



ChaosEater demo

Live Q&A

Settings



Model

anthropic/claude-3-5-son...



Token

Hugging Face token



Cluster selection

kind-chaos-eater-cluster



Clean the cluster

- ☒ Clean the cluster before run
- ☒ Clean the cluster after run
- ☒ New deployment

Max # steady states

3



Max retries

3



Seed (GPTs only)

42



Usage:

Total billing: \$0.03  
Total tokens: 3.997k  
Input tokens: 3.001k  
Output tokens: 0.996k

Command history

kubectl apply -f  
sandbox/cycle\_20241128\_075141/hyp  
othesis/k8s\_ExamplePodRunningAndR  
eady\_pod.yaml --context kind-chaos-  
eater-cluster -n chaos-eater

kubectl logs



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

```
$ kubectl delete all --all-namespaces --context kind-chaos-eater-cluster -l app=example
pod "example-deployment-6cb7ddd6c4-92f5r" deleted
pod "example-deployment-6cb7ddd6c4-g6v8z" deleted
pod "example-deployment-6cb7ddd6c4-jrdmq" 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
```

Input instructions for your Chaos Engineering



```

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 221ms
- pod/example-pod created
- service/example-service created
Waiting for deployments to stabilize...
- pods is ready.
Deployments stabilized in 3.087 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	5s

  

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	ClusterIP	10.96.173.52	<none>

## Summary of each manifest:

nginx/pod.yaml

- This manifest defines a Kubernetes Pod
- The Pod is named 'example-pod'
- It has a label 'app: example'
- The Pod's restart policy is set to 'Never'
- It contains a single container:
  - The container is named 'example-container'
  - It uses the nginx image version 1.17.1
  - The container exposes port 80
- This Pod can be used to run a simple nginx web server

nginx/service.yaml

- This manifest defines a Kubernetes Service resource
- The Service is named 'example-service'
- It selects pods with the label 'app: example'
- The Service exposes port 80 and forwards traffic to port 80 on the selected pods

- It uses TCP protocol for communication
- This Service allows other resources within the cluster to access the selected pods using a stable network endpoint

## Resiliency issues/weaknesses in the manifests:

### Issue #0: Single Pod Deployment

- details: Using a single Pod without a Deployment or ReplicaSet can lead to downtime if the Pod fails or the node it's running on becomes unavailable.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: Kind: Pod is used instead of a Deployment or ReplicaSet

### Issue #1: Restart Policy Set to Never

- details: With restartPolicy set to Never, the Pod will not be restarted if it crashes or terminates, leading to potential downtime.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: spec.restartPolicy: Never

### Issue #2: No Resource Limits

- details: Without resource limits, the container could consume excessive CPU or memory, potentially affecting other workloads on the node or causing OOM kills.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: No resources.limits specified in the container spec

### Issue #3: No Liveness or Readiness Probes

- details: Without health checks, Kubernetes cannot determine if the container is functioning correctly, potentially leading to serving traffic to a malfunctioning Pod.
- manifests having the issues: ['nginx/pod.yaml']
- problematic config: No livenessProbe or readinessProbe specified in the container spec

### Issue #4: No Service Type Specified

- details: Without a specified service type, the default ClusterIP is used, which may not be suitable if external access is required.
- manifests having the issues: ['nginx/service.yaml']
- problematic config: No spec.type specified in the Service

### Issue #5: No Session Affinity

- details: Without session affinity, client requests may be distributed across different Pods, potentially causing issues for stateful applications.
- manifests having the issues: ['nginx/service.yaml']
- problematic config: No spec.sessionAffinity specified in the Service

## Application of the manifests:

### Thoughts:

Based on the provided manifests and their file names (nginx/pod.yaml and nginx/service.yaml), it's clear that this application is using Nginx. The pod specification includes an Nginx container image, and the service is set up to expose port 80, which is the default HTTP port. This strongly suggests that the application is a web server or web application using Nginx as the web server or reverse proxy.

### Assumed application:

This appears to be a simple web server or web application using Nginx. The application offers HTTP services to users, likely serving static content or acting as a reverse proxy for other backend services. It provides a basic web hosting environment that can be easily scaled and managed within a Kubernetes cluster.

## Summary of your instructions for Chaos Engineering:

- The Chaos-Engineering experiment must be completed within 1 minute. (This instruction is relevant to the Experiment phase, where the time constraint for injecting failures and monitoring the system's response is specified.)

## Phase 1: Hypothesis

### Steady-state definition

#### ✓ Steady state #1: ExamplePodRunningAndReady

##### Description

Given the system's configuration and the identified issues, the most critical steady state to define first would be related to the availability of the single Pod. Since there's only one Pod without any replication, and its restart policy is set to 'Never', the system's availability is highly vulnerable. Therefore, we should define a steady state that checks if the Pod is running and ready to serve traffic.

##### Current state inspection

To inspect the 'ExamplePodRunningAndReady' state, we need to check if the 'example-pod' is running and ready. Since this involves checking the status of a Kubernetes resource, we'll use the Kubernetes API. We'll create a Python script that uses the Kubernetes client library to query the pod's status every second for a short duration. This will give us a good snapshot of the pod's state without exceeding the 1-minute time constraint for the entire experiment.

tool: `k8s` duration: `5s`

Inspection script (Python) `k8s_ExamplePodRunningAndReady.py`:

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

def check_pod_status(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()
    pod_name = "example-pod"
    namespace = "default"

    start_time = time.time()
    end_time = start_time + duration
    status_counts = {"Running": 0, "Not Running": 0, "Ready": 0, "Not Ready": 0}
```

```

while time.time() < end_time:
    try:
        pod = v1.read_namespaced_pod(name=pod_name, namespace=namespace)
        pod_status = pod.status.phase
        if pod_status == "Running":
            status_counts["Running"] += 1
        else:
            status_counts["Not Running"] += 1

        if pod.status.conditions:
            ready_condition = next((c for c in pod.status.conditions if c.type == "Ready"), None)
            if ready_condition and ready_condition.status == "True":
                status_counts["Ready"] += 1
            else:
                status_counts["Not Ready"] += 1
        else:
            status_counts["Not Ready"] += 1

    except client.exceptions.ApiException as e:
        print(f"Exception when calling CoreV1Api->read_namespaced_pod_status: {e}")
        status_counts["Not Running"] += 1
        status_counts["Not Ready"] += 1

    time.sleep(1)

total_checks = sum(status_counts.values()) // 2 # Divide by 2 to get average
print(f"Pod status over {duration} seconds:")
print(f"Running: {status_counts['Running']/total_checks*100:.2f}% of the time")
print(f"Ready: {status_counts['Ready']/total_checks*100:.2f}% of the time")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--duration', type=int, default=5, help='Duration in seconds')
    args = parser.parse_args()
    check_pod_status(args.duration)

```

Pod `k8sexamplepodrunningandready-pod` has completed successfully.

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

```

Pod status over 5 seconds:
Running: 100.00% of the time
Ready: 100.00% of the time

```

## Threshold

Given the current state of the system and the identified issues, we need to define a threshold for the ExamplePodRunningAndReady steady state that accounts for the single Pod deployment and its vulnerability to failures. The current state shows that the Pod is running and ready 100% of the time over a 5-second period. However, we should allow for some fluctuations and potential brief periods of unavailability, considering the lack of redundancy and the 'Never' restart policy.

Given that:

1. There's only one Pod without replication.

2. The restart policy is set to 'Never', which means the Pod won't automatically recover from failures.
3. The Chaos Engineering experiment must be completed within 1 minute.

We should set a threshold that allows for some downtime but ensures the Pod is running and ready for the majority of the time. A reasonable threshold would be to require the Pod to be running and ready for at least 90% of the time over a 1-minute period. This allows for up to 6 seconds of downtime, which could account for brief network issues or other transient problems.

The example-pod must be in a Running state and Ready condition for at least 90% of the time over a 1-minute monitoring period.

## Unit test to validate the steady state

To add the threshold assertion to the inspection Python script, I will modify the existing code to create a unit test that inherits from the K8sAPIBase class. The test will run for a specified duration (default 60 seconds) and check if the example-pod is in a Running state and Ready condition for at least 90% of the time. I'll use argparse to allow users to specify a custom duration. The test will collect status information every second, calculate the percentages, and assert that they meet or exceed the 90% threshold.

```
import argparse
import time
from unittest_base import K8sAPIBase

class TestPodStatus(K8sAPIBase):
    def test_pod_running_and_ready(self, duration):
        # Define the pod details
        pod_name = "example-pod"
        namespace = "default"

        # Define the threshold (90%)
        threshold = 0.9

        # Initialize counters
        status_counts = {"Running": 0, "Ready": 0}
        total_checks = 0

        # Start the test loop
        start_time = time.time()
        end_time = start_time + duration

        while time.time() < end_time:
            try:
                # Get the pod status
                pod = self.v1.read_namespaced_pod(name=pod_name, namespace=namespace)

                # Check if the pod is running
                if pod.status.phase == "Running":
                    status_counts["Running"] += 1

                # Check if the pod is ready
                if pod.status.conditions:
                    ready_condition = next((c for c in pod.status.conditions if c.type == "Ready"), None)
                    if ready_condition and ready_condition.status == "True":
                        status_counts["Ready"] += 1

            except Exception as e:
                # Log the error and continue
                print(f"Error: {e}")

        # Calculate the percentages
        running_percent = status_counts["Running"] / total_checks
        ready_percent = status_counts["Ready"] / total_checks

        # Assert that the percentages meet or exceed the threshold
        assert running_percent >= threshold, f"Running threshold not met: {running_percent}"
        assert ready_percent >= threshold, f"Ready threshold not met: {ready_percent}"
```

```

total_checks += 1

except Exception as e:
    print(f"Error occurred while checking pod status: {e}")

    # Wait for 1 second before the next check
    time.sleep(1)

# Calculate percentages
running_percentage = status_counts["Running"] / total_checks
ready_percentage = status_counts["Ready"] / total_checks

# Print summary
print(f"Pod status over {duration} seconds:")
print(f"Running: {running_percentage:.2%} of the time")
print(f"Ready: {ready_percentage:.2%} of the time")

# Assert that the percentages meet or exceed the threshold
assert running_percentage >= threshold, f"Pod was running {running_percentage:.2%} of the time, which is below the threshold of {threshold:.2%}"
assert ready_percentage >= threshold, f"Pod was ready {ready_percentage:.2%} of the time, which is below the threshold of {threshold:.2%}"

print("Test passed: Pod was running and ready above the threshold")

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--duration', type=int, default=60, help='Duration in seconds')
    args = parser.parse_args()

    test = TestPodStatus()
    test.test_pod_running_and_ready(args.duration)

if __name__ == '__main__':
    main()

```

Pod `unittestexamplepodrunningandreadymod0-pod` has completed successfully.

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

```

Pod status over 5 seconds:
Running: 100.00% of the time
Ready: 100.00% of the time
Test passed: Pod was running and ready above the threshold

```

## Steady state #2: ExampleServiceEndpointAvailable

### Description

Given the existing steady state and the system's configuration, the next most critical aspect to monitor would be the Service's ability to route traffic to the Pod. Since we have a single Pod and a Service that's supposed to direct traffic to it, we should ensure that the Service is correctly selecting and routing traffic to the Pod. This is particularly important because if the Service fails to route traffic, even if the Pod is running, the application would be inaccessible. Therefore, we'll define a steady state for the Service's endpoint availability.

## Current state inspection

To inspect the `ExampleServiceEndpointAvailable` state, we need to check if the Service is correctly routing traffic to the Pod. Since this involves testing network communication, we'll use k6 to send HTTP requests to the Service endpoint and verify its availability. We'll use the internal DNS name of the Service to send requests, as this is how other components within the cluster would access it.

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

Inspection script (Javascript) `k6_ExampleServiceEndpointAvailable.js` :

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

export const options = {
  vus: 1,
  duration: '5s',
  thresholds: {
    'http_req_failed': ['rate<0.01'], // Less than 1% of requests s
    'http_req_duration': ['p(95)<500'] // 95% of requests should be
  }
};

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

Pod `k6exampleserviceendpointavailable-pod` has completed sucessfully.

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

```
✓ status is 200
✓ body contains nginx

checks.....: 100.00% 10 out of 10
data_received.....: 4.3 kB   848 B/s
data_sent.....: 550 B    110 B/s
http_req_blocked.....: avg=608.17µs min=4.08µs  med=7.8
http_req_connecting.....: avg=52.46µs  min=0s      med=0s
✓ http_req_duration.....: avg=601.43µs min=414.1µs med=588
  { expected_response:true }...: avg=601.43µs min=414.1µs med=588
✓ http_req_failed.....: 0.00%    0 out of 5
http_req_receiving.....: avg=119.66µs min=62.64µs med=77.
http_req_sending.....: avg=62.36µs  min=11.99µs med=26.
http_req_tls_handshaking.....: avg=0s      min=0s      med=0s
http_req_waiting.....: avg=419.39µs min=281.12µs med=484
http_reqs.....: 5      0.997875/s
iteration_duration.....: avg=1s      min=1s      med=1s
iterations.....: 5      0.997875/s
vus.....: 1      min=1      max=1
vus_max.....: 1      min=1      max=1
```



## ▶ Threshold

Based on the current state and the system configuration, we need to define a threshold for the `ExampleServiceEndpointAvailable` steady state. The k6 test results show that the service is currently performing well, with 100% of checks passing, 0% failed requests, and response times well under 500ms. However, we should allow for some fluctuations in performance while still maintaining a high standard of availability and responsiveness.

Considering that:

1. The system has only one pod, making it more susceptible to downtime.
2. The restart policy is set to 'Never', which could lead to extended downtime if the pod fails.
3. There are no resource limits or health checks in place.

We should set a threshold that is slightly more lenient than the current perfect performance, but still ensures high availability and responsiveness.

The `ExampleServiceEndpointAvailable` steady state is considered maintained if:

1. At least 99% of requests return a 200 status code.
2. At least 99% of response bodies contain the word 'nginx'.
3. The 95th percentile of response times is below 750ms.
4. Less than 1% of requests fail.
5. These conditions must be met over a 1-minute rolling window, with checks performed every 10 seconds.

## 📄 Unit test to validate the steady state

To verify if the steady state satisfies the threshold, we need to modify the existing k6 script by adding new thresholds and adjusting the test duration. We'll add thresholds for the success rate of status checks, the presence of 'nginx' in the response body, and adjust the existing thresholds for response time and failed requests. We'll also increase the test duration to 1 minute to match the specified rolling window, and set the iteration interval to 10 seconds.

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

export const options = {
  vus: 1,
  duration: '1m',
  iterations: 6, // Run every 10 seconds for 1 minute
  thresholds: {
    // Check if at least 99% of requests return a 200 status code
    "checks{status is 200}": ["rate>0.99"],

    // Check if at least 99% of response bodies contain the word 'nginx'
    "checks{body contains nginx}": ["rate>0.99"],

    // Check if the 95th percentile of response times is below 750ms
    "http_req_duration": ["p(95)<750"],

    // Check if less than 1% of requests fail
    "http_req_failed": ["rate<0.01"]
  }
}
```

```

    }
};

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

    check(res, {
        'status is 200': (r) => r.status === 200,
        'body contains nginx': (r) => r.body.includes('nginx')
    });

    sleep(10); // Sleep for 10 seconds between iterations
}

```

Pod `unittestexampleserviceendpointavailablemod0-pod` has failed.

```
time="2024-11-28T07:54:07Z" level=error msg="unable to validate threshold expression: 'body contains nginx'"
```

The error occurs because the threshold expression for the 'body contains nginx' check is not correctly formatted. In k6, when referencing check metrics in thresholds, we need to use square brackets and quotes for the check name. Also, we should remove the fixed number of iterations to support variable durations. Here's how we can fix these issues:

```

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

export const options = {
    vus: 1,
    duration: '1m',
    thresholds: {
        // Check if at least 99% of requests return a 200 status code
        "checks['status is 200']": ["rate>0.99"],

        // Check if at least 99% of response bodies contain the word 'nginx'
        "checks['body contains nginx']": ["rate>0.99"],

        // Check if the 95th percentile of response times is below 750ms
        "http_req_duration": ["p(95)<750"],

        // Check if less than 1% of requests fail
        "http_req_failed": ["rate<0.01"]
    }
};

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

    check(res, {
        'status is 200': (r) => r.status === 200,
        'body contains nginx': (r) => r.body.includes('nginx')
    });

    sleep(10); // Sleep for 10 seconds between iterations
}

```

Pod `unittestexampleserviceendpointavailablemod1-pod` has failed.

```
time="2024-11-28T07:54:24Z" level=error msg="invalid threshold defin
```

The error occurs because k6 doesn't recognize the check names with spaces when defined in the thresholds. To fix this, we need to use the exact metric names that k6 generates for checks. These are in the format 'checks{checkName}'. We'll update the threshold definitions to use this format.

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

export const options = {
  vus: 1,
  duration: '1m',
  thresholds: {
    // Check if at least 99% of requests return a 200 status code
    "checks{status is 200}": ["rate>0.99"],

    // Check if at least 99% of response bodies contain the word 'ng
    "checks{body contains nginx}": ["rate>0.99"],

    // Check if the 95th percentile of response times is below 750ms
    "http_req_duration": ["p(95)<750"],

    // Check if less than 1% of requests fail
    "http_req_failed": ["rate<0.01"]
  }
};

export default function () {
  const res = http.get('http://example-service.default.svc.cluster.1

  check(res, {
    'status is 200': (r) => r.status === 200,
    'body contains nginx': (r) => r.body.includes('nginx')
  });

  sleep(10); // Sleep for 10 seconds between iterations
}
```

Pod `unittestexampleserviceendpointavailablemod2-pod` has failed.

```
time="2024-11-28T07:54:38Z" level=error msg="unable to validate thre
```

The error persists because the check names contain spaces, which are not allowed in k6 metric names. To resolve this, we need to modify the check names to remove spaces and use camelCase or snake\_case. We'll update both the check definitions and the corresponding threshold definitions to use consistent, space-free names.

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

export const options = {
  vus: 1,
  duration: '1m',
  thresholds: {
    // Check if at least 99% of requests return a 200 status code
```

```

    "checks{statusIs200}": ["rate>0.99"],

    // Check if at least 99% of response bodies contain the word 'nginx'
    "checks{bodyContainsNginx}": ["rate>0.99"],

    // Check if the 95th percentile of response times is below 750ms
    "http_req_duration": ["p(95)<750"],

    // Check if less than 1% of requests fail
    "http_req_failed": ["rate<0.01"]
  }
};

export default function () {
  const res = http.get('http://example-service.default.svc.cluster.local')

  check(res, {
    statusIs200: (r) => r.status === 200,
    bodyContainsNginx: (r) => r.body.includes('nginx')
  });

  sleep(10); // Sleep for 10 seconds between iterations
}

```

Pod `unittestexample-service-endpoint-available-mod3-pod` has failed.

```
time="2024-11-28T07:54:54Z" level=error msg="Unable to validate through endpoint"
```

AssertionError: MAX\_MOD\_COUNT\_EXCEEDED: 3

Traceback:

```

File "/opt/conda/lib/python3.10/site-packages/streamlit/runtime/scriptrunner/script_exec_util.py", line 114, in
    exec(code, module.__dict__)
File "/workspace/app/projects/chaos-eater/ChaosEater_demo.py", line 431, in <module>
    main()
File "/workspace/app/projects/chaos-eater/ChaosEater_demo.py", line 397, in main
    output = st.session_state.chaoseater.run_ce_cycle()
File "/workspace/app/projects/chaos-eater/chaos_eater/chaos_eater.py", line 14, in run_ce_cycle
    hypothesis_logs, hypothesis = self.hypothesizer.hypothesize()
File "/workspace/app/projects/chaos-eater/chaos_eater/hypothesis/hypothesizer.py", line 14, in hypothesize
    steady_state_logs, steady_states = self.steady_state_definer.define_steady_state()
File "/workspace/app/projects/chaos-eater/chaos_eater/hypothesis/steady_state_definer.py", line 14, in define_steady_state
    unittest_log, unittest = self.unittest_agent.write_unittest()
File "/workspace/app/projects/chaos-eater/chaos_eater/hypothesis/llm_agents/steady_state_definer.py", line 14, in write_unittest
    assert mod_count < max_mod_loop, f"MAX_MOD_COUNT_EXCEEDED: {max_mod_loop}"

```