opensource.com

Chaos Engineering for Kubernetes

by Jessica Cherry

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

Jessica Cherry

Jessica Cherry is a tech nomad, working on just about anything she can find. She's an avid evangelist of silo-prevention in the IT space, and the importance of information sharing across all teams. She's a believer in educating all, and in open source development.

Jessica Cherry calls herself a lover of all things tech, and it shows in the breadth of her articles. She's written about the latest developments in Kubernetes, CI/CD and Jenkins, Open Street Maps, as well as the "mundane" things we all need at one time or another, like filing taxes, video conferencing, and wedding planning using open source.

Get started with Kubernetes using chaos engineering

Chaos engineering is part science, part planning, and part experiments. It's the discipline of experimenting on a system to build confidence in the system's capability to withstand turbulent conditions in production.

This introductory article explains the basics of how chaos engineering works.

How do I get started with chaos engineering?

In my experience, the best way to start chaos engineering is by taking an incident that has happened before in production and using it as an experiment. Use your past data, make a plan to break your system in a similar way, create a repair strategy, and confirm the outcome turns out exactly how you want. If your plan fails, you have a new way to experiment and move forward toward a new way to handle issues quickly.

Best of all, you can document everything as you go, which means, over time, your entire system will be fully documented so that anyone can be on call without too many escalations and everyone can have a nice break on weekends.

What do you do in chaos engineering?

Chaos engineering has some science behind how these experiments work. I've documented some of the steps:

1. **Define a steady state**: Use a monitoring tool to gather data about what your system looks like functionally when there are no problems or incidents.
2. **Come up with a hypothesis or use a previous incident:** Now that you have defined a steady state, come up with a hypothesis about what would happen (or has happened) during an incident or outage. Use this hypothesis to generate a series of theories about

what could happen and how to resolve the problems. Then you can start a plan to purposely cause the issue.

3. **Introduce the problem:** Use that plan to break your system and begin real-world testing. Gather your broken metrics' states, use your planned fix, and keep track of how long it takes before you reach a resolution. Make sure you document everything for future outages.
4. **Try to disprove your own hypothesis:** The best part of experimenting is trying to disprove what you think or plan. You want to create a different state, see how far you can take it, and generate a different steady state in the system.

Make sure to create a control system in a steady state before you generate the broken variables in another system. This will make it easier to spot the differences in various steady states before, during, and after your experiment.

What do I need for chaos engineering?

The best tools for beginning chaos engineering are:

- Good documentation practices
- Monitoring system to capture your system in a steady and a non-steady state
  - Grafana
  - Prometheus
- Chaos engineering tools
  - Chaos mesh
  - Litmus
- A hypothesis
- A plan

Go forth and destroy

Now that you have the basics in hand, it's time to go forth and destroy your system safely. I would plan to start causing chaos four times a year and work toward monthly destructions.

Chaos engineering is good practice and a great way to keep your internal documentation up to date. Also, new upgrades or application deployments will be smoother over time, and your daily life will be easier with Kubernetes administration.

Start monitoring your Kubernetes cluster with Prometheus and Grafana

In my introductory, one of the main things I covered was the importance of getting the steady state of your working Kubernetes cluster. Before you can start causing chaos, you need to know what the cluster looks like in a steady state.

This article demonstrates how to get those metrics using Prometheus and Grafana. This walkthrough was written on Pop!_OS 20.04, Helm 3, Minikube 1.14.2, and Kubernetes 1.19.

Configure Minikube

Install Minikube in whatever way makes sense for your environment. If you have enough resources, I recommend giving your virtual machine a bit more than the default memory and CPU power:

```
$ minikube config set memory 8192
❕  These changes take effect upon a minikube delete and then a minikube start
$ minikube config set cpus 6
❕  These changes take effect upon a minikube delete and then a minikube start
```

Then start and check your system's status:

```
$ minikube start
minikube v1.14.2 on Debian bullseye/sid
minikube 1.19.0 is available! http://github.com/kubernetes/minikube
To disable this notice: 'minikube config set WantUpdateNotification false'
Using the docker driver based on user configuration
Starting control plane node minikube in cluster minikube
Verifying Kubernetes components...
Done! kubectl is now configured to use "minikube" by default
$ minikube status
```

```
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Install Prometheus

Once the cluster is set up, start your installations. Install Prometheus. First, add the repository in Helm:

```
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-
charts
"prometheus-community" has been added to your repositories
```

Then install your Prometheus Helm chart. You should see:

```
$ helm install prometheus prometheus-community/prometheus
NAME: prometheus
LAST DEPLOYED: Sun May  9 11:37:19 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The Prometheus server can be accessed via port 80 on the following DNS name from
within your cluster:
prometheus-server.default.svc.cluster.local
```

Get the Prometheus server URL by running these commands in the same shell:

```
$ export POD_NAME=$(kubectl get pods --namespace default \
-l "app=prometheus,component=server" -o jsonpath="{.items[0].metadata.name}")
$ kubectl --namespace default port-forward $POD_NAME 9090
```

You can access the Prometheus Alertmanager on port 80 on this DNS name from within your cluster:

```
prometheus-alertmanager.default.svc.cluster.local
```

Get the Alertmanager URL by running these commands in the same shell:

```
$ export POD_NAME=$(kubectl get pods --namespace default \
-l "app=prometheus,component=alertmanager" \
-o jsonpath="{.items[0].metadata.name}")
$ kubectl --namespace default port-forward $POD_NAME 9093
WARNING: Pod Security Policy has been moved to a global property.
You can access the Prometheus PushGateway via port 9091 on this DNS name from
within your cluster: prometheus-pushgateway.default.svc.cluster.local
```

Get the PushGateway URL by running these commands in the same shell:

```
$ export POD_NAME=$(kubectl get pods --namespace default \
-l "app=prometheus,component=pushgateway" \
-o jsonpath="{.items[0].metadata.name}")

$ kubectl --namespace default port-forward $POD_NAME 9091
For more information on running Prometheus, visit https://prometheus.io
```

Check to confirm your pods are running:

```
$ kubectl get pods -n default
NAME                        READY STATUS RESTARTS  AGE
prometheus-alertmanager-ccf   2/2 Running   0      3m22s
prometheus-[...]metrics-685b  1/1 Running   0      3m22s
prometheus-node-exporter-mfc  1/1 Running   0      3m22s
prometheus-pushgateway-74cb6  1/1 Running   0      3m22s
prometheus-server-d9f[...]jw  2/2 Running   0      3m22s
```

Next, expose your port on the Prometheus server pod so that you can see the Prometheus web interface. To do this, you need the service name and port. You also need to come up with a name to open the service using the Minikube service command.

Get the service name for `prometheus-server`:

```
$ kubectl get svc -n default

NAME           TYPE      CLUSTER-IP  EXTERNAL-IP     PORT(S)
kubernetes     ClusterIP 10.96.0.1      <none>       443/TCP
prom[...]alert ClusterIP 10.106.68.12   <none>        80/TCP
pr[...]metrics ClusterIP 10.104.167.239 <none>      8080/TCP
pr[...]xporter ClusterIP None           <none>      9100/TCP
pr[...]gateway ClusterIP 10.99.90.233   <none>      9091/TCP
pr[...]-server ClusterIP 10.103.195.104 <none>      9090/TCP
```

Expose the service as type `Node-port`. Provide a target port of `9090` and a name you want to call the server. The node port is the server listening port. This is an extract of the Helm chart:

```
## Port for Prometheus Service to listen on
port: 9090
```

The command is:

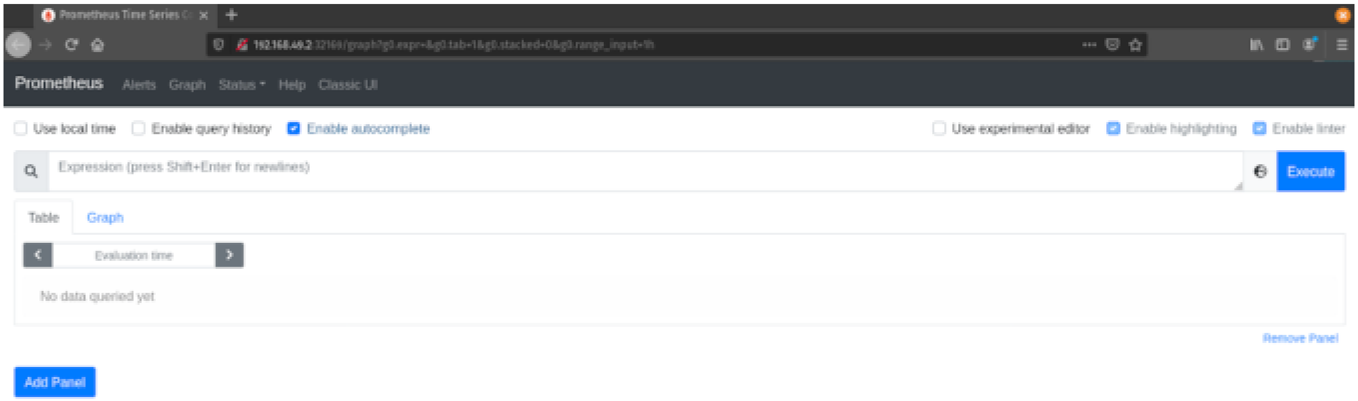
```
$ kubectl expose service prometheus-server --type=NodePort \
--target-port=9090 --name=prom-server
service/prom-server exposed
```

Next, you need Minikube to open the service and browser:

```
$ minikube service prom-server
| NAMESPACE | NAME        | TARGET PORT |  URL
| default   | prom-server |     80      | http://192.168.49.2:32169


Opening service default/prom-server in default browser...
```

Your browser opens, and displays the Prometheus service.



192.168.49.2:32169/graph?g0.expr=&g0.tab=1&g0.stacked=0&g0.range_input=1h

□ Use local time □ Enable query history ☑ Enable autocomplete □ Use experimental editor ☑ Enable highlighting ☑ Enable linter

No data queried yet

(Jess Cherry, CC BY-SA 4.0)

Congratulations! You now have Prometheus installed on your cluster.

Install Grafana

Next, install Grafana and configure it to work with Prometheus. Follow the steps below to expose a service to configure Grafana and collect data from Prometheus to gather your steady state.

Start with getting your Helm chart:

```
$ helm repo add grafana https://grafana.github.io/helm-charts

"grafana" has been added to your repositories
```

Search for your chart:

```
$ helm search repo grafana

NAME                 CHART   VERSION   DESCRIPTION
bitnami/grafana      5.2.11  7.5.5     Grafana is an open[...]
bitnami/grafana-oper 0.6.5   3.10.0    Operator for Grafa[...]
stable/grafana       5.5.7   7.1.1     DEPRECATED
```

Since stable/grafana is depreciated, install bitnami/grafana.

Install the chart

```
$ helm install grafana bitnami/grafana
NAME: grafana
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES: Please be patient while the chart is being deployed.
```

Get the application URL:

```
$ echo "Browse to http://127.0.0.1:8080"
kubectl port-forward svc/grafana 8080:3000 &
```

Get the admin credentials:

```
$ echo "User: admin"
echo "Password: $(kubectl get secret grafana-admin --namespace default \
-o jsonpath="{.data.GF_SECURITY_ADMIN_PASSWORD}" | base64 –decode)"
```

As you can see in the Helm installation output, the target port for Grafana is 3000, so you will use that port for exposing the service to see Grafana's web frontend. Before exposing the service, confirm that your services are running:

```
$ kubectl get pods -A

NAMESPACE   NAME                         READY    STATUS    RESTARTS
```

```
default      grafana-6b84bbcd8f-xt6vd    1/1     Running    0
```

Expose the service:

```
$ kubectl expose service grafana --type=NodePort --target-port=3000 --
name=grafana-server

service/grafana-server exposed
```

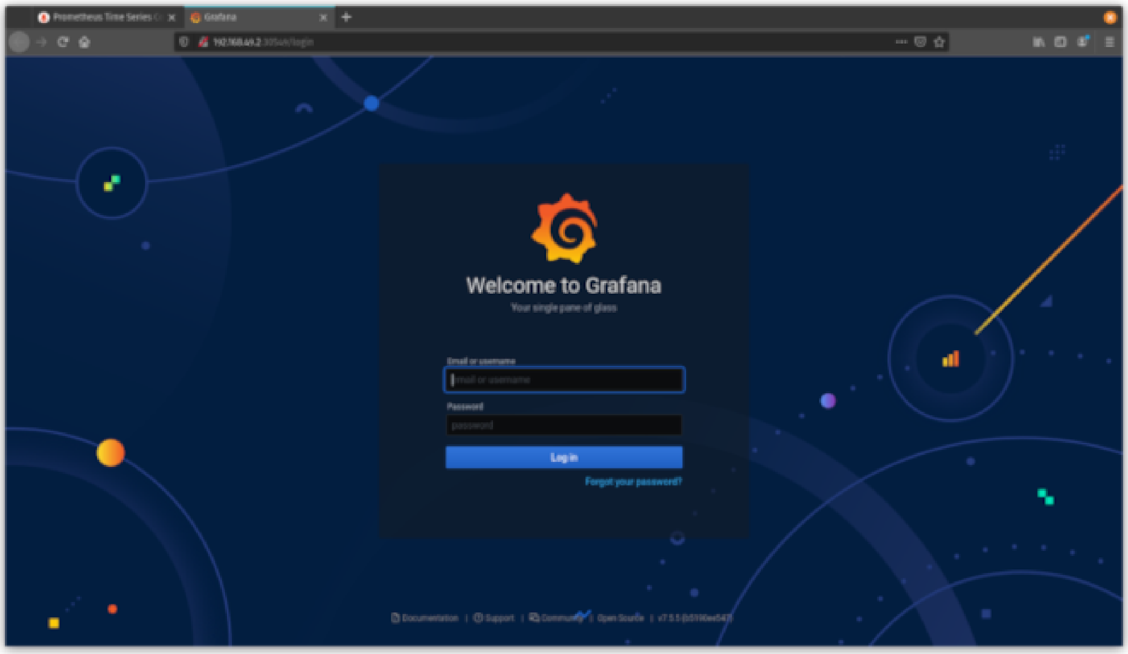
Enable the service to open a browser with a Minikube service:

```
$ minikube service grafana-server

| NAMESPACE | NAME             | TARGET PORT | URL
| default   | grafana-server |        3000 | http://192.168.49.2:30549

Opening service default/grafana-server in default browser...
```

You will see the welcome screen where you can log in.



(Jess Cherry, CC BY-SA 4.0)

Set up credentials to log into Grafana using kubectl. The commands appeared in the installation's output; here are the commands in use:

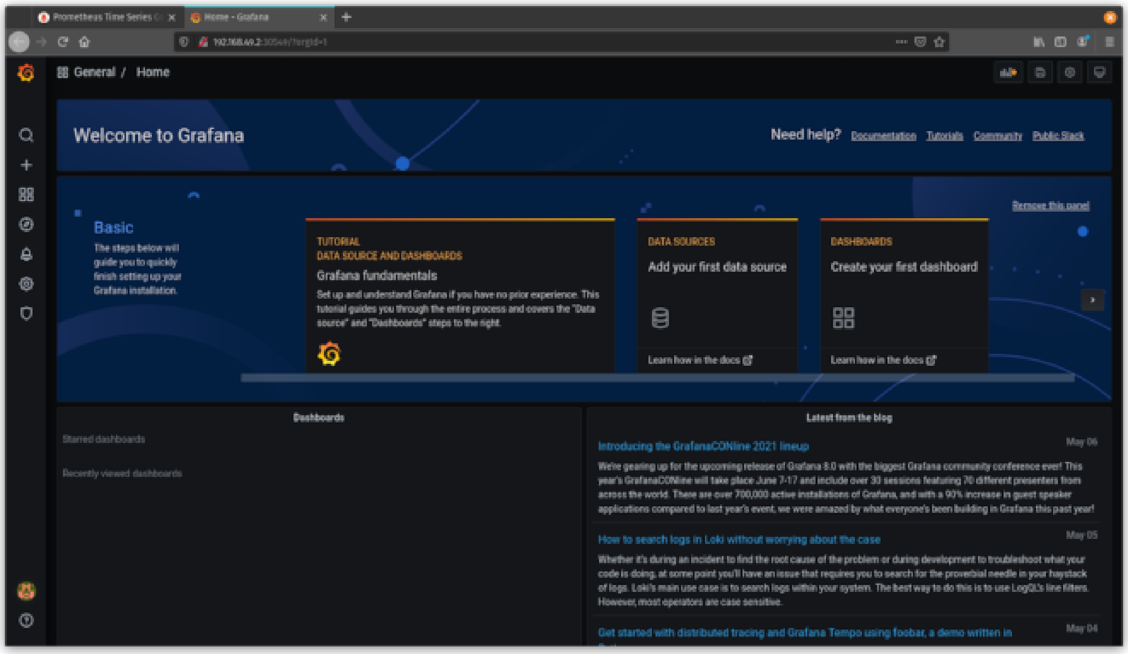
```
$ echo "User: admin"
```

```
User: admin

$ echo "Password: $(kubectl get secret grafana-admin --namespace default \
-o jsonpath="{.data.GF_SECURITY_ADMIN_PASSWORD}" | base64 --decode)"


Password: G6U5VeAejt
```

Log in with your new credentials, and you will see the Grafana dashboard.



The steps below will guide you to quickly finish setting up your Grafana installation.

Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.

Introducing the GrafanaCONline 2021 lineup — May 06

We're gearing up for the upcoming release of Grafana 8.0 with the biggest Grafana community conference ever! This year's GrafanaCONline will take place June 7-17 and include over 30 sessions featuring 70 different presenters from across the world. There are over 700,000 active installations of Grafana, and with a 90% increase in guest speaker applications compared to last year's event, we were amazed by what everyone's been building in Grafana this past year!

How to search logs in Loki without worrying about the case — May 05

Whether it's during an incident to find the root cause of the problem or during development to troubleshoot what your code is doing, at some point you'll have an issue that requires you to search for the proverbial needle in your haystack of logs. Loki's main use case is to search logs within your system. The best way to do this is to use LogQL's line filters. However, most operators are case sensitive.

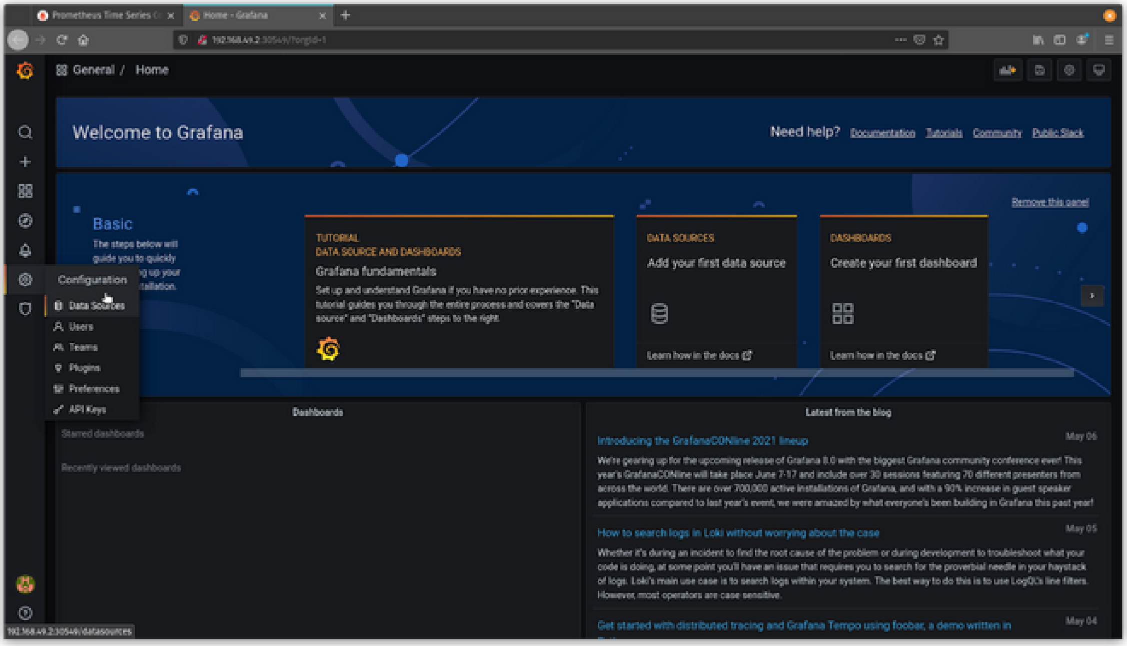
Get started with distributed tracing and Grafana Tempo using foobar, a demo written in — May 04

(Jess Cherry, CC BY-SA 4.0)

Congratulations! You now have a working Grafana installation in your Minikube cluster with the ability to log in. The next step is to configure Grafana to work with Prometheus to gather data and show your steady state.

Configure Grafana with Prometheus

Now that you can log in to your Grafana instance, you need to set up the data collection and dashboard. Since this is an entirely web-based configuration, I will go through the setup using screenshots. Start by adding your Prometheus data collection. Click the **gear icon** on the left-hand side of the display to open the **Configuration** settings, then select **Data Source**.



Welcome to Grafana

Need help?

Basic

The steps below will guide you to quickly ...g up your ...tallation.

TUTORIAL
DATA SOURCE AND DASHBOARDS
Grafana fundamentals

Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.

DATA SOURCES
Add your first data source

DASHBOARDS
Create your first dashboard

Latest from the blog

Introducing the GrafanaCONline 2021 lineup — May 06

We're gearing up for the upcoming release of Grafana 8.0 with the biggest Grafana community conference ever! This year's GrafanaCONline will take place June 7-17 and include over 30 sessions featuring 70 different presenters from across the world. There are over 700,000 active installations of Grafana, and with a 90% increase in guest speaker applications compared to last year's event, we were amazed by what everyone's been building in Grafana this past year!

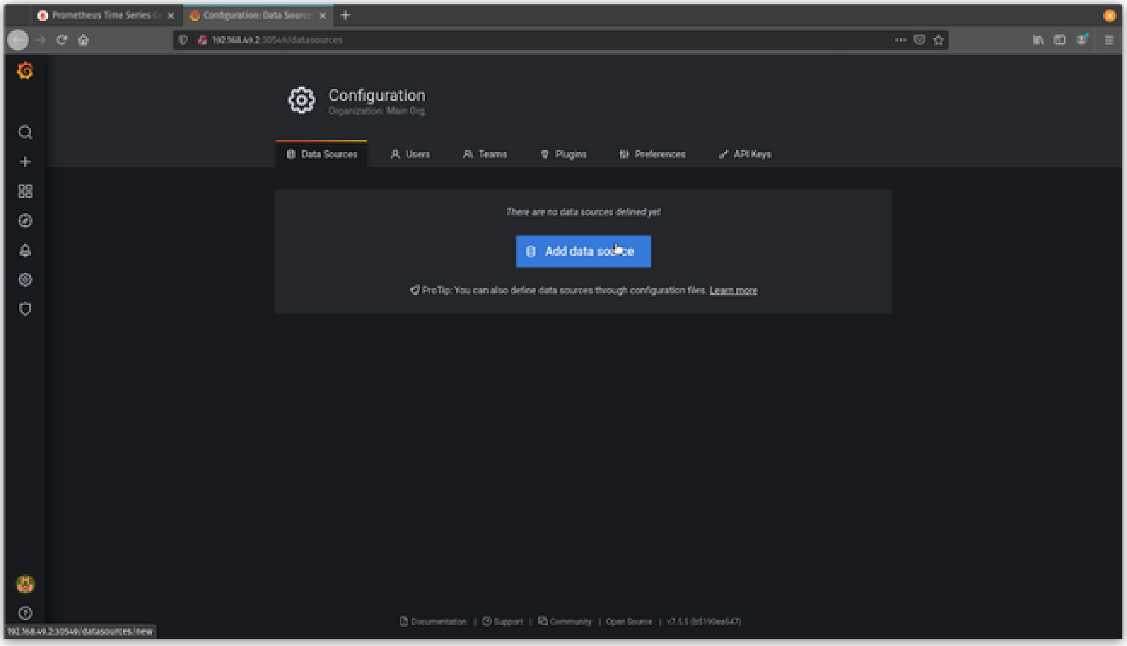
How to search logs in Loki without worrying about the case — May 05

Whether it's during an incident to find the root cause of the problem or during development to troubleshoot what your code is doing, at some point you'll have an issue that requires you to search for the proverbial needle in your haystack of logs. Loki's main use case is to search logs within your system. The best way to do this is to use LogQL's line filters. However, most operators are case sensitive.

Get started with distributed tracing and Grafana Tempo using foobar, a demo written in — May 04

(Jess Cherry, CC BY-SA 4.0)

On the next screen, click **Add data source**.

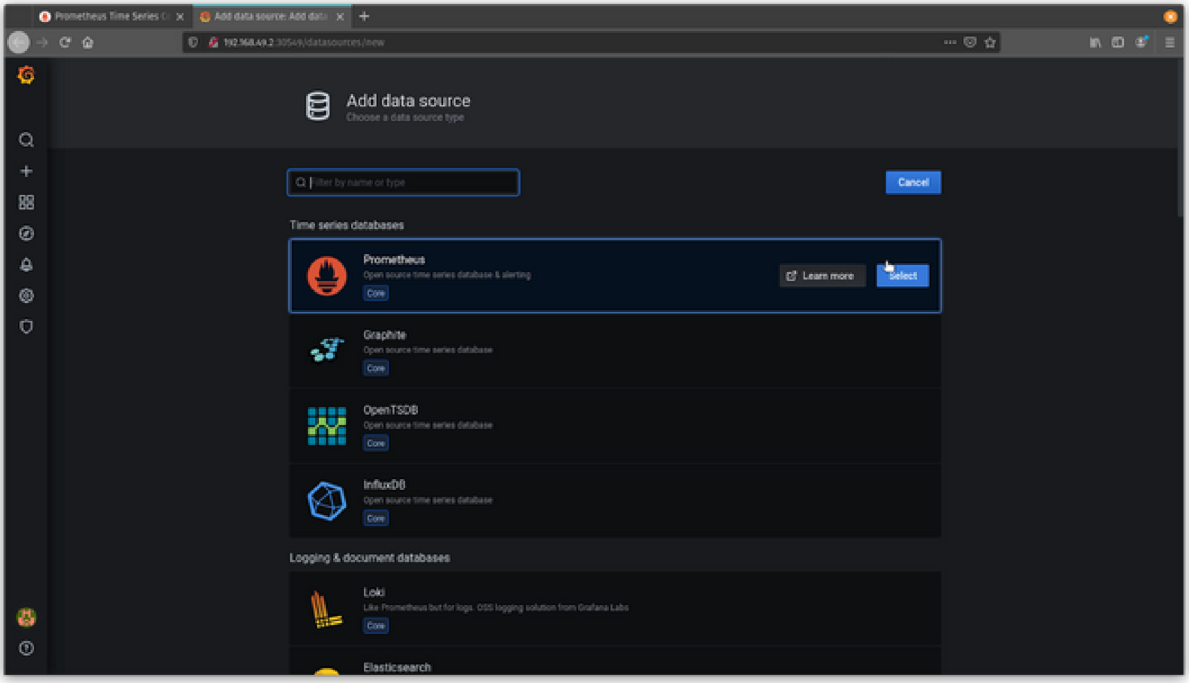

Prometheus Time Series C
Configuration: Data Sources
192.168.49.2:30549/datasources
Configuration
Organization: Main Org.
Data Sources
Users
Teams
Plugins
Preferences
API Keys
Add data source
Learn more
Documentation | Support | Community | Open Source | v7.5.5 (b5190ea547)
192.168.49.2:30549/datasources/new

There are no data sources defined yet

ProTip: You can also define data sources through configuration files.

(Jess Cherry, CC BY-SA 4.0)

Select **Prometheus**.



Prometheus Time Series C × | Add data source: Add data ×

192.168.49.2:30549/datasources/new

Add data source
Choose a data source type

Filter by name or type

Cancel

Time series databases

Prometheus
Open source time series database & alerting
Core

Learn more | Select

Graphite
Open source time series database
Core

OpenTSDB
Open source time series database
Core

InfluxDB
Open source time series database
Core

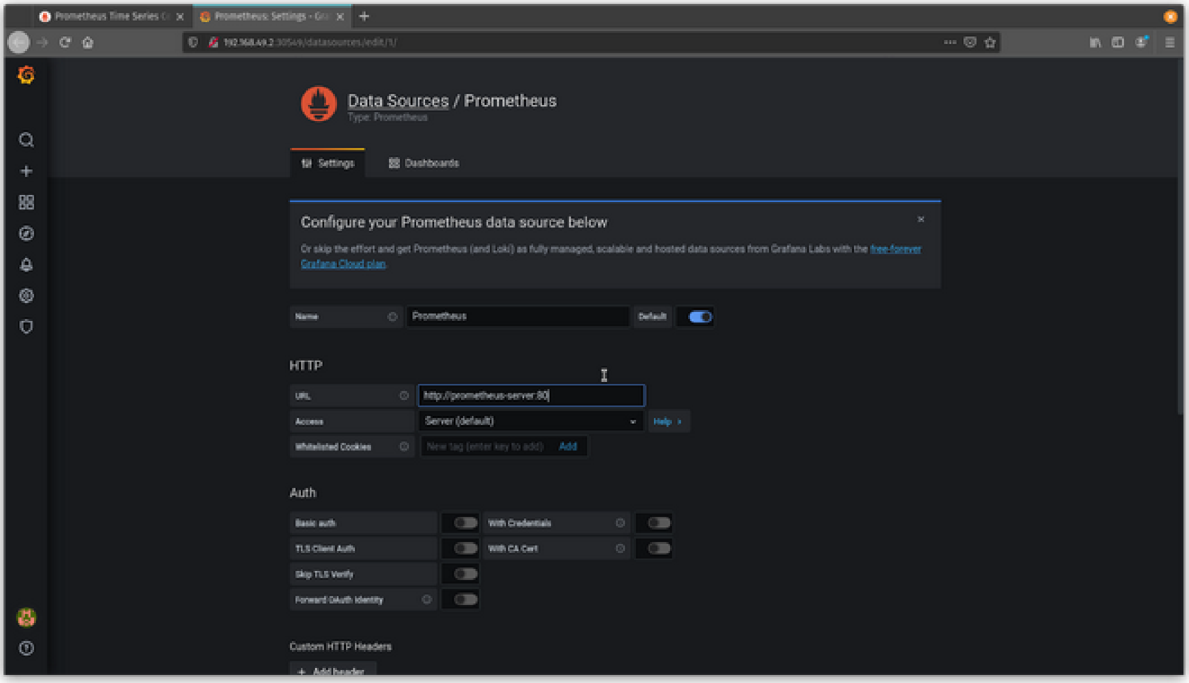
Logging & document databases

Loki
Like Prometheus but for logs. OSS logging solution from Grafana Labs
Core

Elasticsearch

(Jess Cherry, CC BY-SA 4.0)

Because you configured your Prometheus instance to be exposed on port 80, use the service name **prometheus-server** and the server **port 80**.



Prometheus Time Series C × | Prometheus: Settings - Gr ×

192.168.49.2:30549/datasources/edit/1/

Data Sources / Prometheus
Type: Prometheus

Settings | Dashboards

Configure your Prometheus data source below

Or skip the effort and get Prometheus (and Loki) as fully managed, scalable and hosted data sources from Grafana Labs with the free-forever Grafana Cloud plan.

Name: Prometheus
Default: ●

HTTP

URL: http://prometheus-server:80
Access: Server (default)
Help ›
Whitelisted Cookies: New tag (enter key to add) Add

Auth

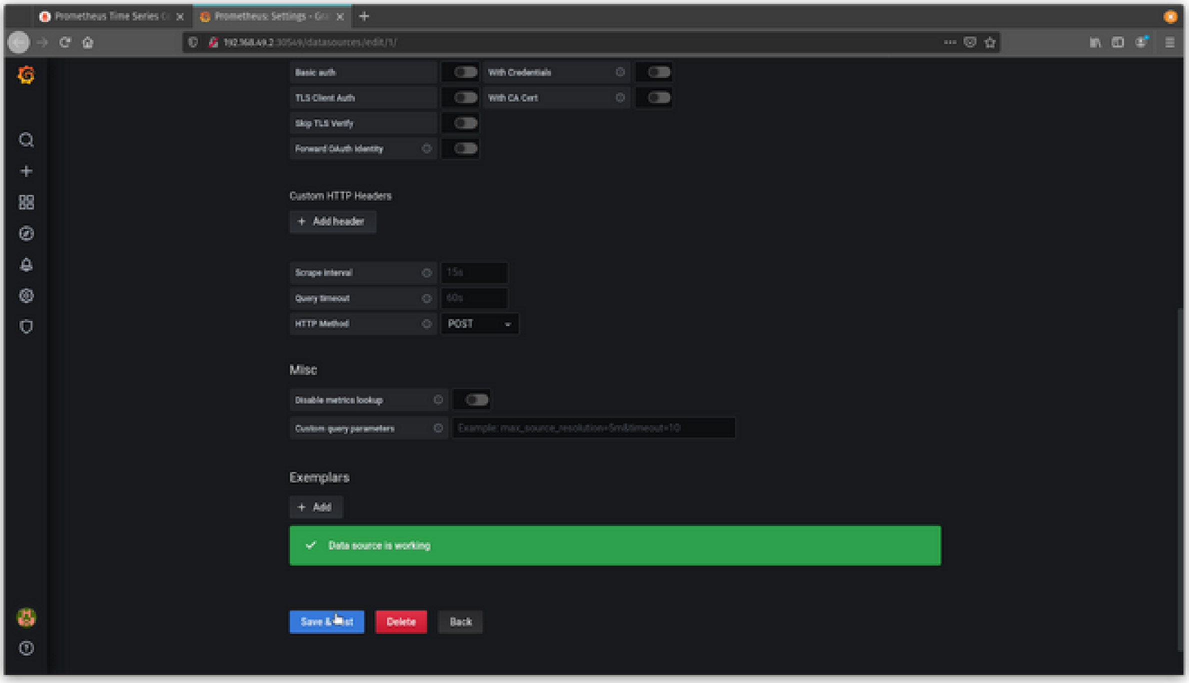
Basic auth ○
With Credentials ○
TLS Client Auth ○
With CA Cert ○
Skip TLS Verify ○
Forward OAuth Identity ○

Custom HTTP Headers

+ Add header

(Jess Cherry, CC BY-SA 4.0)

Save and test your new data source by scrolling to the bottom of the screen and clicking **Save and Test**. You should see a green banner that says **Data source is working**.



Basic auth
With Credentials
TLS Client Auth
With CA Cert
Skip TLS Verify
Forward OAuth Identity

Scrape interval 15s
Query timeout 60s
HTTP Method POST

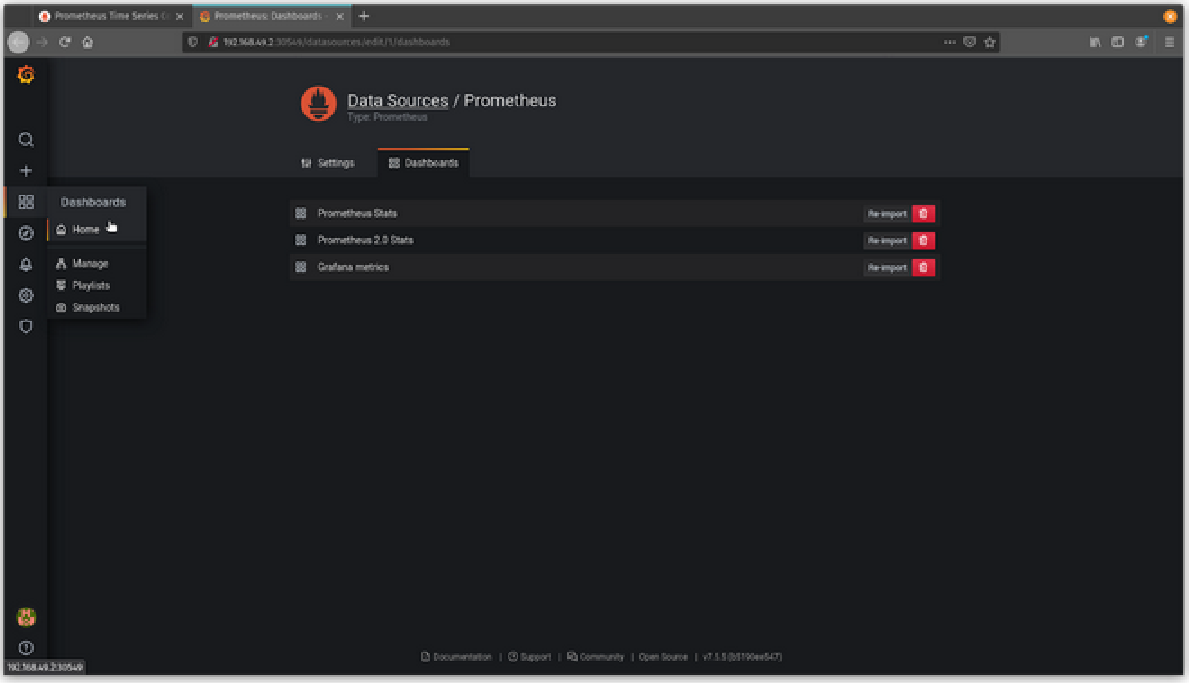
Disable metrics lookup
Custom query parameters Example: max_source_resolution=5m&timeout=10

✓ Data source is working

(Jess Cherry, CC BY-SA 4.0)

Return to the top of the page and click **Dashboards**.

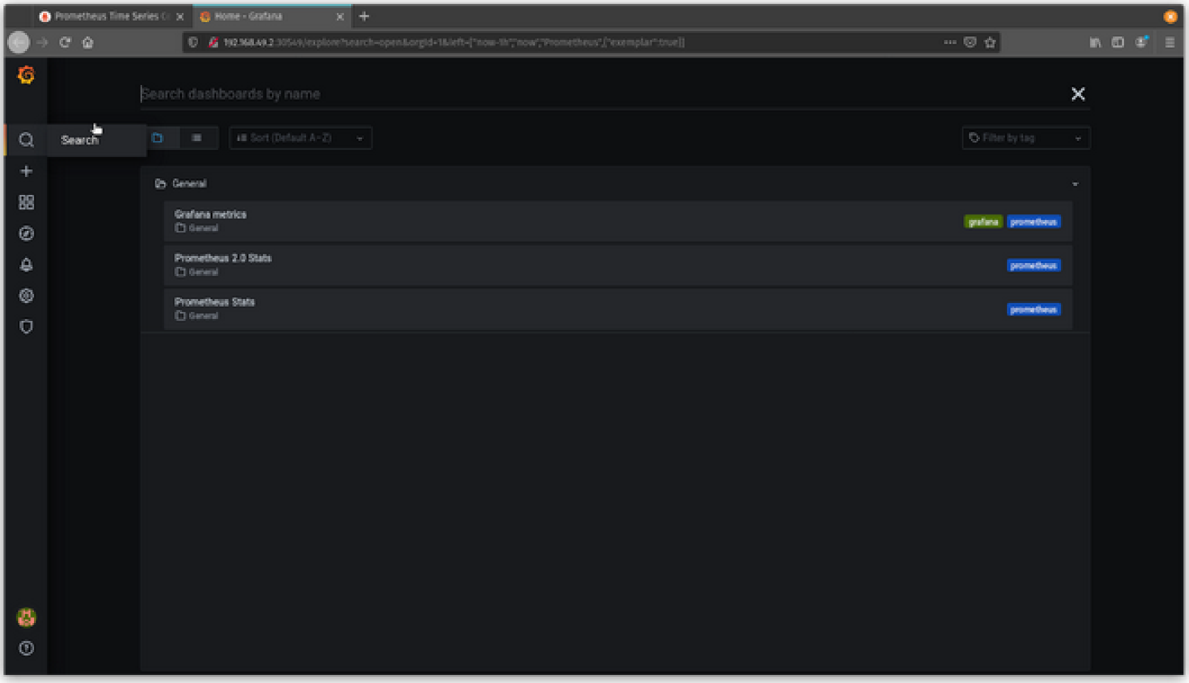
Import all three dashboard options.



Documentation | Support | Community | Open Source | v7.5.5 (b5190ee547)

(Jess Cherry, CC BY-SA 4.0)

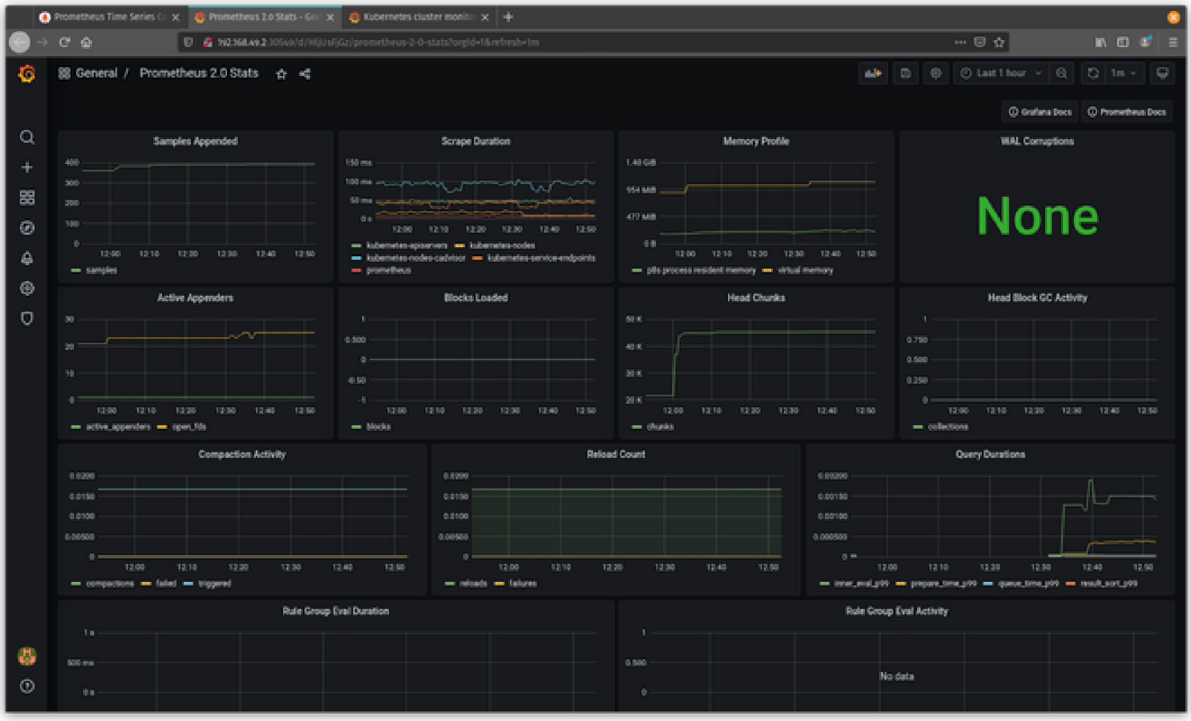
Click the **magnifying glass** icon on the left-hand side to confirm all three dashboards have been imported.



192.168.49.2:30549/explore?search=open&orgId=1&left=["now-1h","now","Prometheus",{"exemplar":true}]

(Jess Cherry, CC BY-SA 4.0)

Now that everything is configured, click **Prometheus 2.0 Stats**, and you should see something similar to this.



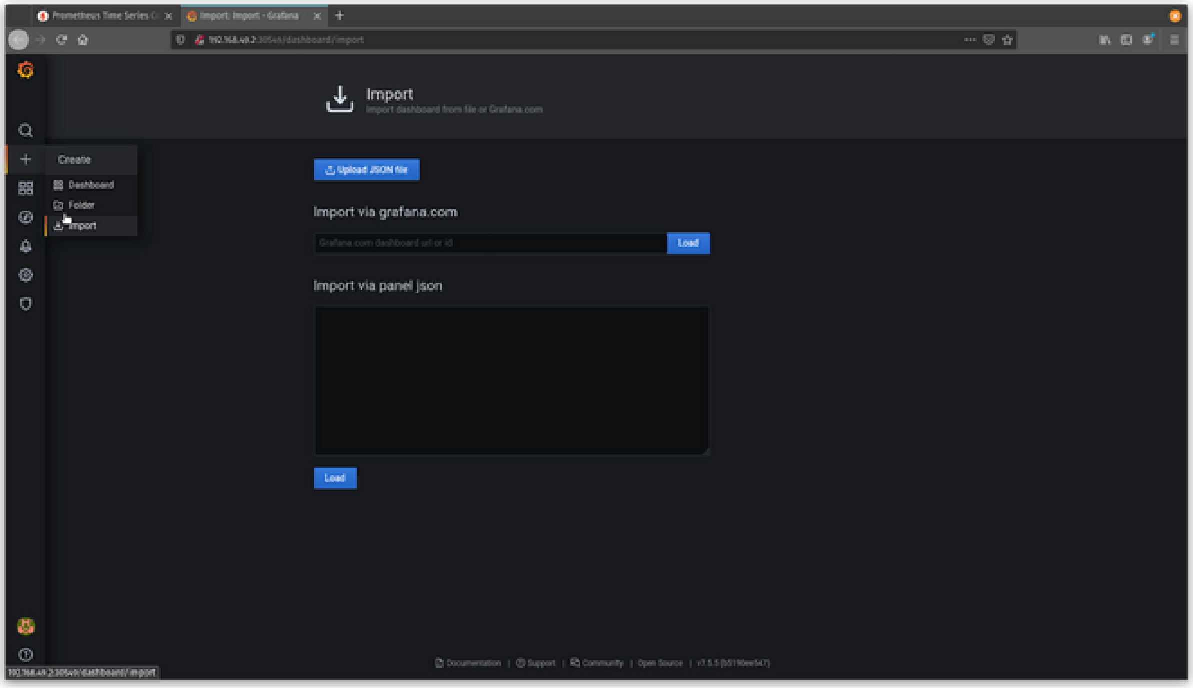
(Jess Cherry, CC BY-SA 4.0)

Congratulations! You have a set up basic data collection from Prometheus about your cluster.

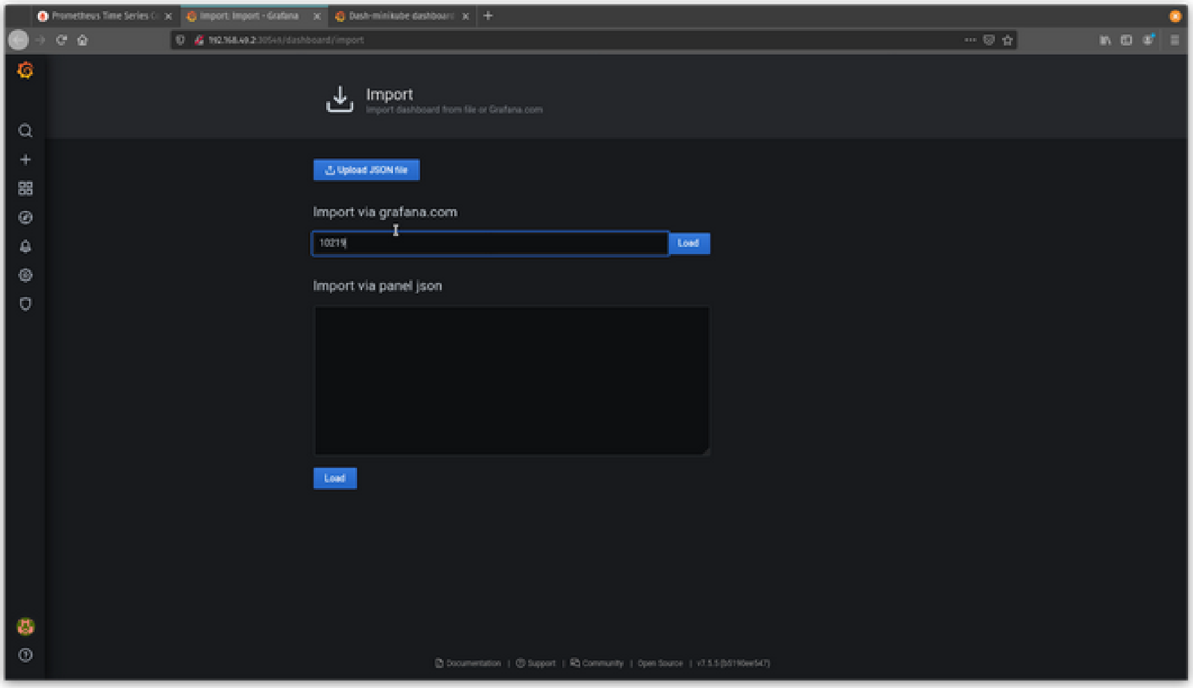
Import more monitoring dashboards

You can import additional detailed dashboards from Grafana Labs' community dashboards collection. I picked two of my favorites, Dash-minikube and Kubernetes Cluster Monitoring, for this quick walkthrough.

To import a dashboard, you need its ID from the dashboards collection. First, click the plus (**+**) sign on the left-hand side to create a dashboard, then click **Import** in the dropdown list, and enter the ID. For Dash-minikube, it's ID 10219.



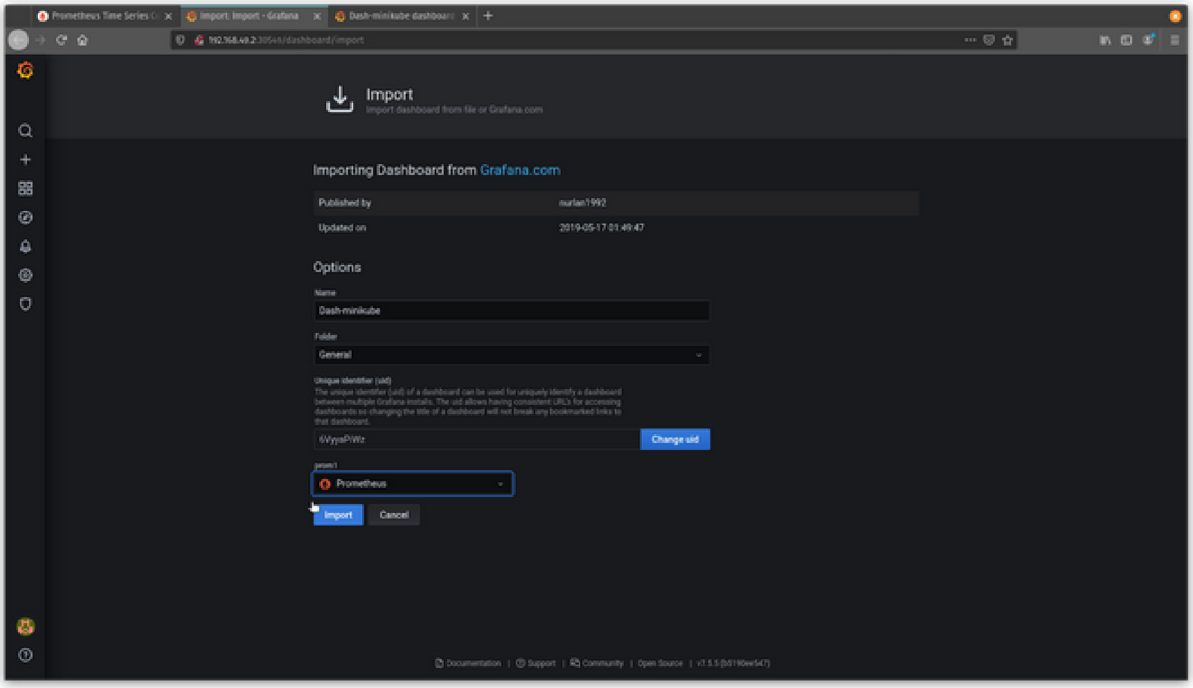
(Jess Cherry, CC BY-SA 4.0)



Import via grafana.com: 10219

(Jess Cherry, CC BY-SA 4.0)

Click **Load**, and enter the data source on the next screen. Since this uses Prometheus, enter your Prometheus data source.



Importing Dashboard from Grafana.com

Published by nurlan1992

Updated on 2019-05-17 01:49:47

Options

Name: Dash-minikube

Folder: General

Unique identifier (uid)
The unique identifier (uid) of a dashboard can be used for uniquely identify a dashboard between multiple Grafana installs. The uid allows having consistent URL's for accessing dashboards so changing the title of a dashboard will not break any bookmarked links to that dashboard.

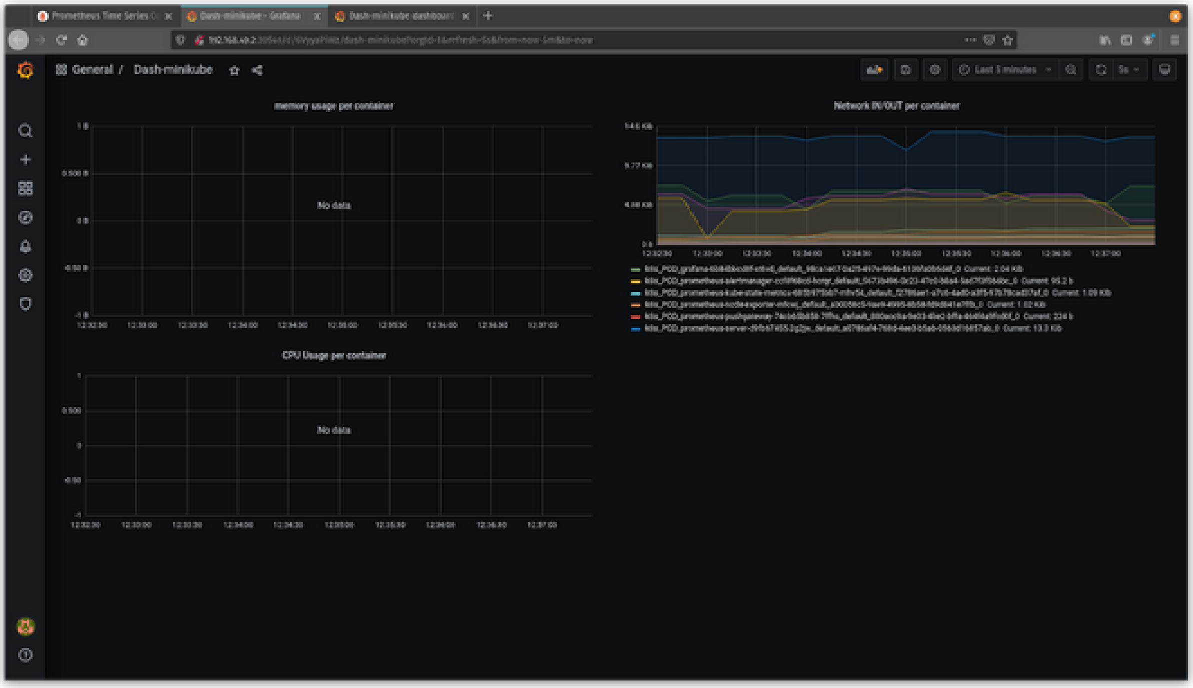
6VyyoPiWz

prom1: Prometheus

Change uid
Import
Cancel
Documentation | Support | Community | Open Source | v7.5.5 (b5190ee547)

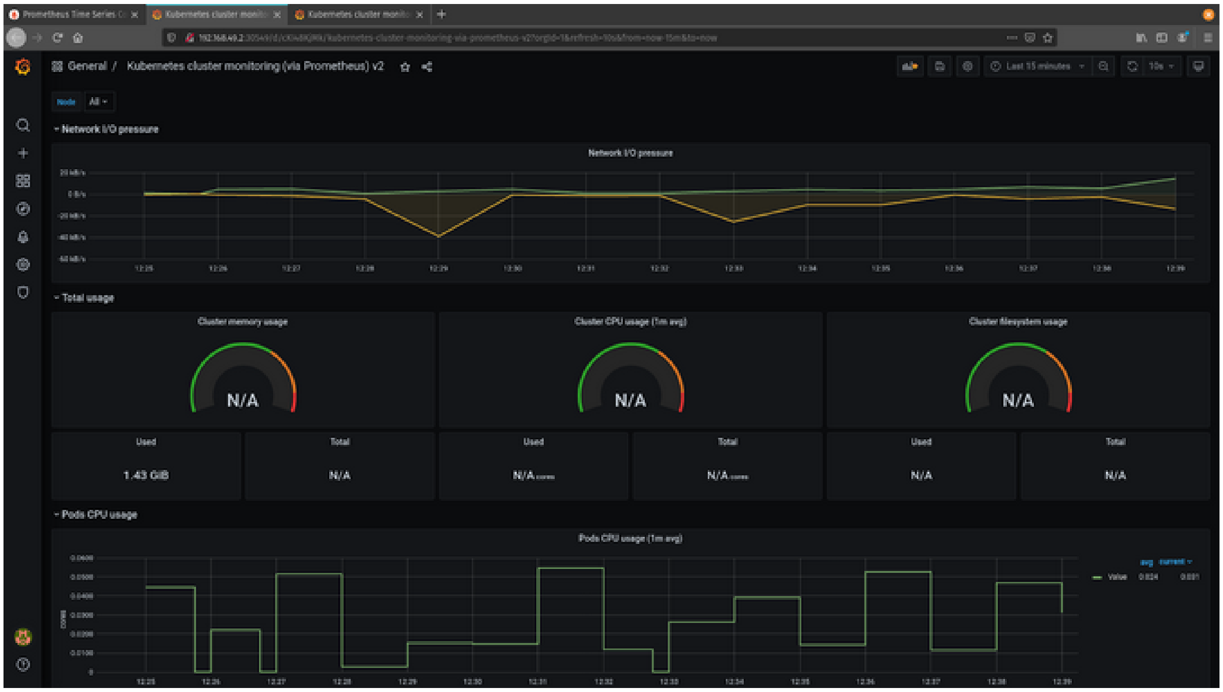
(Jess Cherry, CC BY-SA 4.0)

Click **Import**, and the new dashboard appears:


Prometheus Time Series C
Dash-minikube - Grafana
Dash-minikube dashboard
192.168.49.2:30546/d/6VyyoPiWz/dash-minikube?orgId=1&refresh=5s&from=now-5m&to=now
General / Dash-minikube
Last 5 minutes
5s
memory usage per container
1 B
0.500 B
0 B
-0.50 B
-1 B
No data
12:32:30
12:33:00
12:33:30
12:34:00
12:34:30
12:35:00
12:35:30
12:36:00
12:36:30
12:37:00
Network IN/OUT per container
14.6 KiB
9.77 KiB
4.88 KiB
0 B
k8s_POD_grafana-6b84bbcd8f-xhtw6_default_98ca1e07-0a25-417e-99da-61369a0b6d4f_0 Current: 2.04 KiB
k8s_POD_prometheus-alertmanager-ccf8f68cd-hcnqr_default_5673b496-0c23-47c0-b8a4-5ad7f3f5b6bc_0 Current: 95.2 b
k8s_POD_prometheus-kube-state-metrics-6858975bb7-mhv54_default_f2786ae1-a7c6-4ad0-a3f5-97b79cad37af_0 Current: 1.09 KiB
k8s_POD_prometheus-node-exporter-mfcwj_default_a30058c5-9ae9-4995-8b58-fd9d841e7f9b_0 Current: 1.02 KiB
k8s_POD_prometheus-pushgateway-74cb65b858-7fftw_default_880acc9a-9e03-4be2-bf5a-464f4a9fcd0f_0 Current: 224 b
k8s_POD_prometheus-server-d9fb67d55-2g2px_default_a0786af4-7686-4ee3-b5ab-0563d16857ab_0 Current: 13.3 KiB
CPU Usage per container
1
0.500
0
-0.50
-1
No data

(Jess Cherry, CC BY-SA 4.0)

Now you have a new dashboard to keep track of your Minikube stats. If you follow the same steps using Kubernetes Cluster Monitoring (ID 2115), you will see a more verbose monitoring dashboard.



(Jess Cherry, CC BY-SA 4.0)

Now you can keep track of your steady state with Grafana and Prometheus data collections and visuals.

Final thoughts

With these open source tools, you can collect your cluster's steady state and maintain a good pulse on it. This is important in chaos engineering because it allows you to check everything in a destructive, unstable state and use that data to test your hypothesis about what could happen to its state during an outage.

Test Kubernetes cluster failures and experiments in your terminal

Do you know how your system will respond to an arbitrary failure? Will your application fail? Will anything survive after a loss? If you're not sure, it's time to see if your system passes the Litmus test, a detailed way to cause chaos at random with many experiments.

In the first article in this series, I explained what chaos engineering is, and in the second article, I demonstrated how to get your system's steady state so that you can compare it against a chaos state. This third article shows you how to install and use Litmus to test arbitrary failures and experiments in your Kubernetes cluster. In this walkthrough, I use Pop!_OS 20.04, Helm 3, Minikube 1.14.2, and Kubernetes 1.19.

Configure Minikube

Install Minikube in whatever way makes sense for your environment. If you have enough resources, I recommend giving your virtual machine a bit more than the default memory and CPU power:

```
$ minikube config set memory 8192
❕  These changes take effect upon a minikube delete and then a minikube start
$ minikube config set cpus 6
❕  These changes take effect upon a minikube delete and then a minikube start
```

Then start and check your system's status:

```
$ minikube start
minikube v1.14.2 on Debian bullseye/sid
minikube 1.19.0 is available! http://github.com/kubernetes/minikube
To disable this notice: 'minikube config set WantUpdateNotification false'
Using the docker driver based on user configuration
Starting control plane node minikube in cluster minikube
Verifying Kubernetes components...
```

```
Done! kubectl is now configured to use "minikube" by default
$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Install Litmus

As outlined on Litmus' homepage, the steps to install Litmus are: add your repo to Helm, create your Litmus namespace, then install your chart:

```
$ helm repo add litmuschaos https://litmuschaos.github.io/litmus-helm/
"litmuschaos" has been added to your repositories
$ kubectl create ns litmus
namespace/litmus created
$ helm install chaos litmuschaos/litmus --namespace=litmus
NAME: chaos
LAST DEPLOYED: Sun May  9 17:05:36 2021
NAMESPACE: litmus
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
```

Verify the installation

You can run the following commands if you want to verify all the desired components are installed correctly.

Verify that **api-resources** for chaos are available:

```
# kubectl api-resources | grep litmus
chaosengines     litmuschaos.io   true   ChaosEngine
chaosexperiments litmuschaos.io   true   ChaosExperiment
chaosresults     litmuschaos.io   true   ChaosResult
```

Verify that the Litmus chaos operator deployment is running successfully:

```
# kubectl get pods -n litmus
NAME                      READY   STATUS    RESTARTS
litmus-7d998b6568-nnlcd   1/1     Running          0
```

Run chaos experiments

With this out of the way, you are good to go! Refer to Litmus' chaos experiment documentation to start executing your first experiment.

To confirm your installation is working, check that the pod is up and running correctly:

```
$ kubectl get pods -n litmus
NAME                      READY   STATUS    RESTARTS
litmus-7d6f994d88-2g7wn   1/1     Running          0
```

Confirm that the Custom Resource Definitions (CRDs) are also installed correctly:

```
$ kubectl get crds | grep chaos
chaosengines.litmuschaos.io      2021-05-09T21:05:33Z
chaosexperiments.litmuschaos.io  2021-05-09T21:05:33Z
chaosresults.litmuschaos.io      2021-05-09T21:05:33Z
```

Finally, confirm your API resources are also installed:

```
$ kubectl api-resources | grep chaos
chaosengines      litmuschaos.io   true   ChaosEngine
chaosexperiments  litmuschaos.io   true   ChaosExperiment
chaosresults      litmuschaos.io   true   ChaosResult
```

That's what I call easy installation and confirmation. The next step is setting up deployments for chaos.

Prep for destruction

To test for chaos, you need something to test against. Add a new namespace:

```
$ kubectl create namespace more-apps
namespace/more-apps created
```

Then add a deployment to the new namespace:

```
$ kubectl create deployment ghost --namespace more-apps \
--image=ghost:3.11.0-alpine

deployment.apps/ghost created
```

Finally, scale your deployment up so that you have more than one pod in your deployment to test against:

```
$ kubectl scale deployment/ghost \
--namespace more-apps --replicas=4

deployment.apps/ghost scaled
```

For Litmus to cause chaos, you need to add an annotation to your deployment to mark it ready for chaos. Currently, annotations are available for deployments, StatefulSets, and DaemonSets. Add the annotation `chaos=true` to your deployment:

```
$ kubectl annotate deploy/ghost \
litmuschaos.io/chaos="true" -n more-apps

deployment.apps/ghost annotated
```

Make sure the experiments you will install have the correct permissions to work in the "more-apps" namespace.

Make a new **rbac.yaml** file for the prepper bindings and permissions:

```
$ touch rbac.yaml
```

Then add permissions for the generic testing by copying and pasting the code below into your **rbac.yaml** file. These are just basic, minimal permissions to kill pods in your namespace and give Litmus permissions to delete a pod for a namespace you provide:

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pod-delete-sa
  namespace: more-apps
  labels:
    name: pod-delete-sa
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

```
metadata:
  name: pod-delete-sa
  namespace: more-apps
  labels:
    name: pod-delete-sa
rules:
- apiGroups: [""]
  resources: ["pods","events"]
  verbs: ["create","list","get","patch","update","delete","deletecollection"]
- apiGroups: [""]
  resources: ["pods/exec","pods/log","replicationcontrollers"]
  verbs: ["create","list","get"]
- apiGroups: ["batch"]
  resources: ["jobs"]
  verbs: ["create","list","get","delete","deletecollection"]
- apiGroups: ["apps"]
  resources: ["deployments","statefulsets","daemonsets","replicasets"]
  verbs: ["list","get"]
- apiGroups: ["apps.openshift.io"]
  resources: ["deploymentconfigs"]
  verbs: ["list","get"]
- apiGroups: ["argoproj.io"]
  resources: ["rollouts"]
  verbs: ["list","get"]
- apiGroups: ["litmuschaos.io"]
  resources: ["chaosengines","chaosexperiments","chaosresults"]
  verbs: ["create","list","get","patch","update"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-delete-sa
  namespace: more-apps
  labels:
    name: pod-delete-sa
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-delete-sa
subjects:
- kind: ServiceAccount
  name: pod-delete-sa
  namespace: more-apps
```

Apply the **rbac.yaml** file:

```
$ kubectl apply -f rbac.yaml
serviceaccount/pod-delete-sa created
```

```
role.rbac.authorization.k8s.io/pod-delete-sa created
rolebinding.rbac.authorization.k8s.io/pod-delete-sa created
```

The next step is to prepare your chaos engine to delete pods. The chaos engine will connect the experiment you need to your application instance by creating a **chaosengine.yaml** file and copying the information below into the .yaml file. This will connect your experiment to your namespace and the service account with the role bindings you created above.

This chaos engine file only specifies the pod to delete during chaos testing:

```
apiVersion: litmuschaos.io/v1alpha1
kind: ChaosEngine
metadata:
  name: moreapps-chaos
  namespace: more-apps
spec:
  appinfo:
    appns: 'more-apps'
    applabel: 'app=ghost'
    appkind: 'deployment'
  # It can be true/false
  annotationCheck: 'true'
  # It can be active/stop
  engineState: 'active'
  #ex. values: ns1:name=percona,ns2:run=more-apps
  auxiliaryAppInfo: ''
  chaosServiceAccount: pod-delete-sa
  # It can be delete/retain
  jobCleanUpPolicy: 'delete'
  experiments:
    - name: pod-delete
      spec:
        components:
          env:
            # set chaos duration (in sec) as desired
            - name: TOTAL_CHAOS_DURATION
              value: '30'
            # set chaos interval (in sec) as desired
            - name: CHAOS_INTERVAL
              value: '10'
            # pod failures without '--force' & default
terminationGracePeriodSeconds
            - name: FORCE
              value: 'false'
```

Don't apply this file until you install the experiments in the next section.

Add new experiments for causing chaos

Now that you have an entirely new environment with deployments, roles, and the chaos engine to test against, you need some experiments to run. Since Litmus has a large community, you can find some great experiments in the Chaos Hub.

In this walkthrough, I'll use the generic experiment of killing a pod.

Run a kubectl command to install the generic experiments into your cluster. Install this in your `more-apps` namespace; you will see the tests created when you run it:

```
$ kubectl apply -n more-apps -f \
https://hub.litmuschaos.io/api/chaos/1.13.3?file=charts/generic/experiments.yaml
chaosexperiment.litmuschaos.io/pod-network-duplication created
chaosexperiment.litmuschaos.io/node-cpu-hog created
chaosexperiment.litmuschaos.io/node-drain created
chaosexperiment.litmuschaos.io/docker-service-kill created
chaosexperiment.litmuschaos.io/k8-pod-delete created
chaosexperiment.litmuschaos.io/pod-delete created
chaosexperiment.litmuschaos.io/node-poweroff created
chaosexperiment.litmuschaos.io/k8-service-kill created
[...]
chaosexperiment.litmuschaos.io/pod-memory-hog created
```

Verify the experiments installed correctly:

```
$ kubectl get chaosexperiments -n more-apps
NAME                      AGE
container-kill            72s
disk-fill                 72s
disk-loss                 72s
docker-service-kill       72s
k8-pod-delete             72s
k8-service-kill           72s
kubelet-service-kill      72s
node-cpu-hog              72s
node-drain                72s
node-io-stress            72s
node-memory-hog           72s
[...]
pod-network-loss          72s
```

Run the experiments

Now that everything is installed and configured, use your **chaosengine.yaml** file to run the pod-deletion experiment you defined. Apply your chaos engine file:

```
$ kubectl apply -f chaosengine.yaml
chaosengine.litmuschaos.io/more-apps-chaos created
```

Confirm the engine started by getting all the pods in your namespace; you should see `pod-delete` being created:

```
$ kubectl get pods -n more-apps
NAME                    READY  STATUS             RESTARTS
ghost-5bdd4cdcc4-blmtl  1/1    Running               0
ghost-5bdd4cdcc4-z2lnt  1/1    Running               0
ghost-5bdd4cdcc4-zlcc9  1/1    Running               0
ghost-5bdd4cdcc4-zrs8f  1/1    Running               0
moreapps-chaos-runner   1/1    Running               0
pod-delete-e443qx-lxzfx 0/1    ContainerCreating     0
```

Next, you need to be able to observe your experiments using Litmus. The following command uses the ChaosResult CRD and provides a large amount of output:

```
$ kubectl describe chaosresult moreapps-chaos-pod-delete -n more-apps
Name: moreapps-chaos-pod-delete
Namespace: more-apps
Labels: app.kubernetes.io/component=experiment-job
        app.kubernetes.io/part-of=litmus
        app.kubernetes.io/version=1.13.3
        chaosUID=a6c9ab7e-ff07-4703-abe4-43e03b77bd72
        controller-uid=601b7330-c6f3-4d9b-90cb-2c761ac0567a
        job-name=pod-delete-e443qx
        name=moreapps-chaos-pod-delete
[...]
Spec:
  Engine:      moreapps-chaos
  Experiment:  pod-delete
Status:
  Experiment Status:
    Fail Step:                 N/A
    Phase:                     Completed
    Probe Success Percentage:  100
    Verdict:                   Pass
  History:
    Failed Runs:   0
    Passed Runs:   1
    Stopped Runs:  0
Events:
  Type    Reason     From               Message
  ----    ------     ----               -------
  Normal  Pass       pod-delete-e43qx  experiment: pod-delete, Result: Pass
```

You can see the pass or fail output from your testing as you run the chaos engine definitions.

Congratulations on your first (and hopefully not last) chaos engineering test! Now you have a powerful tool to use and help your environment grow.

Final thoughts

You might be thinking, "I can't run this manually every time I want to run chaos. How far can I take this, and how can I set it up for the long term?"

Litmus' best part (aside from the Chaos Hub) is its scheduler function. You can use it to define times and dates, repetitions or sporadic, to run experiments. This is a great tool for detailed admins who have been working with Kubernetes for a while and are ready to create some chaos. I suggest staying up to date on Litmus and how to use this tool for regular chaos engineering. Happy pod hunting!

Test your Kubernetes experiments with an open source web interface

Have you wanted to cause chaos to test your systems but prefer to use visual tools rather than the terminal? Well, this article is for you, my friend. So far, I've explained what chaos engineering is; in the second article, I demonstrated how to get your system's steady state so that you can compare it against a chaos state; and in the third, I showed how to use Litmus to test arbitrary failures and experiments in your Kubernetes cluster.

In this article, I introduces you to Chaos Mesh, an open source chaos orchestrator with a web user interface (UI) that anyone can use. It allows you to create experiments and display statistics in a web UI for presentations or visual storytelling. The Cloud Native Computing Foundation hosts the Chaos Mesh project, which means it is a good choice for Kubernetes. So let's get started! In this walkthrough, I use Pop!_OS 20.04, Helm 3, Minikube 1.14.2, and Kubernetes 1.19.

Configure Minikube

If you haven't already done so, install Minikube in whatever way makes sense for your environment. If you have enough resources, I recommend giving your virtual machine a bit more than the default memory and CPU power:

```
$ minikube config set memory 8192
❗  These changes take effect upon a minikube delete and then a minikube start
$ minikube config set cpus 6
❗  These changes take effect upon a minikube delete and then a minikube start
```

Then start and check your system's status:

```
$ minikube start
minikube v1.14.2 on Debian bullseye/sid
minikube 1.19.0 is available! http://github.com/kubernetes/minikube
```

```
To disable this notice: 'minikube config set WantUpdateNotification false'
Using the docker driver based on user configuration
Starting control plane node minikube in cluster minikube
Verifying Kubernetes components...
Done! kubectl is now configured to use "minikube" by default
$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Install Chaos Mesh

Start installing Chaos Mesh by adding the repository to Helm:

```
$ helm repo add chaos-mesh https://charts.chaos-mesh.org

"chaos-mesh" has been added to your repositories
```

Then search for your Helm chart:

```
$ helm search repo chaos-mesh

NAME                   CHART   APP     DESCRIPTION
chaos-mesh/chaos-mesh  v0.5.0  v1.2.0  Chaos Mesh® is a cloud-native [...]
```

Once you find your chart, you can begin the installation steps, starting with creating a `chaos-testing` namespace:

```
$ kubectl create ns chaos-testing
namespace/chaos-testing created
```

Next, install your Chaos Mesh chart in this namespace and name it `chaos-mesh`:

```
$ helm install chaos-mesh chaos-mesh/chaos-mesh \
--namespace=chaos-testing
NAME: chaos-mesh
LAST DEPLOYED: Mon May 10 10:08:52 2021
NAMESPACE: chaos-testing
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

```
NOTES: Make sure chaos-mesh components are running
$ kubectl get pods --namespace chaos-testing \
-l app.kubernetes.io/instance=chaos-mesh
```

As the output instructs, check that the Chaos Mesh components are running:

```
$ kubectl get pods --namespace chaos-testing \
-l app.kubernetes.io/instance=chaos-mesh

NAME                               READY   STATUS    RESTARTS
chaos-controller-manager-bfdcb99   1/1     Running   0
chaos-daemon-4mjq2                 1/1     Running   0
chaos-dashboard-865b778d79-729xw   1/1     Running   0
```

Now that everything is running correctly, you can set up the services to see the Chaos Mesh dashboard and make sure the `chaos-dashboard` service is available:

```
$ kubectl get svc -n chaos-testing

NAME                    TYPE   CLUSTER-IP   EXTERNAL-IP   PORT(S)
chaos-daemon           ClusterIP  None          <none>   31767/TCP,31766/TCP
chaos-dashboard        NodePort   10.99.137.187 <none>   2333:30029/TCP
chaos-mesh-contr[...] ClusterIP  10.99.118.132 <none>  0081/TCP,10080/TCP,443/TCP
```

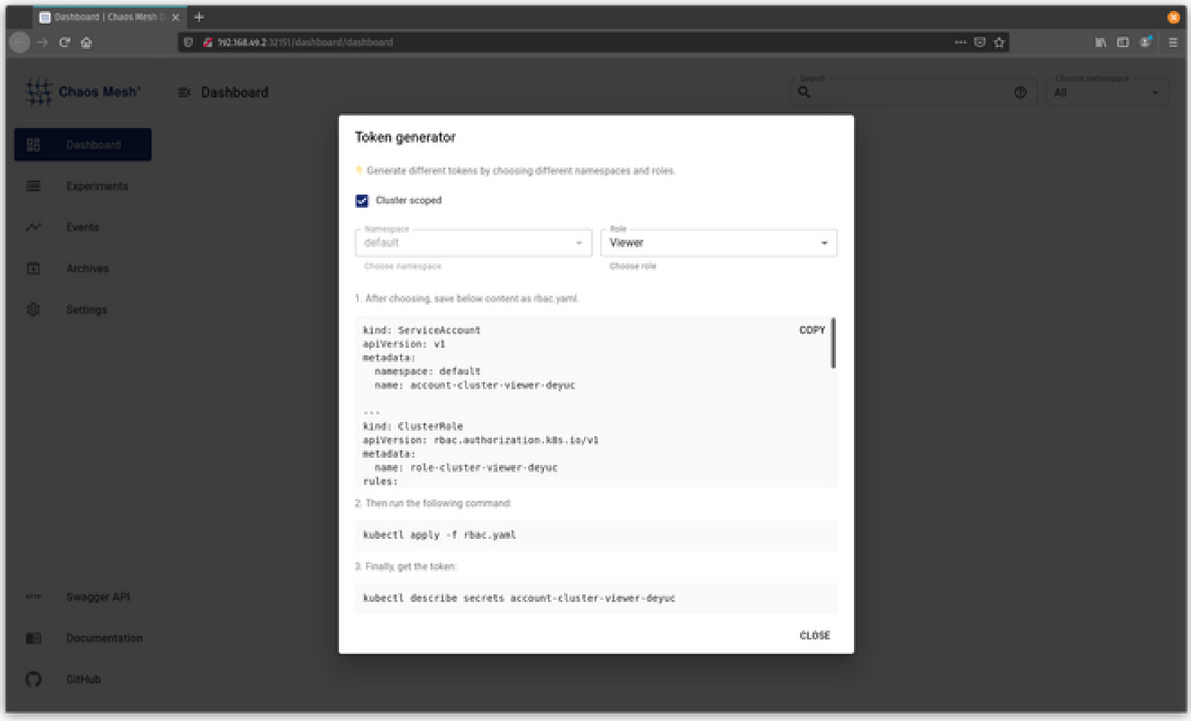
Now that you know the service is running, go ahead and expose it, rename it, and open the dashboard using `minikube service`:

```
$ kubectl expose service chaos-dashboard \
--namespace chaos-testing --type=NodePort \
--target-port=2333 --name=chaos

service/chaos exposed

$ minikube service chaos --namespace chaos-testing
NAMESPACE     | NAME  | TARGET PORT | URL
chaos-testing | chaos | 2333        | http://192.168.49.2:32151
Opening service chaos-testing/chaos in default browser...
```

When the browser opens, you'll see a token generator window. Check the box next to **Cluster scoped**, and follow the directions on the screen.



Token generator

Generate different tokens by choosing different namespaces and roles.

☑ Cluster scoped

Namespace: default (Choose namespace)

Role: Viewer (Choose role)

1. After choosing, save below content as rbac.yaml.

```
kind: ServiceAccount
apiVersion: v1
metadata:
  namespace: default
  name: account-cluster-viewer-deyuc

---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: role-cluster-viewer-deyuc
rules:
```

COPY

2. Then run the following command:

```
kubectl apply -f rbac.yaml
```

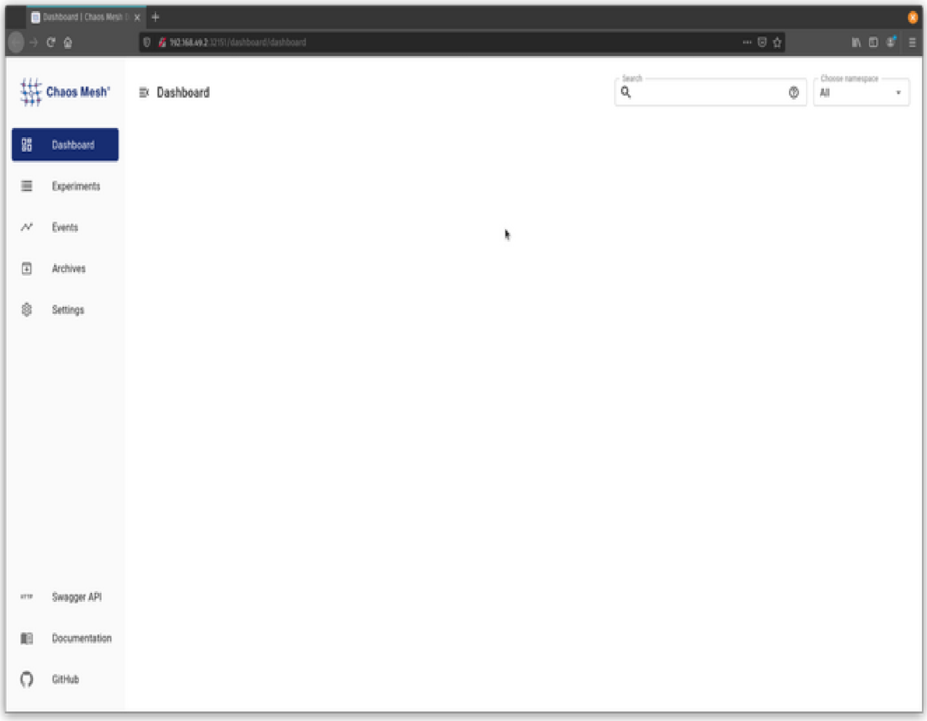
3. Finally, get the token:

```
kubectl describe secrets account-cluster-viewer-deyuc
```

CLOSE

(Jess Cherry, CC BY-SA 4.0)

Then you can log into Chaos Mesh and see the Dashboard.



(Jess Cherry, CC BY-SA 4.0)

You have installed your Chaos Mesh instance and can start working towards chaos testing!

Get meshy in your cluster

Now that everything is up and running, you can set up some new experiments to try. The documentation offers some predefined experiments, and I'll choose StressChaos from the options. In this walkthrough, you will create something in a new namespace to stress against and scale it up so that it can stress against more than one thing.

Create the namespace:

```
$ kubectl create ns app-demo

namespace/app-demo created
```

Then create the deployment in your new namespace:

```
$ kubectl create deployment nginx \
--image=nginx --namespace app-demo

deployment.apps/nginx created
```

Scale the deployment up to eight pods:

```
$ kubectl scale deployment/nginx \
--replicas=8 --namespace app-demo

deployment.apps/nginx scaled
```

Finally, confirm everything is up and working correctly by checking your pods in the namespace:

```
$ kubectl get pods -n app-demo
NAME                     READY   STATUS    RESTARTS
nginx-6799fc88d8-e8d2e   1/1     Running   0
nginx-6799fc88d8-82p8t   1/1     Running   0
nginx-6799fc88d8-dfrlz   1/1     Running   0
nginx-6799fc88d8-kbf75   1/1     Running   0
nginx-6799fc88d8-m25hs   1/1     Running   0
nginx-6799fc88d8-mg4tb   1/1     Running   0
nginx-6799fc88d8-q9m2m   1/1     Running   0
nginx-6799fc88d8-v7q4d   1/1     Running   0
```

Now that you have something to test against, you can begin working on the definition for your experiment. Start by creating `chaos-test.yaml`:

```
$ touch chaos-test.yaml
```

Next, create the definition for the chaos test. Just copy and paste this experiment definition into your `chaos-test.yaml` file:

```
apiVersion: chaos-mesh.org/v1alpha1
kind: StressChaos
metadata:
  name: burn-cpu
  namespace: chaos-testing
spec:
  mode: one
  selector:
    namespaces:
      - app-demo
    labelSelectors:
      app: "nginx"
  stressors:
    cpu:
      workers: 1
  duration: '30s'
  scheduler:
    cron: '@every 2m'
```

This test will burn 1 CPU for 30 seconds every 2 minutes on pods in the `app-demo` namespace. Finally, apply the YAML file to start the experiment and view what happens in your dashboard.

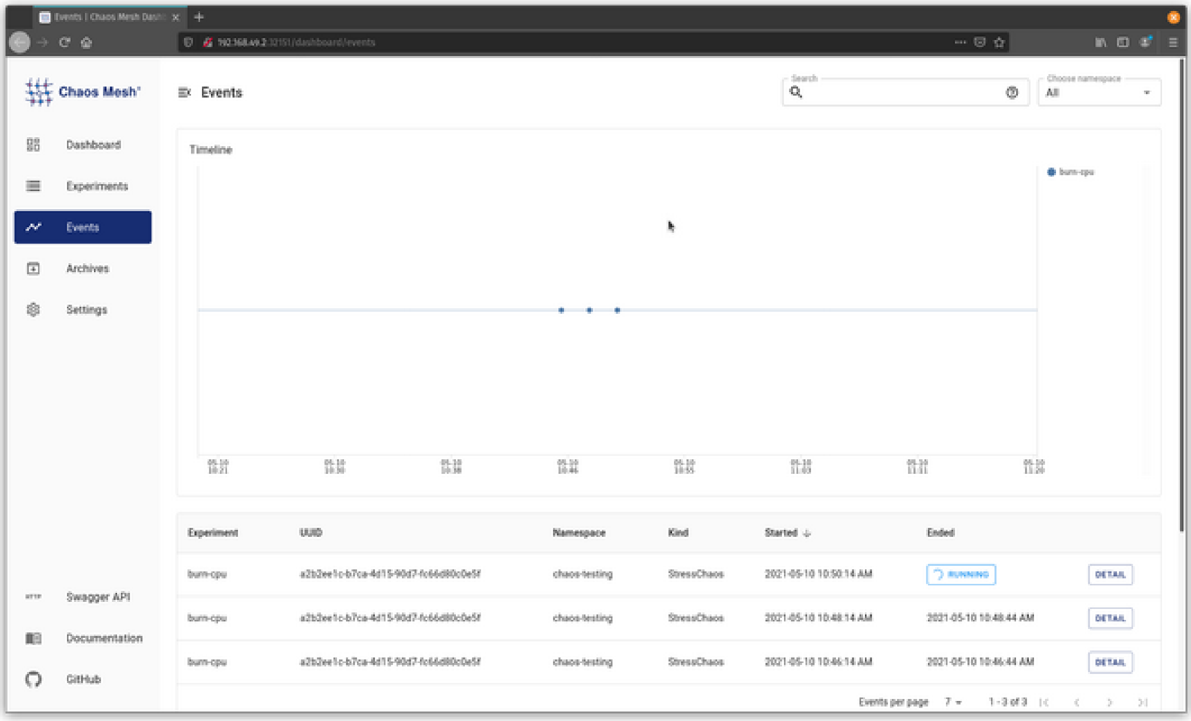
Apply the experiment file:

```
$ kubectl apply -f chaos-test.yaml

stresschaos.chaos-mesh.org/burn-cpu created
```

Then go to your dashboard and click **Experiments** to see the stress test running. You can pause the experiment by pressing the **Pause** button on the right-hand side of the experiment.

Click **Dashboard** to see the state with a count of total experiments, the state graph, and a timeline of running events or previously run tests. Choose **Events** to see the timeline and the experiments below it with details.



Experiment	UUID	Namespace	Kind	Started ↓	Ended	
burn-cpu	a2b2ee1c-b7ca-4d15-90d7-fc66d80c0e5f	chaos-testing	StressChaos	2021-05-10 10:50:14 AM	RUNNING	DETAIL
burn-cpu	a2b2ee1c-b7ca-4d15-90d7-fc66d80c0e5f	chaos-testing	StressChaos	2021-05-10 10:48:14 AM	2021-05-10 10:48:44 AM	DETAIL
burn-cpu	a2b2ee1c-b7ca-4d15-90d7-fc66d80c0e5f	chaos-testing	StressChaos	2021-05-10 10:46:14 AM	2021-05-10 10:46:44 AM	DETAIL

Events per page 7 ▾ 1 - 3 of 3

(Jess Cherry, CC BY-SA 4.0)

Congratulations on completing your first test! Now that you have this working, I'll share more details about what else you can do with your experiments.

But wait, there's more

Other things you can do with this experiment using the command line include:

- Updating the experiment to change how it works
- Pausing the experiment if you need to return the cluster to a steady state
- Resuming the experiment to continue testing
- Deleting the experiment if you no longer need it for testing

Updating the experiment

As an example, update the experiment in your cluster to increase the duration between tests. Go back to your `cluster-test.yaml` and edit the scheduler to change 2 minutes to 20 minutes.

Before:

```
  scheduler:
    cron: '@every 2m'
```

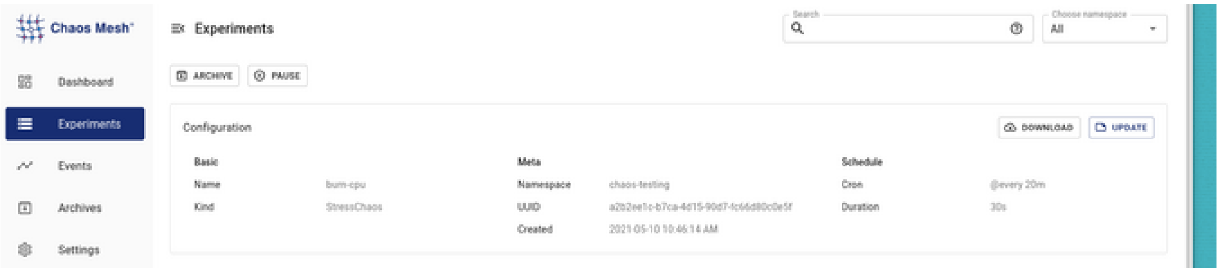
After:

```
  scheduler:
    cron: '@every 20m'
```

Save and reapply your file; the output should show the new stress test configuration:

```
$ kubectl apply -f chaos-test.yaml
stresschaos.chaos-mesh.org/burn-cpu configured
```

If you look in the Dashboard, the experiment should show the new cron configuration.



Configuration

Basic
Name burn-cpu
Kind StressChaos

Meta
Namespace chaos-testing
UUID a2b2ee1c-b7ca-4d15-90d7-fc66d80c0e5f
Created 2021-05-10 10:46:14 AM

Schedule
Cron @every 20m
Duration 30s

(Jess Cherry, CC BY-SA 4.0)

Pausing and resuming the experiment

Manually pausing the experiment on the command line will require adding an annotation to the experiment. Resuming the experiment will require removing the annotation.

To add the annotation, you will need the kind, name, and namespace of the experiment from your YAML file.

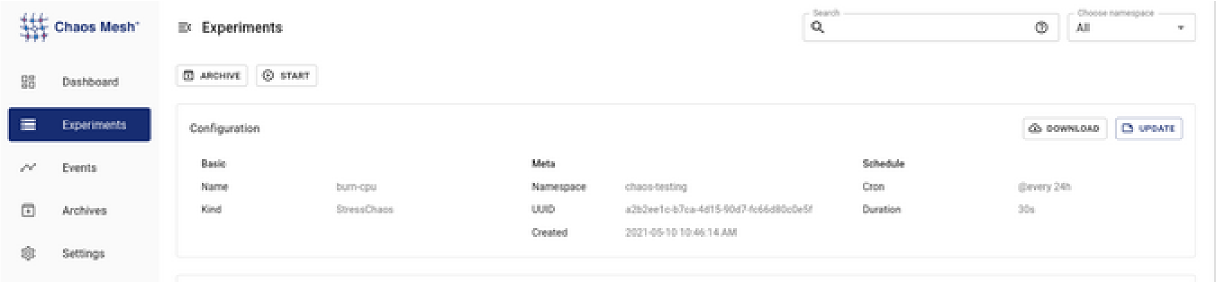
Pause an experiment

You can pause an experiment:

```
$ kubectl annotate stresschaos burn-cpu experiment.chaos-mesh.org/pause=true \
-n chaos-testing


stresschaos.chaos-mesh.org/burn-cpu annotated
```

The web UI shows it is paused.



Configuration

Basic
Name burn-cpu
Kind StressChaos

Meta
Namespace chaos-testing
UUID a2b2ee1c-b7ca-4d15-90d7-fc66d80c0e5f
Created 2021-05-10 10:46:14 AM

Schedule
Cron @every 24h
Duration 30s

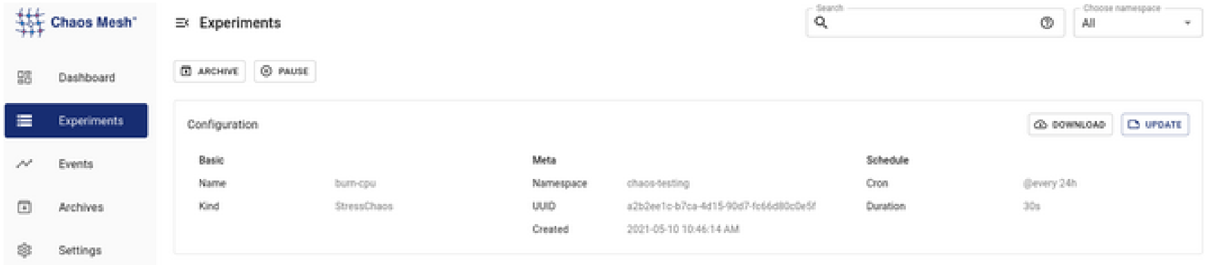
(Jess Cherry, CC BY-SA 4.0)

Resume an experiment

You need the same information to resume your experiment. However, rather than the word `true`, you use a dash to remove the pause.

```
$ kubectl annotate stresschaos burn-cpu experiment.chaos-mesh.org/pause- \
-n chaos-testing
stresschaos.chaos-mesh.org/burn-cpu annotated
```

Now you can see the experiment has resumed in the web UI.



Configuration

Basic
Name burn-cpu
Kind StressChaos

Meta
Namespace chaos-testing
UUID a2b2ee1c-b7ca-4d15-90d7-fc66d80c0e5f
Created 2021-05-10 10:46:14 AM

Schedule
Cron @every 24h
Duration 30s

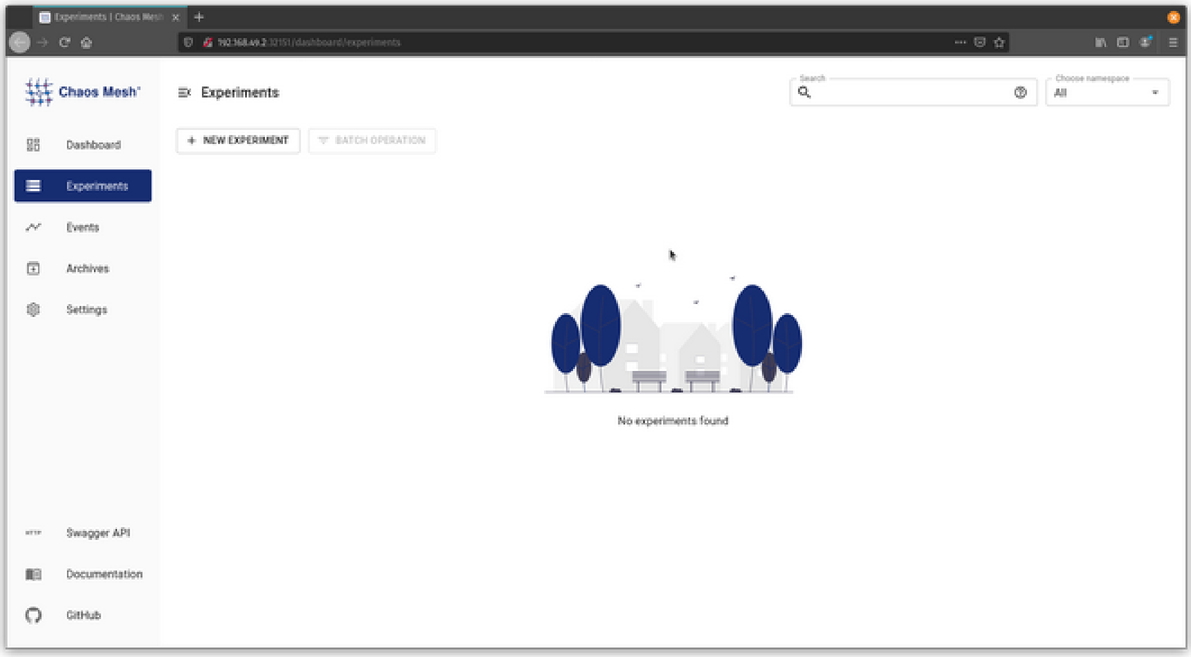
(Jess Cherry, CC BY-SA 4.0)

Remove an experiment

Removing an experiment altogether requires a simple `delete` command with the file name:

```
$ kubectl delete -f chaos-test.yaml
stresschaos.chaos-mesh.org "burn-cpu" deleted
```

Once again, you should see the desired result in the web UI.



No experiments found

(Jess Cherry, [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/))

Many of these tasks were done with the command line, but you can also create your own experiments using the UI or import experiments you created as YAML files. This helps many people become more comfortable with creating new experiments. There is also a Download button for each experiment, so you can see the YAML file you created by clicking a few buttons.

Final thoughts

Now that you have this new tool, you can get meshy with your environment. Chaos Mesh allows more user-friendly interaction, which means more people can join the chaos team. I hope you've learned enough here to expand on your chaos engineering. Happy pod hunting!

Test arbitrary pod failures on Kubernetes with kube-monkey

I have covered multiple chaos engineering tools in this series. The first article in this series explained what chaos engineering is; the second demonstrated how to get your system's steady state so that you can compare it against a chaos state; the third showed how to use Litmus to test arbitrary failures and experiments in your Kubernetes cluster; and the fourth article got into Chaos Mesh, an open source chaos orchestrator with a web user interface.

In this article, I want to talk about arbitrary pod failure. Kube-monkey offers an easy way to stress-test your systems by scheduling random termination pods in your cluster. This aims to encourage and validate the development of failure-resilient services. As in the previous walkthroughs, I’ll use Pop!_OS 20.04, Helm 3, Minikube 1.14.2, and Kubernetes 1.19.

Configure Minikube

If you haven’t already done so, install Minikube in whatever way makes sense for your environment. If you have enough resources, I recommend giving your virtual machine a bit more than the default memory and CPU power:

```
$ minikube config set memory 8192
❗  These changes take effect upon a minikube delete and then a minikube start
$ minikube config set cpus 6
❗  These changes take effect upon a minikube delete and then a minikube start
```

Then start and check your system's status:

```
$ minikube start
minikube v1.14.2 on Debian bullseye/sid
minikube 1.19.0 is available! http://github.com/kubernetes/minikube
To disable this notice: 'minikube config set WantUpdateNotification false'
Using the docker driver based on user configuration
Starting control plane node minikube in cluster minikube
```

```
Verifying Kubernetes components...
Done! kubectl is now configured to use "minikube" by default
$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Preconfiguring with deployments

Start by adding some small deployments to run chaos against. These deployments will need some special labels, so you need to create a new Helm chart. The following labels will help kube-monkey determine what to kill if the app is opted-in to doing chaos and understand what details are behind the chaos:

- **kube-monkey/enabled**: This setting opts you in to starting the chaos
- **kube-monkey/mtbf**: This stands for mean time between failure (in days). For example, if it's set to 3, the Kubernetes (K8s) app expects to have a pod killed approximately every third weekday.
- **kube-monkey/identifier**: This is a unique identifier for the K8s apps; in this example, it will be "nginx"
- **kube-monkey/kill-mode**: The kube-monkey's default behavior is to kill only one pod in the cluster, but you can change it to add more:
    - **kill-all:** Kill every pod, no matter what is happening with a pod
    - **fixed:** Pick a number of pods you want to kill
    - **fixed-percent:** Kill a fixed percent of pods (for example, 50%)
- **kube-monkey/kill-value**: This is where you can specify a value for kill-mode
    - **fixed:** The number of pods to kill
    - **random-max-percent:** Maximum number (0–100) that kube-monkey can kill
    - **fixed-percent:** Percentage (0–100%) of pods to kill

Now that you have this background info, you can start creating a basic Helm chart.

I named this Helm chart `nginx`. I'll show only the changes to the Helm chart deployment labels below. You need to change the deployment YAML file, which is `nginx/templates` in this example:

```
$ /chaos/kube-monkey/helm/nginx/templates$ ls -la
```

```
total 40
drwxr-xr-x 3 jess jess 4096 May 15 14:46 .
drwxr-xr-x 4 jess jess 4096 May 15 14:46 ..
-rw-r--r-- 1 jess jess 1826 May 15 14:46 deployment.yaml
-rw-r--r-- 1 jess jess 1762 May 15 14:46 _helpers.tpl
-rw-r--r-- 1 jess jess  910 May 15 14:46 hpa.yaml
-rw-r--r-- 1 jess jess 1048 May 15 14:46 ingress.yaml
-rw-r--r-- 1 jess jess 1735 May 15 14:46 NOTES.txt
-rw-r--r-- 1 jess jess  316 May 15 14:46 serviceaccount.yaml
-rw-r--r-- 1 jess jess  355 May 15 14:46 service.yaml
drwxr-xr-x 2 jess jess 4096 May 15 14:46 tests
```

In your `deployment.yaml` file, find this section:

```
  template:
    metadata:
      {{- with .Values.podAnnotations }}
      annotations:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      labels:
        {{- include "nginx.selectorLabels" . | nindent 8 }}
```

And make these changes:

```
  template:
    metadata:
      {{- with .Values.podAnnotations }}
      annotations:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      labels:
        {{- include "nginx.selectorLabels" . | nindent 8 }}
        kube-monkey/enabled: enabled
        kube-monkey/identifier: monkey-victim
        kube-monkey/mtbf: '2'
        kube-monkey/kill-mode: "fixed"
        kube-monkey/kill-value: '1'
```

Move back one directory and find the `values` file:

```
$ /chaos/kube-monkey/helm/nginx/templates$ cd ../
$ /chaos/kube-monkey/helm/nginx$ ls
charts  Chart.yaml  templates  values.yaml
```

You need to change one line in the values file, from:

```
replicaCount: 1
```

to:

```
replicaCount: 8
```

This gives you eight different pods to test chaos against.

Move back one more directory and install the new Helm chart:

```
$ /chaos/kube-monkey/helm/nginx$ cd ../
$ /chaos/kube-monkey/helm$ helm install nginxtest nginx
NAME: nginxtest
LAST DEPLOYED: Sat May 15 14:53:47 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES: Get the application URL by running these commands:

$ export POD_NAME=$(kubectl get pods --namespace default -l
"app.kubernetes.io/name=nginx,app.kubernetes.io/instance=nginxtest" -o
jsonpath="{.items[0].metadata.name}")
$ export CONTAINER_PORT=$(kubectl get pod --namespace default $POD_NAME -o
jsonpath="{.spec.containers[0].ports[0].containerPort}")
$ echo "Visit http://127.0.0.1:8080 to use your application"
$ kubectl --namespace default port-forward $POD_NAME 8080:$CONTAINER_PORT
```

Then check the labels in your Nginx pods:

```
$ /chaos/kube-monkey/helm$ kubectl get pods -n default
NAME                     READY   STATUS    RESTARTS
nginxtest-8f967857-88zv7 1/1     Running   0
nginxtest-8f967857-8qb95 1/1     Running   0
nginxtest-8f967857-dlng7 1/1     Running   0
nginxtest-8f967857-h7mmc 1/1     Running   0
nginxtest-8f967857-pdzpq 1/1     Running   0
nginxtest-8f967857-rdpnb 1/1     Running   0
nginxtest-8f967857-rqv2w 1/1     Running   0
nginxtest-8f967857-tr2cn 1/1     Running   0
```

Chose the first pod to describe and confirm the labels are in place:

```
$ kubectl describe pod nginxtest-8f967857-88zv7 -n default
Name:      nginxtest-8f967857-88zv7
Namespace: default
Priority:  0
```

```
Node:  minikube/192.168.49.2
Start Time: Sat, 15 May 2021 15:11:37 -0400
Labels: app.kubernetes.io/instance=nginxtest
        app.kubernetes.io/name=nginx
        kube-monkey/enabled=enabled
        kube-monkey/identifier=monkey-victim
        kube-monkey/kill-mode=fixed
        kube-monkey/kill-value=1
        kube-monkey/mtbf=2
        pod-template-hash=8f967857
```

Configure and install kube-monkey

To install kube-monkey using Helm, you first need to run `git clone on` the kube-monkey repository:

```
$ /chaos$ git clone https://github.com/asobti/kube-monkey
Cloning into 'kube-monkey'...
remote: Enumerating objects: 14641, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 14641 (delta 18), reused 22 (delta 8), pack-reused 14594
Receiving objects: 100% (14641/14641), 30.56 MiB | 39.31 MiB/s, done.
Resolving deltas: 100% (6502/6502), done.
```

Change to the `kube-monkey/helm` directory:

```
$ cd kube-monkey/helm/
```

Then go into the Helm chart and find the `values.yaml` file:

```
$ cd kubemonkey/
$ ls
Chart.yaml  README.md  templates  values.yaml
```

Below, I list just the sections of the `values.yaml` file you need to change. They disable dry-run mode by changing it in the config section to `false`, then add the default namespace to the whitelist so that it can kill the pods you deployed. You must keep the `blacklistedNamespaces` value or you will cause severe damage to your system.

Change this:

```
config:
  dryRun: true
  runHour: 8
  startHour: 10
  endHour: 16
  blacklistedNamespaces:
    - kube-system
  whitelistedNamespaces:  []
```

To this:

```
config:
  dryRun: false
  runHour: 8
  startHour: 10
  endHour: 16
  blacklistedNamespaces:
    - kube-system
  whitelistedNamespaces:  ["default"]
```

In the debug section, set `enabled` and `schedule_immediate_kill` to `true`. This will show the pods being killed.

Change this:

```
  debug:
   enabled: false
   schedule_immediate_kill: false
```

To this:

```
  debug:
   enabled: true
   schedule_immediate_kill: true
```

Run a `helm install`:

```
$ helm install chaos kubemonkey
NAME: chaos
LAST DEPLOYED: Sat May 15 13:51:59 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
1. Wait until the application is rolled out:
$ kubectl -n default rollout status deployment chaos-kube-monkey
2. Check the logs:
```

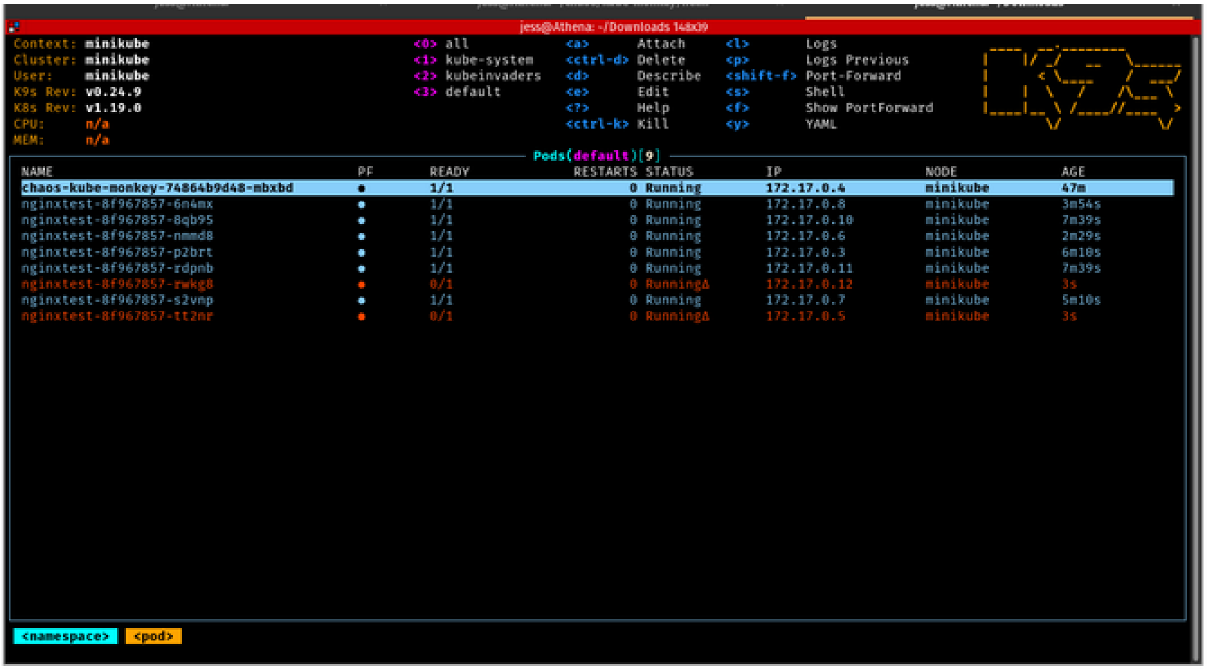
```
$ kubectl logs -f deployment.apps/chaos-kube-monkey -n default
```

Check the kube-monkey logs and see that the pods are being terminated:

```
$ /chaos/kube-monkey/helm$ kubectl logs \
-f deployment.apps/chaos-kube-monkey -n default
********** Today's schedule **********
k8 Api Kind     Kind Name    Termination Time
v1.Deployment   nginxtest    05/15/2021 15:15:22 -0400 EDT
********** End of schedule **********

I0515 19:15:22.343202  1 kubemonkey.go:70] Termination successfully executed for
v1.Deployment nginxtest
I0515 19:15:22.343216  1 kubemonkey.go:73] 0 scheduled terminations left.
I0515 19:15:22.343220  1 kubemonkey.go:76] All terminations done.
I0515 19:15:22.343278  1 kubemonkey.go:19] Debug mode detected!
I0515 19:15:22.343283  1 kubemonkey.go:20] Generating next schedule in 30 sec
```

You can also use K9s and watch the pods die.



```
Context: minikube            <0> all             <a>        Attach       <l>         Logs
Cluster: minikube            <1> kube-system     <ctrl-d>   Delete       <p>         Logs Previous
User:    minikube            <2> kubeinvaders    <d>        Describe     <shift-f>   Port-Forward
K9s Rev: v0.24.9             <3> default         <e>        Edit         <s>         Shell
K8s Rev: v1.19.0                                 <?>        Help         <f>         Show PortForward
CPU:     n/a                                     <ctrl-k>   Kill         <y>         YAML
MEM:     n/a
                                        Pods(default)[9]
NAME                              PF   READY     RESTARTS STATUS     IP             NODE       AGE
chaos-kube-monkey-74864b9d48-mbxbd ●   1/1              0 Running    172.17.0.4     minikube   47m
nginxtest-8f967857-6n4mx          ●    1/1              0 Running    172.17.0.8     minikube   3m54s
nginxtest-8f967857-8qb95          ●    1/1              0 Running    172.17.0.10    minikube   7m39s
nginxtest-8f967857-nmmd8          ●    1/1              0 Running    172.17.0.6     minikube   2m29s
nginxtest-8f967857-p2brt          ●    1/1              0 Running    172.17.0.3     minikube   6m10s
nginxtest-8f967857-rdpnb          ●    1/1              0 Running    172.17.0.11    minikube   7m39s
nginxtest-8f967857-rwkg8          ●    0/1              0 RunningΔ   172.17.0.12    minikube   3s
nginxtest-8f967857-s2vnp          ●    1/1              0 Running    172.17.0.7     minikube   5m10s
nginxtest-8f967857-tt2nr          ●    0/1              0 RunningΔ   172.17.0.5     minikube   3s
```

(Jess Cherry, CC BY-SA 4.0)

Congratulations! You now have a running chaos test with arbitrary failures. Anytime you want, you can change your applications to test at a certain day of the week and time of day.

Final thoughts

While kube-monkey is a great chaos engineering tool, it does require heavy configurations. Therefore, it isn't the best starter chaos engineering tool for someone new to Kubernetes. Another drawback is you have to edit your application's Helm chart for chaos testing to run.

This tool would be best positioned in a staging environment to watch how applications respond to arbitrary failure regularly. This gives you a long-term way to keep track of unsteady states using cluster monitoring tools. It also keeps notes that you can use for recovery of your internal applications in production.

Play Doom on Kubernetes

Do you ever feel nostalgic for Doom and other blocky video games, the ones that didn't require much more than a mouse and the hope that you could survive on a LAN with your friends? You know what I'm talking about; the days when your weekends were consumed with figuring out how you could travel with your desktop and how many Mountain Dews you could fit in your cargo pants pockets? If this memory puts a warm feeling in your heart, well, this article is for you.

Get ready to play Doom again, only this time you'll be playing for a legitimate work reason: doing chaos engineering. I'll be using my fork of Kube DOOM (with a new Helm chart because that's how I sometimes spend my weekends). I also have a pull request with the original Kube DOOM creator that I'm waiting to hear about.

The first article in this series explained what chaos engineering is, and the second demonstrated how to get your system's steady state so that you can compare it against a chaos state. In the next few articles, I introduced some chaos engineering tools you can use: Litmus for testing arbitrary failures and experiments in your Kubernetes cluster; Chaos Mesh, an open source chaos orchestrator with a web user interface; and Kube-monkey for stress-testing your systems by scheduling random termination pods in your cluster.

In this article, I'll use Pop!_OS 20.04, Helm 3, Minikube 1.14.2, a VNC viewer, and Kubernetes 1.19.

Preinstall pods with Helm

Before moving forward, you'll need to deploy some pods into your cluster. To do this, I generated a simple Helm chart and changed the replicas in my values file from 1 to 8.

If you need to generate a Helm chart, you can read my article on creating a Helm chart for guidance. I created a Helm chart named `nginx` and created a namespace to install my chart into using the commands below.

Create a namespace:

```
$ kubectl create ns nginx
```

Install the chart in your new namespace with a name:

```
$ helm install chaos-pods nginx -n nginx
NAME: chaos-pods
LAST DEPLOYED: Sun May 23 10:15:52 2021
NAMESPACE: nginx
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:

$ export POD_NAME=$(kubectl get pods --namespace nginx -l
"app.kubernetes.io/name=nginx,app.kubernetes.io/instance=chaos-pods" -o
jsonpath="{.items[0].metadata.name}")

$ export CONTAINER_PORT=$(kubectl get pod --namespace nginx $POD_NAME -o
jsonpath="{.spec.containers[0].ports[0].containerPort}")

$ echo "Visit http://127.0.0.1:8080 to use your application"

$ kubectl --namespace nginx port-forward $POD_NAME 8080:$CONTAINER_PORT
```

Install Kube DOOM

You can use any Virtual Network Computer (VNC) viewer you want; I installed TigerVNC on my Linux box. There are several ways you can set up Kube DOOM. Before I generated my Helm chart, you could set it up with kind or use it locally with Docker, and the README contains instructions for those uses.

Get started with a `git clone`:

```
$ git clone git@github.com:Alynder/kubedoom.git
Cloning into 'kubedoom'...
```

Then change directory into the `kubedoom/helm` folder:

```
$ cd kubedoom/helm/
```

Since the base values file is already set up correctly, you just need to run a single install command:

```
$ helm install kubedoom kubedoom/ -n kubedoom
NAME: kubedoom
LAST DEPLOYED: Mon May 31 11:16:58 2021
NAMESPACE: kubedoom
STATUS: deployed
REVISION: 1
NOTES: Get the application URL by running these commands:

$ export NODE_PORT=$(kubectl get --namespace kubedoom -o
jsonpath="{.spec.ports[0].nodePort}" services kubedoom-kubedoom-chart)

$ export NODE_IP=$(kubectl get nodes --namespace kubedoom -o
jsonpath="{.items[0].status.addresses[0].address}")

$ echo http://$NODE_IP:$NODE_PORT
```

Everything should be installed, set up, and ready to go.

Play with Kube DOOM

Now you just need to get in there, run a few commands, and start playing your new chaos video game. The first command is a port forward, followed by the VNC viewer connection command. The VNC viewer connection needs a password, which is `idbehold`.

Find your pod for the port forward:

```
$ kubectl get pods -n kubedoom
NAME                                READY    STATUS     RESTARTS
kubedoom-kubedoom-chart-676[...] 1/1      Running    0
```

Run the `port-forward` command using your pod name:

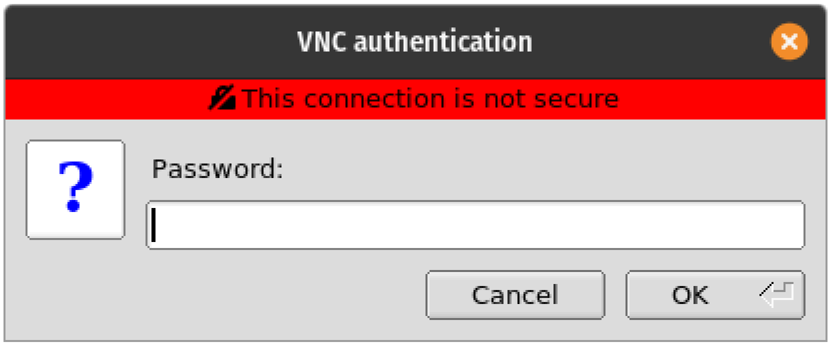
```
$  kubectl port-forward kubedoom-kubedoom-chart-676bcc5c9c-xkwpp \
5900:5900 -n kubedoom

Forwarding from 127.0.0.1:5900 -> 5900
Forwarding from [::1]:5900 -> 5900
```

Everything is ready to play, so you just need to run the VNC viewer command:

```
$ vncviewer viewer localhost:5900
TigerVNC Viewer 64-bit v1.10.1
 CConn:       Connected to host localhost port 5900
```

Next, you'll see the password request, so enter it (`idbehold`, as given above).



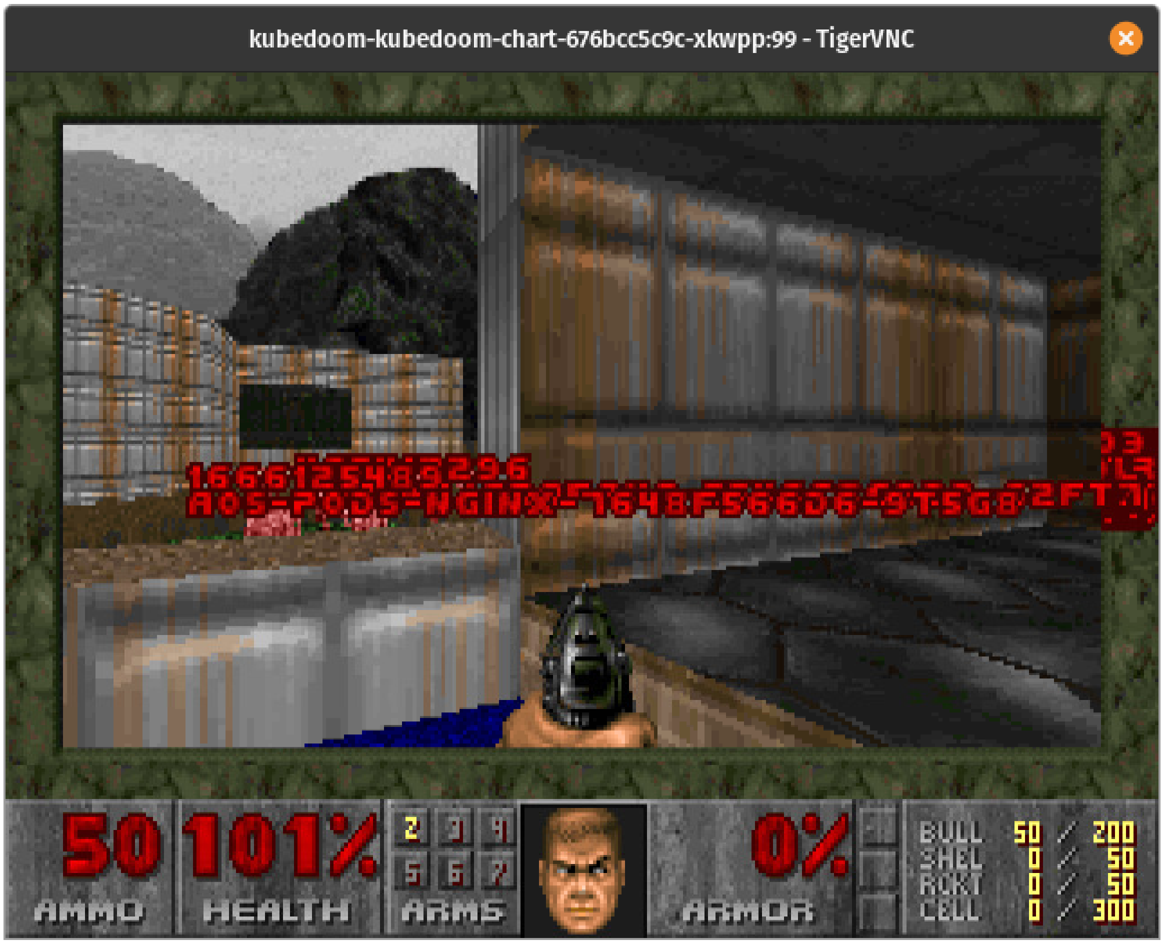
VNC authentication

This connection is not secure

Password:

(Jess Cherry, CC BY-SA 4.0)

Once you’re logged in, you should be able to walk around and see your enemies with pod names.

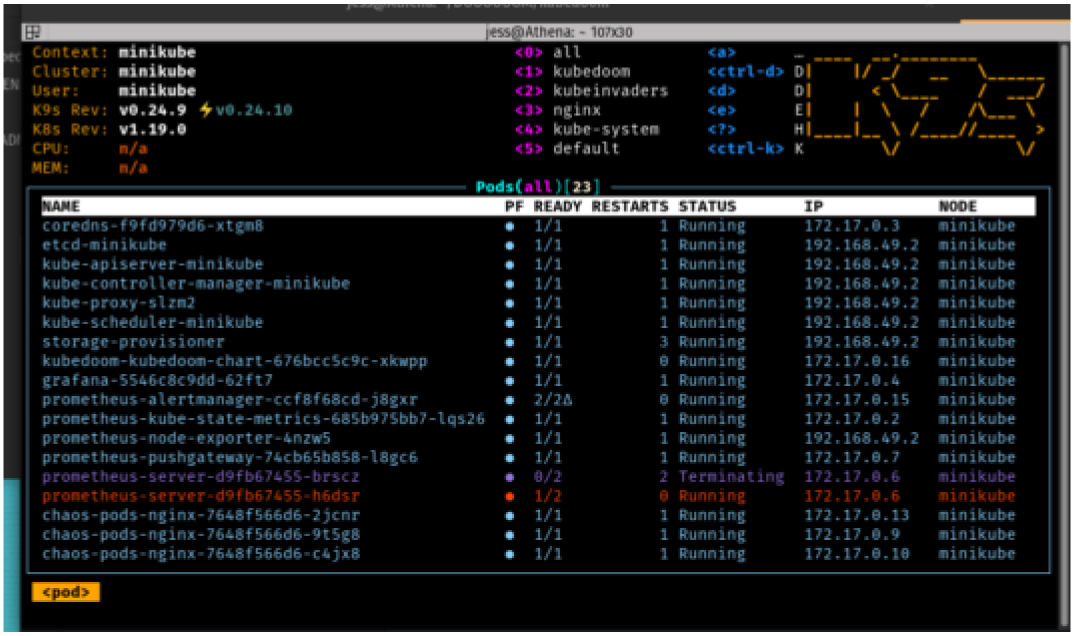


(Jess Cherry, CC BY-SA 4.0)

I'm terrible at this game, so I use some cheats to have a little more fun:

- Type `idspispopd` to walk straight through a wall to get to your army of pods.
- Can't handle the gun? That's cool; I'm bad at it, too. If you type `idkfa` and press the number **5**, you'll get a better weapon.

This is what it looks like when you kill something (I used k9s for this view).



```
jess@Athena: ~ 107x30
Context:  minikube                                      <0> all           <a>
Cluster:  minikube                                      <1> kubedoom      <ctrl-d> D
User:     minikube                                      <2> kubeinvaders  <d>      D
K9s Rev:  v0.24.9 ⚡v0.24.10                            <3> nginx         <e>      E
K8s Rev:  v1.19.0                                       <4> kube-system   <?>      H
CPU:      n/a                                           <5> default       <ctrl-k> K
MEM:      n/a
                                          Pods(all)[23]
 NAME                                            PF READY RESTARTS STATUS       IP            NODE
 coredns-f9fd979d6-xtgm8                         ●  1/1          1 Running      172.17.0.3    minikube
 etcd-minikube                                   ●  1/1          1 Running      192.168.49.2  minikube
 kube-apiserver-minikube                         ●  1/1          1 Running      192.168.49.2  minikube
 kube-controller-manager-minikube                ●  1/1          1 Running      192.168.49.2  minikube
 kube-proxy-slzm2                                ●  1/1          1 Running      192.168.49.2  minikube
 kube-scheduler-minikube                         ●  1/1          1 Running      192.168.49.2  minikube
 storage-provisioner                             ●  1/1          3 Running      192.168.49.2  minikube
 kubedoom-kubedoom-chart-676bcc5c9c-xkwpp        ●  1/1          0 Running      172.17.0.16   minikube
 grafana-5546c8c9dd-62ft7                        ●  1/1          1 Running      172.17.0.4    minikube
 prometheus-alertmanager-ccf8f68cd-j8gxr         ●  2/2Δ         0 Running      172.17.0.15   minikube
 prometheus-kube-state-metrics-685b975bb7-lqs26  ●  1/1          1 Running      172.17.0.2    minikube
 prometheus-node-exporter-4nzw5                  ●  1/1          1 Running      192.168.49.2  minikube
 prometheus-pushgateway-74cb65b858-l8gc6         ●  1/1          1 Running      172.17.0.7    minikube
 prometheus-server-d9fb67455-brscz               ●  0/2          2 Terminating  172.17.0.6    minikube
 prometheus-server-d9fb67455-h6dsr               ●  1/2          0 Running      172.17.0.6    minikube
 chaos-pods-nginx-7648f566d6-2jcnr               ●  1/1          1 Running      172.17.0.13   minikube
 chaos-pods-nginx-7648f566d6-9t5g8               ●  1/1          1 Running      172.17.0.9    minikube
 chaos-pods-nginx-7648f566d6-c4jx8               ●  1/1          1 Running      172.17.0.10   minikube

<pod>
```

(Jess Cherry, CC BY-SA 4.0)

Final notes

Because this application requires a cluster-admin role, you have to really pay attention to the names of the pods—you might run into a kube-system pod, and you'd better run away. If you kill one of those pods, you will kill an important part of the system.

I love this application because it's the quickest gamified way to do chaos engineering. It did remind me of how bad I was at this video game, but it was hilarious to try it. Happy hunting!

What happens when you terminate Kubernetes containers on purpose?

In this final article, I combine all the lessons you've learnt so far. Along with Grafana and Prometheus for monitoring for a steady state on your local cluster, I use Chaos Mesh and a small deployment and two experiments to see the difference between steady and not steady.

Preinstall pods with Helm

Before moving forward, you'll need to deploy some pods into your cluster. To do this, I generated a simple Helm chart and changed the replicas in my values file from 1 to 8.

If you need to generate a Helm chart, you can read the appendix article on creating a Helm chart for guidance. I created a Helm chart named `nginx` and created a namespace to install my chart into using the commands below.

Create a namespace:

```
$ kubectl create ns nginx
```

Install the chart in your new namespace with a name:

```
$ helm install chaos-pods nginx -n nginx
NAME: chaos-pods
LAST DEPLOYED: Sun May 23 10:15:52 2021
NAMESPACE: nginx
STATUS: deployed
NOTES: Get the application URL by running these commands:

$ export POD_NAME=$(kubectl get pods --namespace nginx \
-l "app.kubernetes.io/name=nginx,app.kubernetes.io/instance=chaos-pods" \
-o jsonpath="{.items[0].metadata.name}")
```

```
$ export CONTAINER_PORT=$(kubectl get pod --namespace nginx \
$POD_NAME -o jsonpath="{.spec.containers[0].ports[0].containerPort}")

$ echo "Visit http://127.0.0.1:8080 to use your application"
$ kubectl --namespace nginx port-forward $POD_NAME 8080:$CONTAINER_PORT
```

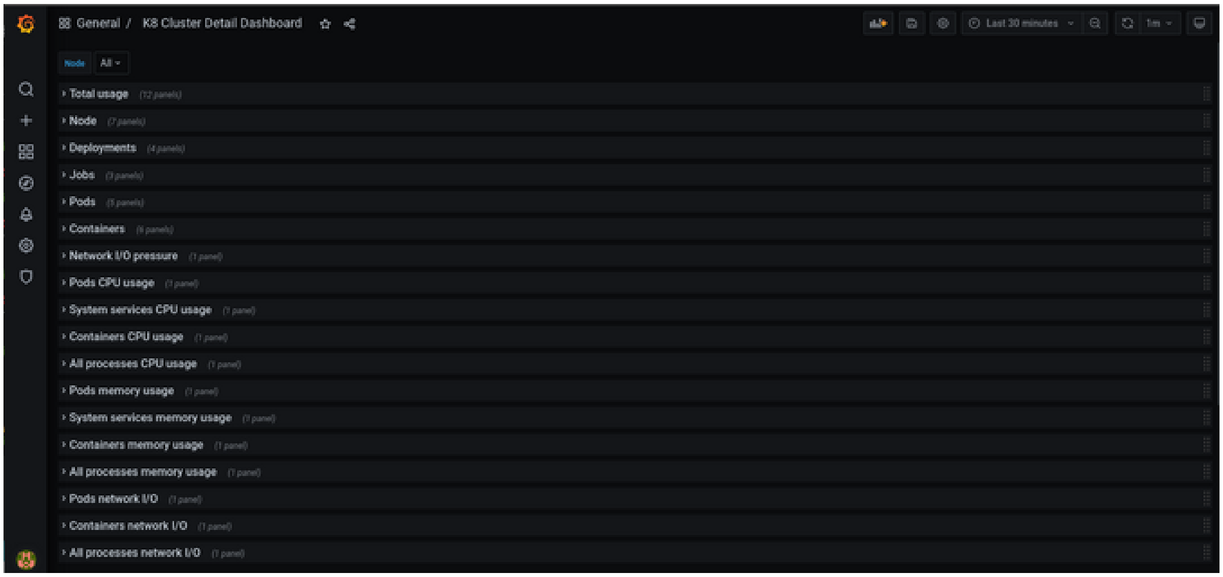
Monitoring and marinating

Next, install and set up Prometheus and Grafana following the steps in the previous articles in this series. However, you must make make the following changes in the installation:

```
$ kubectl create ns monitoring
$ helm install prometheus prometheus-community/prometheus -n monitoring
$ helm install grafana bitnami/grafana -n monitoring
```

Now that everything is installed in separate namespaces, set up your dashboards and let Grafana marinate for a couple of hours to catch a nice steady state. If you're in a staging or dev cluster at work, it would be even better to let everything sit for a week or so.

For this walkthrough, I will use the K8 Cluster Detail Dashboard (dashboard 10856), which provides various drop-downs with details about your cluster.



General / K8 Cluster Detail Dashboard

Node All

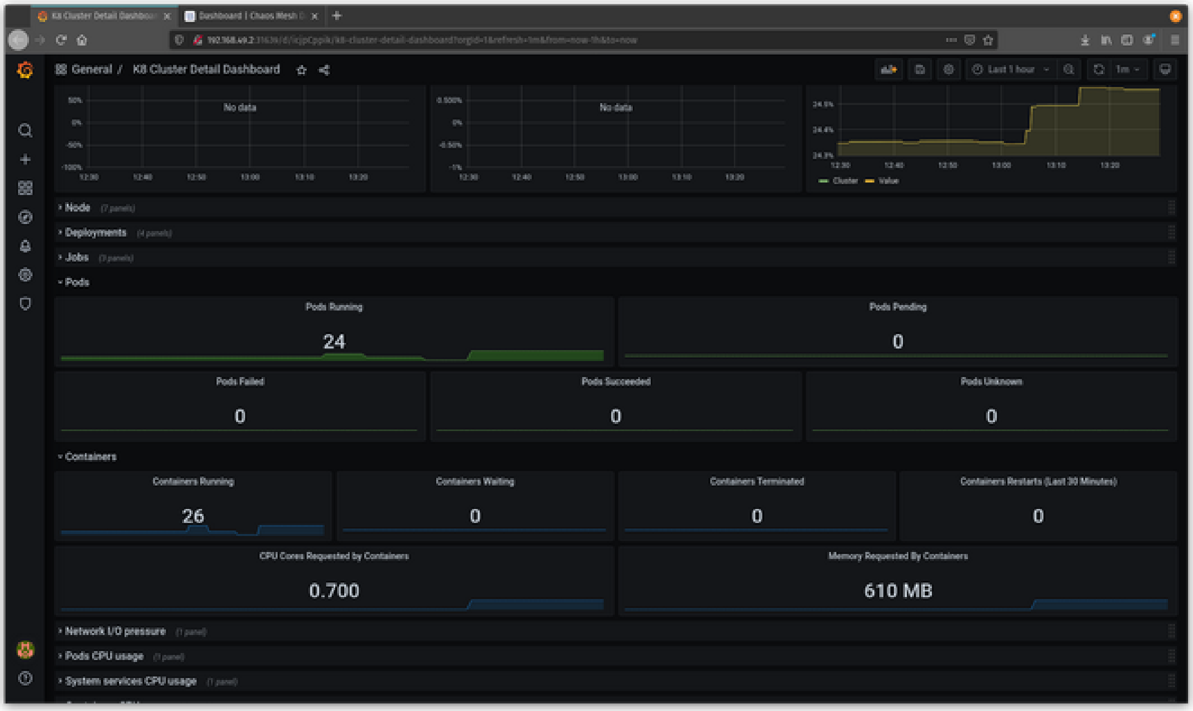
- Total usage (12 panels)
- Node (7 panels)
- Deployments (4 panels)
- Jobs (3 panels)
- Pods (5 panels)
- Containers (6 panels)
- Network I/O pressure (1 panel)
- Pods CPU usage (1 panel)
- System services CPU usage (1 panel)
- Containers CPU usage (1 panel)
- All processes CPU usage (1 panel)
- Pods memory usage (1 panel)
- System services memory usage (1 panel)
- Containers memory usage (1 panel)
- All processes memory usage (1 panel)
- Pods network I/O (1 panel)
- Containers network I/O (1 panel)
- All processes network I/O (1 panel)

(Jess Cherry, CC BY-SA 4.0)

Test #1: Container killing with Grafana and Chaos Mesh

Install and configure Chaos Mesh using the steps in my previous article. Once that is set up, you can add some new experiments to test and observe with Grafana.

Start by setting up an experiment to kill containers. First, look at your steady state.



(Jess Cherry, CC BY-SA 4.0)

Next, make a kill-container experiment pointed at your Nginx containers. I created an `experiments` directory and then the `container-kill.yaml` file:

```
$ mkdir experiments
$ cd experiments/
$ touch container-kill.yaml
```

The file looks like this:

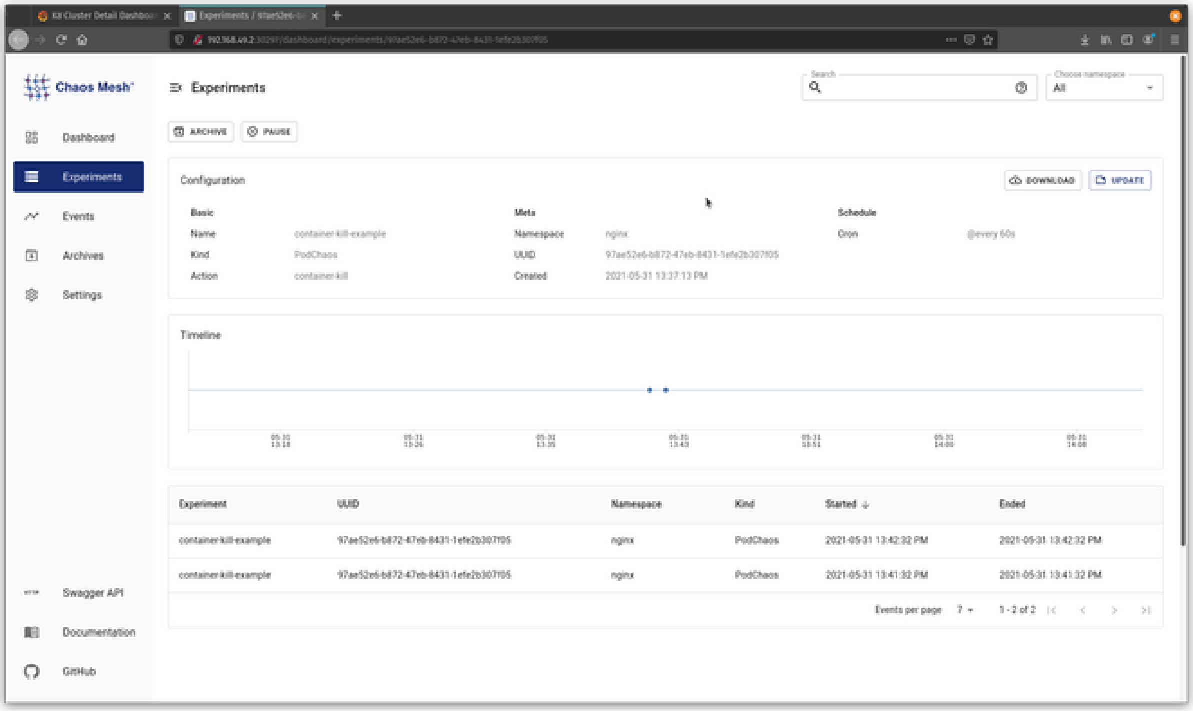
```
apiVersion: chaos-mesh.org/v1alpha1
kind: PodChaos
metadata:
  name: container-kill-example
  namespace: nginx
spec:
  action: container-kill
  mode: one
  containerName: 'nginx'
  selector:
    labelSelectors:
      'app.kubernetes.io/instance': 'nginx'
  scheduler:
    cron: '@every 60s'
```

Once it starts, this experiment kills an `nginx` container every minute.

Apply your file:

```
$ kubectl apply -f container-kill.yaml
podchaos.chaos-mesh.org/container-kill-example created
```

Now that the experiment is in place, watch it running in Chaos Mesh.



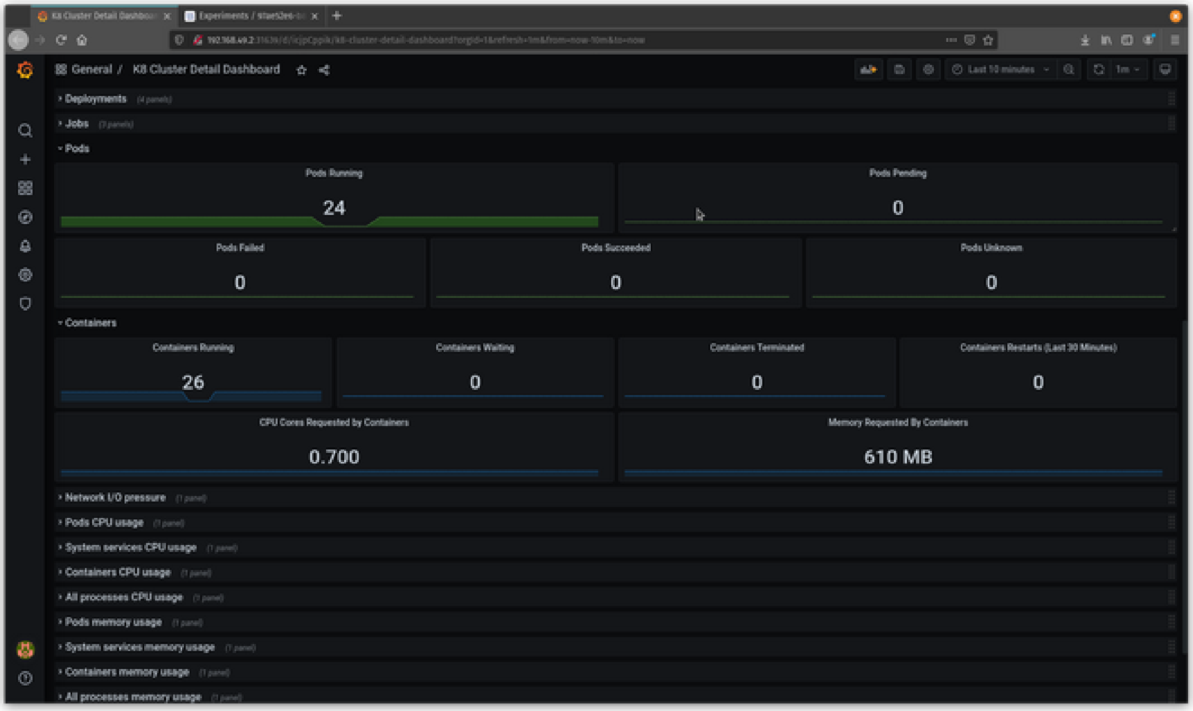
Configuration

Basic		Meta		Schedule	
Name	container-kill-example	Namespace	nginx	Cron	@every 60s
Kind	PodChaos	UUID	97ae52e6-b872-47eb-8431-1efe2b307f05		
Action	container-kill	Created	2021-05-31 13:37:13 PM		

Experiment	UUID	Namespace	Kind	Started ↓	Ended
container-kill-example	97ae52e6-b872-47eb-8431-1efe2b307f05	nginx	PodChaos	2021-05-31 13:42:32 PM	2021-05-31 13:42:32 PM
container-kill-example	97ae52e6-b872-47eb-8431-1efe2b307f05	nginx	PodChaos	2021-05-31 13:41:32 PM	2021-05-31 13:41:32 PM

(Jess Cherry, CC BY-SA 4.0)

Look into Grafana and see a notable change in the state of the pods and containers.

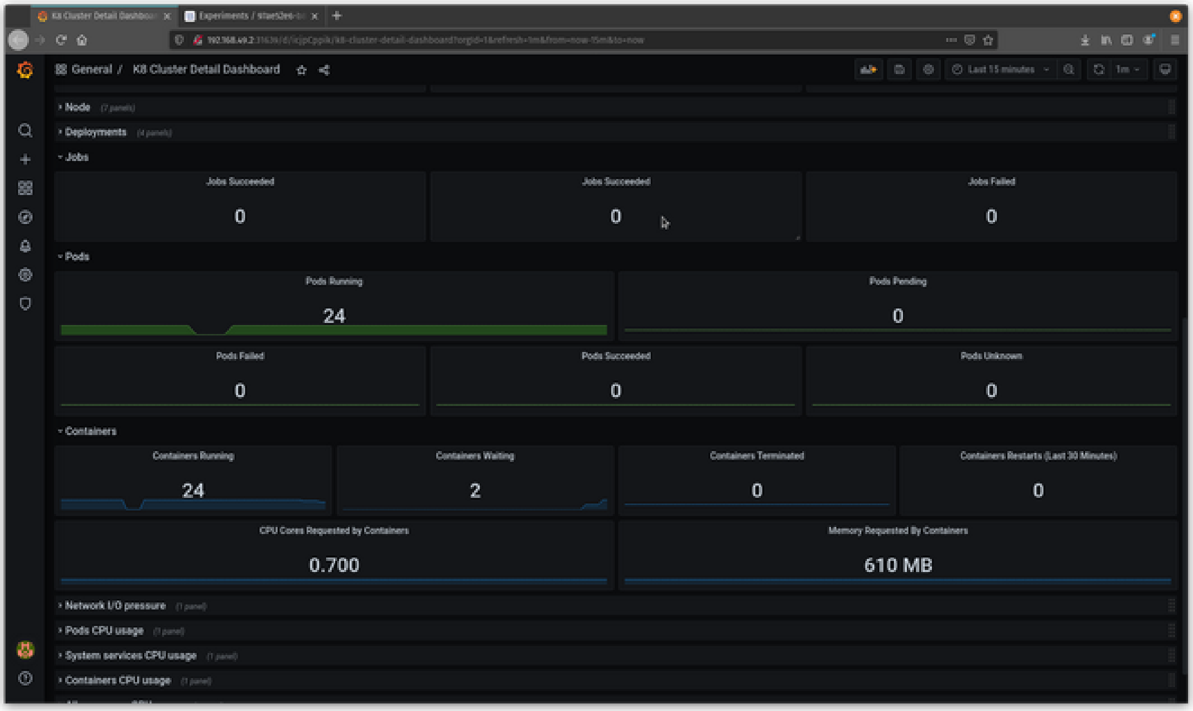

General / K8 Cluster Detail Dashboard
Last 10 minutes
1m
Deployments (4 panels)
Jobs (3 panels)
Pods
Pods Running
24
Pods Pending
0
Pods Failed
0
Pods Succeeded
0
Pods Unknown
0
Containers
Containers Running
26
Containers Waiting
0
Containers Terminated
0
Containers Restarts (Last 30 Minutes)
0
CPU Cores Requested by Containers
0.700
Memory Requested By Containers
610 MB
Network I/O pressure (1 panel)
Pods CPU usage (1 panel)
System services CPU usage (1 panel)
Containers CPU usage (1 panel)
All processes CPU usage (1 panel)
Pods memory usage (1 panel)
System services memory usage (1 panel)
Containers memory usage (1 panel)
All processes memory usage (1 panel)

(Jess Cherry, CC BY-SA 4.0)

If you change the kill time and reapply the experiment, you will see even more going on in Grafana. For example, change `@every 60s` to `@every 30s` and reapply the file:

```
$ kubectl apply -f container-kill.yaml
podchaos.chaos-mesh.org/container-kill-example configured
```

You can see the disruption in Grafana with two containers sitting in waiting status.



Node (7 panels)

Deployments (4 panels)

Jobs

Jobs Succeeded 0

Jobs Succeeded 0

Jobs Failed 0

Pods

Pods Running 24

Pods Pending 0

Pods Failed 0

Pods Succeeded 0

Pods Unknown 0

Containers

Containers Running 24

Containers Waiting 2

Containers Terminated 0

Containers Restarts (Last 30 Minutes) 0

CPU Cores Requested by Containers 0.700

Memory Requested By Containers 610 MB

Network I/O pressure (1 panel)

Pods CPU usage (1 panel)

System services CPU usage (1 panel)

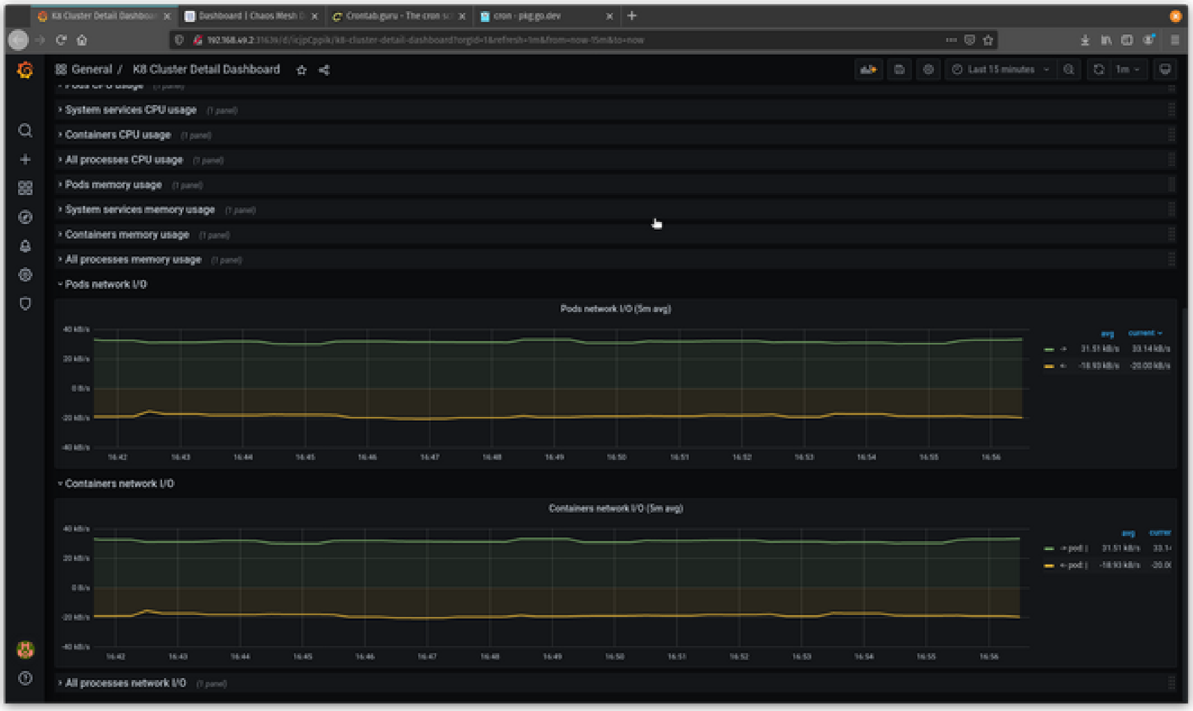
Containers CPU usage (1 panel)

(Jess Cherry, CC BY-SA 4.0)

Now that you know how the containers reacted, go into the Chaos Mesh user interface and pause the experiment.

Test #2: Networking with Grafana and Chaos Mesh

The next test works with network delays to see what happens if there are issues between pods. First, grab your steady state from Grafana.


K8 Cluster Detail Dashboard
Dashboard | Chaos Mesh
Crontab.guru - The cron
cron - pkg.go.dev
General / K8 Cluster Detail Dashboard
Last 15 minutes
1m
System services CPU usage (1 panel)
Containers CPU usage (1 panel)
All processes CPU usage (1 panel)
Pods memory usage (1 panel)
System services memory usage (1 panel)
Containers memory usage (1 panel)
All processes memory usage (1 panel)
Pods network I/O
Pods network I/O (5m avg)
40 kB/s
20 kB/s
0 B/s
-20 kB/s
-40 kB/s
16:42
16:43
16:44
16:45
16:46
16:47
16:48
16:49
16:50
16:51
16:52
16:53
16:54
16:55
16:56
avg
current
-> 31.51 kB/s 33.14 kB/s
<- -18.93 kB/s -20.00 kB/s
Containers network I/O
Containers network I/O (5m avg)
-> pod: | 31.51 kB/s 33.1
<- pod: | -18.93 kB/s -20.0
All processes network I/O (1 panel)

(Jess Cherry, CC BY-SA 4.0)

Create a `networkdelay.yaml` file for your experiment:

```
$ touch networkdelay.yaml
```

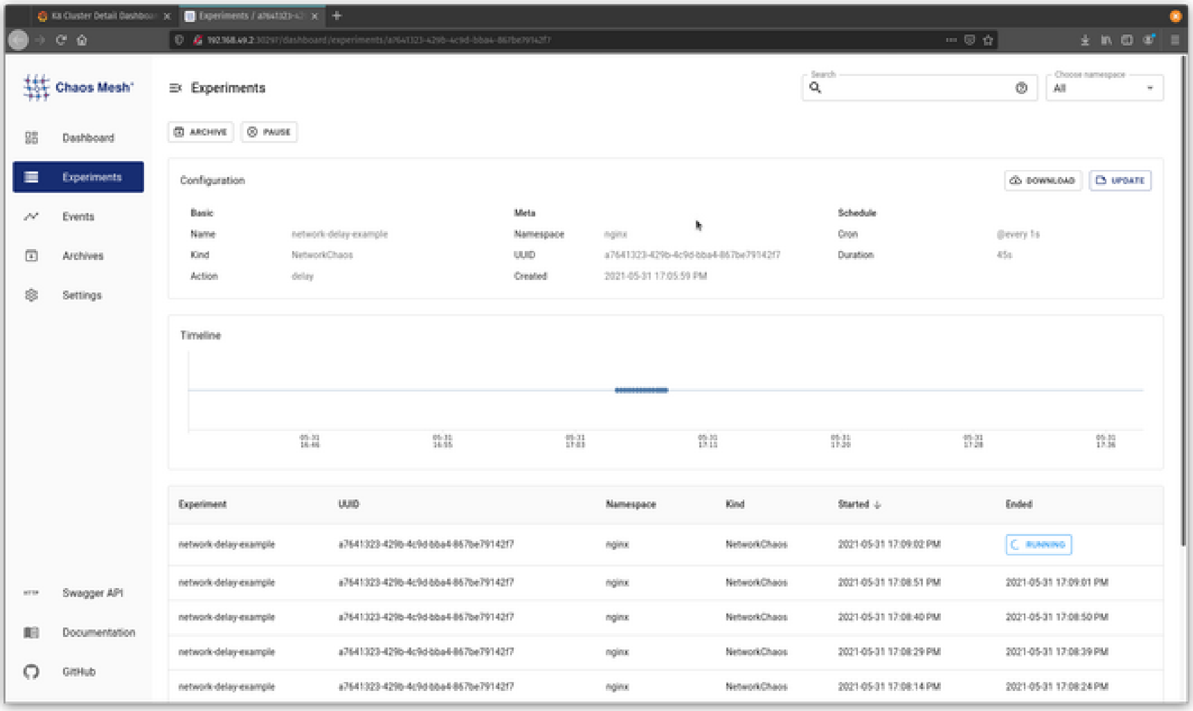
Then add some network delay details. This example runs a delay in the `nginx` namespace against your namespace instances. The packet-sending delay will be 90ms, the jitter will be 90ms, and the jitter correlation will be 25%:

```
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: network-delay-example
  namespace: nginx
spec:
  action: delay
  mode: one
  selector:
    labelSelectors:
      'app.kubernetes.io/instance': 'nginx'
  delay:
    latency: "90ms"
    correlation: "25"
    jitter: "90ms"
  duration: "45s"
  scheduler:
    cron: "@every 1s"
```

Save and apply the file:

```
$ kubectl apply -f  networkdelay.yaml
networkchaos.chaos-mesh.org/network-delay-example created
```

It should show up in Chaos Mesh as an experiment.



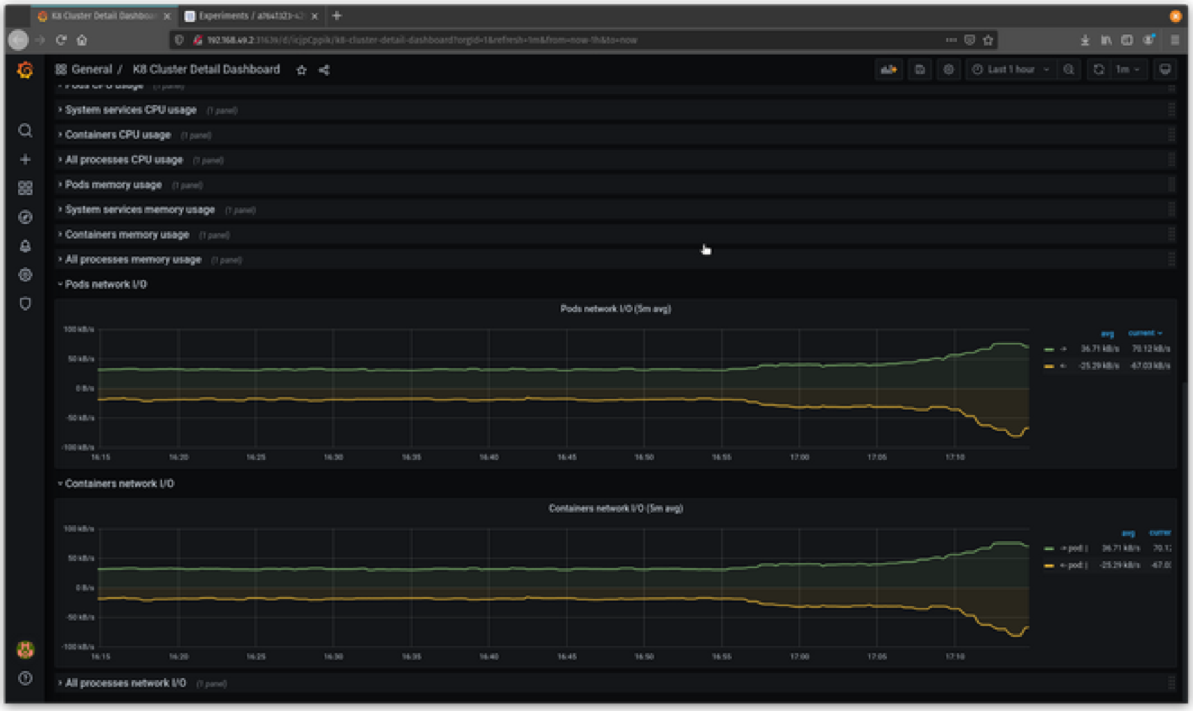
Configuration

Basic		Meta		Schedule	
Name	network-delay-example	Namespace	nginx	Cron	@every 1s
Kind	NetworkChaos	UUID	a7641323-429b-4c9d-bba4-867be79142f7	Duration	45s
Action	delay	Created	2021-05-31 17:05:59 PM		

Experiment	UUID	Namespace	Kind	Started ↓	Ended
network-delay-example	a7641323-429b-4c9d-bba4-867be79142f7	nginx	NetworkChaos	2021-05-31 17:09:02 PM	RUNNING
network-delay-example	a7641323-429b-4c9d-bba4-867be79142f7	nginx	NetworkChaos	2021-05-31 17:08:51 PM	2021-05-31 17:09:01 PM
network-delay-example	a7641323-429b-4c9d-bba4-867be79142f7	nginx	NetworkChaos	2021-05-31 17:08:40 PM	2021-05-31 17:08:50 PM
network-delay-example	a7641323-429b-4c9d-bba4-867be79142f7	nginx	NetworkChaos	2021-05-31 17:08:29 PM	2021-05-31 17:08:39 PM
network-delay-example	a7641323-429b-4c9d-bba4-867be79142f7	nginx	NetworkChaos	2021-05-31 17:08:14 PM	2021-05-31 17:08:24 PM

(Jess Cherry, CC BY-SA 4.0)

Now that it is running pretty extensively using your configuration, you should see an interesting, noticeable change in Grafana.


K8 Cluster Detail Dashboard
Experiments / a7641323-4
192.168.49.2:31639/d/icjpCppik/k8-cluster-detail-dashboard?orgId=1&refresh=1m&from=now-1h&to=now
General / K8 Cluster Detail Dashboard
Last 1 hour
1m
System services CPU usage (1 panel)
Containers CPU usage (1 panel)
All processes CPU usage (1 panel)
Pods memory usage (1 panel)
System services memory usage (1 panel)
Containers memory usage (1 panel)
All processes memory usage (1 panel)
Pods network I/O
Pods network I/O (5m avg)
100 kB/s
50 kB/s
0 B/s
-50 kB/s
-100 kB/s
16:15
16:20
16:25
16:30
16:35
16:40
16:45
16:50
16:55
17:00
17:05
17:10
avg
current
-> 36.71 kB/s 70.12 kB/s
<- -25.29 kB/s -67.03 kB/s
Containers network I/O
Containers network I/O (5m avg)
-> pod: | 36.71 kB/s
<- pod: | -25.29 kB/s
All processes network I/O (1 panel)

(Jess Cherry, CC BY-SA 4.0)

In the graphs, you can see the pods are experiencing a delay.

Congratulations! You have a more detailed way to keep track of and test networking issues.

Chaos engineering final thoughts

Chaos engineering has a lot of evolving yet to do, but the more people involved, the better the testing and tools will get. Chaos engineering can be fun and easy to set up, which means everyone—from your dev team to your administration—can do it. This will make your infrastructure and the apps it hosts more dependable.

Appendix

How to make a Helm chart in 10 minutes

A good amount of my day-to-day involves creating, modifying, and deploying Helm charts to manage the deployment of applications. Helm is an application package manager for Kubernetes, which coordinates the download, installation, and deployment of apps. Helm charts are the way we can define an application as a collection of related Kubernetes resources.

So why would anyone use Helm? Helm makes managing the deployment of applications easier inside Kubernetes through a templated approach. All Helm charts follow the same structure while still having a structure flexible enough to represent any type of application you could run on Kubernetes. Helm also supports versioning since deployment needs are guaranteed to change with time. The alternative is to use multiple configuration files that you manually apply to your Kubernetes cluster to bring an application up. If we've learned anything from seeing infrastructure as code, it's that manual processes inevitably lead to errors. Helm charts give us a chance to apply that same lesson to the world of Kubernetes.

In this example, we'll be walking through using Helm with minikube, a single-node testing environment for Kubernetes. We will make a small Nginx web server application. For this example, I have minikube version 1.9.2 and Helm version 3.0.0 installed on my Linux laptop. To get set up, do the following.

- Download and configure minikube by following the excellent documentation here.
- Download and configure Helm using your favorite package manager listed here or manually from the releases.

Create a Helm chart

Start by confirming we have the prerequisites installed:

```
$ helm version
version.BuildInfo{Version:"v3.6.3"
$ minikube status ## if it shows Stopped, run `minikube start`
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Starting a new Helm chart requires one simple command:

```
$ helm create mychartname
```

For the purposes of this tutorial, name the chart **buildachart**:

```
$ helm create buildachart
Creating buildachart
$ ls buildachart/
Chart.yaml   charts/   templates/   values.yaml
```

Examine the chart's structure

Now that you have created the chart, take a look at its structure to see what's inside. The first two files you see—**Chart.yaml** and **values.yaml**—define what the chart is and what values will be in it at deployment.

Look at **Chart.yaml**, and you can see the outline of a Helm chart's structure:

```
apiVersion: v2
name: buildachart
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart
type: application

# Chart version
version: 0.1.0

# Application version
appVersion: 1.16.0
```

The first part includes the API version that the chart is using (this is required), the name of the chart, and a description of the chart. The next section describes the type of chart (an application by default), the version of the chart you will deploy, and the application version (which should be incremented as you make changes).

The most important part of the chart is the template directory. It holds all the configurations for your application that will be deployed into the cluster. As you can see below, this application has a basic deployment, ingress, service account, and service. This directory also includes a test directory, which includes a test for a connection into the app. Each of these application features has its own template files under **templates/**:

```
$ ls templates/
NOTES.txt  _ helpers.tpl   deployment.yaml
ingress.yaml  service.yaml  serviceaccount.yaml  tests/
```

There is another directory, called **charts**, which is empty. It allows you to add dependent charts that are needed to deploy your application. Some Helm charts for applications have up to four extra charts that need to be deployed with the main application. When this happens, the **values** file is updated with the values for each chart so that the applications will be configured and deployed at the same time. This is a far more advanced configuration (which I will not cover in this introductory article), so leave the **charts/** folder empty.

Understand and edit values

Template files are set up with formatting that collects deployment information from the **values.yaml** file. Therefore, to customize your Helm chart, you need to edit the values file. By default, the **values.yaml** file looks like:

```
# Default values for buildachart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1
image:
  repository: nginx
  pullPolicy: IfNotPresent
imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""
serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  name:
podSecurityContext: {}
  # fsGroup: 2000
securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000
service:
  type: ClusterIP
  port: 80
ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths: []
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - chart-example.local
resources: {}
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi
nodeSelector: {}
tolerations: []
affinity: {}
```

Basic configurations

Beginning at the top, you can see that the **replicaCount** is automatically set to 1, which means that only one pod will come up. You only need one pod for this example, but you can see how easy it is to tell Kubernetes to run multiple pods for redundancy.

The **image** section has two things you need to look at: the **repository** where you are pulling your image and the **pullPolicy**. The pullPolicy is set to **IfNotPresent**; which means that the image will download a new version of the image if one does not already exist in the cluster. There are two other options for this: **Always**, which means it will pull the image on every deployment or restart (I always suggest this in case of image failure), and **Latest**, which will always pull the most up-to-date version of the image available. Latest can be useful if you trust your image repository to be compatible with your deployment environment, but that's not always the case.

Change the value to **Always**.

Before:

```
image:
  repository: nginx
  pullPolicy: IfNotPresent
```

After:

```
image:
  repository: nginx
  pullPolicy: Always
```

Naming and secrets

Next, take a look at the overrides in the chart. The first override is **imagePullSecrets**, which is a setting to pull a secret, such as a password or an API key you've generated as credentials for a private registry. Next are **nameOverride** and **fullnameOverride**. From the moment you ran **helm create**, its name (buildachart) was added to a number of configuration files—from the YAML ones above to the **templates/helper.tpl** file. If you need to rename a chart after you create it, this section is the best place to do it, so you don't miss any configuration files.

Change the chart's name using overrides.

Before:

```
imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""
```

After:

```
imagePullSecrets: []
nameOverride: "cherry-awesome-app"
fullnameOverride: "cherry-chart"
```

Accounts

Service accounts provide a user identity to run in the pod inside the cluster. If it's left blank, the name will be generated based on the full name using the **helpers.tpl** file. I recommend always having a service account set up so that the application will be directly associated with a user that is controlled in the chart.

As an administrator, if you use the default service accounts, you will have either too few permissions or too many permissions, so change this.

Before:

```
serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, the fullname is used
  Name:
```

After:

```
serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, the fullname is used
  Name: cherrybomb
```

Security

You can configure pod security to set limits on what type of filesystem group to use or which user can and cannot be used. Understanding these options is important to securing a Kubernetes pod, but for this example, I will leave this alone.

```
podSecurityContext: {}
  # fsGroup: 2000
securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000
```

Networking

There are two different types of networking options in this chart. One uses a local service network with a **ClusterIP** address, which exposes the service on a cluster-internal IP. Choosing this value makes the service associated with your application reachable only from within the cluster (and through **ingress**, which is set to **false** by default). The other networking option is **NodePort**, which exposes the service on each Kubernetes node's IP address on a statically assigned port. This option is recommended for running minikube, so use it for this how-to.

Before:

```
service:
  type: ClusterIP
  port: 80
ingress:
  enabled: false
```

After:

```
service:
  type: NodePort
  port: 80
ingress:
  enabled: false
```

Resources

Helm allows you to explicitly allocate hardware resources. You can configure the maximum amount of resources a Helm chart can request and the highest limits it can receive. Since I'm using Minikube on a laptop, I'll set a few limits by removing the curly braces and the hashes to convert the comments into commands.

Before:

```
resources: {}
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
```

```
  #   cpu: 100m
  #   memory: 128Mi
```

After:

```
resources:
   limits:
     cpu: 100m
     memory: 128Mi
   requests:
     cpu: 100m
     memory: 128Mi
```

Tolerations, node selectors, and affinities

These last three values are based on node configurations. Although I cannot use any of them in my local configuration, I'll still explain their purpose.

nodeSelector comes in handy when you want to assign part of your application to specific nodes in your Kubernetes cluster. If you have infrastructure-specific applications, you set the node selector name and match that name in the Helm chart. Then, when the application is deployed, it will be associated with the node that matches the selector.

Tolerations, **tainting**, and **affinities** work together to ensure that pods run on separate nodes. Node affinity is a property of *pods* that *attracts* them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite—they allow a *node* to *repel* a set of pods.

In practice, if a node is tainted, it means that it is not working properly or may not have enough resources to hold the application deployment. Tolerations are set up as a key/value pair watched by the scheduler to confirm a node will work with a deployment.

Node affinity is conceptually similar to **nodeSelector:** it allows you to constrain which nodes your pod is eligible to be scheduled based on labels on the node. However, the labels differ because they match rules that apply to scheduling.

```
nodeSelector: {}
tolerations: []
affinity: {}
```

Deploy ahoy!

Now that you've made the necessary modifications to create a Helm chart, you can deploy it using a Helm command, add a name point to the chart, add a values file, and send it to a namespace:

```
$ helm install my-cherry-chart buildachart/ --values buildachart/values.yaml
Release “my-cherry-chart” has been upgraded. Happy Helming!
```

The command's output will give you the next steps to connect to the application, including setting up port forwarding so you can reach the application from your localhost. To follow those instructions and connect to an Nginx load balancer:

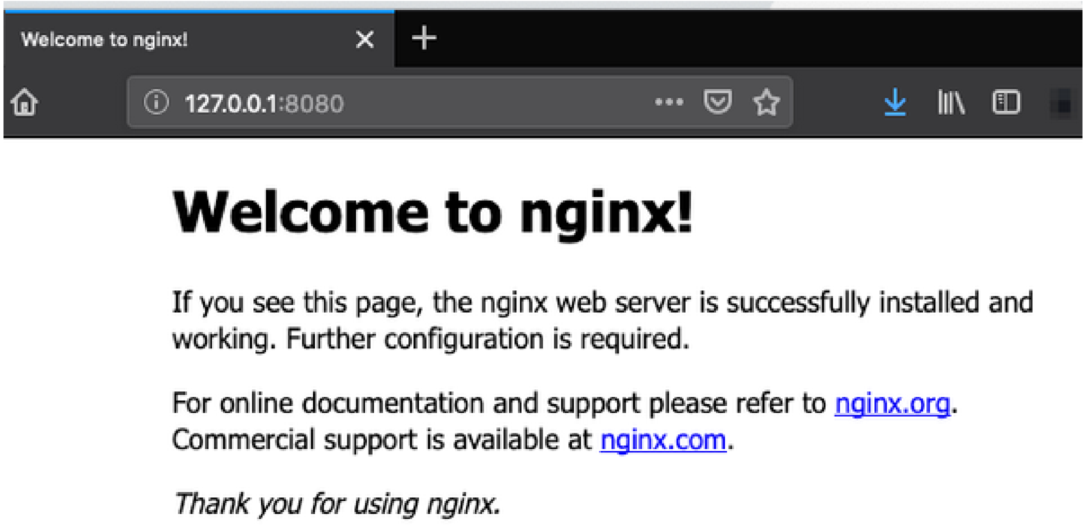
```
$ export POD_NAME=$(kubectl get pods -l \
"app.kubernetes.io/name=buildachart,app.kubernetes.io/instance=my-cherry-chart" \
-o jsonpath="{.items[0].metadata.name}")

$ echo "Visit http://127.0.0.1:8080 to use your application"
Visit http://127.0.0.1:8080 to use your application

$ kubectl port-forward $POD_NAME 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

View the deployed application

To view your application, open your web browser:



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

(Jess Cherry, CC BY-SA 4.0)

Congratulations! You've deployed an Nginx web server by using a Helm chart!

There is a lot to learn as you explore what Helm charts can do. If you want to double-check your work, visit my example repository on GitHub.