Project Details Are From Page 09

Ingress

Ingress Networking

Ingress networking in Kubernetes refers to the way external access to services within a Kubernetes cluster is managed. It provides a powerful mechanism to define rules that allow external HTTP and HTTPS traffic to reach your services. An Ingress resource defines these rules and allows for more complex routing, such as load balancing, SSL termination, and name-based virtual hosting.

Key Concepts

1. **Ingress Resource**: A set of rules that allow inbound connections to reach the cluster services.
2. **Ingress Controller**: A component that implements the Ingress resource, typically deployed as a pod within the cluster. It interprets the Ingress rules and routes traffic accordingly.
3. **Ingress Rules**: Define the routing of HTTP/HTTPS traffic to various services within the cluster based on host, path, etc.
4. **Backend Services**: Kubernetes services that Ingress resources route traffic to.

1. Start Minikube with Ingress Addon

Ensure that Minikube is started with the Ingress addon enabled.

```
minikube start --addons=ingress
```

2. Verify Ingress Controller

Check that the NGINX Ingress controller pods are running.

```
kubectl get pods -n kube-system | grep nginx
```

You should see the NGINX Ingress controller pods listed.

3. Create Sample Services and Deployments

Create sample deployments and services for demonstration.

```
# frontend-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
```

```yaml
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80


# backend-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: hashicorp/http-echo
```

```yaml
        args:
          - "-text=Hello from backend"
        ports:
        - containerPort: 5678
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5678
```

Apply these YAML files to create the deployments and services.

```
kubectl apply -f frontend-deployment.yaml
kubectl apply -f backend-deployment.yaml
```

4. Create an Ingress Resource

Define an Ingress resource to route traffic to these services.

```yaml
# ingress-resource.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
  - host: myapp.local
    http:
      paths:
      - path: /frontend
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
      - path: /backend
        pathType: Prefix
```

```yaml
        backend:
          service:
            name: backend-service
            port:
              number: 80
```

Apply the Ingress resource.

```
kubectl apply -f ingress-resource.yaml
```

5. Update /etc/hosts

Add the hostname defined in the Ingress resource to your /etc/hosts file pointing to the Minikube IP.

```
sudo nano /etc/hosts
```

Add the following line (replace <minikube-ip> with the actual Minikube IP):

```
<minikube-ip> myapp.local
```

Get the Minikube IP using:

```
minikube ip
```

6. Access the Services

You should now be able to access the services via your browser or curl.

```
curl http://myapp.local/frontend
curl http://myapp.local/backend
```

Advanced Use Cases

1. Load Balancing with Sticky Sessions

Scenario: Distribute traffic across multiple instances of a service while maintaining session affinity.

Example:

Ingress Resource with Annotations:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: sticky-ingress
  annotations:
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "route"
spec:
  rules:
  - host: sticky.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: sticky-service
            port:
              number: 80
```

2. Path Rewriting

Scenario: Route traffic to a different path in the backend service.

Example:

Ingress Resource with Path Rewriting:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: rewrite-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: rewrite.local
    http:
      paths:
      - path: /oldpath/(.*)
        pathType: Prefix
        backend:
          service:
            name: new-service
            port:
```

number: 80

nginx.ingress.kubernetes.io/rewrite-target: /

- **Description**: Replaces the path with the specified rewrite target (/). It effectively removes the matched part of the URL path.
- **Example**: If the original request path is /oldpath/something, it will be rewritten to /something before forwarding to the backend service.

nginx.ingress.kubernetes.io/rewrite-target: /newpath

- **Description**: Replaces the path with the specified rewrite target (/newpath). It substitutes the matched part of the URL path with the specified new path.
- **Example**: If the original request path is /oldpath/something, it will be rewritten to /newpath/something before forwarding to the backend service.

nginx.ingress.kubernetes.io/rewrite-target: /$1

- **Description**: Uses a capture group ($1) from the original path's regular expression match to dynamically construct the rewritten path.
- **Example**: If your Ingress path definition includes regex capturing groups like path: /oldpath/(.*), and rewrite-target: /$1 is specified, requests to /oldpath/something will be rewritten to /something.

Key Features of TLS:

1. **Encryption**: TLS encrypts data to ensure that it remains private and secure during transmission. This prevents unauthorized parties from intercepting and reading the data.
2. **Authentication**: TLS supports server-side and optional client-side authentication using digital certificates. This ensures that the parties involved in the communication are who they claim to be.
3. **Compatibility**: TLS is widely supported and used across various applications and services, including web browsers, email clients, instant messaging, and more.

TLS Handshake Process:

TLS communication begins with a handshake process, where the client and server negotiate parameters for secure communication:

- **ClientHello**: The client sends a message containing the TLS version, supported cipher suites, and a random number.
- **ServerHello**: The server responds with its chosen TLS version, cipher suite, and a random number.
- **Certificate Exchange**: The server sends its digital certificate to prove its identity (if required).

- **Key Exchange**: The client and server agree on a shared secret key to be used for symmetric encryption during the session.
- **Finished**: Both parties exchange finished messages to confirm that the handshake was successful and communication can proceed securely.

Usage:

TLS is commonly used to secure HTTP connections (HTTPS), ensuring that sensitive information such as login credentials, payment details, and personal data transmitted over the internet remains confidential and integral.

Create a Secret for TLS Certificate

```
kubectl create secret tls tls-secret --cert=path/to/tls.crt --key=path/to/tls.key
```

Ingress Resource:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  tls:
  - hosts:
    - myapp.local
    secretName: my-tls-secret
  rules:
  - host: myapp.local
    http:
      paths:
      - path: /frontend
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
      - path: /backend
```

```
pathType: Prefix
backend:
  service:
    name: backend-service
    port:
              number: 80
```

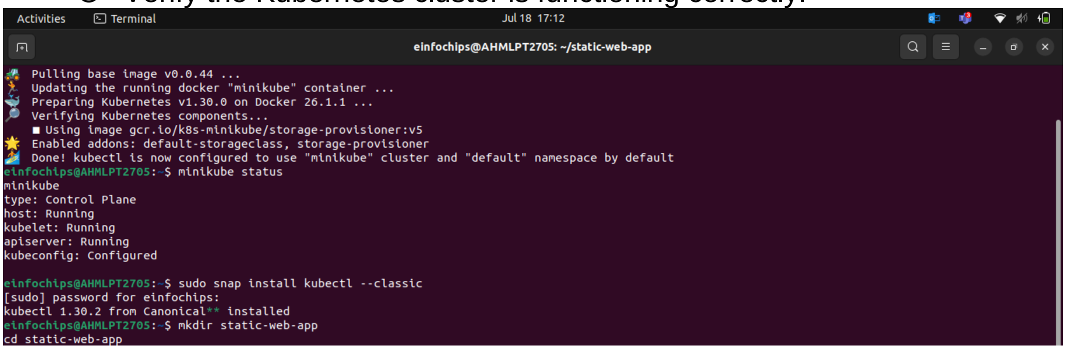
Project Overview

Participants are required to deploy a simple static web application on a Kubernetes cluster using Minikube, set up advanced ingress networking with URL rewriting and sticky sessions, and configure horizontal pod autoscaling to manage traffic efficiently. The project will be divided into stages, with each stage focusing on specific aspects of Kubernetes ingress, URL rewriting, sticky sessions, and autoscaling.

Requirements and Deliverables

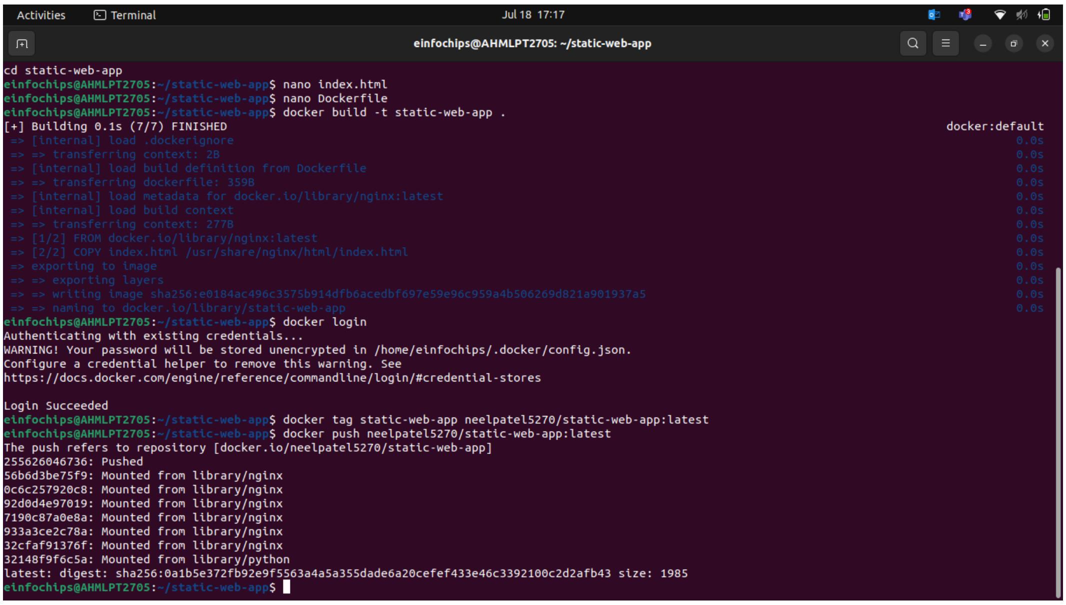
Stage 1: Setting Up the Kubernetes Cluster and Static Web App

1. **Set Up Minikube:**
    - Ensure Minikube is installed and running on the local Ubuntu machine.
    - Verify the Kubernetes cluster is functioning correctly.



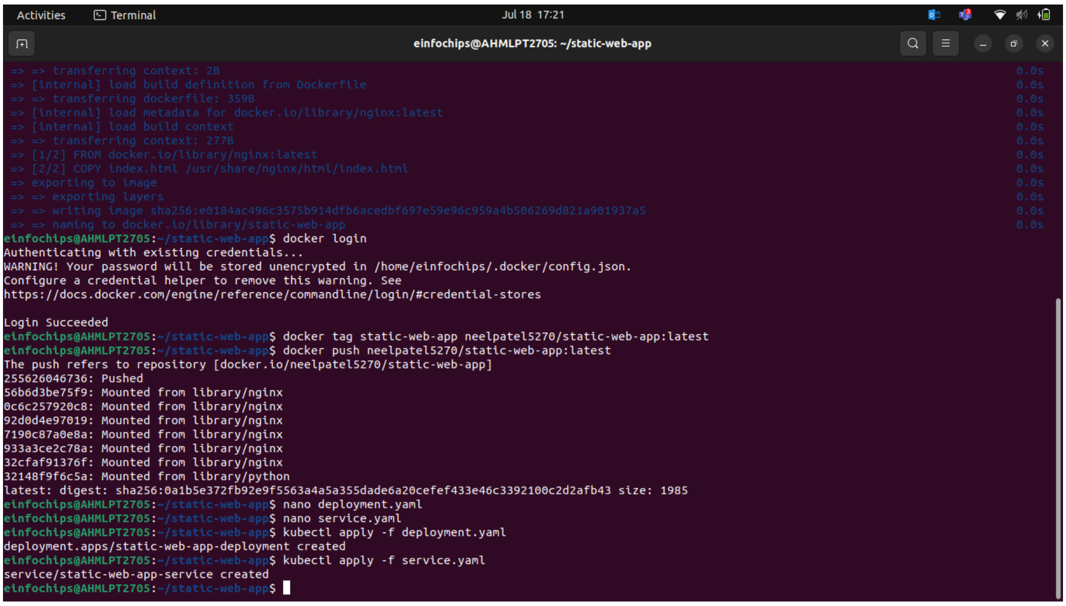
- Start Minikube (`minikube start`)
- Create a directory named `static-web-api` in the current working directory (`mkdir static-web-api`)

2. **Deploy Static Web App:**
    - Create a Dockerfile for a simple static web application (e.g., an HTML page served by Nginx).
    - Build a Docker image for the static web application.
    - Push the Docker image to Docker Hub or a local registry.



Create an `index.html` file in the same directory
Build the Docker Image
docker login
Then push the image

3. **Kubernetes Deployment:**
   - Write a Kubernetes deployment manifest to deploy the static web application.
   - Write a Kubernetes service manifest to expose the static web application within the cluster.
   - Apply the deployment and service manifests to the Kubernetes cluster.



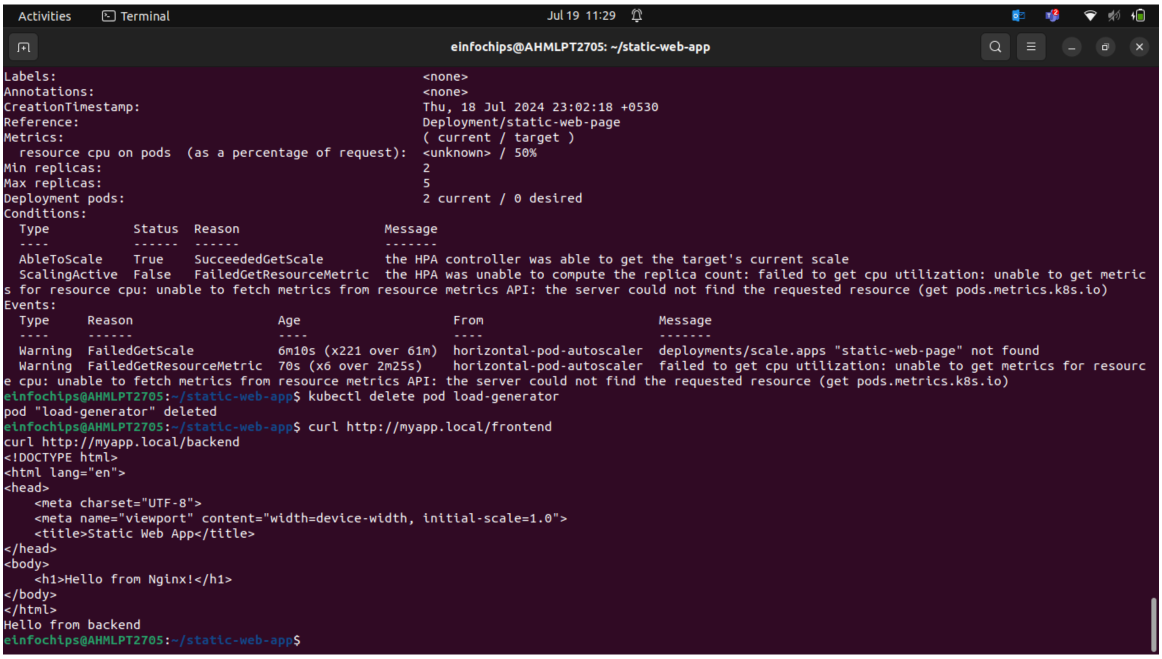
Deliverables:

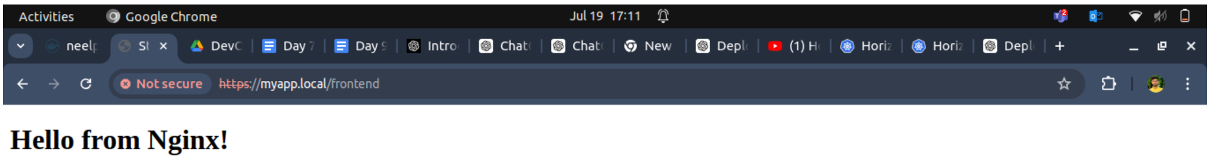
- Dockerfile for the static web app
- Docker image URL
- Kubernetes deployment and service YAML files

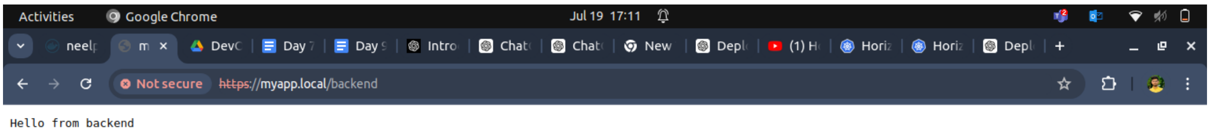
- Create a file `deployment.yaml`
- `Create a file service.yaml`
- `Apply the Deployment and Service Manifests`
- `Minikube IP address`

Stage 2: Configuring Ingress Networking

4. **Install and Configure Ingress Controller:**
   - Install an ingress controller (e.g., Nginx Ingress Controller) in the Minikube cluster.
   - Verify the ingress controller is running and accessible.
5. **Create Ingress Resource:**
   - Write an ingress resource manifest to route external traffic to the static web application.
   - Configure advanced ingress rules for path-based routing and host-based routing (use at least two different hostnames and paths).
   - Implement TLS termination for secure connections.
   - Configure URL rewriting in the ingress resource to modify incoming URLs before they reach the backend services.
   - Enable sticky sessions to ensure that requests from the same client are directed to the same backend pod.







Deliverables:

- Ingress controller installation commands/scripts
- Ingress resource YAML file with advanced routing, TLS configuration, URL rewriting, and sticky sessions
    - minikube addons enable ingress

- Create a Kubernetes Secret to store the TLS certificate
- kubectl create secret tls tls-secret --cert=tls.crt –key=tls.key
- nano ingress-rewriting.yaml
- kubectl apply f ingressrewring.yaml
- Create a `ingress.yaml`
- `Apply the Ingress Manifest`
- `Create a deployment.yaml`
- `Create a service.yaml`

Stage 3: Implementing Horizontal Pod Autoscaling

6. **Configure Horizontal Pod Autoscaler:**
    i. Write a horizontal pod autoscaler (HPA) manifest to automatically scale the static web application pods based on CPU utilization.
    - Set thresholds for minimum and maximum pod replicas.

```
</body>
</html>
Hello from backend
einfochips@AHMLPT2705:~/static-web-app$ kubectl autoscale deployment frontend --cpu-percent=50 --min=2 --max=4
horizontalpodautoscaler.autoscaling/frontend autoscaled
einfochips@AHMLPT2705:~/static-web-app$ kubectl autoscale deployment backend --cpu-percent=50 --min=2 --max=4
horizontalpodautoscaler.autoscaling/backend autoscaled
einfochips@AHMLPT2705:~/static-web-app$
```

    -

7. **Stress Testing:**
    - Perform stress testing to simulate traffic and validate the HPA configuration.
    - Monitor the scaling behavior and ensure the application scales up and down based on the load.

```
 1862  rm hpa.yaml.save
 1863  rm load-generator.yaml
 1864  kubectl delete service static-web-app-service
 1865  ls
 1866  nano deployment.yaml
 1867  nano service.yaml
 1868  nano deployment.yaml
 1869  nano hpa.yaml
 1870  kubectl apply -f deployment.yaml
 1871  kubectl apply -f hpa.yaml
 1872  nano hpa.yaml
 1873  kubectl apply -f hpa.yaml
 1874  nano hpa.yaml
 1875  kubectl apply -f service.yaml
 1876  minikube service my-app-service --url
 1877  kubectl run -i --tty --rm load-generator --image=busybox --restart=Never -- /bin/sh
 1878  kubectl run -i --tty --rm load-generator-new --image=busybox --restart=Never -- /bin/sh
 1879  history
einfochips@AHMLPT2705:~/static-web-app$
```

- Horizontal Pod Autoscaler (HPA) for your static web application based on CPU utilization, you'll need to create an HPA manifest. This manifest will define the rules for scaling your application pods automatically.
- apply the HPA manifest
- **`kubectl get hpa`**
- This command will show you the current status of the HPA, including the current number of replicas and the target metrics.
- **`kubectl get pods -w`**

This command will continuously display the status of the pods in your deployment, allowing you to see how the number of pods changes in response to the load.

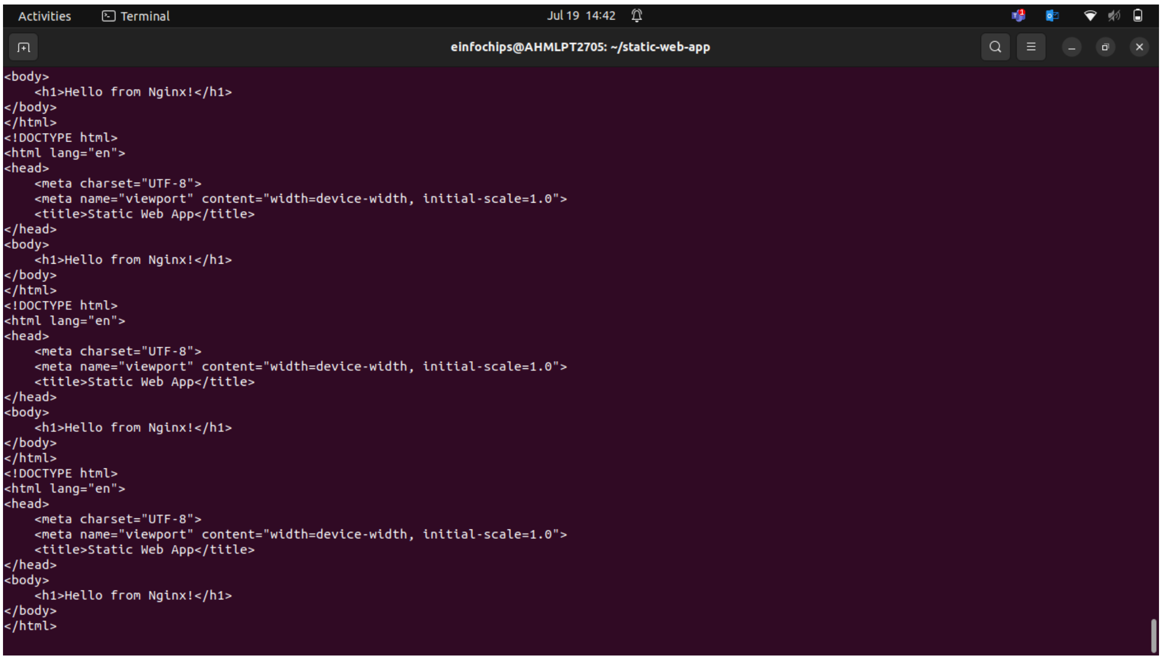
- **Scaling Up:** As the load increases, the HPA should trigger the creation of additional pods to handle the increased traffic.
- **Scaling Down**: Once the load decreases, the HPA should reduce the number of pods back to the minimum defined in the manifest.
- kubectl get hpa

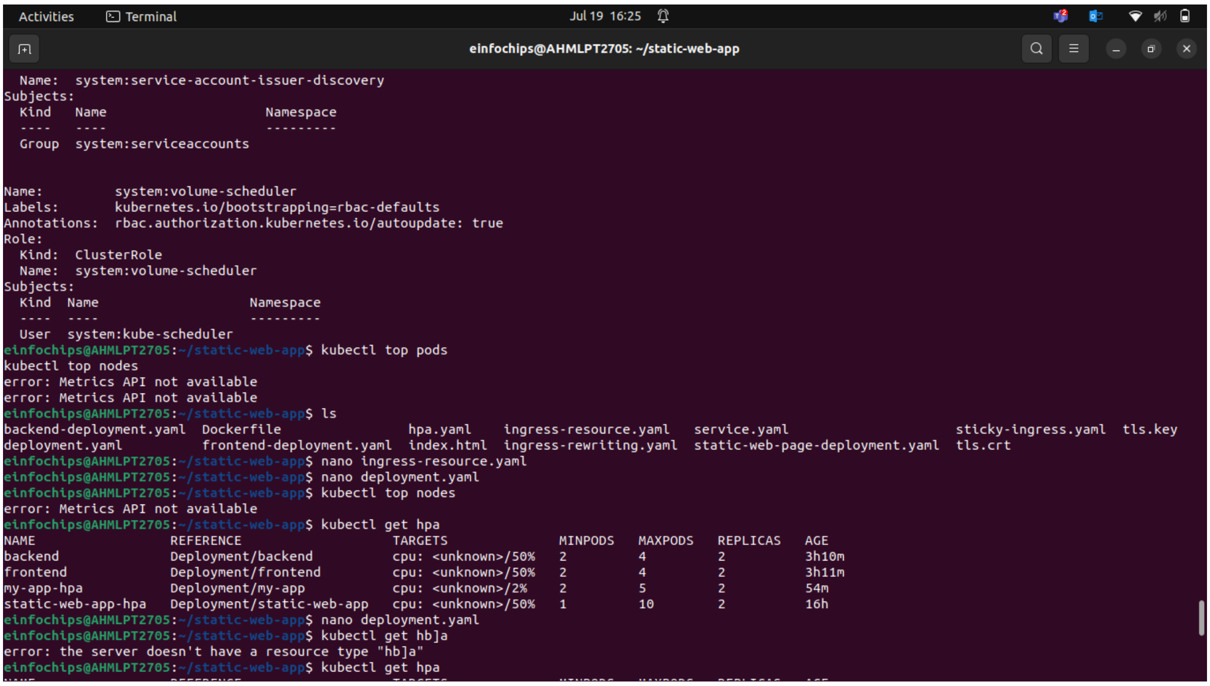
  This output indicates that the HPA is targeting 50% CPU utilization, and currently, the CPU utilization is at 70%, so it has scaled up to 5 replicas.

  This output shows the scaling behavior of the pods, with new pods being created and running as the load increases.

Deliverables:

- Horizontal pod autoscaler YAML file
- Documentation or screenshots of the stress testing process and scaling behavior





Stage 4: Final Validation and Cleanup

8. **Final Validation:**
    - Validate the ingress networking, URL rewriting, and sticky sessions configurations by accessing the web application through different hostnames and paths.
    - Verify the application's availability and performance during different load conditions.
9. **Cleanup:**

- Provide commands or scripts to clean up the Kubernetes resources created during the project (deployments, services, ingress, HPA).

Deliverables:

- Final validation report documenting the testing process and results
- Cleanup commands/scripts