



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-20241124-132854" deleted  
$ kubectl delete workflownode --all --context kind-chaos-eater-cluster -n cl  
workflownode.chaos-mesh.org "fault-injection-overlapped-workflows-t79jj" de  
workflownode.chaos-mesh.org "fault-injection-parallel-workflow-mcz9h" delet  
workflownode.chaos-mesh.org "fault-injection-parallel-workflows-glml5" dele  
workflownode.chaos-mesh.org "fault-injection-phase-fzfw" deleted  
workflownode.chaos-mesh.org "fault-injection-suspend-jtwv9" deleted  
workflownode.chaos-mesh.org "fault-injection-suspend-workflow-84fx" delete  
workflownode.chaos-mesh.org "fault-networkchaos-kzjt" deleted  
workflownode.chaos-mesh.org "fault-podchaos-bvh5n" deleted  
workflownode.chaos-mesh.org "fault-unittest-example-pod-running-76ksg" dele  
workflownode.chaos-mesh.org "fault-unittest-example-service-availability-6l  
workflownode.chaos-mesh.org "post-unittest-example-pod-running-fk8nf" delet  
workflownode.chaos-mesh.org "post-unittest-example-service-availability-g5c  
workflownode.chaos-mesh.org "post-validation-overlapped-workflows-lv7b6" de  
workflownode.chaos-mesh.org "post-validation-phase-prdxr" deleted  
workflownode.chaos-mesh.org "post-validation-suspend-skdtg" deleted  
workflownode.chaos-mesh.org "post-validation-suspend-workflow-9br7r" delete  
workflownode.chaos-mesh.org "post-validation-suspend-workflow2-d69bb" dele  
workflownode.chaos-mesh.org "post-validation-suspend2-9r6xg" deleted  
workflownode.chaos-mesh.org "pre-unittest-example-pod-running-l7cw2" delete  
workflownode.chaos-mesh.org "pre-unittest-example-service-availability-9w96l  
workflownode.chaos-mesh.org "pre-validation-overlapped-workflows-8k75r" del  
workflownode.chaos-mesh.org "pre-validation-phase-9t6dx" deleted  
workflownode.chaos-mesh.org "pre-validation-suspend-bvsm2" deleted
```

Input instructions for your Chaos Engineering



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

```
$ kubectl delete all --all-namespaces --context kind-chaos-eater-cluster -l app=example
pod "example-deployment-598977f874-5d28v" deleted
pod "example-deployment-598977f874-gxppt" deleted
pod "example-deployment-598977f874-hkx5m" 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 80ns
- pod/example-pod created
- service/example-service created
Waiting for deployments to stabilize...
- pods is ready.
Deployments stabilized in 3.093 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	6s

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	ClusterIP	10.96.226.86	<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 a selector to target pods with the label 'app: example'.
- The Service listens on port 80 using the TCP protocol.
- Traffic received on port 80 is forwarded to the target 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: Deploying a single Pod without a controller like Deployment or ReplicaSet can lead to lack of redundancy and no automatic recovery from failures.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: kind: Pod

Application of the manifests:

Thoughts:

The given manifests define a Kubernetes Pod and a Service. The Pod is running an NGINX container, which is a popular web server and reverse proxy. The Service is set up to expose this Pod on port 80, which is the default port for HTTP traffic. The use of NGINX and the exposure of port 80 strongly suggest that the application is intended to serve web content or act as a web server. The file names and the use of NGINX further reinforce this assumption.

Assumed application:

The service offers a web server application, likely serving static web content or acting as a reverse proxy for other services.

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: example-pod-running**



Description

The first steady state to define should be related to the Pod defined in 'nginx/pod.yaml', as it has the most immediate and critical issues: the Pod's restart policy is set to 'Never', and it is a single Pod deployment without redundancy. These issues can lead to downtime if the Pod fails. Therefore, a steady state should be defined to ensure that the Pod is running and available. A measurable output for this steady state is the number of running Pods, which should be exactly 1, as there is only one Pod defined in the manifest.

Current state inspection

To inspect the 'example-pod-running' state, 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 ensure that the Pod is running and available, which can be verified by checking the Pod's status. The K8s API provides the necessary tools to query the current state of the Pod and confirm that it is in the 'Running' state. The script will run for a maximum duration of 5 seconds, checking the Pod's status every second.

tool: `k8s` duration: `5s`

Inspection script (Python) `k8s_example-pod-running.py` :

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

def check_pod_status(namespace, pod_name):
    # 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()
    try:
        pod = v1.read_namespaced_pod(name=pod_name, namespace=namespace)
        return pod.status.phase
    except client.exceptions.ApiException as e:
        print(f"Exception when calling CoreV1Api->read_namespaced_pod")
        return None

def main(duration):
    namespace = 'default'
    pod_name = 'example-pod'
```

```

running_count = 0
for _ in range(duration):
    status = check_pod_status(namespace, pod_name)
    if status == 'Running':
        running_count += 1
    time.sleep(1)
print(f"Pod '{pod_name}' running status checked {duration} times")

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='Check if the pod is running')
    parser.add_argument('--duration', type=int, default=5, help='Duration in seconds')
    args = parser.parse_args()
    main(args.duration)

```

Pod `k8sexample-pod-running-pod` has completed successfully.

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

```
Pod 'example-pod' running status checked 5 times, running count: 5
```

► Threshold

The steady state we are considering is the 'example-pod' being in the 'Running' state. The current state shows that the pod was checked 5 times over a duration of 5 seconds, and it was running each time, resulting in a running count of 5. This indicates that the pod is consistently running without any downtime during the check period. To define a reasonable threshold, we need to account for potential minor fluctuations or delays in pod status updates. Therefore, a threshold that allows for a small tolerance would be appropriate. Given the current state, a threshold where the pod is running at least 80% of the time during the check period would be reasonable. This means that out of 5 checks, the pod should be running at least 4 times to satisfy the threshold, which aligns with the current state where it was running all 5 times.

The pod should be in the 'Running' state at least 80% of the time during the check period.

📄 Unit test to validate the steady state

To add the threshold assertion to the inspection Python script, I will implement a unit test that checks if the pod is in the 'Running' state at least 80% of the time during the specified duration. This will be done by calculating the ratio of the running count to the

total checks and asserting that this ratio is greater than or equal to 0.8. The test will use a for-loop to check the pod status every second for the specified duration, and the assertion will be made at the end of the loop.

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

class TestPodSteadyState(K8sAPIBase):
    def __init__(self):
        super().__init__()

    def check_pod_status(self, namespace, pod_name):
        try:
            pod = self.v1.read_namespaced_pod(name=pod_name, namespace=namespace)
            return pod.status.phase
        except client.exceptions.ApiException as e:
            print(f"Exception when calling CoreV1Api->read_namespaced_pod_status: {e}")
            return None

    def test_pod_running_state(self, duration):
        namespace = 'default'
        pod_name = 'example-pod'
        running_count = 0

        # Check the pod status every second for the specified duration
        for _ in range(duration):
            status = self.check_pod_status(namespace, pod_name)
            if status == 'Running':
                running_count += 1
            time.sleep(1)

        # Calculate the running ratio
        running_ratio = running_count / duration

        # Assert that the pod is running at least 80% of the time
        assert running_ratio >= 0.8, f"Pod '{pod_name}' was not running at least 80% of the time"

def main():
    parser = argparse.ArgumentParser(description='Test if the pod is running at least 80% of the time')
    parser.add_argument('--duration', type=int, default=5, help='Duration in seconds')
    args = parser.parse_args()

    test = TestPodSteadyState()
```

```
test.test_pod_running_state(args.duration)
```

```
if __name__ == '__main__':  
    main()
```

Pod `unittestexample-pod-runningmod0-pod` has completed successfully.

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

✓ Steady state #2: example-service-http-response-200

Description

The next steady state to consider should be related to the Service defined in 'nginx/service.yaml'. The Service is crucial for ensuring that the web server is accessible to clients. A potential issue with the Service could be related to its ability to route traffic correctly to the Pod. Therefore, a steady state should be defined to ensure that the Service is correctly routing traffic to the Pod. A measurable output for this steady state could be the Service's ability to respond to HTTP requests successfully. This can be measured by checking the HTTP response code, which should be 200 (OK) for successful requests.

Current state inspection

To inspect the 'example-service-http-response-200' state, we need to verify that the service is correctly routing traffic to the pod and responding with HTTP 200 status codes. Since this involves checking the communication status and response codes, k6 is the appropriate tool for this task. We will use k6 to send HTTP requests to the service and check if the responses are successful (HTTP 200). The test will run for a short duration with a few virtual users to simulate client requests.

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

Inspection script (Javascript) `k6_example-service-http-response-200.js` :

```
import http from 'k6/http';  
import { check } from 'k6';  
  
export const options = {  
    vus: 5,  
    duration: 5s,  
};
```



```

    duration: '5s',
  });

export default function () {
  const res = http.get('http://example-service.default.svc.cluster.1
  check(res, {
    'is status 200': (r) => r.status === 200,
  });
}

```

Pod `k6example-service-http-response-200-pod` has completed successfully.

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

```
✓ is status 200
```

```

checks.....: 100.00% 71072 out of 71072
data_received.....: 60 MB    12 MB/s
data_sent.....: 7.8 MB    1.6 MB/s
http_req_blocked.....: avg=7.8µs    min=1.09µs    med=3.77µs
http_req_connecting.....: avg=1.92µs    min=0s        med=0s
http_req_duration.....: avg=228.03µs min=75.71µs    med=202.1µs
  { expected_response:true }...: avg=228.03µs min=75.71µs    med=202.1µs
http_req_failed.....: 0.00%    0 out of 71072
http_req_receiving.....: avg=53.35µs    min=11.74µs    med=42.23µs
http_req_sending.....: avg=15.45µs    min=3.45µs     med=11.52µs
http_req_tls_handshaking.....: avg=0s        min=0s        med=0s
http_req_waiting.....: avg=159.22µs   min=50.6µs     med=144.2µs
http_reqs.....: 71072    14212.721341/s
iteration_duration.....: avg=336.69µs   min=125.71µs   med=292.6µs
iterations.....: 71072    14212.721341/s
vus.....: 5        min=5        max=5
vus_max.....: 5        min=5        max=5

```

► Threshold

The steady state we are considering is the ability of the 'example-service' to respond with HTTP status code 200, indicating successful handling of requests. The k6 test results show that 100% of the requests received a status 200 response, which is the desired outcome. To account for potential fluctuations in a real-world scenario, we should set a threshold that allows for a small percentage of requests to fail while still considering the system to be in a steady state. A reasonable threshold would be to

expect at least 95% of the requests to return a status 200, allowing for some minor network issues or transient errors.

At least 95% of HTTP requests should return a status code 200.

Unit test to validate the steady state

To ensure that the steady state satisfies the threshold, I will add a threshold option to the k6 script. This threshold will check that at least 95% of the HTTP requests return a status code of 200. This is done by adding a 'thresholds' property to the 'options' object, specifying the condition that the 'http_req_failed' metric should be less than 5% (i.e., 100% - 95%).

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

export const options = {
  vus: 5,
  duration: '5s',
  thresholds: {
    // Adding a threshold to ensure that at least 95% of requests re
    'http_req_failed': ['rate<0.05'], // Less than 5% of requests sh
  },
};

export default function () {
  const res = http.get('http://example-service.default.svc.cluster.1
  check(res, {
    'is status 200': (r) => r.status === 200,
  });
}
```

Pod `unittestexample-service-http-response-200mod0-pod` has completed successfully.

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

✓ is status 200

```
checks.....: 100.00% 70050 out of 70050
data_received.....: 60 MB    12 MB/s
data_sent.....: 7.7 MB    1.5 MB/s
http_req_blocked.....: avg=8.27µs    min=1.18µs    med=3.84
http_req_connecting.....: avg=2.01µs    min=0s        med=0s
```

```
http_req_duration.....: avg=228.67µs min=75.17µs med=201.
  { expected_response:true }...: avg=228.67µs min=75.17µs med=201.
✓ http_req_failed.....: 0.00%    0 out of 70050
http_req_receiving.....: avg=55.34µs  min=11.07µs med=43.2
http_req_sending.....: avg=15.76µs  min=3.49µs  med=11.9
http_req_tls_handshaking.....: avg=0s      min=0s      med=0s
http_req_waiting.....: avg=157.56µs min=46.31µs med=142.
http_reqs.....: 70050    14007.859899/s
iteration_duration.....: avg=341.12µs min=123.1µs med=294.
iterations.....: 70050    14007.859899/s
vus.....: 5          min=5          max=5
vus_max.....: 5          min=5          max=5
```

Fault definition

✓ Scenario: Cyber Attack

🗨 Description

Given the system's weaknesses, a cyber attack targeting the web server could be highly impactful. The system's main vulnerabilities are the single Pod deployment without redundancy and the Pod's restart policy set to 'Never'. These issues can lead to downtime if the Pod fails, affecting the steady state of the system. To simulate a cyber attack, we can inject faults that exploit these weaknesses. First, we can use PodChaos to simulate a Pod failure, which will test the system's ability to maintain the 'example-pod-running' steady state. Since the Pod won't restart automatically, this will likely cause downtime. Next, we can use NetworkChaos to simulate network latency, which will test the 'example-service-http-response-200' steady state by affecting the Service's ability to route traffic correctly. This sequence simulates a cyber attack by first taking down the Pod and then degrading the network performance, which could happen during a coordinated attack.

🧨 Fault-injection sequence

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

⚙ Detailed fault parameters

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

```

▼ {
  "action" : "pod-kill"
  "mode" : "one"
  ▼ "selector" : {
    ▼ "namespaces" : [
      | 0 : "default"
    ]
    ▼ "labelSelectors" : {
      | "app" : "example"
    }
  }
  "value" : "1"
}

```

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

```

▼ {
  "action" : "delay"
  "direction" : "to"
  ▼ "target" : {
    "mode" : "all"
    ▼ "selector" : {
      ▼ "namespaces" : [
        | 0 : "default"
      ]
      ▼ "labelSelectors" : {
        | "app" : "example"
      }
    }
  }
  "mode" : "all"
  ▼ "selector" : {
    ▼ "namespaces" : [
      | 0 : "default"
    ]
    ▼ "labelSelectors" : {
      | "app" : "example"
    }
  }
  "device" : "eth0"
}

```

```
    "delay" : {  
      "latency" : "100ms"  
      "jitter" : "10ms"  
      "correlation" : "50"  
    }  
  }  
}
```

Phase 2: Chaos Experiment

CE experiment Planning Completed!

Chaos Engineering Experiment Plan

Time Schedule

Given the constraints and the nature of the system, we need to design a Chaos Engineering experiment that fits within a 1-minute timeframe. The experiment is divided into three phases: pre-validation, fault-injection, and post-validation. The pre-validation phase ensures that the system is in a steady state before we introduce any faults. The fault-injection phase is where we introduce the faults to observe the system's behavior under stress. Finally, the post-validation phase checks if the system returns to its steady state after the faults are removed.

Considering the time constraint, we need to allocate time efficiently across these phases. The pre-validation phase should be short but sufficient to confirm the system's steady state. Given the steady states defined, we can allocate 10 seconds for pre-validation to check both the pod's running state and the service's HTTP response. The fault-injection phase is the core of the experiment, where we simulate the cyber attack by introducing PodChaos and NetworkChaos. We should allocate the majority of the time here, around 40 seconds, to observe the system's behavior under these conditions. Finally, the post-validation phase should be long enough to ensure the system returns to its steady state, so we allocate 10 seconds for this phase. This allocation ensures that we have a balanced approach to validating the system's resilience within the 1-minute constraint.

Total experiment time: 1m

Pre-validation Phase: 10s

Fault-injection Phase: 40s

Post-validation Phase: 10s

Pre-validation Phase (10s)

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 perform unit tests to verify each of them. The first steady state, 'example-pod-running', requires checking that the pod is running at least 80% of the time. The second steady state, 'example-service-http-response-200', requires verifying that at least 95% of HTTP requests return a status code 200. Since the total time for the pre-validation phase is 10 seconds, we will allocate 5 seconds for each unit test. These tests will be executed sequentially to ensure that each steady state is independently verified before moving to the fault injection phase. This approach ensures that we have a clear baseline of the system's behavior under normal conditions, which is crucial for comparing against the system's behavior during and after fault injection.

- Verified Steady State #0: `example-pod-running`
 - Workflow Name: `pre-unittest-example-pod-running`
 - Grace Period: `0s`
 - Duration: `5s`
- Verified Steady State #1: `example-service-http-response-200`
 - Workflow Name: `pre-unittest-example-service-http-response-200`
 - Grace Period: `5s`
 - Duration: `5s`

Fault-injection Phase (40s)

In this fault-injection phase, we aim to simulate a cyber attack by injecting two types of faults: PodChaos and NetworkChaos. The total duration for this phase is 40 seconds, so we need to carefully schedule the faults and unit tests to fit within this timeframe.

First, we will inject the PodChaos fault to simulate a Pod failure. This will test the system's ability to maintain the 'example-pod-running' steady state. Since the Pod's restart policy is set to 'Never', this fault will likely cause downtime, directly impacting the first steady state. We will start this fault at the beginning of the phase (grace period of 0s) and let it run for 20 seconds.

Simultaneously, we will run the unit test for the 'example-pod-running' steady state to observe the impact of the PodChaos fault. This test will also start at 0s and run for 20 seconds, allowing us to verify if the Pod remains in the 'Running' state at least 80% of the time during the fault.

Next, we will inject the NetworkChaos fault to simulate network latency. This will test the 'example-service-http-response-200' steady state by affecting the Service's ability to route traffic correctly. We will start this fault at 20 seconds (grace period of 20s) and let it run for the remaining 20 seconds of the phase.

Simultaneously, we will run the unit test for the 'example-service-http-response-200' steady state to observe the impact of the NetworkChaos fault. This test will also start at 20 seconds and run for 20 seconds, allowing us to verify if the Service can still respond with HTTP status code 200 for at least 95% of the requests during the fault.

By staggering the faults and unit tests in this manner, we can effectively observe the system's behavior under each fault condition and ensure that the entire fault-injection phase is completed within the 40-second limit.

- Verified Steady State #0: `example-pod-running`
 - Workflow Name: `fault-unittest-example-pod-running`
 - Grace Period: `0s`
 - Duration: `20s`
- Verified Steady State #1: `example-service-http-response-200`
 - Workflow Name: `fault-unittest-example-service-http-response-200`
 - Grace Period: `20s`
 - Duration: `20s`
- Injected Faults #0: `PodChaos`
 - Workflow Name: `fault-podchaos`
 - Grace Period: `0s`
 - Duration: `20s`
- Injected Faults #1: `NetworkChaos`
 - Workflow Name: `fault-networkchaos`
 - Grace Period: `20s`
 - Duration: `20s`

Post-validation Phase (10s)

In the post-validation phase, we need to ensure that the system has returned to its steady states after the fault injection. Given the 10-second time constraint, we will perform quick checks to verify each steady state. The first steady state, 'example-pod-running', requires the pod to be in the 'Running' state at least 80% of the time. We will check the pod status every second for 5 seconds, allowing a 1-second grace period for the system to stabilize after the fault injection. The second steady state, 'example-

service-http-response-200', requires at least 95% of HTTP requests to return a status code 200. We will perform a quick HTTP request test for 5 seconds, also allowing a 1-second grace period. These tests will be executed sequentially to ensure that each steady state is verified independently, and the system has enough time to recover from the faults.

- Verified Steady State #0: `example-pod-running`
 - Workflow Name: `post-unittest-example-pod-running`
 - Grace Period: `1s`
 - Duration: `5s`
- Verified Steady State #1: `example-service-http-response-200`
 - Workflow Name: `post-unittest-example-service-http-response-200`
 - Grace Period: `1s`
 - Duration: `5s`

Summary

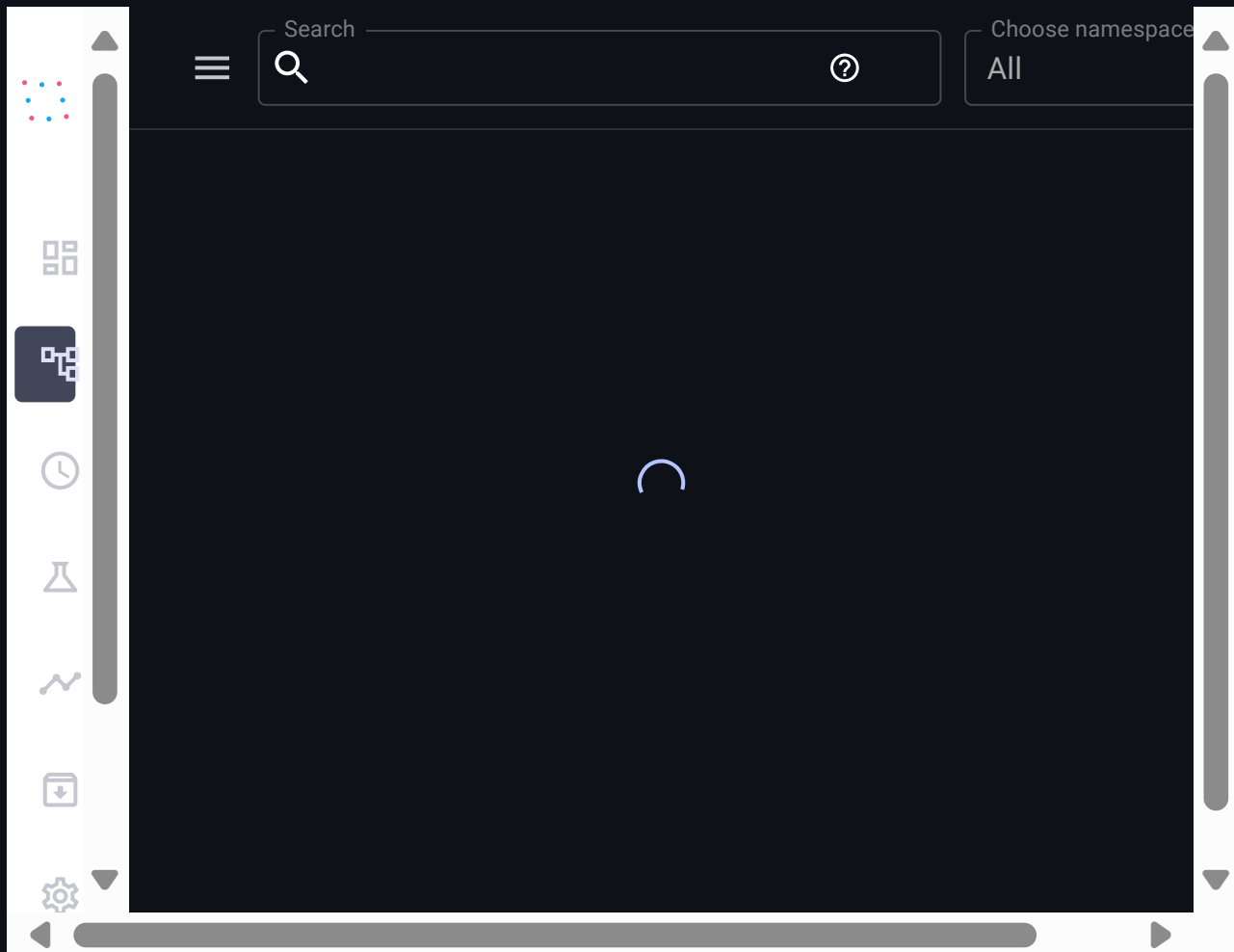
The Chaos Engineering experiment is structured into three phases: pre-validation, fault-injection, and post-validation, all within a 1-minute timeframe.

In the pre-validation phase, which lasts for 10 seconds, two unit tests are conducted sequentially to ensure the system is in a steady state. The first test, 'example-pod-running', is executed immediately at the start of the phase and runs for 5 seconds. This test checks if the pod is running at least 80% of the time. Following this, the second test, 'example-service-http-response-200', begins at the 5-second mark and also runs for 5 seconds, verifying that at least 95% of HTTP requests return a status code 200.

The fault-injection phase spans 40 seconds and involves two types of fault injections: PodChaos and NetworkChaos. The PodChaos fault is injected at the start of the phase and runs for 20 seconds, simulating a Pod failure. Concurrently, a unit test for 'example-pod-running' is conducted to assess the impact of this fault. At the 20-second mark, the NetworkChaos fault is introduced, running for the remaining 20 seconds to simulate network latency. Simultaneously, a unit test for 'example-service-http-response-200' is performed to evaluate the service's response under network stress.

Finally, the post-validation phase, lasting 10 seconds, verifies the system's return to steady states. The 'example-pod-running' test starts after a 1-second grace period and runs for 5 seconds, checking the pod's status. Subsequently, the 'example-service-http-response-200' test begins after another 1-second grace period and runs for 5 seconds, ensuring the service's HTTP response rate is restored. These tests are executed sequentially to confirm the system's recovery from the injected faults.

Completed the chaos experiment!



Phase 3: Analysis

The Chaos Engineering experiment results indicate several critical issues in the system's resilience, particularly related to the single Pod deployment and its restart policy. Here's a detailed analysis of the failures observed during the experiment:

1. Fault-Unittest-Example-Pod-Running Failure:

- **Cause:** The failure of the 'example-pod-running' test during the fault-injection phase is directly linked to the PodChaos fault, which simulates a Pod failure. The Pod's restart policy is set to 'Never', as specified in the `nginx/pod.yaml` manifest. This configuration means that once the Pod is killed, it does not automatically restart, leading to a 404 error when attempting to read the Pod's status. The running ratio of 0.0 confirms that the Pod was not running at all during the test period.
- **Insight:** The lack of a restart mechanism for the Pod is a significant vulnerability. In a real-world scenario, any failure of the Pod would result in downtime, as there is no redundancy or automatic recovery.
- **Recommendation:** To improve resilience, consider using a Deployment or ReplicaSet instead of a standalone Pod. This would provide redundancy and ensure that Pods are

automatically restarted in case of failure.

2. Fault-Unittest-Example-Service-Http-Response-200 Failure:

- **Cause:** The failure of the 'example-service-http-response-200' test during the fault-injection phase is due to the Pod being unavailable, as indicated by the 'connection refused' errors. Since the Service relies on the Pod to handle HTTP requests, the unavailability of the Pod results in the Service being unable to fulfill requests, leading to a failure in meeting the 95% success threshold.
- **Insight:** The Service's dependency on a single Pod without redundancy means that any Pod failure directly impacts the Service's ability to function, leading to a complete service outage.
- **Recommendation:** Implementing a Deployment with multiple replicas would ensure that the Service can continue to route traffic to available Pods, even if one Pod fails.

3. Post-Unittest Failures:

- **Cause:** The failures in the post-validation phase for both 'example-pod-running' and 'example-service-http-response-200' tests indicate that the system did not recover from the injected faults. The Pod remained unavailable, and consequently, the Service continued to fail in handling requests.
- **Insight:** The system's inability to recover post-fault injection highlights the critical need for automatic recovery mechanisms and redundancy.
- **Recommendation:** In addition to using Deployments for redundancy, consider implementing health checks and readiness probes to ensure that Pods are healthy and ready to serve traffic. This would help in automatically replacing unhealthy Pods and maintaining service availability.

Overall, the experiment underscores the importance of addressing the identified weaknesses in the system's configuration, particularly the single Pod deployment and its restart policy. By adopting Kubernetes best practices such as using Deployments and ensuring redundancy, the system's resilience to faults can be significantly improved.

Phase 4: Improvement

Reconfiguration



To improve the resiliency of the system, I will replace the standalone Pod with a Deployment. This will allow for multiple replicas of the Pod, providing redundancy and automatic recovery in case of failure. The Deployment will manage the Pods, ensuring that the desired number of replicas are always running. This change addresses both the Pod restart policy issue and the single Pod deployment issue.

Modification_type: replace

File name: nginx/pod.yaml

Replace the standalone Pod with a Deployment to provide redundancy and automatic recovery. This change addresses the issues of the Pod's restart policy being set to 'Never' and the lack of redundancy in a single Pod deployment.

```
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:
      containers:
        - name: example-container
          image: nginx:1.17.1
          ports:
            - containerPort: 80
```

Deploying reconfigured 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 551ns
- deployment.apps/example-deployment configured
- service/example-service configured
Waiting for deployments to stabilize...
- deployment/example-deployment is ready.
```

Deployments stabilized in 4.093 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=app=example
```

NAMESPACE	NAME	READY	STATUS	REPLICAS
default	pod/example-deployment-54c4586f89-5zh9t	1/1	Running	0
default	pod/example-deployment-54c4586f89-99xld	1/1	Running	0
default	pod/example-deployment-54c4586f89-tw6gc	1/1	Running	0

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	ClusterIP	10.96.61.229	<none>

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

The fault-scope adjustment Completed!

Current fault injection settings:

- Injected Faults #0: `PodChaos`
 - Workflow Name: `fault-podchaos`
 - Grace Period: `0s`
 - Duration: `20s`

Parameters `{{"action": "pod-kill", "mode": "one", "selector": {"namespaces": ["default"], "labelSelectors": {"app": "example"}}}, {"value": "1"}}`

The previous K8s manifest targeted a single pod with the label 'app: example'. In the current K8s manifest, the system has been updated to a Deployment with 3 replicas, all having the same label 'app: example'. The original chaos experiment aimed to simulate a pod failure, which is still relevant. However, since the system now has 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 experiment while adapting to the new deployment structure.

Next fault injection scope:

```
{
  "namespaces" : [
    0 : "default"
  ]
  "labelSelectors" : {
```

```

    "app" : "example"
  }
}

```

Current fault injection settings:

- Injected Faults #0: `NetworkChaos`
 - Workflow Name: `fault-networkchaos`
 - Grace Period: `20s`
 - Duration: `20s`

Parameters `{{"action": "delay", "direction": "to", "target": {"mode": "all", "selector": {"namespaces": ["default"], "labelSelectors": {"app": "example"}}}}, {"mode": "all", "selector": {"namespaces": ["default"], "labelSelectors": {"app": "example"}}, {"device": "eth0", "delay": {"latency": "100ms", "jitter": "10ms", "correlation": "50"}}}}`

The previous K8s manifests had a single Pod, while the current manifests have been updated to a Deployment with 3 replicas. The original fault injection targeted all Pods with the label 'app: example' in the 'default' namespace. Since the label and namespace remain unchanged, the scope of the fault injection should remain the same to target all Pods created by the Deployment. This ensures that the experiment's original goal of testing the system's resilience under network chaos conditions is maintained, even with the increased number of Pods.

Next fault injection scope:

```

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

```

Adjusted unittest

The previous unit test targets a Pod resource 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 at least one of the Pods managed by the Deployment is running at least 80% of the time, as the redundancy is intended to maintain the steady state. Therefore, the unit test requires modification to check the status of Pods created by the Deployment rather than a single Pod. The test should iterate over all Pods with the label 'app: example' and ensure that at least one Pod is running at least 80% of the time.

```

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

class TestDeploymentSteadyState(K8sAPIBase):
    def __init__(self):
        super().__init__()

    def check_pods_status(self, namespace, label_selector):
        try:
            pods = self.v1.list_namespaced_pod(namespace=namespace, label_selector=label_selector)
            return [pod.status.phase for pod in pods.items]
        except client.exceptions.ApiException as e:
            print(f"Exception when calling CoreV1Api->list_namespaced_pod: {e}")
            return []

    def test_deployment_running_state(self, duration):
        namespace = 'default'
        label_selector = 'app=example'
        running_count = 0

        # Check the pod status every second for the specified duration
        for _ in range(duration):
            statuses = self.check_pods_status(namespace, label_selector)
            if any(status == 'Running' for status in statuses):
                running_count += 1
            time.sleep(1)

        # Calculate the running ratio
        running_ratio = running_count / duration

        # Assert that at least one pod is running at least 80% of the time
        assert running_ratio >= 0.8, f"Deployment 'example-deployment' did not reach 80% running ratio"

    def main():
        parser = argparse.ArgumentParser(description='Test if the deployment has reached steady state')
        parser.add_argument('--duration', type=int, default=5, help='Duration to test in seconds')
        args = parser.parse_args()

        test = TestDeploymentSteadyState()
        test.test_deployment_running_state(args.duration)

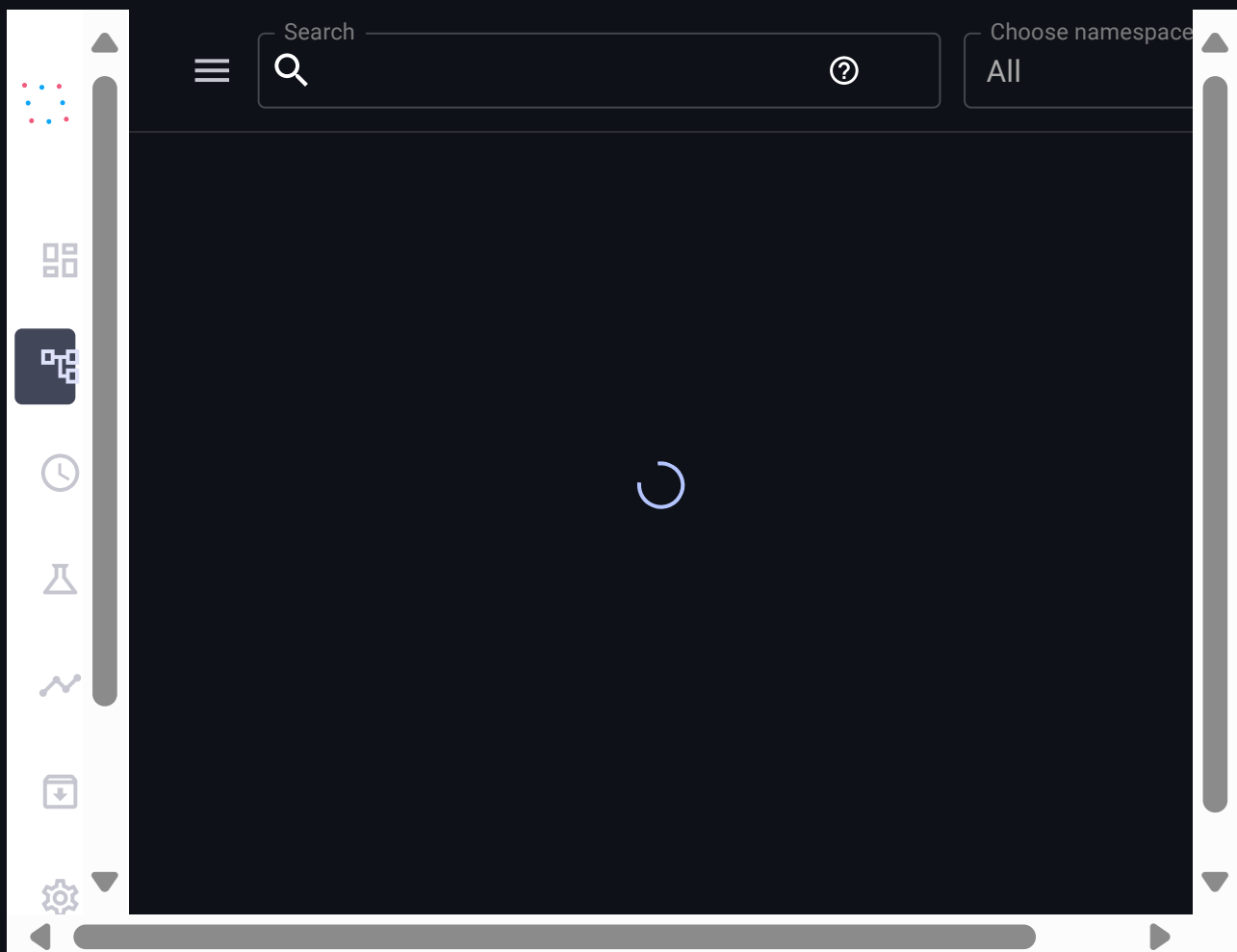
```

```
if __name__ == '__main__':  
    main()
```

Adjusted unittest

The target of the unit test is the service, which is accessed via an HTTP request to verify that at least 95% of requests return a status code 200. In the reconfigured manifests, the Pod has been changed to a Deployment with 3 replicas, but the Service configuration remains unchanged. The unit test is designed to check the availability and response of the service, not the specific configuration of the pods or deployments behind it. Since the Service manifest has not changed, the unit test does not require any modification. The redundancy added by changing the Pod to a Deployment with 3 replicas is intended to ensure the service remains available, which aligns with the existing threshold in the unit test. Therefore, no changes are needed in the unit test code.

Completed the chaos experiment!



Your k8s yaml already has good resilience!!!

Phase EX: Postprocessing

Summary of your k8s yaml



The Chaos Engineering cycle involves a detailed process to test the resilience of a Kubernetes-based system, specifically focusing on a web server application using NGINX. The cycle begins with understanding the user inputs, which include Kubernetes manifests for a Pod and a Service. The Pod manifest defines a single Pod with a 'Never' restart policy, which poses a risk of downtime if the Pod fails. The Service manifest exposes the Pod on port 80, but relies on the Pod's availability to function correctly.

The identified resiliency issues include the Pod's restart policy and the lack of redundancy due to a single Pod deployment. The hypothesis for the experiment is that the system should maintain its steady states even when faults are injected. Two steady states are defined: the Pod should be running at least 80% of the time, and the Service should respond with HTTP status code 200 for at least 95% of requests.

The experiment is structured into three phases: pre-validation, fault-injection, and post-validation, all within a 1-minute timeframe. The pre-validation phase checks the system's steady states before fault injection. The fault-injection phase simulates a cyber attack using Chaos Mesh, introducing PodChaos to kill the Pod and NetworkChaos to delay network traffic. The post-validation phase verifies if the system returns to its steady states after the faults are removed.

The first experiment attempt revealed critical issues: the Pod did not restart after being killed, and the Service failed to handle requests due to the Pod's unavailability. These failures highlighted the need for redundancy and automatic recovery mechanisms. The recommendation was to replace the standalone Pod with a Deployment, providing multiple replicas for redundancy and automatic recovery.

After implementing the recommended changes, the second experiment attempt was successful, with all unit tests passing. The Deployment ensured that the system maintained its steady states even during fault injection, demonstrating improved resilience. This cycle underscores the importance of using Kubernetes best practices, such as Deployments, to enhance system reliability and fault tolerance.

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