



## Your instructions for Chaos Engineering:

- The Chaos-Engineering experiment must be completed within 1 minute.
- When using k6 in steady-state definition, always select a request URL from the following options (other requests are invalid):
  1. <http://front-end.sock-shop.svc.cluster.local/>
  2. <http://front-end.sock-shop.svc.cluster.local/catalogue?size=10>
  3. <http://front-end.sock-shop.svc.cluster.local/detail.html?id=<ID>> Replace <ID> with an available ID: [ 03fef6ac-1896-4ce8-bd69-b798f85c6e0b , 3395a43e-2d88-40de-b95f-e00e1502085b , 510a0d7e-8e83-4193-b483-e27e09ddc34d , 808a2de1-1aaa-4c25-a9b9-6612e8f29a38 , 819e1fbf-8b7e-4f6d-811f-693534916a8b , 837ab141-399e-4c1f-9abc-bace40296bac , a0a4f044-b040-410d-8ead-4de0446aec7e , d3588630-ad8e-49df-bbd7-3167f7efb246 , zzz4f044-b040-410d-8ead-4de0446aec7e ]
  4. <http://front-end.sock-shop.svc.cluster.local/category/>
  5. <http://front-end.sock-shop.svc.cluster.local/category?tags=<TAG>> Replace <TAG> with an available tag: [ magic , action , blue , brown , black , sport , formal , red , green , skin , geek ]
  6. <http://front-end.sock-shop.svc.cluster.local/basket.html>



## Phase 0: Preprocessing

Cleaning the cluster **kind-chaos-eater-cluster** ... Done

```
$ kubectl delete workflow --all --context kind-chaos-eater-cluster -n chaos
No resources found
$ kubectl delete workflownode --all --context kind-chaos-eater-cluster -n cl
No resources found
$ kubectl delete deployments --all --context kind-chaos-eater-cluster -n ch
No resources found
$ kubectl delete pods --all --context kind-chaos-eater-cluster -n chaos-ct
```

Input instructions for your Chaos Engineering



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

## K8s manifest(s) to be deployed:

```
sock-shop-2/manifests/00-sock-shop-ns.yaml
```

```
apiVersion: v1
kind: Namespace
metadata:
  name: sock-shop
```

```
sock-shop-2/manifests/01-carts-dep.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: carts
  labels:
    name: carts
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: carts
  template:
    metadata:
      labels:
        name: carts
    spec:
      containers:
        - name: carts
          image: weaveworksdemos/carts:0.4.8
          env:
            - name: JAVA_OPTS
              value: -Xms64m -Xmx128m -XX:+UseG1GC -Djava.security.egd=file:/dev/./urandom
          resources:
            limits:
              cpu: 300m
              memory: 500Mi
            requests:
              cpu: 100m
              memory: 200Mi
      ports:
```

```

- containerPort: 80
securityContext:
  runAsNonRoot: true
  runAsUser: 10001
  capabilities:
    drop:
      - all
    add:
      - NET_BIND_SERVICE
  readOnlyRootFilesystem: true
volumeMounts:
- mountPath: /tmp
  name: tmp-volume
volumes:
- name: tmp-volume
  emptyDir:
    medium: Memory
nodeSelector:
  beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/02-carts-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: carts
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    name: carts
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 80
      targetPort: 80
  selector:
    name: carts

```

sock-shop-2/manifests/03-carts-db-dep.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: carts-db

```

```

labels:
  name: carts-db
namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: carts-db
  template:
    metadata:
      labels:
        name: carts-db
    spec:
      containers:
      - name: carts-db
        image: mongo
        ports:
        - name: mongo
          containerPort: 27017
        securityContext:
          capabilities:
            drop:
            - all
            add:
            - CHOWN
            - SETGID
            - SETUID
          readOnlyRootFilesystem: true
        volumeMounts:
        - mountPath: /tmp
          name: tmp-volume
      volumes:
      - name: tmp-volume
        emptyDir:
          medium: Memory
      nodeSelector:
        beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/04-carts-db-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: carts-db
  labels:
    name: carts-db

```

```

namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 27017
      targetPort: 27017
  selector:
    name: carts-db

```

sock-shop-2/manifests/05-catalogue-dep.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalogue
  labels:
    name: catalogue
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: catalogue
  template:
    metadata:
      labels:
        name: catalogue
    spec:
      containers:
        - name: catalogue
          image: weaveworksdemos/catalogue:0.3.5
          command: ["/app"]
          args:
            - -port=80
          resources:
            limits:
              cpu: 200m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 100Mi
          ports:
            - containerPort: 80
      securityContext:
        runAsNonRoot: true
        runAsUser: 10001

```

```

    capabilities:
      drop:
        - all
      add:
        - NET_BIND_SERVICE
    readOnlyRootFilesystem: true
  livenessProbe:
    httpGet:
      path: /health
      port: 80
    initialDelaySeconds: 300
    periodSeconds: 3
  readinessProbe:
    httpGet:
      path: /health
      port: 80
    initialDelaySeconds: 180
    periodSeconds: 3
  nodeSelector:
    beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/06-catalogue-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: catalogue
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    name: catalogue
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 80
      targetPort: 80
  selector:
    name: catalogue

```

sock-shop-2/manifests/07-catalogue-db-dep.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:

```

```

name: catalogue-db
labels:
  name: catalogue-db
namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: catalogue-db
  template:
    metadata:
      labels:
        name: catalogue-db
    spec:
      containers:
        - name: catalogue-db
          image: weaveworksdemos/catalogue-db:0.3.0
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: fake_password
            - name: MYSQL_DATABASE
              value: socksdb
          ports:
            - name: mysql
              containerPort: 3306
      nodeSelector:
        beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/08-catalogue-db-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: catalogue-db
  labels:
    name: catalogue-db
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 3306
      targetPort: 3306
  selector:
    name: catalogue-db

```

sock-shop-2/manifests/09-front-end-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end
  namespace: sock-shop
spec:
  replicas: 1
  selector:
    matchLabels:
      name: front-end
  template:
    metadata:
      labels:
        name: front-end
    spec:
      containers:
        - name: front-end
          image: weaveworksdemos/front-end:0.3.12
          resources:
            limits:
              cpu: 300m
              memory: 1000Mi
            requests:
              cpu: 100m
              memory: 300Mi
          ports:
            - containerPort: 8079
          env:
            - name: SESSION_REDIS
              value: "true"
          securityContext:
            runAsNonRoot: true
            runAsUser: 10001
            capabilities:
              drop:
                - all
            readOnlyRootFilesystem: true
      livenessProbe:
        httpGet:
          path: /
          port: 8079
        initialDelaySeconds: 300
        periodSeconds: 3
      readinessProbe:
        httpGet:
          path: /
```



```
    port: 8079
    initialDelaySeconds: 30
    periodSeconds: 3
nodeSelector:
  beta.kubernetes.io/os: linux
```

sock-shop-2/manifests/10-front-end-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: front-end
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    name: front-end
  namespace: sock-shop
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 8079
      nodePort: 30001
  selector:
    name: front-end
```

sock-shop-2/manifests/11-orders-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders
  labels:
    name: orders
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: orders
  template:
    metadata:
      labels:
        name: orders
    spec:
```

```

containers:
  - name: orders
    image: weaveworksdemos/orders:0.4.7
    env:
      - name: JAVA_OPTS
        value: -Xms64m -Xmx128m -XX:+UseG1GC -Djava.security.egd=file:/dev/./urandom
    resources:
      limits:
        cpu: 500m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 300Mi
    ports:
      - containerPort: 80
    securityContext:
      runAsNonRoot: true
      runAsUser: 10001
      capabilities:
        drop:
          - all
        add:
          - NET_BIND_SERVICE
      readOnlyRootFilesystem: true
    volumeMounts:
      - mountPath: /tmp
        name: tmp-volume
    volumes:
      - name: tmp-volume
        emptyDir:
          medium: Memory
    nodeSelector:
      beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/12-orders-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: orders
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    name: orders
  namespace: sock-shop
spec:

```

```
ports:
  # the port that this service should serve on
- port: 80
  targetPort: 80
selector:
  name: orders
```

sock-shop-2/manifests/13-orders-db-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders-db
  labels:
    name: orders-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: orders-db
  template:
    metadata:
      labels:
        name: orders-db
    spec:
      containers:
        - name: orders-db
          image: mongo
          ports:
            - name: mongo
              containerPort: 27017
          securityContext:
            capabilities:
              drop:
                - all
              add:
                - CHOWN
                - SETGID
                - SETUID
            readOnlyRootFilesystem: true
          volumeMounts:
            - mountPath: /tmp
              name: tmp-volume
      volumes:
        - name: tmp-volume
```

```
    emptyDir:
      medium: Memory
  nodeSelector:
    beta.kubernetes.io/os: linux
```

sock-shop-2/manifests/14-orders-db-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: orders-db
  labels:
    name: orders-db
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 27017
      targetPort: 27017
  selector:
    name: orders-db
```

sock-shop-2/manifests/15-payment-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment
  labels:
    name: payment
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: payment
  template:
    metadata:
      labels:
        name: payment
    spec:
      containers:
        - name: payment
          image: weaveworksdemos/payment:0.4.3
          resources:
```

```

    limits:
      cpu: 200m
      memory: 200Mi
    requests:
      cpu: 99m
      memory: 100Mi
  ports:
  - containerPort: 80
  securityContext:
    runAsNonRoot: true
    runAsUser: 10001
    capabilities:
      drop:
      - all
      add:
      - NET_BIND_SERVICE
    readOnlyRootFilesystem: true
  livenessProbe:
    httpGet:
      path: /health
      port: 80
    initialDelaySeconds: 300
    periodSeconds: 3
  readinessProbe:
    httpGet:
      path: /health
      port: 80
    initialDelaySeconds: 180
    periodSeconds: 3
  nodeSelector:
    beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/16-payment-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: payment
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    name: payment
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on

```

```
- port: 80
  targetPort: 80
selector:
  name: payment
```

sock-shop-2/manifests/17-queue-master-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: queue-master
  labels:
    name: queue-master
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: queue-master
  template:
    metadata:
      labels:
        name: queue-master
    spec:
      containers:
        - name: queue-master
          image: weaveworksdemos/queue-master:0.3.1
          env:
            - name: JAVA_OPTS
              value: -Xms64m -Xmx128m -XX:+UseG1GC -Djava.security.egd=file:/dev/./urandom
          resources:
            limits:
              cpu: 300m
              memory: 500Mi
            requests:
              cpu: 100m
              memory: 300Mi
          ports:
            - containerPort: 80
      nodeSelector:
        beta.kubernetes.io/os: linux
```

sock-shop-2/manifests/18-queue-master-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: queue-master
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    name: queue-master
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 80
      targetPort: 80
  selector:
    name: queue-master
```

sock-shop-2/manifests/19-rabbitmq-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rabbitmq
  labels:
    name: rabbitmq
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: rabbitmq
  template:
    metadata:
      labels:
        name: rabbitmq
      annotations:
        prometheus.io/scrape: "false"
    spec:
      containers:
        - name: rabbitmq
          image: rabbitmq:3.6.8-management
          ports:
            - containerPort: 15672
              name: management
            - containerPort: 5672
              name: rabbitmq
```

```

securityContext:
  capabilities:
    drop:
      - all
    add:
      - CHOWN
      - SETGID
      - SETUID
      - DAC_OVERRIDE
  readOnlyRootFilesystem: true
- name: rabbitmq-exporter
  image: kbudde/rabbitmq-exporter
  ports:
    - containerPort: 9090
      name: exporter
  nodeSelector:
    beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/20-rabbitmq-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: rabbitmq
  annotations:
    prometheus.io/scrape: 'true'
    prometheus.io/port: '9090'
  labels:
    name: rabbitmq
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 5672
      name: rabbitmq
      targetPort: 5672
    - port: 9090
      name: exporter
      targetPort: exporter
      protocol: TCP
  selector:
    name: rabbitmq

```

sock-shop-2/manifests/21-session-db-dep.yaml



```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: session-db
  labels:
    name: session-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: session-db
  template:
    metadata:
      labels:
        name: session-db
      annotations:
        prometheus.io.scrape: "false"
    spec:
      containers:
        - name: session-db
          image: redis:alpine
          ports:
            - name: redis
              containerPort: 6379
          securityContext:
            capabilities:
              drop:
                - all
              add:
                - CHOWN
                - SETGID
                - SETUID
            readOnlyRootFilesystem: true
      nodeSelector:
        beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/22-session-db-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: session-db
  labels:
    name: session-db
  namespace: sock-shop

```

```
spec:
  ports:
    # the port that this service should serve on
    - port: 6379
      targetPort: 6379
  selector:
    name: session-db
```

sock-shop-2/manifests/23-shipping-dep.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shipping
  labels:
    name: shipping
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: shipping
  template:
    metadata:
      labels:
        name: shipping
    spec:
      containers:
        - name: shipping
          image: weaveworksdemos/shipping:0.4.8
          env:
            - name: ZIPKIN
              value: zipkin.jaeger.svc.cluster.local
            - name: JAVA_OPTS
              value: -Xms64m -Xmx128m -XX:+UseG1GC -Djava.security.egd=file:/dev/./urandom
          resources:
            limits:
              cpu: 300m
              memory: 500Mi
            requests:
              cpu: 100m
              memory: 300Mi
          ports:
            - containerPort: 80
      securityContext:
        runAsNonRoot: true
```

```

    runAsUser: 10001
    capabilities:
      drop:
        - all
      add:
        - NET_BIND_SERVICE
    readOnlyRootFilesystem: true
    volumeMounts:
      - mountPath: /tmp
        name: tmp-volume
    volumes:
      - name: tmp-volume
        emptyDir:
          medium: Memory
    nodeSelector:
      beta.kubernetes.io/os: linux

```

sock-shop-2/manifests/24-shipping-svc.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: shipping
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    name: shipping
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 80
      targetPort: 80
  selector:
    name: shipping

```

sock-shop-2/manifests/25-user-dep.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: user
  labels:
    name: user
  namespace: sock-shop

```

```
spec:
  replicas: 2
  selector:
    matchLabels:
      name: user
  template:
    metadata:
      labels:
        name: user
    spec:
      containers:
        - name: user
          image: weaveworksdemos/user:0.4.7
          resources:
            limits:
              cpu: 300m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 100Mi
          ports:
            - containerPort: 80
          env:
            - name: mongo
              value: user-db:27017
          securityContext:
            runAsNonRoot: true
            runAsUser: 10001
            capabilities:
              drop:
                - all
              add:
                - NET_BIND_SERVICE
            readOnlyRootFilesystem: true
          livenessProbe:
            httpGet:
              path: /health
              port: 80
            initialDelaySeconds: 300
            periodSeconds: 3
          readinessProbe:
            httpGet:
              path: /health
              port: 80
            initialDelaySeconds: 180
            periodSeconds: 3
```

```
nodeSelector:  
  beta.kubernetes.io/os: linux
```

sock-shop-2/manifests/26-user-svc.yaml

```
apiVersion: v1  
kind: Service  
metadata:  
  name: user  
  annotations:  
    prometheus.io/scrape: 'true'  
  labels:  
    name: user  
  namespace: sock-shop  
spec:  
  ports:  
    # the port that this service should serve on  
    - port: 80  
      targetPort: 80  
  selector:  
    name: user
```

sock-shop-2/manifests/27-user-db-dep.yaml

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: user-db  
  labels:  
    name: user-db  
  namespace: sock-shop  
spec:  
  replicas: 2  
  selector:  
    matchLabels:  
      name: user-db  
  template:  
    metadata:  
      labels:  
        name: user-db  
    spec:  
      containers:  
        - name: user-db  
          image: weaveworksdemos/user-db:0.3.0
```

```

ports:
  - name: mongo
    containerPort: 27017
securityContext:
  capabilities:
    drop:
      - all
    add:
      - CHOWN
      - SETGID
      - SETUID
  readOnlyRootFilesystem: true
volumeMounts:
  - mountPath: /tmp
    name: tmp-volume
volumes:
  - name: tmp-volume
    emptyDir:
      medium: Memory
nodeSelector:
  beta.kubernetes.io/os: linux

```

`sock-shop-2/manifests/28-user-db-svc.yaml`

```

apiVersion: v1
kind: Service
metadata:
  name: user-db
  labels:
    name: user-db
  namespace: sock-shop
spec:
  ports:
    # the port that this service should serve on
    - port: 27017
      targetPort: 27017
  selector:
    name: user-db

```

## Resource statuses

```

$ kubectl get all --all-namespaces --context kind-chaos-eater-cluster --selector=sock-shop

```

NAMESPACE	NAME	READY	STATUS	RESTARTS
sock-shop	pod/carts-6b76bb656d-8mhk4	1/1	Running	0
sock-shop	pod/carts-6b76bb656d-xjmxg	1/1	Running	0

sock-shop	pod/carts-db-55f5544bd-9wqcq	1/1	Running	0	
sock-shop	pod/carts-db-55f5544bd-lv4mn	1/1	Running	0	
sock-shop	pod/catalogue-7c584b65cf-hph6b	1/1	Running	0	
sock-shop	pod/catalogue-7c584b65cf-phgjd	1/1	Running	0	
sock-shop	pod/catalogue-db-85c6bf79db-jpnrj	1/1	Running	0	
sock-shop	pod/catalogue-db-85c6bf79db-l75fv	1/1	Running	0	
sock-shop	pod/front-end-7f9fb85686-nrs6p	1/1	Running	0	
sock-shop	pod/orders-7449dbbd44-7s255	1/1	Running	0	
sock-shop	pod/orders-7449dbbd44-cghtv	1/1	Running	0	
sock-shop	pod/orders-db-5b6bd64894-ktp54	1/1	Running	0	
sock-shop	pod/orders-db-5b6bd64894-tszlb	1/1	Running	0	
sock-shop	pod/payment-6f6578b5bd-2cx94	1/1	Running	0	
sock-shop	pod/payment-6f6578b5bd-r44tk	1/1	Running	0	
sock-shop	pod/queue-master-768f9d697c-g4m6g	1/1	Running	0	
sock-shop	pod/queue-master-768f9d697c-mqwkl	1/1	Running	0	
sock-s...	8m21s				
sock-shop	deployment.apps/user	2/2	2	2	
sock-shop	deployment.apps/user-db	2/2	2	2	
NAMESPACE	NAME		DESIRED	CURRENT	RI
sock-shop	replicaset.apps/carts-6b76bb656d		2	2	2
sock-shop	replicaset.apps/carts-db-55f5544bd		2	2	2
sock-shop	replicaset.apps/catalogue-7c584b65cf		2	2	2
sock-shop	replicaset.apps/catalogue-db-85c6bf79db		2	2	2
sock-shop	replicaset.apps/front-end-7f9fb85686		1	1	1
sock-shop	replicaset.apps/orders-7449dbbd44		2	2	2
sock-shop	replicaset.apps/orders-db-5b6bd64894		2	2	2
sock-shop	replicaset.apps/payment-6f6578b5bd		2	2	2
sock-shop	replicaset.apps/queue-master-768f9d697c		2	2	2
sock-shop	replicaset.apps/rabbitmq-557cd854		2	2	2
sock-shop	replicaset.apps/session-db-6bdf8d69dd		2	2	2
sock-shop	replicaset.apps/shipping-54d8db4bfb		2	2	2
sock-shop	replicaset.apps/user-697597c56d		2	2	2
sock-shop	replicaset.apps/user-db-55664575c6		2	2	2

## Summary of each manifest:

`sock-shop-2/manifests/00-sock-shop-ns.yaml`

- This manifest defines a Kubernetes Namespace.
- The Namespace is named 'sock-shop'.
- Namespaces are used to organize and manage resources within a Kubernetes cluster.

`sock-shop-2/manifests/01-carts-dep.yaml`

- This manifest defines a Deployment in Kubernetes.

- The Deployment is named 'carts' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'carts' application running.
- The Deployment uses the Docker image 'weaveworksdemos/carts:0.4.8'.
- Environment variables are set for Java options to optimize memory usage and disable certain features.
- Resource limits and requests are defined, with a maximum of 300m CPU and 500Mi memory, and a minimum of 100m CPU and 200Mi memory.
- The application listens on port 80 within the container.
- Security context is configured to run the container as a non-root user with specific capabilities.
- The root filesystem is set to be read-only for security purposes.
- A temporary volume is mounted at '/tmp' using an in-memory empty directory.
- The Deployment is scheduled to run on nodes with the Linux operating system.

`sock-shop-2/manifests/02-carts-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'carts'.
- It is annotated to enable Prometheus scraping with 'prometheus.io/scrape: true'.
- The Service is labeled with 'name: carts'.
- It is deployed in the 'sock-shop' namespace.
- The Service exposes port 80 and directs traffic to the same port on the selected pods.
- It uses a selector to target pods with the label 'name: carts'.

`sock-shop-2/manifests/03-carts-db-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'carts-db' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'carts-db' pod running.
- The pods are selected based on the label 'name: carts-db'.
- Each pod runs a single container using the 'mongo' image.
- The container exposes port 27017, which is commonly used by MongoDB.
- Security settings are applied to drop all capabilities and only add CHOWN, SETGID, and SETUID.
- The root filesystem of the container is set to be read-only for security purposes.
- A temporary volume is mounted at '/tmp' using an in-memory emptyDir volume.
- The pods are scheduled to run on nodes with the operating system labeled as 'linux'.

`sock-shop-2/manifests/04-carts-db-svc.yaml`



- This manifest defines a Kubernetes Service.
- The Service is named 'carts-db'.
- It is located in the 'sock-shop' namespace.
- The Service is associated with pods that have the label 'name: carts-db'.
- It exposes port 27017, which is also the target port for the pods.

`sock-shop-2/manifests/05-catalogue-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'catalogue' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'catalogue' application running.
- The Deployment uses the Docker image 'weaveworksdemos/catalogue:0.3.5'.
- The application runs with the command '/app' and listens on port 80.
- Resource limits are set to 200m CPU and 200Mi memory, with requests for 100m CPU and 100Mi memory.
- The container is configured to run as a non-root user with user ID 10001.
- Security settings include dropping all capabilities except 'NET\_BIND\_SERVICE' and using a read-only root filesystem.
- Liveness and readiness probes are configured to check the '/health' endpoint on port 80, with specific initial delays and periods.
- The Deployment is scheduled to run on nodes with the Linux operating system.

`sock-shop-2/manifests/06-catalogue-svc.yaml`

- This manifest defines a Kubernetes Service.
- The service is named 'catalogue'.
- It is annotated for Prometheus scraping with 'prometheus.io/scrape: true'.
- The service is labeled with 'name: catalogue'.
- It is deployed in the 'sock-shop' namespace.
- The service exposes port 80 and directs traffic to the same port on the selected pods.
- It selects pods with the label 'name: catalogue' to route traffic to.

`sock-shop-2/manifests/07-catalogue-db-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'catalogue-db' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas (instances) of the 'catalogue-db' pod running.
- The pods are selected based on the label 'name: catalogue-db'.
- Each pod runs a container named 'catalogue-db' using the image 'weaveworksdemos/catalogue-db:0.3.0'.

- The container is configured with environment variables for 'MYSQL\_ROOT\_PASSWORD' and 'MYSQL\_DATABASE'.
- The container exposes port 3306, which is typically used for MySQL databases.
- The pods are scheduled to run on nodes with the operating system labeled as 'linux'.

`sock-shop-2/manifests/08-catalogue-db-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'catalogue-db'.
- It is located in the 'sock-shop' namespace.
- The Service is configured to expose port 3306.
- It targets port 3306 on the pods it selects.
- The Service uses a selector to match pods with the label 'name: catalogue-db'.

`sock-shop-2/manifests/09-front-end-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'front-end' and is located in the 'sock-shop' namespace.
- It specifies that there should be 1 replica of the front-end application running.
- The Deployment uses a selector to match pods with the label 'name: front-end'.
- The pod template includes a single container named 'front-end'.
- The container uses the image 'weaveworksdemos/front-end:0.3.12'.
- Resource limits are set for the container: 300m CPU and 1000Mi memory.
- Resource requests are set for the container: 100m CPU and 300Mi memory.
- The container exposes port 8079.
- An environment variable 'SESSION\_REDIS' is set to 'true'.
- Security context is configured to run the container as a non-root user with user ID 10001.
- All Linux capabilities are dropped, and the root filesystem is set to read-only.
- A liveness probe is configured to check the root path '/' on port 8079, with an initial delay of 300 seconds and a period of 3 seconds.
- A readiness probe is also configured to check the root path '/' on port 8079, with an initial delay of 30 seconds and a period of 3 seconds.
- The node selector ensures that the pod runs on nodes with the operating system labeled as Linux.

`sock-shop-2/manifests/10-front-end-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'front-end'.
- It is located in the 'sock-shop' namespace.

- The Service type is 'NodePort', which exposes the service on each Node's IP at a static port.
- It listens on port 80 and forwards traffic to target port 8079 on the pods.
- The nodePort is set to 30001, which is the port on each node where the service can be accessed externally.
- The Service is configured to select pods with the label 'name: front-end'.
- An annotation is included to enable Prometheus scraping for monitoring purposes.

`sock-shop-2/manifests/11-orders-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'orders' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'orders' application running.
- The Deployment uses the Docker image 'weaveworksdemos/orders:0.4.7'.
- Environment variables are set for Java options to optimize memory usage and disable certain features.
- Resource limits and requests are defined, with a maximum of 500m CPU and 500Mi memory, and a minimum of 100m CPU and 300Mi memory.
- The application listens on port 80 inside the container.
- Security context is configured to run the container as a non-root user with specific capabilities and a read-only root filesystem.
- A temporary volume is mounted at '/tmp' using an in-memory empty directory.
- The Deployment is scheduled to run on nodes with the Linux operating system.

`sock-shop-2/manifests/12-orders-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'orders'.
- It is annotated for Prometheus scraping, which means it is set up to be monitored by Prometheus.
- The Service is labeled with 'name: orders'.
- It is deployed in the 'sock-shop' namespace.
- The Service exposes port 80 and directs traffic to the same port on the selected pods.
- The Service selects pods with the label 'name: orders' to route traffic to them.

`sock-shop-2/manifests/13-orders-db-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'orders-db' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'orders-db' pod running.
- The pods are selected based on the label 'name: orders-db'.

- Each pod runs a single container using the 'mongo' image.
- The container exposes port 27017 for MongoDB.
- Security settings are applied to drop all capabilities and add specific ones like CHOWN, SETGID, and SETUID.
- The root filesystem of the container is set to read-only for security.
- A temporary volume is mounted at '/tmp' using an in-memory emptyDir.
- The pods are scheduled to run on nodes with the operating system labeled as 'linux'.

`sock-shop-2/manifests/14-orders-db-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'orders-db'.
- It is located in the 'sock-shop' namespace.
- The Service is configured to expose port 27017.
- It targets the same port (27017) on the pods it selects.
- The Service uses a selector to find pods with the label 'name: orders-db'.

`sock-shop-2/manifests/15-payment-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'payment' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'payment' application running.
- The Deployment uses the Docker image 'weaveworksdemos/payment:0.4.3'.
- Resource limits are set for the container, with a maximum of 200m CPU and 200Mi memory, and requests for 99m CPU and 100Mi memory.
- The container listens on port 80.
- Security settings ensure the container runs as a non-root user with user ID 10001, and the root filesystem is read-only.
- The container has a liveness probe and a readiness probe, both checking the '/health' endpoint on port 80.
- The liveness probe starts after 300 seconds and checks every 3 seconds, while the readiness probe starts after 180 seconds and also checks every 3 seconds.
- The Deployment is configured to run on nodes with the Linux operating system.

`sock-shop-2/manifests/16-payment-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'payment'.
- It is annotated for Prometheus scraping, which means it is set up to be monitored by Prometheus.
- The Service is labeled with 'name: payment'.

- It is deployed in the 'sock-shop' namespace.
- The Service exposes port 80 and directs traffic to the same port on the selected pods.
- The Service uses a selector to target pods with the label 'name: payment'.

`sock-shop-2/manifests/17-queue-master-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'queue-master' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas (copies) of the 'queue-master' application running.
- The Deployment uses a container image 'weaveworksdemos/queue-master:0.3.1'.
- Environment variables are set for the container, including Java options for memory management and garbage collection.
- Resource limits and requests are defined, with a maximum of 300m CPU and 500Mi memory, and a minimum of 100m CPU and 300Mi memory.
- The container exposes port 80 for network access.
- The Deployment is configured to run on nodes with the Linux operating system.

`sock-shop-2/manifests/18-queue-master-svc.yaml`

- This manifest defines a Kubernetes Service.
- The service is named 'queue-master'.
- It is annotated for Prometheus scraping with 'prometheus.io/scrape: true'.
- The service is labeled with 'name: queue-master'.
- It is deployed in the 'sock-shop' namespace.
- The service exposes port 80 and directs traffic to the same port on the selected pods.
- It selects pods with the label 'name: queue-master'.

`sock-shop-2/manifests/19-rabbitmq-dep.yaml`

- This manifest defines a Deployment for RabbitMQ in Kubernetes.
- The Deployment is named 'rabbitmq' and is located in the 'sock-shop' namespace.
- It specifies 2 replicas of the RabbitMQ application to be deployed.
- The Deployment uses a selector to match pods with the label 'name: rabbitmq'.
- The pod template includes two containers: one for RabbitMQ and another for RabbitMQ Exporter.
- The RabbitMQ container uses the image 'rabbitmq:3.6.8-management' and exposes ports 15672 (management) and 5672 (RabbitMQ service).
- The RabbitMQ container has a security context that drops all capabilities and adds specific ones like CHOWN, SETGID, SETUID, and DAC\_OVERRIDE, and it uses a read-only root filesystem.

- The RabbitMQ Exporter container uses the image 'kbudde/rabbitmq-exporter' and exposes port 9090 for metrics.
- The Deployment specifies a node selector to ensure that the pods run on nodes with the operating system labeled as Linux.
- Annotations are set to prevent Prometheus from scraping metrics from this deployment.

`sock-shop-2/manifests/20-rabbitmq-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'rabbitmq'.
- It is annotated for Prometheus scraping on port 9090.
- The Service is located in the 'sock-shop' namespace.
- It exposes two ports: 5672 for RabbitMQ and 9090 for an exporter.
- The Service uses TCP protocol for communication.
- It selects pods with the label 'name: rabbitmq'.

`sock-shop-2/manifests/21-session-db-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'session-db' and is located in the 'sock-shop' namespace.
- It specifies 2 replicas of the pod to be created.
- The pods are selected based on the label 'name: session-db'.
- Each pod runs a container using the 'redis' image.
- The container exposes port 6379, which is commonly used by Redis.
- Security settings are applied to drop all capabilities and only add CHOWN, SETGID, and SETUID.
- The root filesystem of the container is set to be read-only for security purposes.
- The pods are scheduled to run on nodes with the operating system labeled as Linux.

`sock-shop-2/manifests/22-session-db-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'session-db'.
- It is located in the 'sock-shop' namespace.
- The Service listens on port 6379 and forwards traffic to the same port on the target pods.
- It uses a selector to target pods with the label 'name: session-db'.

`sock-shop-2/manifests/23-shipping-dep.yaml`

- This manifest defines a Deployment in Kubernetes.

- The Deployment is named 'shipping' and is located in the 'sock-shop' namespace.
- It specifies 2 replicas of the 'shipping' application to be run.
- The Deployment uses the Docker image 'weaveworksdemos/shipping:0.4.8'.
- Environment variables 'ZIPKIN' and 'JAVA\_OPTS' are set for the container.
- Resource limits are set to 300m CPU and 500Mi memory, with requests for 100m CPU and 300Mi memory.
- The container exposes port 80.
- Security context is configured to run the container as a non-root user with user ID 10001.
- The container has a read-only root filesystem and specific capabilities are dropped and added.
- A temporary volume is mounted at '/tmp' using an in-memory emptyDir volume.
- The Deployment is scheduled to run on nodes with the Linux operating system.

`sock-shop-2/manifests/24-shipping-svc.yaml`

- This is a Kubernetes Service manifest.
- The service is named 'shipping'.
- It is annotated for Prometheus scraping with 'prometheus.io/scrape: true'.
- The service is labeled with 'name: shipping'.
- It is deployed in the 'sock-shop' namespace.
- The service exposes port 80 and directs traffic to the same port on the selected pods.
- It selects pods with the label 'name: shipping'.

`sock-shop-2/manifests/25-user-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'user' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'user' application running.
- The Deployment uses the Docker image 'weaveworksdemos/user:0.4.7'.
- Resource limits are set for the container: 300m CPU and 200Mi memory.
- Resource requests are set for the container: 100m CPU and 100Mi memory.
- The container exposes port 80.
- An environment variable 'mongo' is set with the value 'user-db:27017'.
- Security context is configured to run the container as a non-root user with user ID 10001.
- All capabilities are dropped except 'NET\_BIND\_SERVICE', and the root filesystem is set to read-only.
- A liveness probe is configured to check the '/health' endpoint on port 80, starting after 300 seconds and checking every 3 seconds.

- A readiness probe is also configured to check the '/health' endpoint on port 80, starting after 180 seconds and checking every 3 seconds.
- The Deployment is scheduled to run on nodes with the operating system labeled as Linux.

`sock-shop-2/manifests/26-user-svc.yaml`

- This is a Kubernetes Service manifest.
- The service is named 'user'.
- It is annotated for Prometheus scraping with 'prometheus.io/scrape: true'.
- The service is labeled with 'name: user'.
- It is deployed in the 'sock-shop' namespace.
- The service listens on port 80 and forwards traffic to the same port on the selected pods.
- The service selects pods with the label 'name: user'.

`sock-shop-2/manifests/27-user-db-dep.yaml`

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'user-db' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'user-db' pod running.
- The pods are selected based on the label 'name: user-db'.
- Each pod runs a single container using the image 'weaveworksdemos/user-db:0.3.0'.
- The container exposes port 27017, which is typically used by MongoDB.
- Security settings are applied to drop all capabilities and only add CHOWN, SETGID, and SETUID.
- The container's filesystem is set to be read-only, enhancing security.
- A temporary volume is mounted at '/tmp' using an in-memory emptyDir, which is useful for temporary storage needs.
- The pods are scheduled to run on nodes with the operating system labeled as 'linux'.

`sock-shop-2/manifests/28-user-db-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'user-db'.
- It is located in the 'sock-shop' namespace.
- The Service is configured to expose port 27017.
- It targets the same port (27017) on the pods.
- The Service selects pods with the label 'name: user-db'.

## Resiliency issues/weaknesses in the manifests:

Issue #0: Missing Resource Requests



- details: Pods may not get scheduled if the cluster is under resource pressure, leading to potential downtime.
- manifests having the issues: ['sock-shop-2/manifests/03-carts-db-dep.yaml', 'sock-shop-2/manifests/07-catalogue-db-dep.yaml', 'sock-shop-2/manifests/13-orders-db-dep.yaml', 'sock-shop-2/manifests/19-rabbitmq-dep.yaml', 'sock-shop-2/manifests/21-session-db-dep.yaml', 'sock-shop-2/manifests/27-user-db-dep.yaml']
- problematic config: The deployments for carts-db, catalogue-db, orders-db, rabbitmq, session-db, and user-db do not specify resource requests.

#### Issue #1: Single Replica Deployment

- details: The front-end deployment has only one replica, which can lead to downtime if the pod fails.
- manifests having the issues: ['sock-shop-2/manifests/09-front-end-dep.yaml']
- problematic config: spec.replicas: 1

#### Issue #2: High Initial Delay for Liveness Probe

- details: A high initial delay for the liveness probe can delay the detection of a failed pod, leading to longer downtime.
- manifests having the issues: ['sock-shop-2/manifests/05-catalogue-dep.yaml', 'sock-shop-2/manifests/09-front-end-dep.yaml', 'sock-shop-2/manifests/15-payment-dep.yaml', 'sock-shop-2/manifests/25-user-dep.yaml']
- problematic config: livenessProbe.initialDelaySeconds: 300

#### Issue #3: Missing Port Specification in Service

- details: The service may not function correctly if the port is not specified, leading to connectivity issues.
- manifests having the issues: ['sock-shop-2/manifests/02-carts-svc.yaml', 'sock-shop-2/manifests/04-carts-db-svc.yaml', 'sock-shop-2/manifests/06-catalogue-svc.yaml', 'sock-shop-2/manifests/08-catalogue-db-svc.yaml', 'sock-shop-2/manifests/12-orders-svc.yaml', 'sock-shop-2/manifests/14-orders-db-svc.yaml', 'sock-shop-2/manifests/16-payment-svc.yaml', 'sock-shop-2/manifests/18-queue-master-svc.yaml', 'sock-shop-2/manifests/22-session-db-svc.yaml', 'sock-shop-2/manifests/24-shipping-svc.yaml', 'sock-shop-2/manifests/26-user-svc.yaml', 'sock-shop-2/manifests/28-user-db-svc.yaml']
- problematic config: The services for carts, carts-db, catalogue, catalogue-db, orders, orders-db, payment, queue-master, session-db, shipping, user, and user-db do not specify the port that the service should serve on.

### Application of the manifests:

Thoughts:

The manifests provided are part of a Kubernetes deployment for a microservices-based application called 'sock-shop'. This is evident from the namespace 'sock-shop' and the naming conventions used in the manifests, such as 'carts', 'catalogue', 'orders', 'payment', 'shipping', 'user', and their respective databases. These components are typical of an e-commerce application, where users can browse a catalogue, add items to a cart, place orders, and make payments. The use of images like 'weaveworksdemos/carts', 'weaveworksdemos/catalogue', and others from the 'weaveworksdemos' repository further supports this assumption, as these are known to be part of the Sock Shop demo application, which is a microservices reference architecture for an e-commerce site.

Assumed application:

The Sock Shop application is a microservices-based e-commerce platform that simulates an online store where users can browse a catalogue of products, add items to a shopping cart, place orders, and make payments. It is designed to demonstrate the use of microservices in a cloud-native environment, showcasing how different services interact within a Kubernetes cluster.

## Summary of your instructions for Chaos Engineering:

- The Chaos-Engineering experiment must be completed within 1 minute.
- When using k6 in steady-state definition, always select a request URL from the following options (other requests are invalid):
  1. <http://front-end.sock-shop.svc.cluster.local/>
  2. <http://front-end.sock-shop.svc.cluster.local/catalogue?size=10>
  3. <http://front-end.sock-shop.svc.cluster.local/detail.html?id=<ID>> Replace <ID> with an available ID: [ 03fef6ac-1896-4ce8-bd69-b798f85c6e0b , 3395a43e-2d88-40de-b95f-e00e1502085b , 510a0d7e-8e83-4193-b483-e27e09ddc34d , 808a2de1-1aaa-4c25-a9b9-6612e8f29a38 , 819e1fbf-8b7e-4f6d-811f-693534916a8b , 837ab141-399e-4c1f-9abc-bace40296bac , a0a4f044-b040-410d-8ead-4de0446aec7e , d3588630-ad8e-49df-bbd7-3167f7efb246 , zzz4f044-b040-410d-8ead-4de0446aec7e ]
  4. <http://front-end.sock-shop.svc.cluster.local/category/>
  5. <http://front-end.sock-shop.svc.cluster.local/category?tags=<TAG>> Replace <TAG> with an available tag: [ magic , action , blue , brown , black , sport , formal , red , green , skin , geek ]
  6. <http://front-end.sock-shop.svc.cluster.local/basket.html>

## Phase 1: Hypothesis

---

### Steady-state definition

## Description

The first issue to address is 'Missing Resource Requests' in the 'carts-db' deployment. This is a critical issue because without resource requests, the pods may not get scheduled if the cluster is under resource pressure, leading to potential downtime. The 'carts-db' deployment is part of the database layer, which is crucial for the application's functionality. Ensuring that the 'carts-db' pods are running is essential for the system's steady state. Therefore, a steady state should be defined to ensure that the 'carts-db' deployment maintains its desired number of replicas, which is 2, to verify that the system can handle resource pressure without downtime.

## Current state inspection

To inspect the 'carts-db-replicas' state, I will use the Kubernetes API to check the number of replicas for the 'carts-db' deployment. This is because the state involves verifying the deployment's configuration and ensuring that the desired number of replicas is maintained. The Kubernetes API is suitable for this task as it allows us to directly query the deployment's status and verify the number of replicas. The check will be performed over a duration of 5 seconds to ensure stability in the replica count.

tool: `k8s` duration: `5s`

Inspection script (Python) `k8s_carts-db-replicas.py` :

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

def check_carts_db_replicas(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.AppsV1Api()
    namespace = 'sock-shop'
    deployment_name = 'carts-db'

    for _ in range(duration):
        try:
            deployment = v1.read_namespaced_deployment(deployment_name,
```

```

        replicas = deployment.status.replicas
        ready_replicas = deployment.status.ready_replicas
        print(f"Desired replicas: {replicas}, Ready replicas: {ready_replicas}")
    except client.exceptions.ApiException as e:
        print(f"Exception when calling AppsV1Api->read_namespaces: {e}")
    time.sleep(1)

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='Check carts-db replicas')
    parser.add_argument('--duration', type=int, default=5, help='Duration in seconds')
    args = parser.parse_args()
    check_carts_db_replicas(args.duration)

```

Pod `k8scarts-db-replicas-pod` has completed successfully.

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

```

Desired replicas: 2, Ready replicas: 2
Desired replicas: 2, Ready replicas: 2
Desired replicas: 2, Ready replicas: 2
Desired replicas: 2, Ready replicas: 2
Desired replicas: 2, Ready replicas: 2

```

## ► Threshold

The steady state in question is the number of replicas for the 'carts-db' deployment. The current state shows that both the desired and ready replicas are consistently at 2, which is the expected number of replicas as per the deployment configuration. To ensure the system remains in a steady state, we need to define a threshold that accounts for potential fluctuations while still maintaining the necessary redundancy for reliability. Given that the system is expected to handle resource pressure without downtime, the threshold should ensure that at least 2 replicas are ready at any given time. However, to account for transient states during scaling or updates, a reasonable tolerance can be applied. Therefore, the threshold can be set such that at least 2 replicas are ready 95% of the time during the monitoring period, which is 1 minute as per the chaos engineering experiment constraints.

At least 2 ready replicas for 'carts-db' must be maintained 95% of the time within a 1-minute monitoring period.

## Unit test to validate the steady state

To add the threshold assertion to the inspection Python script, I will implement a unit test that checks if the 'carts-db' deployment maintains at least 2 ready replicas 95% of the time over a specified duration. This involves counting the number of times the ready replicas are at least 2 and comparing it to the total number of checks to ensure it meets the 95% threshold. The test will use a for-loop to check the status every second for the specified duration, and at the end, it will assert whether the threshold condition is met.

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

class TestCartsDBReplicas(K8sAPIBase):
    def __init__(self, duration):
        super().__init__()
        self.duration = duration

    def test_carts_db_replicas(self):
        namespace = 'sock-shop'
        deployment_name = 'carts-db'
        v1 = client.AppsV1Api()

        # Initialize counters
        total_checks = 0
        successful_checks = 0

        # Check the deployment status for the specified duration
        for _ in range(self.duration):
            try:
                deployment = v1.read_namespaced_deployment(deployment_name, namespace)
                ready_replicas = deployment.status.ready_replicas
                if ready_replicas is not None:
                    print(f"Ready replicas: {ready_replicas}")

                # Increment total checks
                total_checks += 1

                # Check if the ready replicas meet the threshold
                if ready_replicas >= 2:
                    successful_checks += 1

            except client.exceptions.ApiException as e:
```

```

        print(f"Exception when calling AppsV1Api->read_namespaces")

    time.sleep(1)

    # Calculate the percentage of successful checks
    success_rate = (successful_checks / total_checks) * 100
    print(f"Success rate: {success_rate}%")

    # Assert that the success rate meets the 95% threshold
    assert success_rate >= 95, f"Threshold not met: {success_rate}%"

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

    # Create an instance of the test class with the specified duration
    test = TestCartsDBReplicas(args.duration)
    test.test_carts_db_replicas()

if __name__ == '__main__':
    main()

```

Pod `unittestcarts-db-replicasm00-pod` has completed successfully.

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

```

Ready replicas: 2
Ready replicas: 2
Ready replicas: 2
Ready replicas: 2
Ready replicas: 2
Success rate: 100.0%

```

## Steady state #2: front-end-replicas



### Description

The next issue to address is 'Single Replica Deployment' in the 'front-end' deployment. This is a critical issue because having only one replica can lead to downtime if the pod fails. The front-end is a crucial component as it serves as the entry point for users

interacting with the application. Ensuring that the front-end is always available is essential for the system's steady state. Therefore, a steady state should be defined to ensure that the 'front-end' deployment maintains its desired number of replicas, which is 1, to verify that the system can handle failures without downtime.

## Current state inspection

To inspect the number of replicas for the 'front-end' deployment in the 'sock-shop' namespace, the Kubernetes API is the most appropriate tool. This is because we need to check the deployment configuration and ensure that the desired state (number of replicas) is maintained. The K8s API allows us to directly query the deployment's status and verify the number of replicas. We will use a Python script with the Kubernetes client library to perform this check, iterating every second for a duration of 5 seconds to ensure consistency in the deployment's state.

tool: `k8s` duration: `5s`

Inspection script (Python) `k8s_front-end-replicas.py` :

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

def check_front_end_replicas(namespace, deployment_name, duration):
    # Load Kubernetes configuration based on the environment
    if os.getenv('KUBERNETES_SERVICE_HOST'):
        config.load_incluster_config()
    else:
        config.load_kube_config()

    v1 = client.AppsV1Api()
    for _ in range(duration):
        try:
            deployment = v1.read_namespaced_deployment(deployment_name, namespace)
            replicas = deployment.status.replicas
            ready_replicas = deployment.status.ready_replicas
            print(f"Desired replicas: {replicas}, Ready replicas: {ready_replicas}")
        except client.exceptions.ApiException as e:
            print(f"Exception when calling AppsV1Api->read_namespaced_deployment: {e}")
            time.sleep(1)

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='Check front-end replicas')
    parser.add_argument('--duration', type=int, default=5, help='Duration in seconds')
```

```
args = parser.parse_args()
check_front_end_replicas('sock-shop', 'front-end', args.duration)
```

Pod `k8sfront-end-replicas-pod` has completed successfully.

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

```
Desired replicas: 1, Ready replicas: 1
Desired replicas: 1, Ready replicas: 1
Desired replicas: 1, Ready replicas: 1
Desired replicas: 1, Ready replicas: 1
Desired replicas: 1, Ready replicas: 1
```

## Threshold

The steady state in question is the number of replicas for the 'front-end' deployment. The current state shows that the desired replicas are 1 and the ready replicas are also 1, which means the system is currently in a stable state. However, having only one replica is a potential risk for downtime if the pod fails. To ensure the system remains stable and can handle failures without downtime, the threshold should be set to ensure that at least 1 replica is always ready. Given the constraints of the chaos engineering experiment, which must be completed within 1 minute, and the fact that the current state is stable with 1 ready replica, the threshold should be set to ensure that the 'front-end' deployment maintains at least 1 ready replica 100% of the time during the monitoring period. This threshold accounts for the current state and ensures that the system remains stable under normal conditions.

At least 1 ready replica for the 'front-end' deployment must be maintained 100% of the time during the 1-minute monitoring period.

## Unit test to validate the steady state

To add the threshold assertion to the inspection Python script, I will implement a unit test that checks if the 'front-end' deployment maintains at least 1 ready replica 100% of the time during the specified duration. The test will use a for-loop to check the status every second for the given duration and will assert that the number of ready replicas is always at least 1. If the condition is not met at any point, the test will fail. The threshold is defined as maintaining at least 1 ready replica 100% of the time, which will be implemented as a ratio of successful checks to total checks.



```

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

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

    def test_steady_state(self):
        # Initialize variables to track the number of successful checks
        successful_checks = 0

        # Loop for the specified duration
        for _ in range(self.duration):
            try:
                # Read the deployment status
                deployment = self.v1.read_namespaced_deployment(self.namespace, self.deployment_name)
                ready_replicas = deployment.status.ready_replicas or 0

                # Check if the number of ready replicas is at least 1
                if ready_replicas >= 1:
                    successful_checks += 1

            except client.exceptions.ApiException as e:
                print(f"Exception when calling AppsV1Api->read_namespaced_deployment_status: {e}")

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

        # Calculate the success ratio
        success_ratio = successful_checks / self.duration

        # Assert that the success ratio is 100%
        assert success_ratio == 1.0, f"Steady state not maintained: {success_ratio}"

        # Print success message
        print("Steady state maintained 100% of the time")

def main():
    parser = argparse.ArgumentParser(description='Test front-end replicas')
    parser.add_argument('--duration', type=int, default=60, help='Duration in seconds')

```

```

args = parser.parse_args()

# Create an instance of the test class
test = TestFrontEndReplicas('sock-shop', 'front-end', args.duration)

# Run the test
test.test_steady_state()

if __name__ == '__main__':
    main()

```

Pod `unittestfront-end-replicasm00-pod` has failed.

```

Traceback (most recent call last):
  File "/chaos-eater/sandbox/cycle_20241127_024431/hypothesis/unittest.py", line 10, in <module>
    main()
  File "/chaos-eater/sandbox/cycle_20241127_024431/hypothesis/unittest.py", line 20, in test_steady_state
    test.test_steady_state()
  File "/chaos-eater/sandbox/cycle_20241127_024431/hypothesis/unittest.py", line 30, in test_steady_state
    deployment = self.v1.read_namespaced_deployment(self.deployment_name, self.namespace)
AttributeError: 'CoreV1Api' object has no attribute 'read_namespaced_deployment'

```

The error occurs because the `CoreV1Api` object does not have a method `read_namespaced_deployment`. Instead, the `AppsV1Api` should be used to access deployment-related methods. I will fix the error by changing the API client from `CoreV1Api` to `AppsV1Api` in the `K8sAPIBase` class initialization. This will allow the test to correctly read the deployment status and check the number of ready replicas.

```

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

class TestFrontEndReplicas(K8sAPIBase):
    def __init__(self, namespace, deployment_name, duration):
        super().__init__()
        self.namespace = namespace
        self.deployment_name = deployment_name
        self.duration = duration
        # Use AppsV1Api for deployment operations
        self.apps_v1 = client.AppsV1Api()

    def test_steady_state(self):

```

```

# Initialize variables to track the number of successful checks
successful_checks = 0

# Loop for the specified duration
for _ in range(self.duration):
    try:
        # Read the deployment status
        deployment = self.apps_v1.read_namespaced_deployment(
            name=self.deployment_name, namespace=self.namespace)
        ready_replicas = deployment.status.ready_replicas or 0

        # Check if the number of ready replicas is at least 1
        if ready_replicas >= 1:
            successful_checks += 1

    except client.exceptions.ApiException as e:
        print(f"Exception when calling AppsV1Api->read_namespaced_deployment_status: {e}")

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

# Calculate the success ratio
success_ratio = successful_checks / self.duration

# Assert that the success ratio is 100%
assert success_ratio == 1.0, f"Steady state not maintained: {success_ratio}"

# Print success message
print("Steady state maintained 100% of the time")

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

    # Create an instance of the test class
    test = TestFrontEndReplicas('sock-shop', 'front-end', args.duration)

    # Run the test
    test.test_steady_state()

if __name__ == '__main__':
    main()

```

Pod `unittestfront-end-replicasm01-pod` has completed successfully.

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

Steady state maintained 100% of the time

## Fault definition

### ✓ Scenario: Black Friday Sale

#### 🗨 Description

During a Black Friday sale, the system is expected to handle a significant increase in traffic and load. This event will test the system's ability to maintain steady states under high demand. The primary weaknesses identified are missing resource requests for critical database components and a single replica for the front-end service. These weaknesses could lead to resource contention and downtime, respectively. To simulate the Black Friday event, we will first inject a StressChaos to simulate high CPU and memory usage on the 'carts-db' deployment, which lacks resource requests. This will test if the system can maintain the 'carts-db-replicas' steady state under resource pressure. Next, we will inject a PodChaos to kill the single replica of the 'front-end' deployment to test if the system can maintain the 'front-end-replicas' steady state and recover quickly. This sequence simulates the phenomena of high demand and potential failures during a Black Friday sale, targeting the system's identified weaknesses.

#### 🧨 Fault-injection sequence

`StressChaos` ({'namespace': 'sock-shop', 'deployment': 'carts-db'}) → `PodChaos` ({'namespace': 'sock-shop', 'deployment': 'front-end'})

#### ⚙ Detailed fault parameters

Detailed parameters of `StressChaos` ({'namespace': 'sock-shop', 'deployment': 'carts-db'})

```
▼ {
  "mode" : "all"
  ▼ "selector" : {
    ▼ "namespaces" : [
      0 : "sock-shop"
    ]
  }
}
```

```

    ▼ "labelSelectors" : {
      |   "name" : "carts-db"
      |
    }
  }
  ▼ "stressors" : {
    ▼ "cpu" : {
      |   "workers" : 2
      |   "load" : 100
      |
    }
    ▼ "memory" : {
      |   "workers" : 2
      |   "size" : "256MB"
      |
    }
  }
  ▼ "containerNames" : [
    |   0 : "carts-db"
    |
  ]
}

```

Detailed parameters of `PodChaos` ({'namespace': 'sock-shop', 'deployment': 'front-end'})

```

▼ {
  |   "action" : "pod-kill"
  |   "mode" : "one"
  |   ▼ "selector" : {
  |     |   ▼ "namespaces" : [
  |       |   0 : "sock-shop"
  |       |
  |     ]
  |     ▼ "labelSelectors" : {
  |       |   "name" : "front-end"
  |       |
  |     }
  |   }
}

```

## Phase 2: Chaos Experiment

CE experiment Planning Completed!

## Time Schedule

The chaos engineering experiment is designed to test the system's resilience under simulated fault conditions, specifically during a high-demand event like a Black Friday sale. The experiment is divided into three phases: pre-validation, fault-injection, and post-validation. Given the constraints that the entire experiment must be completed within 1 minute, we need to allocate time efficiently across these phases.

1. **Pre-validation Phase:** This phase ensures that the system is in a steady state before any faults are injected. We will allocate 15 seconds for this phase. During this time, we will verify that the 'carts-db' deployment maintains at least 2 ready replicas 95% of the time and that the 'front-end' deployment maintains at least 1 ready replica 100% of the time. This is crucial to establish a baseline for the system's normal operation.
2. **Fault-injection Phase:** This phase involves injecting faults to simulate the Black Friday event. We will allocate 30 seconds for this phase. The faults include a StressChaos on the 'carts-db' deployment to simulate high CPU and memory usage, and a PodChaos to kill the single replica of the 'front-end' deployment. This phase is critical to observe how the system behaves under stress and failure conditions.
3. **Post-validation Phase:** This phase checks if the system returns to its steady state after the faults are removed. We will allocate 15 seconds for this phase. Similar to the pre-validation phase, we will verify the steady states of the 'carts-db' and 'front-end' deployments to ensure that the system can recover and maintain its expected performance.

The total time for the experiment is 60 seconds, with 15 seconds for pre-validation, 30 seconds for fault injection, and 15 seconds for post-validation. This allocation allows us to thoroughly test the system's resilience while adhering to the time constraint.

Total experiment time: 60s

Pre-validation Phase: 15s

Fault-injection Phase: 30s

Post-validation Phase: 15s

## Pre-validation Phase (15s)

In the pre-validation phase, we need to ensure that the system is in a steady state before we proceed with fault injection. Given the constraints of a 15-second total time for this phase, we will focus on verifying the two defined steady states: 'carts-db-replicas' and 'front-end-replicas'. These checks will be executed sequentially due to the short time frame, ensuring that each steady state is verified independently. The 'carts-db-replicas' steady state will be checked first, as it involves verifying that at least 2 replicas are ready 95% of the time within a 1-minute monitoring period. However, for the pre-validation phase, we will adjust the check to fit within the 15-second window, ensuring that the system is stable at the start of the experiment. Following this, we will verify the 'front-end-replicas' steady state, ensuring that at least 1 replica is ready 100% of the time during the 15-second pre-validation period. This approach ensures that both critical components are functioning as expected before introducing any faults.

- Verified Steady State #0: `carts-db-replicas`
  - Workflow Name: `pre-unittest-carts-db-replicas`
  - Grace Period: `0s`
  - Duration: `7s`
- Verified Steady State #1: `front-end-replicas`
  - Workflow Name: `pre-unittest-front-end-replicas`
  - Grace Period: `7s`
  - Duration: `8s`

### Fault-injection Phase (30s)

The fault-injection phase is designed to simulate a Black Friday sale scenario, which involves high demand and potential failures. The goal is to test the system's ability to maintain its steady states under these conditions. Given the constraints of a 30-second total time for this phase, we need to carefully schedule the fault injections to maximize the observation of system behavior while ensuring that the faults are impactful.

First, we will inject a `StressChaos` fault targeting the `carts-db` deployment. This fault will simulate high CPU and memory usage, testing the system's ability to maintain the `carts-db-replicas` steady state under resource pressure. This fault will be injected at the start of the phase (grace period of 0s) and will last for 15 seconds. This duration is chosen to allow enough time for the system to react to the stress while leaving time for the next fault.

Next, we will inject a `PodChaos` fault targeting the `front-end` deployment. This fault will kill the single replica of the `front-end` service, testing the system's ability to maintain the `front-end-replicas` steady state and recover quickly. This fault will be injected with a grace period of 15 seconds, starting immediately after the `StressChaos`

fault ends, and will last for 15 seconds. This staggered approach ensures that each fault is isolated in time, allowing us to observe the system's response to each fault individually.

The unit tests for verifying the steady states will run concurrently with the fault injections. The `carts-db-replicas` unit test will start at the beginning of the phase and run for the entire 30 seconds, as it needs to monitor the system's ability to maintain the required number of replicas under stress. Similarly, the `front-end-replicas` unit test will also run for the entire 30 seconds, ensuring that the system can maintain at least one ready replica of the `front-end` service despite the pod-kill fault.

- Verified Steady State #0: `carts-db-replicas`
  - Workflow Name: `fault-unittest-carts-db-replicas`
  - Grace Period: `0s`
  - Duration: `30s`
- Verified Steady State #1: `front-end-replicas`
  - Workflow Name: `fault-unittest-front-end-replicas`
  - Grace Period: `0s`
  - Duration: `30s`
- Injected Faults #0: `StressChaos`
  - Workflow Name: `fault-stresschaos`
  - Grace Period: `0s`
  - Duration: `15s`
- Injected Faults #1: `PodChaos`
  - Workflow Name: `fault-podchaos`
  - Grace Period: `15s`
  - Duration: `15s`

### Post-validation Phase (15s)

In the post-validation phase, we need to ensure that the system has returned to its steady states after the fault injection. Given the constraints of a 15-second total time for this phase, we will focus on quickly verifying the two steady states: 'carts-db-replicas' and 'front-end-replicas'. The approach is to run the unit tests for each steady state sequentially, as the time constraint does not allow for simultaneous execution. Each unit test will have a short grace period to allow the system to stabilize after the fault injection, followed by a brief duration to verify the steady state. The thought process is to prioritize the verification of the critical components affected by the fault injection, ensuring that the system can recover and maintain its expected behavior.



- Verified Steady State #0: `carts-db-replicas`
  - Workflow Name: `post-unittest-carts-db-replicas`
  - Grace Period: `0s`
  - Duration: `7s`
- Verified Steady State #1: `front-end-replicas`
  - Workflow Name: `post-unittest-front-end-replicas`
  - Grace Period: `7s`
  - Duration: `8s`

### Summary

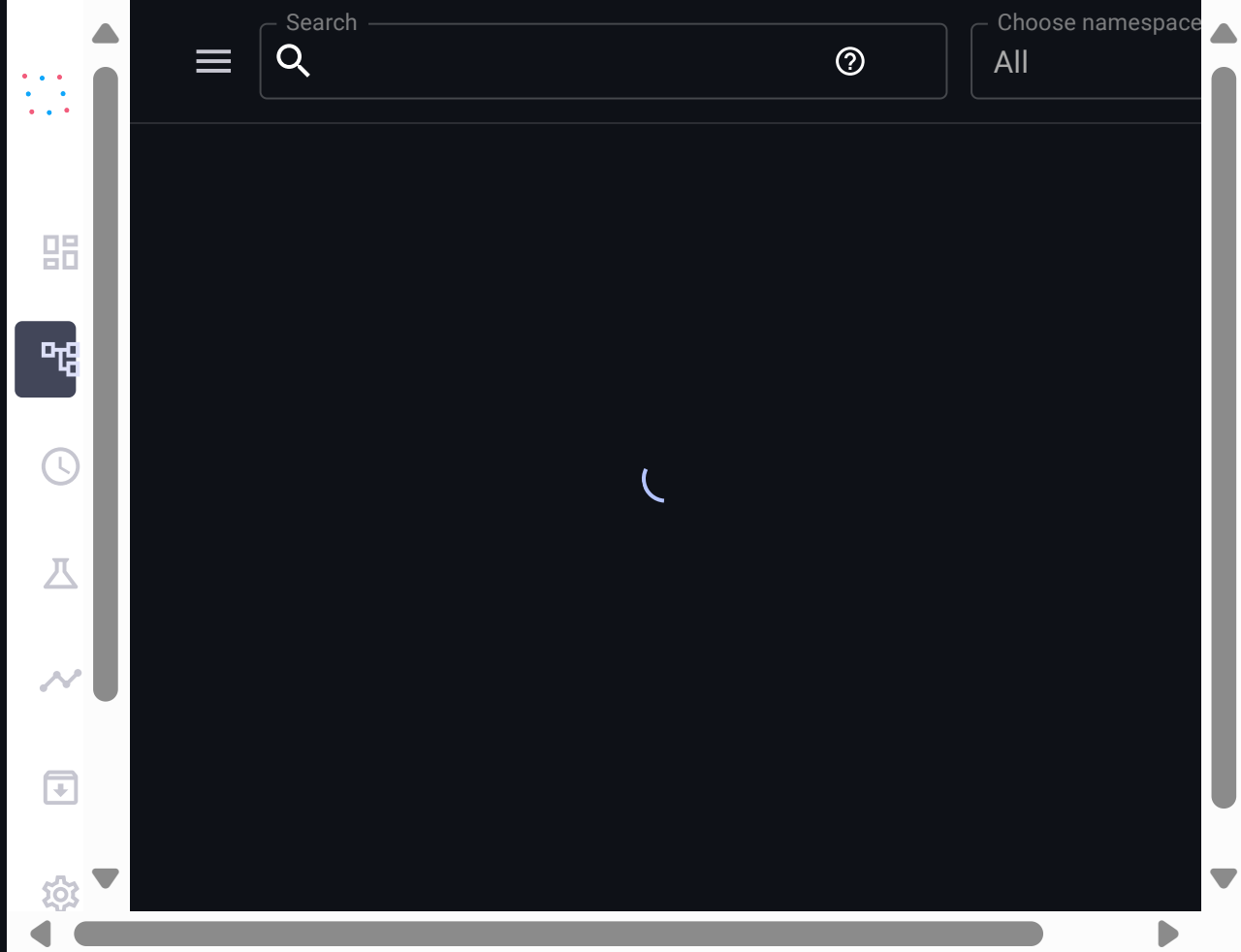
The chaos engineering experiment is structured into three phases: pre-validation, fault-injection, and post-validation, all to be completed within a total of 60 seconds.

In the pre-validation phase, which lasts for 15 seconds, the system's steady state is verified before any faults are introduced. The verification of the 'carts-db-replicas' steady state is conducted first, with no waiting time before it starts, and it runs for 7 seconds. This is followed by the verification of the 'front-end-replicas' steady state, which begins immediately after the first verification ends, with a duration of 8 seconds. This sequential execution ensures that each component is independently verified for stability.

The fault-injection phase spans 30 seconds and is designed to simulate high-demand conditions. It begins with the injection of a 'StressChaos' fault on the 'carts-db' deployment, which starts immediately and lasts for 15 seconds. Following this, a 'PodChaos' fault is injected on the 'front-end' deployment, starting at the 15-second mark and also lasting for 15 seconds. Concurrently, unit tests for both 'carts-db-replicas' and 'front-end-replicas' run for the entire 30 seconds, starting at the beginning of the phase, to monitor the system's ability to maintain steady states under stress.

In the post-validation phase, which also lasts 15 seconds, the system's recovery to steady states is verified. The 'carts-db-replicas' steady state is checked first, starting immediately and running for 7 seconds. This is followed by the 'front-end-replicas' steady state verification, which begins at the 7-second mark and lasts for 8 seconds. This sequential approach ensures that the system's recovery is thoroughly validated after the fault conditions are removed.

**Completed the chaos experiment!**



## Phase 3: Analysis

---

The chaos engineering experiment aimed to test the system's resilience under simulated high-demand conditions, specifically focusing on the 'carts-db' and 'front-end' components. The experiment results indicate that while the 'carts-db' component maintained its steady state throughout the experiment, the 'front-end' component failed to do so during both the fault-injection and post-validation phases.

### 1. Analysis of 'carts-db' Component:

- The 'carts-db' deployment successfully maintained its steady state, with at least 2 ready replicas 95% of the time, as required. This indicates that the system's configuration for 'carts-db', despite lacking explicit resource requests, was able to handle the simulated resource pressure from the 'StressChaos' fault. This success suggests that the current resource allocation and scheduling policies were sufficient to meet the demands placed on 'carts-db' during the experiment.

### 2. Analysis of 'front-end' Component:

- The 'front-end' deployment failed to maintain its steady state during the fault-injection phase, with a success rate of only 43.33%. This failure is directly linked to the 'PodChaos' fault, which killed the single replica of the 'front-end' deployment. The lack of

redundancy (only one replica) meant that any disruption led to immediate downtime, as evidenced by the failure to maintain the steady state.

- In the post-validation phase, the 'front-end' deployment's success rate improved to 87.5%, indicating partial recovery. However, this was still below the required 100% threshold, suggesting that the system took longer than expected to recover from the fault. This delay could be due to factors such as pod scheduling delays or insufficient resources to quickly bring the 'front-end' back online.

### 3. Insights and Recommendations:

- **Increase Redundancy:** The primary issue with the 'front-end' component is the lack of redundancy. Increasing the number of replicas for the 'front-end' deployment from 1 to at least 2 would provide the necessary redundancy to handle pod failures without causing downtime.
- **Resource Requests and Limits:** While the 'carts-db' component managed to maintain its steady state, it is still advisable to define resource requests and limits for all deployments, including 'carts-db', to ensure consistent performance under varying load conditions.
- **Improved Recovery Mechanisms:** Investigate the reasons for the delayed recovery of the 'front-end' component post-fault. This could involve optimizing pod scheduling policies or ensuring that sufficient resources are available to quickly restart failed pods.

In conclusion, the experiment highlighted the critical need for redundancy in the 'front-end' component to prevent downtime during high-demand scenarios. Addressing this issue, along with defining resource requests and improving recovery mechanisms, will enhance the system's resilience and ability to maintain steady states under fault conditions.

## Phase 4: Improvement

### Reconfiguration

To improve the resiliency of the 'front-end' component and avoid failures during high-demand scenarios, I will increase the number of replicas for the 'front-end' deployment from 1 to 2. This change will provide redundancy, ensuring that the system can handle pod failures without downtime. Additionally, I will add resource requests to the 'carts-db' deployment to ensure consistent performance under resource pressure.

Modification\_type: replace

File name: sock-shop-2/manifests/09-front-end-dep.yaml

Increase the number of replicas for the 'front-end' deployment from 1 to 2 to provide redundancy and prevent downtime during pod failures.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: front-end
  template:
    metadata:
      labels:
        name: front-end
    spec:
      containers:
        - name: front-end
          image: weaveworksdemos/front-end:0.3.12
          resources:
            limits:
              cpu: 300m
              memory: 1000Mi
            requests:
              cpu: 100m
              memory: 300Mi
          ports:
            - containerPort: 8079
          env:
            - name: SESSION_REDIS
              value: "true"
          securityContext:
            runAsNonRoot: true
            runAsUser: 10001
            capabilities:
              drop:
                - all
            readOnlyRootFilesystem: true
      livenessProbe:
        httpGet:
          path: /
          port: 8079
        initialDelaySeconds: 300
        periodSeconds: 3
      readinessProbe:
        httpGet:
          path: /
          port: 8079
```

```
    initialDelaySeconds: 30
    periodSeconds: 3
nodeSelector:
  beta.kubernetes.io/os: linux
```

Modification\_type: replace

File name: sock-shop-2/manifests/03-carts-db-dep.yaml

Add resource requests to the 'carts-db' deployment to ensure consistent performance under resource pressure.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: carts-db
  labels:
    name: carts-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: carts-db
  template:
    metadata:
      labels:
        name: carts-db
    spec:
      containers:
        - name: carts-db
          image: mongo
          ports:
            - name: mongo
              containerPort: 27017
          resources:
            requests:
              cpu: 100m
              memory: 200Mi
          securityContext:
            capabilities:
              drop:
                - all
              add:
                - CHOWN
                - SETGID
                - SETUID
```

```
    readOnlyRootFilesystem: true
  volumeMounts:
  - mountPath: /tmp
    name: tmp-volume
  volumes:
  - name: tmp-volume
    emptyDir:
      medium: Memory
  nodeSelector:
    beta.kubernetes.io/os: linux
```

## 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 290ns
- namespace/sock-shop unchanged
- Warning: spec.template.spec.nodeSelector[beta.kubernetes.io/os]: deprecated
- deployment.apps/carts configured
- service/carts configured
- deployment.apps/carts-db configured
- service/carts-db configured
- deployment.apps/catalogue configured
- service/catalogue configured
- deployment.apps/catalogue-db configured
- service/catalogue-db configured
- deployment.apps/front-end configured
- service/front-end configured
- deployment.apps/orders configured
- service/orders configured
- deployment.apps/orders-db configured
- service/orders-db configured
- deployment.apps/payment configured
- service/payment configured
- deployment.apps/queue-master configured
- service/queue-master configured
- deployment.apps/rabbitmq configured
- service/rabbitmq configured
- deployment.apps/session-db configured
- service/session-db configured
- deployment.apps/shipping configured
- service/shipping configured
```

- deployment.apps/user configured
- service/user configured
- deployment.apps/user-db configured
- service/user-db configured

Waiting for deployments to stabilize...

- sock-shop:deployment/carts: waiting for rollout to finish: 1 out of 2 new replicas are available
- sock-shop:deployment/carts-db: creating container carts-db
  - sock-shop:pod/carts-db-5...ueue-master is ready. [10/14 deployment(s) still pending]
- sock-shop:deployment/orders is ready. [9/14 deployment(s) still pending]
- sock-shop:deployment/session-db is ready. [8/14 deployment(s) still pending]
- sock-shop:deployment/carts-db is ready. [7/14 deployment(s) still pending]
- sock-shop:deployment/carts is ready. [6/14 deployment(s) still pending]
- sock-shop:deployment/orders-db is ready. [5/14 deployment(s) still pending]
- sock-shop:deployment/rabbitmq: creating container rabbitmq
  - sock-shop:pod/rabbitmq-6c98597b69-qnk96: creating container rabbitmq
- sock-shop:deployment/rabbitmq is ready. [4/14 deployment(s) still pending]
- sock-shop:deployment/front-end: waiting for rollout to finish: 1 old replicas are available
- sock-shop:deployment/front-end is ready. [3/14 deployment(s) still pending]
- sock-shop:deployment/catalogue: creating container catalogue
  - sock-shop:pod/catalogue-6846f4b58b-2zwgn: creating container catalogue
- sock-shop:deployment/payment: waiting for rollout to finish: 1 old replicas are available
- sock-shop:deployment/user: waiting for rollout to finish: 1 old replicas are available
- sock-shop:deployment/payment is ready. [2/14 deployment(s) still pending]
- sock-shop:deployment/catalogue is ready. [1/14 deployment(s) still pending]
- sock-shop:deployment/user is ready.

Deployments stabilized in 6 minutes 5.213 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=sock-shop
```

NAMESPACE	NAME	READY	STATUS	RESTARTS
sock-shop	pod/carts-7fd8594f7-2nm4k	1/1	Running	0
sock-shop	pod/carts-7fd8594f7-rfxdn	1/1	Running	0
sock-shop	pod/carts-db-5dbddbcc7b-55x8m	1/1	Running	0
sock-shop	pod/carts-db-5dbddbcc7b-pp9jw	1/1	Running	0
sock-shop	pod/catalogue-6846f4b58b-2zwgn	1/1	Running	0
sock-shop	pod/catalogue-6846f4b58b-r5bqt	1/1	Running	0
sock-shop	pod/catalogue-db-6b77cc59d7-bdp6p	1/1	Running	0
sock-shop	pod/catalogue-db-6b77cc59d7-s7s2j	1/1	Running	0
sock-shop	pod/front-end-7bc6bf685f-bjhp9	1/1	Running	0
sock-shop	pod/front-end-7bc6bf685f-mrs7f	1/1	Running	0
sock-shop	pod/orders-86b9477587-4xmfl	1/1	Running	0
sock-shop	pod/orders-86b9477587-whn22	1/1	Running	0
sock-shop	pod/orders-db-746f657687-6csbd	1/1	Running	0

sock-shop	pod/orders-db-746f657687-f7sr7	1/1	Running	0
sock-shop	pod/payment-8f48c77b9-gv6rf	1/1	Running	0
sock-shop	pod/payment-8f48c77b9-rd8kh	1/1	Running	0
sock-shop	pod/queue-master-6697989...	9m18s		
sock-shop	replicaset.apps/orders-86b9477587	2	2	2
sock-shop	replicaset.apps/orders-db-746f657687	2	2	2
sock-shop	replicaset.apps/orders-db-784f55785f	0	0	0
sock-shop	replicaset.apps/payment-86496f594f	0	0	0
sock-shop	replicaset.apps/payment-8f48c77b9	2	2	2
sock-shop	replicaset.apps/queue-master-6697989c86	2	2	2
sock-shop	replicaset.apps/queue-master-95d8c645d	0	0	0
sock-shop	replicaset.apps/rabbitmq-6c98597b69	2	2	2
sock-shop	replicaset.apps/rabbitmq-6f6869fbdf	0	0	0
sock-shop	replicaset.apps/session-db-665b767949	0	0	0
sock-shop	replicaset.apps/session-db-8bf47db5c	2	2	2
sock-shop	replicaset.apps/shipping-5df49fcfb7	2	2	2
sock-shop	replicaset.apps/shipping-6b7f5fcfcf	0	0	0
sock-shop	replicaset.apps/user-545cf57c87	0	0	0
sock-shop	replicaset.apps/user-7d5dd98cdc	2	2	2
sock-shop	replicaset.apps/user-db-6db75dbc8f	2	2	2
sock-shop	replicaset.apps/user-db-876ff477	0	0	0

## The fault-scope adjustment Completed!

Current fault injection settings:

- Injected Faults #0: **StressChaos**
  - Workflow Name: **fault-stresschaos**
  - Grace Period: **0s**
  - Duration: **15s**

Parameters {"mode": "all", "selector": {"namespaces": ["sock-shop"], "labelSelectors": {"name": "carts-db"}}, "stressors": {"cpu": {"workers": 2, "load": 100}, "memory": {"workers": 2, "size": "256MB"}}, "containerNames": ["carts-db"]}

The current Kubernetes manifests show that the 'carts-db' deployment has not changed in terms of its namespace or labels. The 'carts-db' deployment still exists in the 'sock-shop' namespace and retains the label 'name: carts-db'. Therefore, the scope of the fault injection, which targets the 'carts-db' deployment using these labels and namespace, remains valid. There is no need to adjust the scope of the fault injection for the current manifests.

Next fault injection scope:

```

{
  "namespaces" : [

```



```

    0 : "sock-shop"
  ]
  "labelSelectors" : {
    "name" : "carts-db"
  }
}

```

Current fault injection settings:

- Injected Faults #0: PodChaos
  - Workflow Name: fault-podchaos
  - Grace Period: 15s
  - Duration: 15s

Parameters {"action": "pod-kill", "mode": "one", "selector": {"namespaces": ["sock-shop"], "labelSelectors": {"name": "front-end"}}}

The current Kubernetes manifests show that the 'front-end' deployment now has 2 replicas instead of 1. The original chaos experiment targeted the 'front-end' deployment with a PodChaos to kill a single replica. Since the number of replicas has increased, the scope of the fault injection should be adjusted to ensure that the experiment still targets only one pod, maintaining the original goal of testing the system's resilience when a single 'front-end' pod is killed.

Next fault injection scope:

```

{
  "namespaces" : [
    0 : "sock-shop"
  ]
  "labelSelectors" : {
    "name" : "front-end"
  }
}

```

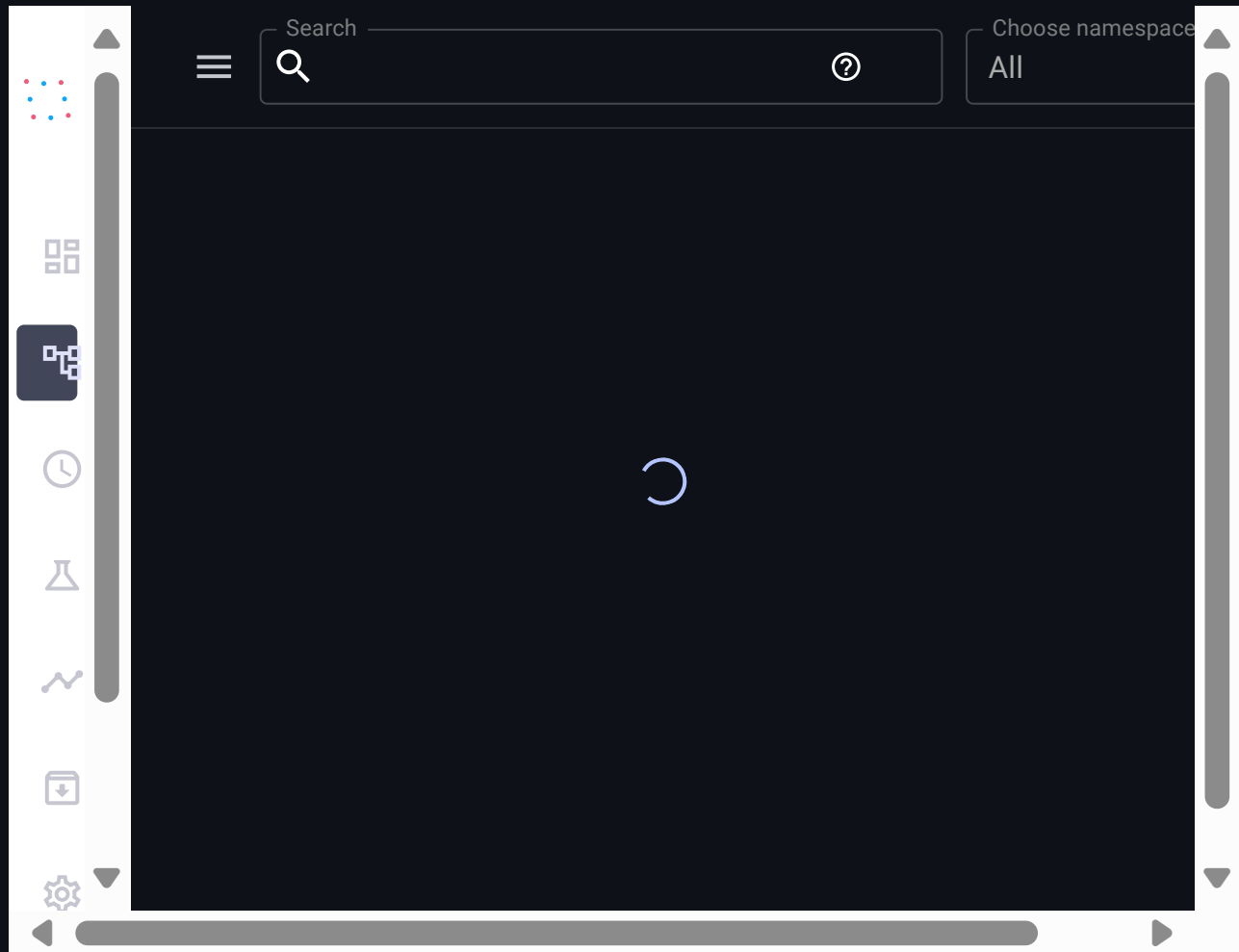
Adjusted unittest

The unit test is specifically targeting the carts-db deployment, which is present in both the previous and reconfigured manifests. Upon comparing the carts-db deployment in both sets of manifests, the only change observed is the addition of resource requests for CPU and memory in the reconfigured manifest. This change does not affect the logic of the unit test, which checks for the number of ready replicas. The test's threshold and logic remain valid as they are based on the number of replicas, which has not changed. Therefore, the unit test does not require any modification.

## Adjusted unittest

The unit test is targeting the 'front-end' deployment in the 'sock-shop' namespace. In the reconfigured manifests, the number of replicas for the 'front-end' deployment has been increased from 1 to 2. This change was likely made to ensure that the steady state is maintained with at least 1 active pod at all times, even if one pod fails. However, the unit test is designed to check if there is at least 1 ready replica, which aligns with the intent of maintaining a steady state. Therefore, the threshold value in the unit test should remain unchanged, as it is still valid to check for at least 1 ready replica. No modification to the unit test is required.

## Completed the chaos experiment!



Your k8s yaml already has good resilience!!!

## Phase EX: Postprocessing

### Summary of your k8s yaml

The Chaos Engineering cycle involves a detailed examination and testing of a Kubernetes-based microservices application, specifically the Sock Shop e-commerce platform. The cycle begins with understanding the user inputs, which include various Kubernetes manifests that

define the deployment and service configurations for different components of the application, such as carts, catalogue, orders, payment, and user services, along with their respective databases.

The system's resiliency issues are identified, including missing resource requests, single replica deployments, high initial delays for liveness probes, and missing port specifications in services. These issues could lead to potential downtime or connectivity problems under certain conditions.

The Chaos Engineering experiment is designed to test the hypothesis that the system can maintain its steady states even when faults are injected. Two steady states are defined: ensuring the 'carts-db' deployment maintains at least 2 ready replicas 95% of the time, and the 'front-end' deployment maintains at least 1 ready replica 100% of the time.

The experiment is divided into three phases: pre-validation, fault-injection, and post-validation, all to be completed within 1 minute. The pre-validation phase checks the system's steady state before fault injection. The fault-injection phase simulates a Black Friday sale scenario using Chaos Mesh, injecting StressChaos on the 'carts-db' deployment and PodChaos on the 'front-end' deployment. The post-validation phase verifies if the system returns to its steady state after the faults are removed.

The first experiment attempt revealed that while the 'carts-db' component maintained its steady state, the 'front-end' component failed to do so during the fault-injection and post-validation phases. The analysis suggested increasing redundancy for the 'front-end' deployment and defining resource requests for the 'carts-db' deployment.

After implementing these improvements, the second experiment attempt was successful, with all unit tests passing, indicating that the system could maintain its steady states under the simulated fault conditions.

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