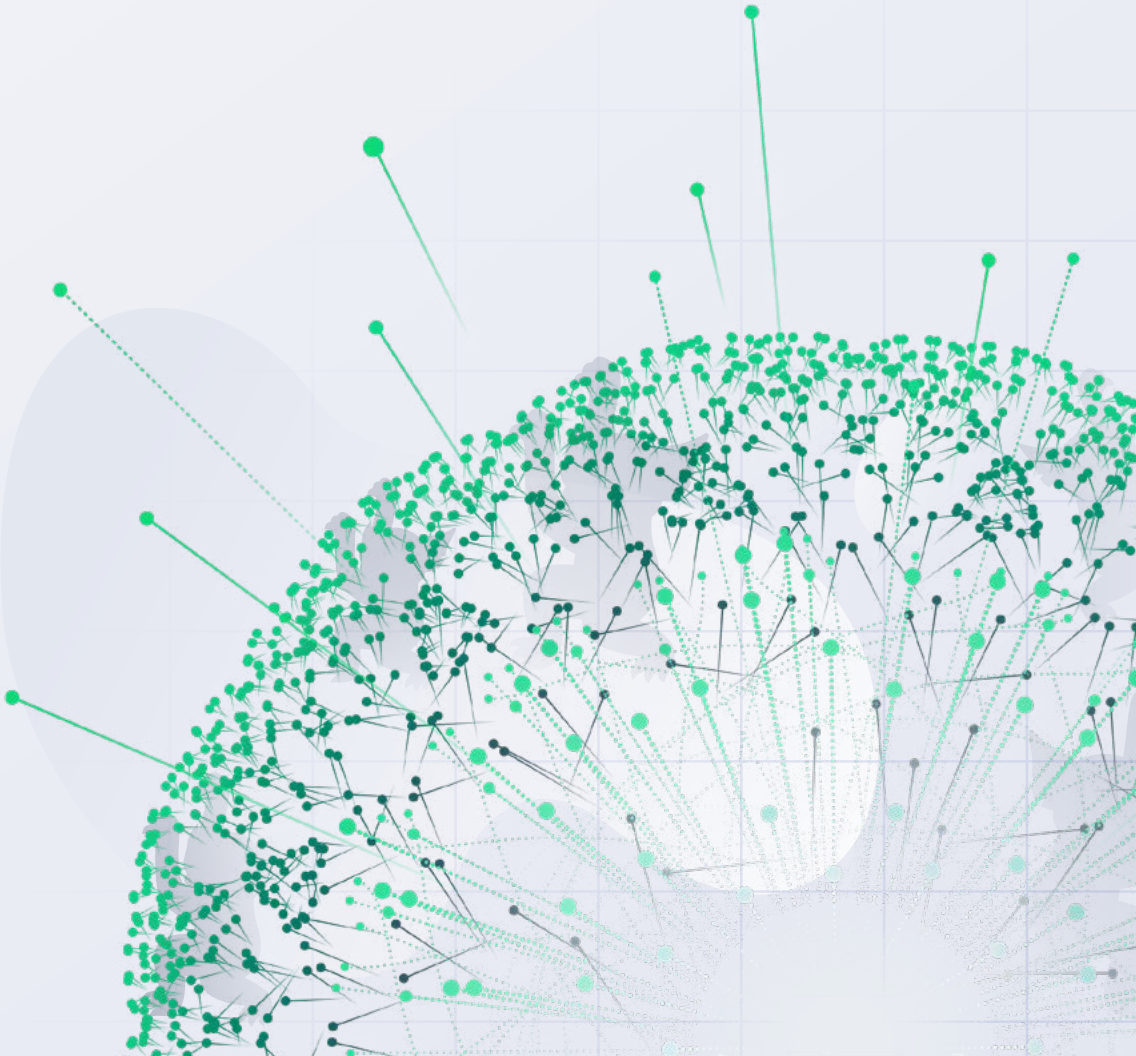
WHITE PAPER

The first 5 Chaos Experiments to run on Kubernetes



Gremlin

Table of Contents

High availability is a critical feature of modern cloud systems

Kubernetes makes it relatively easy to enable replication, load balancing, and automatic failover, but even the most resilient systems can fail. If a node fails, we exhaust our computing resources, or the network temporarily goes offline, what does that mean for our services? Are we confident that they'll continue running reliably, or is there a risk of an outage? The more prepared an organization is for unexpected failures, the less likely it is to experience incidents and customer-impacting outages.

In this white paper, we explain how to improve the resiliency and availability of Kubernetes clusters using Chaos Engineering. You will learn how to identify potential failure modes and points of weakness in a cluster, design experiments to test for these failure modes, then execute those experiments to ensure you've designed a resilient system. The result of this process will help you and your team harden your infrastructure, improve reliability, and keep your applications running smoothly.

Introduction

Kubernetes is transforming the way organizations build and deploy applications. It helps teams of all sizes orchestrate applications on an enterprise scale by providing automatic cluster management, automatic scaling, and automatic replication. When used in a team that embraces DevOps culture, Kubernetes can greatly accelerate development time and release velocity, reduce manual server administration work, and even reduce the time spent setting up redundant systems.

However, no tool is perfect. Kubernetes has many self-healing mechanisms, but there are an infinite amount of possible Kubernetes configurations, hardware architectures, application types, and deployment models. One team's applications and workloads can have drastically different requirements and failure modes compared to another's, necessitating additional testing and fine-tuning.

On top of this, the complexity of Kubernetes means that failures that seem small can have cascading effects. There are plenty of Kubernetes horror stories where something that is expected to happen–like two Pods getting scheduled to different nodes–doesn't, resulting in cluster instability and application outages.

Before we look into how to apply Chaos Engineering to Kubernetes, let's start by reviewing Kubernetes' core concepts. If you're already familiar with Kubernetes, you can skip to the section How Kubernetes clusters can fail.

Kubernetes high-level concepts

Applications designed for Kubernetes typically follow a microservice architecture, where the application is split into discrete, modular, independent, and loosely coupled services. These services communicate with each other via APIs, but are managed completely independently. We can modify, upgrade, and even scale a single service without impacting any other services or the application as a whole. Compare this to a more traditional monolithic architecture, where the entire application is bundled as a single package, making upgrades or scaling far more difficult and costly.

Microservices provide greater flexibility than monoliths by letting teams deploy, modify, and scale parts of the application without needing to change the application as a whole.

Each service is deployed in a container, which is a self-contained process that bundles the software needed

to run the service along with its dependencies and configuration. Containers are lightweight, fast, and reproducible, making them well-suited to microservice-based applications. A container can start or stop in a matter of seconds, can be rebuilt and deployed to multiple hosts, and can be swapped out with another container with effectively no downtime.

As a container orchestrator, Kubernetes deploys and manages containers across a cluster of hosts. Instead of managing containers directly, Kubernetes introduces the concept of Pods, which bundle one or more containers into a single unit that has its own process space, hardware resource pool, and IP address. Pods provide greater control than containers by letting us share resources between containers directly, define container restart policies, and more. For example, if we have an application container that writes logs to a file, and we have a logging container that forwards log files to a remote server, we can deploy both containers together in a single Pod. Now, whenever we deploy this Pod, both our application container and logging container deploy together.

The real power of Kubernetes comes in the form of ReplicaSets and Deployments. ReplicaSets ensure that a Pod has a certain number of copies (or replicas) running at any given time. If a Pod belonging to a ReplicaSet fails, Kubernetes automatically deploys a new Pod to replace it. Deployments are higher-level constructs that let us use ReplicaSets along with other features such as rolling updates, canary deployments, and rolling back changes.

Lastly, Kubernetes is declarative, meaning we define the desired state of our cluster and Kubernetes performs the actions necessary to achieve and maintain that state. This state is most commonly represented using YAML files called manifests, but can also be made directly to the cluster using the kubectl command line tool.

Putting all of this together, deploying an application in Kubernetes involves:

- Packaging the application's code, dependencies, and configuration into a container image.
- Creating a manifest for a Deployment where we define our Deployment, Pod, and container specifications.
- Deploying the manifest to our cluster (using kubectl, the Kubernetes Dashboard, or using another method).

These are the basics that you need to know in order to understand the rest of this guide. There is much more to Kubernetes such as Services, Ingress Controllers, and Volumes. We recommend reading the Kubernetes documentation to learn more about the different Kubernetes object types.

How Kubernetes clusters can fail

Unfortunately, simply deploying multiple replicas of a container isn't enough to prevent potential outages. Kubernetes is a complex system, which means there are many different places where problems can occur. Even relatively small problems–such as a container consuming more CPU or memory than expected–can cause an outage due to the complex and interdependent systems in place in a Kubernetes cluster.

There are many different levels where failures can occur, including:

- **Infrastructure-level failures:** these include power outages, hardware failures, and if we're using a managed Kubernetes service, cloud provider outages.
- **Cluster-level failures:** these include misconfigured clusters, autoscaling problems, and unreliable or insecure network connections between nodes.
- **Node-level failures:** these include automatic or unscheduled reboots, kernel panics, problems with the kubelet process or other Kubernetes-related process, and network connectivity problems.
- **Deployment-level failures:** these include misconfigured Deployments, too few (or too many) replicas, missing or failing container images, and Deployment failures due to limited cluster capacity.
- **Application-level failures:** these include application crashes, source code errors, and unhandled exceptions.

We also need to consider emergent behaviors, where a failure in one component impacts an entirely different component. For example, if our cluster autoscaler isn't working properly, it could cause our nodes to run out of compute resources, which causes failed Deployments and latency for our actively running applications.

One thing that's certain is that Kubernetes clusters are susceptible to a wide range of different failure modes across multiple levels of operation, from the underlying infrastructure all the way up to the applications running in our containers. In order to address these failure modes, we first need a method of identifying them. This is where we can use Chaos Engineering.

Identifying and addressing failure modes with Chaos Engineering

Chaos Engineering is the science of performing intentional experimentation on a system by injecting precise and measured amounts of harm, observing how the system responds, and using our observations to improve the system's resilience. Chaos Engineering provides a structured approach to resiliency testing in order to help us understand how our systems will react (and potentially fail) under different conditions. Performing these tests (called chaos experiments) can reveal previously unseen problems in our clusters so that we can address them before they cause incidents in production.

Chaos Engineering takes a controlled, scientific approach to testing system resiliency. The goal isn't to create chaos, but to mitigate it. Each experiment is well-defined with a clear hypothesis about what we're trying to test, which systems are impacted by the experiment, and a way to stop or revert the experiment in case of an unexpected outcome.

When developing an experiment:

01 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 take a worker node offline, your hypothesis might state that "any Pods running on the node will be rescheduled onto other nodes." The results of the experiment will either prove or disprove your hypothesis.

02 Determine your blast radius, which is the set of systems, applications, and other components that could be affected by this experiment. An experiment with a small blast radius might only affect a single Pod or application, while an experiment with a large blast radius might impact multiple nodes or the entire cluster. We strongly recommend starting with the smallest blast radius possible, and only increasing the blast radius when you have experience running chaos experiments and you feel confident that you can recover from unexpected problems.

03 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.

04 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.

05 Form a conclusion from your results. Does it confirm or refute your hypothesis? Use the results to modify your cluster configuration or Deployments, then repeat the experiment to confirm that your improvements work as expected.

By repeating this process, you'll be able to continuously uncover failure modes, validate your automatic recovery mechanisms, and create a more resilient Kubernetes environment. Over time, you can increase your blast radius and push your systems a little harder to make sure your cluster can handle even the most stressful conditions.

Putting Chaos Engineering Into Practice

Now that we know what Chaos Engineering is and how to apply it to Kubernetes, let's walk through some of the most important experiments to get started with. For these examples, we'll be using a three-node cluster running on Amazon Elastic Kubernetes Service (EKS) with the Gremlin Kubernetes client deployed. Our cluster will also run Online Boutique, a web-based e-commerce application. Lastly, we'll run the experiments using Gremlin.

TRY IT OUT FOR YOURSELF

You can follow along with these experiments if you have a Gremlin account and a Kubernetes cluster for testing. If you don't have an account, you can click this link to sign up for a Gremlin Free account. You can use any standard or managed Kubernetes service including EKS, Azure Kubernetes Service, or Google Kubernetes Engine. You can also use a local Kubernetes cluster tool like K3s, kind, microk8s, or minikube.

ACTIVATE ACCOUNT

EXPERIMENT #1

Using a Pod failure to test a Deployment or ReplicaSet configuration

Pod failures are among the most common types of failures in Kubernetes. Although microservices are designed to be loosely coupled, a failure of one Pod can directly impact the performance and stability of another Pod. In an ideal scenario, Kubernetes will instantly detect a failed Pod and automatically deploy a replica to replace it in order to maintain the minimum number set in our Deployment.

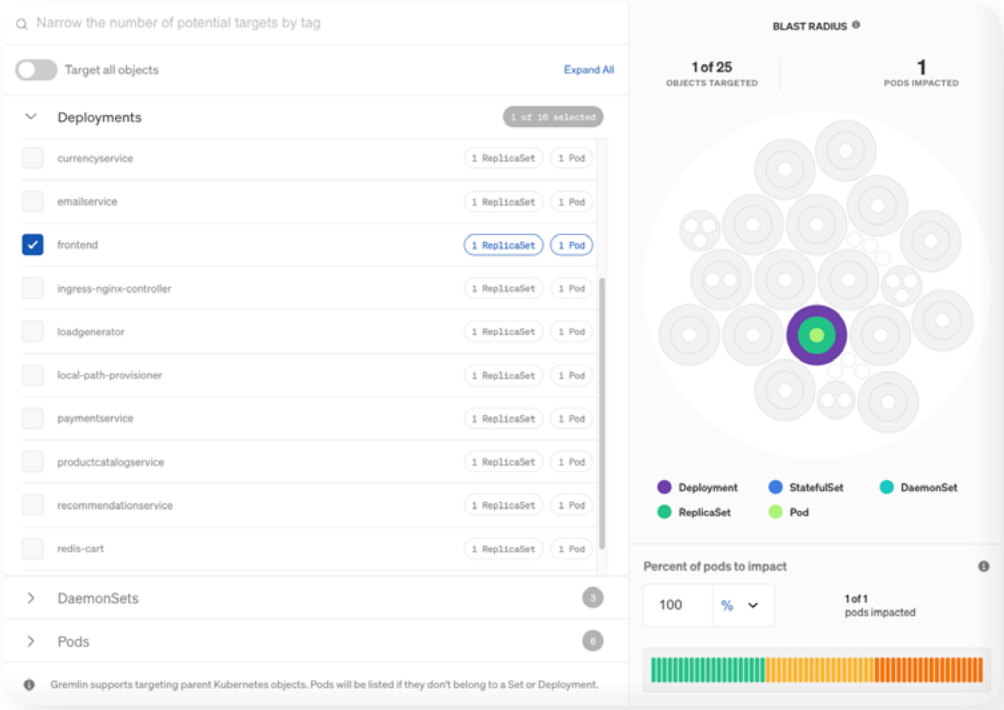
To ensure that this is the case, we'll run an experiment to terminate a running Pod and monitor Kubernetes to verify that the new replica gets deployed successfully. We'll run this experiment on the Online Boutique's frontend Pod, which is responsible for serving the website to users.

Hypothesis

When the frontend Pod shuts down, Kubernetes automatically deploys a replica and routes traffic from the old Pod.

Blast radius

Our blast radius is the frontend Deployment. This includes all three Pod replicas running on each of our nodes.



Metrics

When performing the experiment, we'll monitor our Online Boutique webpage. If it fails to load, then our cluster didn't perform as expected. We might also want to monitor these metrics, as these help us understand the broader impact that this failure is having on our cluster:

- Delays between Pod failure and replica availability.
- Changes in application performance metrics (APM) such as latency, responsiveness, or availability.
- Increased load to other Pods and hosts.

Running the experiment

We'll run a shutdown attack on the frontend Deployment. The shutdown attack will immediately terminate all three Pods in the Deployment. While the attack is running, we'll use kubectl to check the status of the frontend Pods:

```
$ kubectl get pods -lapp=frontend
NAMESPACE     NAME                            READY    STATUS
default       frontend-7d8cfc75b5-2kgdk       1/1      Running
default       frontend-7d8cfc75b5-6p5ks       1/1      Running
default       frontend-7d8cfc75b5-ns5f2       1/1      Running
```

Shortly after running the shutdown attack, new frontend Pods appear:

```
NAMESPACE     NAME                            READY    STATUS
default       frontend-7d8cfc75b5-6xscl       1/1      Running
default       frontend-7d8cfc75b5-stv96       1/1      Running
default       frontend-7d8cfc75b5-z4ll5       1/1      Running
```

If we refresh the page, we'll see that it loads normally with no noticeable downtime or errors. We've confirmed our hypothesis and can confidently say that our cluster can handle Pod failures.

Conclusions

This experiment proved our hypothesis correct, but that doesn't necessarily mean we're 100% resilient. There are other factors that could affect Pod replication. For example, what if we decide to implement label selectors, affinity rules, tolerations, or similar assignment strategies? Do our Pods still get rescheduled reliably? StatefulSets add additional challenges due to each Pod being uniquely identified and ordered. We should consider how these features might influence Pod replication and our ability to recover.

We should also check our metrics after each experiment for any unusual or unexpected results. Was there a notable impact on application performance or availability while the Pods were down? If so, we should increase the number of replicas so that at least one Pod is always available. We can also add a cache in front of our frontend Deployment so we can serve requests even if our frontend is down. Lastly, if two of our three frontend Pods fail, then all of the traffic will be redirected to the third remaining Pod, creating a thundering herd. We should consider increasing the minimum number of replicas or even increasing the capacity of the cluster just to avoid a new kind of failure caused by traffic surges.

EXPERIMENT #2

Exhausting cluster resources to validate stability under load

Computing resources are finite (even in the cloud), and running out of CPU, memory, storage, or some other critical resource can result in system or application failure. Several factors can cause resource exhaustion including traffic surges, slow hardware, poor code optimization, and memory leaks. If one of your nodes is exhausted, Kubernetes will try to intelligently schedule Pods around them, but if the entire cluster is exhausted, it might start evicting lower-priority Pods. Even worse, uncapped resource consumption could prevent the node from communicating with the rest of the cluster, causing it to fail completely.

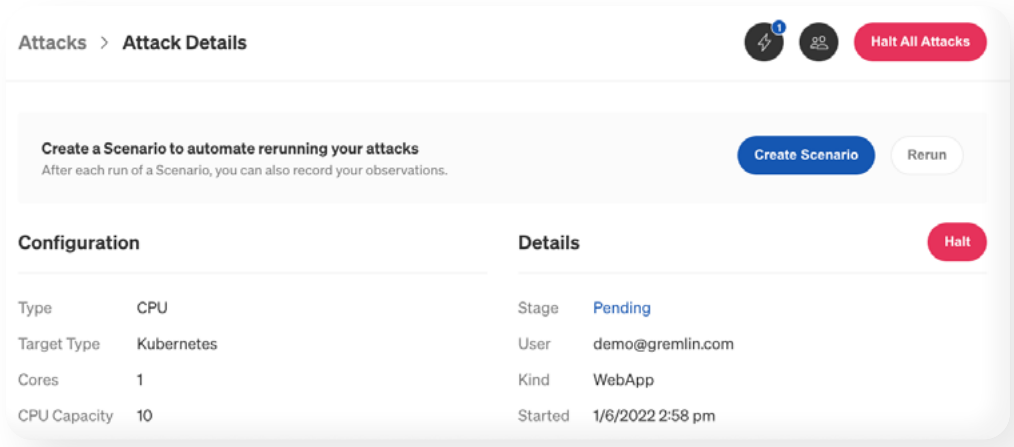
One way to mitigate this risk is to use autoscaling to add additional nodes to your cluster once resource usage hits a certain threshold. However, we still need to confirm that this process works as expected and is sufficient to prevent resource exhaustion under stressful workloads.

Hypothesis

If we exhaust our cluster's RAM, new Deployments will fail and Kubernetes will start evicting lower-priority Pods. In a cloud Kubernetes environment, this will also trigger the Cluster Autoscaler and provision a new worker node.

Blast radius

Our blast radius is the entire cluster. Because the blast radius is so large, we want to have a backup plan in place in case something goes wrong. Gremlin provides a Halt button that we can click to immediately stop and revert the attack if needed, as shown in the following image.



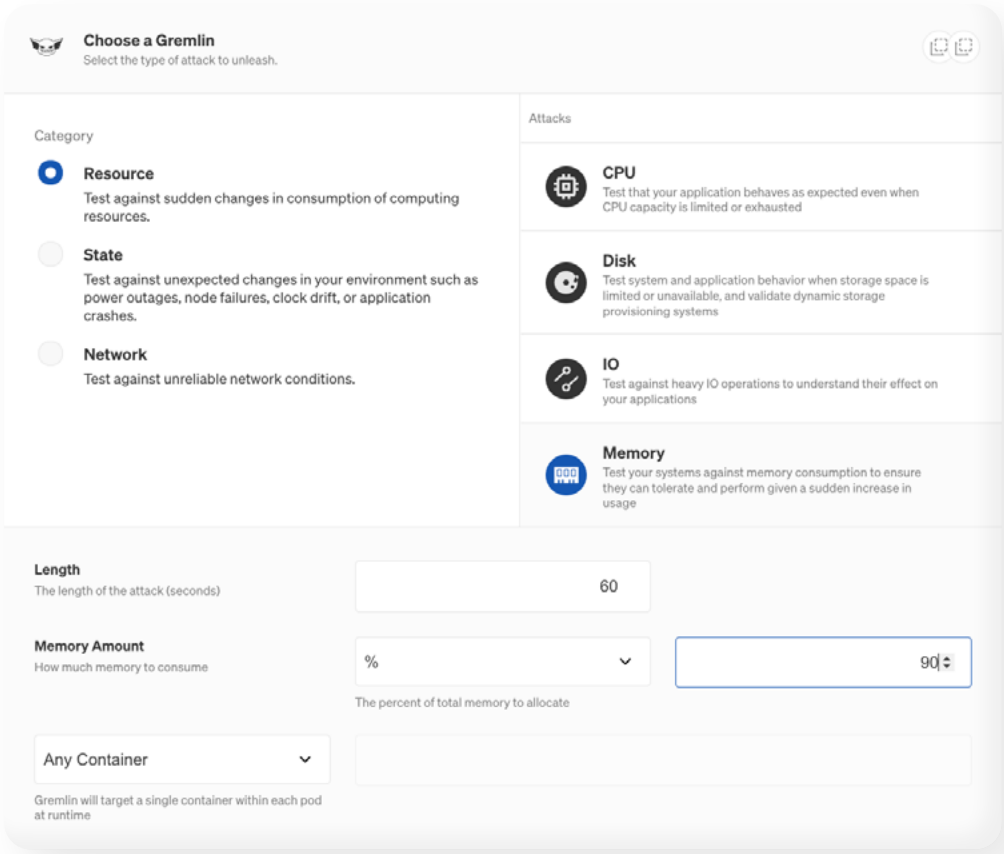
Metrics

Our primary metrics are memory usage and node count. We should also monitor:

- Any Pods evicted or OOMKilled.
- The amount of delay between reaching our consumption threshold and the Cluster Autoscaler deploying a new node.

Running the experiment

We'll use a memory attack to increase the RAM usage on all three of our worker nodes to 90%. We'll then use kubectl to increase the number of `paymentservice` and `productcatalogservice` Pods from 1 to 3. Lastly, we'll use kubectl again to check the state of our Pods.




```
kubectl scale --replicas=3 deploy/paymentservice
kubectl scale --replicas=3 deploy/productcatalogservice
```

Now let's use kubectl to check our Pods:

```
kubectl get pods -A
```

The new Pods are visible, but they're in a Pending state. Since there's no RAM left, these Pods will remain Pending until we either scale the Deployment back down or free

up some RAM. We'll also see that Kubernetes has started evicting Ingress Pods. An Ingress is required for customers to access our website, and without it, we can't serve requests at all. This is an unexpected but significant outcome that we need to address.

```
NAME                                          READY   STATUS
nginx-ingress-default-backend-659bd647bd-jzszx  0/1     Evicted
paymentservice-84ffc75c55-d8ksx                 1/1     Running
paymentservice-84ffc75c55-l2bhn                 0/1     Pending
paymentservice-84ffc75c55-qqlmm                 0/1     Pending
productcatalogservice-d564bdf4c-drqtj           1/1     Running
productcatalogservice-d564bdf4c-mf6b9           1/1     Running
productcatalogservice-d564bdf4c-xfl5t           0/1     Pending
```

Eventually our cluster autoscaler does add a new node to the cluster, which restarts the Ingress Pod and moves the Pending Pods into the Running state, but this process takes several minutes. In the meantime, our site is completely inaccessible to our users.

Conclusions

To avoid this issue, we can increase the amount of RAM in our nodes, lower the memory threshold for autoscaling, or use affinity rules to schedule memory-intensive workloads onto dedicated high-capacity nodes. Alternatively, we could limit the total computing resources that a Pod can use by setting resource quotas. It takes some fine-tuning to find the right balance between preventing overconsumption and not limiting your ability to scale. Throughout this process, we should keep running resource experiments to validate that our changes are effective.

EXPERIMENT #3

Measuring the Impact of Latency

Although microservices are meant to be loosely coupled, this isn't always the case. Problems in one Pod can still impact other Pods: for example, if one Pod is unresponsive to network requests, it could cause increased latency and timeouts in Pods that communicate with it. We saw that Kubernetes can detect and replace failed Pods fairly easily, but detecting and resolving underperforming Pods is much less straightforward.

One possible solution is to have multiple replicas of a Pod and use a load balancer to reroute traffic from slow replicas to faster replicas. Kubernetes supports the use of external container-native load balancing solutions–such as the one provided by Google Kubernetes Engine–that can intelligently redistribute traffic based on Pod responsiveness and latency. Not all clusters have this capability though, so we should run chaos experiments to determine the impact of latency on our application as a whole.

For this experiment, we want to verify that our website is still responsive even if our Redis instance is slow. Online Boutique uses Redis to store and track customer shopping cart state, and since every page displays the shopping cart, we should be able to notice any changes in latency right away.

You can also use liveness probes to monitor for unresponsive Pods.

Hypothesis

Increasing the latency of a downstream Pod (in this case, redis-cart) will cause poor performance for upstream Pods. If an external load balancer is active, traffic will automatically reroute to more responsive Pods.

Blast radius

Our blast radius is a single Deployment (the redis-cart Deployment).

Metrics

Our primary metric is application response time. Specifically, we'll measure the average response time and requests per second for one minute before and during the experiment, then compare the results. In addition, we might also want to track:

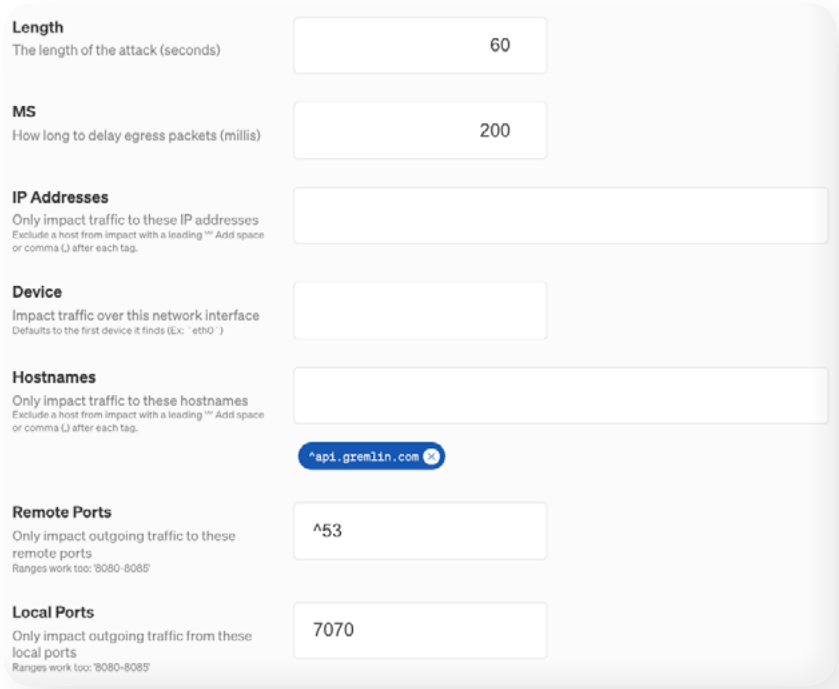
- APM metrics, such as the request rate, error rate, and response time.
- Web and user experience (UX) metrics, such as time to first byte (TTFB), first paint, and page load time.

To measure our request rate and time, we'll use the ApacheBench tool to send requests to our frontend:

```
Requests per second:    5.78 [#/sec] (mean)
Time per request:       173.041 [ms] (mean)
Time per request:       173.041 [ms] (mean, across all concurrent requests)
Transfer rate:          89.94 [Kbytes/sec] received
```

Running the experiment

We'll use a latency attack to introduce latency directly into the redis-cart Deployment. We'll delay traffic by 200 ms for 60 seconds over port 7070, which is the default port used by Redis.



While the attack is running, we'll use ApacheBench again to benchmark performance:

```
Requests per second:    2.54 [#/sec] (mean)
Time per request:       393.712 [ms] (mean)
Time per request:       393.712 [ms] (mean, across all concurrent requests)
Transfer rate:          39.52 [Kbytes/sec] received
```

Before the experiment, we were able to complete 5.7 requests per second with an average time of 173ms per request. With just 200ms of latency, we only completed 2.5 requests per second with an average time of 393 ms per request.

Conclusions

You might have noticed that while we only added 200ms of latency, the average request time increased by more than 200ms. This means that any latency added by Redis has an even larger impact on our site as a whole. We should pay close attention to increases in Pod latency, especially for Pods like Redis which many other Pods depend on.

Seeing the impact that Pod latency has on our application will help with planning liveness checks, load balancing rules, and Pod replication strategies. In turn, these will help prevent poor user experiences by spreading out traffic and reducing the amount of time customers spend waiting for unresponsive services. For example, the redis Pod uses the grpc_health_probe utility and will report an unhealthy status after 5 seconds with no response. When Kubernetes detects this status, it deploys a new Pod to take its place. We might want to reduce this timeout period to 3 or even 1 second to avoid making users wait too long.

If latency becomes a recurring issue, try looking at your network by investigating issues such as bandwidth constraints, packet loss, or overzealous firewall rules. If you use a managed Kubernetes service, consult with your cloud provider on using an external load balancer. When you feel comfortable experimenting on one Deployment, try increasing your blast radius to multiple Deployments, or even to an entire node. Keep in mind that running a network attack on a node will impact all traffic, including cluster communications, unless you whitelist specific ports or hostnames.

EXPERIMENT #4

Validating a High Availability (HA) cluster

A normal Kubernetes cluster only has a single master node. This node is responsible for running services essential to the cluster, such as the Pod scheduler, configuration management service, and API server. However, this creates a single point of failure; if the master node goes down, the cluster becomes degraded. Our applications will likely keep running, but we won't be able to manage Deployments, add nodes, monitor cluster metrics, etc.

Kubernetes High Availability (HA) clusters add resiliency by replicating the cluster's control plane across multiple nodes. With an HA cluster, multiple nodes share the role of master node so that even if one fails, the cluster remains available and intact.

Running chaos experiments on a HA cluster ensures that this replication process is working as intended. Even if we're not using a HA cluster and are instead using a normal single-master cluster, running these experiments helps us test our incident response and disaster recovery procedures in a safe and controlled way. That being said, we should only experiment on non-production single-master clusters unless we have a proven recovery strategy in place.

Hypothesis

If a master node in an HA cluster fails, the cluster will continue to run as normal.

Blast radius

This experiment only targets Kubernetes master nodes directly, but this could impact the entire cluster.

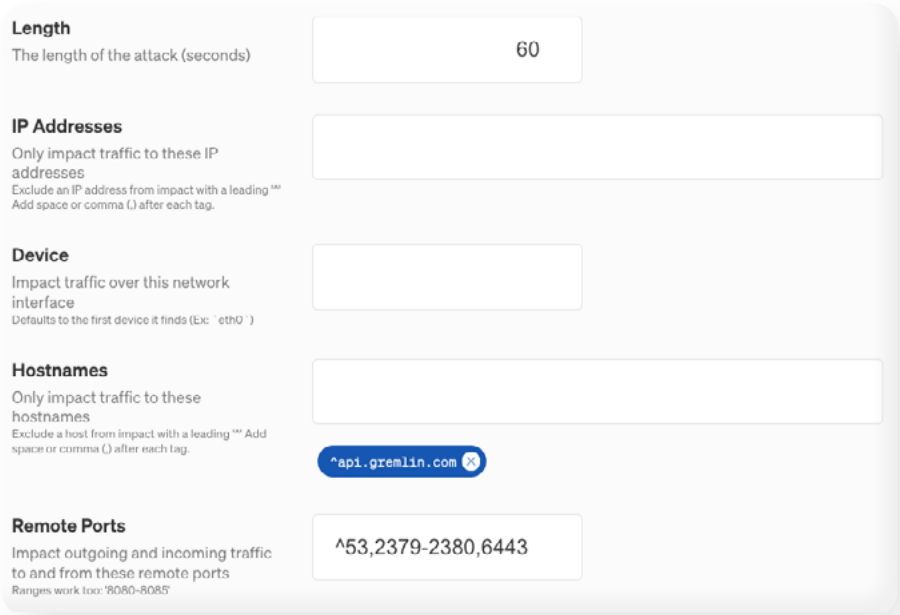
Metrics

The primary metric in this experiment is availability, which we can monitor by using kubectl. We can also test that:

- Remaining masters are fully replicated: they report the same nodes, workloads, and configuration policies.
- Any new Deployments are scheduled successfully.

Running the experiment

We'll use a blackhole attack to block all cluster network traffic on one of our master nodes. We'll run this test for 60 seconds and specifically block ports 2379 and 2380 (for etcd), as well as port 6443 (for the API server). Blocking traffic over these ports effectively removes the node from the cluster.



Alternatively, we could run a shutdown attack like in Experiment #1. The benefit of using a blackhole attack is that the node returns to normal immediately after the experiment ends, whereas we'd need to reboot it after a shutdown.

While the experiment is running, let's use kubectl to get the status of the node. We should get a response showing that the node is in the NotReady state:

```
$ kubectl get node k8s-demo-pool-dd9c
NAME                 STATUS     ROLES                  AGE    VERSION
k8s-demo-pool-dd9c   NotReady   control-plane,master   32m    v1.17.0
```

Since we got a response from kubectl, we can assume that our other master nodes were still available.

Conclusions

We saw that despite losing a master node, the cluster was still accessible and responsive, meaning we configured our HA cluster properly. Our load balancers routed API traffic around the failed node, which meant administration tools like kubectl and the Kubernetes Dashboard kept working as normal and we didn't need to reschedule any workloads.

In case of a master failure, consider removing the old replica and replacing it with a new one in the same zone. The cluster state must be copied to the new instance, which could take some time for larger clusters. When you scale up your chaos experiment to remove a master, track the amount of time it takes for the new instance to come online and factor it into your disaster recovery plan. In addition, consider building your HA cluster with at least three masters, and place master replicas in different zones. This will provide resilience against zone outages.

EXPERIMENT #5

Testing Storage Volume Limits

Many applications–especially stateful applications–require access to persistent storage. Unlike CPU and RAM, storage can take a number of forms including local storage, cloud storage, and temporary storage. For example, an awsElasticBlockStore volume allows Pods to write an AWS EBS volume, while a local volume allows Pods to write to a storage location on the host node.

However, storage is still a finite resource. Pods that use excessive amounts of storage reduce the amount available to other Pods, and excessive consumption of elastic storage increases operating costs. This can be prevented through the use of Persistent Volumes (PVs) and ephemeral storage limits, which allow setting limits on the amount of storage space that can be assigned to a Pod.

For this experiment, we want to ensure that the amount of storage space a Pod uses never exceeds the limits defined in the Deployment. In addition, we'll see how our workloads respond to an out-of-disk scenario and whether it handles the situation gracefully, crashes, or causes data loss. We'll target our redis-cart Deployment, since it's the only Deployment that uses volumes.

We'll first need to set a limit and redeploy Redis. In this example, we'll set it to 1 GiB. First, download the Redis deployment manifest from the Online Boutique GitHub repository:

```
wget https://raw.githubusercontent.com/GoogleCloudPlatform/microservices-demo/master/kubernetes-manifests/redis.yaml
```

We'll open this file in a text editor and change the line containing `emptyDir: {}` to the following:

```
emptyDir: { sizeLimit: 1Gi  }
```

Next, we'll deploy these changes:

```
kubectl apply -f redis.yaml
```

Hypothesis

If Redis tries to store more than 1 GiB of data, Kubernetes will prevent it from writing any additional data.

Blast radius

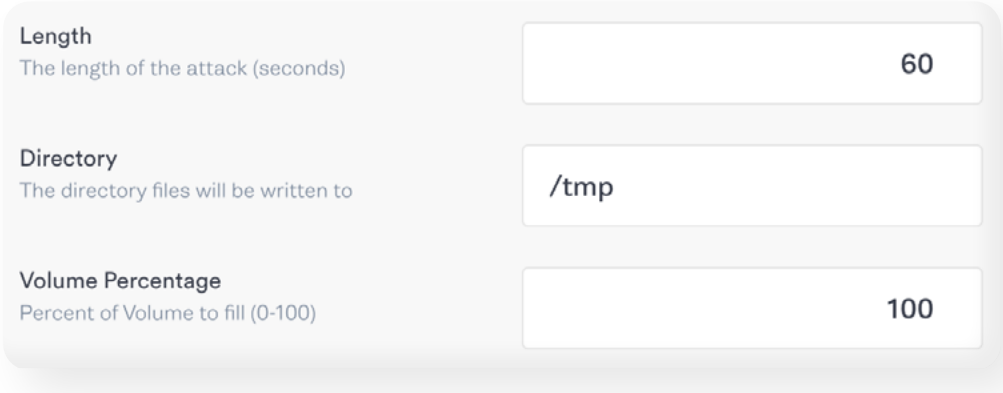
We'll be targeting the redis-cart Deployment, but because the experiment is consuming local storage space, our blast radius includes the node that the Deployment is running on, including all other Pods running on that node.

Metrics

Our primary metric is disk usage, which we can measure using the du command. We'll also want to monitor our Pods to make sure they're still in the Running state.

Running the experiment

We'll use a disk attack to consume 100% of all available disk space by writing to the /tmp directory. We'll monitor the status of the Deployment using kubectl.



When we start the attack and look at our Pods, we see nothing out of the ordinary. But if we check disk usage on the node where the redis-cart Deployment is running, we'll see that an unusually large amount of space is being used by the /var directory. By default, Kubernetes stores data for emptyDir under this directory.

```
$ du -sh /var
31G     /var
```

This is clearly beyond our 1 GiB limit, so let's halt the attack. Right away, disk usage drops back down to around 6 GB. This tells us that Kubernetes did not prevent our Deployment from using more than 1 GiB, and if we let the attack keep running, it probably would have consumed all available disk space.

We can actually test this theory by repeating the attack and letting it run without interruption. If we do so and look at kubectl, we see that the node starts to experience disk pressure:

```
$ kubectl get node k8s-demo-pool-dd97
Events:
  Type     Reason                     Age                From                                                Message
  ----     ------                     ----               ----                                                -------
  Normal   NodeHasDiskPressure       3m6s (x2 over 12m)  Node k8s-demo-pool-dd97 status is now: NodeHasDiskPressure
  Warning  EvictionThresholdMet      2m37s (x3 over 12m) Attempting to reclaim ephemeral-storage
```

Prolonged pressure triggers the Pod eviction policy, and Kubernetes starts by attempting to evict the Gremlin Kubernetes client. This is a safety mechanism built into Kubernetes to prevent a container from crashing the system.

```
NAMESPACE     NAME                                        STATUS
default       nginx-ingress-controller-7796b46979-gc7pw   Evicted
default       nginx-ingress-default-backend-659bd647bd-6wd6f   Running
default       nginx-ingress-default-backend-659bd647bd-jzszx   Evicted
default       paymentservice-84ffc75c55-d8ksx             Running
default       productcatalogservice-d564bdf4c-drqtj       Running
gremlin       chao-5776ffdf87-2cnw6                       Evicted
gremlin       chao-5776ffdf87-4pjh2                       Evicted
gremlin       chao-5776ffdf87-77nq2                       Evicted
```

This only confirms our original finding that Kubernetes isn't enforcing disk usage limits, and as a bonus, we now know how our cluster will respond in a low disk scenario.

Conclusions

We saw that Redis can in fact use more than the 1 GiB limit that we set, and if left unchecked, causes a number of other issues with our cluster and even the Gremlin client. Not only did we refute our hypothesis, but we uncovered a much larger issue. We should review our Deployment manifest and cluster configuration to see why this limit isn't being enforced, make changes, then repeat this experiment to validate that our fix works.

We can further refine our storage rules by setting PV limits or filesystem-level quotas and continuing to monitor disk usage to see how application storage usage changes over time. We'll also want to stay aware of node-specific volume limits, which might impact future deployments.

We can also use disk attacks (and IO attacks) to test our application's exception handling and recovery mechanisms. For example, how does an unresponsive storage device affect the responsiveness of our application? Testing and monitoring storage performance can also uncover lower-level hardware issues, such as failing hard drives (for local storage) or poor quality network conditions (for network attached storage).

Conclusion

Despite its numerous recovery mechanisms, Kubernetes is a complex system, and applications deployed on it are still prone to failure. The only way to ensure a successful and reliable adoption is with frequent, proactive failure testing. Chaos Engineering provides a framework for uncovering reliability problems in a safe, effective, and scientific way, allowing you to both verify your resilience mechanisms and uncover potentially unknown problems.

Chaos Engineering is not about breaking systems and seeing what happens, but rather strategically and scientifically injecting failure into your applications and infrastructure in order to see how they respond. Preparing for failure today can reduce or prevent the impact of failures and outages in the future, saving your organization time, engineering effort, and lost customer goodwill.

Gremlin

Gremlin is the enterprise Chaos Engineering platform on a mission to help build a more reliable internet. Their solutions turn failure into resilience by offering engineers a fully hosted SaaS platform to safely experiment on complex systems, in order to identify weaknesses before they impact customers and cause revenue loss.

ADDITIONAL RESOURCES

- Kubernetes documentation
- How to install and use Gremlin with Kubernetes
- If you're adopting Kubernetes, you need Chaos Engineering