



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

```
$ kubectl delete all --all-namespaces --context kind-chaos-eater-cluster -l app=nginx
pod "example-deployment-6765f47cb6-ktsg7" deleted
pod "example-deployment-6765f47cb6-pqnbj" deleted
pod "example-deployment-6765f47cb6-xbqng" deleted
service "example-service" deleted
deployment.apps "example-deployment" deleted
```

K8s manifest(s) to be deployed:

`nginx/pod.yaml`

apiVersion: v1

Input instructions for your Chaos Engineering



```
  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 201ns
- pod/example-pod created
- service/example-service created
Waiting for deployments to stabilize...
- pods is ready.
Deployments stabilized in 3.086 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	10.96.152.112	<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, which is a specific version of the Nginx web server.
- The container exposes port 80, which is commonly used for HTTP 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 Service forwards traffic 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: Having a single Pod means there is no redundancy. If the Pod fails, the service will be unavailable.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: Only one Pod is defined without a Deployment or ReplicaSet.

Application of the manifests:

Thoughts:

The manifests provided define a Kubernetes Pod and a Service. The Pod runs an Nginx container, which is a popular web server used to serve static content, reverse proxy, or load balance HTTP traffic. The Service 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 HTTP content.

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 issue to address is the Pod's restart policy set to 'Never'. This is a critical issue because if the Pod fails, it will not restart, leading to potential downtime. 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, which should be 1, as there is only one Pod defined in the manifest. This steady state will help verify that the Pod is up and running, which is crucial given the restart policy configuration.

Current state inspection

To inspect whether the 'example-pod' is running, I will use the K8s API. This is because the state to be checked is the status of a Kubernetes resource, specifically the Pod, which is best done using the Kubernetes client libraries. The script will check the status of the Pod every second for a duration of 5 seconds, which is within the allowed maximum duration. This will ensure that the Pod is running and available, addressing the critical issue of the Pod's restart policy being set to 'Never'.

tool: `k8s` duration: `5s`

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

```

import os
import time
from kubernetes import client, config

# 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()

def check_pod_status(namespace, pod_name):
    try:
        pod = v1.read_namespaced_pod(name=pod_name, namespace=namespace)
        return pod.status.phase == 'Running'
    except client.exceptions.ApiException as e:
        print(f"Exception when calling CoreV1Api->read_namespaced_pod_status: {e}")
        return False

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

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='Check if a 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 whether the 'example-pod' is running. 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 during the check period. Given the constraints of the chaos engineering experiment, which must be completed within 1 minute, we can set a threshold that allows for some tolerance in case of brief fluctuations. A reasonable threshold would be that the pod should be running at least 90% of the time during the check period. This allows for a small margin of error while still ensuring that the pod is generally available and running.

The pod should be running at least 90% 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 calculate the percentage of time the pod is running during the specified duration. The threshold is set to 90%, meaning the pod should be running at least 90% of the time during the check period. I will implement a for-loop to check the pod status every second for the specified duration and then assert that the running percentage is greater than or equal to 90%. This ensures that the pod is generally available and running, allowing for a small margin of error.

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

class TestPodRunningState(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 == 'Running'
        except client.exceptions.ApiException as e:
            print(f"Exception when calling CoreV1Api->read_namespaced_pod_status: {e}")
            return False

    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):
    if self.check_pod_status(namespace, pod_name):
        running_count += 1
    time.sleep(1)

# Calculate the running percentage
running_percentage = (running_count / duration) * 100

# Assert that the running percentage is at least 90%
assert running_percentage >= 90, f"Pod '{pod_name}' running percentage is {running_percentage}%"

print(f"Pod '{pod_name}' running status checked {duration} times. Running percentage is {running_percentage}%")

def main():
    parser = argparse.ArgumentParser(description='Test if a pod is running')
    parser.add_argument('--duration', type=int, default=5, help='Duration in seconds')
    args = parser.parse_args()

    test = TestPodRunningState()
    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:

```
Pod 'example-pod' running status checked 5 times. Running percentage is 100%
```

✓ Steady state #2: example-service-availability

🗨 Description

The next issue to address is the 'Single Pod Deployment', which is related to the lack of redundancy. This is a significant issue because if the single Pod fails, the service will be

unavailable. To verify this, we can define a steady state that checks the availability of the service itself. A measurable output for this steady state is the service's response time or availability. Since the service is exposed on port 80, we can check if the service is responding to HTTP requests. This steady state will help verify that the service is available and responsive, which is crucial given the single Pod deployment configuration.

Current state inspection

To inspect the availability of the 'example-service', we need to check if the service is responding to HTTP requests. Since the service is exposed on port 80, we can use k6 to simulate HTTP requests to the service and measure its response time and availability. This approach allows us to verify the service's responsiveness and availability, which is crucial given the single Pod deployment configuration. We will use a small number of virtual users and a short duration to fit within the 1-minute constraint of the Chaos Engineering experiment.

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

Inspection script (Javascript) `k6_example-service-availability.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.1');
  check(res, {
    'status is 200': (r) => r.status === 200,
  });
}
```

Pod `k6example-service-availability-pod` has completed sucessfully.

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

```
✓ status is 200

checks.....: 100.00% 13915 out of 13915
data_received.....: 12 MB    2.4 MB/s
data_sent.....: 1.5 MB    306 kB/s
```



```

http_req_blocked.....: avg=8.11µs    min=1.4µs    med=4.19µs
http_req_connecting.....: avg=2.23µs    min=0s       med=0s
http_req_duration.....: avg=236.57µs  min=90.31µs  med=232.6µs
  { expected_response:true }...: avg=236.57µs  min=90.31µs  med=232.6µs
http_req_failed.....: 0.00%    0 out of 13915
http_req_receiving.....: avg=67.9µs    min=17.89µs  med=61.35µs
http_req_sending.....: avg=18.51µs   min=4.66µs   med=14.77µs
http_req_tls_handshaking.....: avg=0s        min=0s        med=0s
http_req_waiting.....: avg=150.16µs  min=54.98µs  med=151.2µs
http_reqs.....: 13915    2782.666155/s
iteration_duration.....: avg=344.61µs  min=131.12µs med=333.1µs
iterations.....: 13915    2782.666155/s
vus.....: 1      min=1      max=1
vus_max.....: 1      min=1      max=1

```

🚩 Threshold

The steady state we are considering is the availability of the 'example-service'. The k6 test results show that the service is currently responding with a 200 status code for all requests, indicating 100% availability. Given that the system consists of a single Pod, any failure in the Pod would result in the service being unavailable. Therefore, the threshold should ensure that the service remains available and responsive. To account for minor fluctuations and network latency, a reasonable threshold would be to maintain a high availability percentage, slightly below 100% to allow for brief, non-critical failures. A threshold of 99.9% availability is a common standard for web services, allowing for some tolerance while still ensuring high reliability.

Service availability should be at least 99.9% with a response status of 200.

📄 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 service's availability should be at least 99.9% with a response status of 200. This is done by setting a threshold on the 'http_req_failed' metric, ensuring that the failure rate is less than or equal to 0.1% (which corresponds to 99.9% availability).

```

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

export const options = {
  vus: 1,

```

```

duration: '5s',
thresholds: {
  // Ensure that the service availability is at least 99.9%
  'http_req_failed': ['rate<=0.001'], // 0.1% failure rate corresponds to 99.9% availability
},
};

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

```

Pod `unittestexample-service-availabilitymod0-pod` has completed successfully.

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

✓ status is 200

```

checks.....: 100.00% 14565 out of 14565
data_received.....: 12 MB    2.5 MB/s
data_sent.....: 1.6 MB    320 kB/s
http_req_blocked.....: avg=7.92µs   min=1.33µs   med=4.1µs
http_req_connecting.....: avg=2.19µs   min=0s       med=0s
http_req_duration.....: avg=225.1µs  min=82.21µs  med=226µs
  { expected_response:true }...: avg=225.1µs  min=82.21µs  med=226µs
✓ http_req_failed.....: 0.00%    0 out of 14565
http_req_receiving.....: avg=66.97µs  min=15.24µs  med=60.µs
http_req_sending.....: avg=17.27µs  min=4.03µs   med=13.µs
http_req_tls_handshaking.....: avg=0s       min=0s       med=0s
http_req_waiting.....: avg=140.85µs min=53.55µs  med=144µs
http_reqs.....: 14565    2912.709122/s
iteration_duration.....: avg=329.52µs min=117.97µs med=323µs
iterations.....: 14565    2912.709122/s
vus.....: 1      min=1      max=1
vus_max.....: 1      min=1      max=1

```

Fault definition

✓ Scenario: Cyber Attack



Description

Given the system's weaknesses, a cyber attack targeting the single Pod and its network could be highly impactful. The Pod's restart policy set to 'Never' means that if the Pod fails, it will not restart, leading to downtime. Additionally, the single Pod deployment means there is no redundancy, so any failure will make the service unavailable. To simulate a cyber attack, we can start by injecting a PodChaos fault to kill the Pod, testing the system's ability to handle Pod failures. This will directly exploit the lack of redundancy and the restart policy issue. Next, we can simulate a network attack using NetworkChaos to introduce network latency, testing the service's ability to maintain availability under network stress. This sequence simulates a cyber attack by first taking down the Pod and then stressing the network, revealing the system's vulnerabilities in handling such scenarios.

Fault-injection sequence

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

Detailed fault parameters

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

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

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

```
▼ {
  "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 of the chaos engineering experiment, which must 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 a steady state before we introduce any faults. Since we have

two steady states to validate, we should allocate a reasonable amount of time to check both the pod's running status and the service's availability. A duration of 15 seconds should be sufficient for pre-validation, allowing us to run the necessary checks multiple times. The fault-injection phase is where we introduce the chaos to observe the system's behavior under stress. Given the complexity of the faults (PodChaos and NetworkChaos), we should allocate the majority of the time to this phase to ensure that the faults have enough time to manifest and impact the system. A duration of 30 seconds is appropriate for fault injection, allowing us to observe the system's response to both pod failure and network latency. Finally, the post-validation phase is essential to verify that the system returns to its steady states after the faults are removed. We should allocate 15 seconds for post-validation, similar to the pre-validation phase, to ensure that the system stabilizes and meets the defined thresholds for steady states. This allocation results in a total experiment time of 60 seconds, which fits within the 1-minute constraint.

Total experiment time: 60s

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 state before we proceed with fault injection. Given the constraints of a 15-second total time for this phase, we will conduct two unit tests to verify the steady states: one for the pod's running status and another for the service's availability. These tests will be executed sequentially due to the short duration available, ensuring that each steady state is verified independently and thoroughly. The first test will check if the 'example-pod' is running at least 90% of the time over a 5-second period. This is crucial because the pod's restart policy is set to 'Never', and we need to confirm its availability before introducing any faults. The second test will verify the 'example-service' availability, ensuring it responds with a 200 status code at least 99.9% of the time over another 5-second period. This test is essential to confirm that the service is operational and responsive, given the single pod deployment. By staggering these tests, we can focus on each steady state individually, allowing us to identify any issues before proceeding to the fault injection phase.

- 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-availability
 - Workflow Name: pre-unittest-example-service-availability
 - Grace Period: 5s
 - Duration: 5s

Fault-injection Phase (30s)

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 30 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 fault will be injected at the start of the phase (grace period of 0s) and will last for 10 seconds. This duration is chosen to allow enough time for the system to experience the impact of the pod being killed, given the pod's restart policy is set to 'Never'.

Simultaneously, we will run the unit test for the 'example-pod-running' steady state to verify if the pod is running at least 90% of the time during the fault injection. This test will also start at 0s and run for 10 seconds, aligning with the PodChaos duration.

Next, we will inject the NetworkChaos fault to simulate network latency. This fault will start at 10 seconds (after the PodChaos fault ends) and will last for 20 seconds. This staggered approach allows us to observe the system's behavior under network stress after the pod failure has been simulated.

During the NetworkChaos fault, we will run the unit test for the 'example-service-availability' steady state. This test will start at 10 seconds and run for 20 seconds, matching the NetworkChaos duration. This ensures we are checking the service's availability and response time while the network is under stress.

By staggering the faults and aligning the unit tests with the fault durations, we can effectively observe the system's behavior under each fault condition and verify if the steady states are maintained.

- Verified Steady State #0: example-pod-running
 - Workflow Name: fault-unittest-example-pod-running
 - Grace Period: 0s
 - Duration: 10s
- Verified Steady State #1: example-service-availability
 - Workflow Name: fault-unittest-example-service-availability
 - Grace Period: 10s

- Duration: 20s
- Injected Faults #0: PodChaos
 - Workflow Name: fault-podchaos
 - Grace Period: 0s
 - Duration: 10s
- Injected Faults #1: NetworkChaos
 - Workflow Name: fault-networkchaos
 - Grace Period: 10s
 - Duration: 20s

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 perform quick checks to verify the steady states. The two steady states to verify are: 1) the 'example-pod' is running, and 2) the 'example-service' is available. We will execute these checks sequentially due to the short duration, ensuring each test has enough time to gather meaningful data. The first test will check the pod's running status, followed by the service availability test. This order is logical because the pod must be running for the service to be available. 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 checks.

- Verified Steady State #0: example-pod-running
 - Workflow Name: post-unittest-example-pod-running
 - Grace Period: 2s
 - Duration: 6s
- Verified Steady State #1: example-service-availability
 - Workflow Name: post-unittest-example-service-availability
 - Grace Period: 8s
 - Duration: 5s

Summary

The chaos engineering experiment is structured into three phases: pre-validation, fault-injection, and post-validation, all to be completed within a total of 60 seconds.

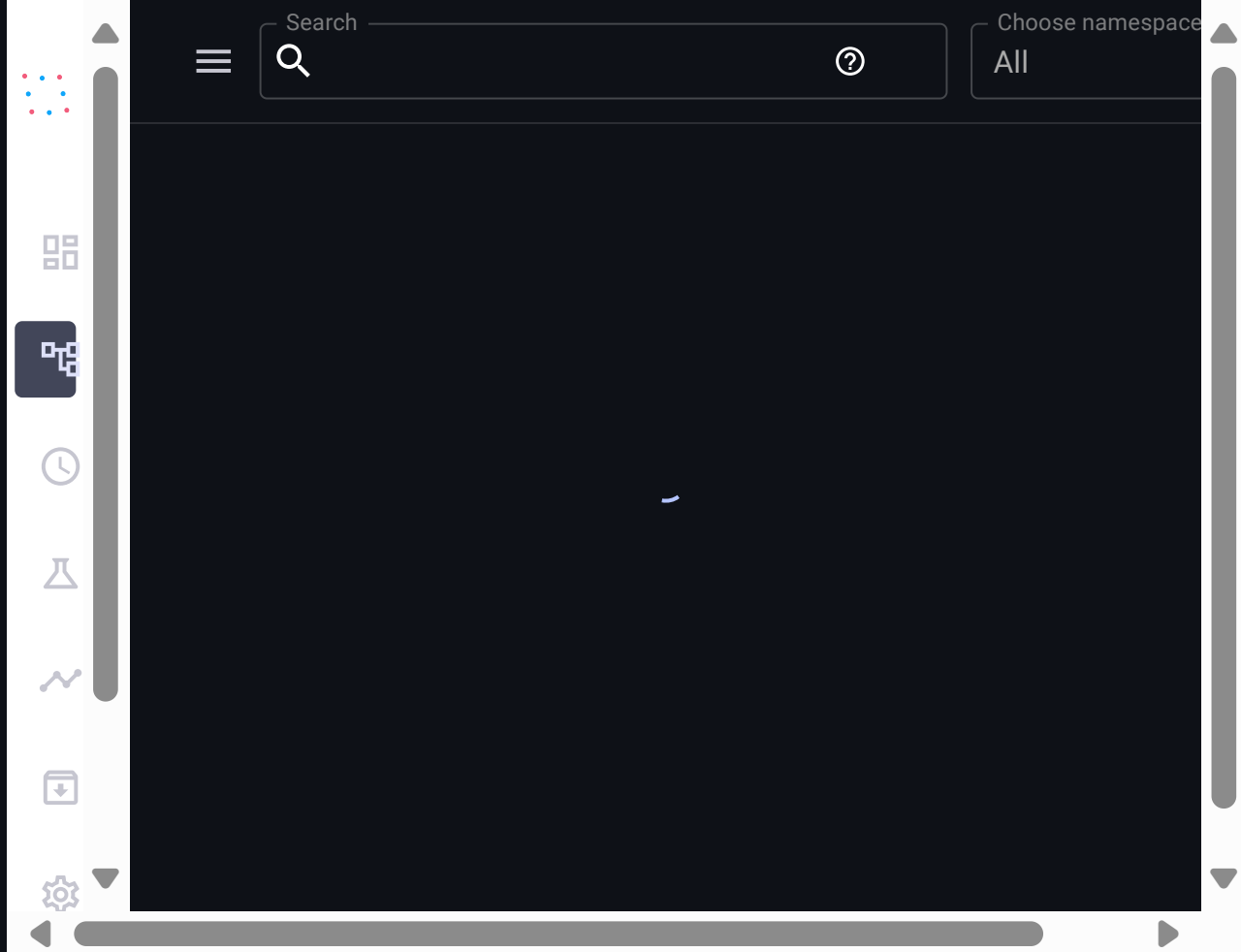
In the pre-validation phase, which lasts for 15 seconds, two unit tests are conducted sequentially to ensure the system is in a steady state before fault injection. The first test,

named 'pre-unittest-example-pod-running', checks the 'example-pod' running status. It starts immediately at the beginning of the phase and runs for 5 seconds. Following this, the second test, 'pre-unittest-example-service-availability', begins at the 5-second mark and also runs for 5 seconds, verifying the service's availability.

The fault-injection phase spans 30 seconds and involves two types of faults: PodChaos and NetworkChaos. Initially, the PodChaos fault, named 'fault-podchaos', is injected at the start of the phase and lasts for 10 seconds. Concurrently, the 'fault-unittest-example-pod-running' unit test runs for the same duration to verify the pod's status during the fault. After the PodChaos fault concludes, the NetworkChaos fault, named 'fault-networkchaos', begins at the 10-second mark and continues for 20 seconds. Simultaneously, the 'fault-unittest-example-service-availability' test runs for 20 seconds, starting at the same time as the NetworkChaos fault, to check the service's availability under network stress.

Finally, the post-validation phase, also 15 seconds long, ensures the system returns to its steady states. The 'post-unittest-example-pod-running' test starts after a 2-second grace period and runs for 6 seconds to verify the pod's status. Subsequently, the 'post-unittest-example-service-availability' test begins at the 8-second mark and runs for 5 seconds, checking the service's availability. This sequential execution allows for a brief stabilization period before each test.

Completed the chaos experiment!



Phase 3: Analysis

The chaos engineering experiment results indicate several critical issues in the system's configuration and its ability to handle faults, particularly in the context of the defined fault scenario. Here is a detailed analysis of the failures observed during the experiment:

1. Pod Restart Policy and Single Pod Deployment:

- The 'fault-unittest-example-pod-running' test failed because the Pod was not found after the PodChaos fault was injected. This is directly related to the Pod's restart policy set to 'Never' in the `nginx/pod.yaml` manifest. When the Pod was killed, it did not restart, leading to a 404 error when attempting to read the Pod's status. This confirms the identified issue #0 (Pod Restart Policy) and issue #1 (Single Pod Deployment), where the lack of redundancy and automatic recovery mechanisms resulted in the Pod being unavailable.

2. Service Availability:

- The 'fault-unittest-example-service-availability' test failed due to connection refusals, indicating that the service was unavailable. This failure is a consequence of the Pod being killed and not restarting, as the Service relies on the Pod to handle requests. The Service's selector in `nginx/service.yaml` targets Pods with the label 'app: example', and

with the Pod being down, there were no endpoints available to serve the requests, leading to the connection refused errors.

3. Post-Validation Failures:

- Both 'post-unittest-example-pod-running' and 'post-unittest-example-service-availability' tests failed, showing that the system did not recover to its steady state after the fault injection. The Pod remained unavailable, and consequently, the Service could not respond to HTTP requests. This further highlights the critical impact of the Pod's restart policy and the lack of redundancy.

Insights and Recommendations:

- **Pod Restart Policy:** Change the Pod's restart policy from 'Never' to 'Always' or 'OnFailure' to ensure that the Pod automatically restarts if it fails. This will help maintain the Pod's availability and reduce downtime.
- **Redundancy and Scalability:** Implement a Deployment or ReplicaSet to manage the Pod. This will provide redundancy by running multiple replicas of the Pod, ensuring that the Service remains available even if one Pod fails.
- **Service Resilience:** Consider using a LoadBalancer or Ingress to manage traffic to multiple Pods, enhancing the system's ability to handle failures and maintain service availability.

By addressing these issues, the system can improve its resilience against similar fault scenarios in the future, ensuring that the steady states are maintained even when faults are injected.

Phase 4: Improvement

Reconfiguration

To improve the resiliency of the system, I will replace the Pod with a Deployment to manage the Pod's lifecycle and provide redundancy. This will address both the restart policy issue and the single Pod deployment issue by ensuring that multiple replicas of the Pod are running and can automatically restart if they fail.

Modification_type: replace

File name: nginx/pod.yaml

Replace the Pod with a Deployment to manage the Pod's lifecycle and provide redundancy. This will ensure that multiple replicas of the Pod are running and can automatically restart 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:
      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 200ns
- deployment.apps/example-deployment configured
- service/example-service configured
Waiting for deployments to stabilize...
- deployment/example-deployment is ready.
Deployments stabilized in 3.098 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-598977f874-gxppt	1/1	Running	0
default	pod/example-deployment-598977f874-hkx5m	1/1	Running	0
default	pod/example-deployment-598977f874-jxg2b	1/1	Running	0

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	ClusterIP	10.96.202.254	<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-598977f874	3	3
default	replicaset.apps/example-deployment-9779766dc	0	0

The fault-scope adjustment Completed!

Current fault injection settings:

- Injected Faults #0: `PodChaos`
 - Workflow Name: `fault-podchaos`
 - Grace Period: `0s`
 - 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, all having the same label 'app: example'. The original chaos experiment aimed to simulate a pod failure, which can still be achieved by targeting one of the pods in the Deployment. Therefore, the scope of the fault injection should remain the same, targeting pods with the label 'app: example' in the 'default' namespace. This ensures that the experiment's goal of simulating a pod failure is maintained, even with the change from a single pod to a Deployment with multiple replicas.

Next fault injection scope:

```

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

```

Current fault injection settings:

- Injected Faults #0: `NetworkChaos`

- Workflow Name: `fault-networkchaos`
- Grace Period: `10s`
- 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, and the NetworkChaos was targeting all pods with the label 'app: example' in the 'default' namespace. The current manifests have been updated to a Deployment with 3 replicas, but the label 'app: example' and the namespace 'default' remain the same. Therefore, the scope of the fault injection should remain unchanged, as it will now target all pods created by the Deployment, which aligns with the original goal of the chaos experiment to test the system's behavior under network stress.

Next fault injection scope:

```
{
  "namespaces": [
    "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 created by the Deployment is running at least 90% of the time. Therefore, the unit test requires modification to account for the change from a Pod to a Deployment with multiple replicas. The test should be updated to check the status of the Pods created by the Deployment, ensuring that the steady state is maintained by the entire redundancy.

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

```

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

    def check_deployment_pods_status(self, namespace, deployment_name):
        try:
            pods = self.v1.list_namespaced_pod(namespace=namespace, label_selector=deployment_name)
            running_pods = [pod for pod in pods if pod.status.phase == 'Running']
            return len(running_pods) > 0
        except client.exceptions.ApiException as e:
            print(f"Exception when calling CoreV1Api->list_namespaced_pod: {e}")
            return False

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

        # Check the deployment pods status every second for the specified duration
        for _ in range(duration):
            if self.check_deployment_pods_status(namespace, deployment_name):
                running_count += 1
            time.sleep(1)

        # Calculate the running percentage
        running_percentage = (running_count / duration) * 100

        # Assert that the running percentage is at least 90%
        assert running_percentage >= 90, f"Deployment '{deployment_name}' running percentage is {running_percentage}%"

        print(f"Deployment '{deployment_name}' running status checked {duration} seconds")

    def main():
        parser = argparse.ArgumentParser(description='Test if a deployment has running pods')
        parser.add_argument('--duration', type=int, default=5, help='Duration to check the deployment status')
        args = parser.parse_args()

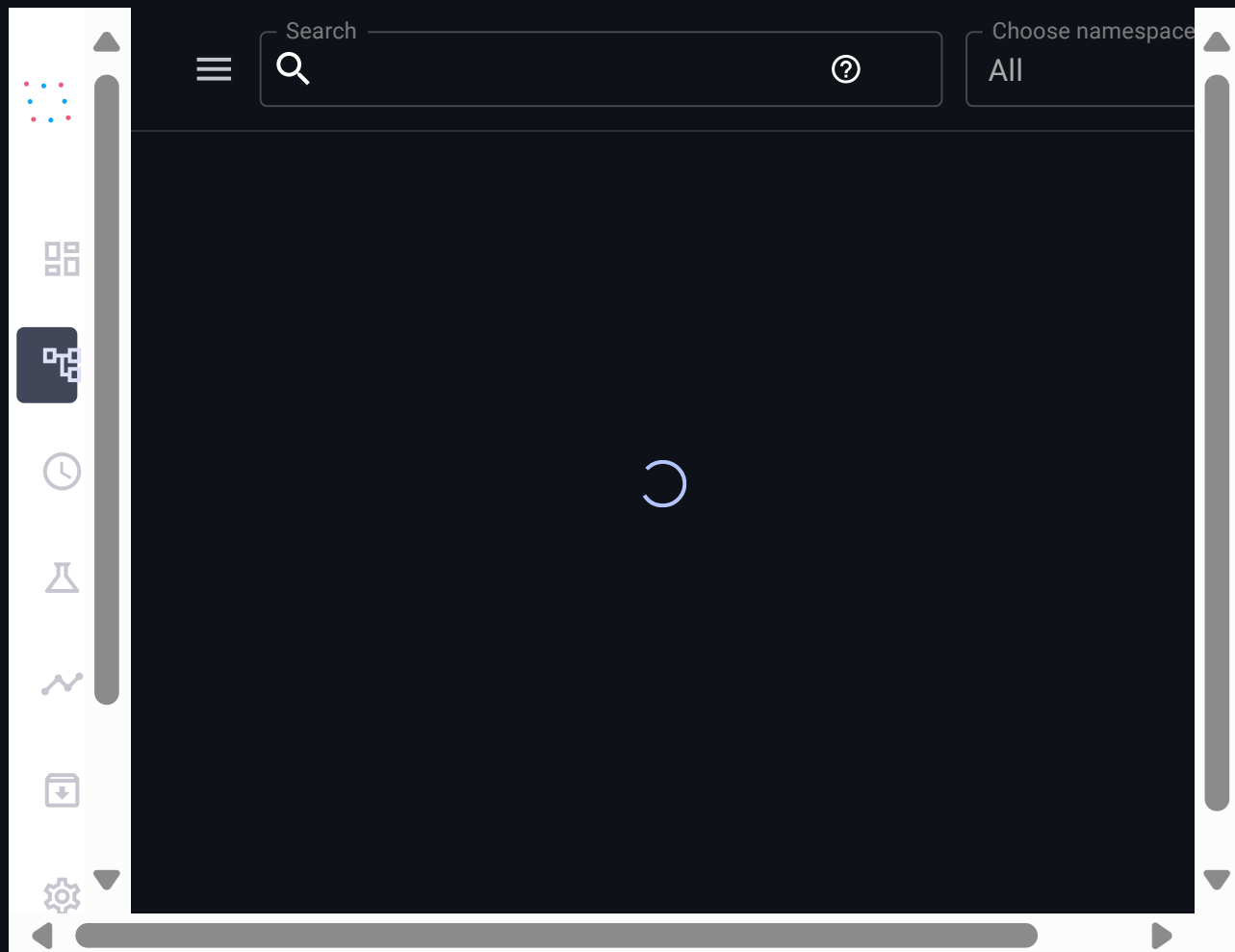
        test = TestDeploymentRunningState()
        test.test_deployment_running_state(args.duration)

if __name__ == '__main__':
    main()

```

The target of the unit test is the service availability, which is associated with the `Service` resource in the Kubernetes manifests. In the previous and reconfigured manifests, the `Service` resource has not changed, so the endpoint and the service configuration remain the same. The unit test checks the availability of the service by sending HTTP requests to the service endpoint and verifying the response status. Since the `Service` resource has not been modified, the unit test does not require any changes. The reconfiguration of the `Pod` to a `Deployment` with 3 replicas is intended to improve the system's ability to meet the availability threshold, but it does not affect the unit test itself, which remains valid for checking the service's availability.

Completed the chaos experiment!



Your k8s yaml already has good resilience!!!

Phase EX: Postprocessing

Summary of your k8s yaml

The Chaos Engineering cycle begins with understanding the user inputs, which include Kubernetes manifests for a Pod and a Service. The Pod manifest defines a single Nginx container with a restart policy of 'Never', and the Service manifest exposes this Pod on port

80. Two main resiliency issues are identified: 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'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 have 99.9% availability with a response status of 200.

The fault scenario simulates a cyber attack using Chaos Mesh, injecting PodChaos to kill the Pod and NetworkChaos to introduce network latency. The experiment is divided into three phases: pre-validation, fault-injection, and post-validation, each with specific tasks and durations to verify the system's behavior under stress.

In the first experiment attempt, the system fails to maintain the steady states during and after the fault injection. The Pod does not restart due to its 'Never' restart policy, and the Service becomes unavailable, confirming the identified issues. Recommendations include changing the Pod's restart policy and implementing a Deployment for redundancy.

After modifying the system by replacing the Pod with a Deployment, the second experiment attempt is successful. All unit tests pass, indicating that the system maintains its steady states even when faults are injected, demonstrating improved resilience.

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