



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-20241124-122145" deleted  
$ kubectl delete workflownode --all --context kind-chaos-eater-cluster -n cl  
workflownode.chaos-mesh.org "fault-injection-overlapped-workflows-5qjjc" de  
workflownode.chaos-mesh.org "fault-injection-parallel-workflow-dpdfq" delet  
workflownode.chaos-mesh.org "fault-injection-parallel-workflows-77xld" dele  
workflownode.chaos-mesh.org "fault-injection-phase-jt4kw" deleted  
workflownode.chaos-mesh.org "fault-injection-suspend-n2zf4" deleted  
workflownode.chaos-mesh.org "fault-injection-suspend-workflow-fvnqq" delete  
workflownode.chaos-mesh.org "fault-networkchaos-c5vks" deleted  
workflownode.chaos-mesh.org "fault-podchaos-l2swl" deleted  
workflownode.chaos-mesh.org "fault-unittest-podrunningstatusnginxpodyaml-g9  
workflownode.chaos-mesh.org "fault-unittest-serviceavailabilitynginxservice  
workflownode.chaos-mesh.org "post-unittest-podrunningstatusnginxpodyaml-s9v  
workflownode.chaos-mesh.org "post-unittest-serviceavailabilitynginxservice  
workflownode.chaos-mesh.org "post-validation-overlapped-workflows-5n2v7" de  
workflownode.chaos-mesh.org "post-validation-phase-8z5fh" deleted  
workflownode.chaos-mesh.org "post-validation-suspend-t2lq4" deleted  
workflownode.chaos-mesh.org "post-validation-suspend-workflow-9jnmX" delete  
workflownode.chaos-mesh.org "post-validation-suspend-workflow2-2nsmt" delet  
workflownode.chaos-mesh.org "post-validation-suspend2-skrx8" deleted  
workflownode.chaos-mesh.org "pre-unittest-podrunningstatusnginxpodyaml-dxng  
workflownode.chaos-mesh.org "pre-unittest-serviceavailabilitynginxservice  
workflownode.chaos-mesh.org "pre-validation-parallel-workflows-6phh7" delet  
workflownode.chaos-mesh.org "pre-validation-phase-zd559" deleted  
workflownode.chaos-mesh.org "the-entry-mdfbq" deleted
```

Input instructions for your Chaos Engineering



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

```
$ kubectl delete all --all-namespaces --context kind-chaos-eater-cluster -l
pod "example-pod" deleted
service "example-service" 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 141ns
- 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=project=chaos-eater
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	pod/example-pod	1/1	Running	0	7s

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	ClusterIP	10.96.77.132	<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 a single container named 'example-container'.
- The container uses the 'nginx:1.17.1' image.
- The container exposes port 80, which is typically used for HTTP 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.
- It 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 is deleted or fails.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: kind: Pod

Application of the manifests:

Thoughts:

The manifests provided are for a Pod and a Service in Kubernetes, both of which are associated with the label 'app: example'. The Pod runs an Nginx container, which is a popular web server and reverse proxy server. The Service is configured to expose the Pod on port 80, which is the default port for HTTP traffic. Given these details, it is logical to assume that the application is a simple web server or a static website hosted using Nginx. The use of Nginx and the exposure of port 80 strongly suggest that the application is intended to serve web content.

Assumed application:

Web server hosting a static website 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: PodRunningStatusNginxPodYaml

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 automatically, leading to potential downtime. Therefore, the steady state should verify that the Pod is running and not in a failed state.

This can be measured by checking the Pod's status to ensure it is 'Running'. 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 Pod's running status, I will use the K8s API. This is because the task requires checking the status of a specific Kubernetes resource, which is best done using the Kubernetes client libraries. The Pod is defined in the 'nginx/pod.yaml' manifest and is deployed in the 'default' namespace since no namespace is specified. The script will check the Pod's status every second for a duration of 5 seconds to ensure it is in the 'Running' state. This approach is suitable for verifying the Pod's status within the given time constraints of the Chaos Engineering experiment.

tool: `k8s` duration: `5s`

Inspection script (Python) `k8s_PodRunningStatusNginxPodYaml.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')
    parser.add_argument('--pod-name', type=str, default='nginx', help='Pod name')
    parser.add_argument('--namespace', type=str, default='default', help='Namespace')
    args = parser.parse_args()
    check_pod_status(args.namespace, args.pod_name, args.duration)
```

```
args = parser.parse_args()
check_pod_status(namespace='default', pod_name='example-pod', du
```

Pod `k8spodrunningstatusnginxpodyaml-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 the Pod's running status. The current state shows that the Pod was running for 5 out of 5 seconds, which is 100% of the time. Given that the Pod's restart policy is set to 'Never', it is crucial to ensure that the Pod remains in the 'Running' state to avoid downtime. However, to account for minor fluctuations or delays in status updates, a reasonable threshold would be to expect the Pod to be in the 'Running' state for at least 90% of the time during the observation period. This allows for some tolerance while still ensuring that the Pod is generally stable and operational.

The Pod should be in the 'Running' state for 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 count how many times it is in the 'Running' state. At the end of the loop, I will calculate the percentage of time the Pod was running and assert that this percentage is greater than or equal to 90%. This ensures that the Pod meets the steady state threshold requirement.

```
import os
import time
import argparse
```

```

from kubernetes import client, config
from unittest_base import K8sAPIBase

class TestPodRunningStatus(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_status(self):
        running_count = 0
        # Loop for the specified duration
        for _ in range(self.duration):
            try:
                # Read the Pod status
                pod = self.v1.read_namespaced_pod(name=self.pod_name, namespace=self.namespace)
                # Check if the Pod is in 'Running' state
                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_status: {e}")
            time.sleep(1)
        # Calculate the percentage of time the Pod was running
        running_percentage = (running_count / self.duration) * 100
        print(f"Pod was running {running_count} out of {self.duration} seconds")
        # Assert that the Pod was running at least 90% of the time
        assert running_percentage >= 90, "Pod did not meet the 90% requirement"

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

```

Pod `unittestpodrunningstatusnginxpodyamlmod0-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, which is 100.0% of the time.
```

✓ Steady state #2: ServiceAvailabilityNginxServiceYaml

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 is deleted or fails. To verify the system's resilience, we should define a steady state that measures the availability of the web service provided by the Pod. Since the Service is responsible for exposing the Pod, we can measure the HTTP response rate or availability of the service. This steady state will ensure that the Service is consistently able to route traffic to the Pod, even if the Pod encounters issues. This is specific to the 'nginx/service.yaml' manifest, which defines the Service that selects the Pod.

Current state inspection

To inspect the availability of the web service provided by the Pod, we need to measure the HTTP response rate or availability of the service. Since the Service is responsible for exposing the Pod, using k6 is appropriate to simulate traffic and check the service's response. This will help us verify if the Service can consistently route traffic to the Pod, even if the Pod encounters issues. We'll use k6 to send HTTP requests to the service and check for successful responses.

tool: `k6` vus: `10` duration: `5s`

Inspection script (Javascript) `k6_ServiceAvailabilityNginxServiceYaml.js` :

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

export let options = {
  vus: 10,
  duration: '5s',
};

export default function () {
  let res = http.get('http://example-service.default.svc.cluster.local');
  check(res, {
```



```
'status is 200': (r) => r.status === 200,
});
}
```

Pod `k6serviceavailabilitynginxserviceyaml-pod` has completed successfully.

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

```
✓ status is 200
```

```
checks.....: 100.00% 81085 out of 81085
data_received.....: 69 MB    14 MB/s
data_sent.....: 8.9 MB    1.8 MB/s
http_req_blocked.....: avg=8.67µs    min=1.2µs    med=3.49µs
http_req_connecting.....: avg=1.88µs    min=0s      med=0s
http_req_duration.....: avg=488.76µs min=96.52µs med=440.9µs
  { expected_response:true }...: avg=488.76µs min=96.52µs med=440.9µs
http_req_failed.....: 0.00%    0 out of 81085
http_req_receiving.....: avg=56.19µs   min=10.68µs med=38.86µs
http_req_sending.....: avg=16.21µs   min=3.57µs   med=10.51µs
http_req_tls_handshaking.....: avg=0s        min=0s        med=0s
http_req_waiting.....: avg=416.35µs  min=60.66µs  med=387.0µs
http_reqs.....: 81085    16210.604071/s
iteration_duration.....: avg=601.02µs  min=155.09µs med=526.5µs
iterations.....: 81085    16210.604071/s
vus.....: 10      min=10      max=10
vus_max.....: 10      min=10      max=10
```

► Threshold

The steady state we are considering is the availability of the web service provided by the Nginx Pod, as measured by the HTTP response rate. The k6 test results show that 100% of the requests received a 200 status code, indicating that the service is fully available. To account for potential fluctuations and ensure the threshold is reasonable, we should allow for a small margin of error. A typical approach is to set the threshold slightly below 100% to accommodate minor network issues or transient errors that might occur in a real-world scenario. Given the current state shows perfect availability, a threshold of 99.5% would be reasonable, allowing for some tolerance while still ensuring high availability.

HTTP response rate should be $\geq 99.5\%$ with status code 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 HTTP response rate with a status code of 200 should be at least 99.5%. This ensures that the test will fail if the availability of the service drops below this threshold, thus verifying the steady state of the system.

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

export let options = {
  vus: 10,
  duration: '5s',
  thresholds: {
    'http_req_failed': ['rate<0.005'], // Allowing for a 0.5% failure rate
    'http_req_duration': ['p(95)<200'], // Optional: 95% of requests
  },
};

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

Pod `unittestserviceavailabilitynginxserviceyamlmod0-pod` has completed successfully.

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

✓ status is 200

```
checks.....: 100.00% 77358 out of 77358
data_received.....: 66 MB    13 MB/s
data_sent.....: 8.5 MB    1.7 MB/s
http_req_blocked.....: avg=9.56µs    min=1.16µs    med=3.8µs
http_req_connecting.....: avg=2.08µs    min=0s        med=0s
✓ http_req_duration.....: avg=499.15µs min=112.5µs    med=453µs
  { expected_response:true }...: avg=499.15µs min=112.5µs    med=453µs
✓ http_req_failed.....: 0.00%    0 out of 77358
http_req_receiving.....: avg=63.92µs   min=12.24µs   med=44.µs
http_req_sending.....: avg=19.31µs   min=3.46µs    med=12.µs
http_req_tls_handshaking.....: avg=0s        min=0s        med=0s
http_req_waiting.....: avg=415.91µs  min=59.78µs   med=393µs
```

```
http_reqs.....: 77358      15470.030934/s
iteration_duration.....: avg=628.21µs min=188.75µs med=549
iterations.....: 77358      15470.030934/s
vus.....: 10      min=10      max=10
vus_max.....: 10      min=10      max=10
```

Fault definition

✓ Scenario: Cyber Attack

🗨 Description

In a cyber attack scenario, the attacker might target the single point of failure in the system, which is the single Pod running the Nginx server. The Pod's restart policy is set to 'Never', meaning if it fails, it won't restart automatically, leading to downtime. Additionally, the lack of redundancy due to the single Pod deployment without a controller like Deployment or ReplicaSet makes the system vulnerable to attacks that can take down the Pod. To simulate this, we can inject a PodChaos fault to kill the Pod, testing the system's ability to maintain the steady state of service availability. This will directly test the system's weakness in handling Pod failures and its impact on the steady state of service availability. Following this, we can simulate a network partition using NetworkChaos to test if the service can still route traffic effectively, even if the Pod is isolated from the network. This sequence simulates a cyber attack where the attacker first takes down the Pod and then attempts to disrupt network communication.

⚙ Fault-injection sequence

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

⚙ 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', 'pod_name': 'example-pod'})

```

{
  "action" : "partition"
  "mode" : "one"
  "selector" : {
    "namespaces" : [
      0 : "default"
    ]
    "labelSelectors" : {
      "app" : "example"
    }
  }
  "direction" : "both"
}

```

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.

1. **Pre-validation Phase:** This phase ensures that the system is in a steady state before we introduce any faults. We need to verify that the Pod is running and the service is available. Since the steady state checks are relatively quick (5 seconds for Pod

status and 5 seconds for service availability), we can allocate 10 seconds for this phase.

2. **Fault Injection Phase:** This is the core of the experiment where we introduce faults to observe the system's behavior. We have two faults to inject: PodChaos to kill the Pod and NetworkChaos to partition the network. Given the time constraint, we will inject these faults sequentially but quickly. We can allocate 30 seconds for this phase, allowing 15 seconds for each fault to take effect and observe the immediate impact.
3. **Post-validation Phase:** After the faults are removed, we need to ensure that the system returns to its steady state. Similar to the pre-validation phase, we will check the Pod's running status and the service's availability. We can allocate 20 seconds for this phase to ensure the system stabilizes and meets the steady state thresholds.

The total time for the experiment is 60 seconds, which is within the 1-minute limit. This allocation allows us to thoroughly test the system's resilience while adhering to the time constraint.

Total experiment time: 60s

Pre-validation Phase: 10s

Fault-injection Phase: 30s

Post-validation Phase: 20s

Pre-validation Phase (10s)

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, we have 10 seconds to perform these checks. We have two steady states to verify: the Pod's running status and the Service's availability. Since both checks are critical to ensure the system's readiness, they will be executed simultaneously to maximize the use of the available time. The Pod's running status will be checked continuously for 5 seconds, while the Service's availability will be tested using a k6 script for the same duration. This approach ensures that both steady states are verified within the 10-second window, allowing us to confidently proceed to the fault injection phase.

- Verified Steady State #0: PodRunningStatusNginxPodYaml
 - Workflow Name: pre-unittest-podrunningstatusnginxpodyaml
 - Grace Period: 0s
 - Duration: 5s

- Verified Steady State #1: `ServiceAvailabilityNginxServiceYaml`
 - Workflow Name: `pre-unittest-serviceavailabilitynginxserviceyaml`
 - Grace Period: `0s`
 - Duration: `5s`

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 total duration for this phase is 30 seconds, so we need to carefully schedule the fault injections and unit tests to fit within this timeframe.

First, we will inject the PodChaos fault to simulate a pod failure. This will test the system's ability to maintain the steady state of service availability when the pod is killed. We will start this fault injection immediately at the beginning of the phase (grace period of 0s) and let it run for 10 seconds.

Simultaneously, we will run the unit test for the first steady state, 'PodRunningStatusNginxPodYaml', to verify if the pod remains in the 'Running' state for at least 90% of the time. This test will also start at 0s and run for 10 seconds, aligning with the PodChaos fault duration.

Next, we will inject the NetworkChaos fault to simulate a network partition. This will test the system's ability to maintain service availability despite network disruptions. We will start this fault injection at 10 seconds (grace period of 10s) and let it run for another 10 seconds.

During the NetworkChaos fault, we will run the unit test for the second steady state, 'ServiceAvailabilityNginxServiceYaml', to verify if the HTTP response rate remains at or above 99.5%. This test will start at 10 seconds and run for 10 seconds, aligning with the NetworkChaos fault duration.

By staggering the fault injections and aligning them with the unit tests, we can effectively observe the system's behavior under each fault condition and ensure that the steady states are verified during the fault-injection phase.

- Verified Steady State #0: `PodRunningStatusNginxPodYaml`
 - Workflow Name: `fault-unittest-podrunningstatusnginxpodyaml`
 - Grace Period: `0s`
 - Duration: `10s`
- Verified Steady State #1: `ServiceAvailabilityNginxServiceYaml`
 - Workflow Name: `fault-unittest-serviceavailabilitynginxserviceyaml`

- Grace Period: 10s
- Duration: 10s
- 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: 10s

Post-validation Phase (20s)

In the post-validation phase, we need to ensure that the system has returned to its steady states after the fault injection. Given the constraints of a 20-second total time for this phase, we will conduct unit tests to verify each steady state. The tests will be executed sequentially due to the short time frame, ensuring that each steady state is verified independently. The first test will check the Pod's running status, as this is critical to ensure that the Pod has recovered from the PodChaos fault. The second test will verify the service availability, ensuring that the network partition did not cause lasting disruption. Each test will have a short grace period to allow the system to stabilize after the fault injection, followed by a brief duration to perform the checks. This approach ensures that we can quickly verify the system's recovery while adhering to the time constraints.

- Verified Steady State #0: PodRunningStatusNginxPodYaml
 - Workflow Name: post-unittest-podrunningstatusnginxpodyaml
 - Grace Period: 2s
 - Duration: 8s
- Verified Steady State #1: ServiceAvailabilityNginxServiceYaml
 - Workflow Name: post-unittest-serviceavailabilitynginxserviceyaml
 - Grace Period: 2s
 - Duration: 8s

Summary

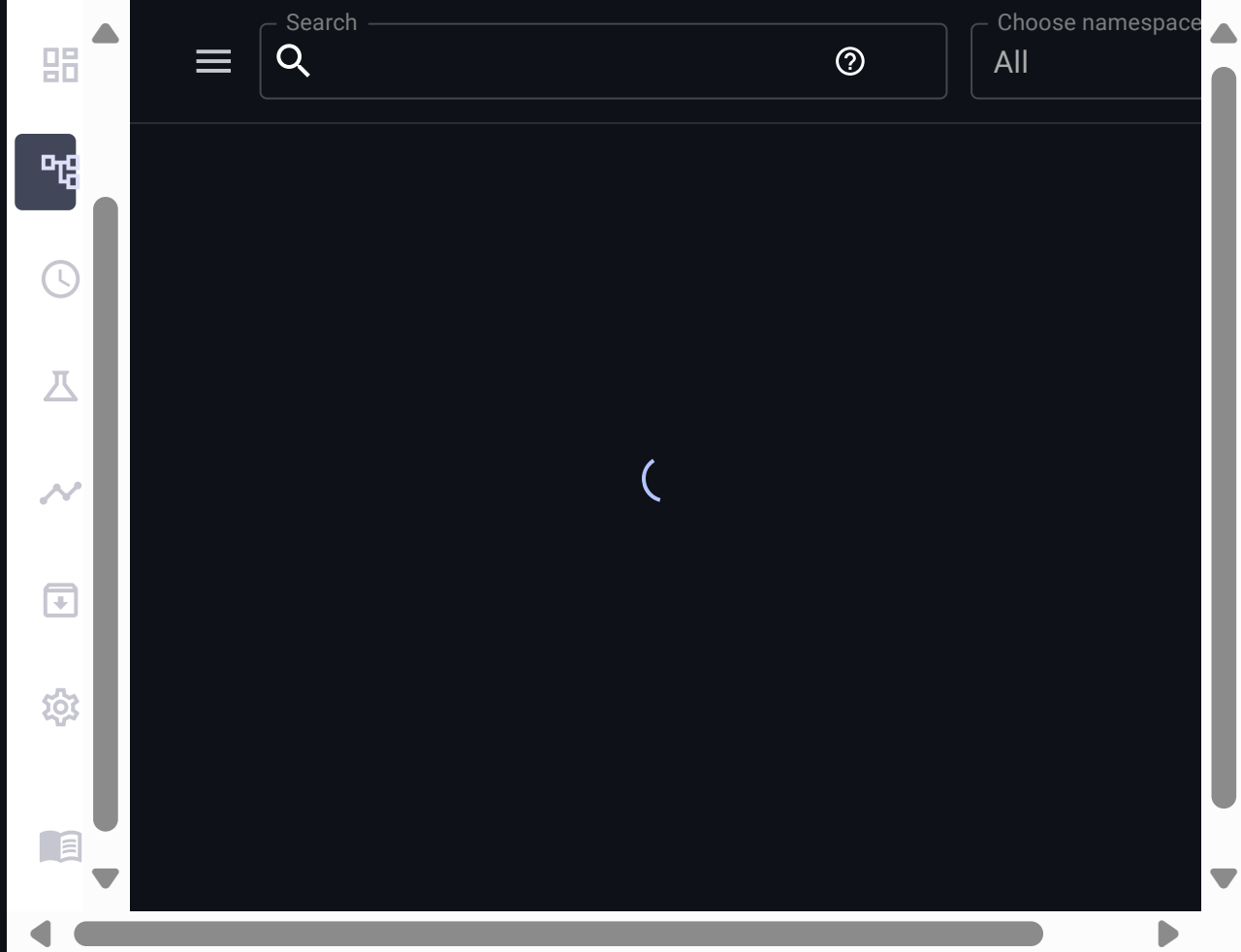
The Chaos Engineering experiment is structured into three phases within a 1-minute timeframe: pre-validation, fault-injection, and post-validation.

In the pre-validation phase, which lasts for 10 seconds, two unit tests are executed simultaneously to ensure the system is in a steady state. The first test, 'PodRunningStatusNginxPodYaml', checks the Pod's running status with a workflow named 'pre-unittest-podrunningstatusnginxpodyaml'. It starts immediately and runs for 5 seconds. Concurrently, the second test, 'ServiceAvailabilityNginxServiceYaml', verifies the service's availability using a k6 script, with a workflow named 'pre-unittest-serviceavailabilitynginxserviceyaml'. This test also starts immediately and runs for 5 seconds.

The fault-injection phase spans 30 seconds and involves two sequential fault injections. Initially, the 'PodChaos' fault is injected to simulate a pod failure, starting at the beginning of the phase and lasting for 10 seconds. This is managed by the 'fault-podchaos' workflow. Simultaneously, the unit test 'PodRunningStatusNginxPodYaml' is conducted to verify the pod's running status, starting at 0 seconds and running for 10 seconds, with the workflow 'fault-unittest-podrunningstatusnginxpodyaml'. Following this, the 'NetworkChaos' fault is injected to simulate a network partition, starting at 10 seconds and lasting for 10 seconds, managed by the 'fault-networkchaos' workflow. Concurrently, the unit test 'ServiceAvailabilityNginxServiceYaml' is executed to verify service availability, starting at 10 seconds and running for 10 seconds, with the workflow 'fault-unittest-serviceavailabilitynginxserviceyaml'.

In the post-validation phase, which lasts for 20 seconds, two unit tests are executed sequentially to ensure the system has returned to its steady state. The first test, 'PodRunningStatusNginxPodYaml', checks the Pod's running status, starting after a 2-second grace period and running for 8 seconds, with the workflow 'post-unittest-podrunningstatusnginxpodyaml'. The second test, 'ServiceAvailabilityNginxServiceYaml', verifies service availability, also starting after a 2-second grace period and running for 8 seconds, with the workflow 'post-unittest-serviceavailabilitynginxserviceyaml'. This sequential execution ensures each steady state is independently verified within the 20-second timeframe.

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-validation tests passed, indicating that the system was initially in a steady state. The Pod was running, and the service was available, confirming that the system was correctly set up before the fault injection.
2. **Fault Injection Phase Failures:**
 - **PodChaos Fault:** The 'PodChaos' fault was injected to kill the Pod. The test 'fault-unittest-podrunningstatusnginxpodyaml' failed because the Pod was not found (HTTP 404 error). This is expected behavior given the Pod's restart policy is set to 'Never', meaning it does not automatically restart after being killed. This confirms the identified resiliency issue #0 (Pod Restart Policy) and issue #1 (Single Pod Deployment) in the system configuration.
 - **NetworkChaos Fault:** The 'NetworkChaos' fault was injected to simulate a network partition. The test 'fault-unittest-serviceavailabilitynginxserviceyaml' failed because the service was unable to connect to the Pod (connection refused errors). This failure is a

direct consequence of the Pod being unavailable due to the PodChaos fault, highlighting the lack of redundancy and automatic recovery mechanisms in the system.

3. Post-Validation Phase Failures:

- **Pod Running Status:** The 'post-unittest-podrunningstatusnginxpodyaml' test failed because the Pod was still not found. This indicates that the system did not recover from the PodChaos fault, as expected due to the 'Never' restart policy.
- **Service Availability:** The 'post-unittest-serviceavailabilitynginxserviceyaml' test also failed, with continued connection refused errors. This further confirms that the service could not route traffic because the Pod was not running, emphasizing the impact of the single Pod deployment without redundancy.

Insights and Recommendations:

- **Pod Restart Policy:** Change the Pod's restart policy from 'Never' to 'Always' or 'OnFailure' to ensure automatic recovery in case of failures.
- **Redundancy and Scalability:** Deploy the Pod using a Deployment or ReplicaSet to manage multiple replicas, providing redundancy and automatic recovery. This will mitigate the single point of failure issue and improve the system's resilience against Pod failures.
- **Service Resilience:** Ensure that the Service can handle Pod failures by having multiple Pods behind it. This can be achieved by scaling the Deployment to have more replicas.

By addressing these issues, the system will be better equipped to maintain its steady states even when faults are injected, thus improving its overall resilience.

Phase 4: Improvement

Reconfiguration

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

Modification_type: replace

File name: nginx/pod.yaml

Replace the Pod manifest with a Deployment manifest to provide redundancy and automatic recovery. The Deployment will manage multiple replicas of the Pod, ensuring that the application remains available even if one Pod fails. The restart policy is set to 'Always' to ensure Pods are restarted automatically if they fail.

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

Deploying reconfigured resources... Done

```
$ scaffold run --kube-context kind-chaos-eater-cluster -l project=chaos-eater
No tags generated
Starting test...
Starting deploy...
Loading images into kind cluster nodes...
Images loaded in 260ns
- deployment.apps/example-deployment configured
- service/example-service configured
Waiting for deployments to stabilize...
- deployment/example-deployment is ready.
Deployments stabilized in 3.109 seconds
You can also run [scaffold run --tail] to get the logs
```

Resource statuses

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

NAMESPACE	NAME	READY	STATUS	RESTA
-----------	------	-------	--------	-------

default	pod/example-deployment-f849656cf-bmc42	1/1	Running	0
default	pod/example-deployment-f849656cf-fth87	1/1	Running	0
default	pod/example-deployment-f849656cf-zzqqq	1/1	Running	0

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	ClusterIP	10.96.194.25	<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-dff474d6d	0	0
default	replicaset.apps/example-deployment-f849656cf	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 previous K8s manifests targeted a single Pod with the label 'app: example' for the PodChaos fault injection. In the current manifests, the system has been updated to use a Deployment with 3 replicas, all of which have the same label 'app: example'. The original goal of the chaos experiment was to test the system's resilience by killing one of the Pods. Since the Deployment now manages multiple Pods, the scope of the fault injection should be adjusted to reflect this change. However, the 'mode: one' parameter in the PodChaos configuration already ensures that only one Pod will be targeted, which aligns with the original experiment's goal. Therefore, the selector remains the same, targeting Pods with the label 'app: example' in the 'default' namespace.

Next fault injection scope:

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

```
}  
}
```

Current fault injection settings:

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

Parameters `{{"action": "partition", "mode": "one", "selector": {"namespaces": ["default"], "labelSelectors": {"app": "example"}}}, "direction": "both"}}`

The previous manifests targeted a single Pod with the label 'app: example' in the 'default' namespace. The current manifests have changed from a single Pod 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 Pod to a Deployment does not affect the label-based targeting, so the selector remains appropriate for the current setup.

Next fault injection scope:

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

Adjusted unittest

The previous unit test targets a Pod resource, specifically checking the running status of a Pod named 'example-pod'. In the reconfigured manifests, the Pod has been replaced by a Deployment named 'example-deployment' with 3 replicas. This change means that the unit test should now target the Deployment instead of a single Pod. The test should verify that at least one of the Pods managed by the Deployment is running at least 90% of the time, as the redundancy is intended to ensure availability. Therefore, the unit test needs to be updated to check the status of Pods created by the Deployment, rather than a single Pod.

```
import os  
import time  
import argparse
```

```

from kubernetes import client, config
from unittest_base import K8sAPIBase

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

    def test_deployment_running_status(self):
        running_count = 0
        # Loop for the specified duration
        for _ in range(self.duration):
            try:
                # List the Pods with the label selector matching the Deployment
                pods = self.v1.list_namespaced_pod(namespace=self.namespace)
                # Check if at least one Pod is in 'Running' state
                if any(pod.status.phase == 'Running' for pod in pods.items):
                    running_count += 1
                print(f"Number of running Pods: {sum(pod.status.phase == 'Running' for pod in pods.items)}")
            except client.exceptions.ApiException as e:
                print(f"Exception when calling CoreV1Api->list_namespaced_pod: {e}")
                time.sleep(1)
        # Calculate the percentage of time at least one Pod was running
        running_percentage = (running_count / self.duration) * 100
        print(f"At least one Pod was running {running_count} out of {self.duration} seconds")
        # Assert that at least one Pod was running at least 90% of the time
        assert running_percentage >= 90, "Deployment did not meet the 90% running requirement"

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Test the running status of a deployment')
    parser.add_argument('--duration', type=int, default=5, help='Duration to run the test in seconds')
    args = parser.parse_args()
    # Create an instance of the test class
    test = TestDeploymentRunningStatus(namespace='default', deployment_name='example-service', duration=args.duration)
    # Run the test
    test.test_deployment_running_status()

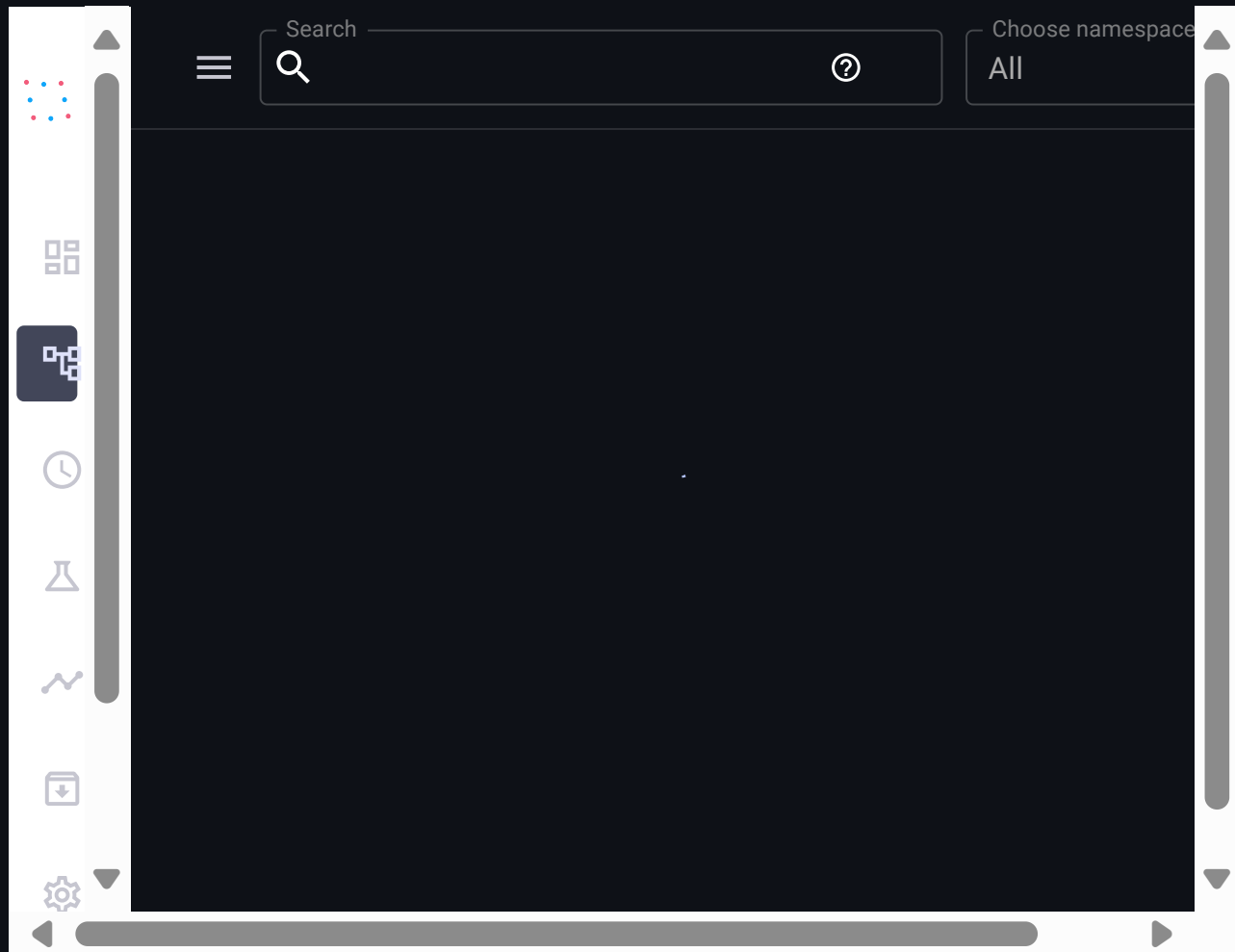
```

Adjusted unittest

The target of the unit test is the service 'example-service', which remains unchanged in the reconfigured manifests. The service still points to the pods labeled with 'app: example', and the service configuration itself has not changed. The reconfiguration involves changing the pod to a deployment with 3 replicas, which is intended to ensure redundancy and maintain the steady state. However, the unit test is designed to test the service's availability and response time, not

the individual pod configuration. Therefore, the unit test does not require any modification, as it will still verify the service's ability to handle requests and meet the specified thresholds.

Completed the chaos experiment!



Your k8s yaml already has good resilience!!!

Phase EX: Postprocessing

Summary of your k8s yaml

The Chaos Engineering cycle described involves a systematic approach to testing the resilience of a Kubernetes-based system running an Nginx web server. The cycle begins with understanding the user inputs, which include Kubernetes manifests for a Pod and a Service. The Pod manifest specifies a single Nginx container with a restart policy of 'Never', while the Service manifest exposes this Pod on port 80. Two 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 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 response rate of at least 99.5% with status code 200. 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 partition the network. 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 in a steady state, fault-injection tests the system's resilience under attack, and post-validation checks if the system returns to its steady state.

The first experiment attempt fails during the fault-injection and post-validation phases, as the Pod does not restart due to its 'Never' restart policy, and the Service cannot connect to the Pod. The analysis suggests changing the Pod's restart policy and deploying it using a Deployment for redundancy.

After modifying the Pod manifest to a Deployment with three replicas and a restart policy of 'Always', the second experiment attempt passes all unit tests, confirming the system's improved resilience. This cycle demonstrates the importance of redundancy and automatic recovery in maintaining system stability under fault conditions.

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