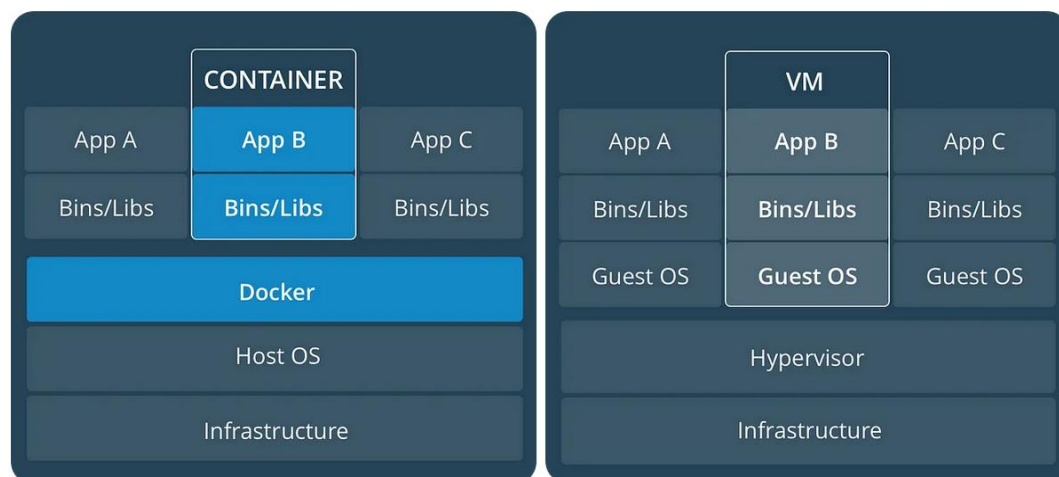


Docker:

- Docker is a set of platform as a service products that use OS-level virtualisation to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating-system kernel and are thus more lightweight than virtual machines.
- For Machine Learning (ML) workflows, Docker enables packaging of the model, dependencies, and inference code into a reproducible and portable unit.

Core Concepts in Docker:

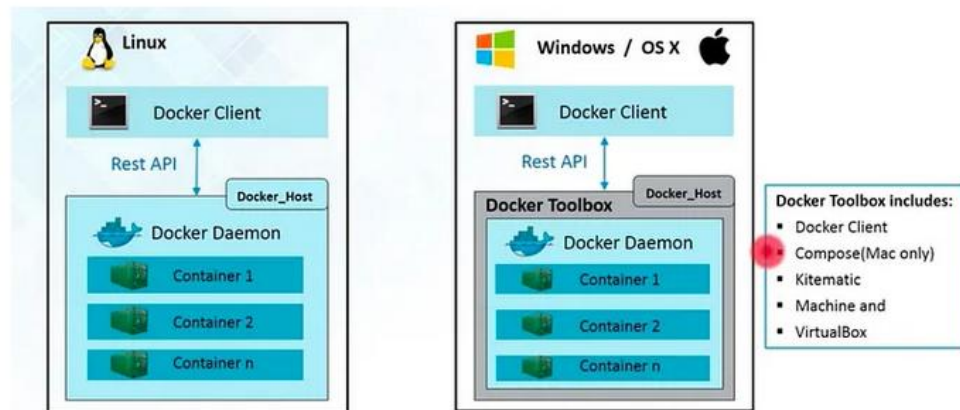
1. Images:
A snapshot of an application and its environment.
Containers:
A running instance of an image.
2. Containers and virtual machines:



- A container runs *natively* on Linux and shares the kernel of the host machine with other containers.
 - A virtual machine (VM) runs a full-blown “guest” operating system with *virtual* access to host resources through a hypervisor.
3. How docker works:
Docker has a **docker engine**, which is the heart of Docker system. It is a client-server application. It has three main components:
 - A server which is a type of long-running process called a daemon process.
 - A client which is Docker CLI(Command Line Interface), and
 - A REST API which is used to communicate between the client(Docker CLI) and the server (Docker Daemon).

In Linux, Docker host runs docker daemon and docker client can be accessed from the terminal.

In Windows/OS X, there is an additional tool called Docker toolbox. This toolbox installs the docker environment on Win/OS system. This toolbox installs the following: Docker Client, Compose, Kitematic, Machine, and Virtual Box.



ML workflow with Docker:

1. Create a Python application (e.g., FastAPI/Flask) to load the ML model and serve predictions.
2. Define environment setup in a Dockerfile.
3. Build the Docker image.
4. Run the Docker container.
5. Test the API via browser or Postman.

Sample Docker file:

```
FROM nvidia/cuda:11.8.0-cudnn8-runtime-ubuntu22.04
WORKDIR /app
COPY . .
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip3 install -r requirements.txt
EXPOSE 5000
CMD ["python3", "app.py"]
```

Explanation:

- **FROM nvidia/cuda...** - This allows GPU acceleration inside the container, essential for deploying GPU-dependent ML models like CNNs, Transformers, etc.
- **COPY** and **WORKDIR** ensure all files are in place and accessible from **/app**.
- **RUN** installs system and Python dependencies.
- **CMD** specifies the entry point for the application (typically the inference API).

Example:

If the directory contains these files:

- Dockerfile:


```
FROM nvidia/cuda:11.8.0-cudnn8-runtime-ubuntu22.04
WORKDIR /app
COPY . .
RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip3 install -r requirements.txt
EXPOSE 5000
CMD ["python3", "app.py"]
```
- app.py


```
from flask import Flask
app = Flask(__name__)
@app.route("/")

def home():
    return "Hello, world!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```
- requirements.txt:


```
Flask
```

Build and run commands:

```
docker build -t myapp .
```

```
docker run -p 5000:5000 myapp
```

Output:

```
○ (base) avinash@fedora:~/internship-deployment/imp/diff$ docker run -p 5000:5000 myapp

=====
== CUDA ==
=====

CUDA Version 11.8.0

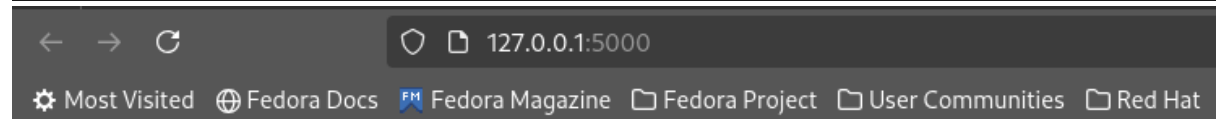
Container image Copyright (c) 2016-2023, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

This container image and its contents are governed by the NVIDIA Deep Learning Container License.
By pulling and using the container, you accept the terms and conditions of this license:
https://developer.nvidia.com/ngc/nvidia-deep-learning-container-license

A copy of this license is made available in this container at /NGC-DL-CONTAINER-LICENSE for your convenience.

WARNING: The NVIDIA Driver was not detected. GPU functionality will not be available.
Use the NVIDIA Container Toolkit to start this container with GPU support; see
https://docs.nvidia.com/datacenter/cloud-native/ .

* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
Press CTRL+C to quit
172.17.0.1 - - [21/Apr/2025 13:51:24] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [21/Apr/2025 13:51:24] "GET /favicon.ico HTTP/1.1" 404 -
[]
```



The screenshot shows a web browser window with the address bar set to 127.0.0.1:5000. The browser's bookmark bar contains links to 'Most Visited', 'Fedora Docs', 'Fedora Magazine', 'Fedora Project', 'User Communities', and 'Red Hat'. The main content area of the browser displays 'Hello, World!'.

Hello, World!

Example 2:
Docker File:

FROM nvidia/cuda:11.8.0-cudnn8-runtime-ubuntu20.04

WORKDIR /app

RUN apt-get update && apt-get install -y \

python3.8 \

python3-pip \

&& rm -rf /var/lib/apt/lists/*

COPY requirements.txt .

RUN pip3 install --no-cache-dir -r requirements.txt

COPY hello_inference.py .

EXPOSE 8000

CMD ["uvicorn", "hello_inference:app", "--host", "0.0.0.0", "--port", "8000"]

hello_inference.py:

from fastapi import FastAPI

import torch

```
import numpy as np
```

```
app = FastAPI()
```

```
@app.get("/")
```

```
async def predict():
```

```
    model = torch.nn.Linear(10, 1)
```

```
    input_data = torch.tensor(np.random.rand(1, 10), dtype=torch.float32)
```

```
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
    model = model.to(device)
```

```
    input_data = input_data.to(device)
```

```
    output = model(input_data)
```

```
    return {"device": str(device), "output": output.item()}
```

requirements.txt:

```
torch>=1.13.0
```

```
numpy>=1.21.0
```

```
fastapi>=0.70.0
```

```
uvicorn[standard]>=0.18.0
```

Output(During Building):

```
(base) avinash@fedora:~/internship-deployment/imp$ docker build -t hello_inference .
[+] Building 5.9s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 691B
=> [internal] load metadata for docker.io/nvidia/cuda:11.8.0-cudnn8-runtime-ubuntu20.04
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/6] FROM docker.io/nvidia/cuda:11.8.0-cudnn8-runtime-ubuntu20.04@sha256:c2336dad71ae5b5ce490a55cc0100d876287d28f429a5d2840c8a3a8e86fef0
=> [internal] load build context
=> => transferring context: 273B
=> CACHED [2/6] WORKDIR /app
=> CACHED [3/6] RUN apt-get update && apt-get install -y python3.8 python3-pip && rm -rf /var/lib/apt/lists/*
=> CACHED [4/6] COPY requirements.txt .
=> CACHED [5/6] RUN pip3 install --no-cache-dir -r requirements.txt
=> CACHED [6/6] COPY hello_inference.py .
=> exporting to image
=> => exporting layers
=> => writing image sha256:90e275a84a4fd208557685731c4885f73b14537b803b3471ff6ed7be2e167fa
=> => naming to docker.io/library/hello_inference

(base) avinash@fedora:~/internship-deployment/imp$ docker run --rm -p 8000:8000 hello_inference

=====
== CUDA ==
=====

CUDA Version 11.8.0

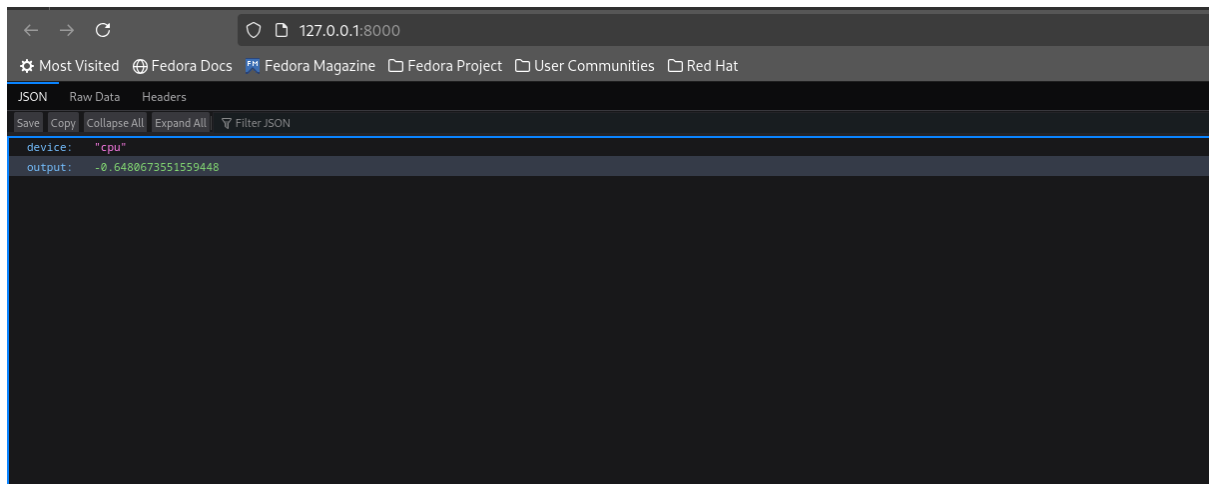
Container image Copyright (c) 2016-2023, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

This container image and its contents are governed by the NVIDIA Deep Learning Container License.
By pulling and using the container, you accept the terms and conditions of this license:
https://developer.nvidia.com/ngc/nvidia-deep-learning-container-license

A copy of this license is made available in this container at /NGC-DL-CONTAINER-LICENSE for your convenience.

WARNING: The NVIDIA Driver was not detected. GPU functionality will not be available.
Use the NVIDIA Container Toolkit to start this container with GPU support; see
https://docs.nvidia.com/datacenter/cloud-native/

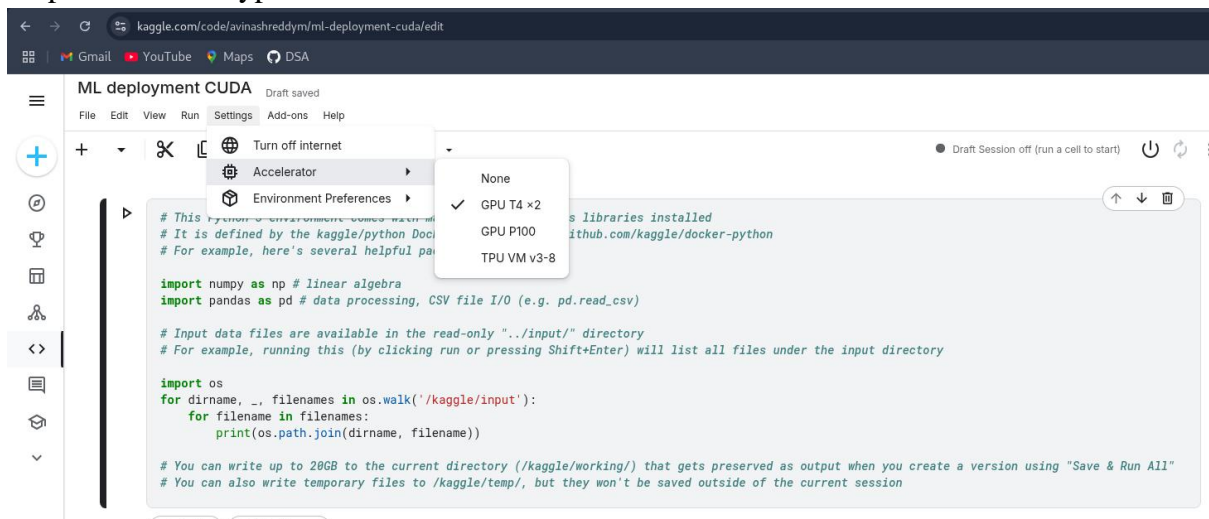
INFO: Started server process [1]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```



So working with the GPU using cloud platforms like Kaggle then:

Step 1: Upload all the files to the kaggle as input

Step 2: Select the type of GPU



Step 3:

!ls /kaggle/input/ml-deployment

!nvidia-smi

Sun Apr 20 09:20:49 2025

NVIDIA-SMI 560.35.03			Driver Version: 560.35.03			CUDA Version: 12.6		
GPU	Name		Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC	
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	
							MIG M.	
0	Tesla T4		Off	00000000:00:04.0	Off		0	
N/A	40C	P8	9W / 70W	1MiB / 15360MiB		0%	Default	N/A
1	Tesla T4		Off	00000000:00:05.0	Off		0	
N/A	40C	P8	9W / 70W	1MiB / 15360MiB		0%	Default	N/A
Processes:								
GPU	GI	CI	PID	Type	Process name	GPU Memory		
	ID	ID				Usage		
No running processes found								

```
!pip install -r /kaggle/working/ml-deployment/requirements.txt
```

```
!python /kaggle/working/ml-deployment/hello_inference.py:
```

```
[6]: !python /kaggle/working/ml-deployment/hello_inference.py
```

```
Hello, Inference! Running on cuda. Output: -0.5899340510368347
```

Dockerfile in ML Model Deployment:

- A Dockerfile serves as the recipe for building the ML serving environment.
- It defines a reproducible container for:
 - Inference code
 - ML model files (e.g., .pkl, .pt, .onnx)
 - Dependency libraries (like TensorFlow, PyTorch, SkLearn)
- GPU-based images (like nvidia/cuda) are used when the model requires GPU acceleration.
- This ensures platform-independent serving (local, cloud, edge devices).
- Docker containers are versionable, scalable, and portable — ideal for production deployments.

ML Deployment Steps with Docker:

1. Train model (locally or in cloud)
2. Save trained model file (e.g., model.pkl)
3. Create serving script (app.py) using Flask, FastAPI, or other frameworks
4. Create Dockerfile with Python environment + dependencies + model + serving script
5. Build image and test locally
6. Deploy to cloud VM, Kubernetes, or edge devices using the Docker image
7. Use GitHub Actions to automate building and pushing to Docker Hub.

.dockerignore File

pycache

**.pyc*

*.pkl

.env

- Ensures unnecessary files are not copied into the Docker image.

WORKDIR /app

- Creates or switches to /app in the container.
- All subsequent commands are executed relative to this directory.

COPY Instruction

COPY . .

- Copies all files from host machine's current directory to the container's current working directory (usually /app).

Container Status:

```
(base) avinash@fedora:~/internship-deployment/imp$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
ba301ece56ab   hello_inference "/opt/nvidia/nvidia_..." 14 minutes ago Up 14 minutes 0.0.0.0:8000->8000/tcp, [::]:8000->8000/tcp   suspicious_mcclintock
```

Logs for Debugging

docker logs <container_id>

If logs show 127.0.0.1 instead of 0.0.0.0, the app may not be accessible externally.

Push Docker Image to Docker Hub:

1. Login to Docker Hub

docker login

2. Tag Image

docker tag avinash0025/myapp:latest

3. Push to Docker Hub

docker push avinash0025/myapp:latest

```
(base) avinash@fedora:~/internship-deployment/imp$ docker push avinash0025/myapp:latest
The push refers to repository [docker.io/avinash0025/myapp]
ead39cf29001: Pushed
4fb0f1b0300c: Pushed
ae610c8943fd: Pushed
8a33682846ef: Pushed
7c62a5a5457e: Pushed
00c354110446: Layer already exists
8a741f0ee517: Layer already exists
63296bbbf98b: Layer already exists
980eb7f7bcb0: Layer already exists
dbd5b7f451e3: Layer already exists
f344b08ff6c5: Layer already exists
86f0cc586e78: Layer already exists
33e57ea5b30a: Layer already exists
851dfefb18192: Layer already exists
6c3e7df31590: Layer already exists
latest: digest: sha256:975d25ff079a539db22c1aff6ca53c20bbaecfe0e072c054d0afeeee4068157b size: 3467
```

4. Pull and Run on Any Machine

docker pull avinash0025/myapp:latest

docker run -d -p 8000:8000 avinash0025/myapp:latest

Docker Compose:

Docker Compose is used to run applications which have multiple containers using a YAML file.

Compose file format:


```

version: "3.7"

services:

...

volumes:

...

networks:

...

```

- Services refers to container's configuration.

services:

frontend:

image: my-vue-app

...

backend:

image: my-springboot-app

...

db:

image: postgres

...

There are multiple settings that we can apply to services like:

1. Pulling an Image
2. Building an Image
3. Configuring the network
4. Setting up the volumes
5. Declaring the dependencies

- Volumes and Networks:

Volumes are the physical areas of disk space shared between the host and the container or even between the containers

Networks define the communication between container and between a container and a host.

- Managing Environment Variables:

Working with environment variables is easy in Compose. We can define static environment variables, and also define dynamic variables with the `${}` notation:

services:

database:

image: "postgres:\${POSTGRES_VERSION}"

environment:

DB: mydb

USER: "\${USER}"

Docker Compose Lifecycle:

To start Docker Compose :

```
docker-compose up
```

After the first time, however, we can simply use *start* to start the services:

```
docker-compose start
```

In case our file has a different name than the default one (*docker-compose.yml*), we can exploit the *-f* and *—file* flags to specify an alternate file name:

```
docker-compose -f custom-compose-file.yml start
```

Compose can also run in the background as a daemon when launched with the *-d* option:

```
docker-compose up -d
```

To safely stop the active services, we can use *stop*, which will preserve containers, volumes, and networks, along with every modification made to them:

```
docker-compose stop
```

To reset the status of our project, instead, we simply run *down*, which will destroy everything with only the exception of external volumes:

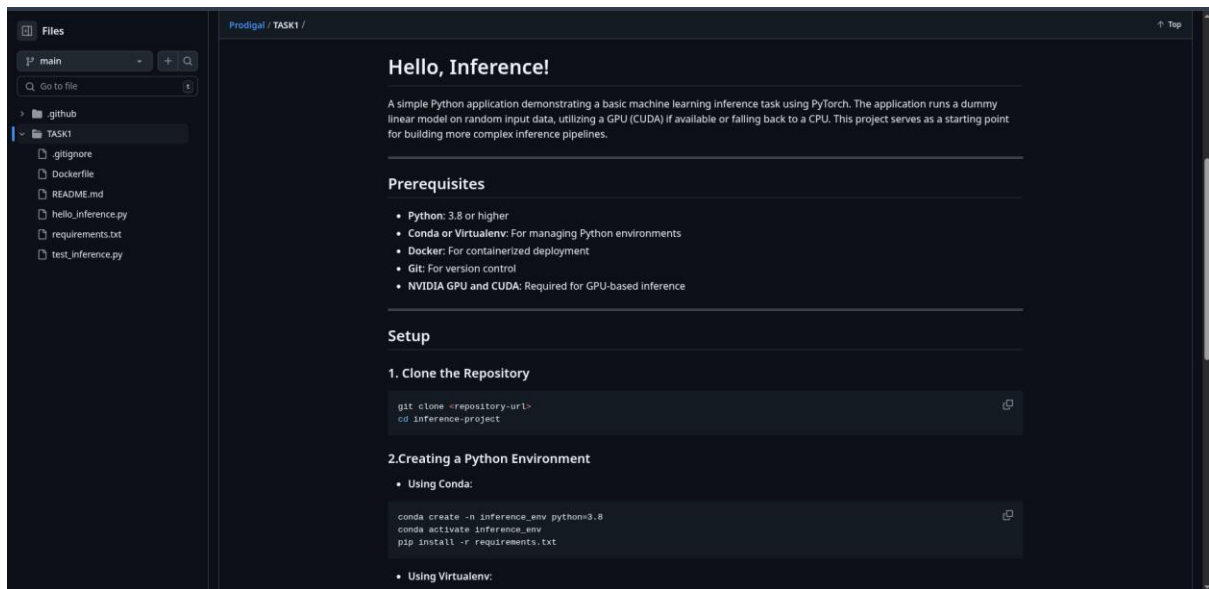
```
docker-compose down
```

Creating a Git repository:

1.Initialize a git repository

2.Creating a Sample Application

- <https://github.com/Avinash00725/Prodigal/tree/main/TASK1>



3. Creating a .yaml(github/workflows/task1.yml)
name: Task1 CI/CD Pipeline

on:

push:

branches:

- main

paths:

- 'TASK1/**'

pull_request:

branches:

- main

paths:

- 'TASK1/**'

workflow_dispatch:

jobs:

test:

runs-on: ubuntu-latest

steps:

- name: Checkout Code

uses: actions/checkout@v3

- name: Set up Python 3.8

uses: actions/setup-python@v4

with:

python-version: '3.8'

- name: Install Dependencies

run: |

python -m pip install --upgrade pip

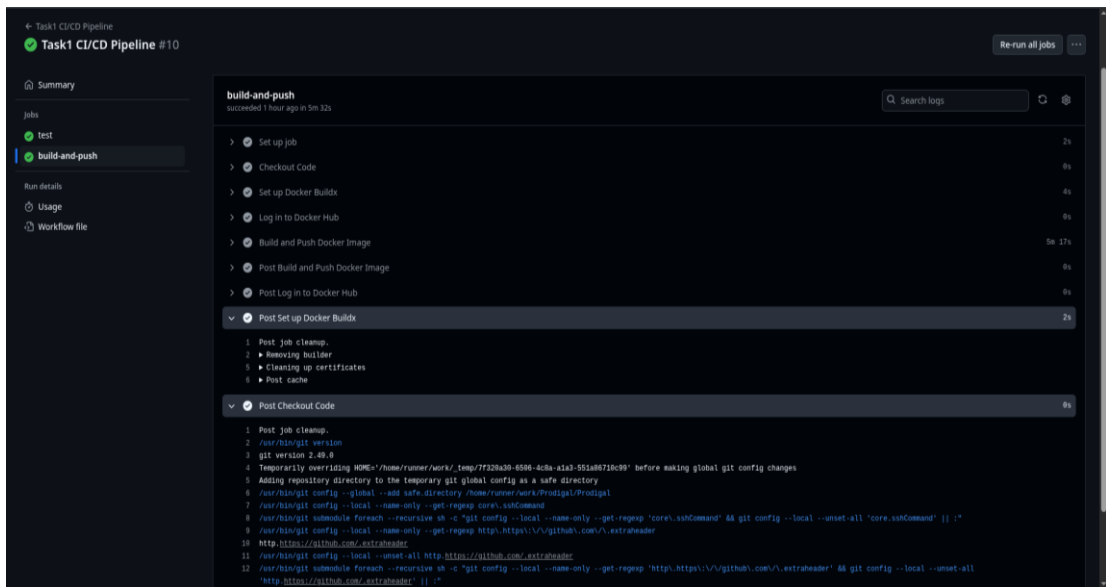
pip install -r TASK1/requirements.txt

- name: Run Unit Tests
run: pytest
working-directory: TASK1/

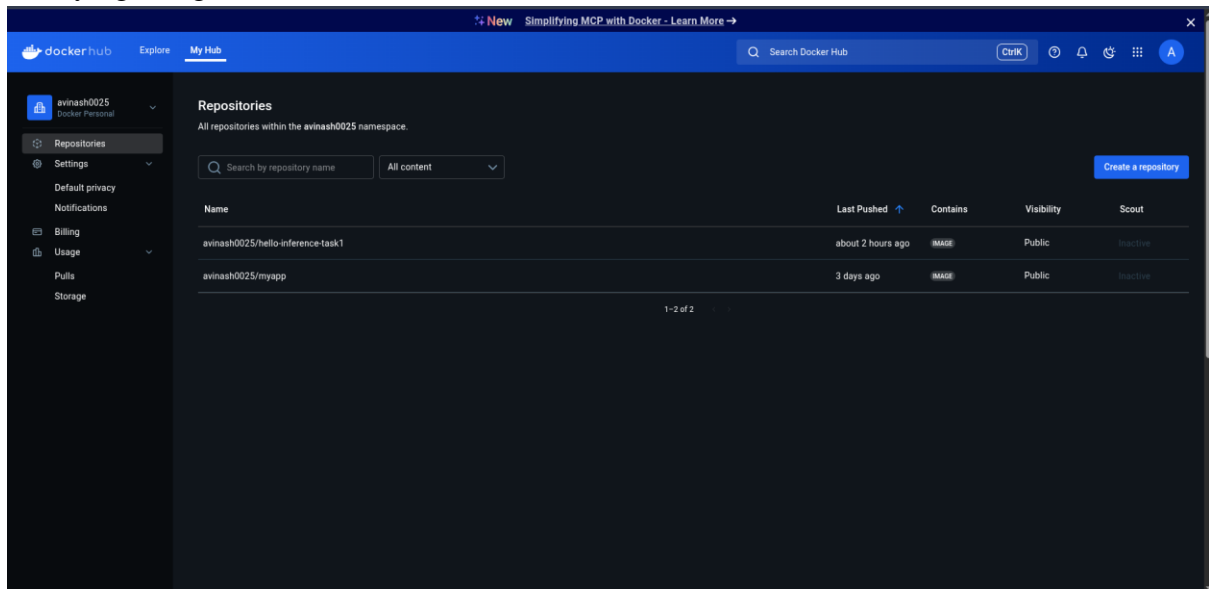
build-and-push:
runs-on: ubuntu-latest
needs: test
if: (github.event_name == 'push' && github.ref == 'refs/heads/main') || github.event_name == 'workflow_dispatch'
steps:
- name: Checkout Code
uses: actions/checkout@v3
- name: Set up Docker Buildx
uses: docker/setup-buildx-action@v3
- name: Log in to Docker Hub
uses: docker/login-action@v3
with:
username: \${ secrets.DOCKER_USERNAME }
password: \${ secrets.DOCKER_PASSWORD }
- name: Build and Push Docker Image
uses: docker/build-push-action@v5
with:
context: TASK1/
file: TASK1/Dockerfile
push: true
tags: avinash0025/hello-inference-task1:latest

Configuration using the Github actions for CI/CD

1. Saving the workflow file(github/workflow/task1.yml)
2. Need to verify the Github Secrets.Ensure that the secrets are set for the github repository like `Settings` > `Secrets and Variables` > `Actions` > `New Repository Secret`.
3. Commit the changes in the repository
4. Check Workflow Execution



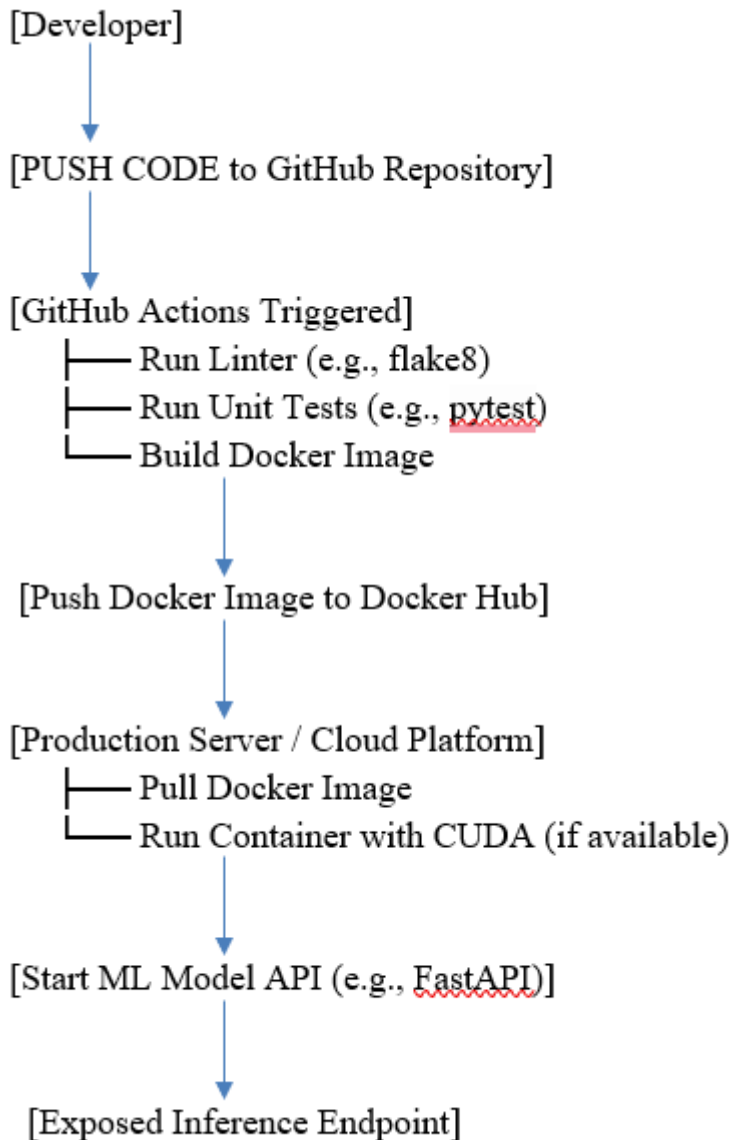
Verifying using the DockerHub:



- The Docker Image hello-inference-task1 build using the CI/CD pipeline using the Github actions .

<https://github.com/Avinash00725/Prodigal/tree/main/TASK1>

Combined CI/CD + Docker Deployment Flow



<https://github.com/Avinash00725/Prodigal/tree/main/TASK1>