



## 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-  
workflow.chaos-mesh.org "chaos-experiment-20241127-042755" deleted  
$ kubectl delete workflownode --all --context kind-chaos-eater-cluster -n cl  
workflownode.chaos-mesh.org "fault-injection-phase-fnsrg" deleted  
workflownode.chaos-mesh.org "fault-networkchaos-k96c7" deleted  
workflownode.chaos-mesh.org "fault-podchaos-w48f4" deleted  
workflownode.chaos-mesh.org "fault-stresschaos-k7a12" deleted
```

Input instructions for your Chaos Engineering



```
workflownode.chaos-mesh.org "post-validation-phase-zrh5v" deleted
workflownode.chaos-mesh.org "pre-unittest-carts-db-pod-count-cxbsp" deleted
workflownode.chaos-mesh.org "pre-unittest-front-end-pod-count-w59js" deleted
workflownode.chaos-mesh.org "pre-validation-parallel-workflows-5j2qk" deleted
workflownode.chaos-mesh.org "pre-validation-phase-zzk86" deleted
workflownode.chaos-mesh.org "the-entry-jmpwk" deleted
$ 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
pod "post-unittest-carts-db-pod-count-59lpp-kjfk" deleted
pod "post-unittest-front-end-pod-count-lf98f-z8c58" deleted
pod "pre-unittest-carts-db-pod-count-cxbsp-nhmrd" deleted
pod "pre-unittest-front-end-pod-count-w59js-5ccpz" deleted
$ 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=chaos-eater
pod "carts-56cc746557-d4jm9" deleted
pod "carts-56cc746557-dvqt5" deleted
pod "carts-db-84dd74485f-jn289" deleted
pod "carts-db-84dd74485f-pggt5" deleted
pod "catalogue-8695b4dcfd-tldhj" deleted
pod "catalogue-8695b4dcfd-vbml7" deleted
pod "catalogue-db-68bb48f867-pbnwt" deleted
pod "catalogue-db-68bb48f867-rg594" deleted
pod "front-end-c669bb67f-xpgbl" deleted
pod "orders-697c586dd7-952x5" deleted
pod "orders-697c586dd7-npwkz" deleted
pod "orders-db-694d59df67-f972v" deleted
pod "orders-db-694d59df67-hc2tj" deleted
pod "payment-6b5cf84897-mn7d6" deleted
pod "payment-6b5cf84897-x5phb" deleted
pod "queue-master-85c79fbdf8-7hlbz" deleted
pod "queue-master-85c79fbdf8-pnlw8" deleted
pod "rabbitmq-c5b8d94c7-7l2ng" deleted
pod "rabbitmq-c5b8d94c7-msw7n" deleted
pod "session-db-65c8df9f69-7xsg7" deleted
pod "session-db-65c8df9f69-k69hd" deleted
pod "shipping-6cc64f8975-cl4ql" deleted
pod "shipping-6cc64f8975-zz945" deleted
pod "user-84fb6b864c-56gzh" deleted
pod "user-84fb6b864c-lptjk" deleted
pod "user-db-745c77dd65-9bnf9" deleted
pod "user-db-745c77dd65-dsbvq" deleted
service "carts" deleted
```

```
service "carts-db" deleted
service "catalogue" deleted
service "catalogue-db" deleted
service "front-end" deleted
service "orders" deleted
service "orders-db" deleted
service "payment" deleted
service "queue-master" deleted
service "rabbitmq" deleted
service "session-db" deleted
service "shipping" deleted
service "user" deleted
service "user-db" deleted
deployment.apps "carts" deleted
deployment.apps "carts-db" deleted
deployment.apps "catalogue" deleted
deployment.apps "catalogue-db" deleted
deployment.apps "front-end" deleted
deployment.apps "orders" deleted
deployment.apps "orders-db" deleted
deployment.apps "payment" deleted
deployment.apps "queue-master" deleted
deployment.apps "rabbitmq" deleted
deployment.apps "session-db" deleted
deployment.apps "shipping" deleted
deployment.apps "user" deleted
deployment.apps "user-db" deleted
replicaset.apps "carts-56cc746557" deleted
replicaset.apps "carts-db-84dd74485f" deleted
replicaset.apps "catalogue-8695b4dcfd" deleted
replicaset.apps "catalogue-db-68bb48f867" deleted
replicaset.apps "front-end-c669bb67f" deleted
replicaset.apps "orders-697c586dd7" deleted
replicaset.apps "orders-db-694d59df67" deleted
replicaset.apps "payment-6b5cf84897" deleted
replicaset.apps "queue-master-85c79fbdf8" deleted
replicaset.apps "rabbitmq-c5b8d94c7" deleted
replicaset.apps "session-db-65c8df9f69" deleted
replicaset.apps "shipping-6cc64f8975" deleted
replicaset.apps "user-84fb6b864c" deleted
replicaset.apps "user-db-745c77dd65" deleted
```

**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

```

```
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
```

```

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

```

## 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 190ns
- namespace/sock-shop unchanged
- Warning: spec.template.spec.nodeSelector[beta.kubernetes.io/os]: deprecated
- deployment.apps/carts created
- service/carts created
- deployment.apps/carts-db created
- service/carts-db created
- deployment.apps/catalogue created
- service/catalogue created
- deployment.apps/catalogue-db created
- service/catalogue-db created

```



- deployment.apps/front-end created
- service/front-end created
- deployment.apps/orders created
- service/orders created
- deployment.apps/orders-db created
- service/orders-db created
- deployment.apps/payment created
- service/payment created
- deployment.apps/queue-master created
- service/queue-master created
- deployment.apps/rabbitmq created
- service/rabbitmq created
- deployment.apps/session-db created
- service/session-db created
- deployment.apps/shipping created
- service/shipping created
- deployment.apps/user created
- service/user created
- deployment.apps/user-db created
- service/user-db created

Waiting for deployments to stabilize...

- sock-shop:deployment/carts is ready. [13/14 deployment(s) still pending]
- sock-shop:deployment/queue-master is ready. [12/14 deployment(s) still pending]
- sock-shop:deployment/carts-db: creating container carts-db
  - sock-shop:pod/carts-db-55b978876b-mdpml: creating container carts-db
- sock-shop:deployment/shipping 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/user: creating container user
  - sock-shop:pod/user-69f4d49b54-2lp5j: creating container user
  - sock-shop:pod/user-69f4d49b54-vpkn1: creating container user
- sock-shop:deployment/user-db: creating container user-db
  - sock-shop:pod/user-db-68746cbc74-h4xq4: creating container user-db
  - sock-shop:pod/user-db-68746cbc74-wnrct: creating container user-db
- sock-shop:deployment/carts-db is ready. [7/14 deployment(s) still pending]
- sock-shop:deployment/user-db 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 is ready. [4/14 deployment(s) still pending]
- sock-shop:deployment/user: waiting for rollout to finish: 0 of 2 updated
- sock-shop:deployment/front-end is ready. [3/14 deployment(s) still pending]
- sock-shop:deployment/user is ready. [2/14 deployment(s) still pending]
- sock-shop:deployment/catalogue is ready. [1/14 deployment(s) still pending]
- sock-shop:deployment/payment is ready.

Deployments stabilized in 3 minutes 4.334 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-658f596954-7zxc2	1/1	Running	0
sock-shop	pod/carts-658f596954-tflvf	1/1	Running	0
sock-shop	pod/carts-db-55b978876b-mdpml	1/1	Running	0
sock-shop	pod/carts-db-55b978876b-p82bt	1/1	Running	0
sock-shop	pod/catalogue-594ddbdbb9-kblmb	1/1	Running	0
sock-shop	pod/catalogue-594ddbdbb9-tm2rv	1/1	Running	0
sock-shop	pod/catalogue-db-77c4db4876-2pqm	1/1	Running	0
sock-shop	pod/catalogue-db-77c4db4876-bklsr	1/1	Running	0
sock-shop	pod/front-end-79cdc649b4-zzn4t	1/1	Running	0
sock-shop	pod/orders-59f55567b6-487r7	1/1	Running	0
sock-shop	pod/orders-59f55567b6-j2st6	1/1	Running	0
sock-shop	pod/orders-db-5dbb89b648-nmzps	1/1	Running	0
sock-shop	pod/orders-db-5dbb89b648-x4zks	1/1	Running	0
sock-shop	pod/payment-7b6d6477c8-5r79f	1/1	Running	0
sock-shop	pod/payment-7b6d6477c8-jjmjw	1/1	Running	0
sock-shop	pod/queue-master-59d99497f5-9ks4l	1/1	Running	0
sock-shop	pod/queue-master-59d99497f5-x4p8m	1/1	Running	0
sock-shop	pod/rabbitmq-d57757696-9ffnc	2/2	Running	0
sock-shop	pod/rabbitmq-d57757696-vqp6z	2/2	Running	0
sock-shop	pod/session-db-5696479b94-2frqx	1/1	Running	0
sock-shop	pod/session-db-5696479b94-pj2ts	1/1	Running	0
sock-shop	pod/shipping-68f8f8b566-ngcjm	1/1	Running	0
sock-shop	pod/shipping-68f8f8b566-qzhkw	1/1	Running	0
sock-shop	pod/user-69f4d49b54-2lp5j	1/1	Running	0
sock-shop	pod/user-69f4d49b54-vpknl	1/1	Running	0
sock-shop	pod/user-db-68746cbc74-h4xq4	1/1	Running	0
sock-shop	pod/user-db-68746cbc74-wnrtc	1/1	Running	0

## 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 in 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 settings ensure the container runs 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/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 the default port for 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 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 labeled with 'name: carts-db'.

- The Service is created in the 'sock-shop' namespace.
- It exposes port 27017 and directs traffic to the same port on the target pods.
- The Service selects pods with the label 'name: carts-db' to route traffic to them.

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

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'catalogue' and is part of 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 initial delays of 300 and 180 seconds respectively.
- 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 to enable Prometheus scraping for monitoring purposes.
- 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 uses a selector to target pods with the label 'name: catalogue'.

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

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'catalogue-db'.
- It is located in the 'sock-shop' namespace.
- The Deployment will create 2 replicas of the pod.
- Each pod will run a container from the image 'weaveworksdemos/catalogue-db:0.3.0'.
- The container is configured with environment variables for MySQL, including a root password and database name.
- The container exposes port 3306, which is commonly used for MySQL.

- The pods are scheduled to run on nodes with the Linux operating system.

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

- This manifest defines a Kubernetes Service.
- The Service is named 'catalogue-db'.
- It is associated with the 'sock-shop' namespace.
- The Service listens on port 3306 and forwards traffic to the same port on the target pods.
- It uses a selector to target 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 '/' 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 '/' 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, allowing external access to the service.

- The Service is configured to be scraped by Prometheus for monitoring, as indicated by the annotation 'prometheus.io/scrape: true'.
- It selects pods with the label 'name: front-end' to route traffic to.

`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 'weaveworksdemos/orders:0.4.7' Docker image for the container.
- 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 container listens on port 80.
- 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 to enable Prometheus scraping with 'prometheus.io/scrape: true'.
- 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 target pods.
- It uses a selector to match pods with the label 'name: orders'.

`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 2 replicas of the 'orders-db' pod to be created.
- The pods are labeled with 'name: orders-db' for identification and selection.
- Each pod runs a single container using the 'mongo' image.
- The container exposes port 27017, which is the default port for MongoDB.
- Security settings are applied to drop all capabilities and add only CHOWN, SETGID, and SETUID.

- The root filesystem of the container is set to 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/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 match 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 part of 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, drops all capabilities except 'NET\_BIND\_SERVICE', and uses a read-only root filesystem.
- Liveness and readiness probes are configured to check the '/health' endpoint on port 80, with initial delays of 300 and 180 seconds respectively, and a period of 3 seconds.
- The Deployment is scheduled 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 for monitoring.
- 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 selects 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 (instances) 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 CPU limit of 300m and memory limit of 500Mi, and requests for 100m CPU and 300Mi memory.
- The container exposes port 80 for network traffic.
- 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' to route traffic to.

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

- This manifest defines a Deployment for RabbitMQ in Kubernetes.
- It is set to run in the 'sock-shop' namespace.
- The Deployment is named 'rabbitmq' and is labeled accordingly.
- It specifies 2 replicas, meaning there will be 2 instances of RabbitMQ running.
- 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 a RabbitMQ exporter.
- The RabbitMQ container uses the image 'rabbitmq:3.6.8-management'.
- It exposes two ports: 15672 for management and 5672 for RabbitMQ operations.
- Security context is set to drop all capabilities and add specific ones like CHOWN, SETGID, SETUID, and DAC\_OVERRIDE.
- The root filesystem is set to read-only for security purposes.
- The RabbitMQ exporter container uses the image 'kbudde/rabbitmq-exporter' and exposes port 9090.
- The Deployment is configured to run on nodes with the label 'beta.kubernetes.io/os: 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 labeled with 'name: rabbitmq'.
- It is deployed in the 'sock-shop' namespace.
- The Service exposes two ports: 5672 for RabbitMQ and 9090 for an exporter.
- The protocol used for the ports is TCP.
- The Service 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 that there should be 2 replicas of the 'session-db' pod running.
- The pods are selected based on the label 'name: session-db'.
- Each pod runs a single 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, with a read-only root filesystem for enhanced security.
- 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 is configured to expose port 6379.
- It targets the same port (6379) on the selected pods.
- The Service uses a selector to match 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 that there should be 2 replicas of the 'shipping' application running.
- The Deployment uses the Docker image 'weaveworksdemos/shipping:0.4.8'.
- Environment variables are set for the application, including 'ZIPKIN' and 'JAVA\_OPTS'.

- Resource limits and requests are defined, with limits set to 300m CPU and 500Mi memory, and requests set to 100m CPU and 300Mi memory.
- The application listens on port 80.
- Security context is configured to run the container as a non-root user with user ID 10001, and it drops all capabilities except 'NET\_BIND\_SERVICE'.
- The root filesystem is set to be read-only.
- A temporary volume is mounted at '/tmp', using an in-memory emptyDir volume.
- The Deployment is scheduled to run on nodes with the operating system labeled as 'linux'.

`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' to route traffic to.

`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, with requests for 100m CPU and 100Mi memory.
- The container listens on 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.
- The container has a read-only root filesystem and drops all capabilities except 'NET\_BIND\_SERVICE'.
- Liveness and readiness probes are configured to check the '/health' endpoint on port 80.
- The liveness probe starts after 300 seconds and checks every 3 seconds.
- The readiness probe starts after 180 seconds and checks every 3 seconds.
- The Deployment is scheduled to run on nodes with the Linux operating system.

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

- This manifest defines a Kubernetes Service.

- The Service is named 'user'.
- It is annotated for Prometheus scraping, which means it is set up for monitoring.
- 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, labeled as 'mongo', which is typically used for MongoDB.
- Security settings are applied to drop all capabilities and only add CHOWN, SETGID, and SETUID, with a read-only root filesystem.
- 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/28-user-db-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'user-db'.
- It is labeled with 'name: user-db'.
- The Service is created in the 'sock-shop' namespace.
- It exposes port 27017 and directs traffic to the same port on the target pods.
- The Service selects pods with the label 'name: user-db' to route traffic to them.

## 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: Missing Liveness and Readiness Probes

- details: Without liveness and readiness probes, Kubernetes cannot determine the health of the containers, which may lead to serving traffic to unhealthy pods.
- manifests having the issues: ['sock-shop-2/manifests/01-carts-dep.yaml', 'sock-shop-2/manifests/03-carts-db-dep.yaml', 'sock-shop-2/manifests/05-catalogue-dep.yaml', 'sock-shop-2/manifests/07-catalogue-db-dep.yaml', 'sock-shop-2/manifests/11-orders-dep.yaml', 'sock-shop-2/manifests/13-orders-db-dep.yaml', 'sock-shop-2/manifests/15-payment-dep.yaml', 'sock-shop-2/manifests/17-queue-master-dep.yaml', 'sock-shop-2/manifests/19-rabbitmq-dep.yaml', 'sock-shop-2/manifests/21-session-db-dep.yaml', 'sock-shop-2/manifests/23-shipping-dep.yaml', 'sock-shop-2/manifests/25-user-dep.yaml', 'sock-shop-2/manifests/27-user-db-dep.yaml']
- problematic config: The deployments for carts, carts-db, catalogue, catalogue-db, orders, orders-db, payment, queue-master, rabbitmq, session-db, shipping, user, and user-db do not have liveness or readiness probes configured.

## Issue #3: Hardcoded Environment Variables

- details: Hardcoded environment variables, such as passwords, can lead to security vulnerabilities.
- manifests having the issues: ['sock-shop-2/manifests/07-catalogue-db-dep.yaml']
- problematic config: env: MYSQL\_ROOT\_PASSWORD is hardcoded with value: fake\_password.

## Application of the manifests:

### Thoughts:

The manifests provided are part of a well-known microservices demo application called 'Sock Shop'. This application is designed to demonstrate the use of microservices architecture and Kubernetes orchestration. The application simulates an e-commerce website for selling socks, with various services such as carts, catalogue, orders, payment, and user management. The file names, labels, and Docker images used in the manifests (e.g., 'weaveworksdemos') are indicative of the Sock Shop application, which is a common reference application in the Kubernetes community.

### Assumed application:

Sock Shop is an e-commerce application that simulates an online sock store. It offers users the ability to browse a catalogue of socks, add items to a cart, place orders, and manage user accounts. The application is composed of multiple microservices, each responsible for a specific function, such as managing the product catalogue, handling user sessions, processing payments, and managing orders.

## 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

#### ✓ Steady state #1: carts-db-replicas

##### Description

The first issue to address is the 'Missing Resource Requests' for 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. To ensure the system's steady state, we need to verify that the 'carts-db' deployment maintains the expected number of replicas, which is 2, even under resource constraints. This can be measured by checking the number of running pods for the 'carts-db' deployment.

## Current state inspection

To inspect the number of replicas for the 'carts-db' deployment, I will use the Kubernetes API. This is because the task involves checking the current state of a Kubernetes resource, specifically the number of running pods for a deployment, which is best achieved using the Kubernetes client libraries. The script will check the number of replicas every second for a duration of 5 seconds, which is sufficient to verify the steady state of the deployment.

tool: `k8s` duration: `5s`

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

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

def check_carts_db_replicas(namespace='sock-shop', deployment_name='carts-db'):
    # 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"Total 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__':
    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(duration=args.duration)
```

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

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

```
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
```

## ► Threshold

The steady state we are considering is the number of replicas for the 'carts-db' deployment. The current state shows that there are 2 total replicas and 2 ready replicas consistently over a period of 5 seconds. This indicates that the system is currently stable with all replicas being ready. Given that the deployment is configured to have 2 replicas, the threshold should ensure that both replicas are ready to maintain the steady state. To account for some fluctuations, we can set a tolerance that allows for a brief period where not all replicas are ready, but the system should recover quickly. A reasonable threshold would be to have at least 1 ready replica 100% of the time and both replicas ready at least 80% of the time during the monitoring period. This ensures that the system remains operational even if one replica temporarily becomes unavailable.

At least 1 ready replica 100% of the time and 2 ready replicas at least 80% of the time during the monitoring period.

## 📄 Unit test to validate the steady state

To verify the steady state of the 'carts-db' deployment, I will write a unit test that checks the number of ready replicas over a specified duration. The test will ensure that there is at least 1 ready replica 100% of the time and 2 ready replicas at least 80% of the time. I will implement a loop that checks the status every second for the specified duration and use assertions to validate the threshold conditions. The test will be run from the command line with a '--duration' argument to specify the monitoring period.

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

class TestCartsDBReplicas(K8sAPIBase):
    def __init__(self, namespace='sock-shop', deployment_name='carts
        super().__init__()
        self.namespace = namespace
```

```

self.deployment_name = deployment_name
self.duration = duration

def test_replicas_threshold(self):
    # Initialize counters for ready replicas
    total_checks = 0
    ready_replicas_count = 0
    fully_ready_replicas_count = 0

    for _ in range(self.duration):
        try:
            # Read the deployment status
            deployment = self.v1.read_namespaced_deployment(self.deployment_name, self.namespace)
            replicas = deployment.status.replicas
            ready_replicas = deployment.status.ready_replicas
            print(f"Total replicas: {replicas}, Ready replicas: {ready_replicas}")

            # Increment the total checks
            total_checks += 1

            # Check if at least 1 replica is ready
            if ready_replicas >= 1:
                ready_replicas_count += 1

            # Check if both replicas are ready
            if ready_replicas == 2:
                fully_ready_replicas_count += 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 percentage of time conditions are met
    one_ready_percentage = (ready_replicas_count / total_checks) * 100
    two_ready_percentage = (fully_ready_replicas_count / total_checks) * 100

    # Assert the threshold conditions
    assert one_ready_percentage == 100, "At least 1 ready replica was not observed"
    assert two_ready_percentage >= 80, "2 ready replicas were not observed for 80% of the time"

    print("Test passed: Steady state conditions are satisfied.")

def main():
    parser = argparse.ArgumentParser(description='Test carts-db replicas threshold')
    parser.add_argument('--deployment-name', type=str, required=True, help='Deployment name')
    parser.add_argument('--namespace', type=str, required=True, help='Namespace')
    parser.add_argument('--duration', type=int, required=True, help='Duration in seconds')
    args = parser.parse_args()

    client = kubernetes.client.ApiClient()
    v1 = kubernetes.client.AppsV1Api(client)

    test = TestReplicasThreshold(v1, args.deployment_name, args.namespace, args.duration)
    test.test_replicas_threshold()

```



```

parser.add_argument('--duration', type=int, default=5, help='Duration of the test')
args = parser.parse_args()

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

if __name__ == '__main__':
    main()

```

Pod `unittestcarts-db-replicasm0d0-pod` has failed.

```

Traceback (most recent call last):
  File "/chaos-eater/sandbox/cycle_20241127_043136/hypothesis/unittest.py", line 10, in <module>
    main()
  File "/chaos-eater/sandbox/cycle_20241127_043136/hypothesis/unittest.py", line 20, in main
    test.test_replicas_threshold()
  File "/chaos-eater/sandbox/cycle_20241127_043136/hypothesis/unittest.py", line 30, in test_replicas_threshold
    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 called 'read\_namespaced\_deployment'. The correct API to use for reading deployments is 'AppsV1Api'. 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 access the deployment information.

```

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

class TestCartsDBReplicas(K8sAPIBase):
    def __init__(self, namespace='sock-shop', deployment_name='carts-db-replicasm0d0', duration=5):
        super().__init__()
        self.namespace = namespace
        self.deployment_name = deployment_name
        self.duration = duration
        # Use the correct API client for deployments
        self.apps_v1 = client.AppsV1Api()

    def test_replicas_threshold(self):

```

```

# Initialize counters for ready replicas
total_checks = 0
ready_replicas_count = 0
fully_ready_replicas_count = 0

for _ in range(self.duration):
    try:
        # Read the deployment status using the correct API client
        deployment = self.apps_v1.read_namespaced_deployment(
            name=self.deployment_name, namespace=self.namespace
        )
        replicas = deployment.status.replicas
        ready_replicas = deployment.status.ready_replicas
        print(f"Total replicas: {replicas}, Ready replicas: {ready_replicas}")

        # Increment the total checks
        total_checks += 1

        # Check if at least 1 replica is ready
        if ready_replicas >= 1:
            ready_replicas_count += 1

        # Check if both replicas are ready
        if ready_replicas == 2:
            fully_ready_replicas_count += 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 percentage of time conditions are met
one_ready_percentage = (ready_replicas_count / total_checks) * 100
two_ready_percentage = (fully_ready_replicas_count / total_checks) * 100

# Assert the threshold conditions
assert one_ready_percentage == 100, "At least 1 ready replica was not observed"
assert two_ready_percentage >= 80, "2 ready replicas were not observed"

print("Test passed: Steady state conditions are satisfied.")

```

```

def main():
    parser = argparse.ArgumentParser(description='Test carts-db replicas')
    parser.add_argument('--duration', type=int, default=5, help='Duration of the test in seconds')
    args = parser.parse_args()

```

```

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

```

```
test = TestCartsDBReplicas(duration=args.duration)
# Run the test
test.test_replicas_threshold()

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

Pod `unittestcarts-db-replicasmod1-pod` has completed successfully.

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

```
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Total replicas: 2, Ready replicas: 2
Test passed: Steady state conditions are satisfied.
```

## ✓ Steady state #2: front-end-replica

### Description

The next issue to address is the 'Single Replica Deployment' for the 'front-end' deployment. This is a critical issue because having only one replica can lead to downtime if the pod fails. To ensure the system's steady state, we need to verify that the 'front-end' deployment maintains at least one running pod at all times. This can be measured by checking the number of running pods for the 'front-end' deployment. Given that the deployment is configured to have only 1 replica, the threshold should ensure that this single replica is always ready to maintain the steady state. A reasonable threshold would be to have at least 1 ready replica 100% of the time during the monitoring period. This ensures that the system remains operational even if there are temporary issues with the pod.

### Current state inspection

To inspect the 'front-end-replica' state, I will use the K8s API to check the number of ready replicas for the 'front-end' deployment in the 'sock-shop' namespace. This is because the task involves verifying the number of running pods, which is a Kubernetes resource state check. The K8s API is suitable for this purpose as it allows us to directly

query the deployment status and ensure that the single replica is always ready. The script will run for a duration of 5 seconds, checking the status every second, to ensure the deployment maintains at least one ready replica 100% of the time.

tool: `k8s` duration: `5s`

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

```
import os
import time
import argparse
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()
    ready_replicas_count = 0

    for _ in range(duration):
        resp = v1.read_namespaced_deployment_status(deployment_name, namespace)
        ready_replicas = resp.status.ready_replicas or 0
        print(f"Ready replicas for {deployment_name}: {ready_replicas}")
        if ready_replicas >= 1:
            ready_replicas_count += 1
        time.sleep(1)

    print(f"{deployment_name} was ready {ready_replicas_count}/{duration} seconds")

if __name__ == '__main__':
    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-replica-pod` has completed successfully.

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

```
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
```

```
Ready replicas for front-end: 1  
front-end was ready 5/5 times.
```

## 🚩 Threshold

The steady state we are considering is the availability of the 'front-end' deployment, which currently has only one replica. The current state shows that the 'front-end' deployment has 1 ready replica consistently over a monitoring period of 5 seconds. To ensure the system remains operational, the threshold should be set such that the 'front-end' deployment maintains at least 1 ready replica 100% of the time during the monitoring period. This accounts for the fact that there is only one replica, and any downtime would mean the service is unavailable. Given the current state satisfies this condition, the threshold should be set to ensure that at least 1 ready replica is present 100% of the time during the monitoring period.

At least 1 ready replica must be present 100% of the time during the monitoring period.

## 📄 Unit test to validate the steady state

To verify the steady state of the 'front-end' deployment, I will write a unit test that checks if there is at least 1 ready replica 100% of the time during the specified monitoring period. The test will use a for-loop to check the status every second for the given duration. The threshold is set to ensure that the 'front-end' deployment maintains at least 1 ready replica 100% of the time, which means the number of times the ready replicas are 1 or more should equal the duration. I will use the Kubernetes Python API to interact with the cluster and check the deployment status. The test will include an assertion to verify that the steady state satisfies the threshold, and it will print a summary of the results at the end.

```
import os  
import time  
import argparse  
from kubernetes import client, config  
from unittest_base import K8sAPIBase  
  
class TestFrontEndReplica(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):
    ready_replicas_count = 0

    # Loop for the specified duration
    for _ in range(self.duration):
        # Get the deployment status
        resp = self.v1.read_namespaced_deployment_status(self.deployment_name, self.namespace)
        ready_replicas = resp.status.ready_replicas or 0
        print(f"Ready replicas for {self.deployment_name}: {ready_replicas}")

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

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

    # Calculate the percentage of time the deployment was ready
    readiness_percentage = (ready_replicas_count / self.duration) * 100
    print(f"{self.deployment_name} was ready {readiness_percentage}% of the time")

    # Assert that the deployment was ready 100% of the time
    assert readiness_percentage == 100, f"{self.deployment_name} was not ready 100% of the time"

def main():
    parser = argparse.ArgumentParser(description='Test front-end replica readiness')
    parser.add_argument('--duration', type=int, default=5, help='Duration of the test in seconds')
    args = parser.parse_args()

    # Create a test instance and run the test
    test = TestFrontEndReplica('sock-shop', 'front-end', args.duration)
    test.test_steady_state()

if __name__ == '__main__':
    main()

```

Pod `unittestfront-end-replicamod0-pod` has failed.

```

Traceback (most recent call last):
  File "/chaos-eater/sandbox/cycle_20241127_043136/hypothesis/unittest.py", line 10, in <module>
    main()
  File "/chaos-eater/sandbox/cycle_20241127_043136/hypothesis/unittest.py", line 20, in main
    test.test_steady_state()
  File "/chaos-eater/sandbox/cycle_20241127_043136/hypothesis/unittest.py", line 30, in test_steady_state
    resp = self.v1.read_namespaced_deployment_status(self.deployment_name, self.namespace)

```

```
resp = self.v1.read_namespaced_deployment_status(self.deployment_name, self.namespace)
AttributeError: 'CoreV1Api' object has no attribute 'read_namespaced_deployment_status'
```

The error occurs because the `CoreV1Api` class does not have a method called `read_namespaced_deployment_status`. Instead, the correct class to use for reading deployment status is `AppsV1Api`. I will fix the error by changing the API client from `CoreV1Api` to `AppsV1Api` in the `K8sAPIBase` class. This will allow the test to correctly call the `read_namespaced_deployment_status` method and check the deployment status. The rest of the test logic remains the same, ensuring that the deployment maintains at least 1 ready replica 100% of the time during the specified duration.

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

class TestFrontEndReplica(K8sAPIBase):
    def __init__(self, namespace, deployment_name, duration):
        super().__init__()
        self.namespace = namespace
        self.deployment_name = deployment_name
        self.duration = duration
        # Use AppsV1Api to interact with deployments
        self.apps_v1 = client.AppsV1Api()

    def test_steady_state(self):
        ready_replicas_count = 0

        # Loop for the specified duration
        for _ in range(self.duration):
            # Get the deployment status
            resp = self.apps_v1.read_namespaced_deployment_status(self.deployment_name, self.namespace)
            ready_replicas = resp.status.ready_replicas or 0
            print(f"Ready replicas for {self.deployment_name}: {ready_replicas}")

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

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

        # Calculate the percentage of time the deployment was ready
        readiness_percentage = (ready_replicas_count / self.duration) * 100
        print(f"Readiness percentage: {readiness_percentage}%")
```

```

        print(f"{self.deployment_name} was ready {ready_replicas_count} times")

        # Assert that the deployment was ready 100% of the time
        assert readiness_percentage == 100, f"{self.deployment_name} was not ready 100% of the time"

def main():
    parser = argparse.ArgumentParser(description='Test front-end replica readiness')
    parser.add_argument('--duration', type=int, default=5, help='Duration of the test in seconds')
    args = parser.parse_args()

    # Create a test instance and run the test
    test = TestFrontEndReplica('sock-shop', 'front-end', args.duration)
    test.test_steady_state()

if __name__ == '__main__':
    main()

```

Pod `unittestfront-end-replicamod1-pod` has completed successfully.

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

```

Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
Ready replicas for front-end: 1
front-end was ready 5/5 times.

```

## Fault definition

### ✓ Scenario: Black Friday Sale

#### Description

During a Black Friday sale, the system is expected to handle a significant increase in traffic, which can stress the system's resources and reveal weaknesses. The 'carts-db' deployment lacks resource requests, which can lead to scheduling issues under resource pressure. The 'front-end' deployment has only one replica, making it a single point of failure. To simulate the Black Friday event, we will first inject a 'StressChaos' fault to simulate high CPU usage on the 'carts-db' pods, testing if the system can



maintain the steady state of having at least 1 ready replica 100% of the time and 2 ready replicas 80% of the time. Next, we will inject a 'PodChaos' fault to kill the 'front-end' pod, testing if the system can maintain the steady state of having at least 1 ready replica 100% of the time. This sequence simulates the increased load and potential failures during a high-traffic event like Black Friday.

## Fault-injection sequence

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

## Detailed fault parameters

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

```
{
  "mode" : "all"
  "selector" : {
    "namespaces" : [
      0 : "sock-shop"
    ]
    "labelSelectors" : {
      "name" : "carts-db"
    }
  }
  "stressors" : {
    "cpu" : {
      "workers" : 2
      "load" : 80
    }
  }
  "containerNames" : [
    0 : "carts-db"
  ]
}
```

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

```
{
  "action" : "pod-kill"
```

```
"mode" : "one"
  "selector" : {
    "namespaces" : [
      0 : "sock-shop"
    ]
    "labelSelectors" : {
      "name" : "front-end"
    }
  }
  "value" : "1"
}
```

## Phase 2: Chaos Experiment

### CE experiment Planning Completed!

#### Chaos Engineering Experiment Plan

##### Time Schedule

The chaos engineering experiment is designed to test the system's resilience under stress conditions and potential failures. Given the constraints, the experiment must be completed within 1 minute. The experiment is divided into three phases: pre-validation, fault-injection, and post-validation.

1. **Pre-validation Phase:** This phase ensures that the system is in a steady state before any faults are injected. We will allocate 20 seconds for this phase. During this time, we will verify that the 'carts-db' deployment maintains at least 1 ready replica 100% of the time and 2 ready replicas 80% of the time, and that the 'front-end' deployment maintains at least 1 ready replica 100% of the time.
2. **Fault-injection Phase:** This phase involves injecting faults to simulate the Black Friday event. We will allocate 20 seconds for this phase. The faults include a 'StressChaos' fault to simulate high CPU usage on the 'carts-db' pods and a 'PodChaos' fault to kill the 'front-end' pod. These faults will test the system's ability to maintain the defined steady states under stress and failure conditions.
3. **Post-validation Phase:** This phase ensures that the system returns to its steady state after the faults are removed. We will allocate 20 seconds for this phase. During this time, we will again verify the steady states for the 'carts-db' and 'front-end' deployments to ensure they meet the defined thresholds.

The total time for the experiment is 60 seconds, with each phase receiving an equal allocation of 20 seconds. This allocation allows for a balanced approach to validating the system's steady state, injecting faults, and confirming recovery.

Total experiment time: 60s

Pre-validation Phase: 20s

Fault-injection Phase: 20s

Post-validation Phase: 20s

### Pre-validation Phase (20s)

In the pre-validation phase, we need to ensure that the system is in a steady state before we introduce any faults. This involves verifying that the current state of the system meets the defined steady state thresholds. Given the constraints of a 20-second total time for this phase, we will execute the unit tests for both steady states simultaneously to maximize efficiency. The first steady state, 'carts-db-replicas', requires checking that at least 1 replica is ready 100% of the time and 2 replicas are ready at least 80% of the time. The second steady state, 'front-end-replica', requires ensuring that the single replica is ready 100% of the time. Both tests will run for the full 20 seconds, with checks occurring every second. This approach ensures that we have sufficient data to confirm the system's readiness before proceeding to the fault injection phase.

- Verified Steady State #0: carts-db-replicas
  - Workflow Name: pre-unittest-carts-db-replicas
  - Grace Period: 0s
  - Duration: 20s
- Verified Steady State #1: front-end-replica
  - Workflow Name: pre-unittest-front-end-replica
  - Grace Period: 0s
  - Duration: 20s

### Fault-injection Phase (20s)

In this fault-injection phase, we aim to simulate a Black Friday sale scenario by introducing two types of faults: 'StressChaos' and 'PodChaos'. The goal is to observe how the system behaves under stress and potential failure conditions. The phase is

limited to 20 seconds, so we need to carefully time the injections and tests to fit within this window.

First, we will inject a 'StressChaos' fault on the 'carts-db' deployment to simulate high CPU usage. This will help us test the system's ability to maintain the steady state of having at least 1 ready replica 100% of the time and 2 ready replicas 80% of the time. We will start this fault injection immediately at the beginning of the phase and let it run for 10 seconds.

Simultaneously, we will run the unit test for the 'carts-db-replicas' steady state to verify if the system can maintain the required number of ready replicas during the stress period. This test will also start at the beginning of the phase and run for 10 seconds.

Next, we will inject a 'PodChaos' fault to kill the 'front-end' pod. This will test the system's ability to maintain the steady state of having at least 1 ready replica 100% of the time for the 'front-end' deployment. We will start this fault injection at the 10-second mark and let it run for 5 seconds.

Finally, we will run the unit test for the 'front-end-replica' steady state to verify if the system can maintain the required readiness during the pod failure. This test will start at the 10-second mark and run for 5 seconds.

By staggering the fault injections and unit tests in this manner, we can effectively simulate the Black Friday scenario and assess the system's resilience under stress and failure conditions.

- Verified Steady State #0: `carts-db-replicas`
  - Workflow Name: `fault-unittest-carts-db-replicas`
  - Grace Period: `0s`
  - Duration: `10s`
- Verified Steady State #1: `front-end-replica`
  - Workflow Name: `fault-unittest-front-end-replica`
  - Grace Period: `10s`
  - Duration: `5s`
- Injected Faults #0: `StressChaos`
  - Workflow Name: `fault-stresschaos`
  - Grace Period: `0s`
  - Duration: `10s`
- Injected Faults #1: `PodChaos`
  - Workflow Name: `fault-podchaos`
  - Grace Period: `10s`

- Duration: 5s

### Post-validation Phase (20s)

In the post-validation phase, we need to ensure that the system has returned to its steady states after the fault injection. The steady states we are concerned with are the 'carts-db-replicas' and 'front-end-replica'. The post-validation phase will involve running unit tests to verify these steady states. Given the total time of 20 seconds for this phase, we will stagger the unit tests to ensure both steady states are verified within this timeframe.

The 'carts-db-replicas' steady state requires that at least 1 ready replica is available 100% of the time and 2 ready replicas are available at least 80% of the time. We will allocate 10 seconds for this test, with a grace period of 0 seconds, as we want to immediately verify the recovery of the 'carts-db' deployment after the stress test.

The 'front-end-replica' steady state requires that at least 1 ready replica is available 100% of the time. We will allocate the remaining 10 seconds for this test, also with a grace period of 0 seconds, to ensure the 'front-end' deployment has recovered from the pod kill fault.

By staggering the tests in this manner, we ensure that both steady states are verified within the 20-second post-validation phase, allowing us to confirm that the system has returned to its expected operational state.

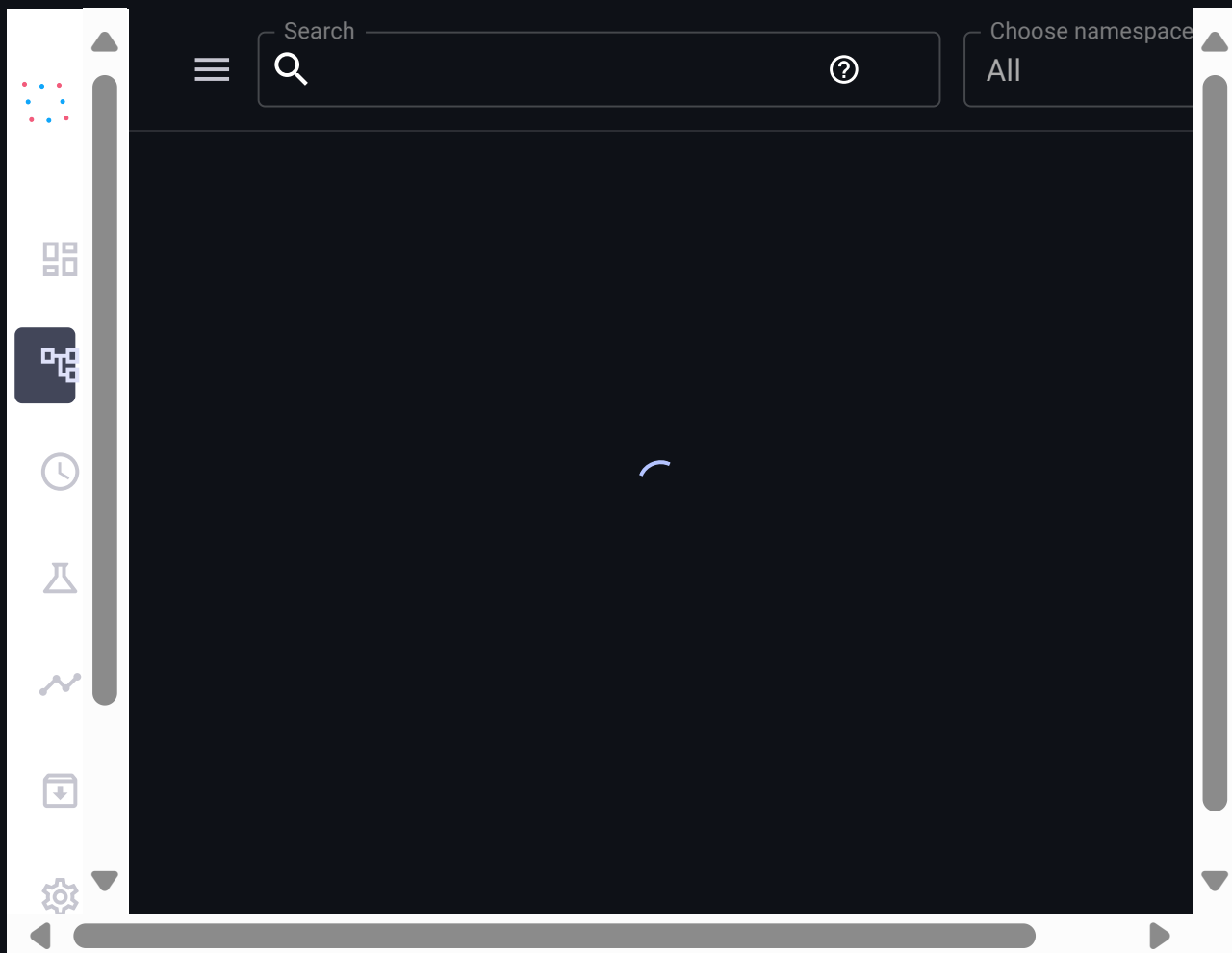
- Verified Steady State #0: carts-db-replicas
  - Workflow Name: post-unittest-carts-db-replicas
  - Grace Period: 0s
  - Duration: 10s
- Verified Steady State #1: front-end-replica
  - Workflow Name: post-unittest-front-end-replica
  - Grace Period: 0s
  - Duration: 10s

### Summary

The chaos engineering experiment is structured into three phases, each lasting 20 seconds, to test the system's resilience. In the pre-validation phase, both unit tests for verifying steady states are executed simultaneously for the entire 20 seconds. The 'carts-db-replicas' test checks that at least one replica is ready 100% of the time and two replicas are ready 80% of the time, while the 'front-end-replica' test ensures one

replica is ready 100% of the time. In the fault-injection phase, the 'StressChaos' fault is injected on the 'carts-db' deployment to simulate high CPU usage, starting immediately and lasting for 10 seconds. Concurrently, the 'carts-db-replicas' unit test runs for 10 seconds to verify readiness under stress. At the 10-second mark, the 'PodChaos' fault is injected to kill the 'front-end' pod, lasting for 5 seconds, while the 'front-end-replica' unit test runs for 5 seconds to check readiness during the fault. In the post-validation phase, the 'carts-db-replicas' unit test runs for the first 10 seconds to verify recovery, followed by the 'front-end-replica' unit test for the next 10 seconds, ensuring both deployments return to their steady states.

## Completed the chaos experiment!



## Phase 3: Analysis

The chaos engineering experiment aimed to test the resilience of the 'sock-shop' system under simulated fault conditions, specifically focusing on the 'carts-db' and 'front-end' deployments. The experiment was structured into three phases: pre-validation, fault-injection, and post-validation. The results indicate that while the 'carts-db' deployment maintained its steady state throughout the experiment, the 'front-end' deployment failed to meet the expected steady state during both the fault-injection and post-validation phases.

### 1. Pre-Validation Phase:

- Both the 'carts-db-replicas' and 'front-end-replica' tests passed, indicating that the system was in a stable state before any faults were injected. This confirms that the initial configuration of the system was functioning as expected.

### 2. Fault-Injected Phase:

- The 'carts-db-replicas' test passed during the fault-injection phase, demonstrating that the 'carts-db' deployment was resilient to the 'StressChaos' fault, which simulated high CPU usage. This suggests that despite the lack of resource requests, the 'carts-db' deployment was able to maintain at least one ready replica 100% of the time and two ready replicas 80% of the time.
- The 'front-end-replica' test failed during the fault-injection phase. The 'PodChaos' fault, which involved killing the 'front-end' pod, resulted in 0 ready replicas throughout the test duration. This failure highlights the critical issue of having only a single replica for the 'front-end' deployment, making it a single point of failure. The system was unable to recover or maintain availability when the single pod was terminated.

### 3. Post-Validation Phase:

- The 'carts-db-replicas' test continued to pass in the post-validation phase, indicating that the 'carts-db' deployment returned to its steady state after the fault was removed.
- The 'front-end-replica' test failed again in the post-validation phase, with 0 ready replicas recorded throughout the test. This suggests that the 'front-end' deployment did not recover after the fault was removed, likely due to the absence of additional replicas to take over when the single pod was killed.

### Insights and Recommendations:

- The failure of the 'front-end' deployment to maintain its steady state during and after the fault-injection phase underscores the importance of addressing the 'Single Replica Deployment' issue. To improve resilience, it is recommended to increase the number of replicas for the 'front-end' deployment. This will ensure that the system can maintain availability even if one pod fails.
- Consider implementing horizontal pod autoscaling for the 'front-end' deployment to dynamically adjust the number of replicas based on traffic and resource usage. This can help the system handle sudden spikes in demand, such as those experienced during events like Black Friday.
- Review and potentially adjust the readiness and liveness probes for the 'front-end' deployment to ensure that Kubernetes can accurately determine the health of the pods and take corrective actions if necessary.

By addressing these issues, the system's resilience can be significantly improved, reducing the risk of downtime during high-traffic events and fault scenarios.

# Phase 4: Improvement

## Reconfiguration

To improve the resiliency of the 'front-end' deployment and avoid failures during fault scenarios, we need to address the 'Single Replica Deployment' issue. The simplest reconfiguration is to increase the number of replicas for the 'front-end' deployment from 1 to 2. This change will ensure that there is at least one additional replica available to handle traffic if one pod fails, thus maintaining the steady state of having at least 1 ready replica 100% of the time.

Modification\_type: replace

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

Increase the number of replicas from 1 to 2 to improve resiliency and ensure availability 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

```

## Deploying reconfigured resources... Done

```

$ scaffold run --kube-context kind-chaos-eater-cluster -l project=chaos-eater
No tags generated
Starting test...
Starting deploy...
Loading images into kind cluster nodes...
Images loaded in 420ns
- 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: creating container carts
  - sock-shop:pod/carts-7cfb5db745-648jk: creating container carts
- sock-shop:deployment/carts-db: creating container carts-db
  - sock-s...-db: waiting for rollout to finish: 1 out of 2 new replicas l
- sock-shop:deployment/carts is ready. [10/14 deployment(s) still pending]
- sock-shop:deployment/catalogue-db is ready. [9/14 deployment(s) still pe
- sock-shop:deployment/queue-master is ready. [8/14 deployment(s) still pe
- sock-shop:deployment/session-db is ready. [7/14 deployment(s) still pend
- sock-shop:deployment/user-db is ready. [6/14 deployment(s) still pending]
- sock-shop:deployment/shipping is ready. [5/14 deployment(s) still pending]
- sock-shop:deployment/carts-db is ready. [4/14 deployment(s) still pending]
- sock-shop:deployment/front-end: waiting for rollout to finish: 1 old rep
- sock-shop:deployment/front-end is ready. [3/14 deployment(s) still pendin
- sock-shop:deployment/catalogue: creating container catalogue
  - sock-shop:pod/catalogue-7c75fd67f4-94jt7: creating container catalogu
- sock-shop:deployment/payment: creating container payment
  - sock-shop:pod/payment-65c6784b49-f99lz: creating container payment
- sock-shop:deployment/user: waiting for rollout to finish: 1 old replicas
- sock-shop:deployment/payment is ready. [2/14 deployment(s) still pending]
- sock-shop:deployment/catalogue is ready. [1/14 deployment(s) still pendin
- sock-shop:deployment/user is ready.

Deployments stabilized in 6 minutes 7.825 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-7cfb5db745-648jk	1/1	Running	0
sock-shop	pod/carts-7cfb5db745-k4xlp	1/1	Running	0
sock-shop	pod/carts-db-dd7df56b7-lccqd	1/1	Running	0
sock-shop	pod/carts-db-dd7df56b7-w8f82	1/1	Running	0
sock-shop	pod/catalogue-7c75fd67f4-94jt7	1/1	Running	0
sock-shop	pod/catalogue-7c75fd67f4-vfxnt	1/1	Running	0
sock-shop	pod/catalogue-db-6645646675-dm6qk	1/1	Running	0
sock-shop	pod/catalogue-db-6645646675-wg8kq	1/1	Running	0
sock-shop	pod/front-end-69548cf559-ks7nc	1/1	Running	0
sock-shop	pod/front-end-69548cf559-tsk9m	1/1	Running	0
sock-shop	pod/orders-6645544599-bv5j2	1/1	Running	0
sock-shop	pod/orders-6645544599-d68mr	1/1	Running	0
sock-shop	pod/orders-db-7897884687-m8hdn	1/1	Running	0
sock-shop	pod/orders-db-7897884687-rwfkx	1/1	Running	0
sock-shop	pod/payment-65c6784b49-bz7m2	1/1	Running	0
sock-shop	pod/payment-65c6784b49-f99lz	1/1	Running	0
sock-shop	pod/queue-master-7ff77db77d-9m19s	1/1	Running	0
sock-shop	replicaset.apps/orders-6645544599	2	2	2
sock-shop	replicaset.apps/orders-db-56776875d7	0	0	0
sock-shop	replicaset.apps/orders-db-7897884687	2	2	2
sock-shop	replicaset.apps/payment-5c5549fd9c	0	0	0
sock-shop	replicaset.apps/payment-65c6784b49	2	2	2
sock-shop	replicaset.apps/queue-master-57bf6dd7c8	0	0	0
sock-shop	replicaset.apps/queue-master-7ff77db77d	2	2	2
sock-shop	replicaset.apps/rabbitmq-58d6f88d5	2	2	2
sock-shop	replicaset.apps/rabbitmq-59d99c7945	0	0	0
sock-shop	replicaset.apps/session-db-769cd76c85	2	2	2
sock-shop	replicaset.apps/session-db-c7b86c77	0	0	0
sock-shop	replicaset.apps/shipping-8574c9fb5f	2	2	2
sock-shop	replicaset.apps/shipping-866c845f5f	0	0	0
sock-shop	replicaset.apps/user-67d8df8487	2	2	2
sock-shop	replicaset.apps/user-d9bbf4567	0	0	0
sock-shop	replicaset.apps/user-db-5bfbfd5fc7	0	0	0
sock-shop	replicaset.apps/user-db-767d6bd885	2	2	2

## The fault-scope adjustment Completed!

Current fault injection settings:

- Injected Faults #0: **StressChaos**
  - Workflow Name: **fault-stresschaos**
  - Grace Period: **0s**
  - Duration: **10s**

```
Parameters {"mode": "all", "selector": {"namespaces": ["sock-shop"], "labelSelectors": {"name": "carts-db"}}, "stressors": {"cpu": {"workers": 2, "load": 80}}, "containerNames": ["carts-db"]}
```

The current Kubernetes manifests show that the 'carts-db' deployment remains unchanged in terms of its namespace and labels. The 'carts-db' deployment is still in the 'sock-shop' namespace and has the label 'name: carts-db'. Therefore, the scope of the fault injection, which targets the 'carts-db' pods using the label selector 'name: carts-db' in the 'sock-shop' namespace, does not need to be adjusted. The original goal of the chaos experiment, which is to test the system's resilience under stress conditions by applying a 'StressChaos' fault to the 'carts-db' pods, remains valid with 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: 10s
  - Duration: 5s

```
Parameters {"action": "pod-kill", "mode": "one", "selector": {"namespaces": ["sock-shop"], "labelSelectors": {"name": "front-end"}}, "value": "1"}
```

The current Kubernetes manifests show that the 'front-end' deployment has increased its replicas from 1 to 2. This change does not affect the label selectors or namespace, which remain the same. Therefore, the scope of the fault injection, which targets the 'front-end' pod using the label 'name: front-end' in the 'sock-shop' namespace, remains valid. The fault injection will still target one of the 'front-end' pods, as specified by the 'mode: one' parameter.

Next fault injection scope:

```
{
  "namespaces" : [
    0 : "sock-shop"
  ]
}
```

```
]
  ▼ "labelSelectors" : {
    |   "name" : "front-end"
    |
  }
}
```

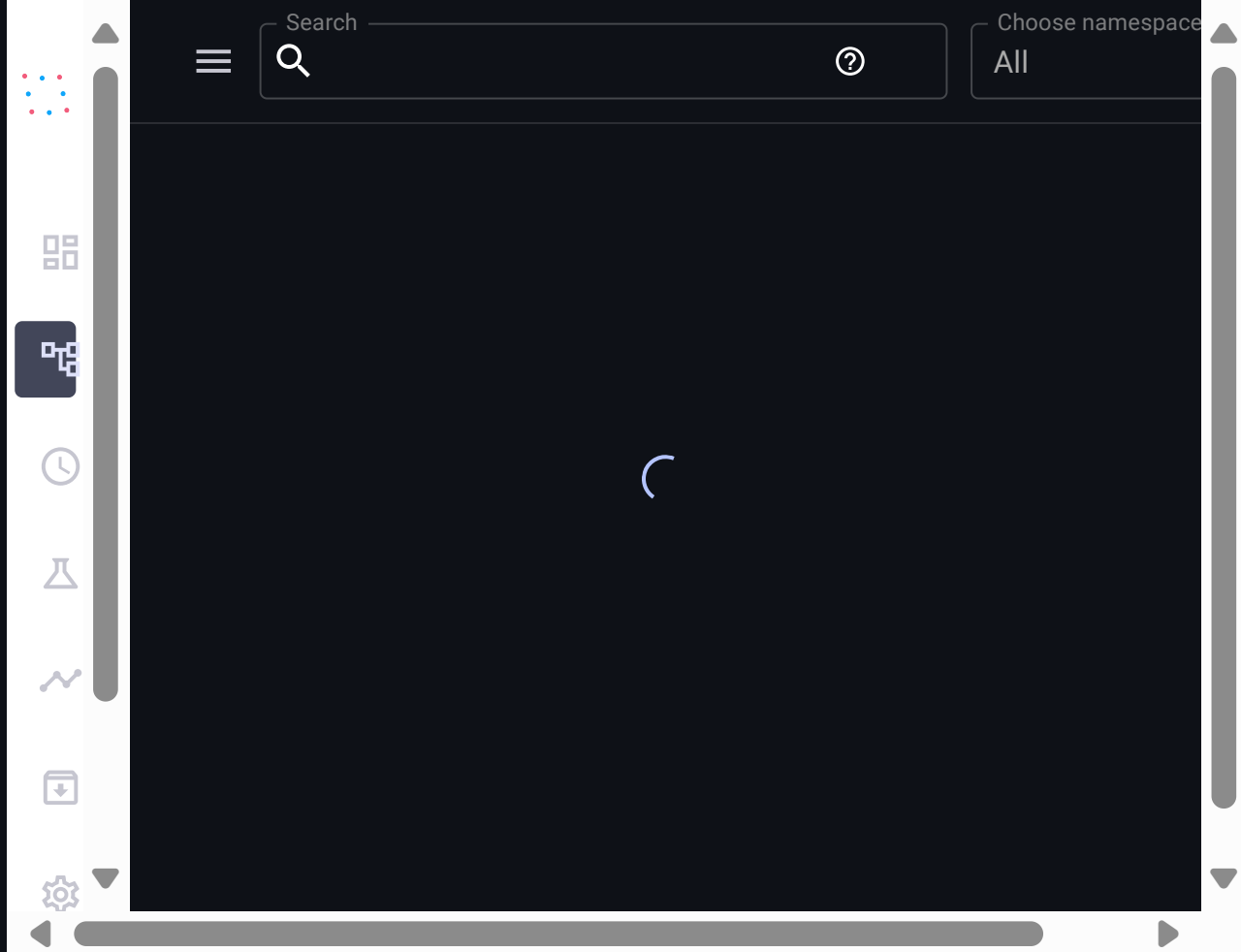
#### Adjusted unittest

The unit test is specifically targeting the `carts-db` deployment, which is defined in the `03-carts-db-dep.yaml` manifest. Upon comparing the previous and reconfigured manifests for `carts-db`, there are no changes in the configuration. The number of replicas remains the same, and there are no modifications to the deployment's specifications that would affect the unit test. Therefore, the unit test does not require any adjustments as the reconfigured manifests do not introduce any changes that would impact the test's logic or its threshold conditions.

#### Adjusted unittest

The unit test is targeting the 'front-end' deployment in the 'sock-shop' namespace. In the reconfigured manifests, the 'front-end' deployment has been modified to increase the number of replicas from 1 to 2. However, the unit test is designed to verify that there is at least 1 ready replica at all times, which is still valid even with the increased number of replicas. The test checks for a minimum of 1 ready replica, and the reconfiguration was likely done to ensure that this condition is met more reliably. Therefore, the unit test does not require any modification, as the threshold of having at least 1 ready replica remains unchanged and is still applicable.

### Completed the chaos experiment!



Your k8s yaml already has good resilience!!!

## Phase EX: Postprocessing

### Summary of your k8s yaml

The Chaos Engineering cycle for the 'sock-shop' system involved a detailed analysis and testing of the system's resilience under simulated fault conditions. The system is composed of multiple Kubernetes manifests that define various microservices, such as 'carts', 'catalogue', 'orders', 'payment', and 'user', each with associated deployments and services. The primary focus of the experiment was to address identified resiliency issues, including missing resource requests, single replica deployments, missing liveness and readiness probes, and hardcoded environment variables.

The experiment was structured into three phases: pre-validation, fault-injection, and post-validation, each lasting 20 seconds, to fit within a total duration of 60 seconds. The pre-validation phase ensured that the system was in a steady state before any faults were injected. The fault-injection phase simulated a Black Friday sale scenario by introducing 'StressChaos' and 'PodChaos' faults to test the system's behavior under stress and potential failure conditions. The post-validation phase verified that the system returned to its steady state after the faults were removed.

The initial experiment revealed that while the 'carts-db' deployment maintained its steady state throughout, the 'front-end' deployment failed during the fault-injection and post-validation phases due to having only a single replica, which made it a single point of failure. This led to a recommendation to increase the number of replicas for the 'front-end' deployment to improve resilience.

After modifying the 'front-end' deployment to have two replicas, a second experiment was conducted. This time, all unit tests passed, indicating that the system successfully maintained its steady states during and after the fault-injection phase. The improvements ensured that the system could handle the simulated high-traffic event and recover from faults, demonstrating enhanced resilience and availability.

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