



## Your instructions for Chaos Engineering:

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



## Phase 0: Preprocessing

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

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

Input instructions for your Chaos Engineering



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

```
$ kubectl delete all --all-namespaces --context kind-chaos-eater-cluster -l  
pod "carts-cd68b5d8b-r4f6k" deleted  
pod "carts-cd68b5d8b-sgsq6" deleted  
pod "carts-db-77454679c8-hh4bs" deleted  
pod "carts-db-77454679c8-t8kns" deleted  
pod "catalogue-69bd9c99d9-ljd6f" deleted  
pod "catalogue-69bd9c99d9-n86d6" deleted  
pod "catalogue-db-6464798844-8d28v" deleted  
pod "catalogue-db-6464798844-9xh5d" deleted  
pod "front-end-5b8c9495cb-h4j2p" deleted  
pod "orders-7f4d894958-hcqzp" deleted  
pod "orders-7f4d894958-m4wn2" deleted  
pod "orders-db-5486cc55fb-754rz" deleted  
pod "orders-db-5486cc55fb-7nfqh" deleted  
pod "payment-7748fd6778-sphzp" deleted  
pod "payment-7748fd6778-vxl25" deleted  
pod "queue-master-6fb6576f9f-cxngf" deleted  
pod "queue-master-6fb6576f9f-s4dzd" deleted  
pod "rabbitmq-bc7f76c4d-4wnlb" deleted  
pod "rabbitmq-bc7f76c4d-sc74r" deleted  
pod "session-db-764c6dbdf4-c2bdd" deleted  
pod "session-db-764c6dbdf4-s7bdd" deleted  
pod "shipping-76fbccf45d-k6gct" deleted  
pod "shipping-76fbccf45d-rjzvb" deleted  
pod "user-7f7c77c49d-6wj4b" deleted  
pod "user-7f7c77c49d-ql4zw" deleted  
pod "user-db-7547b87478-89hzd" deleted  
pod "user-db-7547b87478-prgbt" 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 "catalogue-69bd9c99d9" deleted
replicaset.apps "catalogue-db-6464798844" deleted
replicaset.apps "front-end-5b8c9495cb" deleted
replicaset.apps "orders-7f4d894958" deleted
replicaset.apps "orders-db-5486cc55fb" deleted
replicaset.apps "payment-7748fd6778" deleted
replicaset.apps "queue-master-6fb6576f9f" deleted
replicaset.apps "rabbitmq-bc7f76c4d" deleted
replicaset.apps "session-db-764c6dbdf4" deleted
replicaset.apps "shipping-76fbccf45d" deleted
replicaset.apps "user-7f7c77c49d" deleted
replicaset.apps "user-db-7547b87478" 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

```

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

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: carts-db
  labels:
    name: carts-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: carts-db
  template:
    metadata:
      labels:
        name: carts-db
    spec:
      containers:
        - name: carts-db
          image: mongo
          ports:
            - name: mongo
              containerPort: 27017
          securityContext:
            capabilities:
              drop:
                - all
```

```

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

```

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

```

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

```

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

```

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

```

```

    name: catalogue
template:
  metadata:
    labels:
      name: catalogue
  spec:
    containers:
      - name: catalogue
        image: weaveworksdemos/catalogue:0.3.5
        command: ["/app"]
        args:
          - -port=80
        resources:
          limits:
            cpu: 200m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
          - containerPort: 80
        securityContext:
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
              - all
            add:
              - NET_BIND_SERVICE
          readOnlyRootFilesystem: true
        livenessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 300
          periodSeconds: 3
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 180
          periodSeconds: 3
    nodeSelector:
      beta.kubernetes.io/os: linux

```

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

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalogue-db
  labels:
    name: catalogue-db
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: catalogue-db
  template:
    metadata:
      labels:
        name: catalogue-db
    spec:
      containers:
        - name: catalogue-db
          image: weaveworksdemos/catalogue-db:0.3.0
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: fake_password
            - name: MYSQL_DATABASE
              value: socksdb
          ports:
            - name: mysql
```

```
    containerPort: 3306
  nodeSelector:
    beta.kubernetes.io/os: linux
```

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

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

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end
  namespace: sock-shop
spec:
  replicas: 1
  selector:
    matchLabels:
      name: front-end
  template:
    metadata:
      labels:
        name: front-end
    spec:
      containers:
        - name: front-end
          image: weaveworksdemos/front-end:0.3.12
          resources:
            limits:
              cpu: 300m
              memory: 1000Mi
```

```

      requests:
        cpu: 100m
        memory: 300Mi
    ports:
      - containerPort: 8079
    env:
      - name: SESSION_REDIS
        value: "true"
    securityContext:
      runAsNonRoot: true
      runAsUser: 10001
      capabilities:
        drop:
          - all
      readOnlyRootFilesystem: true
    livenessProbe:
      httpGet:
        path: /
        port: 8079
      initialDelaySeconds: 300
      periodSeconds: 3
    readinessProbe:
      httpGet:
        path: /
        port: 8079
      initialDelaySeconds: 30
      periodSeconds: 3
    nodeSelector:
      beta.kubernetes.io/os: linux

```

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

```

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

```

```
nodePort: 30001
selector:
  name: front-end
```

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders
  labels:
    name: orders
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: orders
  template:
    metadata:
      labels:
        name: orders
    spec:
      containers:
        - name: orders
          image: weaveworksdemos/orders:0.4.7
          env:
            - name: JAVA_OPTS
              value: -Xms64m -Xmx128m -XX:+UseG1GC -Djava.security.egd=file:/dev/./urandom
          resources:
            limits:
              cpu: 500m
              memory: 500Mi
            requests:
              cpu: 100m
              memory: 300Mi
          ports:
            - containerPort: 80
      securityContext:
        runAsNonRoot: true
        runAsUser: 10001
        capabilities:
          drop:
            - all
          add:
            - NET_BIND_SERVICE
```

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

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

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

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

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

```

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

```

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

```

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

```

```
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 972ns
- 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-db is ready. [13/14 deployment(s) still pending]
- sock-shop:deployment/carts: creating container carts
  - sock-shop:pod/carts-7d6dfb46f-nkpfr: creating container carts
  - sock-shop:pod/carts-7d6dfb46f-stq2w: creating container carts
- sock-shop... deployment(s) still pending]
- sock-shop:deployment/shipping is ready. [9/14 deployment(s) still pending]
- sock-shop:deployment/orders-db: creating container orders-db
  - sock-shop:pod/orders-db-57bb99b8c5-9nhns: creating container orders-db
  - sock-shop:pod/orders-db-57bb99b8c5-9tgjn: creating container orders-db
- sock-shop:deployment/user: waiting for rollout to finish: 0 of 2 updated
- sock-shop:deployment/user-db: creating container user-db
  - sock-shop:pod/user-db-c9ff4ddbf-r2nbq: creating container user-db
  - sock-shop:pod/user-db-c9ff4ddbf-tqp7t: creating container user-db
- sock-shop:deployment/carts is ready. [8/14 deployment(s) still pending]
- sock-shop:deployment/rabbitmq is ready. [7/14 deployment(s) still pending]
- sock-shop:deployment/orders is ready. [6/14 deployment(s) still pending]
- sock-shop:deployment/user-db is ready. [5/14 deployment(s) still pending]
- sock-shop:deployment/orders-db is ready. [4/14 deployment(s) still pending]
- sock-shop:deployment/front-end is ready. [3/14 deployment(s) still pending]
- sock-shop:deployment/catalogue is ready. [2/14 deployment(s) still pending]
- sock-shop:deployment/payment is ready. [1/14 deployment(s) still pending]
- sock-shop:deployment/user: waiting for rollout to finish: 1 of 2 updated
- sock-shop:deployment/user is ready.

Deployments stabilized in 3 minutes 5.223 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-7d6dfb46f-nkpfr	1/1	Running	0
sock-shop	pod/carts-7d6dfb46f-stq2w	1/1	Running	0
sock-shop	pod/carts-db-84d97dbfdc-4jxk4	1/1	Running	0
sock-shop	pod/carts-db-84d97dbfdc-rrbcw	1/1	Running	0
sock-shop	pod/catalogue-7f479575d6-hpsqx	1/1	Running	0
sock-shop	pod/catalogue-7f479575d6-k5f2p	1/1	Running	0

sock-shop	pod/catalogue-db-6f5647b5dc-c7lnn	1/1	Running	0	
sock-shop	pod/catalogue-db-6f5647b5dc-j8r54	1/1	Running	0	
sock-shop	pod/front-end-55f6798567-w7mrj	1/1	Running	0	
sock-shop	pod/orders-58c4cccf9-fjzwr	1/1	Running	0	
sock-shop	pod/orders-58c4cccf9-prh97	1/1	Running	0	
sock-shop	pod/orders-db-57bb99b8c5-9nhns	1/1	Running	0	
sock-shop	pod/orders-db-57bb99b8c5-9tgjn	1/1	Running	0	
sock-shop	pod/payment-567b57dbb9-b87f4	1/1	Running	0	
sock-shop	pod/payment-567b57dbb9-v792x	1/1	Running	0	
sock-shop	pod/queue-master-5495cfbd7d-2xmv1	1/1	Running	0	
sock-shop	pod/queue-master-5495cfbd7d-k9h54	1/1	Running	0	
sock-shop	pod/rabbitmq...	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT
sock-shop	service/carts	ClusterIP	10.96.115.234	<none>	
sock-shop	service/carts-db	ClusterIP	10.96.28.82	<none>	
sock-shop	service/catalogue	ClusterIP	10.96.205.72	<none>	
sock-shop	service/catalogue-db	ClusterIP	10.96.139.115	<none>	
sock-shop	service/front-end	NodePort	10.96.119.33	<none>	
sock-shop	service/orders	ClusterIP	10.96.91.52	<none>	
sock-shop	service/orders-db	ClusterIP	10.96.100.252	<none>	
sock-shop	service/payment	ClusterIP	10.96.37.138	<none>	
sock-shop	service/queue-master	ClusterIP	10.96.151.51	<none>	
sock-shop	service/rabbitmq	ClusterIP	10.96.32.236	<none>	
sock-shop	service/session-db	ClusterIP	10.96.181.166	<none>	
sock-shop	service/shipping	ClusterIP	10.96.173.255	<none>	
sock-shop	service/user	ClusterIP	10.96.140.169	<none>	
sock-shop	service/user-db	ClusterIP	10.96.213.157	<none>	

## Summary of each manifest:

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

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

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

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'carts' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'carts' application running.
- The Deployment uses the Docker image 'weaveworksdemos/carts:0.4.8'.
- Environment variables are set for Java options to optimize memory usage and disable certain features.
- Resource limits and requests are defined, with a maximum of 300m CPU and 500Mi memory, and a minimum of 100m CPU and 200Mi memory.

- The application listens on port 80 inside the container.
- Security context is configured to run the container as a non-root user with user ID 10001.
- The container has limited capabilities, only allowing NET\_BIND\_SERVICE, and uses 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 for 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 context is set to drop all capabilities and only add CHOWN, SETGID, and SETUID, with a read-only root filesystem.
- A volume is mounted at '/tmp' using an emptyDir volume, which is stored in memory.
- 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 for Prometheus scraping with 'prometheus.io/scrape: true'.
- The Service is labeled with 'name: catalogue'.
- It is deployed in the 'sock-shop' namespace.
- The Service exposes port 80 and directs traffic to the same port on the selected pods.
- It 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' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas (instances) of the 'catalogue-db' pod running.
- The pods are selected based on the label 'name: catalogue-db'.
- Each pod runs a single container using the image 'weaveworksdemos/catalogue-db:0.3.0'.
- The container is configured with environment variables for 'MYSQL\_ROOT\_PASSWORD' and 'MYSQL\_DATABASE'.
- The container exposes port 3306, which is typically used for MySQL databases.
- The pods are scheduled to run on nodes with the operating system labeled as 'linux'.

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

- This manifest defines a Kubernetes Service.
- The Service is named 'catalogue-db'.

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

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

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'front-end' and is located in the 'sock-shop' namespace.
- It specifies that there should be 1 replica of the front-end application running.
- The Deployment uses a container image 'weaveworksdemos/front-end:0.3.12'.
- Resource limits are set for the container: 300 milliCPU and 1000 MiB of memory.
- Resource requests are set for the container: 100 milliCPU and 300 MiB of 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 additional 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 Deployment is scheduled to run 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 part of the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'orders' application running.

- The Deployment uses the Docker image 'weaveworksdemos/orders:0.4.7'.
- Environment variables are set for Java options to optimize memory usage and disable certain features.
- Resource limits and requests are defined, with a maximum of 500m CPU and 500Mi memory, and a minimum of 100m CPU and 300Mi memory.
- The application listens on port 80 inside the container.
- Security 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/12-orders-svc.yaml`

- This manifest defines a Kubernetes Service.
- The Service is named 'orders'.
- It is annotated for Prometheus scraping, which means it is set up to be monitored by Prometheus.
- The Service is labeled with 'name: orders' for identification and organization.
- It is deployed in the 'sock-shop' namespace, which is a way to group resources in Kubernetes.
- The Service exposes port 80 and directs traffic to the same port on the selected pods.
- The Service uses a selector to target pods with the label 'name: 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 context is set to drop all capabilities and add only CHOWN, SETGID, and SETUID for enhanced security.
- The root filesystem of the container is set to read-only to prevent unauthorized changes.
- 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 find pods with the label 'name: orders-db'.

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

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'payment' and is located in the 'sock-shop' namespace.
- It specifies that there should be 2 replicas of the 'payment' application running.
- The Deployment uses the Docker image 'weaveworksdemos/payment:0.4.3'.
- Resource limits are set for the container: 200m CPU and 200Mi memory.
- Resource requests are set for the container: 99m CPU and 100Mi memory.
- The container listens on port 80.
- Security context is configured to run the container as a non-root user with user ID 10001.
- All capabilities are dropped except 'NET\_BIND\_SERVICE', and the root filesystem is set to read-only.
- A liveness probe is configured to check the '/health' endpoint on port 80, starting after 300 seconds and checking every 3 seconds.
- A readiness probe is also configured to check the '/health' endpoint on port 80, starting after 180 seconds and checking every 3 seconds.
- The Deployment is scheduled to run on nodes with the operating system labeled as Linux.

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

- This is a Kubernetes Service manifest.
- The service is named 'payment'.
- It is annotated for Prometheus scraping with 'prometheus.io/scrape: true'.
- 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.
- It selects pods with the label 'name: payment'.

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

- This manifest defines a Kubernetes Deployment.
- The Deployment is named 'queue-master'.
- It is located in the 'sock-shop' namespace.
- The Deployment will create 2 replicas of the application.
- It uses a selector to match pods with the label 'name: queue-master'.

- The pod template specifies a single container named 'queue-master'.
- The container uses the image 'weaveworksdemos/queue-master:0.3.1'.
- Environment variables are set for Java options to optimize performance.
- Resource limits are set to 300m CPU and 500Mi memory.
- Resource requests are set to 100m CPU and 300Mi memory.
- The container exposes port 80.
- The Deployment is configured to run on Linux nodes using a node selector.

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

- This manifest defines a Kubernetes Service.
- The Service is named 'queue-master'.
- It is annotated to enable 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 uses a selector to target pods with the label 'name: queue-master'.

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

- This manifest defines a Deployment in Kubernetes.
- The Deployment is named 'rabbitmq' and is located in the 'sock-shop' namespace.
- It specifies 2 replicas of the RabbitMQ application.
- The Deployment uses a selector to match pods with the label 'name: rabbitmq'.
- The pod template includes two containers: one for RabbitMQ and another for RabbitMQ exporter.
- The RabbitMQ container uses the image 'rabbitmq:3.6.8-management' and exposes ports 15672 (management) and 5672 (RabbitMQ service).
- The RabbitMQ container has a security context that drops all capabilities and adds specific ones like CHOWN, SETGID, SETUID, and DAC\_OVERRIDE, and it uses a read-only root filesystem.
- The RabbitMQ exporter container uses the image 'kbudde/rabbitmq-exporter' and exposes port 9090.
- The Deployment is configured to run on nodes with the operating system labeled as Linux.

`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 2 replicas of the pod to be created.
- 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.
- The root filesystem of the container is set to be read-only for security purposes.
- The pods are scheduled to run on nodes with the operating system labeled as 'linux'.
- An annotation is set to prevent Prometheus from scraping metrics from this pod.

`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 2 replicas of the 'shipping' application to be run.
- 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 CPU limits at 300m and memory limits at 500Mi.
- The application listens on port 80 within the container.

- Security context is configured to run the container as a non-root user with user ID 10001.
- The container has a read-only root filesystem and specific capabilities are dropped and added.
- A temporary volume is mounted at '/tmp' using an in-memory empty directory.
- The Deployment is scheduled to run on nodes with the Linux operating system.

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

- This manifest defines a Kubernetes Service.
- 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 listens on port 80 and forwards traffic to the same port on the target 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 to enable Prometheus scraping with 'prometheus.io/scrape: true'.
- The Service is labeled with 'name: user'.

- 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: 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 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: user-db'.

## Resiliency issues/weaknesses in the manifests:

### Issue #0: Missing Resource Requests

- details: Pods may not get scheduled if resources are constrained, 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: No resource requests specified for CPU and memory.

### Issue #1: Single Replica Deployment

- details: Single replica deployments 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 application, leading to potential downtime or traffic being sent 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: No livenessProbe or readinessProbe defined.

## 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.

## Issue #4: Missing Service Port Configuration

- details: Services without defined ports may not function correctly, leading to connectivity issues.
- manifests having the issues: ['sock-shop-2/manifests/02-carts-svc.yaml', 'sock-shop-2/manifests/04-carts-db-svc.yaml', 'sock-shop-2/manifests/06-catalogue-svc.yaml', 'sock-shop-2/manifests/08-catalogue-db-svc.yaml', 'sock-shop-2/manifests/12-orders-svc.yaml', 'sock-shop-2/manifests/14-orders-db-svc.yaml', 'sock-shop-2/manifests/16-payment-svc.yaml', 'sock-shop-2/manifests/18-queue-master-svc.yaml', 'sock-shop-2/manifests/22-session-db-svc.yaml', 'sock-shop-2/manifests/24-shipping-svc.yaml', 'sock-shop-2/manifests/26-user-svc.yaml', 'sock-shop-2/manifests/28-user-db-svc.yaml']
- problematic config: Service ports are defined but commented out, leading to potential misconfiguration.

## 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. The application simulates an e-commerce website for selling socks, and the

manifests include various components such as front-end, back-end services, databases, and messaging systems. The naming conventions, such as 'carts', 'catalogue', 'orders', 'payment', 'shipping', and 'user', align with typical e-commerce functionalities. Additionally, the use of images from 'weaveworksdemos' further indicates that this is the Sock Shop demo application.

Assumed application:

Sock Shop is a microservices-based e-commerce application that simulates an online store for selling socks. It provides functionalities such as browsing a catalogue, managing a shopping cart, placing orders, processing payments, handling shipping, and managing user accounts.

## 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 specified options.

## Phase 1: Hypothesis

### Steady-state definition

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

##### Description

The first issue to address is the 'Missing Resource Requests' in the 'carts-db' Deployment. This is a critical issue because without resource requests, the pods may not be scheduled if the cluster is under resource constraints, leading to potential downtime. The steady state for this manifest should ensure that the 'carts-db' pods are running and available, which can be measured by checking the number of replicas that are in the 'Running' state.

##### Current state inspection

To inspect the 'carts-db-replicas-running' state, I will use the K8s API. The reason for choosing the K8s API is that it allows us to directly query the Kubernetes cluster to check the status of the 'carts-db' Deployment, specifically the number of replicas that are in the 'Running' state. This is more appropriate than using k6, which is better suited for testing communication and performance metrics. The script will run for a maximum of 5 seconds, checking the status every second, to ensure that the 'carts-db' pods are running and available.

tool: `k8s` duration: `5s`

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

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

# 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()

def check_carts_db_replicas(namespace='sock-shop', deployment_name='carts-db-replicas'):
    try:
        deployment = v1.read_namespaced_deployment(name=deployment_name, namespace=namespace)
        replicas = deployment.status.replicas
        ready_replicas = deployment.status.ready_replicas
        print(f"Total replicas: {replicas}, Ready replicas: {ready_replicas}")
        return ready_replicas == replicas
    except client.exceptions.ApiException as e:
        print(f"Exception when calling AppsV1Api->read_namespaced_deployment_status: {e}")
        return False

def main(duration):
    success_count = 0
    for _ in range(duration):
        if check_carts_db_replicas():
            success_count += 1
        time.sleep(1)
    print(f"Carts-db replicas running successfully for {success_count} times")

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

Pod `k8scarts-db-replicas-running-pod` has completed sucessfully.

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
Carts-db replicas running successfully for 5/5 seconds.
```

## 🚩 Threshold

The steady state we are considering is the number of 'carts-db' replicas that are running and ready. The current state shows that there are 2 replicas, and both are ready, which is the expected state. Given that the deployment specifies 2 replicas, the threshold should ensure that both replicas are running and ready. To account for some fluctuations and ensure the system remains stable, we can define a threshold that allows for a small percentage of time where not all replicas are ready, as long as the system recovers quickly. Since the experiment duration is 1 minute, we can set a threshold that requires the 'carts-db' replicas to be ready for at least 90% of the time during this period. This means that out of 60 seconds, the replicas should be ready for at least 54 seconds. This threshold provides a reasonable tolerance for transient issues while ensuring the system remains stable.

The 'carts-db' replicas must be ready for at least 90% of the time during a 1-minute monitoring period, which equates to at least 54 out of 60 seconds.

## 📄 Unit test to validate the steady state

To add the threshold assertion to the inspection Python script, I will implement a check to ensure that the 'carts-db' replicas are ready for at least 90% of the specified duration. This will be done by counting the number of seconds the replicas are ready and comparing it to 90% of the total duration. If the ready count meets or exceeds this threshold, the test will pass; otherwise, it will fail. The threshold is implemented as a percentage (90%) of the total duration, allowing for variable durations.

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

class TestCartsDBReplicas(K8sAPIBase):
    def __init__(self):
```

```

super().__init__()
self.v1_apps = client.AppsV1Api()

def check_carts_db_replicas(self, namespace='sock-shop', deployment='carts-db-replicas'):
    try:
        deployment = self.v1_apps.read_namespaced_deployment(namespace, deployment)
        replicas = deployment.status.replicas
        ready_replicas = deployment.status.ready_replicas
        print(f"Total replicas: {replicas}, Ready replicas: {ready_replicas}")
        return ready_replicas == replicas
    except client.exceptions.ApiException as e:
        print(f"Exception when calling AppsV1Api->read_namespaced_deployment_status: {e}")
        return False

def test_replicas_ready_threshold(self, duration):
    success_count = 0
    for _ in range(duration):
        if self.check_carts_db_replicas():
            success_count += 1
        time.sleep(1)
    # Calculate the threshold as 90% of the duration
    threshold = 0.9 * duration
    print(f"Carts-db replicas running successfully for {success_count} out of {duration}")
    # Assert that the success count meets or exceeds the threshold
    assert success_count >= threshold, f"Replicas were not ready for {duration} seconds"

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

    test = TestCartsDBReplicas()
    test.test_replicas_ready_threshold(args.duration)

if __name__ == '__main__':
    main()

```

Pod `unittestcarts-db-replicas-runningmod0-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
Carts-db replicas running successfully for 5/5 seconds.
```

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

### 🗨 Description

The next issue to address is the 'Single Replica Deployment' in the 'front-end' Deployment. This is a critical issue because having only a single replica can lead to downtime if the pod fails. The steady state for this manifest should ensure that the 'front-end' pod is running and available, which can be measured by checking the number of replicas that are in the 'Running' state. Since there is only one replica, the steady state should ensure that this single replica is running and ready for the entire duration of the monitoring period.

### 🔍 Current state inspection

To inspect the 'front-end-replica-running' state, I will use the K8s API. The goal is to ensure that the single replica of the 'front-end' deployment is running and ready. The K8s API is suitable for checking the status of Kubernetes resources, such as deployments and pods. I will write a Python script using the K8s client library to check the status of the 'front-end' pod in the 'sock-shop' namespace. The script will run for a duration of 5 seconds, checking the pod status every second, and will print a summary of the results at the end.

tool: `k8s` duration: `5s`

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

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

# Load Kubernetes configuration based on the environment
if os.getenv('KUBERNETES_SERVICE_HOST'):
    config.load_incluster_config()
else:
    config.load_kube_config()

v1 = client.CoreV1Api()
```

```

namespace = 'sock-shop'
deployment_name = 'front-end'

# Function to check the status of the front-end pod
def check_front_end_status():
    pods = v1.list_namespaced_pod(namespace=namespace, label_selector=deployment_name)
    running_pods = [pod for pod in pods if pod.status.phase == 'Running']
    ready_pods = [pod for pod in running_pods if all(container.ready for container in pod.status.container_statuses)]
    return len(ready_pods)

def main(duration):
    for _ in range(duration):
        running_and_ready = check_front_end_status()
        print(f'Running and ready front-end pods: {running_and_ready}')
        time.sleep(1)
    print('Status check completed.')

if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description='Check front-end pod status')
    parser.add_argument('--duration', type=int, default=5, help='Duration of the check in seconds')
    args = parser.parse_args()
    main(args.duration)

```

Pod `k8sfront-end-replica-running-pod` has completed successfully.

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

```

Running and ready front-end pods: 1
Running and ready front-end pods: 1
Running and ready front-end pods: 1
Running and ready front-end pods: 1
Running and ready front-end pods: 1
Status check completed.

```

## ► Threshold

The steady state we are considering is the availability of the 'front-end' pod, which currently has a single replica. The current state shows that the single replica is consistently running and ready, as indicated by the repeated output of 'Running and ready front-end pods: 1'. Given that the system is expected to maintain this state, the threshold should ensure that the single replica remains running and ready for the entire duration of the monitoring period. Since the monitoring period in the script is 5 seconds, and the Chaos Engineering experiment must be completed within 1 minute,

we can set a threshold that requires the front-end pod to be running and ready for at least 95% of the time during a 1-minute period. This accounts for any minor fluctuations or delays in pod readiness checks.

The 'front-end' pod must have at least 1 running and ready replica for 95% of the time within a 1-minute monitoring period.

## Unit test to validate the steady state

To add the threshold assertion to the inspection Python script, I will implement a unit test that checks if the 'front-end' pod is running and ready for at least 95% of the specified duration. This will involve counting the number of successful checks and comparing it to the total number of checks to ensure the threshold is met. The test will use a for-loop to repeatedly check the pod status every second for the specified duration, and at the end, it will assert that the percentage of successful checks meets or exceeds the 95% threshold.

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

class TestFrontEndReplicaRunning(K8sAPIBase):
    def __init__(self):
        super().__init__()
        self.namespace = 'sock-shop'
        self.deployment_name = 'front-end'

    def check_front_end_status(self):
        # List all pods in the specified namespace with the label 'r
        pods = self.v1.list_namespaced_pod(namespace=self.namespace,
        # Filter pods that are in the 'Running' state
        running_pods = [pod for pod in pods if pod.status.phase == 'R
        # Further filter pods that are ready
        ready_pods = [pod for pod in running_pods if all(container.r
        return len(ready_pods)

    def test_steady_state(self, duration):
        successful_checks = 0
        total_checks = duration

        for _ in range(duration):
            running_and_ready = self.check_front_end_status()
            print(f'Running and ready front-end pods: {running_and_r
```

```

        if running_and_ready >= 1:
            successful_checks += 1
            time.sleep(1)

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

    # Assert that the success rate is at least 95%
    assert success_rate >= 95, f'Success rate {success_rate}% is below 95%'

def main():
    parser = argparse.ArgumentParser(description='Test front-end pods')
    parser.add_argument('--duration', type=int, default=60, help='Duration of the test')
    args = parser.parse_args()

    test = TestFrontEndReplicaRunning()
    test.test_steady_state(args.duration)

if __name__ == '__main__':
    main()

```

Pod `unittestfront-end-replica-runningmod0-pod` has completed successfully.

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

```

Running and ready front-end pods: 1
Running and ready front-end pods: 1
Running and ready front-end pods: 1
Running and ready front-end pods: 1
Running and ready front-end pods: 1
Success rate: 100.0%

```

## Fault definition

### ✓ Scenario: Black Friday Sale

#### Description

During a Black Friday sale, the system is expected to handle a significant increase in traffic and load. This event will test the system's ability to maintain steady states under

high demand. The 'carts-db' and 'front-end' components are critical to the user experience, as they handle shopping cart operations and the user interface, respectively. The 'carts-db' deployment lacks resource requests, which could lead to scheduling issues under resource constraints. The 'front-end' deployment has a single replica, making it vulnerable to downtime if the pod fails. To simulate the Black Friday event, we will first inject a StressChaos fault to increase CPU and memory usage on the 'carts-db' pods, testing their ability to remain running without resource requests. Next, we will inject a PodChaos fault to kill the 'front-end' pod, testing the system's ability to recover and maintain availability with a single replica. This sequence of fault injections will reveal potential weaknesses in resource allocation and redundancy, ensuring the system can handle the increased load and maintain steady states.

## 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
    }
    "memory" : {
      "workers" : 2
      "size" : "256MB"
    }
  }
}
```

```

    }
  }
  "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 simulated fault conditions, specifically during a high-demand event like a Black Friday sale. The experiment must be completed within 1 minute, so careful time allocation is crucial. The experiment is divided into three phases: pre-validation, fault-injection, and post-validation.

1. **Pre-validation Phase (20 seconds):** This phase ensures that the system is in a steady state before any faults are injected. We will verify that the 'carts-db' replicas are running and ready for at least 90% of the time, and the 'front-end' replica is running and ready for at least 95% of the time. Given the short duration of the

experiment, a 20-second pre-validation period is sufficient to confirm the system's initial stability.

2. **Fault Injection Phase (20 seconds):** During this phase, we will inject the specified faults using Chaos Mesh. The 'carts-db' will undergo a StressChaos fault to simulate high CPU and memory usage, while the 'front-end' will experience a PodChaos fault to simulate a pod failure. The 20-second duration allows us to observe the system's behavior under stress and its ability to maintain steady states.
3. **Post-validation Phase (20 seconds):** After fault injection, we need to ensure that the system returns to its steady states. This phase will verify that the 'carts-db' replicas and the 'front-end' replica meet their respective thresholds for readiness and availability. A 20-second post-validation period is adequate to confirm recovery and stability.

The total time for the experiment is 60 seconds, with each phase allocated 20 seconds to balance validation and fault observation effectively.

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 its expected steady state before we proceed with fault injection. This involves verifying that the critical components, 'carts-db' and 'front-end', are operating as expected. 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 'carts-db-replicas-running' test will check that the 'carts-db' replicas are running and ready for at least 90% of the time during a 10-second monitoring period. Similarly, the 'front-end-replica-running' test will verify that the 'front-end' pod is running and ready for at least 95% of the time during the same 10-second period. By running these tests concurrently, we ensure that both components are in their expected steady states within the limited time frame, allowing us to proceed confidently to the fault injection phase.

- Verified Steady State #0: carts-db-replicas-running
  - Workflow Name: pre-unittest-carts-db-replicas-running
  - Grace Period: 0s
  - Duration: 10s

- Verified Steady State #1: `front-end-replica-running`
  - Workflow Name: `pre-unittest-front-end-replica-running`
  - Grace Period: `0s`
  - Duration: `10s`

### Fault-injection Phase (20s)

In this fault-injection phase, we aim to simulate a Black Friday sale scenario by injecting two types of faults: StressChaos on the 'carts-db' pods and PodChaos on the 'front-end' pod. The total duration for this phase is 20 seconds, so we need to carefully time the injections to observe their effects on the system's steady states.

First, we will inject the StressChaos fault on the 'carts-db' pods. This fault will increase CPU and memory usage, testing the pods' ability to handle high load without resource requests. We will start this fault at the beginning of the phase (grace period of 0s) and let it run for 10 seconds. This duration is chosen to allow enough time for the stress to impact the system while leaving time for the next fault.

Next, we will inject the PodChaos fault on the 'front-end' pod. This fault will kill the pod, testing the system's ability to recover and maintain availability with a single replica. We will start this fault 10 seconds into the phase (grace period of 10s) and let it run for the remaining 10 seconds. This staggered approach allows us to observe the effects of each fault separately and ensures that the system is under stress for the entire duration of the phase.

By staggering the faults, we can better understand how each component reacts to stress and failure, providing insights into potential weaknesses in resource allocation and redundancy.

- 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: `10s`

### 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. Given the faults injected during the Black Friday Sale simulation, we will focus on verifying the steady states of the 'carts-db' and 'front-end' components. The post-validation phase is 20 seconds long, so we need to efficiently schedule the unit tests to verify the steady states within this time frame.

The approach is to run the unit tests for both steady states simultaneously, as they are independent checks and can be executed in parallel. This allows us to maximize the use of the available time and ensure both components are back to their expected states. The 'carts-db-replicas-running' test will check if the 'carts-db' replicas are running and ready for at least 90% of the time during the 20-second period, while the 'front-end-replica-running' test will verify that the 'front-end' pod is running and ready for at least 95% of the time.

By running these tests concurrently, we can quickly determine if the system has recovered from the faults and returned to its steady states, ensuring that the system is resilient and can handle similar scenarios in the future.

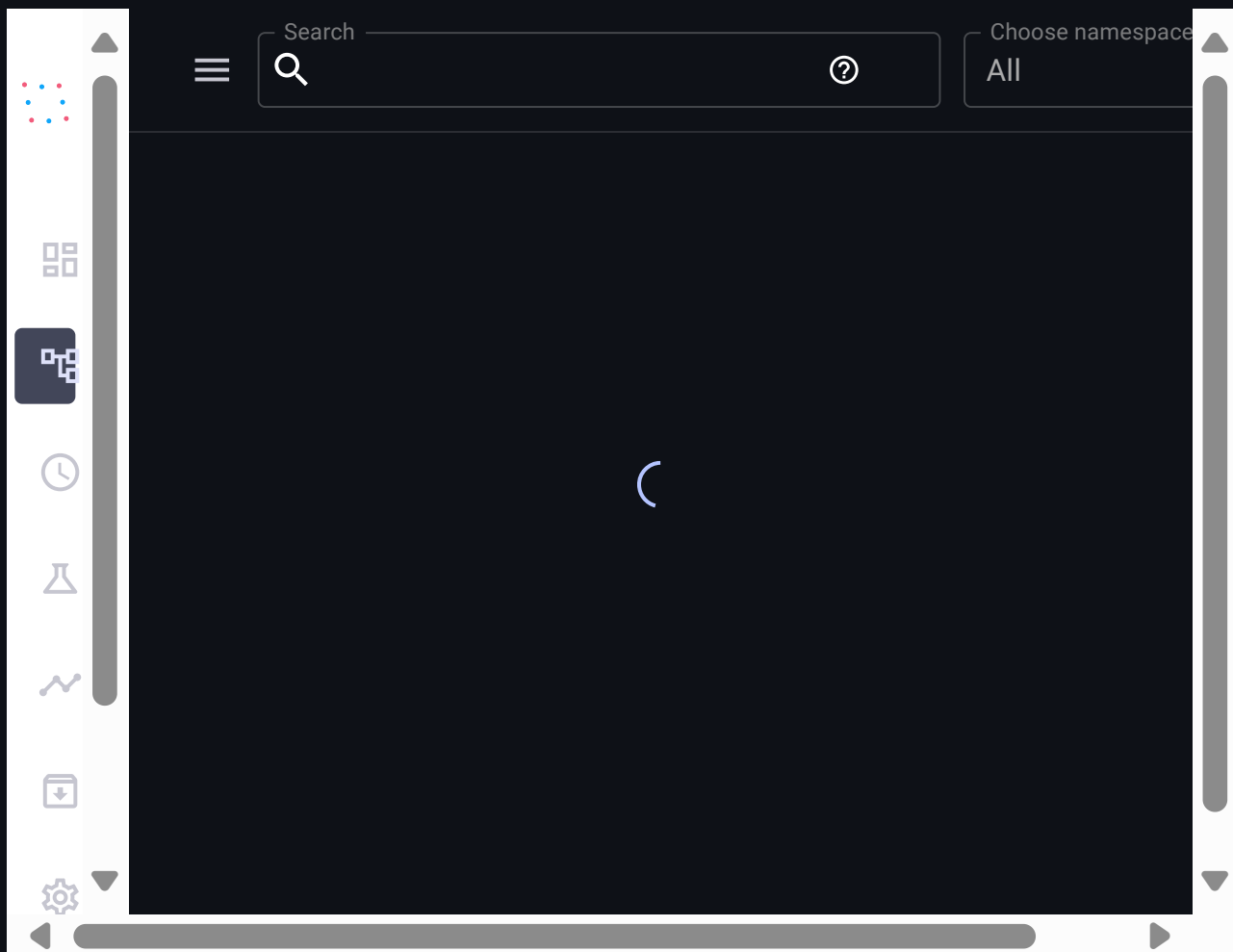
- Verified Steady State #0: `carts-db-replicas-running`
  - Workflow Name: `post-unittest-carts-db-replicas-running`
  - Grace Period: `0s`
  - Duration: `20s`
- Verified Steady State #1: `front-end-replica-running`
  - Workflow Name: `post-unittest-front-end-replica-running`
  - Grace Period: `0s`
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

### Summary

The Chaos Engineering experiment is structured into three phases, each lasting 20 seconds, to test the system's resilience during a simulated high-demand event. In the pre-validation phase, which lasts for 20 seconds, two unit tests are executed simultaneously to ensure the system is in a steady state before fault injection. The 'carts-db-replicas-running' test, with a workflow name of 'pre-unittest-carts-db-replicas-running', and the 'front-end-replica-running' test, with a workflow name of 'pre-unittest-front-end-replica-running', both start immediately at the beginning of the phase and run for 10 seconds. This concurrent execution confirms the readiness of the 'carts-db' replicas and the 'front-end' pod. In the fault-injection phase, also 20 seconds long, two faults are injected in a staggered manner. The 'StressChaos' fault, with a workflow name of 'fault-stresschaos', is injected at the start of the phase and runs for 10 seconds, simulating high CPU and memory usage on the 'carts-db' pods. Following this,

the 'PodChaos' fault, with a workflow name of 'fault-podchaos', is injected 10 seconds into the phase and runs for the remaining 10 seconds, simulating a pod failure on the 'front-end'. This staggered approach allows for separate observation of each fault's impact. Finally, in the post-validation phase, which is again 20 seconds long, the system's recovery is verified by running two unit tests concurrently. The 'carts-db-replicas-running' test, with a workflow name of 'post-unittest-carts-db-replicas-running', and the 'front-end-replica-running' test, with a workflow name of 'post-unittest-front-end-replica-running', both start immediately and run for the full 20 seconds. This ensures that both the 'carts-db' replicas and the 'front-end' pod have returned to their expected steady states, confirming the system's resilience and recovery capabilities.

## Completed the chaos experiment!



## Phase 3: Analysis

The Chaos Engineering experiment aimed to test the resilience of the 'sock-shop' system under simulated high-demand conditions, specifically focusing on the 'carts-db' and 'front-end' components. The experiment was structured into three phases: pre-validation, fault injection, and post-validation. The results indicate that while the 'carts-db' component maintained its

steady state throughout the experiment, the 'front-end' component failed to meet its steady state threshold during the post-validation phase.

### 1. Pre-Validation Phase:

- Both the 'carts-db-replicas-running' and 'front-end-replica-running' tests passed, indicating that the system was in a steady state before the fault injection. This confirms that the initial conditions were correctly set, with the 'carts-db' replicas running and the single 'front-end' pod operational.

### 2. Fault Injection Phase:

- The 'StressChaos' fault was injected first, targeting the 'carts-db' pods with increased CPU and memory usage. Despite the lack of resource requests in the 'carts-db' deployment, the system managed to maintain the steady state for 'carts-db', as evidenced by the successful post-validation test.
- The 'PodChaos' fault was injected next, targeting the 'front-end' pod by simulating a pod failure. This fault exposed a critical vulnerability in the system: the single replica configuration of the 'front-end' deployment.

### 3. Post-Validation Phase:

- The 'carts-db-replicas-running' test passed, confirming that the 'carts-db' component recovered successfully from the stress conditions.
- The 'front-end-replica-running' test failed, with a success rate of only 10%, far below the required 95% threshold. The logs indicate that the 'front-end' pod was not running and ready for the majority of the post-validation period. This failure is directly linked to the single replica configuration of the 'front-end' deployment, which lacks redundancy. When the 'PodChaos' fault killed the 'front-end' pod, there was no additional replica to take over, leading to downtime.

### Insights and Recommendations:

- The failure of the 'front-end' component highlights the critical issue of having a single replica deployment. To improve resilience, it is recommended to increase the number of replicas for the 'front-end' deployment. This will provide redundancy and 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 load conditions.
- Review and enhance the readiness and liveness probes for the 'front-end' deployment to ensure that Kubernetes can effectively manage pod health and recovery.
- While the 'carts-db' component maintained its steady state, it is advisable to define resource requests to prevent potential scheduling issues under resource constraints in future scenarios.

In conclusion, the experiment successfully identified a significant resilience issue in the 'front-end' component, providing valuable insights for improving the system's fault tolerance and availability.

## Phase 4: Improvement

### Reconfiguration

To improve the resiliency of the 'front-end' component and avoid failures, we will increase the number of replicas for the 'front-end' deployment from 1 to 2. This change will provide redundancy, ensuring that the system can maintain availability even if one pod fails. Additionally, we will add resource requests to the 'carts-db' deployment to prevent potential scheduling issues under resource constraints.

Modification\_type: replace

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

Increase the number of replicas from 1 to 2 to provide redundancy and improve availability.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: front-end
  namespace: sock-shop
spec:
  replicas: 2
  selector:
    matchLabels:
      name: front-end
  template:
    metadata:
      labels:
        name: front-end
    spec:
      containers:
        - name: front-end
          image: weaveworksdemos/front-end:0.3.12
          resources:
            limits:
              cpu: 300m
              memory: 1000Mi
            requests:
              cpu: 100m
              memory: 300Mi
```

```

ports:
- containerPort: 8079
env:
- name: SESSION_REDIS
  value: "true"
securityContext:
  runAsNonRoot: true
  runAsUser: 10001
  capabilities:
    drop:
    - all
  readOnlyRootFilesystem: true
livenessProbe:
  httpGet:
    path: /
    port: 8079
  initialDelaySeconds: 300
  periodSeconds: 3
readinessProbe:
  httpGet:
    path: /
    port: 8079
  initialDelaySeconds: 30
  periodSeconds: 3
nodeSelector:
  beta.kubernetes.io/os: linux

```

Modification\_type: replace

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

Add resource requests to ensure the 'carts-db' pods can be scheduled even under resource constraints.

```

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

```

```

metadata:
  labels:
    name: carts-db
spec:
  containers:
    - name: carts-db
      image: mongo
      ports:
        - name: mongo
          containerPort: 27017
      resources:
        requests:
          cpu: 100m
          memory: 200Mi
      securityContext:
        capabilities:
          drop:
            - all
          add:
            - CHOWN
            - SETGID
            - SETUID
        readOnlyRootFilesystem: true
      volumeMounts:
        - mountPath: /tmp
          name: tmp-volume
  volumes:
    - name: tmp-volume
      emptyDir:
        medium: Memory
  nodeSelector:
    beta.kubernetes.io/os: linux

```

## Deploying reconfigured resources... Done

```

$ skaffold run --kube-context kind-chaos-eater-cluster -l project=chaos-eater
No tags generated
Starting test...
Starting deploy...
Loading images into kind cluster nodes...
Images loaded in 391ns
- namespace/sock-shop unchanged
- Warning: spec.template.spec.nodeSelector[beta.kubernetes.io/os]: deprecated
- deployment.apps/carts configured
- service/carts configured

```

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

Waiting for deployments to stabilize...

- sock-shop:deployment/carts: waiting for rollout to finish: 1 out of 2 new replicas are available
- sock-shop:deployment/carts-db: creating container carts-db
  - sock-shop:pod/carts-db-6...ated...
- sock-shop:deployment/catalogue-db is ready. [10/14 deployment(s) still pending]
- sock-shop:deployment/carts is ready. [9/14 deployment(s) still pending]
- sock-shop:deployment/orders is ready. [8/14 deployment(s) still pending]
- sock-shop:deployment/session-db is ready. [7/14 deployment(s) still pending]
- sock-shop:deployment/queue-master 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: waiting for rollout to finish: 1 old replicas are ready for update
- sock-shop:deployment/rabbitmq is ready. [4/14 deployment(s) still pending]
- sock-shop:deployment/front-end: waiting for rollout to finish: 1 old replicas are ready for update
- sock-shop:deployment/front-end is ready. [3/14 deployment(s) still pending]
- sock-shop:deployment/catalogue: waiting for rollout to finish: 1 old replicas are ready for update
- sock-shop:deployment/payment: creating container payment
  - sock-shop:pod/payment-65b4dd6f9b-kzkql: creating container payment
- sock-shop:deployment/user: waiting for rollout to finish: 1 old replicas are ready for update
- sock-shop:deployment/catalogue is ready. [2/14 deployment(s) still pending]
- sock-shop:deployment/user is ready. [1/14 deployment(s) still pending]
- sock-shop:deployment/payment is ready.

Deployments stabilized in 6 minutes 6.87 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-75d9599844-5lf6h	1/1	Running	0
sock-shop	pod/carts-75d9599844-nclz5	1/1	Running	0
sock-shop	pod/carts-db-6794d4f987-4xnn	1/1	Running	0
sock-shop	pod/carts-db-6794d4f987-ldcv4	1/1	Running	0
sock-shop	pod/catalogue-577bcf65d7-nksxg	1/1	Running	0
sock-shop	pod/catalogue-577bcf65d7-rcbhd	1/1	Running	0
sock-shop	pod/catalogue-db-56666f68cd-sdnjp	1/1	Running	0
sock-shop	pod/catalogue-db-56666f68cd-w8s9n	1/1	Running	0
sock-shop	pod/front-end-5cd74f858b-h76kh	1/1	Running	0
sock-shop	pod/front-end-5cd74f858b-r2774	1/1	Running	0
sock-shop	pod/orders-77b6947c8d-f78s6	1/1	Running	0
sock-shop	pod/orders-77b6947c8d-scm7g	1/1	Running	0
sock-shop	pod/orders-db-59f7c5f79d-t92qn	1/1	Running	0
sock-shop	pod/orders-db-59f7c5f79d-wxq5d	1/1	Running	0
sock-shop	pod/payment-65b4dd6f9b-kzkql	1/1	Running	0
sock-shop	pod/payment-65b4dd6f9b-vr7fp	1/1	Running	0
sock-shop	pod/payment-7855f7898c-s...	6m8s		
sock-shop	replicaset.apps/orders-8575b8c969	0	0	0
sock-shop	replicaset.apps/orders-db-59f7c5f79d	2	2	2
sock-shop	replicaset.apps/orders-db-76dfdc6865	0	0	0
sock-shop	replicaset.apps/payment-65b4dd6f9b	2	2	2
sock-shop	replicaset.apps/payment-7855f7898c	0	0	0
sock-shop	replicaset.apps/queue-master-584db96944	0	0	0
sock-shop	replicaset.apps/queue-master-778c957b8f	2	2	2
sock-shop	replicaset.apps/rabbitmq-6db7899db6	2	2	2
sock-shop	replicaset.apps/rabbitmq-7f84854df5	0	0	0
sock-shop	replicaset.apps/session-db-6fc87fc86c	0	0	0
sock-shop	replicaset.apps/session-db-7ff969cbdd	2	2	2
sock-shop	replicaset.apps/shipping-5f477dd4bd	2	2	2
sock-shop	replicaset.apps/shipping-77669895df	0	0	0
sock-shop	replicaset.apps/user-86bfdd744f	0	0	0
sock-shop	replicaset.apps/user-86d655596	2	2	2
sock-shop	replicaset.apps/user-db-555dc88f6	2	2	2
sock-shop	replicaset.apps/user-db-85c4bb8bd7	0	0	0

## 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}, "memory": {"workers": 2, "size": "256MB"}}, "containerNames": ["carts-db"]}}`

The current K8s manifests show that the 'carts-db' deployment has added resource requests for CPU and memory, which were not present in the previous manifests. However, this change does not affect the scope of the fault injection, as the target remains the same: the 'carts-db' pods in the 'sock-shop' namespace. The label selector 'name: carts-db' is still valid and correctly identifies the target pods for the StressChaos fault. Therefore, there is no need to adjust the scope of the fault injection based on 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: `10s`

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 or namespace used for the PodChaos fault injection, which targets the 'front-end' pod. The original goal of the chaos experiment is to test the system's resilience by simulating a pod failure in the 'front-end' service. Since the label and namespace remain the same, the scope of the fault injection does not need to be adjusted. The

selector will continue to target the 'front-end' pods in the 'sock-shop' namespace using the label 'name: front-end'.

Next fault injection scope:

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

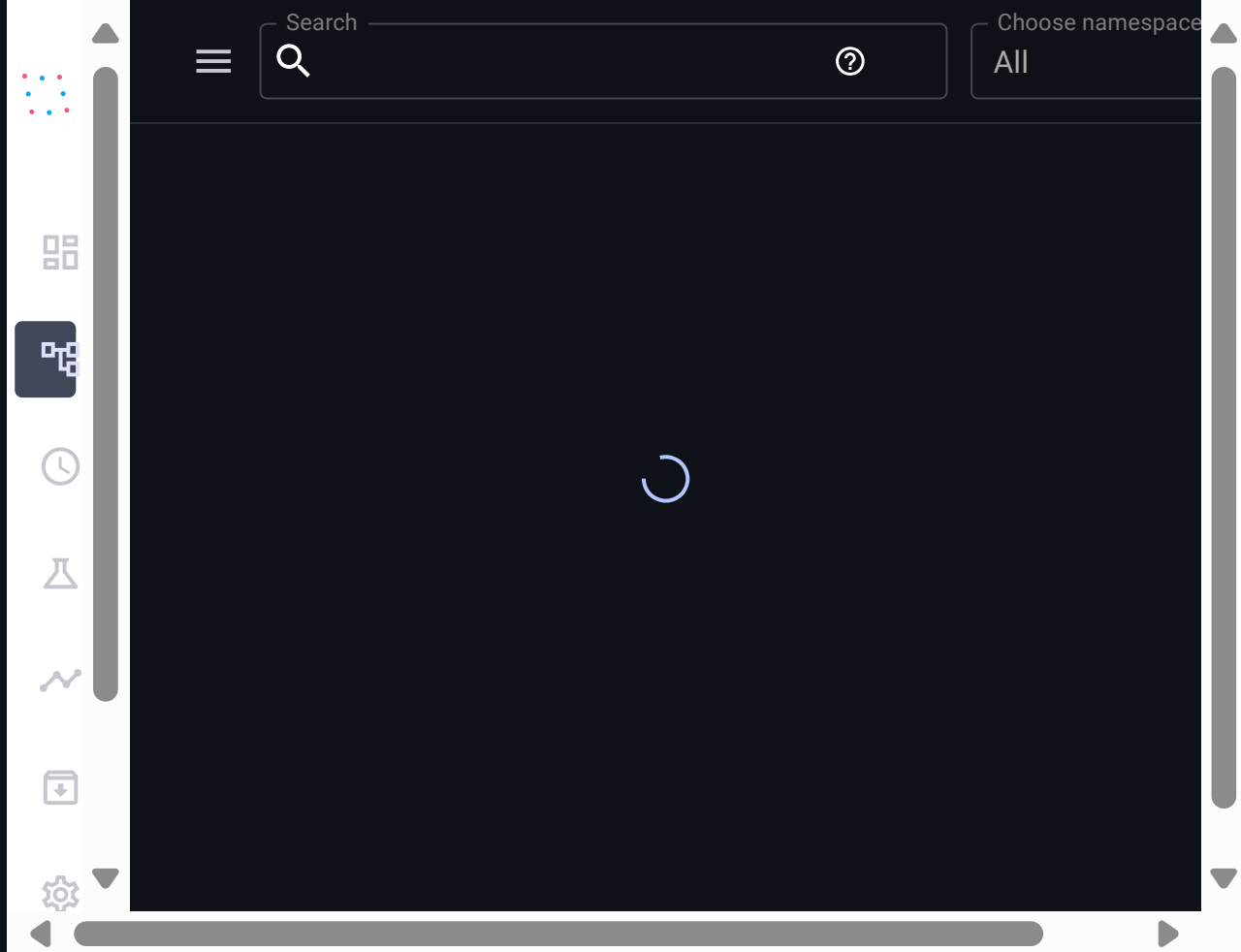
Adjusted unittest

The unit test is specifically targeting the 'carts-db' deployment, checking if the number of ready replicas matches the total replicas. In the reconfigured manifests, the 'carts-db' deployment has not undergone any changes in terms of the number of replicas or other relevant configurations that would affect the unit test. The only addition in the reconfigured manifest is the resource requests for CPU and memory, which do not impact the logic of the unit test. Therefore, the unit test does not require any modifications as the steady state condition it verifies remains unchanged.

Adjusted unittest

The unit test is focused on the 'front-end' deployment, specifically checking if at least one pod is running and ready at all times. In the reconfigured manifests, the 'front-end' deployment has been updated to have 2 replicas instead of 1. However, the unit test's threshold is to ensure that at least 1 pod is running and ready, which aligns with the intent of the test to verify redundancy. Since the threshold remains the same and the reconfiguration was made to ensure the system satisfies this threshold, the unit test does not require any modification. The test will still verify that at least 1 pod is running and ready, which is consistent with the reconfigured state of having 2 replicas.

**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 high-demand conditions, such as a Black Friday sale. The system is composed of various Kubernetes manifests that define deployments and services for different components of the Sock Shop application, a microservices-based e-commerce platform.

### Step 0: User-input Understanding

The system's architecture was analyzed through its Kubernetes manifests, which include definitions for namespaces, deployments, and services. Key components such as 'carts', 'catalogue', 'orders', 'payment', 'shipping', 'user', and their respective databases were identified. Each component's configuration, including replicas, resource limits, security contexts, and service ports, was reviewed to understand the system's current state and potential weaknesses.

# Identified Resiliency Issues

Several resiliency issues were identified:

1. **Missing Resource Requests:** Some deployments, like 'carts-db', lacked resource requests, risking scheduling issues under resource constraints.
2. **Single Replica Deployment:** The 'front-end' deployment had only one replica, making it vulnerable to downtime if the pod failed.
3. **Missing Liveness and Readiness Probes:** Many deployments lacked these probes, which are crucial for Kubernetes to manage pod health.
4. **Hardcoded Environment Variables:** Security vulnerabilities were noted due to hardcoded passwords in environment variables.
5. **Missing Service Port Configuration:** Some services had commented-out port configurations, leading to potential connectivity issues.

## Step 1: Hypothesis Definition

The hypothesis was that the system would maintain its steady states even when faults were injected. Two steady states were defined:

- **Carts-db Replicas Running:** Ensuring 'carts-db' replicas are ready 90% of the time during a 1-minute period.
- **Front-end Replica Running:** Ensuring the 'front-end' pod is ready 95% of the time during a 1-minute period.

## Step 2.1: Chaos-Engineering Experiment

The experiment was structured into three phases, each lasting 20 seconds:

1. **Pre-validation Phase:** Verified the system's initial steady state.
2. **Fault Injection Phase:** Simulated high load and pod failure using Chaos Mesh, injecting StressChaos on 'carts-db' and PodChaos on 'front-end'.
3. **Post-validation Phase:** Checked if the system returned to its steady states.

## Step 2.2, 3, 4: Experiment Execution, Analysis, and Improvement

**1st Try Results:**

- The 'carts-db' component maintained its steady state, but the 'front-end' component failed post-validation due to its single replica configuration.

**Analysis:**

- The single replica of 'front-end' led to downtime when the pod was killed, highlighting the need for redundancy.

#### Improvements:

- Increased 'front-end' replicas from 1 to 2.
- Added resource requests to 'carts-db' to ensure scheduling under constraints.

#### 2nd Try Results:

- All unit tests passed, confirming the system's resilience and recovery capabilities after improvements.

The experiment successfully identified and addressed critical resilience issues, enhancing the system's fault tolerance and availability.

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