CSE 816: Software Production Engineering Project

IoT Edge MLOps Pipeline

A Kubernetes-Orchestrated Self-Healing Framework

Sanchit Kumar Dogra IMT2022035
Anurag Ramaswamy IMT2022103

December 10, 2025

Project Links

GitHub Repository

Docker Hub Repository

Contents

Abstract

This project implements a comprehensive DevOps and MLOps framework designed to automate the lifecycle of an IoT Edge Intelligence application. This project targets the **IoT/MLOps domain**, featuring real-time sensor data ingestion, model drift detection and automated self-healing (retraining).

The infrastructure is built on **Kubernetes (Minikube)**, managed via **Ansible Playbooks** with modular roles and automated through a **Jenkins CI/CD pipeline** with GitHub webhooks. The system includes robust monitoring via the **ELK Stack** (Logstash MQTT Input), secure credential management using **Ansible Vault** and dynamic scalability using **Horizontal Pod Autoscaling (HPA)**.

This report details the system architecture, the implementation of advanced security and scalability features and validation of the automated SDLC workflows.

1 Introduction

1.1 Project Objective

The primary objective of this project is to simulate a production-grade software environment for a complex distributed system. Rather than deploying a monolithic application, we created a microservices-based solution that integrates Data Engineering, Machine Learning and DevOps principles.

1.2 Project Domain

This project targets the **MLOps and IoT (Internet of Things)** domains.

- **IoT Data Ingestion:** The system utilizes an MQTT Broker (Mosquitto) to ingest high-frequency sensor data (Temperature, Humidity, VOC) from simulated edge devices.
- **Self-Healing MLOps:** The system does not just deploy a static model. It monitors prediction error (RMSE) in real-time. If "Model Drift" is detected (RMSE $>$ Threshold), the system automatically triggers a retraining pipeline in the cloud context, updating the edge inference engine without downtime.
- **Edge Computing:** The inference logic runs in a lightweight container optimized for edge deployment, decoupled from the training infrastructure.

1.3 Tools Used

- **Version Control:** Git & GitHub
- **CI/CD:** Jenkins (Pipeline as Code) with Webhooks
- **Containerization:** Docker & Docker Hub
- **Orchestration:** Kubernetes (Minikube)
- **Configuration Management:** Ansible (Modular Roles & Vault)
- **Monitoring:** ELK Stack (Elasticsearch, Logstash, Kibana)
- **Visualization:** Streamlit (Dashboard) & MLflow (Experiments)
- **Testing & Security:** Pytest (Unit Testing), Bandit (SAST), Safety (Dependency Analysis)

2 System Architecture

2.1 Project Directory Structure

To maintain a clean separation of concerns between application code, infrastructure definitions, and automation logic, the repository is structured as follows. This hierarchy separates the source code (`app`, `cloud`, `devices`) from the deployment logic (`ansible`, `k8s`).

```
.
|-- ansible/
|   |-- roles/k8s_deploy/tasks/
|   |   `-- main.yml        # Modular Role definition
|   |-- deploy.yml          # Main Playbook wrapper
|   |-- inventory.ini       # Localhost inventory
|   `-- secrets.yml         # Encrypted Ansible Vault credentials
|-- app/
|   `-- edge_infer.py       # Drift Detection & Inference Logic
|-- cloud/
|   `-- train.py            # Model Training Script
|-- dashboard/
|   `-- dashboard.py        # Streamlit Visualization
|-- devices/
|   `-- publisher.py        # IoT Sensor Simulator
|-- k8s/                    # Kubernetes Manifests
|   |-- app-deployment.yaml # Main Pod (App + Publisher + Dashboard)
|   |-- elk-stack.yaml      # Elasticsearch & Kibana
|   |-- hpa.yaml            # Horizontal Pod Autoscaler
|   |-- logstash.yaml       # Log Shipper Service
|   |-- mlflow.yaml         # MLflow Tracking Server
|   `-- mosquitto.yaml      # MQTT Broker Service
|-- logstash/
|   |-- Dockerfile          # Custom Image build
|   `-- logstash.conf       # MQTT Input Configuration
|-- mosquitto/
|   `-- mosquitto.conf      # Broker Configuration
|-- tests/
|   `-- test_core.py        # Unit & Integration Tests
|-- .env                    # Local Environment Variables (Ignored in GitHub)
|-- .gitignore
|-- Dockerfile              # Main Application Dockerfile
|-- Jenkinsfile             # CI/CD Pipeline Definition
|-- Makefile                # Automation Entry Points
|-- README.md
|-- docker-compose.yml      # (Legacy/Local testing)
|-- report.pdf
|-- requirements.txt        # Python Dependencies
```

2.2 High-Level Design

The architecture follows a microservices pattern deployed on a Kubernetes cluster.

- **Broker Service:** Handles MQTT communication between sensors and services.
- **Inference Service:** Subscribes to sensor data, runs the ML model, and publishes predictions. It supports hot-reloading of models.
- **Logstash Service:** A dedicated log shipper that subscribes to the MQTT broker and forwards structured logs to Elasticsearch.
- **MLflow Service:** Tracks experiment metrics and stores model artifacts.
- **ELK Stack:** Provides centralized logging and visualization.

2.3 Kubernetes Resource Definitions

We adopted a GitOps-style approach where the entire infrastructure state is defined declaratively in YAML manifests:

- **app-deployment.yaml:** Defines the main Pod containing the Inference, Publisher, and Dashboard containers using the Sidecar pattern.
- **mlflow.yaml:** Deploys the MLflow Tracking Server for logging experiments and model artifacts.
- **hpa.yaml:** Configures auto-scaling to maintain 1-3 replicas based on a 50% CPU target.
- **elk-stack.yaml:** Deploys the Elasticsearch database and Kibana dashboard for log aggregation.
- **logstash.yaml:** Deploys the custom log shipper to forward MQTT messages to Elasticsearch.
- **mosquitto.yaml:** Deploys the MQTT Broker service to handle sensor communication.

2.4 The DevOps Workflow

The SDLC is fully automated via Jenkins:

1. **Code Push:** Developer pushes code to GitHub.
2. **Webhook Trigger:** GitHub notifies Jenkins via a Webhook (Ngrok).
3. **Build & Test:** Jenkins builds the Docker image and runs syntax tests.
4. **Push Registry:** The artifact is pushed to Docker Hub with a version tag.
5. **Deploy:** Jenkins triggers Ansible.
6. **Configure:** Ansible (using Roles and Vault) applies K8s manifests to update the cluster seamlessly.
7. **Testing & Security:** Pytest, Bandit, Safety

2.5 High Level Architecture Diagram

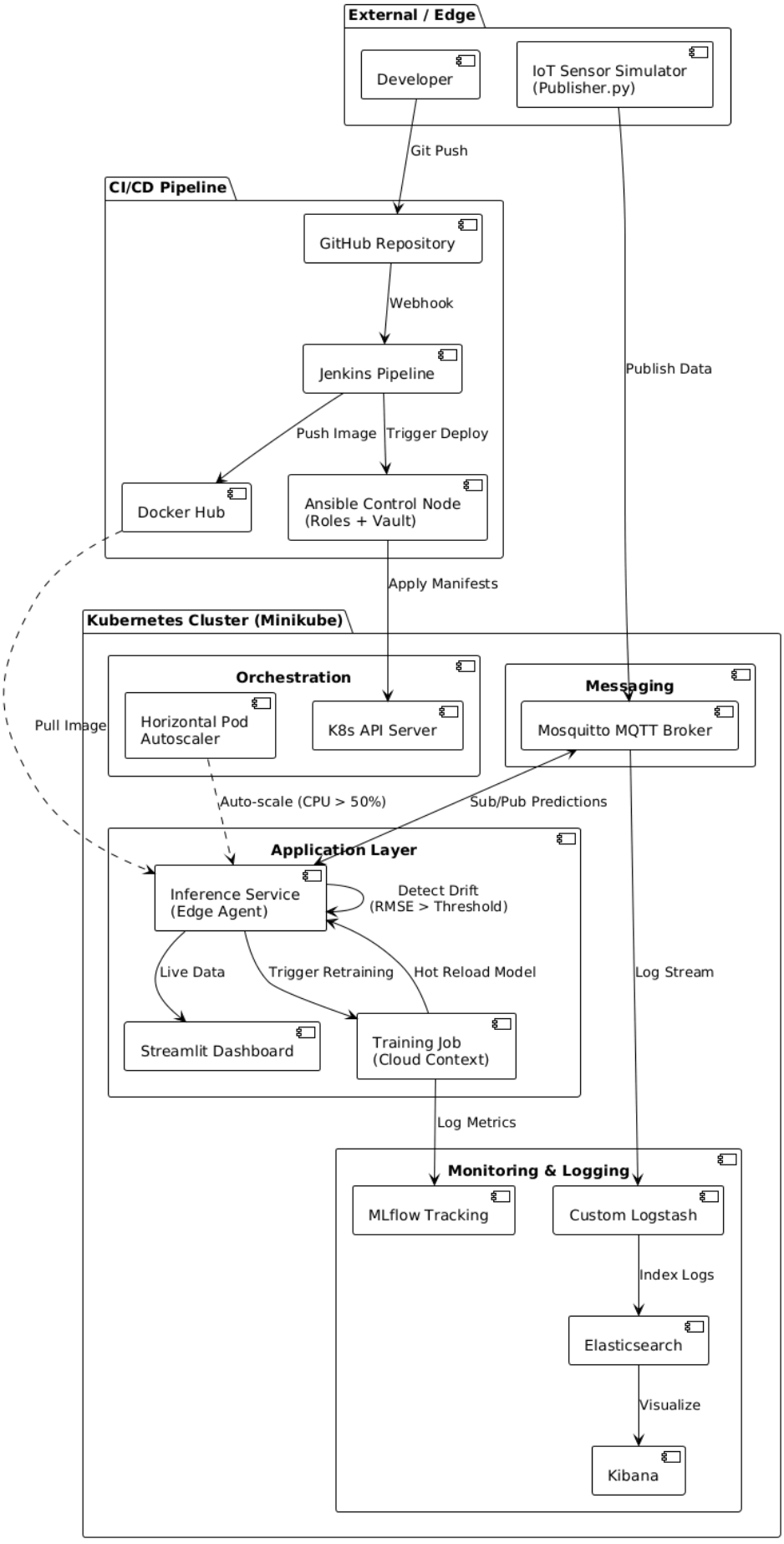

Figure 1: Architecture diagram

3 Implementation of Advanced Features

3.1 Modular Design: Ansible Roles

To ensure maintainability and modular design, we moved away from a monolithic playbook structure. We refactored the infrastructure code into a standardized Ansible Role named `k8s_deploy`.

- **Directory Structure:**

```
ansible/
 deploy.yml          # Entry point
 roles/
     k8s_deploy/
         tasks/
             main.yml # Contains all task definitions
```

- **Execution Flow:** The `deploy.yml` playbook acts as a wrapper. It defines the target hosts ('localhost') and loads secure variables ('secrets.yml'), but delegates the actual execution logic to the role.
- **Task Logic:** Inside `roles/k8s_deploy/tasks/main.yml`, the tasks are executed sequentially to ensure dependency order:
  1. **Infrastructure:** Deploys the MQTT Broker ('mosquitto.yaml').
  2. **Monitoring:** Deploys MLflow, followed by the ELK Stack and the custom Logstash adapter.
  3. **Application:** Deploys the main Edge Application ('app-deployment.yaml').
  4. **Scalability:** Applies the Horizontal Pod Autoscaler ('hpa.yaml').

This encapsulation allows us to reuse the deployment logic across different environments (dev/prod) simply by calling the role in different playbooks.

```
---
- name: Deploy Framework to K8s
  hosts: localhost
  gather_facts: false

  # For secure storage of sensitive data
  vars_files:
    - secrets.yml

  # For modular design
  roles:
    - k8s_deploy
```

Listing 1: Main Playbook utilizing Roles and Vault

3.2 Secure Storage: Ansible Vault

Sensitive credentials, specifically the configuration secrets, are not hardcoded in plain text. We implemented **Ansible Vault** to encrypt the 'secrets.yml' file using AES256 encryption.

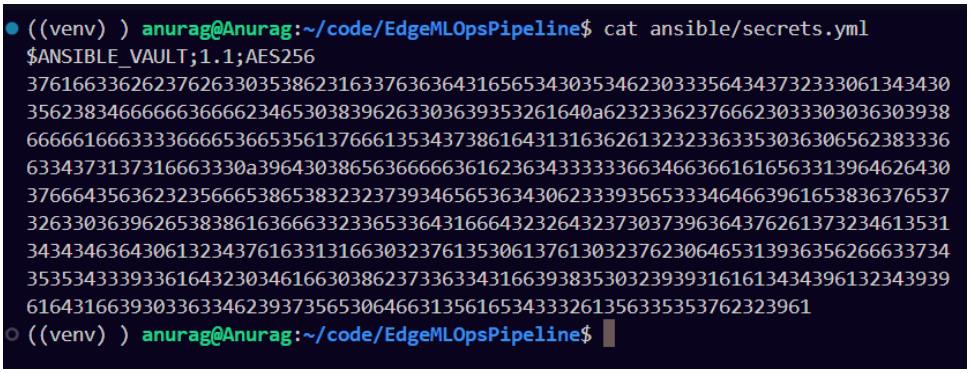

Figure 2: Ansible Vault encryption verification.

The decryption process adapts to the environment to ensure security:

- **Local Deployment:** When running `make up`, the user is prompted interactively to enter the Vault password.
- **CI/CD Pipeline:** In Jenkins, the password is never exposed in logs. It is stored in the **Jenkins Secure Credentials Store** (ID: `vault-pass-secret`) and injected dynamically into the Ansible command during the deploy stage.

3.3 Scalability: Horizontal Pod Autoscaling (HPA)

To ensure High Availability (HA), we implemented a Kubernetes HPA resource. The system monitors the CPU usage of the Inference Service. If usage exceeds 50%, Kubernetes automatically spawns additional pods (up to 3) to handle the load.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: spe-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: spe-app
  minReplicas: 1
  maxReplicas: 3
  targetCPUUtilizationPercentage: 50
```

Listing 2: HPA Configuration (k8s/hpa.yaml)

4 Innovative Solutions

In developing this pipeline, we implemented specific architectural patterns to solve challenges inherent to Edge AI and collaborative DevOps environments.

4.1 Autonomous Self-Healing Edge AI

We moved beyond standard static deployments by automating the role of the data scientist within the infrastructure itself. The application acts as an intelligent agent capable of maintaining its own performance.

- **Real-time Drift Detection:** Instead of relying on delayed batch analysis, the inference container calculates the Rolling Root Mean Squared Error (RMSE) on incoming sensor data in real-time. This allows immediate detection of performance degradation.
- **Autonomous Retraining:** Unlike standard monitoring which simply alerts a human, our system acts immediately. If the RMSE breaches the defined threshold, the edge container autonomously triggers a subprocess to retrain the model using the most recent dataset accumulated in the shared volume.
- **Zero-Touch Hot Reloading:** Once the new model artifact is generated, the inference engine detects the file change and hot-reloads the model into memory without crashing the service or requiring a container restart. This creates a closed-loop system capable of maintaining accuracy in volatile IoT environments without manual intervention.

4.2 Dynamic Multi-User CI/CD Architecture

A major challenge in shared DevOps repositories is managing conflicting container registry credentials and environment-specific configurations (e.g., Image Names). We devised a **Dynamic Injection Strategy** to resolve this:

- We replaced hardcoded image references in Kubernetes manifests with a placeholder variable (`DOCKER_USER_PLACEHOLDER`).
- We configured the Jenkins Pipeline to read Global Properties defined at the system level (`MY_DOCKER_USER`, `MY_EMAIL`).
- During deployment, the pipeline dynamically injects the specific developer's registry details into the manifests using stream editing (`sed`) before applying them to the cluster.

This innovative approach allows multiple developers to work on the exact same codebase and Jenkinsfile simultaneously, while deploying to their own private registries and receiving personalized notifications without a single git merge conflict.

5 Testing Strategy & Quality Assurance

To ensure the reliability of the automated pipeline, we implemented a robust testing framework that runs automatically within the Jenkins CI/CD process. This "Shift-Left" testing approach ensures that logical errors are caught early, while security vulnerabilities are logged for review.

5.1 Evolution of Testing Strategy

Initially, the pipeline utilized a basic `Automated Test` stage which only performed syntax validation using `py_compile`. While this caught syntax errors, it did not verify the logical correctness of the MLOps components. We replaced this with an advanced **Robust Testing** stage that executes a comprehensive suite of Unit Tests and Security Scans.

5.2 Unit & Integration Testing (Pytest)

We utilized the **Pytest** framework to validate the core logic of the MLOps pipeline. The tests are defined in `tests/test_core.py` and cover the following critical components:

- **Mathematical Logic Verification:** We wrote unit tests for the `calculate_rolling_rmse()` function. By injecting known values into a mocked prediction buffer, we verified that the Root Mean Square Error calculation accurately quantifies model drift.
- **Feature Engineering Checks:** The training script relies on creating "Lag Features" (historical time-steps). We implemented tests to verify that the `create_lag_features()` function correctly shifts the Pandas DataFrame without causing data leakage.
- **Mock Integration Testing (Drift Trigger):** Testing the "Self-Healing" capability usually requires waiting for real-time data drift. To validate this during CI/CD, we used the `unittest.mock` library:
  - We simulated an "Edge Case" where the sensor sends erratic data (Threshold + 1000).
  - We mocked the internal state of the inference engine.
  - **Result:** The test verifies that the system correctly identifies the drift and attempts to trigger the `cloud/train.py` subprocess, validating the self-healing mechanism without running a heavy training job.

5.3 Security Scanning (DevSecOps)

To satisfy the "Robustness" requirement, we integrated security scanning tools directly into the pipeline.

- **Bandit (SAST):** Scans Python source code for security flaws like hardcoded passwords or unsafe deserialization.
- **Safety (Dependency Scanning):** Checks `requirements.txt` against known CVE databases to ensure no compromised libraries are installed.

5.4 Pipeline Integration & Optimization

The tests are orchestrated via the **Robust Testing** stage using a hybrid quality gate approach. To optimize build time, all tests were run inside a **single Docker container instance**.

- **Hard Quality Gate (Unit Tests):** The `pytest` command runs without error suppression. If logic tests fail, the pipeline **aborts immediately**.
- **Soft Quality Gate (Security):** Security tools run with the `|| true` flag. Vulnerabilities are logged for developer review in the Jenkins console, but do not block the build during the development phase.

6 Results and Verification

6.1 CI/CD Automation

The following screenshot demonstrates the successful execution of the Jenkins Pipeline, triggered by a GitHub Webhook. All stages (Checkout, Build, Test, Push, Deploy, Post Actions) passed successfully. It also automatically sends an email indicating Build Success/Failure on completion.

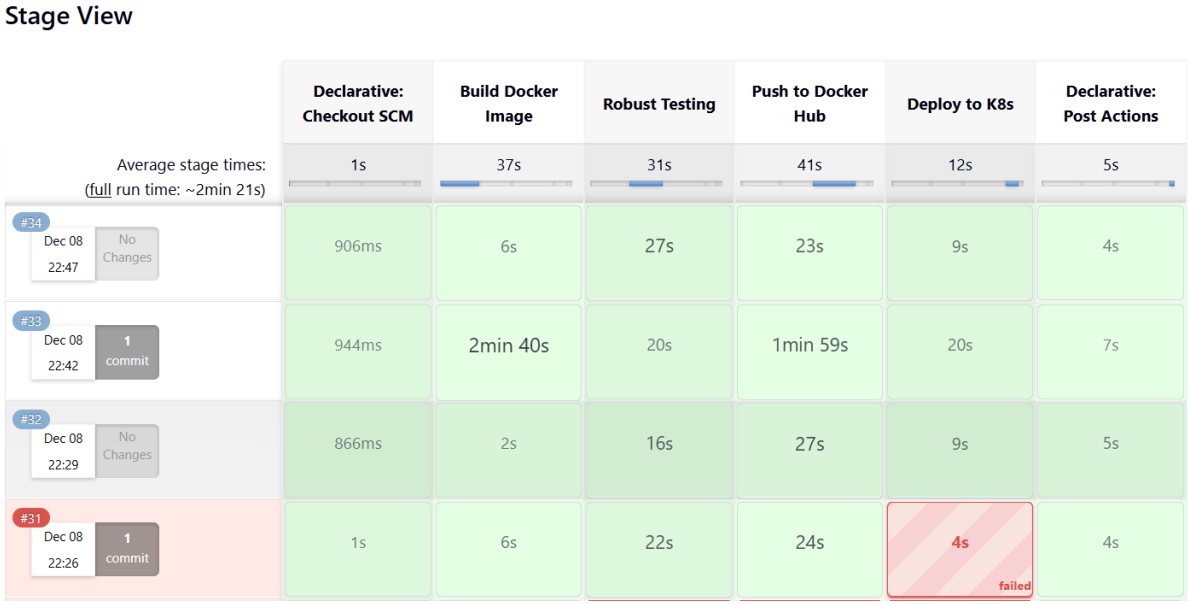

Figure 3: Jenkins Pipeline Stage View showing automated deployment.

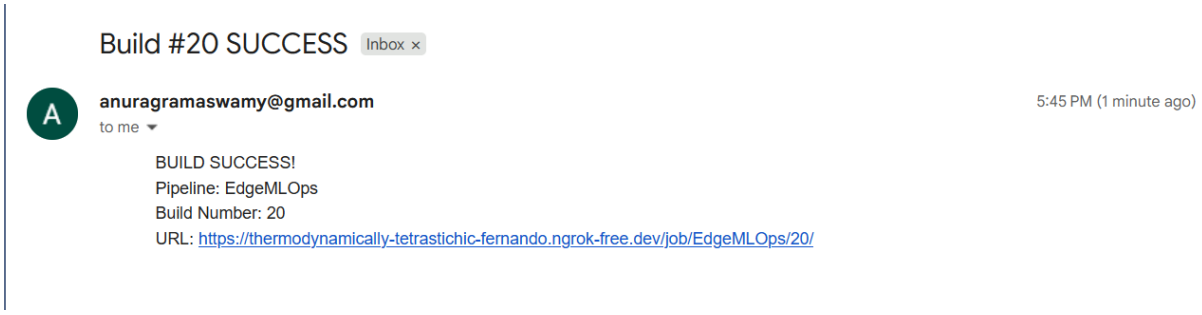

Figure 4: Automated email received after Build Success

6.2 Scalability Verification

The terminal output below confirms that the Kubernetes Horizontal Pod Autoscaler (HPA) is active and monitoring the target deployment.

```
((venv) ) anurag@Anurag:~/code/EdgeMLOpsPipeline$ kubectl get hpa
NAME          REFERENCE              TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
spe-app-hpa   Deployment/spe-app     cpu: 1%/50%   1         3         1          4h19m
```

Figure 5: Kubernetes HPA Status output showing target percentages.

6.3 Application Status (Dashboard)

The Streamlit dashboard verifies that the application is running, processing data, and using a valid model version.

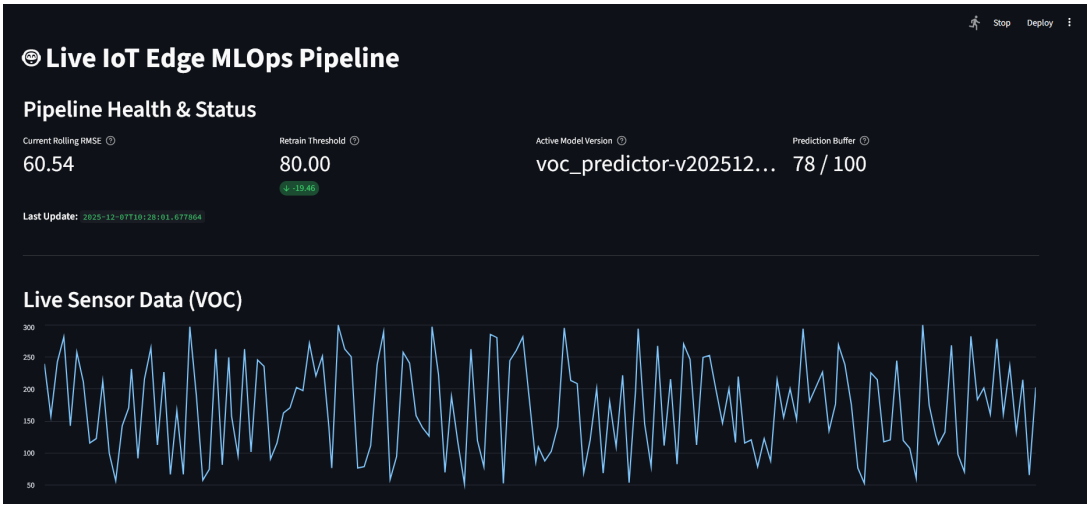

Figure 6: Real-time Dashboard showing active model and sensor data.

Testing Scenarios:
To validate the system's responsiveness, we tested with varying drift thresholds:

1. **Rapid Retraining (Threshold = 50.0):** We lowered the threshold to force the system into a "High Sensitivity" mode. This resulted in the model updating every 10-15 seconds, proving the retraining trigger mechanism works under stress. The dashboard showed "Calculating..." as the buffer reset frequently (1/100 samples).

2. **Stable Operation (Threshold = 75.0):** We adjusted the threshold to a realistic operating value. This allowed the system to stabilize, accumulate a full prediction buffer (100/100 samples), and display real-time RMSE calculations while maintaining the active model version.

6.4 Monitoring Infrastructure (ELK Stack)

We successfully deployed the ELK stack. Logstash captures MQTT messages, which are indexed in Elasticsearch and visualized in Kibana.

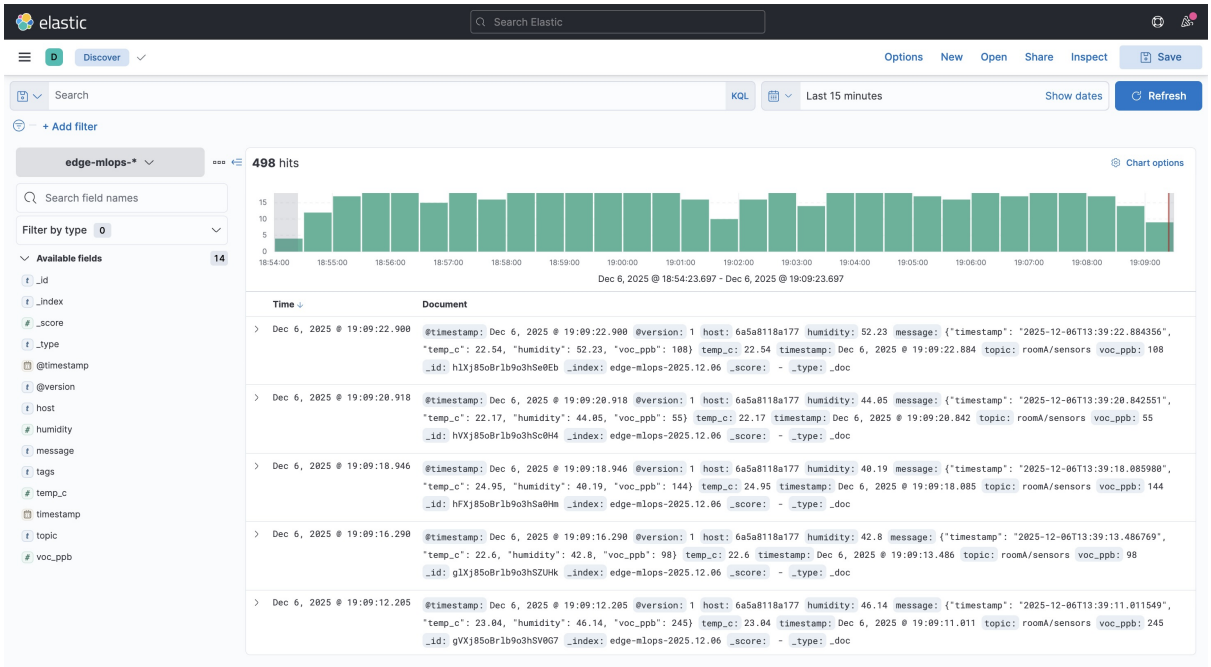

Figure 7: Kibana Dashboard visualizing live MQTT logs via Logstash.

6.5 MLOps Experiment Tracking

MLflow tracks the automated training runs triggered by our self-healing logic.

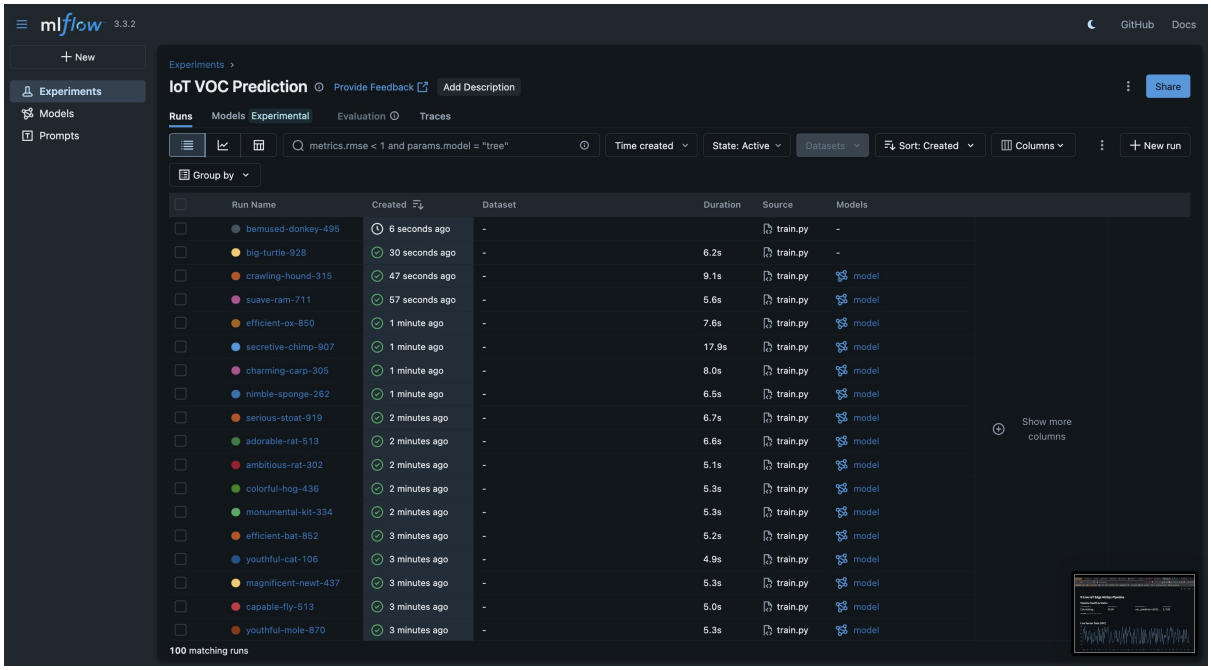

Figure 8: MLflow UI showing history of automated training runs.

7 Challenges Faced

During the implementation of this project, we encountered and overcame several technical challenges:

1. **Resource Constraints on Minikube:** Running the full ELK Stack alongside Jenkins and MLflow triggered memory pressure evictions on the host machine. We resolved this by enforcing strict resource limits in the Kubernetes manifests and increasing the Minikube VM allocation to 6GB. Additionally, we had to manually configure the kernel parameter `vm.max_map_count` to allow Elasticsearch to bootstrap correctly.

2. **Ansible Python Interpreter Conflicts:** Ansible struggled to switch between the system Python and the virtual environment Python, causing module errors. We fixed this by enforcing the python interpreter path in the Makefile variable definitions.

3. **Decoupling Log Ingestion (Logstash):** Initially, we attempted to use the embedded Logstash instance running inside the ELK container. However, adding the required MQTT input plugins to the pre-packaged image caused repeated container crashes and configuration conflicts.

   To solve this, we decoupled the architecture by building a custom, lightweight Logstash Docker image. We deployed this as a separate microservice (`logstash.yaml`) which buffers logs independently before forwarding them to the Elasticsearch database.

4. **Multi-User Collaboration & Registry Conflicts:** Since the two of us were working on the same repository, hardcoding Docker Hub credentials and Email addresses in the `Jenkinsfile` caused conflicts. One's push would attempt to login to the other's Docker registry, causing permission failures.

   To solve this, we implemented a **Dynamic Configuration Injection** strategy:

   - We replaced hardcoded values in Kubernetes manifests with a placeholder (`DOCKER_USER_PLACEHOLDER`).
   - In **Jenkins**, we utilized Global Properties (`MY_DOCKER_USER`, `MY_EMAIL`) to inject specific user details at runtime.
   - Locally, we updated the `Makefile` to read from a non-versioned `.env` file, using 'sed' commands to dynamically swap the image names during deployment and revert them immediately after.

   This allowed both developers to use the exact same codebase while pushing to their respective registries and receiving personal email notifications.

5. **Jenkins & Minikube Connectivity:** Since Minikube assigns dynamic IP/Ports on restart, Jenkins 'kubeconfig' would often break. We automated the synchronization of the config file using a Makefile target 'make fix-jenkins' to ensure seamless connectivity.

8 Conclusion

- This project successfully delivers a robust, production-ready DevOps pipeline tailored for the Edge AI domain. We have moved beyond simple containerization to build a fully orchestrated ecosystem that is secure, modular, and self-healing.
- By strictly adhering to Software Production Engineering principles, we implemented modular infrastructure code (Ansible Roles), encrypted sensitive configuration (Vault), and ensured high availability under load (Kubernetes HPA). The successful integration of these tools into a unified automation pipeline demonstrates a complete modernization of the software delivery lifecycle.
- The complete source code of the project is available in the GitHub repository along with the instructions to run it in the `README.md` file.