---

# ✍️ Final Project Documentation: Guardian NLP

## 1. Project Overview

"Guardian NLP" is an end-to-end MLOps project that demonstrates the complete lifecycle of a machine learning application, from model creation to automated deployment. The project's goal is to build, package, and orchestrate a toxic comment classifier. The system is designed to analyze a piece of text and return a classification (toxic/not-toxic) with a confidence score. This document outlines the entire workflow, showcasing a professional-grade process involving data versioning, containerization, local Kubernetes deployment, and a full CI/CD pipeline for automation.

---

## 2. System Architecture

The project architecture is a comprehensive MLOps pipeline designed for reproducibility, scalability, and automation.

1. **Model Development**: The process begins with training a model in a local Jupyter environment.
2. **Versioning**: The trained model artifacts are versioned with **DVC** and stored in a local remote, while the corresponding pointer files are tracked in **Git**.
3. **API Serving**: The model is wrapped in a **Flask** API to serve predictions over HTTP.
4. **Containerization**: The entire application, including the Flask server and all dependencies, is packaged into a portable **Docker** image.
5. **Local Orchestration**: The Docker container is deployed to a local **Kubernetes** cluster (Minikube), demonstrating how to manage and scale the application in a production-like environment.
6. **CI/CD Automation**: The entire build and publishing process is automated with **GitHub Actions**. A push to the GitHub repository automatically triggers a workflow that builds the Docker image and pushes it to a public **Docker Hub** registry.

---

## 3. Technology Stack

- **Python**: The core programming language for all development.
- **Jupyter Notebook & Anaconda**: For interactive model development and environment management.
- **Pandas & Scikit-learn**: For data manipulation, feature extraction, and model training.
- **Git**: For source code version control.
- **DVC (Data Version Control)**: For versioning large model artifacts.
- **Flask**: A lightweight web framework for building the prediction API.
- **Docker**: The containerization platform used to package the application.
- **Kubernetes (Minikube)**: For orchestrating the containerized application locally.
- **GitHub & GitHub Actions**: For code hosting and CI/CD automation.
- **Docker Hub**: As a public container registry for storing the Docker image.

---

## 4. Model Development Phase

### 4.1. Data Loading and Preprocessing

The project used the "Jigsaw Toxic Comment Classification Challenge" dataset from Kaggle. The text was cleaned to create a normalized dataset for the model.

Python

```
# Code Snippet: Loading and Preprocessing
import pandas as pd
import re

df = pd.read_csv('data/train.csv')

def preprocess_text(text):
```

```
    text = text.lower()
    text = re.sub(r'[^a-z\s]', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

df['cleaned_text'] = df['comment_text'].apply(preprocess_text)
```

### 4.2. Feature Extraction and Model Training

The cleaned text was split into training and testing sets and converted into numerical vectors using TfidfVectorizer. A Logistic Regression model was then trained on this data, achieving an accuracy of **95.66%**.

Python

```
# Code Snippet: Training and Evaluation
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

X = df['cleaned_text']
y = df['toxic']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

vectorizer = TfidfVectorizer(max_features=5000)
X_train_tfidf = vectorizer.fit_transform(X_train)

model = LogisticRegression(max_iter=1000)
model.fit(X_train_tfidf, y_train)
```

### 4.3. Model Versioning with DVC

The trained model (model.pkl) and vectorizer (vectorizer.pkl) were versioned using DVC to

ensure reproducibility. A local remote was configured, and the artifacts were pushed to it.

Bash

```
# DVC Commands
pip install dvc
git init
dvc init
dvc remote add -d localremote "D:\path\to\dvc-storage"
dvc add model.pkl vectorizer.pkl
git commit -m "feat: track models with dvc"
dvc push
```

---

## 5. API Development and Containerization

### 5.1. Flask API (app.py)

A Flask application was created to serve the model. The script loads the DVC-tracked model, defines the preprocessing function, and creates a /predict endpoint.

Python

```
# Code Snippet: app.py (abridged)
# Note: In a real CI/CD pipeline, the app would start with a 'dvc pull' command
# to retrieve the model files. For local testing, the files are already present.
import pickle
from flask import Flask, request, jsonify

app = Flask(__name__)
# Load model and vectorizer
```

```
with open('vectorizer.pkl', 'rb') as f:
    vectorizer = pickle.load(f)
with open('model.pkl', 'rb') as f:
    model = pickle.load(f)

@app.route('/predict', methods=['POST'])
def predict():
    # ... prediction logic ...
    pass
```

### 5.2. Containerization with Docker (Dockerfile)

A Dockerfile was created to package the Flask application and its dependencies into a portable container.

Dockerfile

```
# Code Snippet: Dockerfile
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
# Note: For a CI/CD setup, this Dockerfile would be modified
# to first install DVC and run 'dvc pull'
CMD ["python", "app.py"]
```

---

## 6. Kubernetes Deployment

The containerized application was deployed to a local Kubernetes cluster managed by **Minikube**.

### 6.1. Cluster Setup

Minikube was started using the Docker driver to bypass hardware virtualization requirements. The local Docker image was then loaded into the cluster.

Bash

```
minikube start --driver=docker
minikube image load guardian-nlp-api
```

### 6.2. Deployment and Service (deployment.yaml, service.yaml)

A **Deployment** file was created to manage the application pods, specifying three replicas for high availability. A **Service** file was created to expose the deployment via a stable network address (NodePort).

Bash

```
# Kubernetes Commands
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
minikube service guardian-nlp-service --url
```

---

## 7. CI/CD Automation with GitHub Actions

The final step was to create a CI/CD pipeline to automate the build and publishing of the Docker image.

### 7.1. Workflow Configuration (main.yaml)

A GitHub Actions workflow was defined in .github/workflows/main.yaml. The workflow is triggered on every push to the main branch. It checks out the code, logs into Docker Hub using secrets, and then builds and pushes the Docker image to a public repository.

YAML

```yaml
# Code Snippet: .github/workflows/main.yaml
name: CI/CD Pipeline
on:
  push:
    branches: [ "main" ]
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - name: Check out repo
        uses: actions/checkout@v4
      - name: Log in to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKOERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${{ secrets.DOCKERHUB_USERNAME }}/guardian-nlp-api:latest
```

This completes the MLOps lifecycle, resulting in a fully versioned, containerized, orchestrated, and automated machine learning application.