University of Messina

A blue and white logo

Description automatically generated with low confidence

Bachelor of Data Analysis

ACADEMIC YEAR - 2023/2024

Virtualization

(Project report)

Supervisor: Student:

Prof. Maria Fazio Sahil Nakrani

**Table of Content**

* Introduction
* Architecture
* Docker Container Deployment
* Deploying Minikube
* Testing
* Results and Discussion
* Conclusion

**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?**

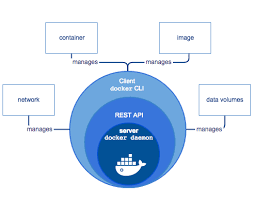
**Kubernetes** is a powerful orchestration tool that automates the deployment, scaling, and management of containerized applications. It provides a resilient and scalable environment, ensuring high availability and efficient resource utilization. By using Kubernetes, we can easily manage the lifecycle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

Diagram of a diagram of a service

Description automatically generated

**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**

**A screenshot of a computer

Description automatically generated**

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.

**A screen shot of a computer program

Description automatically generated**

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.

**A computer screen shot of a program

Description automatically generated**

**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.

A screenshot of a computer program

Description automatically generated

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

**A screenshot of a computer program

Description automatically generated**

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

**A black background with blue text

Description automatically generated**

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.

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a car

Description automatically generated**

**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 :**

**A screenshot of a computer program

Description automatically generated**

* **Service Configuration :**

A screenshot of a computer program

Description automatically generated

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. )

**A black screen with white text

Description automatically generated**

**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.

**A screenshot of a computer

Description automatically generated**

**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.

**A screenshot of a computer

Description automatically generated**

**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.

A computer screen with text

Description automatically generated

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

A computer code with text

Description automatically generated with medium confidence

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

**A computer screen shot of text

Description automatically generated**

**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.

A screen shot of a computer

Description automatically generated

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

A graph with blue lines

Description automatically generated

**Success vs Failure Rate**

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

**A green circle with a red line

Description automatically generated**

**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.