Configuration Manual

MSc Research Project
Cloud Computing

Jialing Chen
Student ID: x23158131

School of Computing
National College of Ireland

Supervisor: Shaguna Gupta

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Jialing Chen		
Student ID:	x23158131		
Programme:	Cloud Computing	**Year:**	2024
Module:	MSc Research Project		
Lecturer:	Shaguna Gupta		
Submission Due Date:	01/09/2025		
Project Title:	Rule-based Task Scheduling in Cloud-Edge Computing for Energy and Resource Utilization Optimization		
Word Count:	2389		

Signature: Jialing Chen

Date: 26th August, 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Jialing Chen
23158131

1 Introduction

This manual explains how to set up and run the experiments for the rule-based task scheduling system in a cloud-edge environment. The setup uses basic tools like Minikube for creating a local Kubernetes cluster, and common monitoring tools like Prometheus and Grafana. Each step is explained in simple terms to make it easy to follow and repeat the experiments.

2 Environment Setup

2.1 Basic Requirements

Running these tests needs a good computer with enough power. The basic setup should have 8GB of memory and 4 processor cores to work well. While this seems like a lot of power, most modern computers have these features. If the computer isn't strong enough, the tests might be slow or have problems, since it needs to run both cloud and edge parts together.

The first program to install is Docker Desktop. This needs to be ready before anything else starts. Docker helps by putting programs into boxes that can run anywhere. It's like packing a suitcase with everything needed for a trip - once packed, it works the same way everywhere it goes.

Two more tools help run Kubernetes on the computer. Kubectl works like a remote control, sending commands to Kubernetes. Minikube creates a small test system that acts like a real cloud setup. This lets anyone try out different settings without spending money on big cloud services - just like having a small practice version at home.

2.2 Creating Kubernetes Cluster

Start by creating a test system with Minikube (Kubernetes.io, 2024) to make some computers act like cloud and edge machines:

1. Get Minikube running with the right amount of computer power:

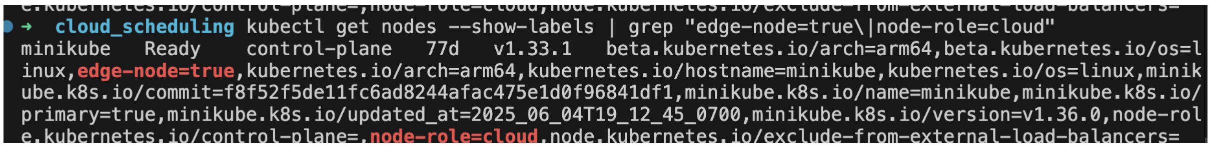
```
minikube start --cpus=4 --memory=8192mb --nodes=2
```

2. Mark each computer as either cloud or edge:

```
kubectl label node minikube node-role="cloud"
kubectl label node minikube-m02 edge-node="true"
```

3. Check that both nodes are ready:

   ```
   kubectl get nodes --show-labels
   ```



```
e.kubernetes.io/control-plane=,node-role=cloud,node.kubernetes.io/exclude-from-external-load-balancers=
● → cloud_scheduling kubectl get nodes --show-labels | grep "edge-node=true\|node-role=cloud"
  minikube   Ready    control-plane   77d   v1.33.1   beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=l
  inux,edge-node=true,kubernetes.io/arch=arm64,kubernetes.io/hostname=minikube,kubernetes.io/os=linux,minik
  ube.k8s.io/commit=f8f52f5de11fc6ad8244afac475e1d0f96841df1,minikube.k8s.io/name=minikube,minikube.k8s.io/
  primary=true,minikube.k8s.io/updated_at=2025_06_04T19_12_45_0700,minikube.k8s.io/version=v1.36.0,node-rol
  e.kubernetes.io/control-plane=,node-role=cloud,node.kubernetes.io/exclude-from-external-load-balancers=
```

Figure 1: Node Labels Check Result

2.3 Setting Up Monitoring Tools

2.3.1 Installing Prometheus

Prometheus (Prometheus.io, 2024) helps collect important information about how the system is running:

1. Add Prometheus repository:

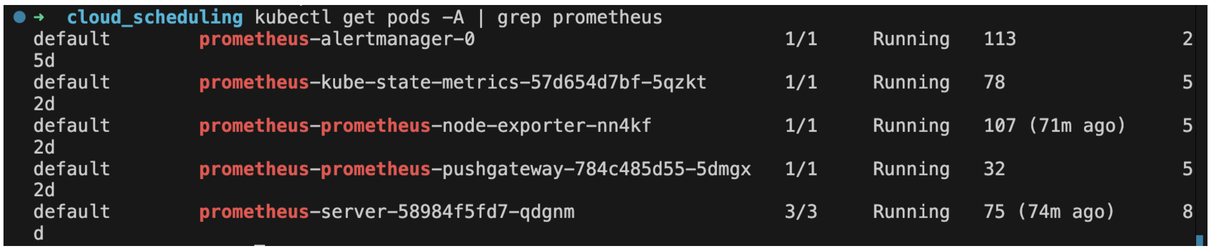
   ```
   helm repo add prometheus-community \
   https://prometheus-community.github.io/helm-charts
   ```

2. Create monitoring namespace:

   ```
   kubectl create namespace monitoring
   ```

3. Install Prometheus:

   ```
   helm install prometheus prometheus-community/kube-prometheus-stack \
   -n monitoring
   ```



```
● → cloud_scheduling kubectl get pods -A | grep prometheus
  default       prometheus-alertmanager-0                              1/1     Running   113              2
  5d
  default       prometheus-kube-state-metrics-57d654d7bf-5qzkt         1/1     Running   78               5
  2d
  default       prometheus-prometheus-node-exporter-nn4kf              1/1     Running   107 (71m ago)    5
  2d
  default       prometheus-prometheus-pushgateway-784c485d55-5dmgx     1/1     Running   32               5
  2d
  default       prometheus-server-58984f5fd7-qdgnm                     3/3     Running   75 (74m ago)     8
  d
```

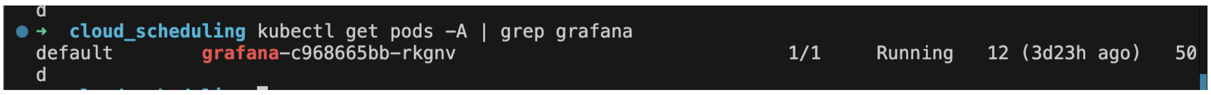
Figure 2: Prometheus Installation

2.3.2 Setting Up Grafana

Grafana (Grafana Labs, 2024) helps see the collected information in easy-to-understand graphs:

1. Access Grafana dashboard:

   ```
   kubectl port-forward svc/prometheus-grafana 3000:80 -n monitoring
   ```



```
d
● → cloud_scheduling kubectl get pods -A | grep grafana
default          grafana-c968665bb-rkgnv                           1/1     Running   12 (3d23h ago)   50
d
```

Figure 3: Grafana Dashboard Access

Import basic dashboard templates for:

- CPU usage monitoring
- Memory usage tracking

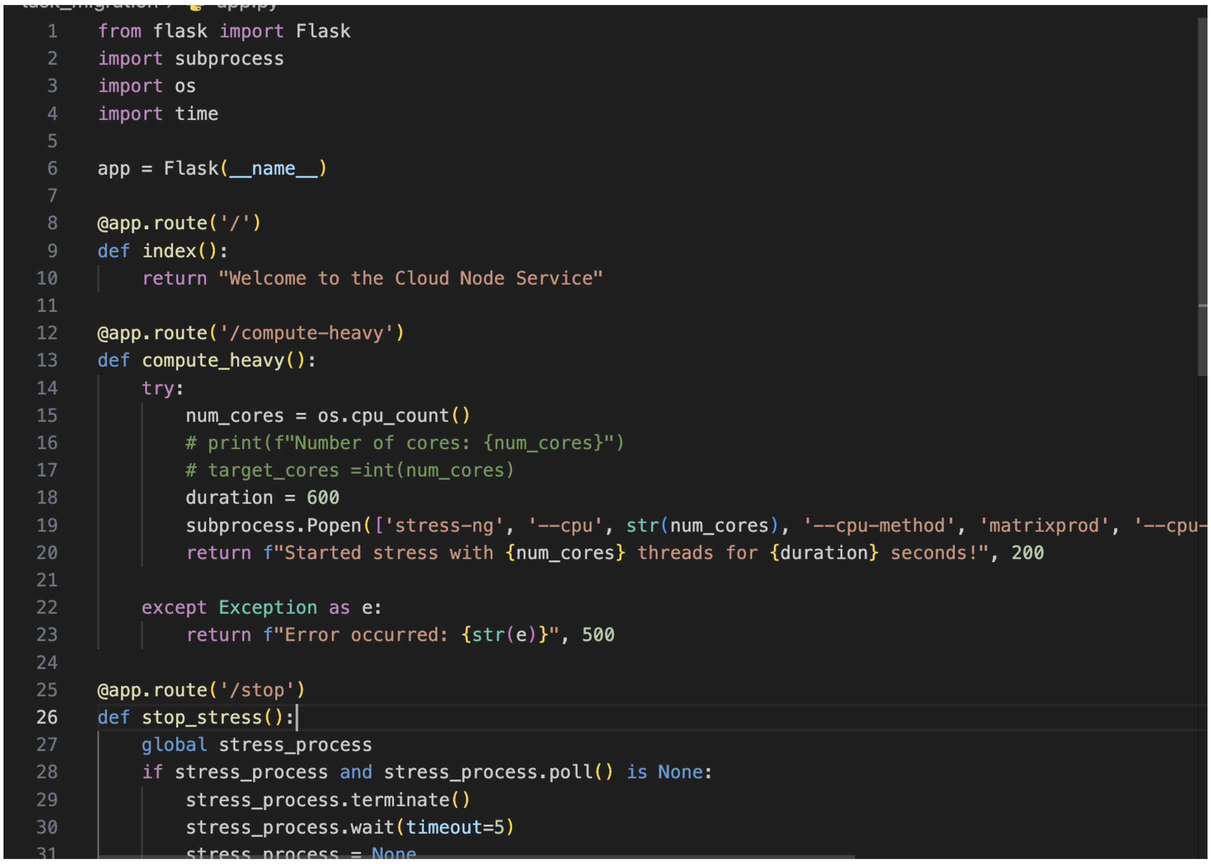
3 Experiment Configurations

3.1 Experiment 1: Task Migration (R1)

This experiment shows how tasks move between cloud and edge nodes based on CPU usage.

3.1.1 Core Application Setup

The main program is built using Flask and runs different types of tasks. It has simple commands to start and stop heavy workloads. The program is packed into a container to make it easy to run anywhere in the system.

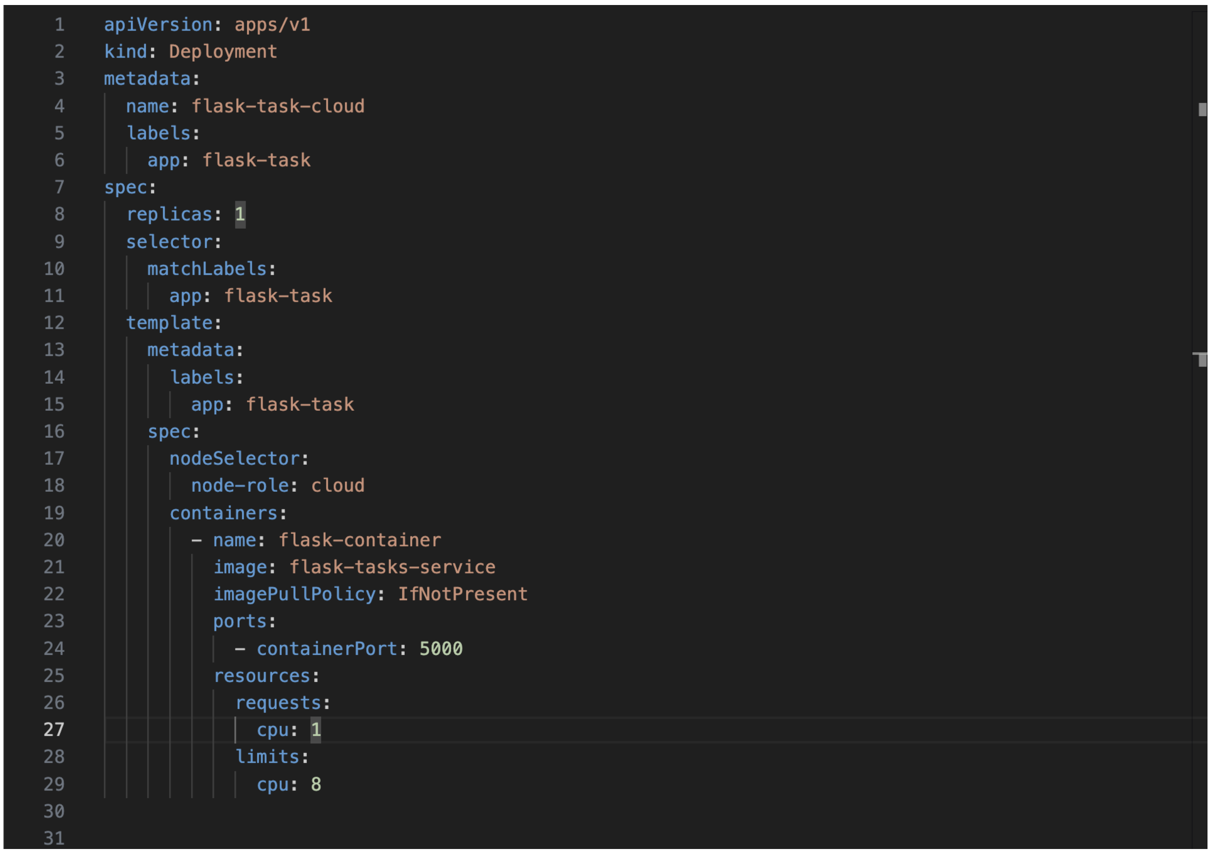


```python
from flask import Flask
import subprocess
import os
import time

app = Flask(__name__)

@app.route('/')
def index():
    return "Welcome to the Cloud Node Service"

@app.route('/compute-heavy')
def compute_heavy():
    try:
        num_cores = os.cpu_count()
        # print(f"Number of cores: {num_cores}")
        # target_cores =int(num_cores)
        duration = 600
        subprocess.Popen(['stress-ng', '--cpu', str(num_cores), '--cpu-method', 'matrixprod', '--cpu-
        return f"Started stress with {num_cores} threads for {duration} seconds!", 200

    except Exception as e:
        return f"Error occurred: {str(e)}", 500

@app.route('/stop')
def stop_stress():
    global stress_process
    if stress_process and stress_process.poll() is None:
        stress_process.terminate()
        stress_process.wait(timeout=5)
        stress_process = None
```

Figure 4: Core Application Code

3.1.2 Cloud Node Setup

The cloud deployment configuration specifies resource limits and node selection rules. The deployment-cloud.yaml sets CPU limits to 8 cores and minimum requests to 1 core, ensuring proper resource allocation for the workload tests.

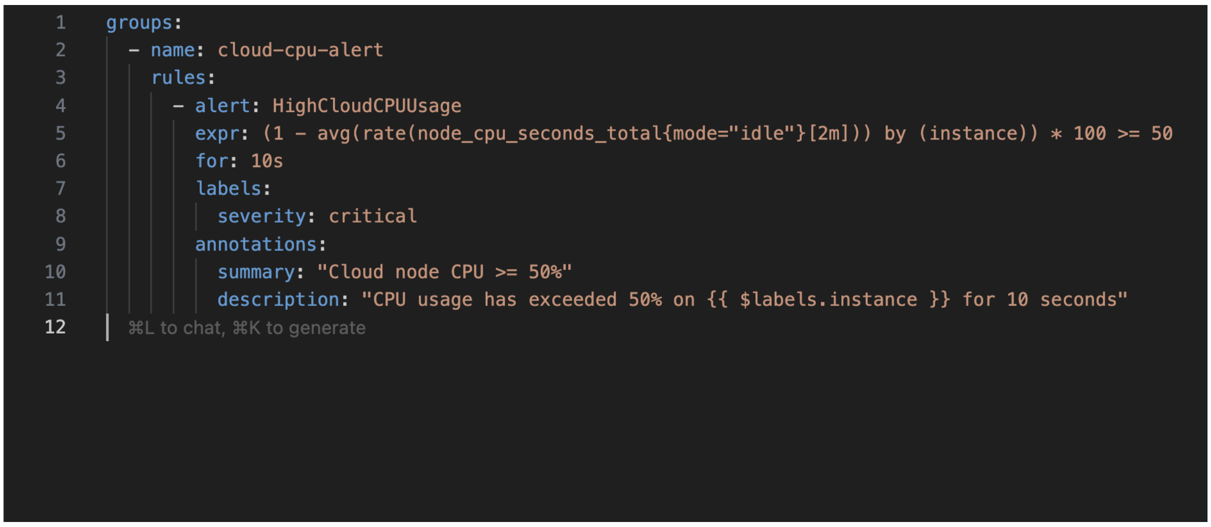


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-task-cloud
  labels:
    app: flask-task
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flask-task
  template:
    metadata:
      labels:
        app: flask-task
    spec:
      nodeSelector:
        node-role: cloud
      containers:
        - name: flask-container
          image: flask-tasks-service
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5000
          resources:
            requests:
              cpu: 1
            limits:
              cpu: 8
```

Figure 5: Cloud Node Deployment Configuration

3.1.3 Monitoring Configuration

Prometheus handles the monitoring setup with custom alert rules. The cpu-alert-rules.yaml defines when migration should trigger based on CPU usage. The prometheus-config.yaml sets up the data collection rules.

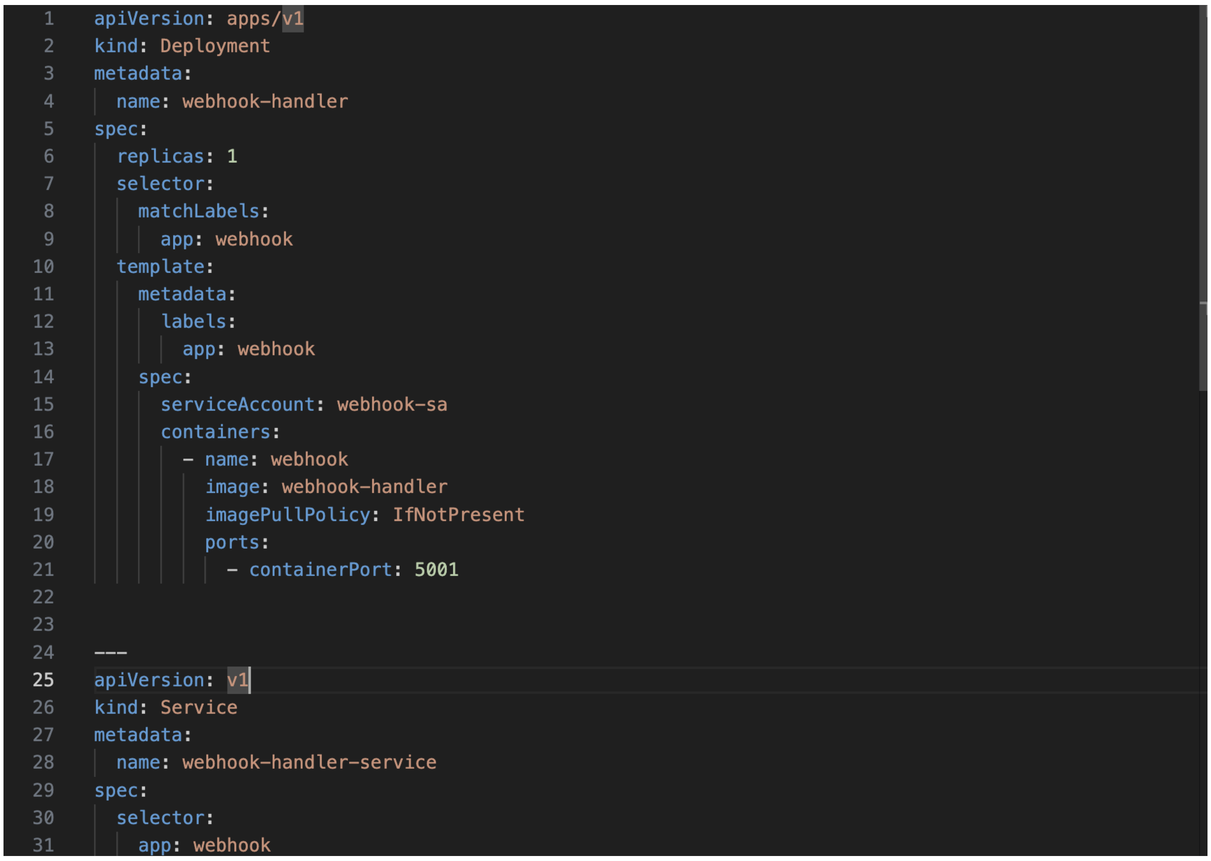


```yaml
groups:
  - name: cloud-cpu-alert
    rules:
      - alert: HighCloudCPUUsage
        expr: (1 - avg(rate(node_cpu_seconds_total{mode="idle"}[2m])) by (instance)) * 100 >= 50
        for: 10s
        labels:
          severity: critical
        annotations:
          summary: "Cloud node CPU >= 50%"
          description: "CPU usage has exceeded 50% on {{ $labels.instance }} for 10 seconds"
```

Figure 6: CPU Alert Rules Configuration

3.1.4 Migration Control

The webhook system manages task migration between nodes. The webhook-deploy.yaml and webhook-rbac.yaml set up the migration service with proper permissions. A custom webhook handler processes migration requests when CPU thresholds are exceeded.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webhook-handler
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webhook
  template:
    metadata:
      labels:
        app: webhook
    spec:
      serviceAccount: webhook-sa
      containers:
      - name: webhook
        image: webhook-handler
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5001


---
apiVersion: v1
kind: Service
metadata:
  name: webhook-handler-service
spec:
  selector:
    app: webhook
```

Figure 7: Webhook Deployment Configuration

3.1.5 Service Configuration

The service-cloud.yaml defines how the application is exposed within the cluster. It sets up the necessary port mappings and service discovery rules for the migration system to work properly.

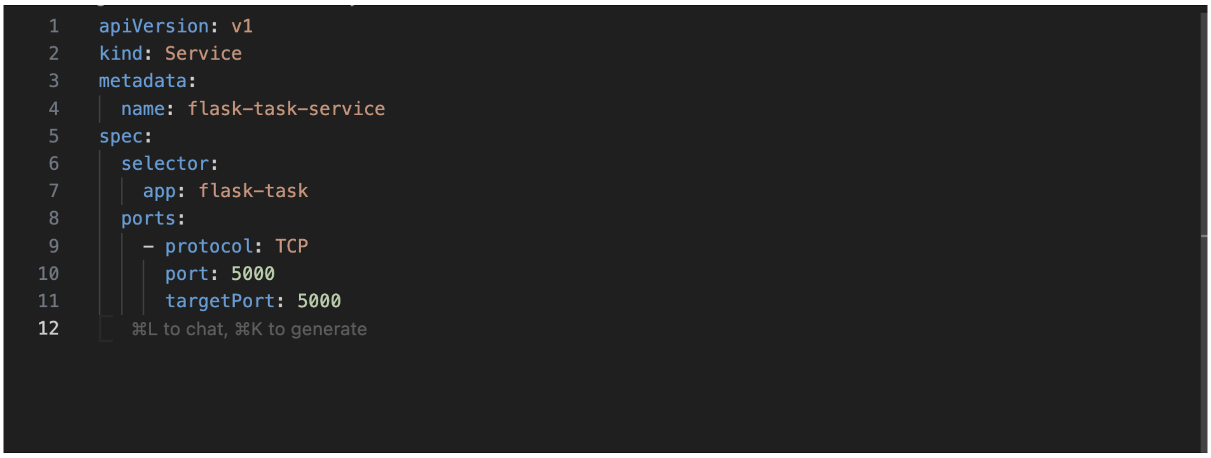

```yaml
apiVersion: v1
kind: Service
metadata:
  name: flask-task-service
spec:
  selector:
    app: flask-task
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
```

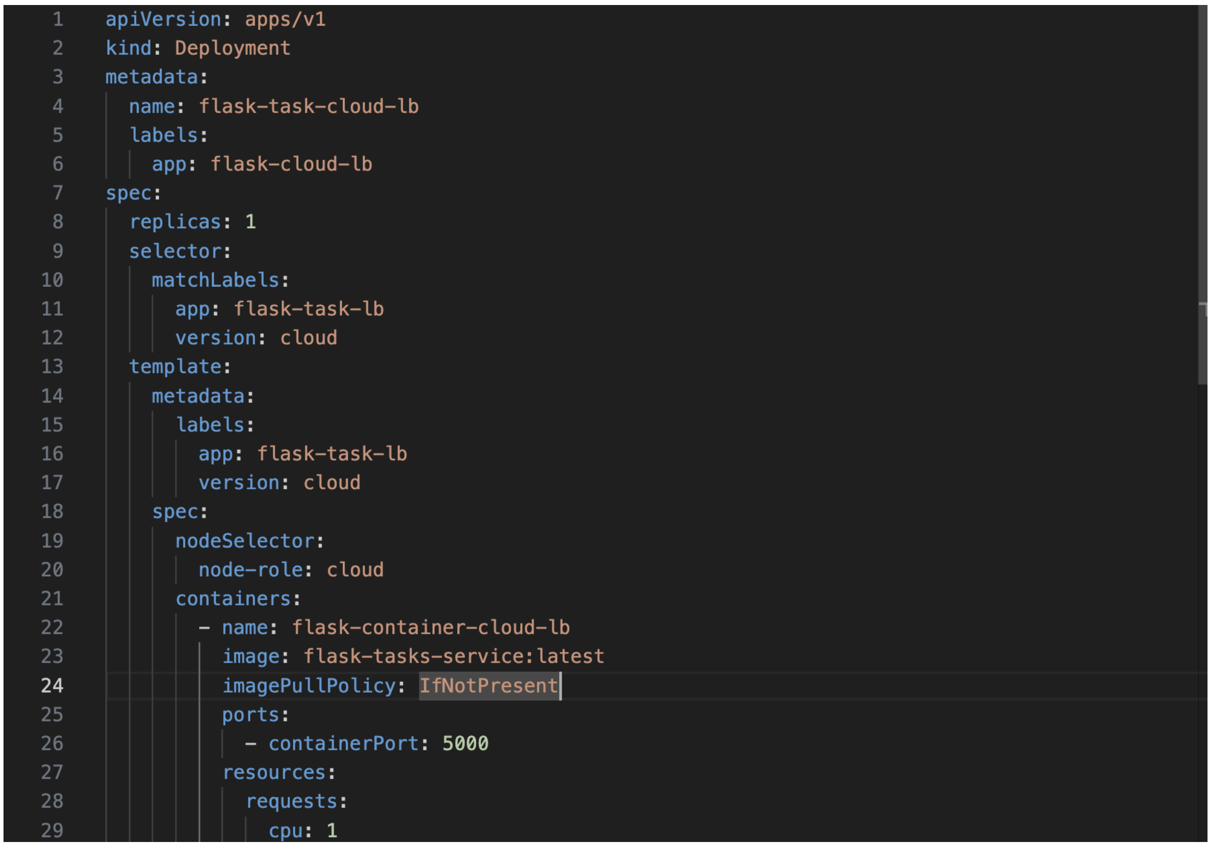
Figure 8: Service Configuration

3.2 Experiment 2: Load Balancing (R2)

This experiment uses Istio (Istio.io, 2024), a powerful service mesh platform, to direct traffic based on user location. Istio helps manage network traffic and implement location-based routing rules in a cloud-edge environment.

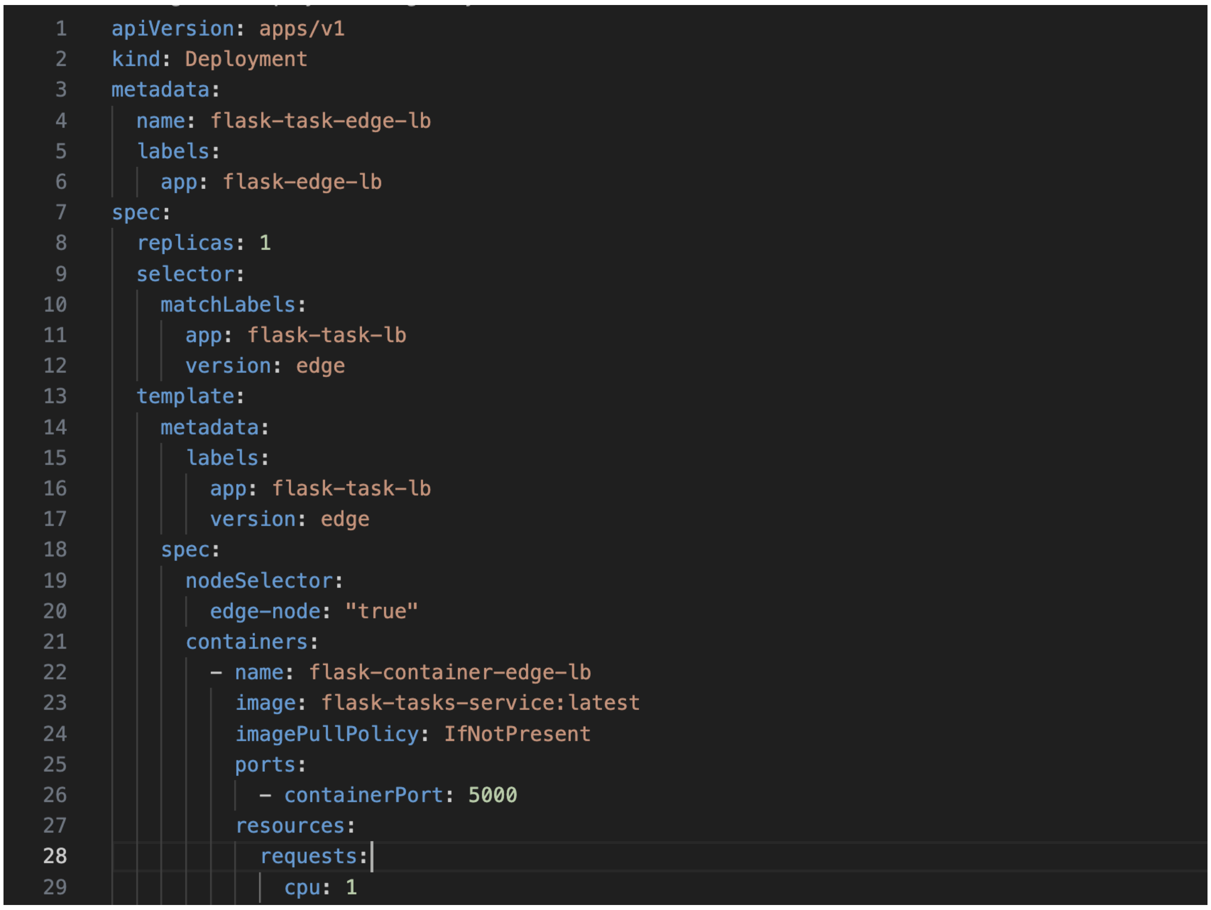
3.2.1 Basic Service Setup

Programs run on both cloud and edge computers. Each one gets different power limits. Cloud computers can do more heavy work with 8 processors, while edge computers handle smaller tasks. This helps share work across different places effectively.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-task-cloud-lb
  labels:
    app: flask-cloud-lb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flask-task-lb
      version: cloud
  template:
    metadata:
      labels:
        app: flask-task-lb
        version: cloud
    spec:
      nodeSelector:
        node-role: cloud
      containers:
        - name: flask-container-cloud-lb
          image: flask-tasks-service:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5000
          resources:
            requests:
              cpu: 1
```

Figure 9: Cloud Server Configuration

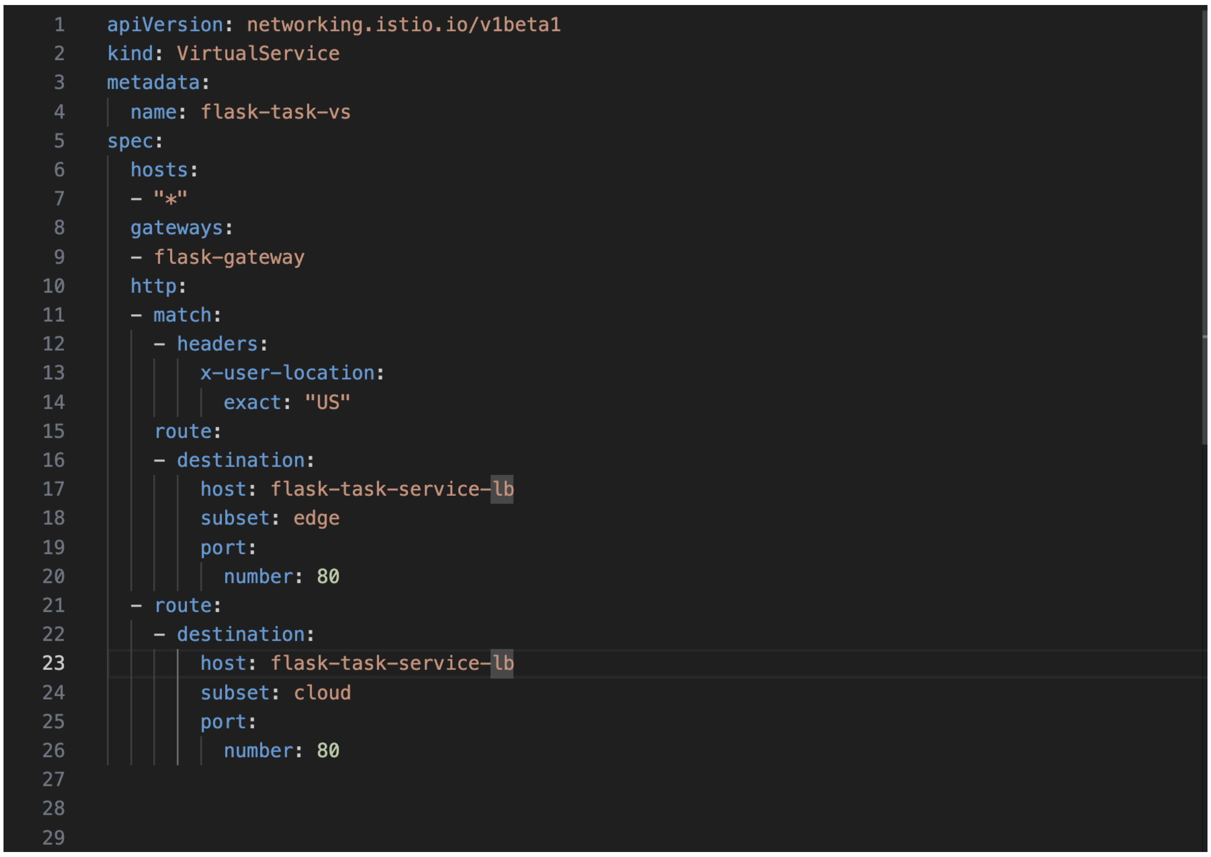


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-task-edge-lb
  labels:
    app: flask-edge-lb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flask-task-lb
      version: edge
  template:
    metadata:
      labels:
        app: flask-task-lb
        version: edge
    spec:
      nodeSelector:
        edge-node: "true"
      containers:
        - name: flask-container-edge-lb
          image: flask-tasks-service:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5000
          resources:
            requests:
              cpu: 1
```

Figure 10: Edge Server Configuration

3.2.2 Traffic Routing Setup

Basic rules help control where requests go in the network. Each request gets checked to see where it came from. US requests go to nearby edge computers for faster service. Everything else gets sent to cloud computers for processing.

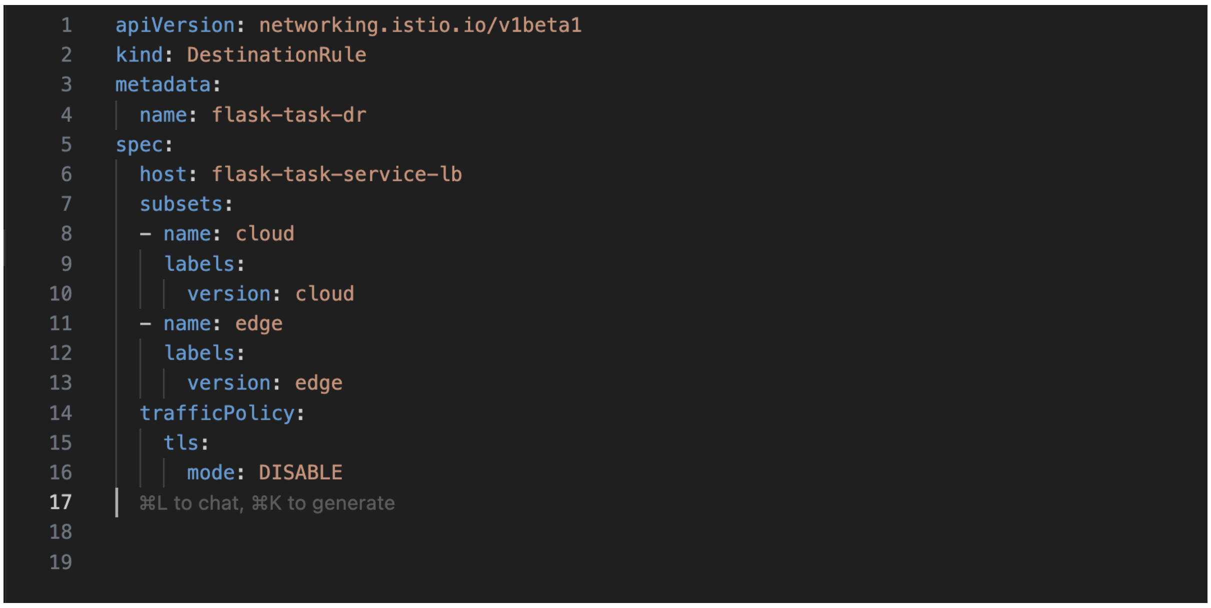


```yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: flask-task-vs
spec:
  hosts:
  - "*"
  gateways:
  - flask-gateway
  http:
  - match:
    - headers:
        x-user-location:
          exact: "US"
    route:
    - destination:
        host: flask-task-service-lb
        subset: edge
        port:
          number: 80
  - route:
    - destination:
        host: flask-task-service-lb
        subset: cloud
        port:
          number: 80
```

Figure 11: Traffic Routing Rules

3.2.3 Server Group Settings

Computers get sorted into cloud or edge groups. This grouping helps control how work moves between different parts of the network. The settings make sure requests go to the right group of servers and keep everything running smoothly.

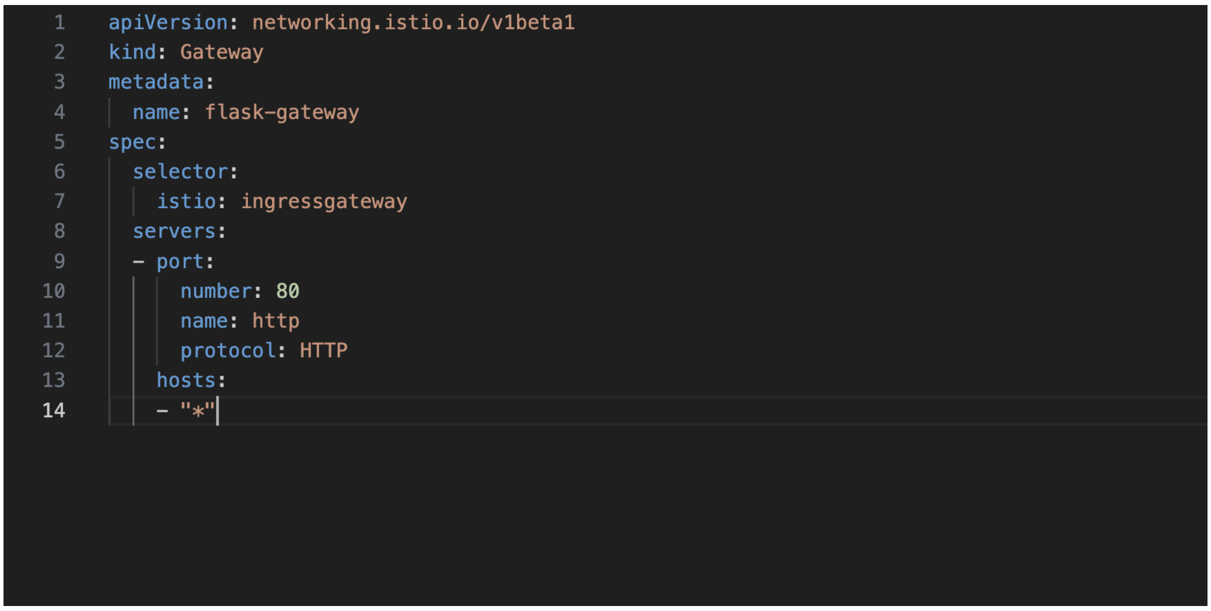


```yaml
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: flask-task-dr
spec:
  host: flask-task-service-lb
  subsets:
  - name: cloud
    labels:
      version: cloud
  - name: edge
    labels:
      version: edge
  trafficPolicy:
    tls:
      mode: DISABLE
  ⌘L to chat, ⌘K to generate
```

Figure 12: Server Group Configuration

3.2.4 Network Gateway Setup

The gateway works like a front door for all incoming traffic. It helps manage how requests come into the system and makes sure they get sent to the right place. This setup helps balance the work across all servers.

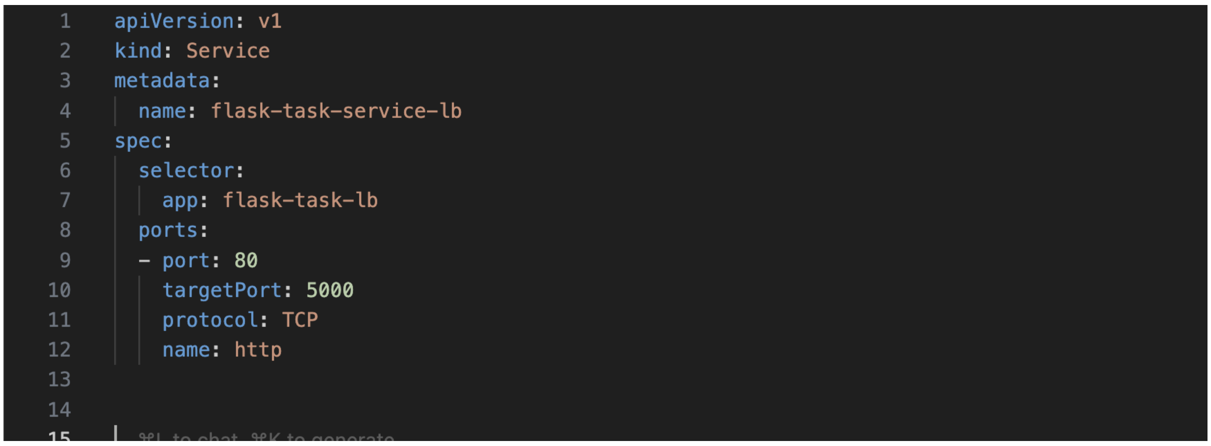


```yaml
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: flask-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

Figure 13: Gateway Configuration

3.2.5 Service Connection Setup

The service settings tell different parts of the system how to talk to each other. They set up the right connections and make sure everything can work together properly. This helps keep the whole system running smoothly.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: flask-task-service-lb
spec:
  selector:
    app: flask-task-lb
  ports:
  - port: 80
    targetPort: 5000
    protocol: TCP
    name: http


  ⌘L to chat, ⌘K to generate
```

Figure 14: Service Connection Settings

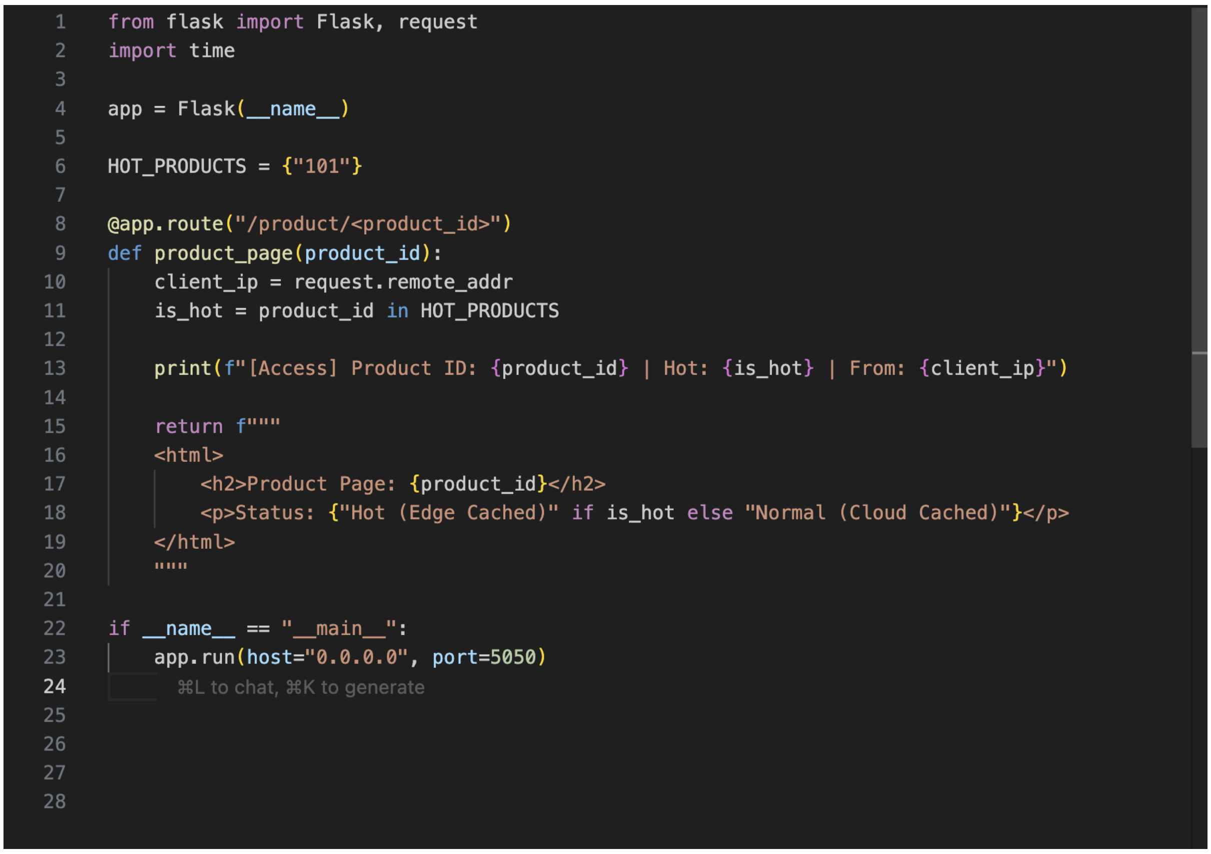
3.3 Experiment 3: Cache Policy (R3)

This test uses Istio (Istio.io, 2024) to handle product requests in a smarter way. When someone looks for a popular product, Istio helps send their request to the right place -

usually to an edge server that already has the product information saved. This makes things faster because the information is closer to where it's needed.

3.3.1 Basic Product Service Setup

The system uses a simple product service that shows different items. It runs on port 5050 and can tell if a product is popular (hot) or normal. The service keeps track of which products are accessed more often.

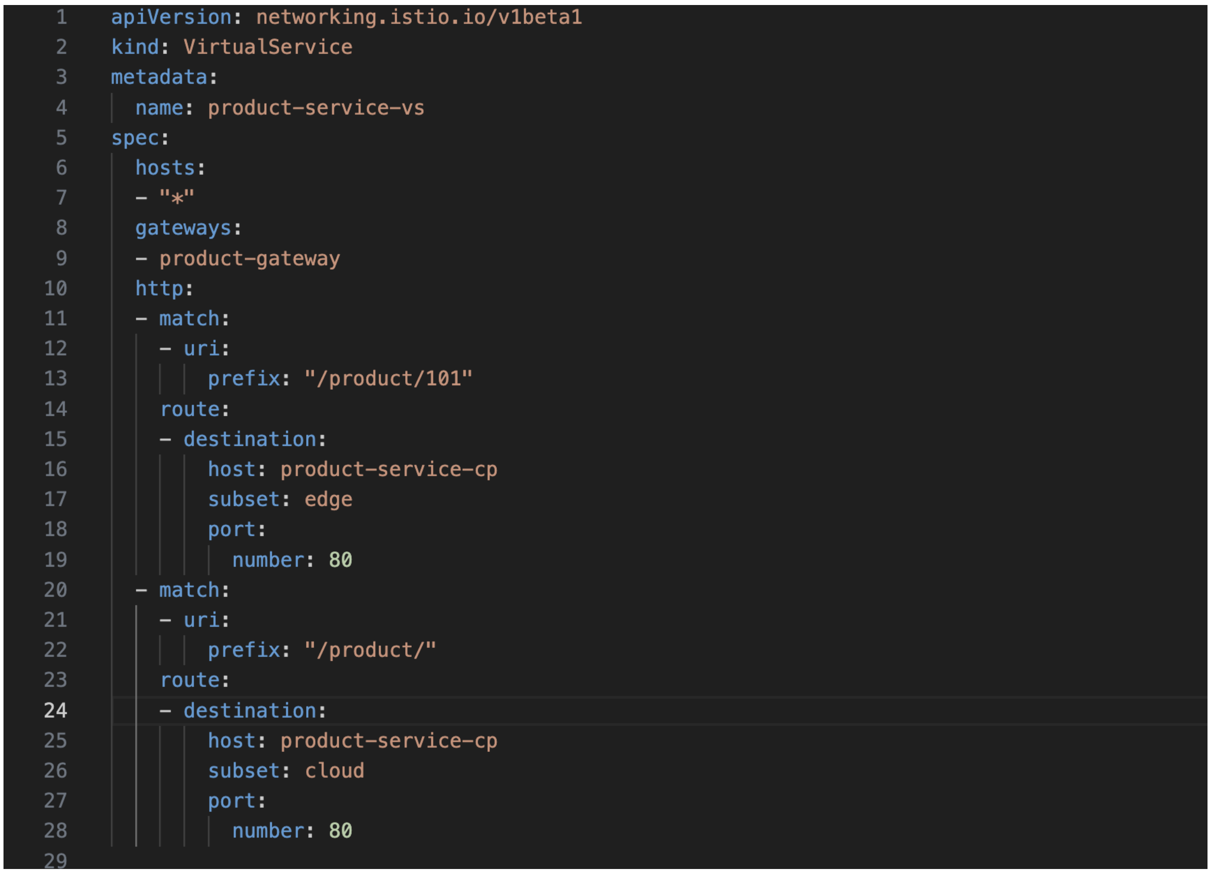


```python
from flask import Flask, request
import time

app = Flask(__name__)

HOT_PRODUCTS = {"101"}

@app.route("/product/<product_id>")
def product_page(product_id):
    client_ip = request.remote_addr
    is_hot = product_id in HOT_PRODUCTS

    print(f"[Access] Product ID: {product_id} | Hot: {is_hot} | From: {client_ip}")

    return f"""
    <html>
        <h2>Product Page: {product_id}</h2>
        <p>Status: {"Hot (Edge Cached)" if is_hot else "Normal (Cloud Cached)"}</p>
    </html>
    """

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5050)
        ⌘L to chat, ⌘K to generate
```

Figure 15: Product Service Implementation

3.3.2 Cache Location Rules

The system has special rules for storing product information. Popular products are saved closer to users (on edge servers), while other products stay in the main cloud storage. This helps get popular items to users faster.

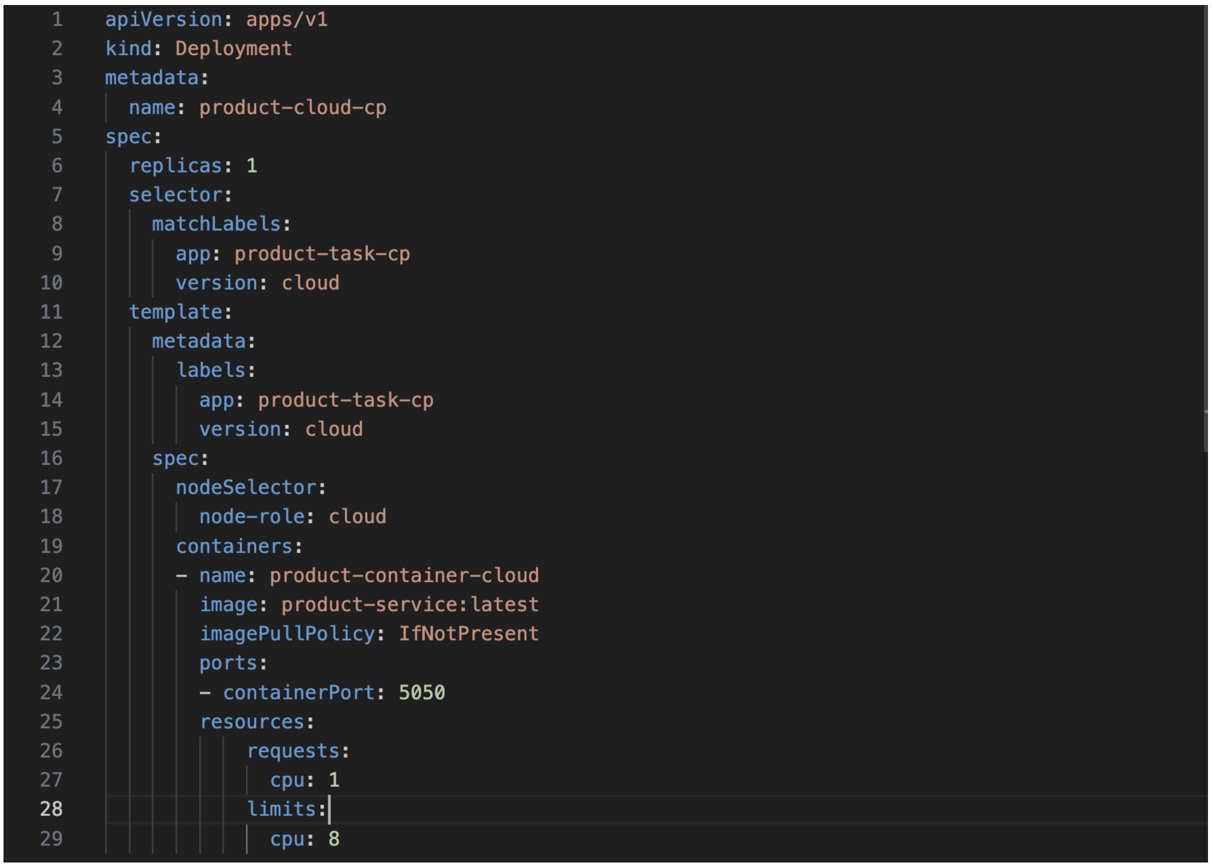


```yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: product-service-vs
spec:
  hosts:
  - "*"
  gateways:
  - product-gateway
  http:
  - match:
    - uri:
        prefix: "/product/101"
    route:
    - destination:
        host: product-service-cp
        subset: edge
        port:
          number: 80
  - match:
    - uri:
        prefix: "/product/"
    route:
    - destination:
        host: product-service-cp
        subset: cloud
        port:
          number: 80
```

Figure 16: Cache Location Rules Configuration

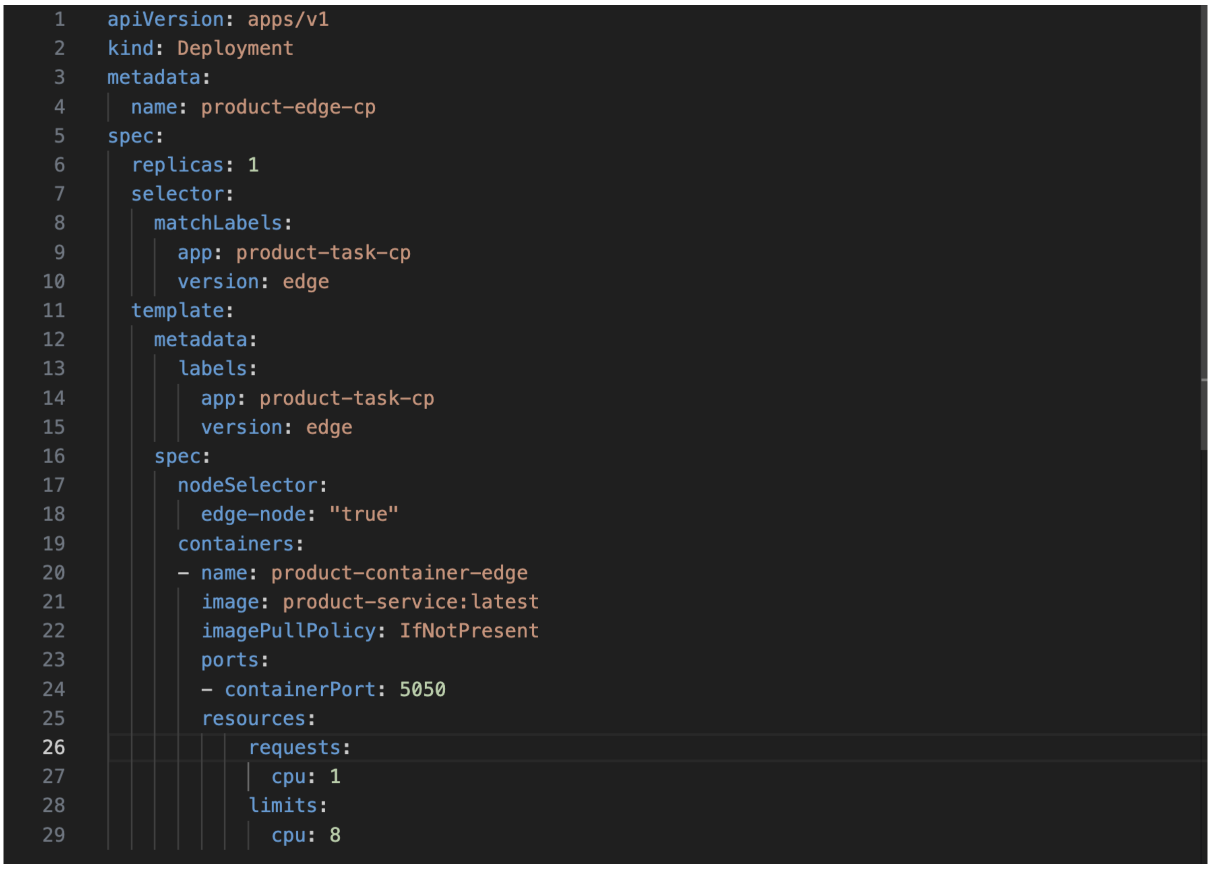
3.3.3 Server Deployment Setup

The system runs on both cloud and edge servers. Each server has its own job - edge servers handle popular items, while cloud servers handle everything else. This split helps manage product information better.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: product-cloud-cp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: product-task-cp
      version: cloud
  template:
    metadata:
      labels:
        app: product-task-cp
        version: cloud
    spec:
      nodeSelector:
        node-role: cloud
      containers:
      - name: product-container-cloud
        image: product-service:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5050
        resources:
            requests:
              cpu: 1
            limits:
              cpu: 8
```

Figure 17: Cloud Server Deployment Configuration

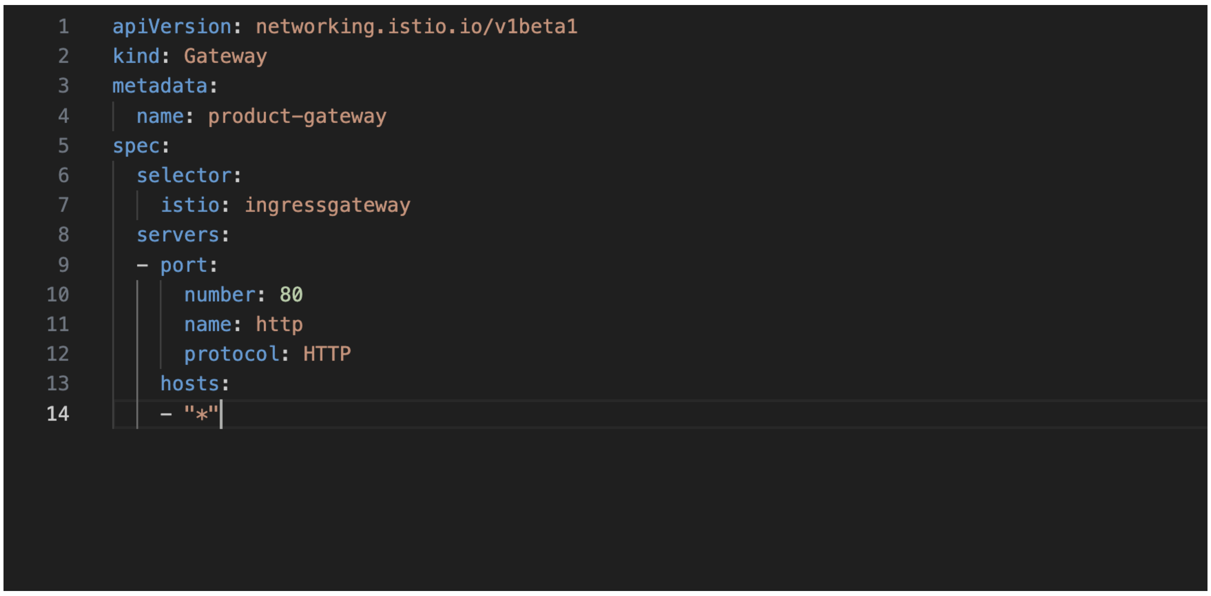


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: product-edge-cp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: product-task-cp
      version: edge
  template:
    metadata:
      labels:
        app: product-task-cp
        version: edge
    spec:
      nodeSelector:
        edge-node: "true"
      containers:
      - name: product-container-edge
        image: product-service:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5050
        resources:
          requests:
            cpu: 1
          limits:
            cpu: 8
```

Figure 18: Edge Server Deployment Configuration

3.3.4 Traffic Management

The system uses a gateway to handle incoming requests. When someone looks for a product, the gateway checks if it's a popular item (like product 101) and sends the request to the right server. This helps get product information faster.

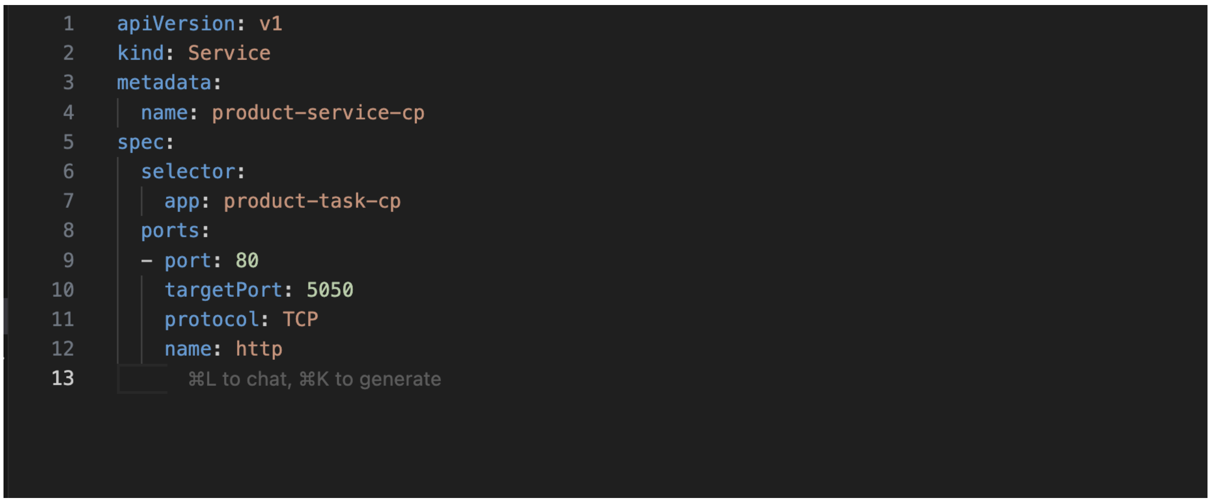


```yaml
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: product-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

Figure 19: Gateway Traffic Management Configuration

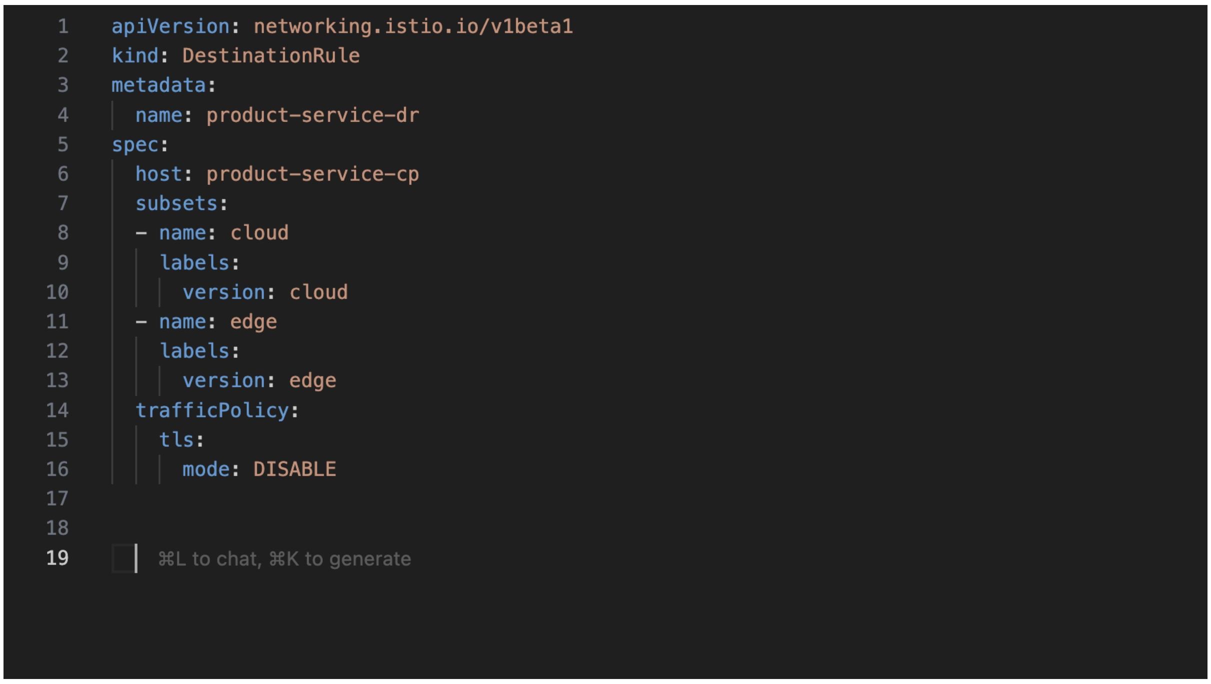
3.3.5 Connection Settings

The service settings make sure all parts of the system can talk to each other. They help connect the cloud and edge servers and make sure product information flows smoothly between them.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: product-service-cp
spec:
  selector:
    app: product-task-cp
  ports:
  - port: 80
    targetPort: 5050
    protocol: TCP
    name: http
      ⌘L to chat, ⌘K to generate
```

Figure 20: Service Connection Configuration



```yaml
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: product-service-dr
spec:
  host: product-service-cp
  subsets:
  - name: cloud
    labels:
      version: cloud
  - name: edge
    labels:
      version: edge
  trafficPolicy:
    tls:
      mode: DISABLE


    ⌘L to chat, ⌘K to generate
```

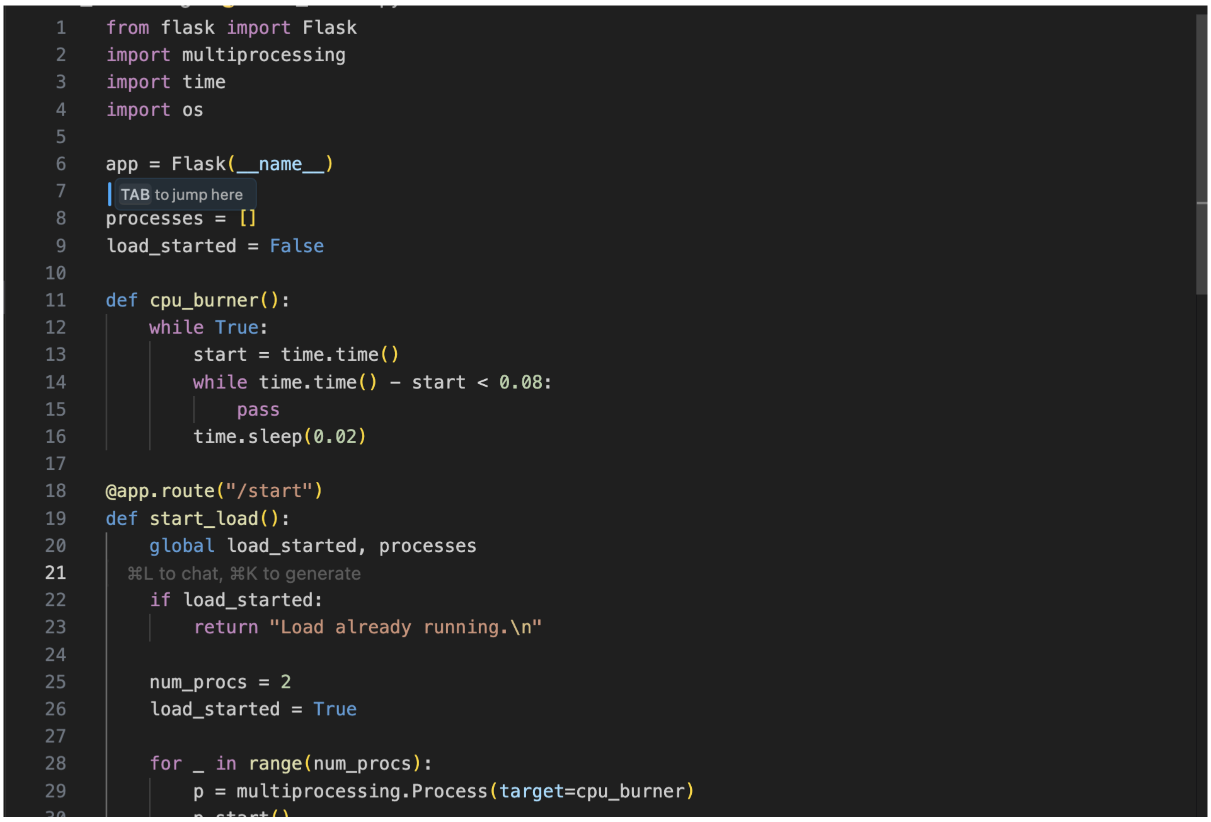
Figure 21: Destination Rules Configuration

3.4 Experiment 4: Cloud Scheduling (R4)

This test uses HPA (Horizontal Pod Autoscaler) (Kubernetes.io/HPA, 2024) to add or remove pods automatically when needed. HPA watches how busy the system is and helps make sure there are enough pods to handle all the work, kind of like opening more checkout counters in a store when it gets crowded.

3.4.1 Basic Service Setup

The system runs a service that can create different levels of work. It has simple commands to start and stop tasks, and can measure how busy the computer is. The service runs on port 6060 and can handle multiple tasks at once.

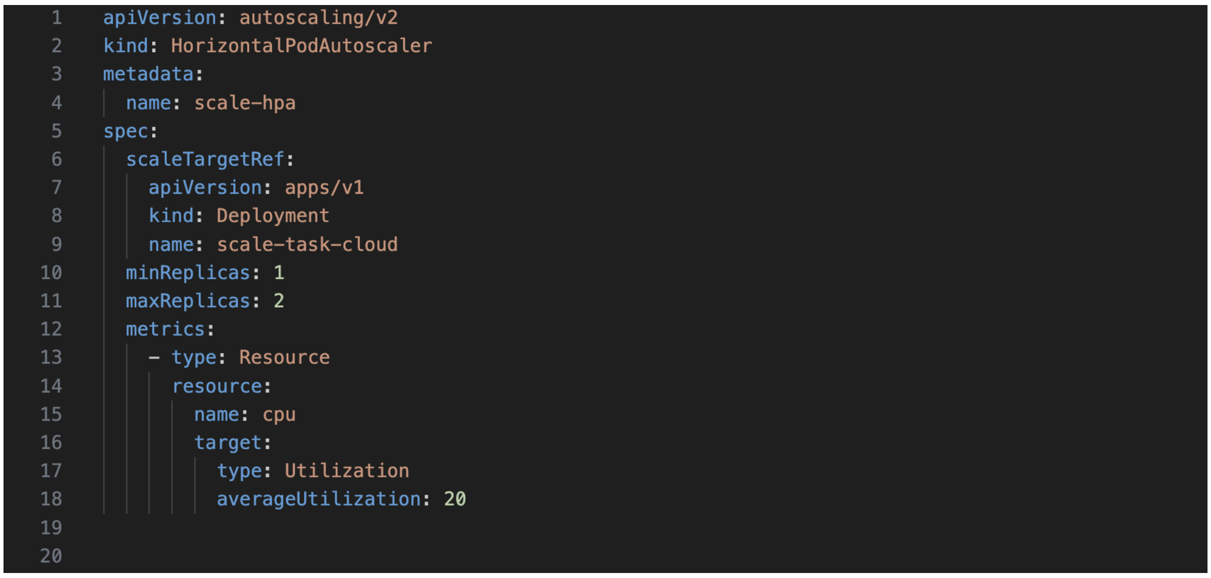


```
from flask import Flask
import multiprocessing
import time
import os

app = Flask(__name__)
TAB to jump here
processes = []
load_started = False

def cpu_burner():
    while True:
        start = time.time()
        while time.time() - start < 0.08:
            pass
        time.sleep(0.02)

@app.route("/start")
def start_load():
    global load_started, processes
    ⌘L to chat, ⌘K to generate
    if load_started:
        return "Load already running.\n"

    num_procs = 2
    load_started = True

    for _ in range(num_procs):
        p = multiprocessing.Process(target=cpu_burner)
```

Figure 22: Service Implementation

3.4.2 Automatic Scaling Rules

The system can grow or shrink based on how busy it gets. When the computer gets too busy (over 20% CPU use), it can add more space to handle the work. It can use between 1 and 2 copies of the service to handle different amounts of work.

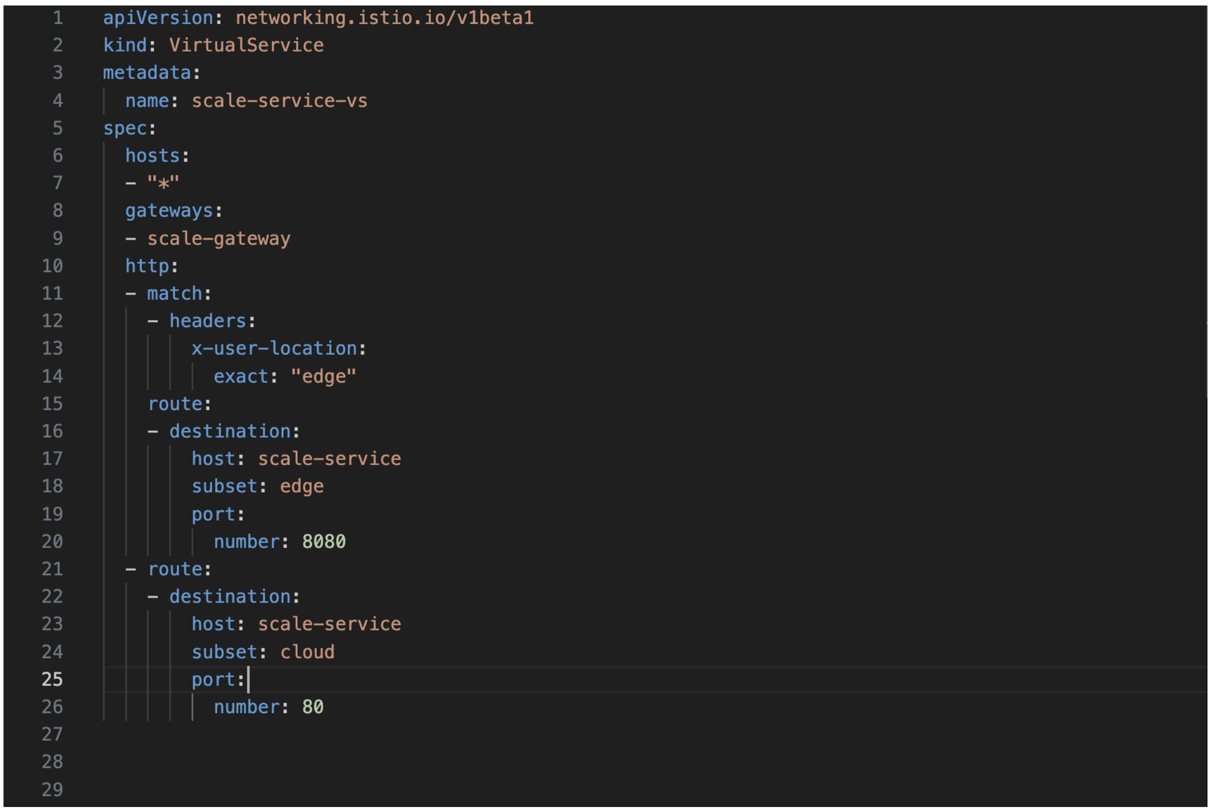


```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: scale-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: scale-task-cloud
  minReplicas: 1
  maxReplicas: 2
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 20


```

Figure 23: HPA Scaling Rules Configuration

3.4.3 Work Distribution Setup

The system splits work between different servers. It uses special rules to decide which server should handle each task. This helps make sure no single server gets too busy.

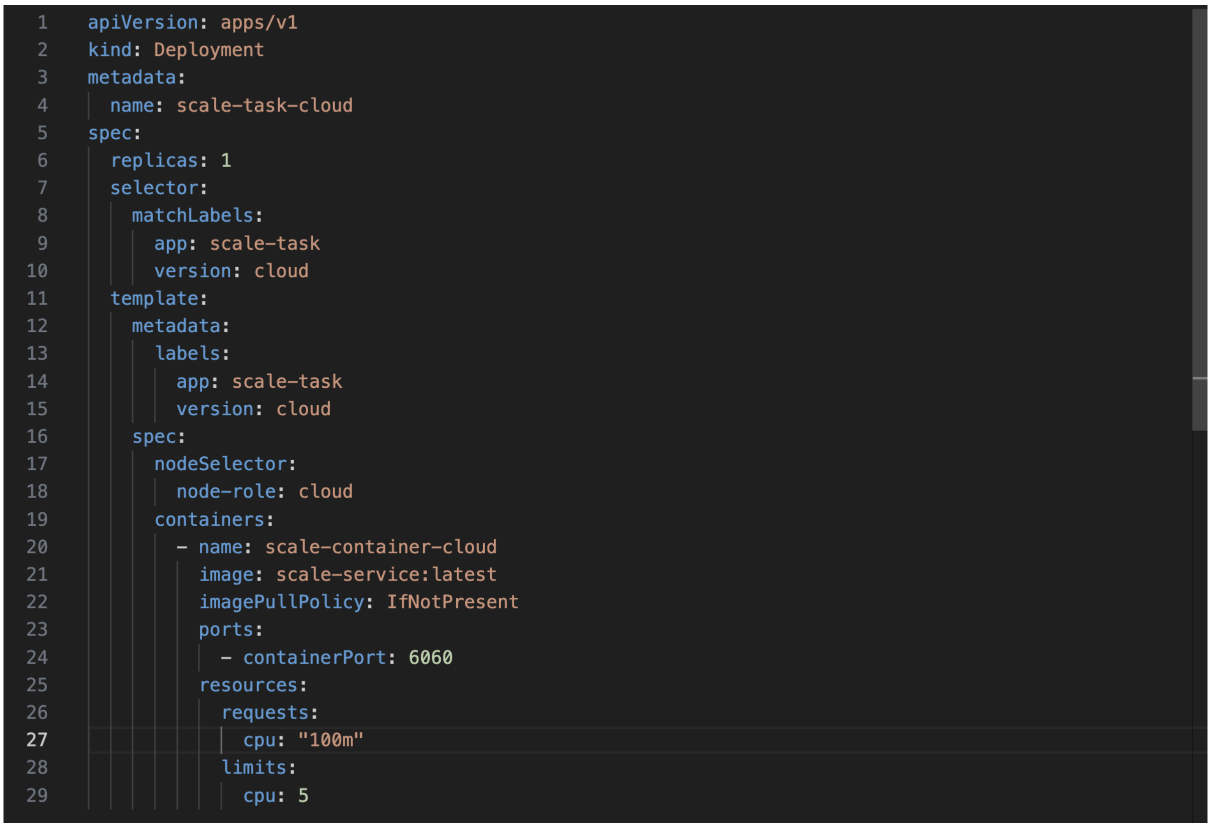


```yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: scale-service-vs
spec:
  hosts:
  - "*"
  gateways:
  - scale-gateway
  http:
  - match:
    - headers:
        x-user-location:
          exact: "edge"
    route:
    - destination:
        host: scale-service
        subset: edge
        port:
          number: 8080
  - route:
    - destination:
        host: scale-service
        subset: cloud
        port:
          number: 80



```

Figure 24: Work Distribution Configuration

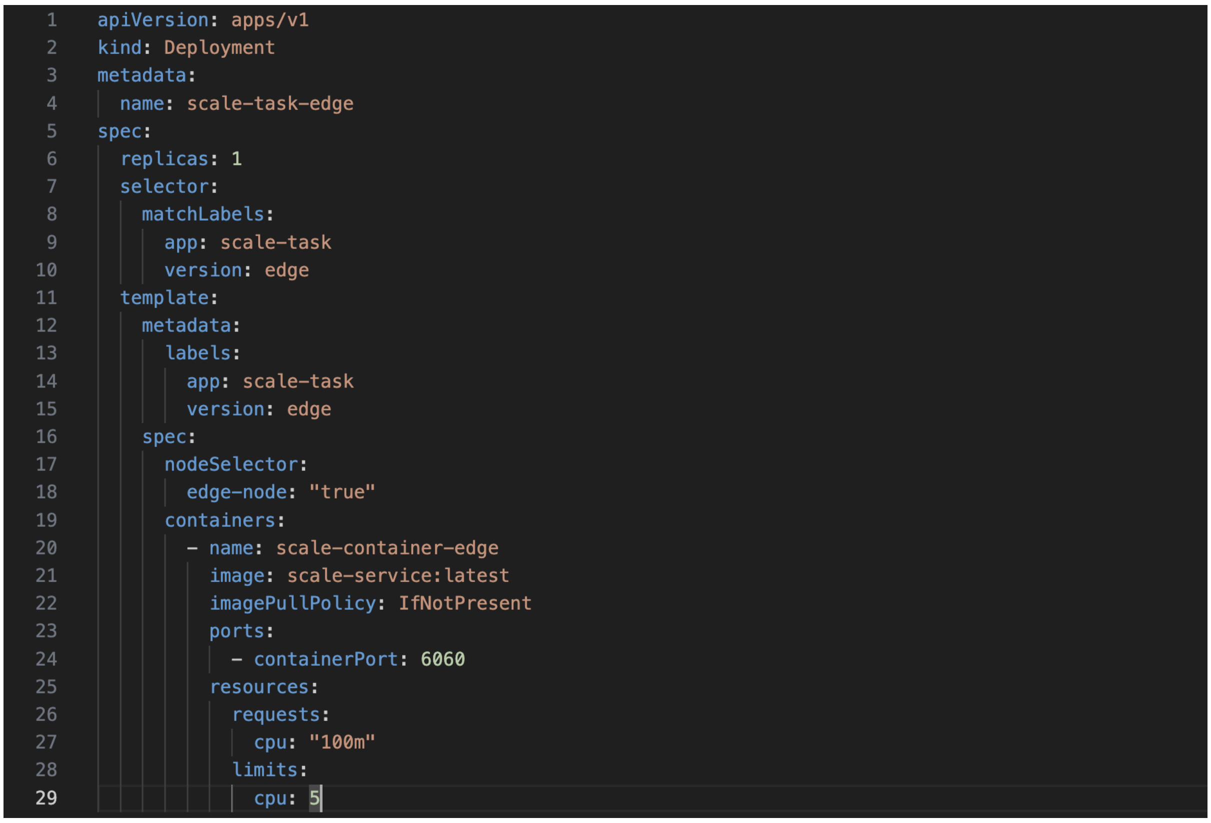
3.4.4 Server Configuration

The system runs on both cloud and edge servers. Each server has its own settings for how much work it can handle. The cloud server can handle bigger tasks, while the edge server takes care of smaller ones.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: scale-task-cloud
spec:
  replicas: 1
  selector:
    matchLabels:
      app: scale-task
      version: cloud
  template:
    metadata:
      labels:
        app: scale-task
        version: cloud
    spec:
      nodeSelector:
        node-role: cloud
      containers:
        - name: scale-container-cloud
          image: scale-service:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 6060
          resources:
            requests:
              cpu: "100m"
            limits:
              cpu: 5
```

Figure 25: Cloud Server Configuration

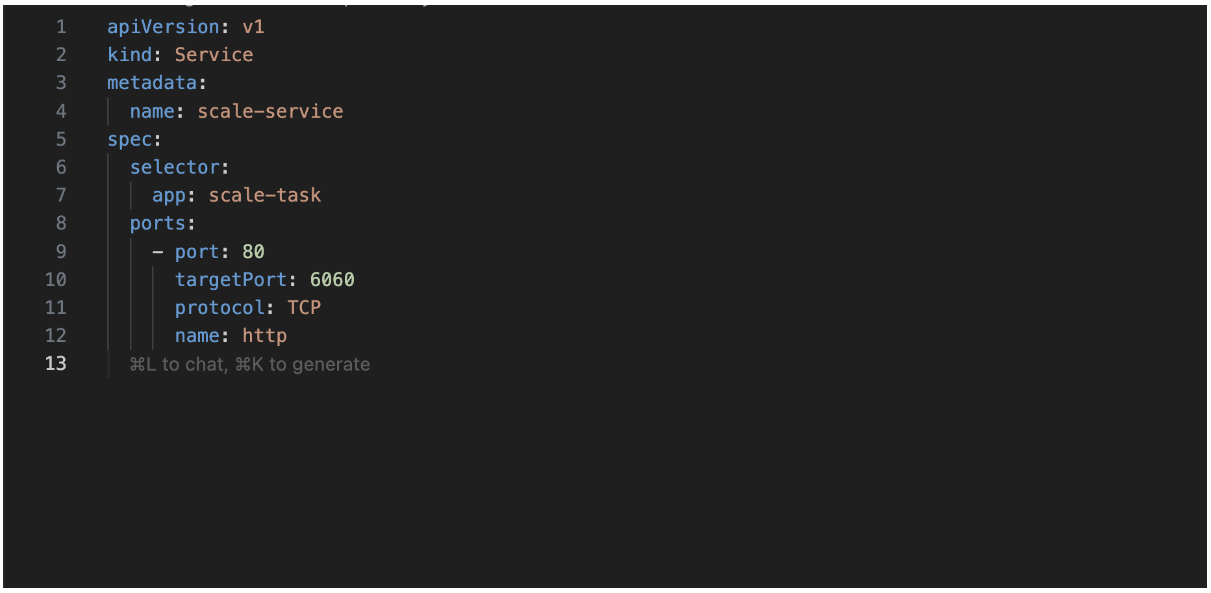


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: scale-task-edge
spec:
  replicas: 1
  selector:
    matchLabels:
      app: scale-task
      version: edge
  template:
    metadata:
      labels:
        app: scale-task
        version: edge
    spec:
      nodeSelector:
        edge-node: "true"
      containers:
        - name: scale-container-edge
          image: scale-service:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 6060
          resources:
            requests:
              cpu: "100m"
            limits:
              cpu: 5
```

Figure 26: Edge Server Configuration

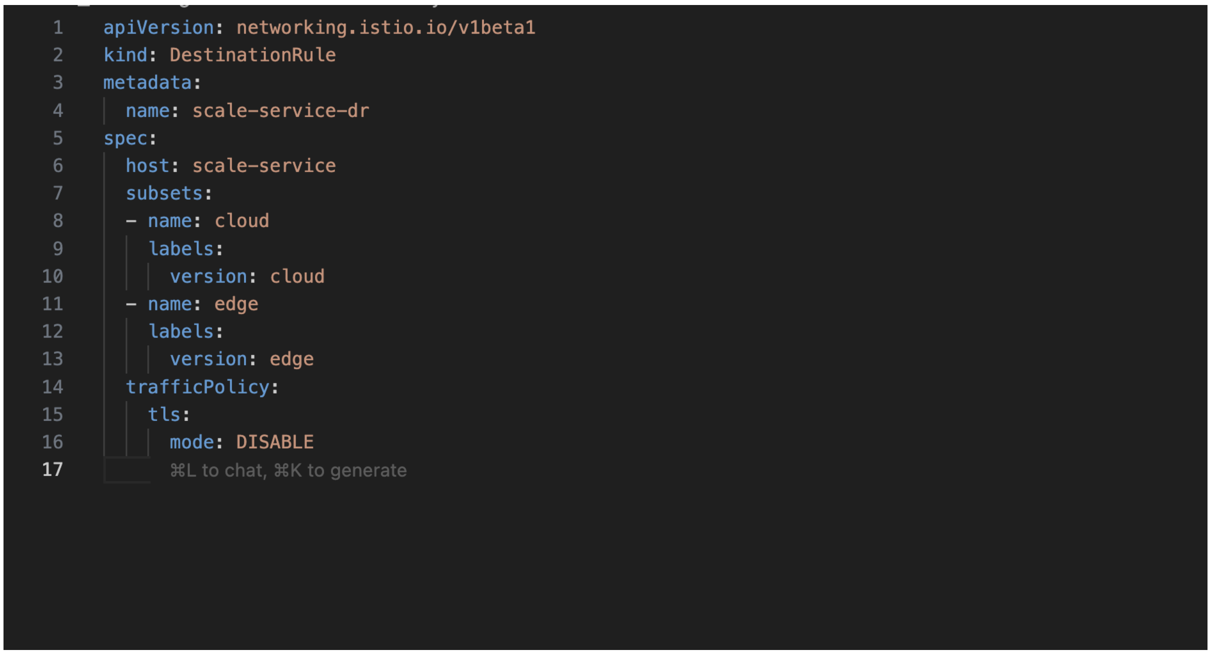
3.4.5 Connection Management

Basic rules help all parts of the network talk to each other. This makes work flow easily between computers and keeps everything running smoothly.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: scale-service
spec:
  selector:
    app: scale-task
  ports:
    - port: 80
      targetPort: 6060
      protocol: TCP
      name: http
  ⌘L to chat, ⌘K to generate
```

Figure 27: Service Connection Configuration



```yaml
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: scale-service-dr
spec:
  host: scale-service
  subsets:
  - name: cloud
    labels:
      version: cloud
  - name: edge
    labels:
      version: edge
  trafficPolicy:
    tls:
      mode: DISABLE
      ⌘L to chat, ⌘K to generate
```

Figure 28: Destination Rules Configuration

4 Running the Experiments

4.1 Task Migration Test (R1)

The first test checks how work moves between cloud and edge when computers get busy. Here are the simple steps:

4.1.1 Getting ready

```
Run programs on cloud computers
kubectl apply -f task-migration/deployment-cloud.yaml
kubectl apply -f task-migration/service-cloud.yaml

Check if everything starts correctly
kubectl get pods -o wide
```

4.1.2 Starting the test

- Visit http://localhost:5000/start-load in a browser
- Give it half a minute to start working
- Watch the Grafana screen as processor use increases

4.1.3 Looking at results

- Work moves to different computers when processor use reaches 50
- Grafana shows how busy each computer gets
- Check `kubectl get pods -o wide` to find where programs are running

4.2 Load Balancing Test (R2)

The second test shows how requests from different locations get handled. Follow these steps:

4.2.1 First steps

```
Run test programs on cloud and edge computers
kubectl apply -f load-balancing/deployment-cloud-lb.yaml
kubectl apply -f load-balancing/deployment-edge-lb.yaml
kubectl apply -f load-balancing/service-lb.yaml
```

4.2.2 Check different places

```
Send test requests from US locations
curl -H "x-user-location: US" http://localhost:8080/start-load

Send test requests from other places
curl http://localhost:8080/start-load
```

4.2.3 Check what happens

- Requests from the US go to the edge node
- Other requests go to the cloud node
- Grafana shows how requests get split up

4.3 Cache Policy Test (R3)

This test checks how the system handles popular products. Here's the process:

4.3.1 Setting up

```
kubectl apply -f cache-policy/deployment-cloud-cp.yaml
kubectl apply -f cache-policy/deployment-edge-cp.yaml
kubectl apply -f cache-policy/service-cp.yaml
```

4.3.2 Testing different products

```
Try getting a popular product (product 101)
http://localhost:5050/product/101

Try getting a regular product (product 102)
http://localhost:5050/product/102
```

4.3.3 Seeing what happens

- Popular product (101) requests go to the edge node
- Regular product requests go to the cloud node
- Grafana shows the speed difference between both

4.4 Cloud Scheduling Test (R4)

This test shows how the system adjusts resources when busy. Here's how to do it:

4.4.1 Getting it running

```
kubectl apply -f cloud-scheduling/deployment-cloud-cs.yaml
kubectl apply -f cloud-scheduling/service-cpu-load.yaml
kubectl apply -f cloud-scheduling/hpa-scale.yaml
```

4.4.2 Making the system busy

- Go to `http://localhost:6060/start` to get things going
- Wait a few minutes to let the load build up
- Check how many pods are running with:

```
kubectl get hpa
kubectl get pods
```

4.4.3 Watching what happens

- When CPU hits 20%, the system adds more pods
- Grafana shows new pods being created
- When things quiet down, extra pods go away

4.5 Important Tips

- Take a break between tests to let the system settle down
- Clean up old test stuff before starting new tests
- Keep an eye on Grafana charts to spot any problems
- If something goes wrong, check the pod logs:

```
kubectl logs <pod-name>
```

5 Troubleshooting Tips

Here are some common problems that might pop up during the experiments and how to fix them:

5.1 Pod Problems

If pods won't start:

```
kubectl describe pod <pod-name>
kubectl get nodes
```

Look for things like not enough CPU or memory on the nodes. Sometimes just waiting a minute and trying again helps.

5.2 Monitoring Issues

If Grafana isn't showing any data:

```
kubectl get pods -n monitoring
kubectl logs -n monitoring prometheus-grafana-<pod-id>
```

Make sure all monitoring pods are running. Sometimes restarting Prometheus helps:

```
kubectl rollout restart deployment prometheus-server -n monitoring
```

5.3 Traffic Routing Problems

If requests aren't going to the right place:

```
kubectl get virtualservice
kubectl describe virtualservice
```

Check if the Istio rules are set up right. The x-user-location header needs to be exactly "US" (uppercase) for edge routing.

5.4 Cache Problems

If product 101 isn't faster than others:

- Check if edge pods are running
- Look at the routing rules
- Try restarting the edge service:

```
kubectl rollout restart deployment edge-deployment
```

5.5 Scaling Issues

If pods aren't scaling automatically:

```
kubectl describe hpa
kubectl top pods
```

Check if the metrics server is working and CPU usage is being measured correctly.

5.6 Quick Fixes

Most problems can be fixed by:

- Waiting a minute or two
- Deleting pods and letting them restart
- Checking the logs with `kubectl logs`
- Making sure all services are running
- Restarting the specific service that's having trouble

If these steps don't help, try cleaning up everything and starting fresh:

```
kubectl delete -f <problem-deployment-file>
kubectl apply -f <problem-deployment-file>
```

6 References

Kubernetes.io. (2024). Minikube - Kubernetes Local Machine Solution. [online] Available at: https://minikube.sigs.k8s.io/docs/ [Accessed 15 Mar. 2024].

Prometheus.io. (2024). Prometheus - Monitoring system & time series database. [online] Available at: https://prometheus.io/docs/introduction/overview/ [Accessed 15 Mar. 2024].

Grafana Labs. (2024). Grafana - The open observability platform. [online] Available at: https://grafana.com/docs/grafana/latest/ [Accessed 15 Mar. 2024].

Istio.io. (2024). Istio / What is Istio?. [online] Available at: https://istio.io/latest/about/service-mesh/ [Accessed 15 Mar. 2024].

Kubernetes.io/HPA. (2024). Horizontal Pod Autoscaling. [online] Available at: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ [Accessed 15 Mar. 2024].