



## Your instructions for Chaos Engineering:

The Chaos-Engineering experiment must be completed within 1 minute.



## Phase 0: Preprocessing

Cleaning the cluster `kind-chaos-eater-cluster` ... Done

```
$ kubectl delete workflow --all --context kind-chaos-eater-cluster -n chaos-
workflow.chaos-mesh.org "chaos-experiment-20241124-125548" deleted
$ kubectl delete workflownode --all --context kind-chaos-eater-cluster -n cl
workflownode.chaos-mesh.org "fault-injection-overlapped-workflows-s2j6j" de
workflownode.chaos-mesh.org "fault-injection-phase-9vmc2" deleted
workflownode.chaos-mesh.org "fault-injection-suspend-w7xxb" deleted
workflownode.chaos-mesh.org "fault-injection-suspend-workflow-j8bqs" delete
workflownode.chaos-mesh.org "fault-injection-suspend-workflow2-mtg5k" delete
workflownode.chaos-mesh.org "fault-injection-suspend2-nrlb2" deleted
workflownode.chaos-mesh.org "fault-networkchaos-df9sz" deleted
workflownode.chaos-mesh.org "fault-podchaos-6njzf" deleted
workflownode.chaos-mesh.org "fault-unittest-example-pod-running-qcc98" dele
workflownode.chaos-mesh.org "fault-unittest-example-service-availability-98!
workflownode.chaos-mesh.org "post-unittest-example-pod-running-zjd9m" delete
workflownode.chaos-mesh.org "post-unittest-example-service-availability-2df
workflownode.chaos-mesh.org "post-validation-overlapped-workflows-p4c4f" de
workflownode.chaos-mesh.org "post-validation-phase-qsjq5" deleted
workflownode.chaos-mesh.org "post-validation-suspend-sk8s2" deleted
workflownode.chaos-mesh.org "post-validation-suspend-workflow-5crck" delete
workflownode.chaos-mesh.org "pre-unittest-example-pod-running-sgsb5" delete
workflownode.chaos-mesh.org "pre-unittest-example-service-availability-hldj
workflownode.chaos-mesh.org "pre-validation-parallel-workflows-6wd58" delete
workflownode.chaos-mesh.org "pre-validation-phase-4lgcj" deleted
workflownode.chaos-mesh.org "the-entry-g75nb" deleted
$ kubectl delete deployments --all --context kind-chaos-eater-cluster -n ch
No resources found
```

Input instructions for your Chaos Engineering



```
$ kubectl delete services --all --context kind-chaos-eater-cluster -n chaos-
No resources found
```

```
$ kubectl delete all --all-namespaces --context kind-chaos-eater-cluster -l
pod "example-deployment-6977c97c99-7dcbd" deleted
pod "example-deployment-6977c97c99-gbxgz" deleted
pod "example-deployment-6977c97c99-mr5cc" deleted
service "example-service" deleted
deployment.apps "example-deployment" deleted
```

## K8s manifest(s) to be deployed:

nginx/pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: example
spec:
  restartPolicy: Never
  containers:
  - name: example-container
    image: nginx:1.17.1
    ports:
    - containerPort: 80
```

nginx/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

## Deploying resources... Done

```
$ skaffold run --kube-context kind-chaos-eater-cluster -l project=chaos-eater
No tags generated
Starting test...
Starting deploy...
Loading images into kind cluster nodes...
Images loaded in 281ns
- pod/example-pod created
- service/example-service created
Waiting for deployments to stabilize...
- pods is ready.
Deployments stabilized in 3.088 seconds
You can also run [skaffold run --tail] to get the logs
```

## Resource statuses

```
$ kubectl get all --all-namespaces --context kind-chaos-eater-cluster --selector=project=chaos-eater
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	pod/example-pod	1/1	Running	0	5s

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	ClusterIP	10.96.107.80	<none>

## Summary of each manifest:

nginx/pod.yaml

- This manifest defines a Kubernetes Pod.
- The Pod is named 'example-pod'.
- It includes metadata with a label 'app: example'.
- The Pod's restart policy is set to 'Never', meaning it won't restart automatically if it fails.
- The Pod contains one container named 'example-container'.
- The container uses the 'nginx:1.17.1' image.
- The container exposes port 80 for network traffic.

nginx/service.yaml

- This manifest defines a Kubernetes Service.
- The Service is named 'example-service'.
- It uses the 'v1' API version.

- The Service selects pods with the label 'app: example'.
- It exposes the Service on port 80 using the TCP protocol.
- The target port for the Service is also port 80, meaning it forwards traffic to port 80 on the selected pods.

## Resiliency issues/weaknesses in the manifests:

### Issue #0: Pod Restart Policy

- details: The Pod will not restart automatically if it fails, which can lead to downtime.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: restartPolicy: Never

### Issue #1: Single Pod Deployment

- details: Using a single Pod without replication can lead to a single point of failure.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: kind: Pod

## Application of the manifests:

### Thoughts:

The manifests provided are for a Kubernetes Pod and Service that utilize the Nginx web server. The Pod manifest specifies a container running the Nginx image, which is a popular open-source web server used for serving static content, reverse proxying, and load balancing. The Service manifest is configured to expose this Pod on port 80, which is the default port for HTTP traffic. Given the use of Nginx and the configuration of the Service, it is logical to assume that these manifests are intended to deploy a simple web server application. The file names and the use of Nginx further support this assumption.

### Assumed application:

A simple web server application using Nginx to serve static web content or act as a reverse proxy.

## Summary of your instructions for Chaos Engineering:

- The Chaos-Engineering experiment must be completed within 1 minute.

## Phase 1: Hypothesis

---

### Steady-state definition

✓ **Steady state #1: PodRunningState\_nginx\_pod**



## Description

The first issue to address is the Pod's restart policy set to 'Never', which means the Pod will not restart automatically if it fails. This is a critical issue because it can lead to downtime if the Pod encounters any failure. Therefore, the steady state should ensure that the Pod is running and available. A measurable output for this steady state is the number of running Pods. Since there is only one Pod defined in the manifest, the steady state should verify that this Pod is in the 'Running' state.

## Current state inspection

To inspect the PodRunningState\_nginx\_pod, I will use the K8s API. The reason for choosing the K8s API is that it allows us to directly check the status of the Pod within the Kubernetes cluster. We need to verify that the Pod is in the 'Running' state, which is a specific status of a Kubernetes resource. The K8s API provides the necessary functionality to query the current state of the Pod and ensure it is running. The script will check the Pod's status every second for a duration of 5 seconds, which is sufficient to confirm the Pod's state within the 1-minute constraint of the Chaos Engineering experiment.

tool: `k8s` duration: `5s`

Inspection script (Python) `k8s_PodRunningState_nginx_pod.py` :

```
import os
import time
from kubernetes import client, config

def check_pod_status(namespace, pod_name, duration):
    # Load Kubernetes configuration based on the environment
    if os.getenv('KUBERNETES_SERVICE_HOST'):
        config.load_incluster_config()
    else:
        config.load_kube_config()

    v1 = client.CoreV1Api()
    running_count = 0

    for _ in range(duration):
        try:
            pod = v1.read_namespaced_pod(name=pod_name, namespace=namespace)
            if pod.status.phase == 'Running':
                running_count += 1
            print(f"Pod {pod_name} status: {pod.status.phase}")
        except client.exceptions.ApiException as e:
```

```

        print(f"Exception when calling CoreV1Api->read_namespace
time.sleep(1)

print(f"Pod {pod_name} was running {running_count} out of {durat

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='Check the running
    parser.add_argument('--duration', type=int, default=5, help='Dur
    args = parser.parse_args()
    check_pod_status(namespace='default', pod_name='example-pod', du

```

Pod `k8spodrunningstatenginxpod-pod` has completed successfully.

The inspection script's results (current states) are as follows:

```

Pod example-pod status: Running
Pod example-pod status: Running
Pod example-pod status: Running
Pod example-pod status: Running
Pod example-pod status: Running
Pod example-pod was running 5 out of 5 seconds.

```

## ► Threshold

The steady state we are considering is the 'PodRunningState\_nginx\_pod', which requires the Pod to be in the 'Running' state. The current state shows that the Pod was running 5 out of 5 seconds, which indicates a stable and consistent running state. To define a reasonable threshold, we should consider a percentage of time the Pod should be running within a given duration. Since the current state shows 100% uptime, we can introduce a small tolerance to account for potential brief fluctuations or delays in status updates. A reasonable threshold would be that the Pod should be running at least 90% of the time during the observation period. This allows for some minor fluctuations while still ensuring the Pod is generally stable and available.

The Pod should be in the 'Running' state at least 90% of the time during the observation period.

## 📄 Unit test to validate the steady state

To add the threshold assertion to the inspection Python script, I will calculate the percentage of time the Pod is in the 'Running' state during the specified duration. The

threshold is defined as the Pod being in the 'Running' state at least 90% of the time. Therefore, I will assert that the running percentage is greater than or equal to 90%. This will ensure that the Pod meets the steady state requirement. The test will loop for the specified duration, checking the Pod's status every second, and then calculate the running percentage at the end to perform the assertion.

```
import os
import time
import argparse
from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestPodRunningState(K8sAPIBase):
    def __init__(self, namespace, pod_name, duration):
        super().__init__()
        self.namespace = namespace
        self.pod_name = pod_name
        self.duration = duration

    def test_pod_running_state(self):
        running_count = 0

        # Loop for the specified duration, checking the Pod's status
        for _ in range(self.duration):
            try:
                pod = self.v1.read_namespaced_pod(name=self.pod_name, namespace=self.namespace)
                if pod.status.phase == 'Running':
                    running_count += 1
                print(f"Pod {self.pod_name} status: {pod.status.phase}")
            except client.exceptions.ApiException as e:
                print(f"Exception when calling CoreV1Api->read_namespaced_pod: {e}")
            time.sleep(1)

        # Calculate the percentage of time the Pod was running
        running_percentage = (running_count / self.duration) * 100
        print(f"Pod {self.pod_name} was running {running_count} out of {self.duration} seconds")

        # Assert that the Pod was running at least 90% of the time
        assert running_percentage >= 90, f"Pod {self.pod_name} was running {running_percentage}% of the time"

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Test the running state of a Pod')
    parser.add_argument('--duration', type=int, default=5, help='Duration of the test in seconds')
    args = parser.parse_args()

    # Create an instance of the test class and run the test
```

```
test = TestPodRunningState(namespace='default', pod_name='example-pod')
test.test_pod_running_state()
```

Pod `unittestpodrunningstatenginxpodmod0-pod` has completed successfully.

The inspection script's results (current states) are as follows:

```
Pod example-pod status: Running
Pod example-pod status: Running
Pod example-pod status: Running
Pod example-pod status: Running
Pod example-pod status: Running
Pod example-pod was running 5 out of 5 seconds.
```

## ✓ Steady state #2: ServiceRoutingState\_nginx\_service ^

### Description

The next issue to address is the 'Single Pod Deployment', which can lead to a single point of failure. This is a critical issue because if the single Pod fails, the entire service becomes unavailable. To ensure the system's resiliency, we need to verify that the Service is able to route traffic to the Pod. A measurable output for this steady state is the Service's ability to successfully forward requests to the Pod. Since the Service is configured to expose the Pod on port 80, the steady state should verify that the Service is able to handle HTTP requests and return a successful response (e.g., HTTP 200 status code). This will ensure that the Service is functioning correctly and can route traffic to the Pod, even if there is only one Pod available.

### Current state inspection

To inspect the ServiceRoutingState\_nginx\_service, we need to verify that the Service is able to route traffic to the Pod and return a successful HTTP response. Since this involves checking the communication status and response from the service, k6 is the appropriate tool to use. We will simulate HTTP requests to the service using k6 and check for successful responses (HTTP 200 status code). The test will run for a short duration with a few virtual users to ensure the service is functioning correctly.

tool: `k6` vus: `1` duration: `5s`

Inspection script (Javascript) `k6_ServiceRoutingState_nginx_service.js` :



```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  vus: 1,
  duration: '5s',
};

export default function () {
  const res = http.get('http://example-service.default.svc.cluster.local');
  check(res, {
    'is status 200': (r) => r.status === 200,
  });
}
```

Pod `k6serviceroutingstatenginxservice-pod` has completed successfully.

The inspection script's results (current states) are as follows:

```
✓ is status 200
```

```
checks.....: 100.00% 14157 out of 14157
data_received.....: 12 MB    2.4 MB/s
data_sent.....: 1.6 MB    311 kB/s
http_req_blocked.....: avg=8.02µs   min=1.37µs   med=4.22µs
http_req_connecting.....: avg=2.2µs    min=0s        med=0s
http_req_duration.....: avg=231.14µs min=79.6µs   med=233.6µs
  { expected_response:true }...: avg=231.14µs min=79.6µs   med=233.6µs
http_req_failed.....: 0.00%    0 out of 14157
http_req_receiving.....: avg=67.01µs  min=17.18µs  med=62.29µs
http_req_sending.....: avg=17.67µs  min=3.9µs    med=14.78µs
http_req_tls_handshaking.....: avg=0s       min=0s       med=0s
http_req_waiting.....: avg=146.44µs min=54.61µs  med=149.9µs
http_reqs.....: 14157    2831.146568/s
iteration_duration.....: avg=338.88µs min=127.81µs med=334.8µs
iterations.....: 14157    2831.146568/s
vus.....: 1      min=1      max=1
vus_max.....: 1      min=1      max=1
```

## ► Threshold

The steady state we are evaluating is the Service's ability to route traffic to the Pod and return a successful HTTP response. The k6 test results show that 100% of the requests

received a 200 status code, indicating that the Service is currently functioning correctly. To define a reasonable threshold, we should consider a success rate that accounts for minor fluctuations while still ensuring the system is operating normally. Given the current success rate is 100%, a threshold of 95% success rate for HTTP 200 responses would be reasonable. This allows for some tolerance in case of transient issues, while still maintaining a high standard for system availability.

HTTP 200 response success rate  $\geq 95\%$

## Unit test to validate the steady state

To add the threshold check to the k6 script, I will include a 'thresholds' property in the 'options' section. This property will specify that the success rate of HTTP 200 responses should be at least 95%. This is done by using the 'http\_req\_failed' metric, which tracks the percentage of failed requests. By setting the threshold to 'http\_req\_failed: ['rate<0.05']', we ensure that the success rate is at least 95%, as the failure rate must be less than 5%.

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  vus: 1,
  duration: '5s',
  thresholds: {
    // Ensure that the success rate for HTTP 200 responses is at least 95%
    'http_req_failed': ['rate<0.05'],
  },
};

export default function () {
  const res = http.get('http://example-service.default.svc.cluster.local');
  check(res, {
    'is status 200': (r) => r.status === 200,
  });
}
```

Pod `unittestservice-routingstatenginx-service-mod0-pod` has completed successfully.

The inspection script's results (current states) are as follows:

```
✓ is status 200
```

```

checks.....: 100.00% 14790 out of 14790
data_received.....: 13 MB    2.5 MB/s
data_sent.....: 1.6 MB    325 kB/s
http_req_blocked.....: avg=8.42µs    min=1.33µs    med=4.2
http_req_connecting.....: avg=2.23µs    min=0s        med=0s
http_req_duration.....: avg=213.2µs   min=82.68µs   med=205
  { expected_response:true }...: avg=213.2µs   min=82.68µs   med=205
✓ http_req_failed.....: 0.00%    0 out of 14790
http_req_receiving.....: avg=62.4µs    min=13.08µs   med=57.
http_req_sending.....: avg=18.07µs   min=3.71µs    med=15.
http_req_tls_handshaking.....: avg=0s        min=0s        med=0s
http_req_waiting.....: avg=132.72µs  min=55.45µs   med=127
http_reqs.....: 14790    2957.8035/s
iteration_duration.....: avg=323.39µs  min=124.44µs  med=312
iterations.....: 14790    2957.8035/s
vus.....: 1        min=1        max=1
vus_max.....: 1        min=1        max=1

```

## Fault definition

### ✓ Scenario: Cyber Attack Simulation

#### 🗨 Description

Given the system's weaknesses, a cyber attack targeting the availability of the web server could be highly impactful. The Pod's restart policy set to 'Never' and the single Pod deployment create vulnerabilities that could lead to downtime if the Pod fails. To simulate a cyber attack, we can introduce network disruptions and resource exhaustion to test the system's resilience. First, we will simulate network latency to see if the Service can still route traffic effectively, which tests the 'ServiceRoutingState\_nginx\_service'. Then, we will introduce a Pod failure to test the 'PodRunningState\_nginx\_pod', checking if the system can recover from a Pod failure without automatic restarts. This sequence simulates a cyber attack by first degrading network performance and then causing a Pod failure, testing the system's ability to maintain its steady states under stress.

#### ⚙ Fault-injection sequence

```

NetworkChaos ({'namespace': 'default', 'service': 'example-service'}) → PodChaos
({'namespace': 'default', 'pod': 'example-pod'})

```

## ⚙ Detailed fault parameters

Detailed parameters of `NetworkChaos` ({'namespace': 'default', 'service': 'example-service'})

```
▼ {
  "action" : "delay"
  "direction" : "to"
  "mode" : "all"
  ▼ "selector" : {
    ▼ "namespaces" : [
      | 0 : "default"
    ]
    ▼ "labelSelectors" : {
      | "app" : "example"
    }
  }
  ▼ "delay" : {
    | "latency" : "100ms"
    | "jitter" : "10ms"
  }
}
```

Detailed parameters of `PodChaos` ({'namespace': 'default', 'pod': 'example-pod'})

```
▼ {
  "action" : "pod-kill"
  "mode" : "one"
  ▼ "selector" : {
    ▼ "namespaces" : [
      | 0 : "default"
    ]
    ▼ "labelSelectors" : {
      | "app" : "example"
    }
  }
}
```

## Phase 2: Chaos Experiment

## Chaos Engineering Experiment Plan

### Time Schedule

Given the constraints of the experiment needing to be completed within 1 minute, we need to allocate time efficiently across the three phases: pre-validation, fault-injection, and post-validation. The pre-validation phase is crucial to ensure that the system is in its expected steady state before introducing any faults. Since we have two steady states to validate, we should allocate a short but sufficient time to verify both the Pod's running state and the Service's routing capability. A reasonable allocation would be 15 seconds for pre-validation. The fault-injection phase is where we introduce the chaos to observe the system's behavior under stress. Since we have two types of faults to inject (network delay and pod kill), we should allocate the majority of the time here to observe the system's response. A 30-second duration for fault injection allows us to introduce both faults and monitor their effects. Finally, the post-validation phase is necessary to ensure that the system returns to its steady states after the faults are removed. We should allocate 15 seconds for post-validation to verify that both the Pod and Service have returned to their expected states. This allocation ensures that we have a balanced approach to validating the system's resiliency within the 1-minute constraint.

Total experiment time: 1m

Pre-validation Phase: 15s

Fault-injection Phase: 30s

Post-validation Phase: 15s

### Pre-validation Phase (15s)

In the pre-validation phase, we need to ensure that the system is in its expected steady states before we proceed with fault injection. Given the two steady states defined, we will conduct unit tests to verify each of them. The tests will be executed sequentially within the 15-second time frame allocated for this phase. The first test will verify the 'PodRunningState\_nginx\_pod', ensuring that the Pod is running at least 90% of the time. This test will be conducted first because it directly checks the availability of the Pod, which is critical for the subsequent service routing test. The second test will verify the 'ServiceRoutingState\_nginx\_service', ensuring that the Service can successfully route HTTP requests with a 95% success rate for HTTP 200 responses. This test will

follow immediately after the first test. The sequential execution ensures that we first confirm the Pod's availability before checking the Service's routing capability, as the latter depends on the former. Each test is designed to run for a short duration to fit within the 15-second limit, with a brief grace period to allow for any initial setup or delays.

- Verified Steady State #0: `PodRunningState_nginx_pod`
  - Workflow Name: `pre-unittest-podrunningstatenginxpod`
  - Grace Period: `0s`
  - Duration: `7s`
- Verified Steady State #1: `ServiceRoutingState_nginx_service`
  - Workflow Name: `pre-unittest-serviceroutingstatenginxservice`
  - Grace Period: `7s`
  - Duration: `8s`

### Fault-injection Phase (30s)

In this fault-injection phase, we aim to simulate a cyber attack by introducing two types of faults: network latency and pod failure. The total duration for this phase is 30 seconds, so we need to carefully schedule the fault injections and unit tests to fit within this timeframe.

First, we will introduce a network delay using NetworkChaos to simulate network latency. This will help us observe how the system handles degraded network performance and affects the 'ServiceRoutingState\_nginx\_service'. We will start this fault injection immediately at the beginning of the phase and let it run for 15 seconds.

Simultaneously, we will run the unit test for 'ServiceRoutingState\_nginx\_service' to verify if the service can still route traffic effectively under network delay conditions. This test will also start at the beginning and run for 15 seconds.

After the network delay fault injection and the corresponding unit test, we will introduce a pod failure using PodChaos. This will help us observe the system's behavior when the pod is killed, affecting the 'PodRunningState\_nginx\_pod'. We will start this fault injection at the 15-second mark and let it run for 10 seconds.

Simultaneously, we will run the unit test for 'PodRunningState\_nginx\_pod' to verify if the pod remains in the running state at least 90% of the time during the observation period. This test will start at the 15-second mark and run for 10 seconds.

This staggered approach allows us to observe the system's behavior under each fault condition separately while ensuring that both steady states are tested within the 30-

second limit.

- Verified Steady State #0: `ServiceRoutingState_nginx_service`
  - Workflow Name: `fault-unittest-serviceroutingstatenginxservice`
  - Grace Period: `0s`
  - Duration: `15s`
- Verified Steady State #1: `PodRunningState_nginx_pod`
  - Workflow Name: `fault-unittest-podrunningstatenginxpod`
  - Grace Period: `15s`
  - Duration: `10s`
- Injected Faults #0: `NetworkChaos`
  - Workflow Name: `fault-networkchaos`
  - Grace Period: `0s`
  - Duration: `15s`
- Injected Faults #1: `PodChaos`
  - Workflow Name: `fault-podchaos`
  - Grace Period: `15s`
  - Duration: `10s`

### Post-validation Phase (15s)

In the post-validation phase, we need to ensure that the system has returned to its steady states after the fault injection. Given the 15-second time constraint, we will conduct unit tests to verify each steady state. The tests will be executed sequentially due to the short duration, ensuring that each steady state is verified independently. The first test will check the 'PodRunningState\_nginx\_pod' to confirm that the Pod is running and available. The second test will verify the 'ServiceRoutingState\_nginx\_service' to ensure that the Service can successfully route traffic to the Pod. Each test will have a brief grace period to allow the system to stabilize after the fault injection, followed by a short duration to perform the verification. This approach ensures that both steady states are validated within the available time, confirming the system's recovery.

- Verified Steady State #0: `PodRunningState_nginx_pod`
  - Workflow Name: `post-unittest-podrunningstatenginxpod`
  - Grace Period: `2s`
  - Duration: `6s`
- Verified Steady State #1: `ServiceRoutingState_nginx_service`
  - Workflow Name: `post-unittest-serviceroutingstatenginxservice`

- Grace Period: 8s
- Duration: 7s

### Summary

The Chaos-Engineering-experiment plan is structured into three phases: pre-validation, fault-injection, and post-validation, all to be completed within a 1-minute timeframe.

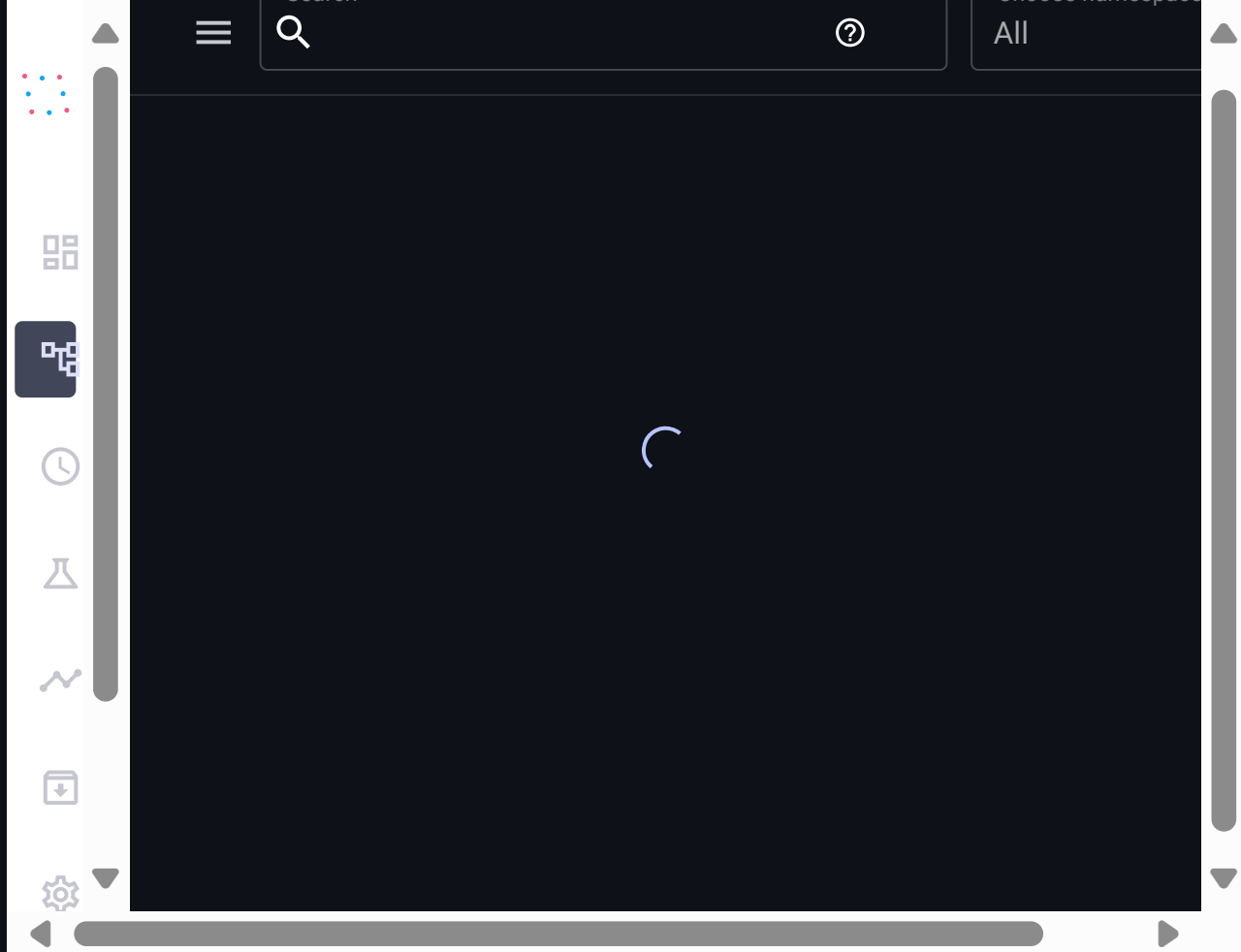
In the pre-validation phase, which lasts for 15 seconds, two unit tests are executed sequentially. The first test, 'PodRunningState\_nginx\_pod', is conducted with a workflow named 'pre-unittest-podrunningstatenginxpod'. It starts immediately with no delay and runs for 7 seconds to ensure the Pod is running at least 90% of the time. Following this, the second test, 'ServiceRoutingState\_nginx\_service', is executed with the workflow 'pre-unittest-serviceroutingstatenginxservice'. It begins after a 7-second grace period and runs for 8 seconds to verify that the Service can route HTTP requests with a 95% success rate for HTTP 200 responses.

The fault-injection phase spans 30 seconds and involves two simultaneous fault injections and unit tests. At the start of this phase, a network delay is introduced using 'NetworkChaos' with the workflow 'fault-networkchaos', running for 15 seconds. Concurrently, the 'ServiceRoutingState\_nginx\_service' unit test is executed with the workflow 'fault-unittest-serviceroutingstatenginxservice', also for 15 seconds, to assess the service's routing capability under network delay conditions. At the 15-second mark, a pod failure is introduced using 'PodChaos' with the workflow 'fault-podchaos', running for 10 seconds. Simultaneously, the 'PodRunningState\_nginx\_pod' unit test is conducted with the workflow 'fault-unittest-podrunningstatenginxpod', also for 10 seconds, to verify the pod's availability during the fault.

In the post-validation phase, lasting 15 seconds, two unit tests are executed sequentially to confirm the system's recovery. The first test, 'PodRunningState\_nginx\_pod', is conducted with the workflow 'post-unittest-podrunningstatenginxpod'. It starts after a 2-second grace period and runs for 6 seconds to ensure the Pod is running and available. The second test, 'ServiceRoutingState\_nginx\_service', follows with the workflow 'post-unittest-serviceroutingstatenginxservice'. It begins after an 8-second grace period and runs for 7 seconds to verify the Service's ability to route traffic to the Pod. This structured approach ensures a comprehensive assessment of the system's resilience and recovery within the 1-minute constraint.

**Completed the chaos experiment!**





## Phase 3: Analysis

---

The Chaos Engineering experiment results indicate several critical failures in the system's ability to maintain its steady states under the simulated fault conditions. Here's a detailed analysis of the failures:

### 1. Failed Test: `fault-unittest-podrunningstatenginxpod`

- **Observation:** The test failed because the Pod 'example-pod' was not found during the fault injection phase, resulting in a 404 error from the Kubernetes API.
- **Cause:** This failure is directly linked to the 'PodChaos' fault, which involved killing the Pod. Since the Pod's restart policy is set to 'Never', it did not automatically restart after being killed, leading to its unavailability.
- **Insight:** The 'restartPolicy: Never' configuration in the Pod manifest is a significant vulnerability. In a production environment, this would lead to downtime whenever the Pod fails, as there is no mechanism to bring it back online automatically.
- **Recommendation:** Change the Pod's restart policy to 'Always' or 'OnFailure' to ensure it restarts automatically after a failure, improving resilience against such disruptions.

### 2. Failed Test: `post-unittest-podrunningstatenginxpod`

- **Observation:** Similar to the fault phase, the Pod was not found during the post-validation phase, resulting in a 404 error.
- **Cause:** The Pod did not recover after the 'PodChaos' fault due to the same restart policy issue. Since the Pod was not restarted, it remained unavailable throughout the post-validation phase.
- **Insight:** This further emphasizes the critical nature of the restart policy configuration. Without automatic recovery, the system cannot return to its steady state after a fault.
- **Recommendation:** Implement a replication strategy using a Deployment or ReplicaSet to ensure multiple instances of the Pod are running, providing redundancy and automatic recovery.

### 3. Failed Test: post-unittest-serviceroutingstatenginxs-service

- **Observation:** The Service failed to route traffic, resulting in connection refused errors during the post-validation phase.
- **Cause:** The failure of the Pod meant there were no available endpoints for the Service to route traffic to, leading to the connection errors.
- **Insight:** The single Pod deployment is a single point of failure. When the Pod is unavailable, the Service cannot function, highlighting the need for redundancy.
- **Recommendation:** Use a Deployment with multiple replicas to ensure that the Service always has available Pods to route traffic to, even if one or more Pods fail.

Overall, the experiment highlights the critical need for redundancy and automatic recovery mechanisms in the system. Addressing the identified issues by adjusting the Pod's restart policy and implementing a Deployment with multiple replicas will significantly enhance the system's resilience against similar fault scenarios in the future.

## Phase 4: Improvement

### Reconfiguration

To improve the system's resiliency, I will replace the Pod manifest with a Deployment manifest. This will allow for multiple replicas of the Pod, providing redundancy and automatic recovery in case of failure. Additionally, I will set the restart policy to 'Always' to ensure Pods are restarted automatically if they fail.

Modification\_type: replace

File name: nginx/pod.yaml

Replace the Pod manifest with a Deployment manifest to provide redundancy and automatic recovery. Set the restart policy to 'Always' to ensure Pods are restarted automatically if they fail.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  labels:
    app: example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      restartPolicy: Always
      containers:
        - name: example-container
          image: nginx:1.17.1
          ports:
            - containerPort: 80
```

## Deploying reconfigured resources... Done

```
$ scaffold run --kube-context kind-chaos-eater-cluster -l project=chaos-eater
No tags generated
Starting test...
Starting deploy...
Loading images into kind cluster nodes...
Images loaded in 240ns
- deployment.apps/example-deployment configured
- service/example-service configured
Waiting for deployments to stabilize...
- deployment/example-deployment is ready.
Deployments stabilized in 4.095 seconds
You can also run [scaffold run --tail] to get the logs
```

## Resource statuses

```
$ kubectl get all --all-namespaces --context kind-chaos-eater-cluster --select
```

NAMESPACE	NAME	READY	STATUS	REST
-----------	------	-------	--------	------

default	pod/example-deployment-7658c5b56-gfpr9	1/1	Running	0
default	pod/example-deployment-7658c5b56-kx6cr	1/1	Running	0
default	pod/example-deployment-7658c5b56-rjxfq	1/1	Running	0

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	ClusterIP	10.96.223.211	<none>

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILABLE
default	deployment.apps/example-deployment	3/3	3	3

NAMESPACE	NAME	DESIRED	CURRENT
default	replicaset.apps/example-deployment-6db6cbcf84	0	0
default	replicaset.apps/example-deployment-7658c5b56	3	3

## The fault-scope adjustment Completed!

Current fault injection settings:

- Injected Faults #0: `NetworkChaos`
  - Workflow Name: `fault-networkchaos`
  - Grace Period: `0s`
  - Duration: `15s`

Parameters `{{"action": "delay", "direction": "to", "mode": "all", "selector": {"namespaces": ["default"], "labelSelectors": {"app": "example"}}}, {"delay": {"latency": "100ms", "jitter": "10ms"}}`

The previous K8s manifests targeted a single Pod with the label 'app: example' for the NetworkChaos fault injection. In the current manifests, the Pod has been replaced by a Deployment with 3 replicas, but the label 'app: example' remains the same. Therefore, the scope of the fault injection should still target all Pods with the label 'app: example' in the 'default' namespace. The change from a single Pod to a Deployment with multiple replicas does not require a change in the label selector, as the label remains consistent across all replicas. Thus, the scope of the fault injection remains appropriate for the current setup.

Next fault injection scope:

```

{
  "namespaces": [
    "default"
  ],
  "labelSelectors": {
    "app": "example"
  }
}

```

```
}
```

Current fault injection settings:

- Injected Faults #0: PodChaos
  - Workflow Name: fault-podchaos
  - Grace Period: 15s
  - Duration: 10s

```
Parameters {"action": "pod-kill", "mode": "one", "selector": {"namespaces": ["default"],  
"labelSelectors": {"app": "example"}}}}
```

The previous K8s manifests targeted a single Pod with the label 'app: example'. The current manifests have changed to a Deployment with 3 replicas, but the label 'app: example' remains the same. Therefore, the scope of the fault injection should still target Pods with the label 'app: example' in the 'default' namespace. However, since there are now multiple replicas, the 'mode' of the PodChaos should be adjusted to 'one' to ensure that only one Pod is affected at a time, maintaining the original goal of the chaos experiment.

Next fault injection scope:

```
{  
  "namespaces" : [  
    0 : "default"  
  ]  
  "labelSelectors" : {  
    "app" : "example"  
  }  
}
```

Adjusted unittest

The previous unit test targets a Pod resource, specifically checking the running state of a Pod named 'example-pod'. In the reconfigured manifests, the Pod has been replaced by a Deployment named 'example-deployment' with 3 replicas. This change means that the unit test should now target the Deployment instead of a single Pod. The test should verify that all Pods created by the Deployment are running, rather than just one Pod. Therefore, the unit test requires modification to account for these changes. The test should iterate over all Pods created by the Deployment and ensure that each is running at least 90% of the time during the specified duration.

```
import os  
import time  
import argparse  
from kubernetes import client, config
```

```

from unittest_base import K8sAPIBase

class TestDeploymentRunningState(K8sAPIBase):
    def __init__(self, namespace, deployment_name, duration):
        super().__init__()
        self.namespace = namespace
        self.deployment_name = deployment_name
        self.duration = duration

    def test_deployment_running_state(self):
        running_count = 0

        # Loop for the specified duration, checking the Pods' status every 1 second
        for _ in range(self.duration):
            try:
                # List all Pods with the label app=example
                pods = self.v1.list_namespaced_pod(namespace=self.namespace, label_selector='app=example')
                all_running = all(pod.status.phase == 'Running' for pod in pods.items)
                if all_running:
                    running_count += 1
                print(f"Deployment {self.deployment_name} Pods status: {[pod.status.phase for pod in pods.items]}")
            except client.exceptions.ApiException as e:
                print(f"Exception when calling CoreV1Api->list_namespaced_pod: {e}")
            time.sleep(1)

        # Calculate the percentage of time all Pods were running
        running_percentage = (running_count / self.duration) * 100
        print(f"Deployment {self.deployment_name} Pods were all running {running_percentage}% of the time")

        # Assert that all Pods were running at least 90% of the time
        assert running_percentage >= 90, f"Deployment {self.deployment_name} Pods were not running 90% of the time"

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Test the running status of a deployment')
    parser.add_argument('--duration', type=int, default=5, help='Duration to run the test in seconds')
    args = parser.parse_args()

    # Create an instance of the test class and run the test
    test = TestDeploymentRunningState(namespace='default', deployment_name='example', duration=args.duration)
    test.test_deployment_running_state()

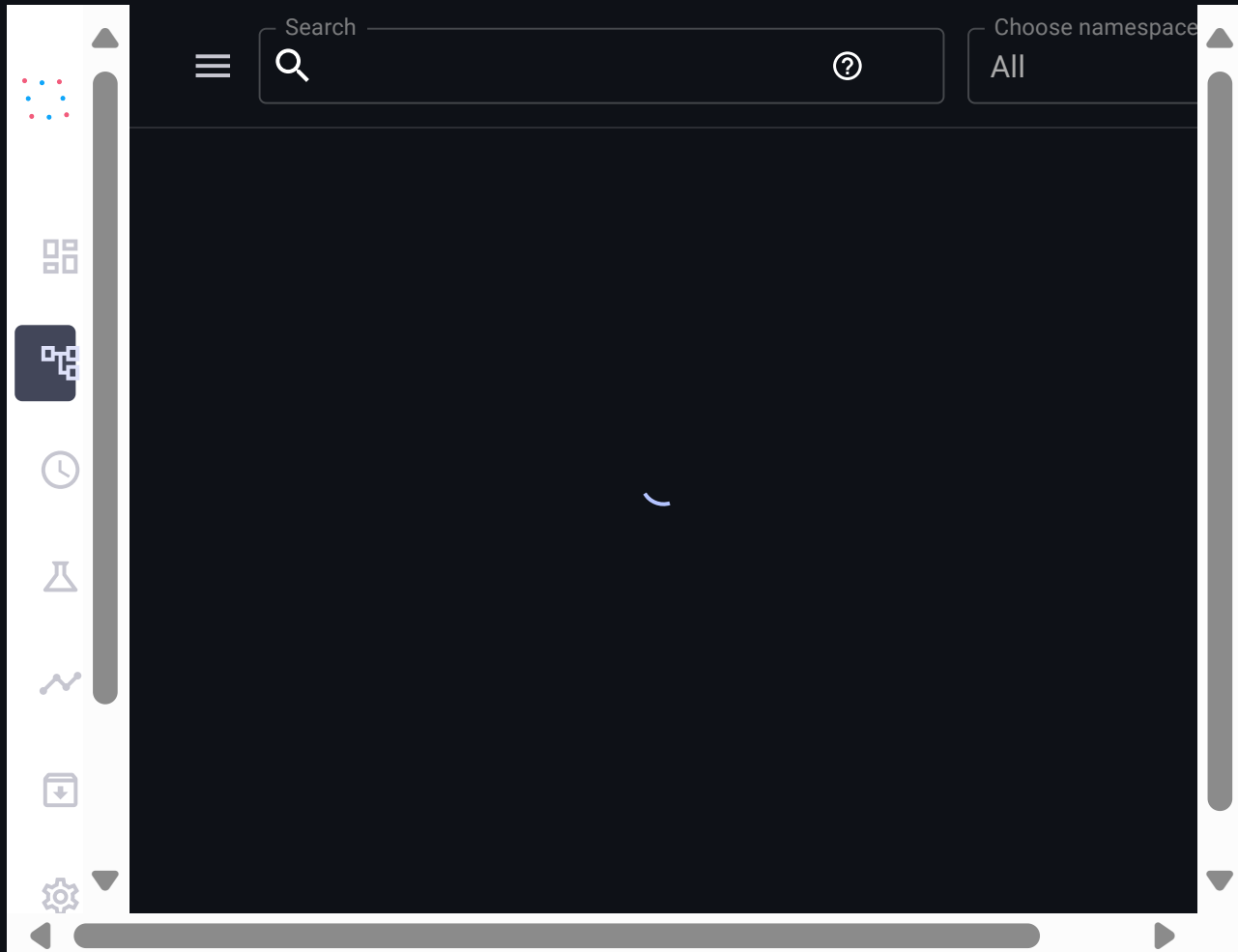
```

## Adjusted unittest

The target of the unit test is the HTTP service provided by the Kubernetes Service resource, which remains unchanged in the reconfigured manifests. The Service still points to the same application, and the endpoint URL used in the test remains valid. However, the underlying Pod

has been changed to a Deployment with 3 replicas, which introduces redundancy. This change is intended to ensure higher availability and reliability of the service. Since the unit test is focused on the service's HTTP response and not directly on the Pod or Deployment, the test itself does not require modification. The threshold for the success rate of HTTP 200 responses remains appropriate, as the redundancy should help maintain this threshold. Therefore, no changes are needed for the unit test.

## Completed the chaos experiment!



Your k8s yaml already has good resilience!!!

## Phase EX: Postprocessing

### Summary of your k8s yaml ^

The Chaos Engineering cycle described involves a systematic approach to testing the resilience of a Kubernetes-based system running an Nginx web server. The cycle begins with understanding the user inputs, which include Kubernetes manifests for a Pod and a Service. The Pod manifest specifies a single Nginx container with a restart policy of 'Never', while the Service manifest exposes this Pod on port 80. Two primary resiliency issues are identified:

the Pod's restart policy and the single Pod deployment, both of which pose risks of downtime and single points of failure.

The hypothesis for the experiment is that the system's steady states will be maintained even when faults are injected. Two steady states are defined: the Pod should be running at least 90% of the time, and the Service should successfully route HTTP requests with a 95% success rate. These are tested using Python scripts and K6 JavaScript, respectively.

The fault scenario simulates a cyber attack using Chaos Mesh, introducing network delays and Pod failures to test the system's resilience. The experiment is divided into three phases: pre-validation, fault-injection, and post-validation, each with specific tasks and time allocations to ensure the system's behavior is thoroughly assessed within a 1-minute timeframe.

In the first experiment attempt, several tests fail, particularly those related to the Pod's availability, due to the 'Never' restart policy and the single Pod deployment. The analysis highlights the need for redundancy and automatic recovery mechanisms, recommending changes to the Pod's restart policy and the use of a Deployment with multiple replicas.

After implementing these improvements, the second experiment attempt shows all tests passing, indicating that the system can maintain its steady states under fault conditions. The changes made include replacing the Pod manifest with a Deployment manifest, setting the restart policy to 'Always', and increasing the number of replicas to ensure redundancy and resilience.

[Download output \(.zip\)](#)