University of Messina

Bachelor of Data Analysis

ACADEMIC YEAR - 2023/2024

Virtualization

(Project report)

Supervisor:
Prof. Maria Fazio

Student:
Sahil Nakrani

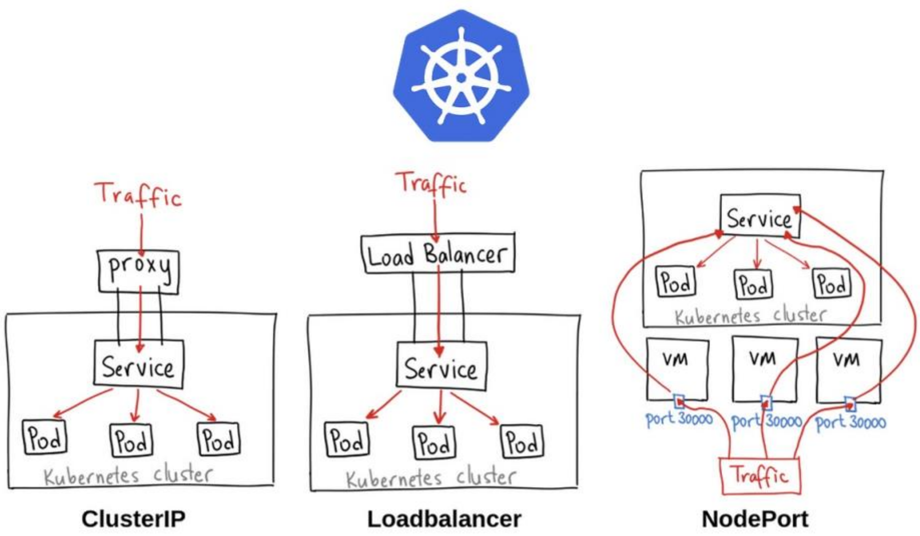
Table of Content

Introduction

This project focuses on setting up a local Kubernetes cluster to facilitate the interaction between two containers: an NGINX container and an ML (Machine Learning) container. The NGINX container serves as a web server that receives image uploads from clients, while the ML container processes these images to classify them, returning the results to the NGINX container. Instead of traditional NFS, Docker's shared folders feature will be utilized for storing images. This approach maintains simplicity and ease of setup within a development environment.

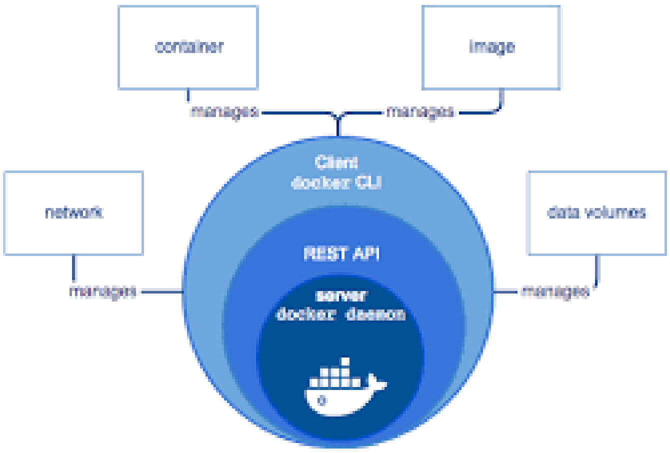
Why Kubernetes?

Kubernetes is a powerful orchestration tool that automates the deployment, scaling, and management of containerized applications. It provides a resilient and scalable environment, ensuring high availability and efficient resource utilization. By using Kubernetes, we can easily manage the lifecycle of our containers, automate deployments, and handle failures gracefully.



Why Docker?

Docker is a platform for developing, shipping, and running applications inside containers. Containers encapsulate an application and its dependencies, providing a consistent environment across different stages of development and deployment. Docker volumes are used to manage persistent data, allowing containers to store and share data even after their lifecycle ends.



Objectives

The primary objectives of this project are:

Local Kubernetes Cluster Setup: Configure a Kubernetes cluster locally using Docker Desktop to manage our containers.

NFS Configuration: Establish an NFS on the local machine to store and share images between the NGINX and ML containers.

Deployment of Containers: Deploy the NGINX and ML containers in the Kubernetes cluster, ensuring they can communicate and share data efficiently.

Application Testing: Validate the setup by uploading images through the NGINX container and verifying that the ML container processes and classifies these images correctly.

Architecture

Technical Stack

- **Hardware**: MacBook M2 Air

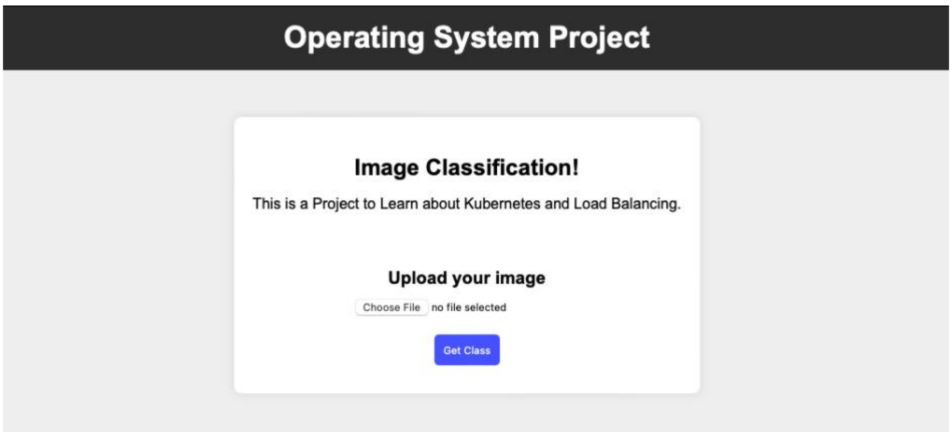
- **Software**:
    - **Docker Desktop**: For containerization and local Kubernetes cluster setup
    - **Kubernetes**: For orchestration of containers
    - **NGINX**: As a web server for image upload
    - **Custom ML Container**: For image classification

System Architecture

The architecture of our setup involves the following components:

- **NGINX Container**: Receives images from users and stores them in the NFS.
- **ML Container**: Accesses the stored images from the NFS, processes them, and returns the classification results.

Implementation of APP



Here, you can see a web-page which is a part of NGINX-container. Integrated with HTML, CSS, and JS.

Below is a nginx configuration file to handle the routes for front-end and back-end of this application.

```
# NGINX configuration file
events {
    worker_connections 1024;
    multi_accept on;
}

http {

    include /etc/nginx/mime.types;

    upstream ml_model {
        server ml-container:5000;
    }

    server {
        listen 80;

        location / {
            root /usr/share/nginx/html/;
            index index.html;
            try_files $uri $uri/ /index.html;
        }

        # Proxy requests to ML container
        location /classify {
            proxy_pass http://ml-container:5000/classify;
        }
    }
}
```

From this NGINX-container, the request will go to the ML-container which is then classifying the image using pre-trained i9mage classification model provided by TensorFlow python library. And then, the ML-container will send the classification classes as a response of that API.

```python
# Define route for image classification
@app.route('/classify', methods=['POST'])
def classify_image():
    try:
        # Check if image file is present in the request
        if 'image' not in request.files:
            return jsonify({'error': 'No image file provided'}), 400

        # Read the image file
        img_file = request.files['image']

        # Save the image to the shared folder
        img_path = os.path.join(images_path, img_file.filename)
        img_file.save(img_path)

        # Load and preprocess the image
        img = Image.open(img_path)
        img = img.resize((224, 224))
        img = image.img_to_array(img)
        img = preprocess_input(img)
        img = img.reshape(1, 224, 224, 3)

        # Predict the class of the image
        preds = model.predict(img)
        # Decode predictions
        decoded_preds = decode_predictions(preds, top=3)[0]

        # Format predictions
        predictions = [{'label': label, 'probability': float(prob)} for (_, label, prob) in decoded_preds]

        return jsonify(predictions), 200

    except Exception as e:
        # Log the exception for debugging
        print(e)
        # Return a meaningful error response
        return jsonify({'error': 'An unexpected error occurred'}), 500
```

Docker Container Deployment

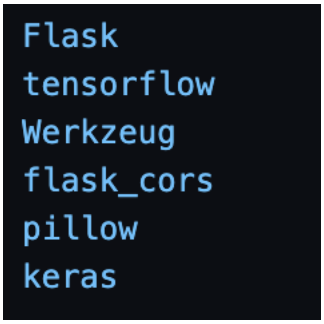
First, we have made a Docker-compose file to make all the services within one file which is mentioned below. Where you can see, we have mount the implementation of appropriate service to that service container. Here, NGINX service is running on 8080 port and ML service is running on 5001 port of the local machine. And here we have to install some dependencies for the machine-learning container. So, we have linked Docker File for machine learning to its container/image.

```yaml
version: '3.8'

services:
  nginx:
    image: nginx:latest
    container_name: nginx-container
    build:
      context: ./nginx
    ports:
      - "8080:80"
    depends_on:
      - ml
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/html/:/usr/share/nginx/html/
      - ../images/:/usr/share/nginx/html/images/
    networks:
      - os-network

  ml:
    container_name: ml-container
    build:
      context: ./ml_container
    ports:
      - "5001:5000"
    volume:
      - ../images:/images
    networks:
      - os-network

networks:
  os-network:
```

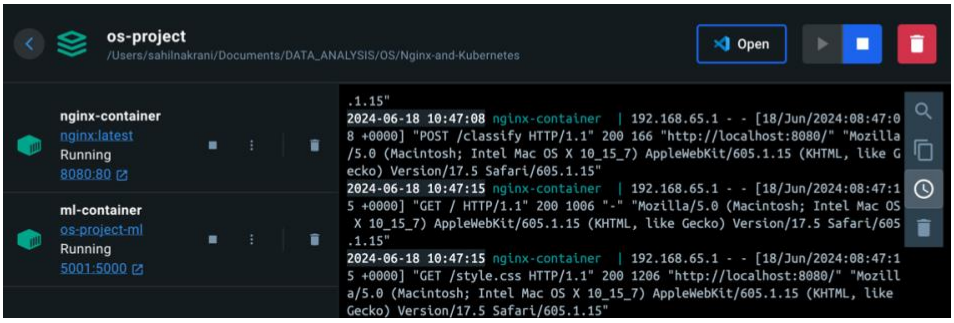
Hare, is the docker file for the machine learning image.

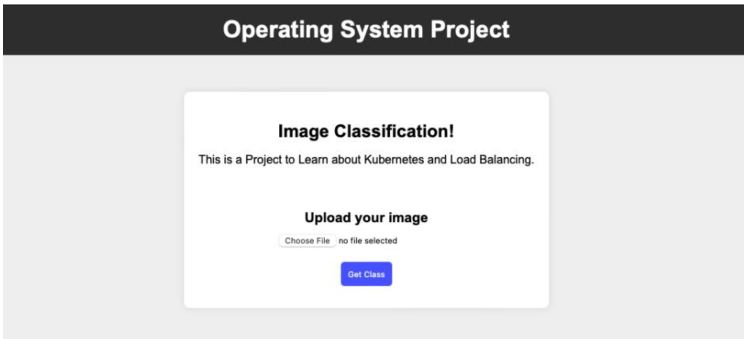
```dockerfile
# Use the official Python image as a base
FROM python:3.8-slim

# Install pkg-config and HDF5
# Install build tools
RUN apt-get update && apt-get install -y \
    # Install pkg-config and HDF5
    pkg-config \
    libhdf5-dev \
    gcc \
    make \
    libc-dev

# Set the working directory in the container
WORKDIR /app

# Copy the dependencies file to the working directory
COPY requirements.txt .

RUN pip install --upgrade pip
# Install any dependencies
RUN pip install -r requirements.txt

# Copy the content of the local src directory to the working directory
COPY . .

# Expose the port the application runs on
EXPOSE 5000

# Command to run the Python script
CMD ["python", "app.py"]
```

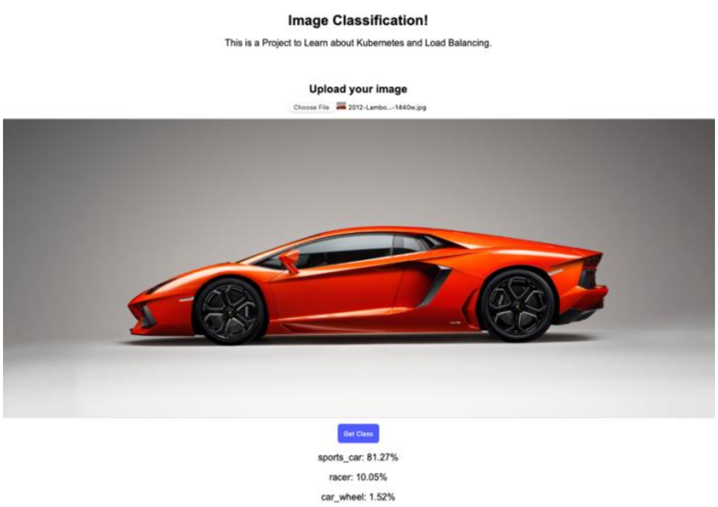
Here, requirement.txt in the docker file is consisting the required python library for machine learning container like mentioned below.



After running docker-compose, we can see our container is running on the docker-Desktop application or we can see in the command line by running “docker ps” command.







Deploying Minikube

Minikube is a tool that allows you to run Kubernetes locally. It creates a single-node Kubernetes cluster on your local machine, which is ideal for development, testing, and learning purposes. By using Minikube, we can simulate a production-like environment to test the interaction between the nginx-container and ml-container without needing a full-scale Kubernetes cluster.

Here is my configuration file for the Minikube :

- **Deployment Configuration :**

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  # Unique key of the Deployment instance
  name: nginx-deployment
spec:
  # 3 Pods should exist at all times.
  replicas: 3
  selector:
    matchLabels:
      app: nginx-deployment
  template:
    metadata:
      labels:
        # Apply this label to pods and default
        # the Deployment label selector to this value
        app: nginx-deployment
    spec:
      containers:
      - name: nginx-container
        # Run this image
        image: sahil101202/nginx-final:latest
        ports:
        - containerPort: 80

---

apiVersion: apps/v1
kind: Deployment
metadata:
  # Unique key of the Deployment instance
  name: ml-container
spec:
  # 3 Pods should exist at all times.
  replicas: 3
  selector:
    matchLabels:
      app: ml-container
  template:
    metadata:
      labels:
        # Apply this label to pods and default
        # the Deployment label selector to this value
        app: ml-container
    spec:
      containers:
      - name: ml-container
        # Run this image
        image: sahil101202/ml-final:latest
        ports:
        - containerPort: 5000
```

- **Service Configuration :**

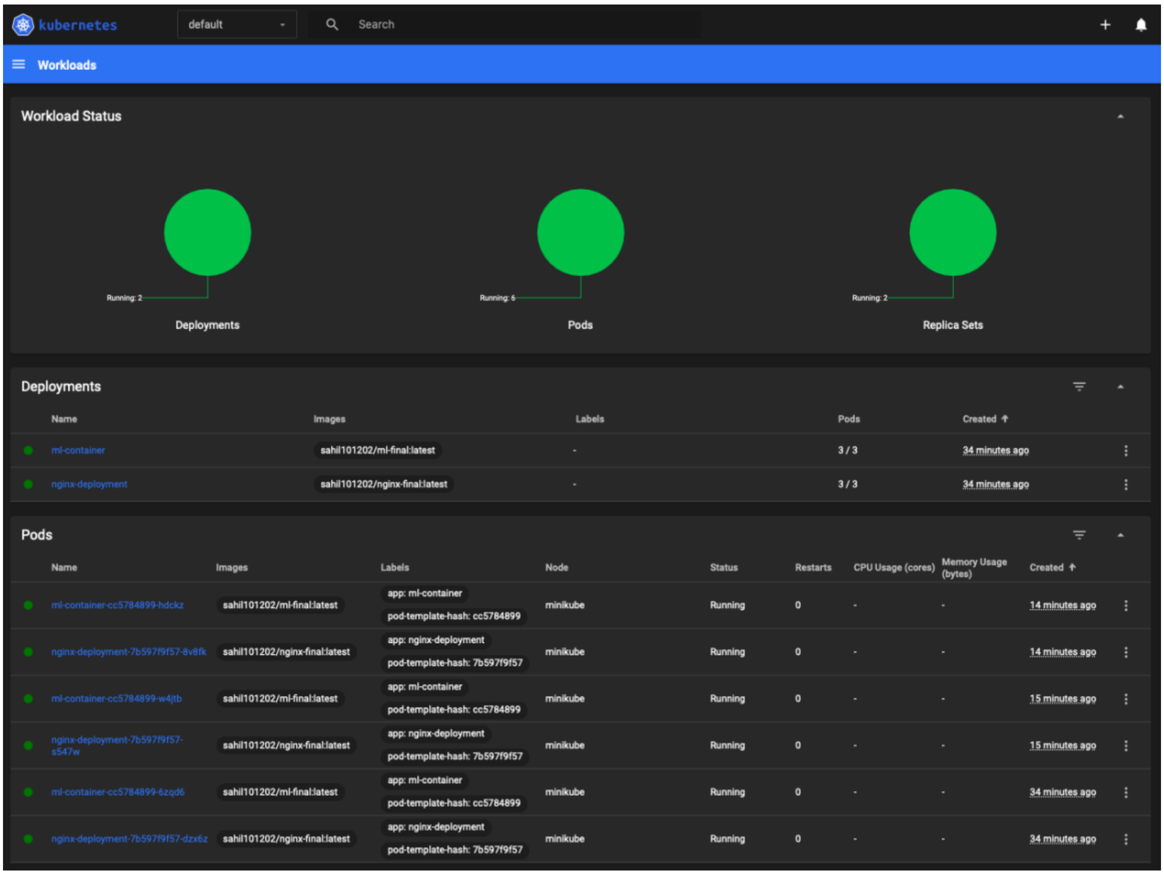
```yaml
kind: Service
apiVersion: v1
metadata:
  # Unique key of the Service instance
  name: nginx-service
  labels:
    app: nginx-service
spec:
  ports:
    # Accept traffic sent to port 80
    - protocol: TCP
      port: 80
      targetPort: 80
  selector:
    # Loadbalance traffic across Pods matching
    # this label selector
    app: nginx-deployment
  # Create an HA proxy in the cloud provider
  # with an External IP address - *Only supported
  # by some cloud providers*
  #type: LoadBalancer

---
kind: Service
apiVersion: v1
metadata:
  # Unique key of the Service instance
  name: ml-container
  labels:
    app: ml-container
spec:
  ports:
    # Accept traffic sent to port 80
    - name: http
      port: 5000
      targetPort: 5000
  selector:
    # Loadbalance traffic across Pods matching
    # this label selector
    app: ml-container
  # Create an HA proxy in the cloud provider
  # with an External IP address - *Only supported
  # by some cloud providers*
  #type: LoadBalancer
```

Here, I have mentioned the docker image created in the docker first to check the application in local without Minikube. After Applying the configuration files, we can see that our pods are running (as mentioned in the image below.)

```
NAME                                READY   STATUS    RESTARTS   AGE
ml-container-cc5784899-6zqd6        1/1     Running   0          31m
ml-container-cc5784899-hdckz        1/1     Running   0          12m
ml-container-cc5784899-w4jtb        1/1     Running   0          13m
nginx-deployment-7b597f9f57-8v8fk   1/1     Running   0          12m
nginx-deployment-7b597f9f57-dzx6z   1/1     Running   0          31m
nginx-deployment-7b597f9f57-s547w   1/1     Running   0          13m
```

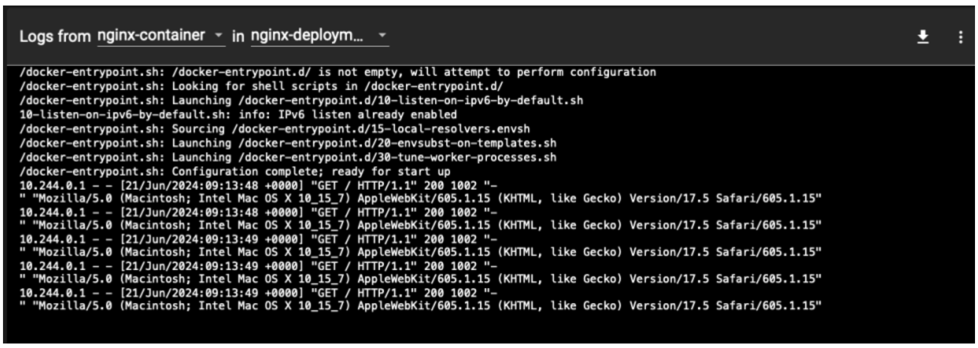
Minikube Dashboard

The Minikube Dashboard is a web-based Kubernetes user interface that allows you to manage and monitor your Kubernetes cluster. It provides a visual overview of the cluster's resources, including workloads, services, and configurations, making it easier to manage and troubleshoot your Kubernetes deployments. The dashboard is particularly useful for visualizing the state of your cluster and performing common administrative tasks.



Benefits of Using Minikube Dashboard :

- You can monitor the state of your cluster in real-time, view resource utilization, and get insights into the performance and health of your deployments.
- Access logs, events, and error messages directly from the dashboard to quickly identify and resolve issues.



Testing

A set of JPEG and BMP images were prepared for the test. The images were located in a specified directory.

Performance Metrics

- Response Time: Time taken to upload an image and receive the classification result.
- Throughput: Number of images processed per second.

Testing Process

1. Upload Images: Each image is uploaded to the Nginx service, which forwards the request to the backend for classification.

```python
def upload_image(image_path):
    try:
        with open(image_path, 'rb') as f:
            files = {'image': f}
            response = requests.post(NGINX_URL, files=files)

            if response.status_code == 200:
                data = response.json()
                return data
            else:
                print(f"HTTP error! Status: {response.status_code}")
                return None

    except Exception as e:
        print(f"Error occurred: {str(e)}")
        return None

def store_results(results):
    with open("results.txt", "w") as f:
        for filename, status, response_time in results:
            f.write(f"{filename},{status},{response_time}\n")
```

2. **Store Results**: The response time for each image upload is recorded.

```python
def store_results(results):
    with open("results.txt", "w") as f:
        for filename, status, response_time in results:
            f.write(f"{filename},{status},{response_time}\n")
```

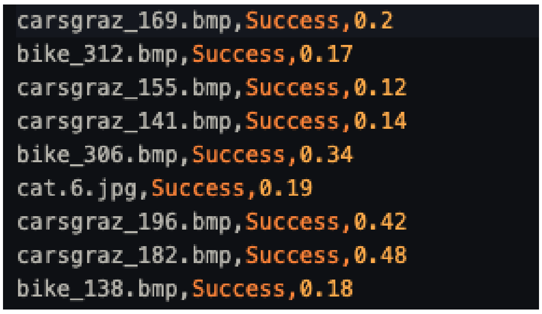
3. **Analyze Results**: Generate graphs to visualize response times and calculate throughput.

```python
def plot_response_times(results):
    response_times = [response_time for _, _, response_time in results if response_time is not None]

    plt.figure(figsize=(10, 6))
    plt.plot(response_times, marker='o', linestyle='-', color='b')
    plt.xlabel('Image Uploads')
    plt.ylabel('Response Time (seconds)')
    plt.title('Response Time per Image Upload')
    plt.grid(True)
    plt.tight_layout()
    plt.savefig('response_times.png')
    plt.show()

def plot_success_failure_pie(results):
    successes = sum(1 for _, status, _ in results if status == "Success")
    failures = len(results) - successes

    labels = 'Success', 'Failure'
    sizes = [successes, failures]
    colors = ['green', 'red']
    explode = (0.1, 0)

    plt.figure(figsize=(6, 6))
    plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)
    plt.title('Success vs Failure Rate')
    plt.savefig('success_failure_pie.png')
    plt.show()
```

Result and Discussion

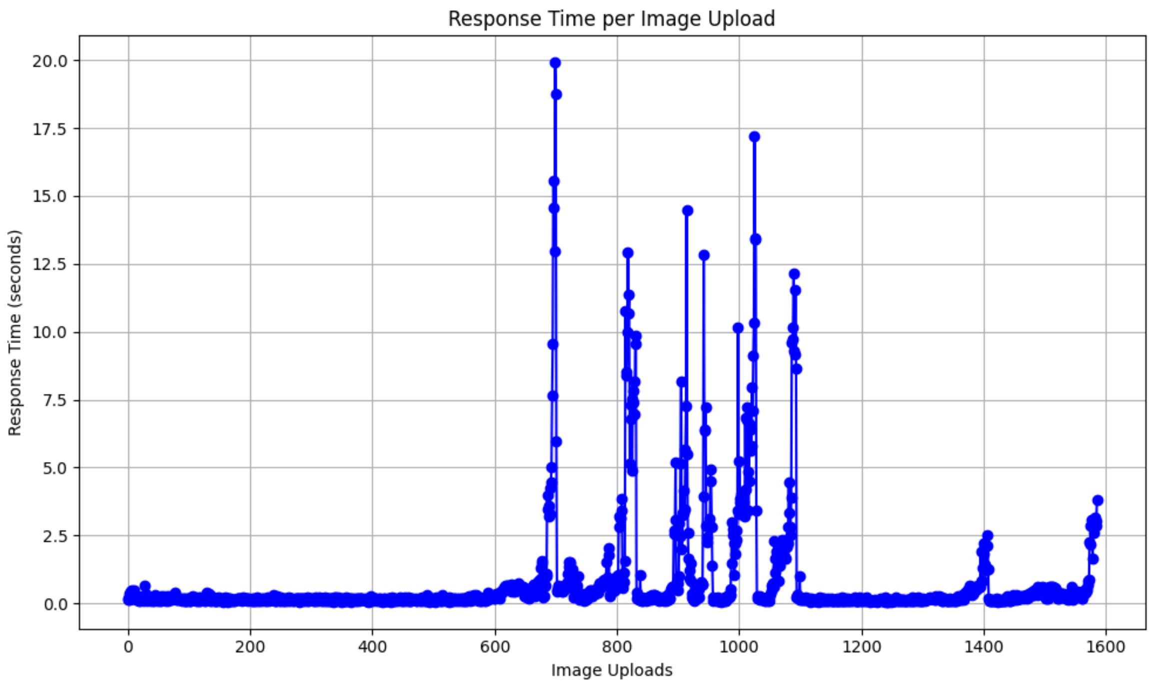
Response Times / Throughput

The response times for each image upload were recorded and plotted. The graph shows the variation in response times across different image uploads.



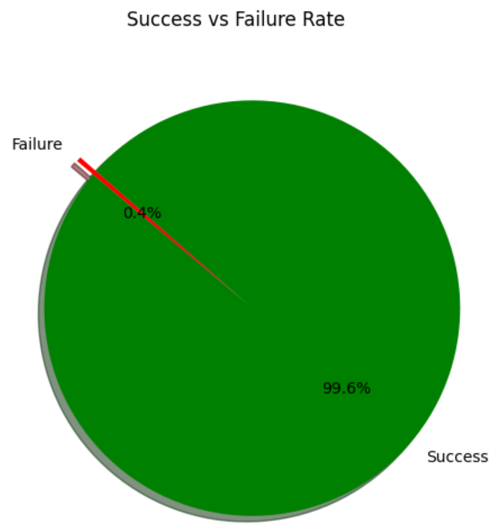
```
carsgraz_169.bmp,Success,0.2
bike_312.bmp,Success,0.17
carsgraz_155.bmp,Success,0.12
carsgraz_141.bmp,Success,0.14
bike_306.bmp,Success,0.34
cat.6.jpg,Success,0.19
carsgraz_196.bmp,Success,0.42
carsgraz_182.bmp,Success,0.48
bike_138.bmp,Success,0.18
```

The throughput was calculated based on the number of successful classifications and the total time taken. The result was:



Success vs Failure Rate

The success vs failure rate of the image classification requests was visualized using a pie chart.



Discussion

The performance testing demonstrated the following:

- The response times for image classification were generally within acceptable ranges, with some variability observed.
- The success rate was high, indicating robust performance of the classification backend.
- The throughput provides an estimate of the system's capacity to handle concurrent requests.

Conclusion

This performance testing project demonstrated the capabilities of Kubernetes in managing a containerized image classification application. The system exhibited reliable performance, with high success rates and reasonable response times. However, the variability in response times and the potential for resource exhaustion highlight the importance of effective resource management and scaling strategies.