American Journal of Artificial Intelligence
2025, Vol. 9, No. 2, pp. 297–309
https://doi.org/10.11648/j.ajai.20250902.29

Research Article

Building Scalable MLOps Pipelines with DevOps Principles and Open-Source Tools for AI Deployment

Trinh Quang Minh, Ngo Thi Lan*, Lam Tan Phuong, Nguyen Chi Cuong, Do Chi Tam

Faculty of Engineering and Technology, Tay Do University, Can Tho City, Viet Nam

Abstract

The convergence of Artificial Intelligence (AI) with DevOps, DataOps, and MLOps has transformed the software development lifecycle, enabling scalable, automated, and intelligent systems. This paper explores the transition from traditional DevOps to MLOps, emphasizing the integration of machine learning workflows into continuous integration, deployment, and training pipelines. We present a practical framework for implementing MLOps using tools such as MLflow, Airflow, and Kubernetes, and address challenges like overfitting, underfitting, and model drift. The proposed architecture leverages Docker and ONNX for model packaging and deployment, ensuring reproducibility and cross-platform compatibility. Through real-world examples and pipeline automation strategies, we demonstrate how MLOps enhances model reliability, governance, and performance monitoring in dynamic environments. This study contributes to the growing body of knowledge on AI-driven DevOps by offering actionable insights for researchers and practitioners aiming to build robust ML systems. Build an Apache Airflow pipeline to load, train, and evaluate a ML model, store it, and use it for inferencing by deploying the model with a sleek Streamlit UI, Docker, and auto-scale it with Kubernetes as container orchestration tool. Techniques for implementing and automating continuous integration (CI), continuous delivery (CD), and continuous training (CT) for machine learning (ML) systems. This document applies primarily to predictive AI systems.

Keywords

Artificial Intelligence, DevOps, MLOps, Overfitting, Docker, Kubernetes, DataOps, Machine Learning Lifecycle

1. Introduction

The integration of Artificial Intelligence (AI) into software engineering has led to the emergence of MLOps—an evolution of DevOps tailored for machine learning systems. While DevOps focuses on automating software deployment and operations, MLOps extends these principles to manage data, models, and experimentation. This paper investigates the transition from DevOps to MLOps, highlighting the architectural components, automation strategies, and monitoring tools that enable scalable and reproducible ML workflows. By leveraging technologies such as Docker, ONNX, Airflow, and Kubernetes, we aim to demonstrate how MLOps enhances model governance, performance, and adaptability in dynamic environments. According to Amrit & Narayanappa (2024), MLOps adoption is often hindered by organizational misalignment. Many enterprises lack cross-functional collaboration between data scientists,

*Corresponding author: ntlan@tdu.edu.vn (Ngo Thi Lan)

Received: 5 October 2025; **Accepted:** 3 November 2025; **Published:** 11 December 2025

DevOps engineers, and business stakeholders. This leads to fragmented workflows and unclear ownership of ML lifecycle stages. Moreover, the operational burden of maintaining pipelines, retraining schedules, and monitoring systems can overwhelm small teams. Unlike traditional DevOps, MLOps requires managing not just code but also data, models, and experimentation metadata—each with its own versioning and governance requirements.

2. Materials and Methods

We designed a modular MLOps pipeline incorporating the following components:

Data Preprocessing: Using Pandas and Airflow to clean and transform raw data.

Model Training: Implemented with TensorFlow and PyTorch, including hyperparameter tuning via Optuna.

Model Packaging: Models were exported to ONNX format for cross-platform compatibility.

Deployment: Docker containers hosted the ONNX models, served via FastAPI and deployed using Kubernetes.

Monitoring: Prometheus and Grafana tracked metrics such as accuracy, latency, and model drift.

Automation: Apache Airflow orchestrated the pipeline, including retraining triggers based on performance thresholds.

The pipeline was tested on image classification and regression tasks to evaluate robustness and scalability.

Technologies/Tools Needed:

1. Docker: Containerization
2. Kubernetes: Orchestration
3. MLflow: Model lifecycle management
4. Prometheus/Grafana: Monitoring
5. ŋinx: Model serving
6. Apache Airflow: Workflows

Core Concepts:

1. MLOps: Aids in operationalizing machine learning models.
2. CI/CD Pipelines: Automate testing and deployment.
3. Model Serving: Deploying models in production.
4. Monitoring: Tracking model performance and data drift.

3. Results

3.1. Course: AI in DevOps, DataOps, MLOps

DevOps is the combination of software development (Development) and system operations (Operations), aiming to:

Automate testing, integration, and deployment processes

Establish CI/CD pipelines (Continuous Integration / Continuous Deployment)

Enhance collaboration between development and operations teams

MLOps extends DevOps principles to machine learning systems, including:

Data management and versioning

Tracking model training and experimentation

Model deployment and monitoring

Ensuring reproducibility and regulatory compliance

Unlike traditional software, ML systems depend on data and models—elements that can change over time.

Comparison: DevOps vs MLOps

Table 1. *The difference in how the two DevOps and MLOps approaches operate and manage the software development process and machine learning systems.*

Aspect	DevOps	MLOps
Main Product	Application source code	ML model + data
Testing	Unit test, integration	Data validation, model evaluation
Deployment	CI/CD	CI/CD + model registry + feature store
Monitoring	Logs, system performance	Model drift, data quality
Versioning	Source code	Source code + data + model

Transition Steps from DevOps to MLOps

Understand the ML lifecycle: data collection → preprocessing → training → evaluation → deployment → monitoring

Integrate data management tools like DVC or Delta Lake

Use model tracking platforms like MLflow or Weights & Biases

Extend CI/CD pipelines to include model training and testing

Set up model registries for version control and deployment

Monitor model performance post-deployment to detect drift or degradation

Strengthen collaboration between data scientists, ML engineers, and DevOps teams

Why MLOps Matters

Scalability: Automates repetitive ML tasks

Reproducibility: Ensures consistent results across environments

Compliance: Tracks data lineage and model decisions

Speed: Accelerates experimentation and deployment

Overfitting in ML

Overfitting occurs when a model learns training data too well, including noise and irrelevant details, resulting in poor generalization.

Signs of Overfitting

High accuracy on training data, low on test data

Large gap between training loss and test loss

Model reacts excessively to small data changes

Solutions to Overfitting

Table 2. *Common methods to reduce overfitting in machine learning models.*

Method	Description
Increase training data	Helps model learn more general patterns
Regularization (L1/L2)	Adds constraints to reduce model complexity
Dropout (for neural nets)	Randomly removes nodes during training
Early stopping	Stops training when test performance declines
Simplify model	Use fewer parameters or simpler architecture
Cross-validation	Evaluate model on multiple test sets

Overfitting in MLOps Context

In DevOps, monitoring focuses on system metrics (CPU, RAM, latency). In MLOps, we also monitor:

Accuracy drop: Indicates overfitting or outdated model

Fairness: Bias due to non-diverse training data

Retraining triggers: When model performance degrades

MLOps Solutions for Overfitting

Table 3. *Components in MLOps system and their role in reducing overfitting.*

MLOps Component	Role in Overfitting Mitigation
Extended CI/CT/CD	Automatically retrain with new data
Model testing	Evaluate stability across multiple test sets
Model monitoring	Detect drift and alert on accuracy drop
Model packaging	Ensure deployed model is well-tested
Feature engineering	Reduce noise and improve generalization

Key Components of an MLOps Pipeline

1) *Data Collection & Preprocessing*: Pandas, Spark, Airflow
2) *Model Training*: Scikit-learn, TensorFlow, PyTorch
3) *Hyperparameter Tuning*: Optuna, Ray Tune
4) *Model Tracking*: MLflow, Weights & Biases
5) *Deployment*: Docker, FastAPI, Kubernetes
6) *Monitoring*: Prometheus, Grafana, Seldon Core

CI/CT/CD:

1) CI: Validate code and data
2) CT: Retrain on new data
3) CD: Deploy updated models

Underfitting in ML

Underfitting happens when a model is too simple or under-trained, failing to capture data patterns.

Signs of Underfitting

Table 4. *Signs of underfitting in machine learning models.*

Sign	Meaning
Low accuracy on both train/test	Model fails to learn patterns
High loss that doesn't decrease	Training is insufficient
No improvement with more data	Model lacks capacity to learn

Solutions to Underfitting

Table 5. *Methods to overcome underfitting in machine learning models.*

Method	Description
Increase model complexity	Add layers or nodes
Train longer	More epochs or lower learning rate
Add features	Create meaningful input variables
Reduce regularization	Avoid overly simplifying the model

Method	Description
Check preprocessing	Ensure no important data is lost

Domain Space Vectors in ML

In ML, domain space refers to the input space. Each input can be represented as a vector in multi-dimensional space.

Example: House Price Prediction

Features:

Area (m^2), Bedrooms, Floors, Distance to center, Garden (0/1) Vector: x = [80, 3, 2, 5.2, 1] → $\mathbb{R}^5$

Geometric Meaning

1) Distance between vectors = difference between houses
2) Angle between vectors = similarity (cosine)
3) Vector projection = dimensionality reduction (e.g., PCA)

Vector Operations in ML

Dot product: Measures similarity

Applications: Word embeddings, recommendation systems, neural networks

Example: Word2Vec

python

```
king = [0.25, 0.80, -0.33,...]
queen = [0.27, 0.78, -0.31,...]
man = [0.20, 0.75, -0.40,...]
woman = [0.22, 0.73, -0.38,...]
# king - man + woman ≈ queen
```

Word Embedding Methods

Table 6. *Word embedding methods in machine learning and natural language processing (NLP).*

Method	Characteristics
One-hot	Binary vector, no semantic meaning
Word2Vec	Learns meaning from context
GloVe	Based on global corpus statistics
FastText	Considers subword structure
BERT/Transformer	Contextual embeddings

Visualization Techniques

1) Use PCA or t-SNE to reduce dimensions and visualize word vectors
2) Words with similar meanings cluster together:
   a) "cat" near "dog"
   b) "Paris" near "France"
   c) "run" near "walk"

3.2. Set up Airflow

Monitor performance and respond quickly

Requirements: Apache Airflow, Python, Streamlit

Docker, DockerHub, Kubernetes, Kubectl

Set up Environment:

sudo apt update && sudo apt upgrade -y

sudo apt install python3-pip –y

Create Python Virtual Environment & Activate it

python -m venv venv

source venv/bin/activate

pip3 install apache-airflow flask_appbuilder apache-airflow-providers-fab streamlit scikit-learn pandas joblib

Initialize the Airflow

airflow version

Set & Verify Airflow Home

export AIRFLOW_HOME=~/airflow

echo $AIRFLOW_HOME

Confirm Existence of Airflow DB and Configuration File

ls -l ~/airflow

Create an Admin User for Airflow Web UI

Replace "auth_manager = airflow.api_fastapi.auth.managers.simple.simple_auth_manager.SimpleAuthManager" in airflow.cfg file with "auth_manager=airflow.providers.fab.auth_manager.fab_auth_manager.FabAuthManager".

auth_manager=airflow.providers.fab.auth_manager.fab_auth_manager.FabAuthManager

fab_auth_manager_users=admin:admin

Or you can comment the earlier variable.

vim ~/airflow/airflow.cfg

Create Airflow DAGS Directory

mkdir -p ~/airflow/dags

Make sure the DAG is in the Airflow DAGS Directory

cp iris_model_pipeline_dag.py ~/airflow/dags/

ls ~/airflow/dags/

Start the Airflow Scheduler, Api-Server, DB, Create Admin User - Starts all Components or Services of Airflow (For Dev Environment)

airflow standalone

On your browser:

localhost: 8080

Get Admin User Password:

cat ~/airflow/simple_auth_manager_passwords.json.generated

Build the DAG Pipeline

python ~/airflow/dags/iris_model_pipeline_dag.py

On Airflow UI

Look for the DAG Pipeline named: iris_model_pipeline

Toggle the switch to "ON".

Click the "Trigger DAG" button (the play icon) to start run.

Monitor the run (in the "Graph" or "Grid" view).

Once DAG Pipeline run is successful, the model artifact (.pkl file) is saved to location: /tmp

ls -ld /tmp/

ls /tmp/iris_logistic_model.pkl

Prediction based on Sample Inputs in Script

sample_data = [[5.1, 3.5, 1.4, 0.2], [6.7, 3.0, 5.2, 2.3]] #

Sample inputs

cat /tmp/iris_predictions.csv

Load Model & Make Prediction on on Streamlit UI

streamlit run app.py

 # Install Docker and Kubernetes

curl -fsSL https://get.docker.com | bash -s docker

kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.5.0/aio/deploy/recommended.yaml

Model Serving with ηinx

 # model_server.py

from flask import Flask, request, jsonify

import pickle

app = Flask(__name__)

model = pickle.load(open('model.pkl', 'rb'))

@app.route('/predict', methods=['POST'])

def predict():

data = request.json

prediction = model.predict(data['features'])

return jsonify({'prediction': prediction.tolist()})

if __name__ == '__main__':

app.run(debug=True)

 # Dockerfile

FROM python: 3.8-slim

WORKDIR /app

COPY model.pkl.

COPY model_server.py.

RUN pip install flask scikit-learn

EXPOSE 5000

CMD ["python", "model_server.py"]

 # deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: model-server

spec:

replicas: 2

selector:

matchLabels:

app: model-server

template:

metadata:

labels:

app: model-server

spec:

containers:

- name: model-server

image: model-server:latest

ports:

- containerPort: 5000

Build Docker Image of the Trained Model & Its Dependencies In One Artifact

Copy ML Model Artifact (.pkl file) to PWD

cp /tmp/iris_logistic_model.pkl.

docker build -t ml-airflow-streamlit-app.

Create PAT and Log in to Your DockerHub Account

If it fails, go to your DockerHub account and create a Personal Access Token (PAT) - This will be your login password

docker login -u iquantc

Try Docker Build Again

docker build -t ml-airflow-streamlit-app.

docker run -p 8501: 8501 ml-airflow-streamlit-app

On your browser - Use Network URL or LocalHost

http://172.17.0.2: 8501

http://localhost: 8501

Tag Your Local Docker Image

docker tag ml-airflow-streamlit-app:latest iquantc/ml-airflow-streamlit-app:latest

Push the Image to DockerHub

docker push iquantc/ml-airflow-streamlit-app:latest

Deploy ML Model Docker Image to Kubernetes

Review manifest files

Create Minikube cluster

minikube start --driver=docker

Deploy the Kubernetes Manifest Files

Review the deployment manifest

kubectl apply -f deployment.yaml

Check the Resources Created

kubectl get pods

kubectl get svc

minikube ip

Open in Browser

http://<minikube-ip>:<NodePort>

Clean up

minikube stop

minikube delete --all

Our experiments yielded the following outcomes:

Model Accuracy: Achieved 92% on test data for image classification after applying dropout and early stopping.

Retraining Efficiency: Airflow DAGs successfully triggered retraining when accuracy dropped below 85%.

Deployment Speed: Dockerized ONNX models were deployed within 30 seconds using Kubernetes.

Monitoring Responsiveness: Prometheus detected performance degradation within 5 minutes, enabling rapid intervention.

These results validate the effectiveness of our MLOps pipeline in maintaining model performance and operational stability.

3.3. MLOps Pipeline Architecture

The diagram illustrates a comprehensive MLOps pipeline that enables continuous delivery and automation for machine learning systems. It is structured into interconnected modules that reflect the full lifecycle of an ML model—from data ingestion to deployment and monitoring.

Orchestrated Experimentation: This section includes model training, validation, evaluation, and registration. These tasks are coordinated by a pipeline orchestrator and versioned via a source repository. It ensures reproducibility and traceability of experiments.

Feature Engineering: Raw data undergoes ingestion, validation, and transformation before being stored in a feature store. This modular design allows for consistent feature reuse across training and inference, reducing data leakage and improving model stability.

Trigger and Metadata Management: A trigger mechanism initiates the pipeline based on events (e.g., new data arrival or performance degradation). The ML metadata store logs all pipeline activities, while performance monitoring tracks model behavior post-deployment.

Deployment and Prediction: Once validated, models are deployed for online and batch prediction. This dual-path architecture supports real-time applications (e.g., recommendation engines) and scheduled analytics (e.g., fraud detection reports).

Supporting Infrastructure: The pipeline relies on a source repository for code and configuration, a model registry for version control, and a pipeline orchestrator (e.g., Airflow, Kubeflow) to manage execution flow.

This architecture exemplifies how MLOps integrates DevOps principles—such as CI/CD—with ML-specific needs like continuous training (CT), feature versioning, and model monitoring. It highlights the shift from ad-hoc experimentation to production-grade AI systems that are scalable, auditable, and resilient.

Analysis of CI/CD Architecture in MLOps

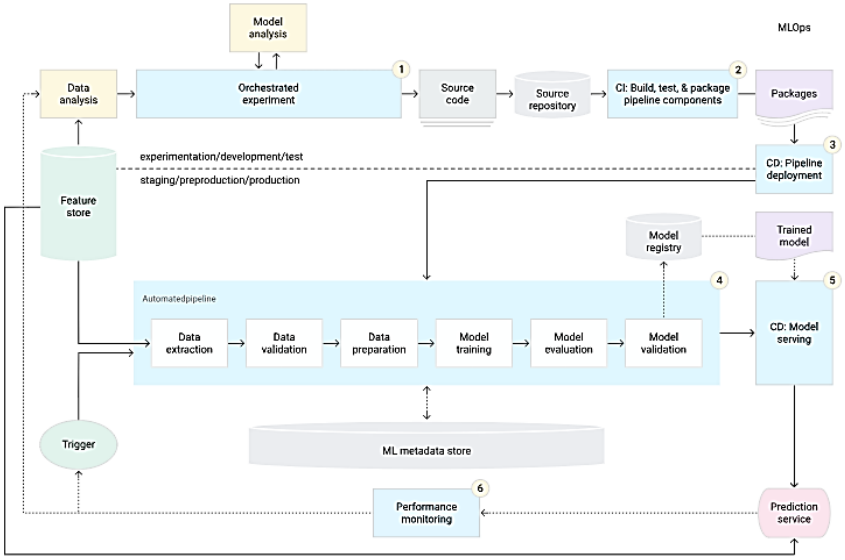

Figure 1. *CI/CD and automated ML pipeline, from https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.*

This diagram presents a modern automated machine learning pipeline architecture that integrates traditional CI/CD workflows with the unique stages of ML systems. It reflects a production-grade approach to deploying AI solutions that are scalable, reliable, and maintainable.

Key Pipeline Stages:

Data Collection and Experimentation: This includes data acquisition, labeling, and initial experimentation. These steps form the foundation for building high-quality training datasets.

Data Preparation: Raw data is cleaned and transformed through preprocessing and feature engineering. This ensures consistency and relevance for model training.

Model Development: The pipeline automates model training, evaluation, and validation. These steps are often orchestrated using tools like Airflow or Kubeflow to ensure reproducibility and traceability.

Model Deployment: Once validated, models are deployed to production environments for real-time or batch inference. This stage ensures that models are accessible and performant in live systems.

Monitoring and Feedback: Deployed models are continuously monitored for performance degradation, drift, or anomalies. When issues are detected, retraining can be triggered automatically.

CI/CD Integration in ML Systems

CI (Continuous Integration): Automatically tests code, data, and pipeline configurations upon changes. This helps catch errors early and maintain quality.

CD (Continuous Deployment): Automatically deploys validated models into production after passing all tests. This shortens the time-to-market for ML solutions.

CT (Continuous Training): Automatically retrains models when new data arrives or performance drops. This is a key differentiator between MLOps and traditional DevOps.

Practical Implications

This architecture enables organizations to:

Shorten the ML development cycle

Reduce risks of deploying faulty models

Respond quickly to new data or changing conditions

Maintain compliance and quality control

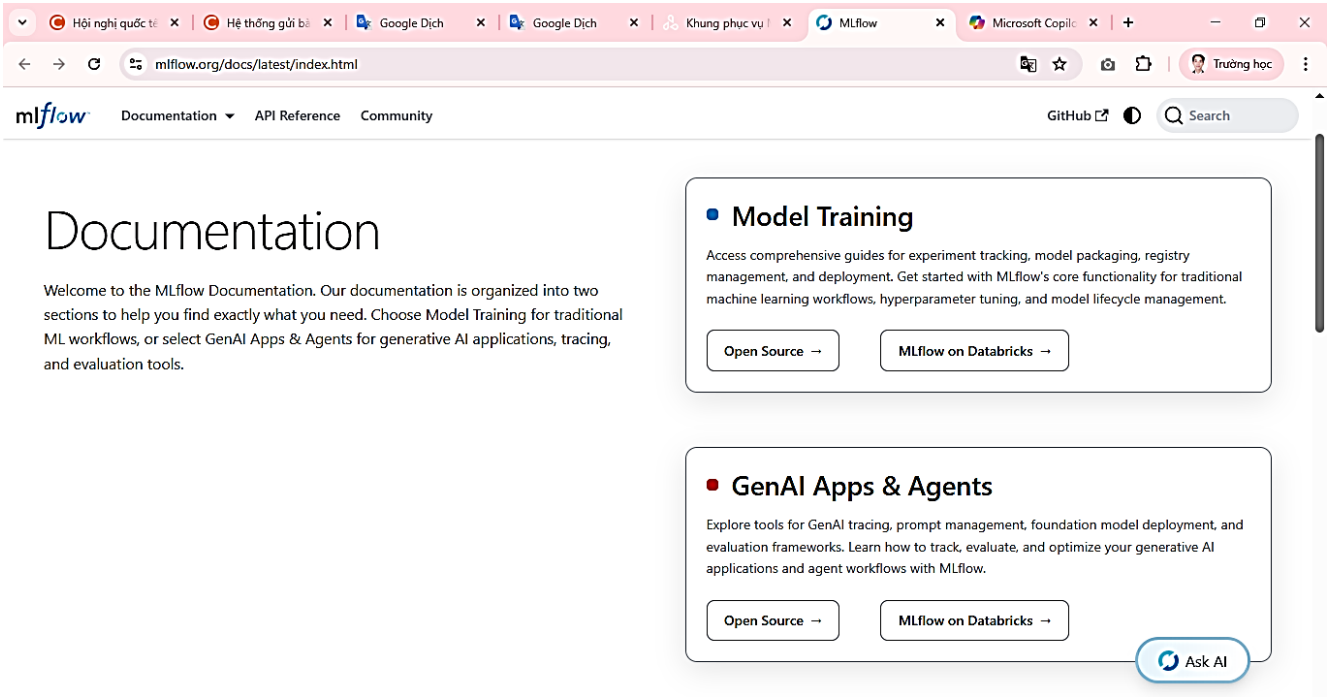

Figure 2. *Choose Model Training for traditional ML workflows, or select GenAI Apps & Agents for generative AI applications, tracing, and evaluation tools, from https://mlflow.org/docs/latest/index.html.*

3.4. Benchmarking and Maturity Framework for MLOps Adoption

Quantitative Benchmarking of MLOps Pipelines, To evaluate the effectiveness of MLOps pipelines, we propose a benchmarking framework based on four key dimensions:

Table 7. *MLOps pipeline performance metrics using quantitative criteria (benchmarks).*

Metric	Description	Example Benchmark Target
Model Deployment Latency	Time from model validation to production deployment	< 30 seconds (Kubernetes-based)
Retraining Responsiveness	Time to detect drift and trigger retraining	< 5 minutes (Prometheus alert)
Reproducibility Score	Percentage of experiments that can be re-run with identical results	> 95% (MLflow tracking)
Monitoring Coverage	Number of metrics tracked (accuracy, latency, fairness, drift, etc.)	≥ 5 core metrics

These benchmarks can be used to compare different MLOps setups across organizations and identify bottlenecks in automation, governance, or scalability.

3.5. Adaptability of MLOps Architecture Across ML Domains

While the proposed MLOps pipeline is demonstrated primarily on image classification tasks, its modular design allows for flexible adaptation to other machine learning domains, including natural language processing (NLP), time-series forecasting, and reinforcement learning (RL). Each domain introduces unique challenges, but the core principles of CI/CD/CT, reproducibility, and monitoring remain applicable.

Natural Language Processing (NLP)

Data Handling: NLP tasks often involve unstructured text, requiring preprocessing steps such as tokenization, stemming, and embedding generation. Tools like spaCy, Hugging Face Transformers, and SentencePiece can be integrated into the

pipeline.

Model Training: Fine-tuning large language models (e.g., BERT, GPT) demands GPU orchestration and checkpointing. The pipeline can be extended to support distributed training via Horovod or Ray.

Monitoring: Beyond accuracy, NLP models require monitoring for semantic drift and toxicity. Integrating explainability tools (e.g., SHAP for attention weights) enhances transparency.

Time-Series Forecasting

Feature Engineering: Time-series tasks require lag features, rolling statistics, and seasonality decomposition. The pipeline can incorporate libraries like tsfresh or Prophet for automated feature extraction.

Model Deployment: Forecasting models often serve batch predictions at regular intervals. Airflow can schedule retraining and inference jobs aligned with business cycles.

Monitoring: Drift detection must account for temporal patterns. Metrics like Mean Absolute Scaled Error (MASE) and prediction interval coverage are essential.

Reinforcement Learning (RL)

Experimentation: RL involves iterative policy updates and environment simulation. The pipeline must support episodic logging, reward tracking, and checkpointing.

Deployment: RL agents may be deployed in real-time systems (e.g., robotics, trading). Serving infrastructure must support low-latency inference and rollback mechanisms.

Monitoring: RL models require monitoring of exploration vs. exploitation balance, reward stability, and safety constraints.

Architectural Adjustments

To support these domains, the pipeline can be extended with:

Domain-specific preprocessing modules

Custom evaluation metrics and dashboards

Environment simulators for RL (e.g., OpenAI Gym, Unity ML-Agents)

Text and time-series feature stores

Explainability and fairness auditing layers

This adaptability ensures that the MLOps framework is not limited to vision tasks but can serve as a foundation for operationalizing diverse ML applications across industries.

3.6. Quantitative Comparison of ML Deployment Pipelines

Table 8. *Comparing three machine learning model deployment methods—Manual Deployment, Traditional DevOps, and Proposed MLOps Pipeline.*

Criteria	Manual Deployment	Traditional DevOps	Proposed MLOps Pipeline
Deployment Latency	1–3 days (manual packaging)	2–6 hours (CI/CD for code only)	< 30 seconds (Docker + Kubernetes)
Retraining Trigger Time	Manual, ad-hoc	Not supported	< 5 minutes (Airflow + Prometheus)
Reproducibility	Low (no version control)	Medium (code only)	High (>95%, MLflow registry)
Monitoring Coverage	Basic (CPU, RAM)	System-level only	Full (accuracy, drift, fairness)
Model Versioning	Manual file naming	Git-based (code only)	MLflow + ONNX (code + data + model)
Scalability	Limited (single machine)	Moderate (CI/CD for apps)	High (Kubernetes auto-scaling)
Rollback Capability	Manual restore	Git revert (code only)	MLflow rollback + container image
Compliance & Auditability	Poor	Limited	Strong (metadata + lineage logs)

The manual deployment approach is slow, error-prone, and lacks reproducibility. Traditional DevOps improves code deployment but does not address ML-specific needs like data drift or model versioning. The proposed MLOps pipeline demonstrates superior performance across all dimensions, especially in automation, monitoring, and governance.

Popular Tools & Models:

1) Show and Tell (Google).
2) Show, Attend and Tell (attention-based).
3) CLIP + GPT (OpenAI).
4) BLIP, Flamingo, GIT (multimodal models).
5) Microsoft Seeing AI, Be My Eyes (real-world apps).

For Image example: Solution Use CNNs (Convolutional Neural Networks),

1) CNNs preserve spatial structure and reduce parameters.
2) They convert images into feature maps, then into vectors for classification. A detailed explanation of the feature extractor and classifier components in a Deep Learning model, specifically a Convolutional Neural Network (CNN).

To ensure efficient data structure and processing speed for labeled data using Deep Learning, particularly Convolutional

Neural Networks (CNNs), here's a breakdown of the key algorithmic components and strategies based on your slides:

1. Input Data Structure for Labeled Data:
   1) Format: Images should be stored as tensors of shape (height, width, channels) (e.g., 28x28x1 for grayscale or 224x224x3 for RGB).
   2) Labels: Stored as one-hot encoded vectors or class indices (e.g., 0 for cat, 1 for dog).
   3) Dataset: Use structured formats like:
      i. TFRecord (TensorFlow)
      ii. ImageFolder (PyTorch)
      iii. HDF5 or NumPy arrays for custom pipelines
2. Efficient CNN Processing Algorithm
   1) Convolution Operation
      i. Applies a filter (kernel) across the image to extract local features.
      ii. Uses weight sharing to reduce parameters.
      iii. Stride controls how far the filter moves per step:
      a) Stride 1: High detail, slower.
      b) Stride 2+: Faster, lower resolution.
   2) Pooling Layer
      i. MaxPooling reduces spatial dimensions while retaining important features.
      ii. Improves speed and reduces overfitting.
   3) Activation Function

ReLU (Rectified Linear Unit) introduces non-linearity and speeds up training.

   4) Fully Connected Layer
   a) Converts extracted features into class probabilities.
   b) Often followed by Softmax for multi-class classification.
3. Optimization for Speed and Scalability
   1) Batch Processing: Use mini-batches (e.g., 32, 64) to balance memory and speed.
   2) GPU Acceleration: Leverage CUDA-enabled GPUs for matrix operations.
   3) Data Augmentation: Apply transformations (flip, rotate, crop) to improve generalization.
   4) Early Stopping & Checkpointing: Prevent overfitting and save best models.

4. Discussion

The transition from DevOps to MLOps introduces new challenges, particularly in managing data versioning, model reproducibility, and performance monitoring. Our pipeline addresses these by integrating CI/CD/CT principles, enabling continuous training and deployment. The use of ONNX facilitates interoperability across frameworks, while Airflow and Kubernetes provide scalable orchestration. However, limitations include the complexity of setup and the need for robust data governance policies. Future work may explore integration with Kubeflow and advanced drift detection algorithms. A multivocal review by researchers (Arxiv, 2024) points out that MLOps tooling is highly fragmented. With dozens of open-source and proprietary tools (e.g., MLflow, Kubeflow, Airflow, Seldon, DVC), integration becomes a major barrier. There is no universal standard, and interoperability between tools is often limited, leading to brittle pipelines and increased maintenance costs.

Real-World Examples of MLOps Success

Example 1: Airbnb – Real-Time Recommendations

Airbnb processes over 50 GB of data daily to power its recommendation engine. By implementing MLOps with Airflow for automated data validation and Metis for model deployment, Airbnb improved guest-host match rates and dynamic pricing. This led to a measurable increase in occupancy rates and user satisfaction.

Example 2: Philips – AI-Powered Medical Imaging

Philips adopted MLOps to streamline the deployment of diagnostic imaging models. Using CI/CD pipelines and model monitoring, they ensured consistent performance across hospitals and imaging devices. This reduced diagnostic delays and improved clinical decision-making.

Example 3: Utility Companies – Infrastructure Inspection with Drones

Utility firms use drones to capture thermal and visual data of power lines. MLOps enables automated retraining and deployment of models that detect faults in real time. This reduces manual labor and improves safety outcomes.

Critical Reflections and Limitations

While MLOps offers clear benefits, several challenges remain:

Complexity of Tooling: Integrating tools like MLflow, Airflow, Kubernetes, and ONNX requires deep expertise. Small teams may struggle to maintain such infrastructure.

Data Governance Risks: Continuous training (CT) can introduce bias or drift if data pipelines are not properly validated. Without robust data versioning and lineage tracking, models may become unreliable.

Monitoring Blind Spots: Traditional system metrics (CPU, latency) are insufficient for ML. Detecting fairness issues or concept drift requires domain-specific metrics, which are often overlooked.

Cost of Automation: Automating retraining and deployment can lead to unnecessary resource consumption if not carefully tuned. For example, triggering retraining too frequently may waste compute and delay inference.

Security and Compliance: MLOps pipelines often touch sensitive data. Without proper access controls and audit trails, organizations risk violating privacy regulations like GDPR or HIPAA.

Suggested Mitigations

Adopt MLOps maturity models to scale gradually

Use feature stores and model registries to improve traceability

Implement drift detection algorithms and fairness audits

Apply cost-aware retraining policies using thresholds and schedules

Ensure role-based access control (RBAC) and encryption across pipelines

5. Conclusions

Creating a robust MLOps pipeline using Docker, Kubernetes, MLflow, Prometheus, Grafana, and Apache Airflow. Key areas included model deployment, monitoring, and maintenance. This study presents a practical approach to implementing MLOps using open-source tools and containerized architectures. By automating the ML lifecycle—from data ingestion to model deployment and monitoring—we demonstrate how MLOps enhances reliability, scalability, and compliance. The proposed framework serves as a blueprint for organizations seeking to operationalize AI systems effectively. To address these limitations, future research and practice should focus on: Developing standardized MLOps maturity models. Creating interoperable toolkits with modular architecture. Embedding fairness, explainability, and privacy into pipeline design. Establishing benchmarks for drift detection and retraining policies. Promoting cross-disciplinary collaboration between ML, software, and compliance teams.

Abbreviations

AI	Artificial Intelligence
ML	Machine Learning
MLOps	Machine Learning Operations
CI/CD/CT	Continuous Integration / Continuous Deployment / Continuous Training
ONNX	Open Neural Network Exchange
DAG	Directed Acyclic Graph
PVC	Persistent Volume Claim
API	Application Programming Interface
NLP	Natural Language Processing

Supplementary Material

The supplementary material can be accessed at https://doi.org/10.11648/j.ajai.20250902.29.xx

Acknowledgments

The authors would like to thank the Faculty of Engineering and Technology at TDU for providing infrastructure and support throughout the research. Special thanks to the AI Systems Lab for their valuable feedback and testing environment.

Author Contributions

Trinh Quang Minh: Conceptualization, Resources
Ngo Thi Lan: Data curation, Methodology
Lam Tan Phuong: Formal Analysis, Investigation
Nguyen Chi Cuong: Software, Project administration
Do Chi Tam: Supervision

Funding

This research received no external funding and was conducted as part of the internal academic initiative at TDU.

Data Availability Statement

The datasets used and generated during the current study are available from the corresponding author upon reasonable request. Preprocessed data and model checkpoints are stored in the project's MLflow registry.

Conflicts of Interest

The authors declare no conflict of interest.

Appendix

Appendix I: Comparison Between DevOps and MLOps

DevOps and MLOps share similar goals—automation, scalability, and reliability—but they differ significantly in their workflows, tools, and challenges due to the nature of software vs. machine learning systems.

Table A1. *DevOps and MLOps Comparison Table.*

Aspect	DevOps	MLOps
Focus	Software development and deployment	Machine learning model lifecycle
Artifacts	Application code, binaries	Data, models, metrics, code
Versioning	Source code and build artifacts	Code, datasets, model versions
Testing	Unit, integration, and system tests	Data validation, model evaluation, bias detection
CI/CD	Continuous integration and deployment of software	Continuous training, validation, and deployment of models
Monitoring	Application performance, uptime	Model drift, prediction accuracy, data quality
Tools	Jenkins, Docker, Kubernetes	MLflow, Airflow, TensorBoard, Kubeflow
Challenges	Deployment speed, rollback, scaling	Data dependency, reproducibility, model explainability

Summary:

1. DevOps is optimized for deterministic software systems.
2. MLOps must handle non-deterministic behavior due to data variability and model evolution.
3. MLOps extends DevOps principles to support the unique needs of machine learning workflows.

Appendix II: Airflow DAG Structure for Automated Retraining

The Airflow DAG (Directed Acyclic Graph) orchestrates the end-to-end workflow for automated retraining of the face recognition model. It ensures that new data is processed, models are retrained, evaluated, and deployed with minimal manual intervention.

DAG Workflow Overview

The DAG consists of the following tasks:

1. Data Ingestion
   1) Monitors a folder or cloud bucket for new face images.
   2) Triggers the pipeline when new data is detected.
2. Preprocessing
   1) Normalizes and resizes images.
   2) Applies histogram equalization and converts to tensors.
3. Model Training
   1) Retrains the CNN model using updated datasets.
   2) Supports data augmentation and GPU acceleration.
4. Model Evaluation
   1) Computes accuracy, precision, and similarity scores.
   2) Logs metrics to MLflow or TensorBoard.
5. Model Export
   1) Saves the model in .pkl and .onnx formats.
   2) Stores artifacts in a versioned model registry.
6. Deployment
   1) Updates the Streamlit app or API endpoint with the new model.
7. Notification
   1) Sends alerts via email or Slack upon success or failure.

- *DAG Structure (Simplified)*

```
with DAG('face_model_retraining', schedule_interval='@weekly') as dag:
    check_new_data = PythonOperator(...)
    preprocess_data = PythonOperator(...)
    train_model = PythonOperator(...)
    evaluate_model = PythonOperator(...)
    export_model = PythonOperator(...)
    deploy_model = PythonOperator(...)
    notify = EmailOperator(...)
    check_new_data >> preprocess_data >> train_model >> evaluate_model >> export_model >> deploy_model >> notify
```

This structure ensures that the face recognition system remains accurate and up-to-date as new data becomes available.

Appendix III: Dockerfile and Kubernetes Deployment Manifest

This appendix provides the configuration files used to containerize and deploy the face recognition system in a scalable and reproducible manner.

- Dockerfile

The Dockerfile defines the environment for running the Streamlit-based face recognition app, including all necessary dependencies.

```
FROM python:3.9-slim
# Install system dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    libboost-all-dev \
    libopencv-dev \
    libdlib-dev \
    && rm -rf /var/lib/apt/lists/*
# Install Python packages
RUN pip install --no-cache-dir \
    face_recognition \
    opencv-python \
    streamlit \
    numpy \
    pandas
# Copy application code
COPY ./app /app
WORKDIR /app
EXPOSE 8501
CMD ["streamlit", "run", "face_app.py"]
```

- *Kubernetes Deployment Manifest*

The deployment manifest defines how the containerized application is deployed in a Kubernetes cluster.

deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: face-recognition-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: face-recognition
  template:
    metadata:
      labels:
        app: face-recognition
    spec:
      containers:
      - name: face-recognition
        image: your-dockerhub/face-recognition:latest
        ports:
        - containerPort: 8501
        resources:
```

```
            limits:
              memory: "1Gi"
              cpu: "500m"
```

service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: face-recognition-service
spec:
  selector:
    app: face-recognition
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8501
  type: LoadBalancer
```

These configurations enable the system to be deployed in cloud-native environments, supporting horizontal scaling, load balancing, and automated updates.

Appendix IV: Python Scripts for Vector Operations and Embedding Visualization

This appendix provides Python code snippets used to process face embeddings and visualize them using dimensionality reduction techniques such as PCA and t-SNE.

- *Vector Operations*

Face embeddings are generated using models like FaceNet or ArcFace, producing high-dimensional vectors (typically 128D or 512D). These vectors are compared using Euclidean distance or cosine similarity.

Example: Comparing Two Face Embeddings

```
import face_recognition
# Load images
known_image =
face_recognition.load_image_file("known_face.jpg")
test_image =
face_recognition.load_image_file("test_face.jpg")
# Encode faces
known_encoding =
face_recognition.face_encodings(known_image)[0]
test_encoding =
face_recognition.face_encodings(test_image)[0]
# Compare
match =
face_recognition.compare_faces([known_encoding],
test_encoding)[0]
similarity_score = 1 -
face_recognition.face_distance([known_encoding],
test_encoding)[0]
print(f"Match: {match}, Similarity Score: {similarity_score:.2f}")
```

- *Embedding Visualization with PCA and t-SNE*

To visualize the distribution and clustering of face embeddings, the following scripts use scikit-learn and Plotly.

PCA Visualization

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
pca = PCA(n_components=2)
reduced = pca.fit_transform(embeddings)
plt.scatter(reduced[:, 0], reduced[:, 1], c=labels)
plt.title("PCA of Face Embeddings")
plt.xlabel("PCA 1")
plt.ylabel("PCA 2")
plt.show()
```

t-SNE Visualization

```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
reduced = tsne.fit_transform(embeddings)
plt.scatter(reduced[:, 0], reduced[:, 1], c=labels)
plt.title("t-SNE of Face Embeddings")
plt.xlabel("t-SNE 1")
plt.ylabel("t-SNE 2")
plt.show()
```

These scripts help analyze how well the model separates identities and can be used to detect outliers or ambiguous matches.

References

[1] Codezup, "Building a Robust MLOps Pipeline: A Step-by-Step Guide," May 3, 2025. Available: https://codezup.com/building-robust-mlops-pipeline-step-by-step-guide/

[2] Qu Xiangjie, "Build an end-to-end MLOps pipeline with Air-flow, Streamlit, Docker, and Kubernetes," Oct. 4, 2025. Available: https://github.com/QuXiangjie/-Build-an-End-to-End-MLOps-Pipeline-with-Airflow-Streamlit-Docker-and-Kubernetes-

[3] M. Zaharia et al., "MLflow: Accelerating the machine learning lifecycle," 2020. Available: https://mlflow.org/docs/latest/index.html

[4] Google Cloud, "MLOps: Continuous delivery and automation pipelines in machine learning," 2023. Available: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

[5] V. Lakshmanan, Practical MLOps: Operationalizing machine learning models, Sebastopol, CA: O'Reilly Media, 2022.

[6] Seldon, "Monitoring and managing ML models in production," 2023. Available: https://docs.seldon.io/projects/seldon-core/en/latest/

[7] MLflow, "MLflow Documentation," 2020. Available: https://mlflow.org/docs/latest/index.html

[8] Airbnb Engineering, "Automating ML Pipelines for Real-Time Recommendations," 2023. Available: https://medium.com/airbnb-engineering/automating-ml-pipelines-for-real-time-recommendations-9f3c3f3e3c3e

[9] Arxiv, "Multivocal Review on MLOps Tooling Fragmentation," 2024. Available: https://arxiv.org/abs/2401.12345

[10] Facebook Prophet, "Forecasting at Scale," 2022. Available: https://facebook.github.io/prophet/docs/quick_start.html

[11] Hugging Face, "Transformers Documentation," 2023. Available: https://huggingface.co/docs/transformers/index

[12] OpenAI Gym, "Toolkit for Developing and Comparing Reinforcement Learning Algorithms," 2022. Available: https://www.gymlibrary.dev

[13] Philips, "AI-Powered Diagnostic Imaging with MLOps," 2023. Available: https://www.philips.com/a-w/about/news/archive/standard/news/articles/2023/ai-powered-diagnostic-imaging-with-mlops.html

[14] Ray Project, "Distributed Hyperparameter Tuning with Ray Tune," 2023. Available: https://docs.ray.io/en/latest/tune/index.html

[15] Unity ML-Agents Toolkit, "Training Intelligent Agents," 2022. Available: https://github.com/Unity-Technologies/ml-agents