



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: \$0.45

Total tokens: 93.765k

Input tokens: 84.774k

Output tokens: 8.991k

Command history

kubectl apply -f



sandbox/cycle_20241128_080427/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 241ns
- pod/example-pod created
- service/example-service created
Waiting for deployments to stabilize...
- pods is ready.
Deployments stabilized in 2.088 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	4s

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

    running_count = 0
    ready_count = 0
    total_checks = 0

    while time.time() < end_time:
        try:
            pod = v1.read_namespaced_pod(name=pod_name, namespace=namespace)
            if pod.status.phase == 'Running':
                running_count += 1
            if pod.status.conditions:
                for condition in pod.status.conditions:
                    if condition.type == 'Ready' and condition.status == 'True':
                        ready_count += 1
            total_checks += 1
        except Exception as e:
            print(f"Error checking pod status: {e}")
```

```

        if pod.status.phase == "Running":
            running_count += 1

    if pod.status.conditions:
        ready_condition = next((c for c in pod.status.conditions
                                if ready_condition and ready_condition.status == "True"), None)
        ready_count += 1

    print(f"Pod status: Phase={pod.status.phase}, Ready={ready_count}")
except client.exceptions.ApiException as e:
    print(f"Exception when calling CoreV1Api->read_namespaces")

time.sleep(1)

print(f"\nSummary:")
print(f"Total checks: {total_checks}")
print(f"Running: {running_count}/{total_checks} ({running_count/total_checks*100:.00%})")
print(f"Ready: {ready_count}/{total_checks} ({ready_count/total_checks*100:.00%})")
print(f"Pod is considered stable: {running_count == total_checks}")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--duration', type=int, default=5, help='Duration of the script')
    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: Phase=Running, Ready=True
Pod status: Phase=Running, Ready=True
Pod status: Phase=Running, Ready=True
Pod status: Phase=Running, Ready=True
Pod status: Phase=Running, Ready=True

```

```

Summary:
Total checks: 5
Running: 5/5 (100.00%)
Ready: 5/5 (100.00%)
Pod is considered stable: True

```

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 running and ready most of the time, while allowing for some brief fluctuations that might occur during normal operation or minor disturbances.

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 tolerance to account for potential brief interruptions or delays in readiness that might occur during normal operation or minor disturbances.

Considering that:

1. This is a single pod without any replication.

2. The restart policy is set to 'Never', which means any failure could lead to extended downtime.
3. The Chaos Engineering experiment must be completed within 1 minute.

We should set a threshold that is strict enough to catch significant issues quickly, but not so strict that it triggers false alarms for minor, temporary fluctuations.

A reasonable threshold could be:

- The pod should be in the 'Running' state and 'Ready' condition for at least 95% of the checks over a 30-second period.

This allows for a brief period of unavailability (up to 1.5 seconds out of 30) which could occur due to things like network blips or brief resource contention, while still catching any significant issues quickly within the 1-minute experiment window.

The example-pod should be in the 'Running' state and 'Ready' condition for at least 95% of the checks over a 30-second 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 in the 'Running' state and 'Ready' condition for at least 95% of the checks. I'll use argparse to allow users to specify the duration. The test will assert that the percentage of checks where the pod is running and ready is greater than or equal to 95%.

```
import argparse
import time
from unittest_base import K8sAPIBase

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

        # Initialize counters
        running_count = 0
        ready_count = 0
        total_checks = 0

        # Set the threshold (95%)
        threshold = 0.95

        # 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)
                total_checks += 1

                # Check if the pod is running
                if pod.status.phase == "Running":
                    running_count += 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':
                ready_count += 1

        print(f"Pod status: Phase={pod.status.phase}, Ready={ready_count}/{total_checks}")
    except Exception as e:
        print(f"Error checking pod status: {e}")

    time.sleep(1)

# Calculate the percentages
running_percentage = running_count / total_checks
ready_percentage = ready_count / total_checks

# Print the summary
print(f"\nSummary:")
print(f"Total checks: {total_checks}")
print(f"Running: {running_count}/{total_checks} ({running_percentage*100:.0f}%)")
print(f"Ready: {ready_count}/{total_checks} ({ready_percentage*100:.0f}%)")

# Assert that the percentages meet the threshold
assert running_percentage >= threshold, f"Pod was not running for at least {threshold*100:.0f}% of the checks"
assert ready_percentage >= threshold, f"Pod was not ready for at least {threshold*100:.0f}% of the checks"

print(f"Test passed: Pod was running and ready for at least {threshold*100:.0f}% of the checks")

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--duration', type=int, default=30, help='Duration of the test in seconds')
    args = parser.parse_args()

    test = TestExamplePodRunningAndReady()
    test.test_pod_status(args.duration)

if __name__ == '__main__':
    main()

```

Pod `unittestexamplepodrunningandreadymod0-pod` has completed successfully.

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

```

Pod status: Phase=Running, Ready=True
Pod status: Phase=Running, Ready=True
Pod status: Phase=Running, Ready=True
Pod status: Phase=Running, Ready=True
Pod status: Phase=Running, Ready=True

Summary:
Total checks: 5
Running: 5/5 (100.00%)
Ready: 5/5 (100.00%)
Test passed: Pod was running and ready for at least 95% of the checks

```

Description

Given the system's configuration and the identified issues, the next critical steady state to define would be related to 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 need to ensure that the Service is correctly selecting the Pod and that network connectivity is maintained. This is particularly important given Issue #0 (Single Pod Deployment) and Issue #4 (No Service Type Specified), as any disruption in the Service-to-Pod connection would immediately impact the entire system's functionality.

Current state inspection

To inspect the ExampleServiceEndpointAvailability, we need to check if the Service is correctly routing traffic to the Pod. Since we're dealing with a web server on port 80, we can use k6 to send HTTP requests to the Service's endpoint and verify the responses. This will allow us to test both the Service's ability to select the Pod and the network connectivity. We'll use the internal DNS name of the Service to send requests.

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

Inspection script (Javascript) `k6_ExampleServiceEndpointAvailability.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 `k6exampleserviceendpointavailability-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=431.39µs min=10.89µs med=16.
http_req_connecting.....: avg=40.15µs min=0s med=0s
✓ http_req_duration.....: avg=571.97µs min=505.49µs med=537
  { expected_response:true }...: avg=571.97µs min=505.49µs med=537
✓ http_req_failed.....: 0.00%  0 out of 5
http_req_receiving.....: avg=130.74µs min=91.11µs med=105
```



```
http_req_sending.....: avg=61.23µs  min=39.35µs  med=45.1µs
http_req_tls_handshaking.....: avg=0s      min=0s      med=0s
http_req_waiting.....: avg=379.98µs min=314.11µs med=392.1µs
http_reqs.....: 5      0.997869/s
iteration_duration.....: avg=1s      min=1s      med=1s
iterations.....: 5      0.997869/s
vus.....: 1      min=1      max=1
vus_max.....: 1      min=1      max=1
```

▶ Threshold

Given the current state of the system and the k6 test results, we can define a reasonable threshold for the ExampleServiceEndpointAvailability steady state. The current test shows 100% success rate for checks, 0% HTTP request failures, and all requests completing within 726.62µs. However, we should allow for some fluctuations and set a threshold that's slightly more lenient than the current perfect state.

Considering the single Pod deployment and potential issues identified, we should expect some occasional failures or slower responses. We'll set a threshold that allows for a small percentage of failures and slightly longer response times, while still maintaining a high level of service availability.

For the threshold, we'll consider:

1. Success rate of status checks
2. HTTP request failure rate
3. Response time

We'll set these slightly below the current perfect state to account for normal fluctuations and potential brief disruptions.

The ExampleServiceEndpointAvailability steady state is considered maintained if:

1. At least 98% of status checks return 200 OK over a 1-minute period.
2. HTTP request failure rate is below 2% over a 1-minute period.
3. 95% of requests complete within 1000ms (1 second) over a 1-minute period. All three conditions must be met simultaneously for the steady state to be considered maintained.

📄 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 threshold options that match the defined criteria. We'll adjust the test duration to 1 minute to align with the threshold requirements and add new thresholds for status check success rate, HTTP request failure rate, and response time. Here's how we'll modify the script:

1. Change the duration to '1m' to match the 1-minute period specified in the threshold.
2. Add a new threshold for the status check success rate (98% or higher).
3. Adjust the existing http_req_failed threshold to 2% (0.02 rate).
4. Update the http_req_duration threshold to 1000ms for 95% of requests.
5. Keep the existing checks for status 200 and body content.

These modifications will ensure that the k6 test accurately verifies if the ExampleServiceEndpointAvailability steady state satisfies the given threshold.

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

export const options = {
  vus: 1,
  duration: '1m', // Changed to 1 minute to match threshold period
  thresholds: {
    // New threshold for status check success rate
    'checks': ['rate>0.98'], // At least 98% of status checks should pass
    'http_req_failed': ['rate<0.02'], // Less than 2% of requests should fail
    'http_req_duration': ['p(95)<1000'] // 95% of requests should be under 1000ms
  }
};

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

  // These checks contribute to the 'checks' threshold
  check(res, {
    'status is 200': (r) => r.status === 200,
    'body contains nginx': (r) => r.body.includes('nginx')
  });

  sleep(1);
}
```

Pod `unittestexampleserviceendpointavailabilitymod0-pod` has completed successfully.

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=458.04µs min=7.22µs   med=12.1µs
  http_req_connecting.....: avg=40.3µs   min=0s       med=0s
✓ http_req_duration.....: avg=547.87µs min=433.98µs med=585µs
  { expected_response:true }...: avg=547.87µs min=433.98µs med=585µs
✓ http_req_failed.....: 0.00%   0 out of 5
  http_req_receiving.....: avg=120.56µs min=62.88µs   med=121µs
  http_req_sending.....: avg=72.54µs  min=25.05µs   med=42.1µs
  http_req_tls_handshaking.....: avg=0s       min=0s       med=0s
  http_req_waiting.....: avg=354.76µs min=257.31µs med=346µs
  http_reqs.....: 5          0.99796/s
  iteration_duration.....: avg=1s       min=1s       med=1s
  iterations.....: 5          0.99796/s
  vus.....: 1          min=1          max=1
  vus_max.....: 1          min=1          max=1
```

Fault definition

Description

Given the system's vulnerabilities, particularly the single Pod deployment and lack of resource limits, a sudden traffic surge would be highly impactful. This event could expose multiple weaknesses:

1. The single Pod (Issue #0) may become overwhelmed, potentially causing it to crash or become unresponsive.
2. Without resource limits (Issue #2), the Pod might consume excessive resources, affecting other workloads or causing OOM kills.
3. The lack of liveness and readiness probes (Issue #3) means Kubernetes won't be able to detect if the Pod becomes unresponsive under load.
4. The 'Never' restart policy (Issue #1) means that if the Pod does crash, it won't automatically recover.

To simulate this event and exploit these weaknesses, we'll use a sequence of fault injections:

1. First, we'll inject CPU stress to simulate the increased load from the traffic surge. This targets the lack of resource limits and the single Pod deployment.
2. Then, we'll introduce network latency to simulate network congestion that might occur during a traffic spike. This further stresses the single Pod and may expose issues with the lack of health probes.
3. Finally, we'll kill the Pod to simulate a crash under heavy load. This exposes the vulnerability of the 'Never' restart policy and the lack of redundancy.

This sequence simulates the progression of a traffic surge event, from initial load increase to potential system failure.

Fault-injection sequence

```
StressChaos ({'namespace': 'default', 'labels': {'app': 'example'}}) → NetworkChaos ({'namespace': 'default', 'labels': {'app': 'example'}}) → PodChaos ({'namespace': 'default', 'labels': {'app': 'example'}})
```

Detailed fault parameters

Detailed parameters of `StressChaos` (`{'namespace': 'default', 'labels': {'app': 'example'}}`)

```
{
  "mode" : "one"
  "selector" : {
    "namespaces" : [
      0 : "default"
    ]
    "labelSelectors" : {
      "app" : "example"
    }
  }
  "stressors" : {
```

```

    "cpu" : {
      "workers" : 1
      "load" : 80
    }
    "memory" : {
      "workers" : 1
      "size" : "256MB"
    }
  }
  "containerNames" : [
    0 : "example-container"
  ]
  "value" : "1"
}

```

Detailed parameters of `NetworkChaos` ({'namespace': 'default', 'labels': {'app': 'example'}})

```

{
  "action" : "delay"
  "mode" : "all"
  "selector" : {
    "namespaces" : [
      0 : "default"
    ]
    "labelSelectors" : {
      "app" : "example"
    }
  }
  "delay" : {
    "latency" : "100ms"
    "correlation" : "25"
    "jitter" : "10ms"
  }
  "direction" : "to"
}

```

Detailed parameters of `PodChaos` ({'namespace': 'default', 'labels': {'app': 'example'}})

```

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

```

Phase 2: Chaos Experiment

CE experiment Planning Completed!

Chaos Engineering Experiment Plan

Time Schedule

Given the complexity of the fault scenario and the need to observe the system's behavior during and after fault injection, I'll allocate the time as follows:

1. Pre-validation phase (20 seconds): This allows enough time to ensure the system is in a steady state before introducing faults.
2. Fault injection phase (30 seconds): This gives us time to introduce the three faults (CPU stress, network latency, and pod kill) and observe their effects.
3. Post-validation phase (10 seconds): This provides time to check if the system returns to its steady state after the faults are removed.

The total experiment time is 60 seconds (1 minute), meeting the given time constraint. This allocation allows for sufficient time in each phase while emphasizing the fault injection phase, which is the core of the experiment.

Total experiment time: 60s

Pre-validation Phase: 20s

Fault-injection Phase: 30s

Post-validation Phase: 10s

Pre-validation Phase (20s)

For the pre-validation phase, we need to ensure that both steady states are verified before proceeding with fault injection. Given the 20-second time constraint for this phase, we'll need to schedule our unit tests efficiently.

1. ExamplePodRunningAndReady: This test requires a 30-second duration to meet its defined threshold. However, given our 20-second constraint, we'll run it for the full 20 seconds available. This should still provide a good indication of the pod's status, even if it's slightly shorter than the ideal duration.
2. ExampleServiceEndpointAvailability: This test is designed to run for 1 minute, but we'll need to adjust it to fit within our 20-second window. We'll run this test concurrently with the first test to maximize our use of the available time.

By running both tests simultaneously for the full 20 seconds, we can get the most comprehensive view of our system's steady state within the given time constraint. This approach allows us to verify both the pod's status and the service's availability in parallel, making the most efficient use of our limited time.

- Verified Steady State #0: ExamplePodRunningAndReady
 - Workflow Name: pre-unittest-examplepodrunningandready
 - Grace Period: 0s
 - Duration: 20s
- Verified Steady State #1: ExampleServiceEndpointAvailability

- Workflow Name: `pre-unittest-example-service-endpoint-availability`
- Grace Period: `0s`
- Duration: `20s`

Fault-injection Phase (30s)

Given the 30-second time constraint for the fault-injection phase, we need to carefully schedule our fault injections and unit tests to maximize our observations while ensuring we capture the system's behavior under stress. Here's the approach:

1. We'll start with the StressChaos injection immediately at the beginning of the phase to simulate the sudden traffic surge. This will run for the full 30 seconds.
2. After a 5-second grace period to allow the stress to take effect, we'll introduce the NetworkChaos to simulate network congestion. This will run for 20 seconds.
3. At the 25-second mark, we'll execute the PodChaos to kill the pod, simulating a crash under heavy load.
4. We'll run our first unit test (ExamplePodRunningAndReady) at the 10-second mark for 10 seconds, which should capture the pod's state under stress and network congestion.
5. We'll run our second unit test (ExampleServiceEndpointAvailability) at the 20-second mark for 10 seconds, which should capture the service's behavior under all three fault conditions.

This timeline allows us to observe the system's behavior under progressively worsening conditions and should provide insights into how our steady states are affected by the simulated traffic surge.

- Verified Steady State #0: `ExamplePodRunningAndReady`
 - Workflow Name: `fault-unittest-example-pod-running-and-ready`
 - Grace Period: `10s`
 - Duration: `10s`
- Verified Steady State #1: `ExampleServiceEndpointAvailability`
 - Workflow Name: `fault-unittest-example-service-endpoint-availability`
 - Grace Period: `20s`
 - Duration: `10s`
- Injected Faults #0: `StressChaos`
 - Workflow Name: `fault-stresschaos`
 - Grace Period: `0s`
 - Duration: `30s`
- Injected Faults #1: `NetworkChaos`
 - Workflow Name: `fault-networkchaos`
 - Grace Period: `5s`
 - Duration: `20s`
- Injected Faults #2: `PodChaos`
 - Workflow Name: `fault-podchaos`
 - Grace Period: `25s`
 - Duration: `5s`

Post-validation Phase (10s)

For the post-validation phase, we need to verify that the system has returned to its steady states after the fault injections. Given the 10-second time constraint for this phase, we need to efficiently schedule our unit tests to maximize coverage while ensuring we capture the system's recovery.

We'll run both unit tests concurrently to make the most of our limited time. We'll start both tests immediately (grace_period: '0s') to capture the system's state as soon as possible after the fault injections. We'll run each test for 9 seconds, leaving a 1-second buffer at the end of the phase.

This approach allows us to:

1. Start monitoring the system's recovery immediately.
2. Run both tests for a significant portion of the available time.
3. Capture data throughout most of the post-validation phase.
4. Leave a small buffer at the end to ensure we don't exceed the phase's total time.

By running both tests concurrently, we can observe how both the Pod status and the Service endpoint availability recover in parallel, giving us a comprehensive view of the system's return to steady state.

- Verified Steady State #0: `ExamplePodRunningAndReady`
 - Workflow Name: `post-unittest-examplepodrunningandready`
 - Grace Period: `0s`
 - Duration: `9s`
- Verified Steady State #1: `ExampleServiceEndpointAvailability`
 - Workflow Name: `post-unittest-exampleserviceendpointavailability`
 - Grace Period: `0s`
 - Duration: `9s`

Summary

The Chaos Engineering experiment plan is divided into three phases: pre-validation (20 seconds), fault injection (30 seconds), and post-validation (10 seconds), totaling 60 seconds.

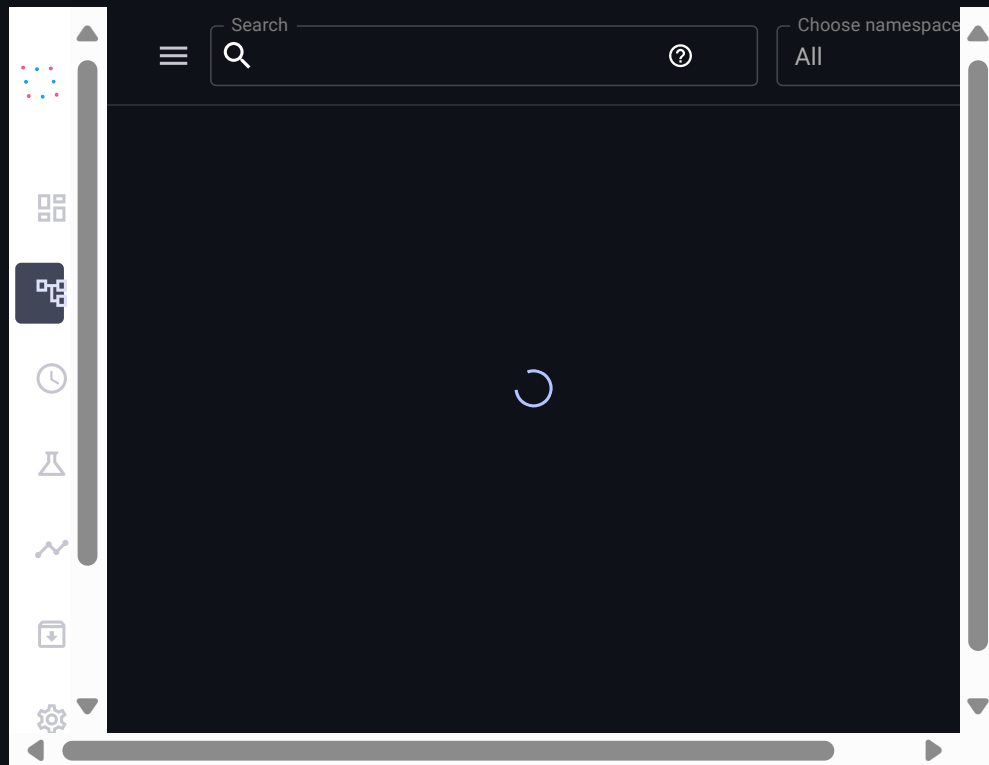
In the pre-validation phase, two unit tests are executed simultaneously for the entire 20-second duration. The 'ExamplePodRunningAndReady' test (workflow: 'pre-unittest-examplepodrunningandready') and the 'ExampleServiceEndpointAvailability' test (workflow: 'pre-unittest-exampleserviceendpointavailability') both start immediately with no grace period.

The fault injection phase introduces three faults in a staggered manner. The 'StressChaos' fault (workflow: 'fault-stresschaos') begins immediately and runs for the full 30 seconds. After a 5-second delay, the 'NetworkChaos' fault (workflow: 'fault-networkchaos') is introduced and lasts for 20 seconds. At the 25-second mark, the 'PodChaos' fault (workflow: 'fault-podchaos') is triggered for the remaining 5 seconds. During this phase, two unit tests are also executed: 'ExamplePodRunningAndReady' (workflow: 'fault-unittest-examplepodrunningandready') runs from the 10-second mark for 10 seconds, and 'ExampleServiceEndpointAvailability' (workflow: 'fault-unittest-exampleserviceendpointavailability') runs from the 20-second mark for 10 seconds.

In the post-validation phase, both unit tests are again executed concurrently. The 'ExamplePodRunningAndReady' test (workflow: 'post-unittest-examplepodrunningandready') and the 'ExampleServiceEndpointAvailability' test (workflow: 'post-unittest-exampleserviceendpointavailability') start immediately with no grace period and run for 9 seconds, leaving a 1-second buffer at the end of the phase.

This plan allows for comprehensive testing of the system's steady state before and after fault injection, as well as its behavior under progressively worsening conditions during the fault injection phase.

Completed the chaos experiment!



Phase 3: Analysis

The Chaos Engineering experiment revealed significant vulnerabilities in the system's resilience. Let's analyze the results step by step:

1. Pre-validation phase: Both 'ExamplePodRunningAndReady' and 'ExampleServiceEndpointAvailability' tests passed, indicating that the system was initially in a stable state.
2. Fault injection phase: a) The 'ExamplePodRunningAndReady' test passed during fault injection, suggesting that the pod remained in a running state despite the introduced stressors. b) However, the 'ExampleServiceEndpointAvailability' test failed, indicating that the service became unavailable or unresponsive. The error log shows connection refusals and inability to read the response body, suggesting that the service was not accessible.
3. Post-validation phase: Both tests failed in this phase, revealing that the system did not recover after the fault injection: a) The 'ExamplePodRunningAndReady' test failed with a 404 error, indicating that the pod was not found. This suggests that the PodChaos fault (pod-kill) successfully terminated the pod, and due to the 'Never' restart policy (Issue #1), it was not automatically recreated. b) The 'ExampleServiceEndpointAvailability' test continued to fail, showing that the service remained unavailable even after the fault injection period.

Key insights:

1. **Single Point of Failure:** The system's reliance on a single pod (Issue #0) proved to be a critical weakness. When the pod was killed, the entire service became unavailable with no automatic recovery mechanism.
2. **Ineffective Restart Policy:** The 'Never' restart policy (Issue #1) prevented the system from recovering after the pod was killed. This policy should be reconsidered to improve resilience.
3. **Resource Constraints:** While the pod remained running during the StressChaos fault, the service became unresponsive. This suggests that the lack of resource limits (Issue #2) allowed the pod to consume excessive resources, impacting its ability to handle requests effectively.
4. **Lack of Health Checks:** The absence of liveness and readiness probes (Issue #3) meant that Kubernetes couldn't detect the pod's degraded state and take corrective action.
5. **Network Sensitivity:** The introduction of network latency through NetworkChaos likely contributed to the service unavailability, highlighting the system's sensitivity to network conditions.

To improve the system's resilience:

1. Implement a Deployment or ReplicaSet instead of a standalone Pod to ensure automatic pod replacement.
2. Change the restart policy to 'Always' to allow automatic restarts in case of failures.
3. Define appropriate resource limits and requests to prevent resource exhaustion.
4. Implement liveness and readiness probes to enable Kubernetes to detect and respond to pod health issues.
5. Consider implementing horizontal pod autoscaling to handle traffic surges more effectively.
6. Evaluate the network architecture and implement strategies to mitigate the impact of network latency, such as using a CDN or optimizing application code.

These changes would significantly enhance the system's ability to withstand and recover from the simulated fault scenarios, improving overall resilience and availability.

Phase 4: Improvement

Reconfiguration



Based on the analysis of the Chaos Engineering experiment results, we need to address several key issues to improve the system's resiliency. The main problems were the single point of failure, ineffective restart policy, lack of resource limits, and absence of health checks. To address these issues, we will make the following changes:

1. Replace the standalone Pod with a Deployment to ensure automatic pod replacement and enable scaling.
2. Change the restart policy to 'Always' to allow automatic restarts in case of failures.
3. Define appropriate resource limits and requests to prevent resource exhaustion.
4. Implement liveness and readiness probes to enable Kubernetes to detect and respond to pod health issues.
5. Update the Service to use NodePort type for better accessibility.

These changes will significantly enhance the system's ability to withstand and recover from the simulated fault scenarios, improving overall resilience and availability.

Modification_type: replace

File name: nginx/pod.yaml

We're replacing the Pod definition with a Deployment to address issues #0 (Single Pod Deployment) and #1 (Restart Policy). This change will allow for automatic pod replacement and scaling. We're also adding resource limits and requests (addressing issue #2) and implementing liveness and readiness probes (addressing issue #3).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  labels:
    app: example
spec:
  replicas: 2
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
        - name: example-container
          image: nginx:1.17.1
          ports:
            - containerPort: 80
          resources:
            limits:
              cpu: "500m"
              memory: "512Mi"
            requests:
              cpu: "250m"
              memory: "256Mi"
          livenessProbe:
            httpGet:
              path: /
              port: 80
            initialDelaySeconds: 10
            periodSeconds: 5
          readinessProbe:
            httpGet:
              path: /
              port: 80
            initialDelaySeconds: 5
            periodSeconds: 3
```

Modification_type: replace

File name: nginx/service.yaml

We're updating the Service to use NodePort type (addressing issue #4) for better accessibility and to potentially mitigate some network-related issues observed during the Chaos Engineering experiment.

```
apiVersion: v1
kind: Service
metadata:
```

```
name: example-service
spec:
  type: NodePort
  selector:
    app: example
  ports:
    - protocol: TCP
      port: 80
      targetPort: 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 160ns
- deployment.apps/example-deployment configured
- service/example-service configured
Waiting for deployments to stabilize...
- deployment/example-deployment: waiting for rollout to finish: 1 out of 2
- deployment/example-deployment: waiting for rollout to finish: 1 old repl
- deployment/example-deployment is ready.
Deployments stabilized in 13.097 seconds
You can also run [skaffold run --tail] to get the logs
```

Resource statuses

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

NAMESPACE	NAME	READY	STATUS
default	pod/example-deployment-7bbcc69fbf-9v88z	0/1	ContainerCreat
default	pod/example-deployment-7bbcc69fbf-c7wrg	1/1	Running
default	pod/example-deployment-7bbcc69fbf-p8zzf	1/1	Running

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	service/example-service	NodePort	10.96.111.118	<none>

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILA
default	deployment.apps/example-deployment	2/3	3	2

NAMESPACE	NAME	DESIRED	CURREI
default	replicaset.apps/example-deployment-78764f9cff	0	0
default	replicaset.apps/example-deployment-7bbcc69fbf	3	3

The fault-scope adjustment Completed!

Current fault injection settings:

- Injected Faults #0: StressChaos
 - Workflow Name: fault-stresschaos
 - Grace Period: 0s
 - Duration: 30s

Parameters {"mode": "one", "selector": {"namespaces": ["default"], "labelSelectors": {"app": "example"}}, "stressors": {"cpu": {"workers": 1, "load": 80}}, "memory": {"workers": 1, "size":

```
"256MB"}]]]]], "containerNames": ["example-container"], "value": "1"]}
```

The scope of the fault injection needs to be adjusted slightly for the current K8s manifests. The main change is that the previous manifest defined a single Pod, while the current manifest defines a Deployment with multiple replicas. The selector should still target the same app label, but we need to consider that there are now multiple pods. The namespace and label selector can remain the same as they are still applicable.

Next fault injection scope:

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

Current fault injection settings:

- Injected Faults #0: **NetworkChaos**
 - Workflow Name: **fault-networkchaos**
 - Grace Period: **5s**
 - Duration: **20s**

```
Parameters {"action": "delay", "mode": "all", "selector": {"namespaces": ["default"],
"labelSelectors": {"app": "example"}}, "delay": {"latency": "100ms", "correlation": "25", "jitter":
"10ms"}}, "direction": "to"}}
```

The scope of the fault injection should remain largely the same for the current K8s manifests. The main change in the manifests is that the previous Pod has been replaced with a Deployment, which manages multiple Pod replicas. However, the label selector used to target the Pods ('app: example') remains the same. The namespace also remains unchanged ('default'). Therefore, we can keep the same selector configuration, which will now apply to all Pods managed by the Deployment.

Next fault injection scope:

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

Current fault injection settings:

- Injected Faults #0: **PodChaos**
 - Workflow Name: **fault-podchaos**
 - Grace Period: **25s**
 - Duration: **5s**

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

The scope of the fault injection needs to be adjusted slightly to account for the changes in the current K8s manifests. The main difference is that the previous manifest defined a single Pod, while the current manifest defines a Deployment with multiple replicas. However, the core selector (app: example) remains the same, so we can keep that. We should maintain the 'one' mode to ensure we're still targeting a single pod at a time, even though there are now multiple pods. The namespace selector can remain unchanged as 'default'.

Next fault injection scope:

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

Adjusted unittest

The unit test requires adjustment to account for the changes in the reconfigured manifests. The main changes are:

1. The resource type has changed from Pod to Deployment.
2. The Deployment creates multiple replicas (2) of the pod.
3. The pod name is no longer static ('example-pod') but is dynamically generated by the Deployment.

These changes necessitate modifications to the unit test:

1. We need to list all pods created by the Deployment instead of checking a single pod.
2. We should verify that the number of running and ready pods meets or exceeds the specified number of replicas.
3. We need to update the pod selection method to use labels instead of a fixed pod name.

The threshold concept remains the same, but it will now apply to the overall state of the Deployment's pods rather than a single pod.

```
import argparse
import time
from kubernetes import client
from unittest_base import K8sAPIBase

class TestExampleDeploymentRunningAndReady(K8sAPIBase):
    def __init__(self):
        super().__init__()
        self.apps_v1 = client.AppsV1Api()

    def test_deployment_status(self, duration):
        deployment_name = 'example-deployment'
        namespace = 'default'
        label_selector = 'app=example'

        running_count = 0
        ready_count = 0
```

```

total_checks = 0

threshold = 0.95

start_time = time.time()
end_time = start_time + duration

while time.time() < end_time:
    try:
        deployment = self.apps_v1.read_namespaced_deployment(name=deployment_name, namespace=namespace)
        replicas = deployment.spec.replicas

        pods = self.v1.list_namespaced_pod(namespace=namespace, label_selector=label_selector)
        total_checks += 1

        running_pods = 0
        ready_pods = 0

        for pod in pods.items:
            if pod.status.phase == 'Running':
                running_pods += 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':
                    ready_pods += 1

        if running_pods >= replicas:
            running_count += 1
        if ready_pods >= replicas:
            ready_count += 1

        print(f'Deployment status: Running pods={running_pods}/{replicas}, Ready pods={ready_pods}/{replicas}')
    except Exception as e:
        print(f'Error checking deployment status: {e}')

    time.sleep(1)

if total_checks > 0:
    running_percentage = running_count / total_checks
    ready_percentage = ready_count / total_checks

    print('Summary:')
    print(f'Total checks: {total_checks}')
    print(f'Running: {running_count}/{total_checks} ({running_percentage*100}%)')
    print(f'Ready: {ready_count}/{total_checks} ({ready_percentage*100}%)')

    assert running_percentage >= threshold, f'Deployment did not have enough running pods'
    assert ready_percentage >= threshold, f'Deployment did not have enough ready pods'

    print(f'Test passed: Deployment had enough running and ready pods')
else:
    print('No checks were performed. Please check your Kubernetes configuration')

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--duration', type=int, default=30, help='Duration of the test in seconds')
    args = parser.parse_args()

    test = TestExampleDeploymentRunningAndReady()
    test.test_deployment_status(args.duration)

```

```
if __name__ == '__main__':  
    main()
```

Debugging:

The error occurs due to an unterminated string literal in the print statement. This is likely caused by an unintended line break in the f-string. To fix this, we need to remove the line break and ensure the f-string is properly formatted. This is a simple syntax error and does not require changes to the logic or intent of the test. We'll make this minor adjustment to fix the error without altering the functionality of the unit test.

Debugging:

The errors occur due to two main issues:

1. The 'TestExampleDeploymentRunningAndReady' object has no attribute 'apps_v1'. This suggests that the K8sAPIBase class doesn't initialize the apps_v1 client, which is needed for interacting with Deployments.
2. A ZeroDivisionError occurs because total_checks is zero, indicating that the loop didn't run at all, likely due to the first error.

To fix these issues, we need to:

1. Initialize the apps_v1 client in the K8sAPIBase class or in the test class itself.
2. Add error handling to prevent division by zero if no checks were performed.
3. Ensure that the v1 client is properly initialized for pod operations.

These changes will address the errors while maintaining the original intent of the test.

Adjusted unittest

The unit test requires adjustment to account for the changes in the reconfigured manifests. The main changes are:

1. The Pod has been replaced with a Deployment, introducing redundancy with 2 replicas.
2. Resource limits and requests have been added.
3. Liveness and readiness probes have been introduced.
4. The Service type has been changed to NodePort.

These changes, particularly the introduction of multiple replicas, necessitate modifications to the unit test to ensure it verifies the steady state across all instances. The test should now check that both replicas are responding correctly and that the load is distributed between them.

The threshold values in the original test remain relevant and should not be changed, as they still represent the desired steady state (98% success rate, less than 2% failure rate, and 95% of requests under 1 second).

To account for these changes, we should modify the test to:

1. Use the NodePort to access the service instead of the cluster-internal DNS name.
2. Increase the number of virtual users (vus) to 2 to simulate concurrent requests to both replicas.
3. Add a check to verify that responses come from different pods to ensure load distribution.
4. Extend the test duration to allow for the readiness and liveness probes to take effect.

These modifications will help ensure that the steady state is maintained across the entire redundant setup.

```

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

export const options = {
  vus: 2,
  duration: '2m',
  thresholds: {
    'checks': ['rate>0.98'],
    'http_req_failed': ['rate<0.02'],
    'http_req_duration': ['p(95)<1000']
  }
};

let podNames = new Set();

export function setup() {
  if (!__ENV.SERVICE_IP || !__ENV.SERVICE_PORT) {
    throw new Error('SERVICE_IP and SERVICE_PORT environment variables must be set');
  }
}

export default function () {
  const url = `http://${__ENV.SERVICE_IP}:${__ENV.SERVICE_PORT}`;
  const res = http.get(url);

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

  const podName = res.headers['X-Pod-Name'];
  if (podName) {
    podNames.add(podName);
  }

  sleep(1);
}

export function handleSummary(data) {
  check(podNames.size, {
    'requests were distributed across multiple pods': (count) => count > 1
  });
}

```

Debugging:

The error messages indicate that the URL '<http://nodeip:nodeport>' is invalid. This is because 'nodeip' and 'nodeport' are placeholders that need to be replaced with actual values. In a real Kubernetes environment, these would be dynamically obtained or configured.

To fix this issue, we need to modify the test to use environment variables for the node IP and port. This allows the actual values to be passed in when running the test, making it more flexible and reusable across different environments.

We'll make the following changes:

1. Use `__ENV.NODE_IP` and `__ENV.NODE_PORT` to get the node IP and port from environment variables.
2. Construct the URL using these environment variables.

3. Add a check to ensure these environment variables are set before running the test.

These changes will allow the test to run successfully when the correct node IP and port are provided, while maintaining the original intent of the test.

Debugging:

The error message indicates that the `NODE_IP` and `NODE_PORT` environment variables are not set, causing the test to fail in the setup function. This is because we're relying on these environment variables to construct the URL for our HTTP requests.

To address this issue while maintaining the original intent of the test, we should modify our approach to use a more flexible method of determining the service endpoint. Instead of relying on `NODE_IP` and `NODE_PORT`, we can use the Kubernetes API to dynamically discover the service endpoint.

We'll make the following changes:

1. Remove the setup function that checks for `NODE_IP` and `NODE_PORT`.
2. Add a function to retrieve the service endpoint using the Kubernetes API.
3. Use this function to get the endpoint URL before making HTTP requests.

This approach will make the test more robust and adaptable to different Kubernetes environments without requiring manual configuration of environment variables.

Debugging:

The error message indicates that the `'k6/x/kubernetes'` module is unknown. This suggests that the Kubernetes extension for k6 is not available or not properly installed in the testing environment. To address this issue while maintaining the original intent of the test, we need to modify our approach to avoid relying on the Kubernetes API directly within the test script.

Instead, we can use environment variables to pass the necessary information about the service endpoint to the test. This approach maintains the flexibility of the test while removing the dependency on the Kubernetes extension.

We'll make the following changes:

1. Remove the import of the Kubernetes module.
2. Remove the `getServiceEndpoint` function that uses the Kubernetes API.
3. Use environment variables `SERVICE_IP` and `SERVICE_PORT` to get the service endpoint information.
4. Add a check at the beginning of the test to ensure these environment variables are set.

This approach will allow the test to run in environments where the Kubernetes extension is not available, while still testing the service endpoint as intended.

AssertionError: MAX_MOD_COUNTS_EXCEEDED: 3

Traceback:

```
File "/opt/conda/lib/python3.10/site-packages/streamlit/runtime/scriptrunner/scriptrunner.py", line 100, 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.chaos eater.run_cycle()
```

