IIIT-Bangalore

Software Production Engineering

CSE - 816

Professors:
Prof. B Thangaraju

Submitted by:
Madhav Girdhar (IMT2022009)
Bijeet Basak (IMT2022510)

Introduction

In this project we have created a End-to-End MLOps Pipeline for Brain Tumor Classification using MobileNetV2. Machine Learning models in real-world applications require not only high accuracy but also reliable deployment, monitoring, versioning, and automation. Traditional ML workflows often fail in production due to lack of reproducibility, poor deployment strategies, and manual intervention. This project implements a complete MLOps pipeline for Brain Tumor Classification using deep learning and modern DevOps practices. The system integrates model training, data versioning, CI/CD automation, containerization, model registry, and Kubernetes deployment, providing a scalable and production-ready solution.

Project Objective

The primary objective of this project is to design and implement an automated, secure, and scalable CI/CD pipeline for deploying a machine learning–based application on a Kubernetes cluster using modern DevOps tools. The specific objectives of the project are as follows:

- To automate the build, test, and deployment process using Jenkins to reduce manual intervention and deployment errors.
- To containerize the machine learning application using Docker and manage image versions through DockerHub.
- To deploy and manage the application on Kubernetes, ensuring scalability, reliability, and zero-downtime updates.
- To use Ansible for configuration management and automated Kubernetes deployment, promoting reusability and consistency through Ansible roles.
- To securely manage sensitive configuration data such as Kubernetes credentials using Ansible Vault and Jenkins credential management.
- To integrate DVC (Data Version Control) for tracking datasets and machine learning models, ensuring data reproducibility and version control across experiments.
- To incorporate MLflow for managing the machine learning lifecycle, including experiment tracking, model versioning, and performance comparison.
- To enable seamless integration between GitHub, Jenkins, Docker, Ansible, Kubernetes, DVC, and MLflow for an end-to-end MLOps workflow.

Tools Used

This project integrates multiple tools from the Machine Learning and DevOps ecosystem to build a scalable and automated MLOps pipeline. Each tool was selected based on its reliability, industry adoption, and suitability for the project requirements.

1. **Python** : Python is the primary programming language used for model development, data preprocessing, backend API development, and pipeline scripting due to its rich ML and DevOps ecosystem.
2. **TensorFlow & Keras** : TensorFlow with Keras API is used to develop and train the deep learning model.
    - Implements MobileNetV2 for image classification
    - Supports transfer learning and fine-tuning
    - Efficient for both training and inference
3. **Data Version Control (DVC)** : DVC is used to version and manage large datasets separately from Git.
    - Tracks raw and processed datasets.
    - Ensures reproducibility across experiments.
    - Enables data synchronization using dvc pull and dvc push.
4. **Git & GitHub** : Git is used for source code version control, while GitHub acts as the central repository.
    - Tracks changes in code and configurations
    - Triggers Jenkins pipelines on repository updates
    - Enables collaborative development
5. **Jenkins** : Jenkins is used to build the CI/CD pipeline for automation.
    - Automates model training and deployment.
    - Builds Docker images
    - Pushes images to Docker Hub
    - Deploys updated services to Kubernetes
6. **MLflow** : MLflow is used as the model tracking and registry platform.
    - Tracks training experiments and metrics
    - Stores model artifacts
    - Manages multiple model versions
    - Promotes selected models to Production stage

7. **Docker & DockerHub** : Docker is used to containerize the model inference API for consistent and portable deployment, while Docker Hub serves as a container registry to store and distribute the Docker images used by Kubernetes.

8. **Kubernetes (Minikube)** : Kubernetes is used for container orchestration and deployment.

    - Manages application pods and services.
    - Supports rolling updates
    - Ensures high availability during updates

9. **Flask** : Flask is used to build a lightweight REST API for model inference.

    - Accepts MRI images via HTTP requests
    - Performs real-time predictions
    - Exposes endpoints for frontend integration

10. **Ansible** : Used for configuration management, ensuring automated and consistent setup of environments and deployment tasks.

    - Implemented Ansible Vault and modular Ansible roles for secure credential handling and reusable infrastructure automation.

11. **Ngrok** :

    - Used to expose the local Jenkins server to the internet for receiving GitHub webhooks.
    - Enables automatic triggering of the Jenkins CI/CD pipeline whenever updates are pushed to the GitHub repository.

12. **ELK Stack (Elasticsearch, Logstash, Kibana):** :

    - The ELK stack was integrated into the system to enable centralized logging, monitoring, and real-time visualization of all activities across the MLOps pipeline.
    - Elasticsearch: Stores and indexes logs collected from Jenkins, Flask API, Docker containers, and Kubernetes pods.
    - Logstash: Processes and transforms logs before sending them to Elasticsearch. Configured pipelines collect: Jenkins job logs, Flask API inference logs, Kubernetes pod logs, Docker container logs
    - Kibana: Provides dashboards to visualize pipeline executions, API request logs, error rates, pod events, and deployment history. Helps identify performance bottlenecks, failures, and long-term trends across the system.
    - ELK enables observability and monitoring, improving the reliability of the deployed ML service.

1 Workflow

The end-to-end workflow of this project follows a highly structured DevOps–MLOps pipeline designed to ensure complete automation, reproducibility, scalability, and secure deployment of the machine learning model. The workflow spans across data management, model lifecycle tracking, continuous integration, containerization, orchestration, and secure infrastructure automation. This integrated pipeline reflects modern industry standards for deploying ML-driven applications in production environments.

The complete workflow proceeds through the following major stages:

- **Dataset Update and Version Control Using DVC:** The workflow begins with updating or appending new training and testing samples to the dataset. All modifications are tracked using Data Version Control (DVC), which maintains version histories of datasets and intermediate artifacts. Commands such as `dvc commit` and `dvc push` ensure that data changes are stored in remote storage, while `git add`, `git commit`, and `git push` synchronize metadata and pipeline definitions with the Git repository.

- **Automated Pipeline Trigger via Webhook Integration:** Upon every push to the GitHub repository, a Jenkins pipeline is triggered automatically through a webhook configured using Ngrok. This allows the CI/CD server to respond instantly to any code or data update, enabling continuous and event-driven automation.

- **Reproducible ML Pipeline Execution with DVC:** The Jenkins pipeline starts by performing a `dvc pull` operation to fetch all necessary artifacts, models, and datasets to ensure environmental consistency. It then executes `dvc repro`, which reconstructs the entire ML workflow including data preprocessing, feature engineering, model training, evaluation, and inference generation. Every stage is executed deterministically, ensuring that results are reproducible regardless of the execution environment.

- **Experiment Tracking and Model Registry Management Using MLflow:** After the model is trained, detailed experiment metadata such as hyperparameters, metrics, loss curves, and artifacts are logged in MLflow. The best-performing model is then promoted and registered as the *Production Model*. This registry-based approach ensures clear lineage, traceability, and governance for all deployed models.

- **Containerization and Versioned Image Publishing with Docker:** The production-ready model is loaded into the inference service, after which a new Docker image is constructed. The image is tagged using the Jenkins build identifier to ensure versioned and traceable deployments. The image is pushed to DockerHub, providing a centralized and reliable artifact repository that can be accessed by Kubernetes during rollout.

- **Infrastructure Automation Using Ansible Roles:** Jenkins triggers an Ansible playbook that utilizes a structured Ansible role designed specifically for Kubernetes deployment management. The role includes modular tasks for updating manifest files, applying configuration changes, executing rollouts, and notifying handlers for restarting services whenever required. This role-based architecture ensures maintainability and scalability of deployment automation.

- **Secure Credential Management Using Ansible Vault:** Kubernetes cluster credentials such as API endpoints, service account tokens, and certificates are stored in encrypted form using Ansible Vault. These credentials are decrypted only during runtime, ensuring secure, compliant, and audit-ready automation practices.

- **Automated Deployment and Rolling Updates on Kubernetes:** The Ansible role applies updated Kubernetes manifests that reference the newly built Docker image. Kubernetes performs a rolling update, replacing older pods with new ones gradually, ensuring zero downtime for the inference service.

- **Dynamic Scaling Using Horizontal Pod Autoscaling (HPA):** Horizontal Pod Autoscaling is integrated into the cluster to automatically adjust the number of running pods based on CPU utilization or user inference load. This ensures that the system remains performant under high demand and cost-efficient during low activity periods.

- **Centralized Logging and Monitoring Using ELK Stack:** After deployment, all logs generated across the MLOps ecosystem are forwarded to the ELK Stack for centralized analysis. Logstash collects logs from multiple components of the pipeline, including the Jenkins build and deployment stages, the Flask API (covering request details, inference times, and error traces), Kubernetes pod events such as scaling actions and restarts, and Docker container logs. These logs are then passed to Elasticsearch, where they are indexed and stored for efficient querying and long-term access. Kibana sits on top of Elasticsearch and provides interactive dashboards that visualize API request statistics, model performance latency, Kubernetes pod health, and trends in Jenkins job executions. Through this integrated monitoring system, observability is significantly enhanced, enabling proactive debugging, early anomaly detection, and improved stability of the deployed machine learning service.

This workflow demonstrates a fully automated and production-grade MLOps pipeline, integrating multiple industry-standard tools in a cohesive manner. It ensures consistency across data, code, and infrastructure while enabling scalable, reliable, and secure deployment of machine learning models.

Additional Tasks and Contributions

In addition to the core implementation of the CI/CD and MLOps pipeline, several additional tasks and enhancements were undertaken to improve the usability, automation, security, and scalability of the overall system. These contributions strengthen both the functional and operational aspects of the project. The major additional tasks include:

- **Frontend Interface for MRI Inference:** A user-friendly frontend interface was designed and developed to allow users to upload MRI images and receive predictions from the deployed machine learning model. This interface enhances accessibility and provides an intuitive method for interacting with the inference service.
- **Implementation of Ansible Roles:** Modular and reusable Ansible roles were implemented to manage Kubernetes deployment YAML files and automate tasks such as updating images, restarting deployments, and reloading configurations. This improved structure ensures cleaner playbooks, better maintainability, and easier scalability.
- **Secure Configuration Using Ansible Vault:** Sensitive cluster configurations, including the Kubernetes API server endpoint, authentication tokens, and certificates, were securely stored using Ansible Vault. This ensures that all critical credentials remain encrypted and protected throughout the automation workflow.
- **Adoption of MLOps Practices:** The project adopted domain-specific MLOps methodologies to ensure reproducibility, experiment tracking, data versioning, and automated deployment of machine learning models. Tools such as DVC and MLflow were integrated to handle dataset versioning, pipeline reproducibility, model registry management, experiment tracking, and production model promotion.
- **Horizontal Pod Autoscaling (HPA) in Kubernetes:** HPA was implemented to automatically scale the number of running pods based on CPU utilization or load generated by incoming inference requests. This ensures system responsiveness, cost efficiency, and robustness under varying workloads.
- **Integration of ELK Stack for Monitoring and Log Management:** A complete ELK-based monitoring solution was added to the project to improve visibility and debugging capabilities. Configured Logstash pipelines to collect logs from Jenkins, the Flask inference API, and Kubernetes pods. Centralized all logs inside Elasticsearch, ensuring they could be searched and analyzed efficiently. Built Kibana dashboard to visualize API response times, Total Predictions, Average of Confidence over Time, Average of Confidence and Count of predictions below threshold. This significantly improved system reliability and allowed easy monitoring of the pipeline and model behavior in production.

Results

Here are the some of screenshots of the work which we have done successfully.

- Jenkins Pipeline view for model life cycle automation.

Figure 1: Jenkins Pipeline execution View Model execution.

- Jenkins Pipeline view for Front end Visualization.

Figure 2: Jenkins Pipeline execution View for Front end.

- Model Registry for model version controlling and mark them in production.

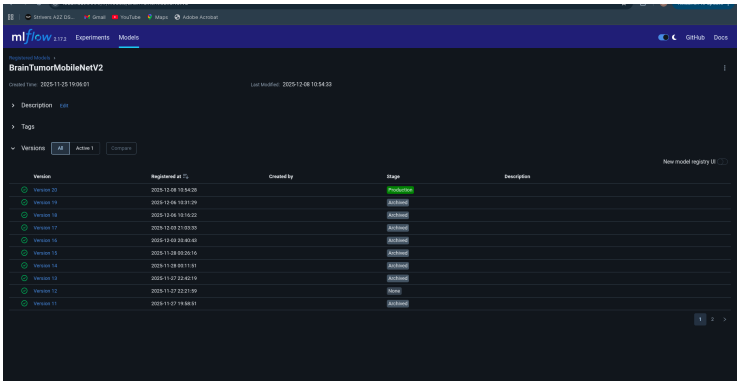

Figure 3: Model Registry.

- Kubernetes dashboard for visualizing deployed containers.

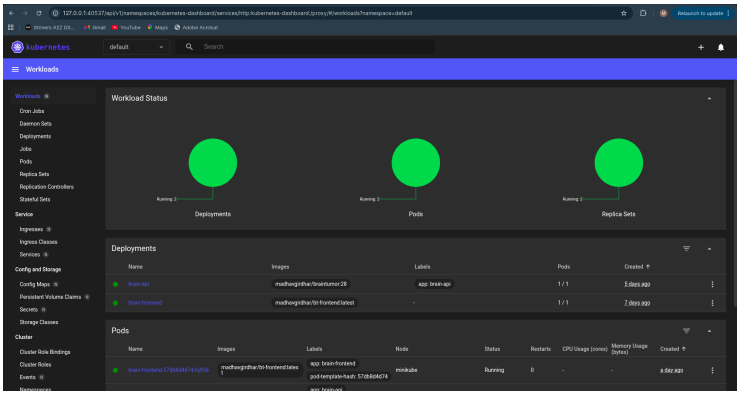

Figure 4: Kubernetes dashboard.

- Final Application predicting type of Tumor.

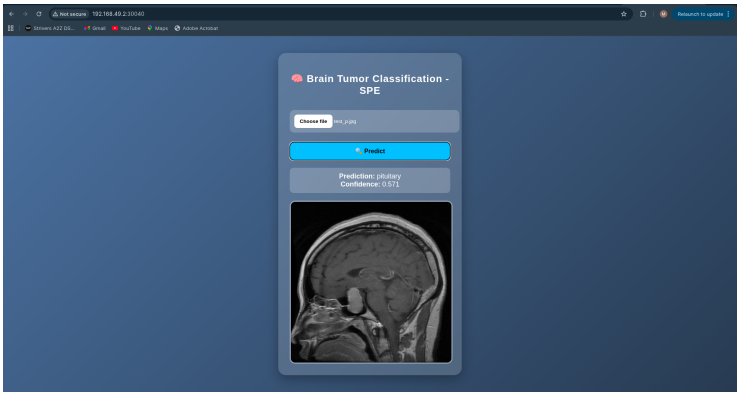

Figure 5: Final Application view.

- Kibana Dashboard

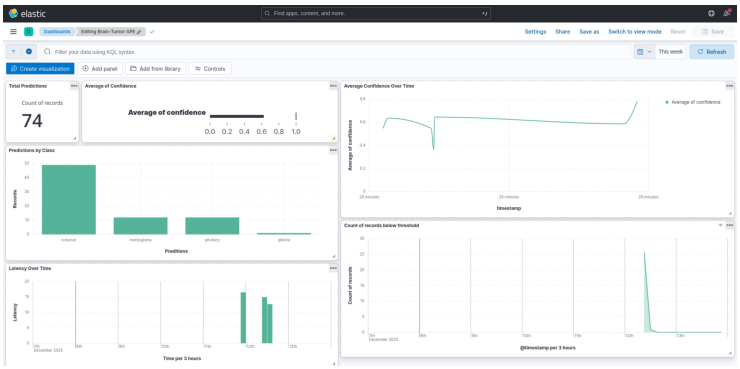

Figure 6: Kibana dashboard showing centralized monitoring and log analytics.

2 Challenges Faced

During the project, several challenges were encountered while connecting different tools and making the complete MLOps pipeline work smoothly. The main challenges are explained below in simple and easy-to-understand language:

- **Kubernetes Authentication Problems:** One major challenge was authenticating Kubernetes from Jenkins and Ansible. The kubeconfig file created by Minikube had local machine paths for certificates, which did not work on the Jenkins server. We had to create a portable kubeconfig with embedded certificates and secure it using Ansible Vault.
- **Handling Sensitive Credentials Safely:** Storing important credentials such as Kubernetes tokens, API server details, and DockerHub login securely was difficult. Sometimes deployments failed due to incorrect or missing credentials. This issue was solved by encrypting all sensitive information with Ansible Vault and managing secure credentials inside Jenkins.
- **Using DVC in the CI Pipeline:** Integrating DVC into the Jenkins pipeline was a challenge because the local environment and the Jenkins environment were different. DVC sometimes could not find datasets or artifacts, which caused errors. We had to properly configure DVC remote storage and fix pipeline paths to make DVC work smoothly inside Jenkins.
- **MLflow Model Tracking Issues:** Working with MLflow also caused issues at the beginning. Sometimes the model artifacts were not saved correctly, or the experiment paths were wrong. There were also conflicts when promoting a model to the Production stage. These issues were fixed by setting correct experiment locations and properly registering model versions.
- **Making the End-to-End Pipeline Stable:** Since the project used many tools together (GitHub, Jenkins, DVC, MLflow, Docker, Ansible, Kubernetes), it was difficult to make everything work correctly in one flow. If one small step failed, the whole pipeline stopped. Many rounds of testing, debugging, and fixing were needed to achieve a fully stable and automated workflow.
- **ELK Stack Integration Challenges:** Setting up the ELK Stack using Docker Compose initially caused issues because Elasticsearch, Logstash, and Kibana required specific memory configurations and version compatibility. Logstash sometimes failed to receive logs from Docker and Kubernetes due to incorrect pipeline configurations and port mapping errors. Additionally, ensuring that Jenkins and the Flask API forwarded logs consistently required several adjustments in log drivers and pipeline settings.

CodeBase and DockerHub Image

- **GitHub - ML (model)** : GitHub Repository
- **GitHub - Frontend** : GitHub Repository
- **DockerHub - ML (model)** : DockerHub Image
- **DockerHub - Frontend** : DockerHub Image