



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: 3.995k

Input tokens: 2.992k

Ouput tokens: 1.003k

Command history

kubectl apply -f sandbox/cycle_20241128_075739/hypothesis/k8s_ExamplePodRunningAndReady_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-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
```

Input instructions for your Chaos Engineering



```
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 71ns
- pod/example-pod created
- service/example-service created
Waiting for deployments to stabilize...
- pods is ready.
Deployments stabilized in 3.09 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	10.96.198.17	<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 named 'example-container'
- The container 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 the service name

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 manifest specifies an Nginx container image, and the Service manifest is designed to expose this Pod. Nginx is commonly used as a web server, reverse proxy, or load balancer. Given the simple configuration and the use of port 80, which is the standard HTTP port, it's reasonable to assume this is a basic web server setup.

Assumed application:

This appears to be a simple web server application using Nginx. The application serves static content or acts as a reverse proxy for other backend services over HTTP on port 80. It provides

basic web hosting capabilities, potentially for a small website or as a frontend for a larger application architecture.

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 system behavior 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=ns)
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
print(f"Pod status over {duration} seconds:")
print(f"Running: {status_counts['Running']}/{total_checks} checks")
print(f"Ready: {status_counts['Ready']}/{total_checks} checks")

if status_counts['Running'] == total_checks and status_counts['Ready'] == total_checks:
    print("Steady state hypothesis: Pod is consistently running and ready.")
else:
    print("Steady state hypothesis: Pod is not consistently running and ready.")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--duration', type=int, default=5, help='Duration of the check 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: 5/5 checks
Ready: 5/5 checks
Steady state hypothesis: Pod is consistently running and ready.

```

▶ Threshold

Given the current state of the system and the nature of the steady state we're monitoring (ExamplePodRunningAndReady), we need to define a threshold that ensures the pod is consistently running and ready, while allowing for some minor fluctuations that might occur during normal operation or during a chaos engineering experiment.

The current state shows that the pod was running and ready for 5 out of 5 checks over a 5-second period. This indicates perfect health, but we should allow for some potential

brief interruptions or fluctuations.

Considering that:

1. We only have a single pod without any replication.
2. The restart policy is set to 'Never', which means any downtime could be critical.
3. The chaos engineering experiment must be completed within 1 minute.

We should set a threshold that allows for brief interruptions but ensures the pod is running and ready for the vast majority of the time. A reasonable threshold would be to require the pod to be running and ready for at least 90% of the checks over a 30-second period. This allows for some brief fluctuations while still maintaining a high standard of availability.

The pod must be in a 'Running' state and 'Ready' condition for at least 90% of the checks over a 30-second monitoring period.

Unit test to validate the steady state

To add the threshold assertion to the inspection Python script, I'll modify the existing code to create a unit test that inherits from the K8sAPIBase class. The test will run for a specified duration (default 30 seconds) and check if the pod is running and ready for at least 90% of the checks. I'll use argparse to allow users to specify a custom duration. The test will assert that the running and ready percentages meet or exceed the 90% threshold.

```
import argparse
import time
from unittest_base import K8sAPIBase

class TestPodRunningAndReady(K8sAPIBase):
    def test_pod_status(self, duration):
        pod_name = "example-pod"
        namespace = "default"
        threshold = 0.90 # 90% threshold

        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 = self.v1.read_namespaced_pod(name=pod_name, namespace=namespace)
                pod_status = pod.status.phase
                if pod_status == "Running":
                    status_counts["Running"] += 1
            except:
                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 Exception as e:
    print(f"Exception when calling CoreV1Api->read_namespaces")
    status_counts["Not Running"] += 1
    status_counts["Not Ready"] += 1

time.sleep(1)

total_checks = sum(status_counts.values()) // 2
running_percentage = status_counts["Running"] / total_checks
ready_percentage = status_counts["Ready"] / total_checks

print(f"Pod status over {duration} seconds:")
print(f"Running: {status_counts['Running']}/{total_checks} checks")
print(f"Ready: {status_counts['Ready']}/{total_checks} checks")

# Assert that both running and ready percentages meet or exceed threshold
assert running_percentage >= threshold and ready_percentage >= threshold

```

TypeError: write() argument must be str, not None

Traceback:

```

File "/opt/conda/lib/python3.10/site-packages/streamlit/runtime/scriptrunner/script_runner.py", line 450, 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
    write_file(file_path, unittest["code"])
File "/workspace/app/projects/chaos-eater/chaos_eater/utils/functions.py", line 14, in write_file
    f.write(content)

```