



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-20241124-124141" deleted  
$ kubectl delete workflownode --all --context kind-chaos-eater-cluster -n cl  
workflownode.chaos-mesh.org "fault-injection-overlapped-workflows-8qbx" de  
workflownode.chaos-mesh.org "fault-injection-parallel-workflow-dpgwq" dele  
workflownode.chaos-mesh.org "fault-injection-parallel-workflows-77zbn" dele  
workflownode.chaos-mesh.org "fault-injection-phase-rwmdt" deleted  
workflownode.chaos-mesh.org "fault-injection-suspend-l7mwf" deleted  
workflownode.chaos-mesh.org "fault-injection-suspend-workflow-dft2q" delete  
workflownode.chaos-mesh.org "fault-networkchaos-bl7j7" deleted  
workflownode.chaos-mesh.org "fault-podchaos-b7tqw" deleted  
workflownode.chaos-mesh.org "fault-unittest-podrunningstatusnginxpodyaml-kq  
workflownode.chaos-mesh.org "fault-unittest-serviceavailabilitynginxservice  
workflownode.chaos-mesh.org "post-unittest-podrunningstatusnginxpodyaml-s8f  
workflownode.chaos-mesh.org "post-unittest-serviceavailabilitynginxservice  
workflownode.chaos-mesh.org "post-validation-parallel-workflows-ft7cq" dele  
workflownode.chaos-mesh.org "post-validation-phase-rbxm7" deleted  
workflownode.chaos-mesh.org "pre-unittest-podrunningstatusnginxpodyaml-pnd9  
workflownode.chaos-mesh.org "pre-unittest-serviceavailabilitynginxserviceyar  
workflownode.chaos-mesh.org "pre-validation-parallel-workflows-x5w7f" dele  
workflownode.chaos-mesh.org "pre-validation-phase-nxfrr" deleted  
workflownode.chaos-mesh.org "the-entry-84pfw" deleted  
$ kubectl delete deployments --all --context kind-chaos-eater-cluster -n cha  
No resources found  
$ kubectl delete pods --all --context kind-chaos-eater-cluster -n chaos-eat  
No resources found
```

Input instructions for your Chaos Engineering



```
$ kubectl delete all --all-namespaces --context kind-chaos-eater-cluster -l
pod "example-deployment-f849656cf-5khq8" deleted
pod "example-deployment-f849656cf-bmc42" deleted
pod "example-deployment-f849656cf-zzqqq" 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 90ns
- pod/example-pod created
- service/example-service created
Waiting for deployments to stabilize...
- pods is ready.
Deployments stabilized in 3.095 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 | RESTARTS | AGE |
|-----------|-----------------|-------|---------|----------|-----|
| default | pod/example-pod | 1/1 | Running | 0 | 6s |

| NAMESPACE | NAME | TYPE | CLUSTER-IP | EXTERNAL-IP |
|-----------|-------------------------|-----------|--------------|-------------|
| default | service/example-service | ClusterIP | 10.96.216.95 | <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 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: Using a single Pod without a controller like Deployment or ReplicaSet can lead to lack of redundancy and no automatic recovery if the Pod fails.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: kind: Pod

Application of the manifests:

Thoughts:

The manifests provided are for a basic setup of an Nginx web server. The pod.yaml file specifies a pod running an Nginx container, and the service.yaml file sets up a service to expose this pod on port 80. The use of the Nginx image and the exposure of port 80, which is the default HTTP port, strongly suggest that this setup is intended to serve web content. The file names and the use of Nginx, a popular web server, further support this assumption.

Assumed application:

Web server hosting static or dynamic web content using Nginx.

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', 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 verify that the Pod is running and has not failed. This can be measured by checking the Pod's

status to ensure it is in the 'Running' state. This steady state is specific to the 'nginx/pod.yaml' manifest, which defines the Pod with the problematic restart policy.

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 verify that the Pod is in the 'Running' state, which is a specific status check that can be efficiently performed using the K8s API. The check will be performed every second for a duration of 5 seconds to ensure the Pod remains in the desired state.

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, 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 status: {pod.status.phase}")
        except client.exceptions.ApiException as e:
            print(f"Exception when calling CoreV1Api->read_namespaced_pod")
        time.sleep(1)
    print(f"Pod was running {running_count} out of {duration} seconds")

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='Check the status of a pod')
    parser.add_argument('--duration', type=int, default=5, help='Duration in seconds')
    args = parser.parse_args()
    check_pod_status(namespace='default', pod_name='example-pod', duration=args.duration)
```

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

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

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

Threshold

The steady state we are considering is that the Pod should be in the 'Running' state. The current state shows that the Pod was running 5 out of 5 seconds, which indicates a 100% uptime during the observation period. Given that the Chaos Engineering experiment must be completed within 1 minute, we can set a threshold that allows for some minor fluctuations but still ensures high availability. A reasonable threshold would be that the Pod should be in the 'Running' state at least 90% of the time during the observation period. This allows for brief interruptions or delays in status updates while still maintaining a high level of reliability.

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 set to 90%, meaning the Pod should be running at least 90% of the time. I will implement a for-loop to check the Pod's status every second for the given duration and then assert that the running percentage meets or exceeds the threshold. This ensures that the Pod's availability is within acceptable limits.

```
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
    # Check the pod status every second for the specified duration
    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 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 running percentage
    running_percentage = (running_count / self.duration) * 100
    print(f"Pod was running {running_count} out of {self.duration} seconds")
    print(f"Running percentage: {running_percentage}%")
    # Assert that the running percentage meets the threshold
    assert running_percentage >= 90, f"Pod running percentage {running_percentage}% is below threshold 90%"

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Test the running state of a pod')
    parser.add_argument('--duration', type=int, default=5, help='Duration in seconds')
    args = parser.parse_args()
    # Create an instance of the test class
    test = TestPodRunningState(namespace='default', pod_name='example-pod', duration=args.duration)
    # Run the test
    test.test_pod_running_state()

```

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

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

```

Pod status: Running
Pod status: Running
Pod status: Running
Pod status: Running
Pod status: Running
Pod was running 5 out of 5 seconds.
Running percentage: 100.0%

```

Description

The next issue to address is the lack of redundancy due to the use of a single Pod without a controller like Deployment or ReplicaSet. This can lead to a lack of automatic recovery if the Pod fails. To verify the system's resilience, we should define a steady state that measures the availability of the service provided by the Pod. Since the Service 'example-service' is responsible for exposing the Pod, we can measure the service's availability by checking the response time or success rate of HTTP requests to the service. This steady state is specific to the 'nginx/service.yaml' manifest, which defines the Service that forwards traffic to the Pod.

Current state inspection

To inspect the availability of the 'example-service', we need to measure the response time or success rate of HTTP requests to the service. Since this involves checking the communication status and metrics, such as request sending and response time, k6 is the appropriate tool for this task. We will use k6 to send HTTP requests to the service and verify its availability by checking the response status. The service is exposed on port 80, and we will use the internal DNS name 'example-service.default.svc.cluster.local:80' to send requests. We will configure k6 to run with 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.local:80');
  check(res, {
    'status is 200': (r) => r.status === 200,
  });
}
```


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

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

```
✓ status is 200
```

```
checks.....: 100.00% 14968 out of 14968
data_received.....: 13 MB    2.5 MB/s
data_sent.....: 1.6 MB    329 kB/s
http_req_blocked.....: avg=7.92µs   min=1.24µs   med=4.11µs
http_req_connecting.....: avg=2.2µs    min=0s        med=0s
http_req_duration.....: avg=216.27µs min=80.03µs   med=218.1µs
  { expected_response:true }...: avg=216.27µs min=80.03µs   med=218.1µs
http_req_failed.....: 0.00%    0 out of 14968
http_req_receiving.....: avg=64.56µs  min=17.24µs  med=58.25µs
http_req_sending.....: avg=17.12µs  min=4.13µs   med=13.36µs
http_req_tls_handshaking.....: avg=0s        min=0s        med=0s
http_req_waiting.....: avg=134.58µs min=53.84µs   med=138.6µs
http_reqs.....: 14968    2993.500375/s
iteration_duration.....: avg=320.27µs min=120.11µs  med=313.4µs
iterations.....: 14968    2993.500375/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', which is measured by the success rate of HTTP requests. The k6 test results show that 100% of the requests returned a status of 200, indicating that the service is fully available. To define a reasonable threshold, we should consider a success rate that accounts for minor fluctuations or transient issues that might occur in a real-world scenario. A success rate of 99% is a common threshold for availability, allowing for some tolerance while still ensuring high availability. This threshold is also satisfied by the current state, which shows a 100% success rate.

HTTP request success rate \geq 99%

📄 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 requests should be at least 99%. This is done by using the 'http_req_failed' metric, which tracks

the percentage of failed requests. The threshold will be set to ensure that the percentage of failed requests is less than or equal to 1%, which corresponds to a success rate of 99% or higher.

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

export const options = {
  vus: 1,
  duration: '5s',
  thresholds: {
    // Define a threshold for the success rate of HTTP requests
    'http_req_failed': ['rate<=0.01'], // This ensures that the fail
  },
};

export default function () {
  const res = http.get('http://example-service.default.svc.cluster.1
  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% 14775 out of 14775
data_received.....: 13 MB    2.5 MB/s
data_sent.....: 1.6 MB    325 kB/s
http_req_blocked.....: avg=7.74µs   min=1.49µs   med=4.6
http_req_connecting.....: avg=2.18µs   min=0s       med=0s
http_req_duration.....: avg=220.16µs min=84.15µs  med=218
  { expected_response:true }...: avg=220.16µs min=84.15µs  med=218
✓ http_req_failed.....: 0.00%    0 out of 14775
http_req_receiving.....: avg=65.69µs   min=19.05µs  med=58.
http_req_sending.....: avg=17.06µs   min=4.12µs   med=13.
http_req_tls_handshaking.....: avg=0s        min=0s       med=0s
http_req_waiting.....: avg=137.4µs   min=55.79µs  med=137
http_reqs.....: 14775    2954.803361/s
iteration_duration.....: avg=324.72µs  min=131.81µs med=314
iterations.....: 14775    2954.803361/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 simulation can be impactful. The system has a single Pod with a 'Never' restart policy, making it vulnerable to downtime if the Pod fails. Additionally, the lack of redundancy means that any failure in the Pod will directly affect the service availability. To simulate a cyber attack, we can start by injecting a 'PodChaos' fault to kill the Pod, testing the system's ability to maintain the 'example-pod-running' steady state. This will reveal the impact of the Pod's restart policy and lack of redundancy. Next, we can inject a 'NetworkChaos' fault to introduce network latency, simulating a DDoS attack, and testing the 'example-service-availability' steady state. This will reveal how the system handles network disruptions and whether the service can maintain a high success rate under stress. The sequence of injecting Pod failure first, followed by network latency, simulates a scenario where an attacker first disrupts the Pod and then targets the network, maximizing the impact on the system.

🧨 Fault-injection sequence

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

⚙ Detailed fault parameters

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

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

```

    "labelSelectors" : {
      "app" : "example"
    }
  }
}

```

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

```

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

```

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 will be divided into three phases: pre-validation, fault-injection, and post-validation. The pre-validation phase will ensure that the system is in a steady state before any faults are injected. This involves checking that the Pod is running and the service is available. The

fault-injection phase will simulate a cyber attack by first killing the Pod and then introducing network latency. This will test the system's resilience to these faults. Finally, the post-validation phase will verify that the system returns to its steady states after the faults are removed. Given the 1-minute constraint, a reasonable time allocation would be 15 seconds for pre-validation, 30 seconds for fault injection, and 15 seconds for post-validation. This allows enough time to observe the system's behavior during each phase while staying within the overall time limit.

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 constraints, we have 15 seconds to perform these checks. We have two steady states to verify: 'example-pod-running' and 'example-service-availability'. To efficiently utilize the time, we will run both unit tests simultaneously. This approach allows us to maximize the use of the available time and ensure that both steady states are verified within the 15-second window. The 'example-pod-running' test will check if the Pod is in the 'Running' state for at least 90% of the time, while the 'example-service-availability' test will verify that the HTTP request success rate is at least 99%. By running these tests concurrently, we can quickly confirm that the system is stable and ready for the fault injection phase.

- Verified Steady State #0: example-pod-running
 - Workflow Name: pre-unittest-example-pod-running
 - Grace Period: 0s
 - Duration: 15s
- Verified Steady State #1: example-service-availability
 - Workflow Name: pre-unittest-example-service-availability
 - Grace Period: 0s
 - Duration: 15s

Fault-injection Phase (30s)

In this fault-injection phase, we aim to simulate a cyber attack scenario by injecting two types of faults: 'PodChaos' and 'NetworkChaos'. The goal is to observe the system's

behavior under these conditions and assess its resilience. Given the 30-second time constraint for this phase, we need to carefully schedule the fault injections and unit tests to maximize the insights gained while ensuring the experiment remains within the allotted time.

The approach is to stagger the fault injections and unit tests to observe the system's response to each fault type separately. We will start with the 'PodChaos' fault, which will kill the Pod, and then follow with the 'NetworkChaos' fault, which introduces network latency. This sequence simulates an attack where the Pod is disrupted first, followed by network interference.

The 'PodChaos' fault will be injected at the beginning of the phase, with a short duration to allow time for the system to react. Immediately after the 'PodChaos' fault, we will run the unit test for the 'example-pod-running' steady state to verify if the Pod remains in the 'Running' state despite the fault. This test will help us understand the impact of the Pod's restart policy and lack of redundancy.

Next, we will inject the 'NetworkChaos' fault, which introduces network latency. After this fault is injected, we will run the unit test for the 'example-service-availability' steady state to check if the service can maintain a high success rate under network stress. This test will reveal how the system handles network disruptions.

By staggering the fault injections and unit tests, we can isolate the effects of each fault and gain a clearer understanding of the system's resilience. The timing is designed to ensure that each fault and test fits within the 30-second phase, allowing us to complete the experiment efficiently.

- Verified Steady State #0: `example-pod-running`
 - Workflow Name: `fault-unittest-example-pod-running`
 - Grace Period: `10s`
 - Duration: `5s`
- Verified Steady State #1: `example-service-availability`
 - Workflow Name: `fault-unittest-example-service-availability`
 - Grace Period: `25s`
 - Duration: `5s`
- Injected Faults #0: `PodChaos`
 - Workflow Name: `fault-podchaos`
 - Grace Period: `0s`
 - Duration: `10s`
- Injected Faults #1: `NetworkChaos`
 - Workflow Name: `fault-networkchaos`

- 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, allowing us to focus on one steady state at a time and ensure accurate validation. The first test will check the 'example-pod-running' steady state, ensuring the Pod is back in the 'Running' state. The second test will verify the 'example-service-availability' steady state, ensuring the service is available with a high success rate. 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 validation.

- Verified Steady State #0: example-pod-running
 - Workflow Name: post-unittest-example-pod-running
 - Grace Period: 0s
 - Duration: 7s
- Verified Steady State #1: example-service-availability
 - Workflow Name: post-unittest-example-service-availability
 - Grace Period: 7s
 - Duration: 8s

Summary

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

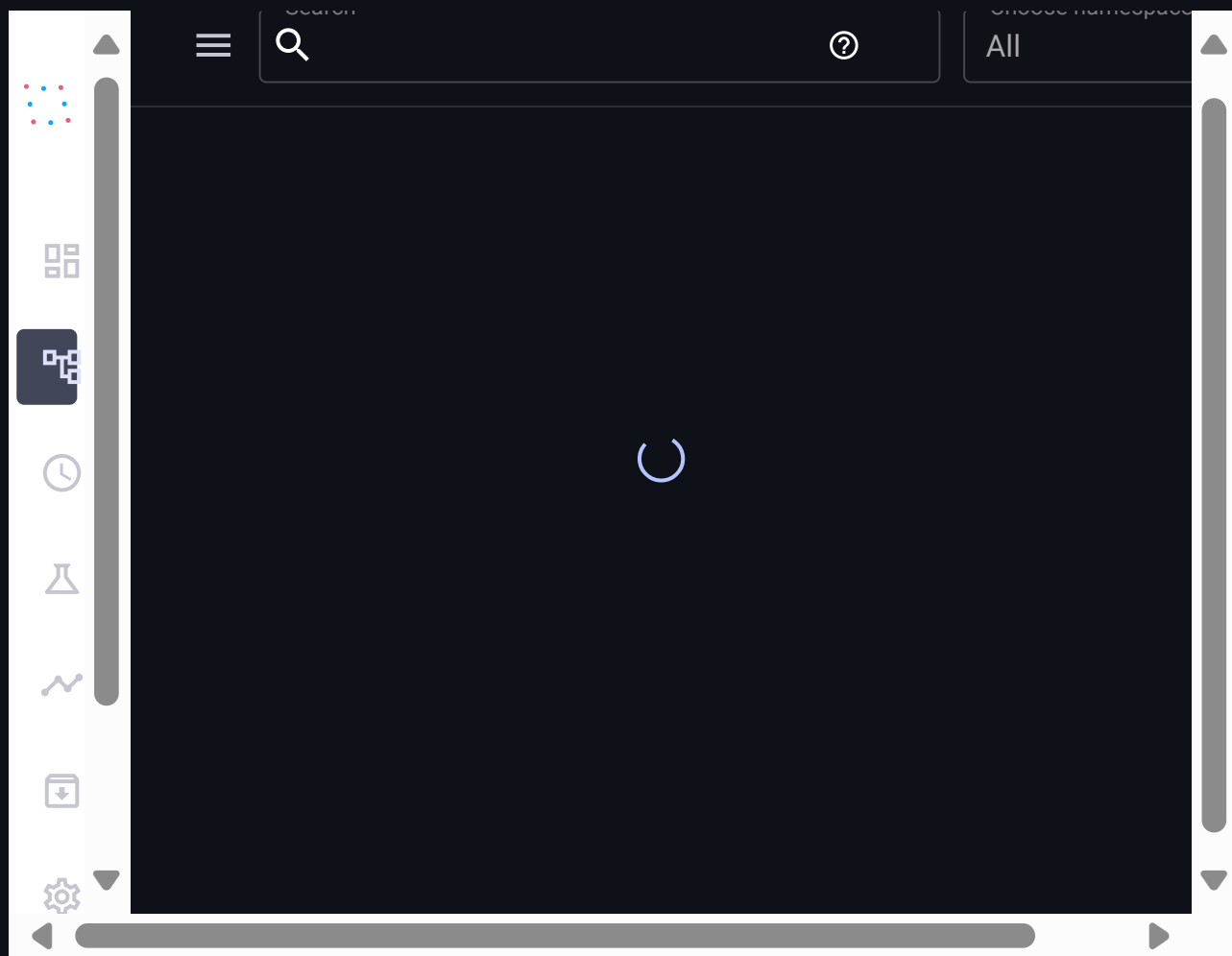
In the pre-validation phase, which lasts for 15 seconds, two unit tests are executed simultaneously to verify the system's steady state. The 'example-pod-running' test, with the workflow name 'pre-unittest-example-pod-running', and the 'example-service-availability' test, with the workflow name 'pre-unittest-example-service-availability', both start immediately at the beginning of this phase and run for the entire 15 seconds. These tests ensure that the Pod is running and the service is available before proceeding to fault injection.

The fault-injection phase spans 30 seconds and involves staggered execution of faults and unit tests. Initially, the 'PodChaos' fault, with the workflow name 'fault-podchaos', is injected at the start of this phase and lasts for 10 seconds. Concurrently, at 10 seconds into the phase, the 'example-pod-running' unit test, with the workflow name

'fault-unittest-example-pod-running', is conducted for 5 seconds to assess the Pod's resilience. Following this, at 15 seconds, the 'NetworkChaos' fault, with the workflow name 'fault-networkchaos', is introduced for 10 seconds. Subsequently, at 25 seconds, the 'example-service-availability' unit test, with the workflow name 'fault-unittest-example-service-availability', is executed for 5 seconds to evaluate the service's performance under network stress.

In the post-validation phase, which also lasts 15 seconds, unit tests are performed sequentially to confirm the system's return to steady states. The 'example-pod-running' test, with the workflow name 'post-unittest-example-pod-running', starts immediately and runs for 7 seconds. Following this, the 'example-service-availability' test, with the workflow name 'post-unittest-example-service-availability', begins at 7 seconds and continues for the remaining 8 seconds. This phase ensures that both the Pod and service have stabilized after the fault injections.

Completed the chaos experiment!



Phase 3: Analysis

The Chaos Engineering experiment results indicate a failure in maintaining the steady states during and after the fault injection phase. Let's analyze the reasons for these failures step by step:

1. **Pre-Validation Phase Success:** Both `pre-unittest-example-pod-running` and `pre-unittest-example-service-availability` tests passed, confirming that the system was in a steady state before the fault injection. The Pod was running, and the service was available, as expected.
2. **Fault-Unit Test Failures:**
 - **`fault-unittest-example-pod-running`:** This test failed because the Pod was not found during the fault injection phase. The 'PodChaos' fault, which involved killing the Pod, was injected at the start of the fault-injection phase. Since the Pod's restart policy is set to 'Never', it did not restart after being killed, leading to a 0% running time during the test. This directly correlates with the identified issue #0 (Pod Restart Policy) and issue #1 (Single Pod Deployment) in the system configuration.
 - **`fault-unittest-example-service-availability`:** This test also failed due to the service being unavailable. The 'PodChaos' fault resulted in the Pod being killed, and since there was no redundancy (issue #1), the service had no backend Pod to route traffic to, leading to connection refusals. The subsequent 'NetworkChaos' fault, which introduced network latency, further compounded the issue by simulating a network disruption, but the primary cause of failure was the absence of a running Pod.
3. **Post-Validation Phase Failures:**
 - **`post-unittest-example-pod-running`:** The Pod was still not found, as indicated by the 404 error. This is expected because the Pod was not configured to restart automatically after being killed, and no manual intervention or controller (like a Deployment) was in place to recreate it.
 - **`post-unittest-example-service-availability`:** The service remained unavailable, as the Pod was not running, leading to continued connection refusals. This highlights the critical impact of having a single Pod without redundancy or a restart mechanism.

Insights and Recommendations:

- **Pod Restart Policy:** Change the Pod's restart policy from 'Never' to 'Always' or 'OnFailure' to ensure it restarts automatically after a failure.
- **Redundancy and Scalability:** Use a Deployment or ReplicaSet instead of a single Pod to manage the Nginx container. This will provide redundancy and ensure that if one Pod fails, others can take over, maintaining service availability.
- **Service Resilience:** Ensure that the Service is backed by multiple Pods to handle failures and maintain availability. This can be achieved by scaling the Deployment to have multiple replicas.
- **Monitoring and Alerts:** Implement monitoring and alerting mechanisms to detect and respond to Pod failures promptly, ensuring minimal downtime.

By addressing these issues, the system can improve its resilience and maintain steady states even during fault scenarios.

Phase 4: Improvement

Reconfiguration

To improve the resiliency of the system, I will replace the existing Pod manifest with a Deployment manifest. This will allow for automatic restarts and scaling of the Nginx container, providing redundancy and ensuring that the service remains available even if one Pod fails. The Deployment will manage the Pods and ensure that the desired number of replicas are running at all times.

Modification_type: replace

File name: nginx/pod.yaml

Replace the Pod manifest with a Deployment manifest to provide redundancy and automatic recovery. The Deployment will manage multiple replicas of the Pod, ensuring that the service remains available even if one Pod fails.

```
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 180ns
- deployment.apps/example-deployment configured
- service/example-service configured
Waiting for deployments to stabilize...
- deployment/example-deployment is ready.
Deployments stabilized in 4.105 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 | RES |
|-----------|---|-------|---------|-----|
| default | pod/example-deployment-6977c97c99-7dcbd | 1/1 | Running | 0 |
| default | pod/example-deployment-6977c97c99-7x9gb | 1/1 | Running | 0 |
| default | pod/example-deployment-6977c97c99-gbxgz | 1/1 | Running | 0 |

| NAMESPACE | NAME | TYPE | CLUSTER-IP | EXTERNAL-IP |
|-----------|-------------------------|-----------|-------------|-------------|
| default | service/example-service | ClusterIP | 10.96.10.57 | <none> |

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

| NAMESPACE | NAME | DESIRED | CURREI |
|-----------|---|---------|--------|
| default | replicaset.apps/example-deployment-65968c46f | 0 | 0 |
| default | replicaset.apps/example-deployment-6977c97c99 | 3 | 3 |

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 current K8s manifests have changed from a single Pod to a Deployment with 3 replicas. This means that the system now has redundancy, and killing one Pod will not have the same impact as before. However, the original goal of the chaos experiment was to test the system's resilience to a Pod being killed, so we should maintain the same label selector to target one of the Pods in the Deployment. The namespace remains the same, so no changes are needed there.

Next fault injection scope:

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

Current fault injection settings:

- Injected Faults #0: **NetworkChaos**
 - Workflow Name: **fault-networkchaos**
 - Grace Period: **15s**
 - Duration: **10s**

```
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'. 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. 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. The fault injection will now affect all Pods created by the Deployment that match the label, which aligns with the original goal of testing the system's resilience under network chaos conditions.

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 rather than a single Pod, and it should verify that all Pods created by the Deployment maintain the steady state. The test should be updated to check the status of all Pods managed by the Deployment and ensure that the running percentage meets the threshold for the entire set of Pods. The test should iterate over all Pods with the label 'app: example' and calculate the running percentage for the entire group.

```

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

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

    def test_deployment_running_state(self):
        running_count = 0
        total_pods = 0
        # Check the pod status every second for the specified duration
        for _ in range(self.duration):
            try:
                pods = self.v1.list_namespaced_pod(namespace=self.namespace)
                total_pods = len(pods)
                running_pods = [pod for pod in pods if pod.status.phase == 'Running']
                running_count += len(running_pods)
                print(f"Running pods: {len(running_pods)} out of {total_pods}")
            except client.exceptions.ApiException as e:
                print(f"Exception when calling CoreV1Api->list_namespaced_pod: {e}")
            time.sleep(1)

        # Calculate the running percentage
        running_percentage = (running_count / (self.duration * total_pods))

```

```

        print(f"Pods were running {running_count} out of {self.duration * total}")
        print(f"Running percentage: {running_percentage}%")
        # Assert that the running percentage meets the threshold
        assert running_percentage >= 90, f"Pod running percentage {running_

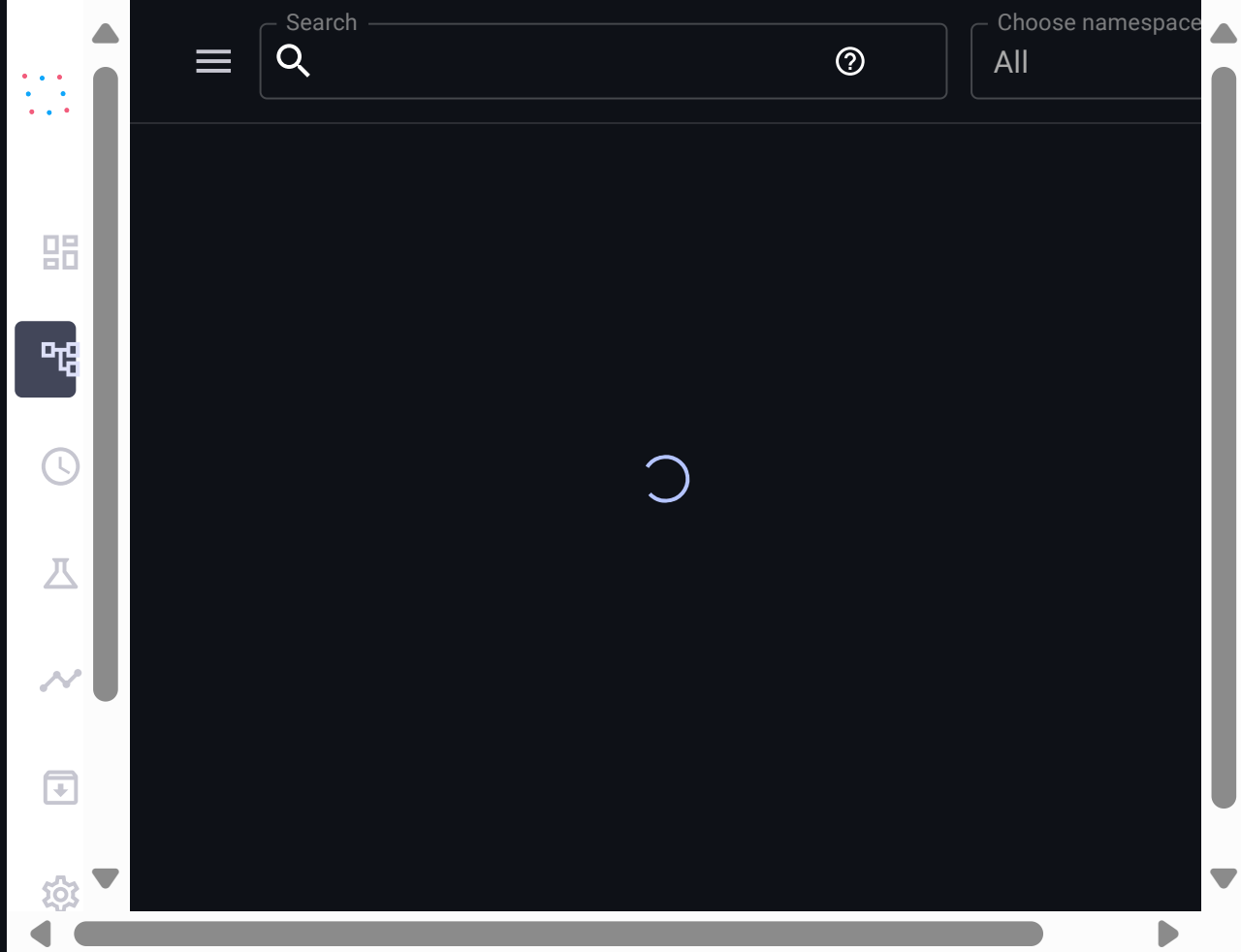
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Test the running state of
    parser.add_argument('--duration', type=int, default=5, help='Duration to
    args = parser.parse_args()
    # Create an instance of the test class
    test = TestDeploymentRunningState(namespace='default', label_selector='
    # Run the test
    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, now managed by a Deployment instead of a standalone Pod. The reconfiguration introduces redundancy by scaling the application to 3 replicas, but this does not affect the endpoint or the HTTP request being tested. The unit test checks the availability and success rate of HTTP requests to the service, which should still be valid as the service endpoint and port remain the same. Therefore, the unit test does not require any modification.

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 structured approach to testing the resilience of a Kubernetes-based system, specifically focusing on an Nginx web server setup. The 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 'Never' restart policy, while the Service manifest exposes this Pod on port 80. Identified weaknesses 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 90% of the time, and the Service should have an HTTP request success rate of at least 99%. These are tested using Python scripts and K6 JavaScript, respectively.

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, all within a 1-minute timeframe.

Pre-validation ensures the system is stable, fault-injection tests resilience under attack, and post-validation checks recovery.

Initially, the experiment fails due to the Pod not restarting after being killed and the Service becoming unavailable. Analysis reveals the need for a Deployment to manage multiple Pod replicas, providing redundancy and automatic recovery. After modifying the Pod manifest to a Deployment with three replicas, the experiment is rerun successfully, with all unit tests passing, confirming the system's improved resilience.

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