions such as:

- Do you have a secondary DNS server and do your services fallback automatically to it?
- Do you gracefully reroute calls back to the primary once it's back online?
- How do your services behave during a major DNS outage, such as the **DynDNS** or **Akamai** outages?

> Learn how you can **test your disaster recovery procedures using Gremlin.**

How to start running experiments

Now that you understand the different experiment types and the use cases they help solve, you can start thinking about how to apply them to your own systems. However, this process is slightly more involved than just running an experiment, waiting for it to finish, and seeing if anything happens.

We've outlined much of this process in our webinar: **Navigating the reliability minefield: Finding and fixing your hidden reliability risks.** To summarize, there's a process teams should follow when reliability testing their systems. At a high level, it involves:

1. **Identifying the services to test.**
2. **Modeling failure modes.**
3. **Performing experiments.**
4. **Using the results to make improvements.**

Identify the services you want to test

The first step is determining what to test. We recommend starting by breaking down your infrastructure into discrete, standalone units that can be tested independently of each other. For the sake of simplicity, we use the term "service" to describe these systems and software. A service is the specific functionality provided by one or more systems within an environment, such as a checkout service or authentication service. They typically have clear interfaces and can be deployed independently of each other. They can include microservices, cron jobs, daemons, and monolithic applications. Testing services is important because it's easier to isolate reliability problems to individual services, and engineering teams are often already structured around services.

Teams often manage many services at a time, so start by prioritizing the ones you want to test first. If you want to have a big impact from your efforts, start with the services closest to the customer, such as APIs or front end services. These are the services that'll have the greatest business impact if they fail. But if you'd prefer to take a more cautious approach to testing, you could start with lower risk, less critical services, prove your results, then move on to business-critical services.

Model failure modes

Once you've identified the services you want to test, identify their failure modes. Failure modes are the way that a service can fail or produce an unexpected outcome. This originates from Failure Modes and Effects Analysis (FMEA), a decades-old method for identifying all possible failures in a design, manufacturing, or assembly process, product, or service (learn more in our blog post **Achieving FMEA goals faster with Chaos Engineering**).

Trying to list every single individual failure mode in a complex system is extremely difficult, which is why we recommend starting with broad categories:

- **Scalability:** How well does your service behave when resources (such as CPU or memory) are limited or exhausted?
- **Redundancy:** How well does your service respond to losing a host or zone?
- **Dependencies:** How well does your service handle losing access to other services that it depends on?

We can further break these categories down into individual test cases, such as:

- If I increase CPU consumption, does my service scale as expected? Does the same happen for RAM?
- If my service loses connection to a host, will it continue operating? Does the same happen when it loses connection to an entire zone?
- Does my service keep running if it has a slow or dropped connection to a soft (not required) dependency? What about a hard (required) dependency?

Test failure modes

Once you've identified your use cases, your services to test, and their failure modes, you can start testing them. This is where fault injection comes in. With fault injection tools, you can create these failure modes in a controlled manner instead of waiting for them to happen in the real-world. This process of using fault injection to test your systems' reliability is called Chaos Engineering.

At Gremlin, we categorize fault injections into three categories:

1 **Resource experiments:** Tests against sudden changes in consumption of computing resources, such as CPU or Memory.

2 **Network experiments:** Test against unreliable network conditions, such as ones that might make dependencies unavailable.

3 **State experiments:** Test against unexpected changes in your environment such as power outages, node failures, clock drift, or application crashes that might make hosts, zones, or dependencies unavailable.

Within these categories, there's a variety of different experiment types that can be customized to fit dozens of different environments and cover almost every scenario you're going to run into (Gremlin has these tests pre-built and ready to run on your services). Ideally, you'd get to a point where you're testing every one of your services against every potential failure mode. To start, identify your most important services and their failure modes using the use cases mentioned in this ebook and start there. You can use our free **reliability tracker tool** to help with identifying and prioritizing which service/failure mode combinations to start with.

Chaos Engineering Resources

Chaos Engineering includes a lot more than we're covering in this whitepaper. Share these resources with your team to learn more about Chaos Engineering and start testing your services:

- **What Is Chaos Engineering?**
- **Chaos Engineering: The History, Principles, and Practice**
- **Chaos Engineering Adoption Guide**

If you're not sure which tool to use, you can also check out **The Guide to Chaos Engineering Tools.**

Continually improve and track your progress

Once you've developed a reliability testing process, automate it as part of your day-to-day engineering operations. Run regular (e.g. weekly) reliability tests, document and discuss newly discovered failure modes with your team, and track fixes made to your service(s). This demonstrates to the organization that your reliability efforts are paying off and contributing value to the business, especially if you can demonstrate that your work prevented a potentially large outage on a critical service.

Gremlin provides many different tools for automating and tracking reliability. For one, Gremlin keeps track of each reliability test performed, along with the outcome. You simply:

1. **Define your services** in Gremlin.
2. Link Gremlin to each service's most important monitors and alerts using **Health Checks.**
3. Run Gremlin's suite of pre-built **reliability tests.**

For each service you test, Gremlin generates a **reliability score** ranging from 0 to 100. This score is an objective measure of a service's reliability. You can see how each service's score changes over time, identify the exact cause of a failure, and generate reports for your managers.

For more fine-grained testing, Gremlin also gives you access to its full suite of fault injection tools. You can fully customize and run individual faults on your systems, orchestrate multiple attacks as part of a **Scenario**, and even execute large-scale **GameDays** with your team. By first showing a baseline of reliability, then improvements to reliability, you can demonstrate to leadership that your work has had a positive impact. And if they dig deeper, you have clear, demonstrable impacts that you can show them by tying specific tests to specific failure modes on specific services.

In the end, you're not only able to show leadership that your work has had clear results, but you're also creating real, demonstrable value for the business by preventing costly outages and downtime.

Conclusion

Gremlin's various fault injection and experiment types intentionally cover a broad range of potential failure modes. As software systems become larger and more complex, engineers must consider the many ways these systems can fail, and how practices like Chaos Engineering and fault injection help address those failures. Fault injection isn't about creating chaos or intentionally causing failure, but to verify the resilience of our systems and making sure they behave the way we expect them to. Each failure mode, reliability test, and experiment type tests a wide range of behaviors, and by using each one effectively, we can gain a comprehensive and thorough understanding of how resilient our systems really are.

Now that you've seen how each of these tests can be applied to your systems, consider how they can help you achieve your technical and business initiatives. Start designing experiments that make use of each attack type, and plan a time to execute them. Not only will this maximize the value you get out of Gremlin, but it will help your technology, people, and processes become more reliable.

Gremlin

Gremlin is the Enterprise Reliability Platform that helps teams proactively test their systems, build and enforce reliability and resiliency standards, and automate their reliability practices organization-wide.

Learn more at **gremlin.com**.