Implementation of an MLOps Pipeline: From ETL to Deployment

Tanish Pathania
IMT2022049
IIIT-Bangalore

Dr. Thangaraju B
Professor
IIIT-Bangalore

Siddharth Vikram
IMT2022534
IIIT-Bangalore

***Abstract*—We deliver here a production-grade MLOps [1] pipeline that accommodates the entire machine learning application lifecycle from ingestion of data to deployment in near-time in a Kubernetes context. The pipeline has three primary phases of operation. First, a mature ETL process reads in semi-structured and structured data, cleanses and transforms it, and saves the output in a MongoDB instance that is scalable. Second, dynamic model training reads in fresh data from MongoDB and employs reproducible, modulated scripts to achieve on-demand, decoupled computation. Third, all parts of it are Dockerized and shipped to Minikube hosting in a local cluster of Kubernetes. Deployment and service configurations reveal near-time inference via NodePort services. The system follows current DevOps techniques of modulated deployment, compositionality, and scaling and forms the basis of realizing additional extensions such as CI/CD and observability in the near future and deployment to the cloud. This is a working, cloud-agnostic method of deployment and scaling of ML systems.**

***Index Terms*—MLOps, Machine Learning, ETL Pipeline, Docker, MongoDB, Kubernetes, Minikube, CI/CD, Containerization**

I INTRODUCTION

Machine learning (ML) deployment in production is usually hindered by disjointed workflows, manual intervention, and a pipeline that is not end-to-end automated. Conventional ML pipelines—covering data ingestion through to deployment—are normally disjointed, script-based, and not very much integrated with contemporary DevOps methods. This disjointedness creates major hurdles in scaling, reproducibility, and maintainability and also inhibits the operationalization of ML models in actual applications.

A number of systemic problems lie behind this inefficiency. Pipelines are often constructed as discrete pieces that interact with each other in places where friction arises during multiple-staged transitions. Repetitive and manual work is typical of retraining and deployment tasks with little to no automation. Lack of systematic version control makes reproducing results or auditing model activity upon deployment even more challenging. These deficiencies worsen with datasets increasing and model architectures expanding in size and scope so that the limitations of the normal ML workflow in performance, observability, and operational burden stand revealed.

Moreover, conventional machine learning systems often lack seamless integration with DevOps tooling. Unlike software development pipelines, ML workflows involve non-deterministic outputs, evolving data dependencies, and iterative experimentation—all of which require specialized infrastructure to manage effectively. Without robust CI/CD integration, model development and deployment remain siloed, delaying delivery cycles and increasing technical debt.

As a response to these constraints, here is a unified and modular pipeline of MLOps that automatically manages the entire machine learning life cycle. It is organized in three main components. An ETL pipeline is used to retrieve and transform raw data and store it in a MongoDB database in the first place. Secondly, a dynamic training module fetches the same data during runtime to train the ML algorithms without depending upon persistent intermediate state. Thirdly, the pipeline is making use of Docker in order to dockerize the training and inference workflow with Kubernetes—provisioned locally with Minikube—handling deployment and orchestration of services.

By combining containerization, orchestration, and scale-out storage with DevOps best practices, this MLOps workflow meets the key shortcomings of existing ML systems. The pipelines resulting from it guarantee reproducibility and ease of deployment and set the stage for hassle-free CI/CD compatibility, making it a promising base for production-ready machine learning deployments.

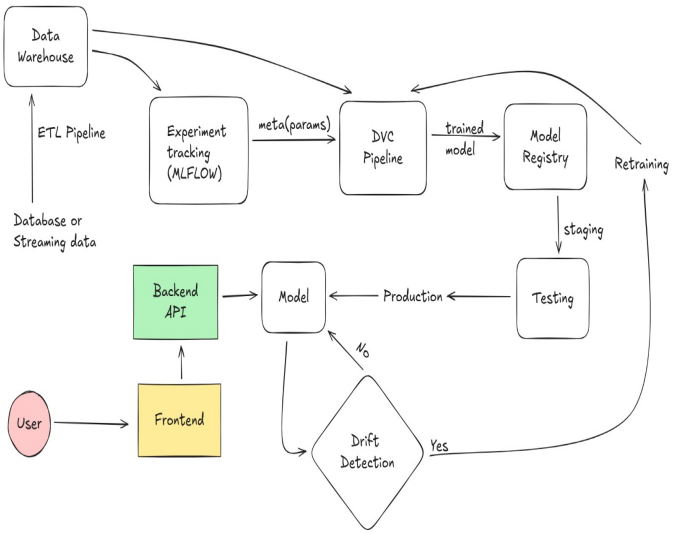

Fig. 1. Flowchart demonstrating MLOps Pipeline

II RELATED WORK

A Open-Source MLOps Frameworks

A number of open-source tools have driven MLOps best practices. Prominent among these are MLflow [2], Kubeflow [3], and TensorFlow Extended (TFX) [4]. MLflow eases experiment tracking, model tracking, and automated deployment. Kubeflow, built to be compatible with Kubernetes, offers an end-to-end orchestration and monitoring solution for ML pipelines. TFX, built by Google, provides end-to-end automated ML pipelines [5], with TensorFlow-based environments in mind.

B. Industry Implementations

At scale, in the industry, strong examples of this can be observed in large-scale systems such as Uber's Michelangelo [6] and Google's TFX pipeline [4]. Michelangelo unifies model training, deployment, and monitoring, specifically designed to be suited to Uber's high-demand requirements. The TFX pipeline of Google, though TensorFlow-centric, offers strong scalable ML workflows capabilities suitable in production environments.

C. Key Differences and Limitations

These platforms do well with individual aspects of the ML lifecycle, though they can be complicated to configure and don't always provide flexibility to use anything other than TensorFlow. In addition, they tend to be geared toward scalable environments and therefore less suited to small teams or low-overhead environments.

It is an enhancement of those principles applied to offer an integrated, modular pipeline that incorporates all of ML's stages, ranging from ETL to deployment, to promote flexibility and scalability and to make setup simpler.

Framework	Key Features	Primary Focus	Compatibility
MLflow	Experiment tracking, model management, deployment automation	General-purpose ML lifecycle management	Flexible, supports multiple frameworks
Kubeflow	ML workflow orchestration, Kubernetes-based deployment	ML model deployment and orchestration	Best for Kubernetes environments
TensorFlow Extended (TFX)	End-to-end ML pipeline automation, TensorFlow-centric	End-to-end pipeline automation for TensorFlow	Optimized for TensorFlow models
Michelangelo (Uber)	Model training, deployment, monitoring at scale	Large-scale ML deployment at Uber	Customized for Uber's infrastructure
Google TFX Pipeline	Scalable ML workflows, TensorFlow support	Production-grade ML pipelines	TensorFlow-centric, scalable

Fig. 2. Comparison of Open-Source MLOps Frameworks

III. METHODOLOGY

A. ETL Pipeline Implementation

The ETL pipeline [7] of this project consists of a number of individual components, each performing some particular task as part of the data processing pipeline. The orchestration of these components is controlled by etl_pipeline.py, and individual scripts handle particular parts of the pipeline.

1) Data Extraction

Data extraction from raw CSV files in the `data/raw/` directory is handled by the `extract.py` module. The main pipeline orchestrator starts the data extraction process and makes sure that raw data is loaded into a format that can be transformed. Each CSV file is parsed, processed, and stored in memory for later steps by the extraction script.

```
import pandas as pd
import os
from logger import get_logger

logger = get_logger()

def extract_data(data_folder="../data/raw"):
   dataframes = {}
   try:
      for file in os.listdir(data_folder):
         if file.endswith(".csv"):
            file_path = os.path.join(data_folder,
                file)
            df = pd.read_csv(file_path)
            dataframes[file[:-4]] = df
            logger.info(f"Extracted {len(df)}
                records from {file}.")
   except Exception as e:
      logger.error(f"Error extracting data: {e}")
   return dataframes
```

Listing 1. extract.py – Data Extraction from CSVs

2) Data Transformation and Cleaning

- **`Handling Missing Data`** are among the crucial data cleaning and preprocessing tasks carried out by the `transform.py` module: In order to deal with missing values, the module either removes the corresponding rows or columns or imputes them.
- **`Feature Selection`**: Choosing pertinent features for model training may be a part of the transformation step.
- **`Normalization and Scaling`**: This guarantees the standardisation of data features, particularly if the model calls for input within a consistent range.

These processes get the data ready for database loading and model training.

```
import pandas as pd
from logger import get_logger

logger = get_logger()
```

```
def transform_data(dataframes):
    for name, df in dataframes.items():
        for column in df.columns:
            if df[column].isnull().any():
                if df[column].dtype == 'object':
                    df[column] = df[column].fillna(df[
                        column].mode()[0])
                else:
                    df[column] = df[column].fillna(df[
                        column].mean())
        logger.info(f"Transformed data for {name}:
            filled missing values.")
    return dataframes
```

Listing 2. transform.py – Data Cleaning and Transformation

3) Data Loading to MongoDB

Several essential functions are integrated by the `load.py` module:

- **`Logging`**: It makes use of the `logger.py` module to make sure that every pipeline step is recorded for monitoring and debugging.
- **`Data Validation`**: The `validate.py`: module verifies the consistency and quality of the data before loading it into MongoDB. The `database.py` module is used to dynamically input the cleaned data into a MongoDB database, abstracting the connection logic and schema design.

The schema is made to allow for flexible querying for model training and further analysis, and the MongoDB database is organised to manage the data effectively.

4) Pipeline Orchestration

The `etl_pipeline.py` script orchestrates the entire ETL pipeline. By combining the loading, extraction, and transformation processes, this script makes sure that data moves smoothly from one step to the next. It is in charge of controlling the pipeline's overall execution and invoking the relevant functions in each module.

```
from extract import extract_data
from transform import transform_data
from load import load_data_to_mongo
from logger import get_logger
logger = get_logger()
if __name__ == "__main__":
    logger.info("Starting ETL Pipeline...")
    dataframes = extract_data()
    transformed_data = transform_data(dataframes)
    load_data_to_mongo(transformed_data)
    logger.info("ETL Pipeline completed successfully!
        ")
```

Listing 3. etl_pipeline.py – Orchestration of ETL Stages

B. Execution and Setup with Makefile

A `Makefile` is used to control the pipeline's execution, automating the setup, execution, and cleanup procedures. Several significant targets are included in the `Makefile`:

- **`run`**: Completes the ETL pipeline by invoking the main pipeline script (`etl_pipeline.py`), executing the extraction and validation processes, and setting up the required directories.
- **`install`**: Installs all necessary dependencies as listed in **`requirements.txt`**.
- **`clean`**: Eliminates logs and temporary files produced while the pipeline was running. item textbf texttttest: Verifies that the extracted data satisfies quality standards by performing a data validation check prior to processing starting.
- **`check-env`**: Confirms that the environment is set up correctly by ensuring that important environment variables like **`MONGO_URI`** are set and that required files like **`.env`** are present.

File References

The following is how the above-described components are arranged in the project directory:

- **`config.py`**: Manages configuration details specific to one's environment.
- **`database.py`**: Oversees MongoDB interactions and schema design.
- **`logger.py`**: Offers logging functionality for monitoring pipeline execution.
- **`extract.py`**: Oversees the process of extracting raw data from CSV files.
- **`transform.py`**: Implements preprocessing and data cleaning procedures.
- **`validate.py`**: Verifies the quality of the data before it is loaded into the database.
- **`etl_pipeline.py`**: Coordinates the complete ETL procedure.

When combined, these files provide a robust and adaptable ETL pipeline that can readily grow to accommodate more data sources or transformations.

C. Model Training

This project element exemplifies contemporary machine learning engineering. Its sturdy, modular architecture allows it to seamlessly and automatically coordinate the training process while dynamically retrieving data from MongoDB. The architecture makes use of MLflow's sophisticated experiment tracking to guarantee that each training cycle is auditable and repeatable. The dynamic data retrieval mechanism ensures that the model uses the most recent dataset available at runtime by removing reliance on static storage.

1) Dynamic Data Retrieval from MongoDB

By creating a robust connection to MongoDB using a standardised configuration supplied in `config.py`, the `data_warehouse_connect.py` module plays a crucial role. With each execution, this design guarantees that the

pipeline retrieves the most recent data. Using this dynamic connection, the `preprocess.py` script performs extensive data cleaning, feature engineering, and normalisation, methodically preparing the dataset for the best possible training results.

By using the most recent database entries, this procedure guarantees that the data is cleaned and ready for training.

2) Model Training Pipeline

The `train.py` script, which is in charge of training a logistic regression model on the preprocessed data, is the central component of this work. It logs metrics, models, hyperparameters, and experiments using MLflow. The following steps are part of the training pipeline:

- loading the target variable `y.csv` and the preprocessed feature set `X.csv`.
- dividing the data into sets for testing and training.
- To guarantee the logistic regression [8] model performs at its best, the features are scaled using `StandardScaler`.
- Using the test set to train and assess the logistic regression model.
- logging the accuracy score, model coefficients, and other training and model metrics to MLflow [2].

The core training logic is demonstrated in the following snippet, `train.py`, which covers data loading, model training, evaluation, and logging via MLflow

```
import pandas as pd, mlflow, mlflow.sklearn
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

mlflow.set_experiment("Loan_Approval_Experiment")
with mlflow.start_run():
   X = pd.read_csv("data/X.csv")
   y = pd.read_csv("data/y.csv")["Loan_Status"]
   X_train, X_test, y_train, y_test =
       train_test_split(X, y, test_size=0.2)

   scaler = StandardScaler()
   X_train = scaler.fit_transform(X_train)
   X_test = scaler.transform(X_test)

   model = LogisticRegression(max_iter=1000).fit(
       X_train, y_train)
   mlflow.log_metric("accuracy", accuracy_score(
       y_test, model.predict(X_test)))
   mlflow.sklearn.log_model(model, "
       logistic_regression_model")
```

This pipeline guarantees that all artefacts are saved for future use and that the model is trained reproducibly.

The `Makefile` is used to execute this orchestration, which simplifies the procedure and exhibits excellent modularity and repeatability. By encapsulating the end-to-end execution, the commands `make preprocess`, `make train`, and `make all` strengthen the system's efficiency and integrity.

3) Model Inference Endpoint

To serve predictions using Flask, the `inference.py` script offers a RESTful endpoint. It converts incoming JSON data, loads the scaler parameters and trained model coefficients, and generates binary loan approval predictions.

```
@app.route("/predict", methods=["POST"])
def predict():
   data = request.get_json()
   df = pd.DataFrame(data)
   df = df[coefficients.index]
   df = (df - scaler_mean) / scaler_var.pow(0.5)
   linear_output = df.mul(coefficients).sum(axis=1)
       + intercept
   predictions = (linear_output > 0).astype(int).
       tolist()
   return jsonify(predictions)
```

This stateless microservice enables seamless integration with downstream applications and deployment pipelines.

4) Execution Control via Makefile

In addition to automating the workflow's essential steps, the `Makefile` guarantees that each stage can be independently tested and deployed. This incorporation:

- minimises the possibility of human error by reducing manual overhead. offers a scalable solution that can be incorporated into continuous delivery pipelines for CI/CD. keeps a high level of flexibility to adjust to changing production needs.

```
all: preprocess train
preprocess:
 python src/preprocess.py
train:
 python src/train.py
inference:
 python src/inference.py
install:
 pip install -r requirements.txt
setup:
 mkdir -p logs data/raw model
clean:
 rm -rf logs/*.log __pycache__ model data mlruns
```

The combination of dynamic data handling, thorough preprocessing, and efficient model training is embodied by this model training framework, which is more than just a set of scripts. It is a sophisticated, scalable solution. In the field of MLOps, it is a remarkable piece of work that exemplifies both operational effectiveness and technical excellence.

D. Containerization and Deployment

Docker [9] and Kubernetes are used to containerise and deploy the entire workflow, which includes preprocessing, training, and inference, in order to operationalise the machine learning pipeline in a reproducible and scalable way. The deployment method using Minikube and the step-by-step containerisation strategy are explained in this section.

1) Dockerization

Each stage of the ML lifecycle is divided into distinct, self-contained layers using a multi-stage `Dockerfile` [10].

Using a lightweight Python base image, the first stage manages preprocessing. Dependencies are installed using a specific `stage_1.txt` requirements file, source code is copied, and the preprocessing script is run to produce feature and label datasets that are clean.

To create a trained model and matching scaler parameters, the second stage imports the artefacts from the first stage, installs the training-specific dependencies mentioned in `stage_2.txt`, and runs `train.py`. The third and last step involves using Flask to instantiate an inference server. The stage exposes the application on port 5000 and contains the trained artefacts from the training stage as well as all runtime dependencies from `stage_3.txt`.

```
# Stage 1: Preprocessing
FROM python:3.12-slim AS stage1
WORKDIR /app
COPY requirements/stage_1.txt ./
RUN pip install -r stage_1.txt
COPY src/ src/
RUN python src/preprocess.py # Generates cleaned
    data

# Stage 2: Training
FROM python:3.12-slim AS stage2
COPY --from=stage1 /app/data/X.csv /app/data/y.csv
    ./data/
COPY requirements/stage_2.txt ./
RUN pip install -r stage_2.txt
RUN python src/train.py # Produces model/scaler

# Stage 3: Inference
FROM python:3.12-slim AS stage3
COPY --from=stage2 /app/model/*.csv ./model/
COPY requirements/stage_3.txt ./
RUN pip install -r stage_3.txt
EXPOSE 5000
CMD ["flask", "run", "--host=0.0.0.0"] # Web service
```

Listing 4. Multi-stage Dockerfile

For every pipeline component, this design pattern guarantees clear dependency management, smaller image sizes, and concern separation. The `--from=stageX` directive is used to pass artefacts between stages, preserving state across layers without requiring the persistence of intermediate containers.

2) Image Distribution via Docker Hub

To ensure safe and automated publishing, each stage-specific image is tagged and uploaded to Docker Hub.

The build and run lifecycle of every component is coordinated locally by the `docker-compose.yml` file. `depends_on` constraints are used to run the services in a sequential manner, making sure that the preprocessing container is finished before training starts and that training is finished before inference starts. To transfer data and model artefacts between pipeline stages, shared host directories are mounted into the containers. Before cluster deployment, this setup facilitates end-to-end validation, local reproducibility, and smooth debugging.

3) Deployment using Minikube

The pipeline is installed on a local Kubernetes cluster that Minikube has provisioned for production orchestration and simulation. Declarative manifests located in the `k8s/` directory define the deployment.

The `inference-deployment.yaml` file sources the image from Docker Hub and defines a Kubernetes deployment with a single replica of the inference container. In order to improve portability and cluster independence, ephemeral volumes are mounted using `emptyDir` to replicate shared storage in a cloud-native manner without depending on hostPath volumes.

```
apiVersion: apps/v1
kind: Deployment
spec:
 replicas: 1
 template:
   spec:
    containers:
    - name: inference
      image: tanish688/experiment_tracking-inference
          :latest
      ports:
        - containerPort: 5000
      volumeMounts: # Ephemeral storage
        - mountPath: /model
          name: model-volume
    volumes:
      - name: model-volume
        emptyDir: {}
```

Listing 5. Deployment Manifest

The corresponding `inference-service.yaml` defines a `NodePort` service that maps the container's port 5000 to a node-level port 30007, enabling external access to the model inference endpoint.

```
apiVersion: v1
kind: Service
spec:
 type: NodePort
 ports:
   - port: 5000
     nodePort: 30007 # External access
 selector:
   app: inference
```

Listing 6. NodePort Service

In addition to encapsulating DevOps best practices like versioned artefact tracking and containerised delivery, this infrastructure uses microservices and infrastructure as code to validate real-world deployment strategies. For downstream integration, the Docker and Kubernetes stack offers modularity, scalability, and CI/CD readiness.

IV. RESULTS AND DISCUSSION

Kubernetes deployment, image distribution, and container orchestration were used to confirm that the entire MLOps pipeline was executed successfully. The experimental procedure

is described in this section, along with important findings from each phase.

A. Containerization Verification

The containerized ML stages—preprocessing, training, and inference—were independently built and tested using Docker Compose with the following command:

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

The detached mode execution ensured all services ran concurrently in isolated environments. Figure 3 displays the Compose output, confirming successful builds and service execution.

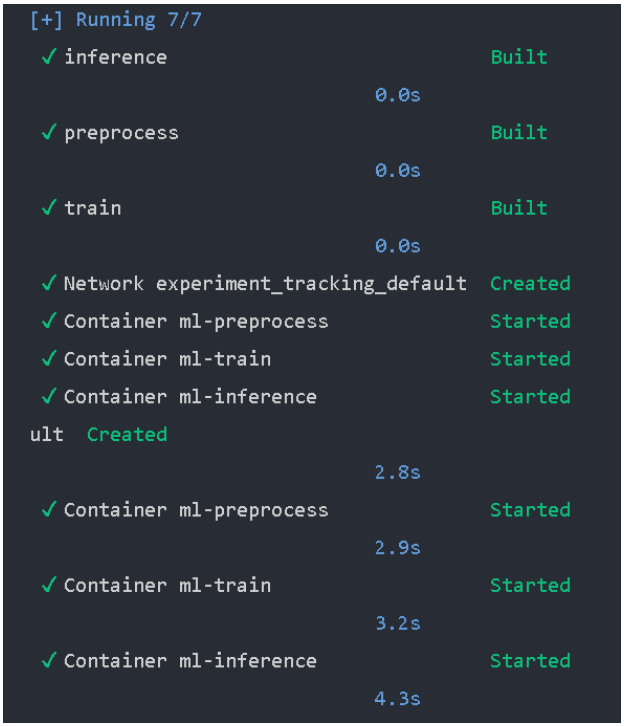

Fig. 3. Docker Compose execution of all ML pipeline stages

Because of redundant files and bundled dependencies, the initial Docker images weighed about 3 GB. In order to address this, we used Docker volumes to externalise intermediate artefacts and data, which resulted in a 66.7% reduction in image size to 1 GB. We separated the build environment from the lightweight runtime image and used a multi-stage build approach to further optimise. By eliminating superfluous build-time dependencies, the final image size was 300 MB, which was 70% smaller than the 1 GB intermediate image and 90% smaller than the initial 3 GB baseline. Our Kubernetes-based infrastructure's deployment efficiency, container startup time, and resource usage were all greatly enhanced by these optimisations, which made use of docker volume [11].

Optimization Stage	Image Size	Reduction
Initial (Baseline)	3 GB	—
After Using Volumes	1 GB	66.7%
After Multi-stage Build	300 MB	70% from 1 GB
Total Reduction	**300 MB**	**90% from 3 GB**

TABLE I
DOCKER IMAGE SIZE REDUCTION ACROSS OPTIMIZATION STAGES

B. Image Registry Push and Optimization

We used Docker Hub, a centralised registry for storing and sharing our container images, to make it easier to collaborate and deploy across various environments. We supported a continuous integration and delivery (CI/CD) [12] workflow by facilitating smooth sharing between the development, testing, and production phases by pushing versioned images to Docker Hub.

Every pipeline image was successfully uploaded to Docker Hub by means of:

```
docker push tanish688/experiment_tracking-preprocess
    :latest
docker push tanish688/experiment_tracking-train:
    latest
docker push tanish688/experiment_tracking-inference:
    latest
```

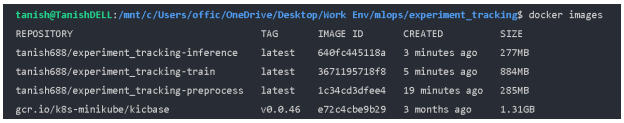

Fig. 4. Container images published on Docker Hub

C. Kubernetes Deployment and Validation

The inference service was deployed to a local Kubernetes cluster using the following commands:

```
kubectl apply -f k8s/inference-deployment.yaml
kubectl apply -f k8s/inference-service.yaml
```

Cluster state and logs were verified to ensure the pod was running and the inference service was correctly exposed via NodePort. Figure 5 presents a snapshot of the pod and service states, confirming successful deployment.

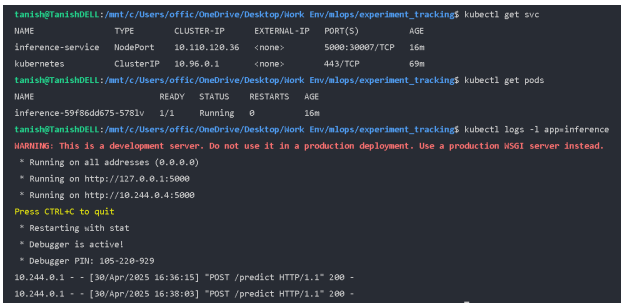

Fig. 5. Kubernetes pod and service logs confirming deployment

D. ETL and Experiment Pipeline Execution

The ETL pipeline and experiment tracking components were executed more efficiently thanks to the automation made possible by Makefile. By offering discoverable CLI operations, this interface improved repeatability and decreased manual errors.

The ETL pipeline's help menu is displayed in Figure 6, and its execution output is shown in Figure 7. Similarly, the help and result output for the experiment tracking pipeline are shown in Figures 8 and 9.

```
(venv) tanish@TanishDELL:/mnt/c/Users/offic/OneDrive/Desktop/Work Env/mlops/etl_pipeline$ make help
Available targets:
  run           : Execute the ETL pipeline (default)
  install       : Install Python dependencies
  setup         : Create required directories
  clean         : Remove generated files
  test          : Run Validation Test
  check-env     : Check Environment
  help          : Show this help message
```

Fig. 6. ETL pipeline Makefile help command output

```
(venv) tanish@TanishDELL:/mnt/c/Users/offic/OneDrive/Desktop/Work Env/mlops/etl_pipeline$ make run
Setting up directories...
Directory structure ready.
Starting ETL Pipeline...
Extracted 367 records from test.csv.
Extracted 614 records from train.csv.
Data contains missing values.
Data validation passed. 367 records ready for loading.
Data contains missing values.
Data validation passed. 614 records ready for loading.
Starting ETL Pipeline...
Extracted 367 records from test.csv.
Extracted 614 records from train.csv.
Transformed data for test: filled missing values.
Transformed data for train: filled missing values.
Connected to MongoDB.
Successfully upserted 367 records into test collection in MongoDB.
Successfully upserted 614 records into train collection in MongoDB.
ETL Pipeline completed successfully!
ETL process completed!
```

Fig. 7. Result of executing the ETL pipeline

```
(venv) tanish@TanishDELL:/mnt/c/Users/offic/OneDrive/Desktop/Work Env/mlops/experiment_tracking$ make help
Available targets:
  all         : Run preprocessing and training (default)
  preprocess  : Run preprocessing step
  train       : Run training step
  inference   : Run inference step
  install     : Install Python dependencies
  setup       : Create necessary directories
  clean       : Remove generated files
  help        : Show this help message
```

Fig. 8. Experiment tracking Makefile help output

```
(venv) tanish@TanishDELL:/mnt/c/Users/offic/OneDrive/Desktop/Work Env/mlops/experiment_tracking$ make
Running preprocessing...
Preprocessing completed.
Running training...
Model Accuracy: 0.79
```

Fig. 9. Execution output of experiment tracking pipeline

E. Full Request Lifecycle Validation

A terminal-based test of the deployed inference API confirmed a full data-to-inference lifecycle. Figure 10 shows a successful request-response exchange between the client and the deployed model endpoint.

```
(venv) tanish@TanishDELL:/mnt/c/Users/offic/OneDrive/Desktop/Work Env/mlops$
curl -X POST "http://192.168.49.2:30007/predict" -H "Content-Type: application/json" -d '[
  {
    "ApplicantIncome": 5000,"CoapplicantIncome": 0.0,
    "Credit_History": 1.0,"LoanAmount": 150.0,
    "Loan_Amount_Term": 360.0,"Gender_Female": 0.0,
    "Gender_Male": 1.0,"Married_No": 1.0,
    "Married_Yes": 0.0,"Dependents_0": 1.0,
    "Dependents_1": 0.0,"Dependents_2": 0.0,
    "Dependents_3+": 0.0,"Education_Graduate": 1.0,
    "Education_Not Graduate": 0.0,"Self_Employed_No": 1.0,
    "Self_Employed_Yes": 0.0,"Property_Area_Rural": 0.0,
    "Property_Area_Semiurban": 1.0,"Property_Area_Urban": 0.0
  }
]'
[
  1
]
```

Fig. 10. End-to-end lifecycle test of deployed model via API

Using structured JSON input, the test sent a POST request to `http://192.168.49.2:30007/predict`. The inference engine successfully processed the input and returned a valid prediction, as indicated by the service's response of `[1]`. This confirms that:

- Traffic is being routed correctly by the Kubernetes infrastructure (Deployment + Service).
- The inference logic works and the model loads successfully. The network and serialisation layers are functioning properly.

F. Discussion

The pipeline accomplished the three main MLOps [13] goals of deployability, reproducibility, and modularisation. Host-level dependencies were abstracted away by containerisation. Docker Create a local orchestration that has been verified. Remote reuse was made possible by image pushing. To serve real-time machine learning models, Kubernetes acted as a platform-neutral orchestrator. Volume mounts, service exposure mechanisms, and logs all worked as they should have.

The complete source code for this implementation is available in our GitHub repository. To access it, refer https://github.com/Tanish-pat/MLOps.

Future research can concentrate on CI/CD automation, live monitoring of prediction quality and service health, and persistent storage [14] for model registries.

V. CONCLUSION

This project shows how data engineering, model training, containerisation, and Kubernetes-based deployment can all be seamlessly integrated into an end-to-end MLOps pipeline. From ETL procedures that extract, validate, and load raw data into MongoDB to model training initiated by dynamic runtime data access, the pipeline is modular, reproducible, and dynamically

orchestrated. Minikube was used to containerise, distribute, and deploy the application in a local environment with reliability using Docker and Kubernetes.

This system guarantees maintainability, lowers operational overhead, and encourages faster iteration by abstracting the ML lifecycle into distinct, automatable stages. In the field of machine learning, the use of Makefiles, logging systems, and multi-stage Dockerfiles strengthens software engineering discipline. All things considered, the project offers a local-first, production-grade template that can be expanded and scaled for enterprise machine learning workflows.

VI. FUTURE WORK

While the pipeline is complete in its current state, several enhancements are planned to further increase its robustness and production-readiness:

- **CI/CD Integration:** Incorporate GitHub Actions or Jenkins for continuous integration and deployment, ensuring automatic rebuilds, tests, and deployments on code changes.
- **Cloud-Native Deployment:** Extend the current local Kubernetes deployment to managed cloud services such as Google Kubernetes Engine (GKE), Amazon EKS, or Azure AKS for improved scalability and availability.
- **Model Versioning and Monitoring:** Integrate MLflow Tracking Server or Weights & Biases to manage model versions, monitor performance in production, and enable rollback capabilities.
- **Real-Time Data Ingestion:** Migrate the ETL pipeline to a streaming architecture using Kafka or Spark Streaming to support real-time model updates.
- **Role-Based Access Control (RBAC):** Secure model endpoints and database access through Kubernetes-native authentication mechanisms and secret management tools.

These enhancements will help mature the system into a fully production-grade MLOps platform, capable of supporting high-volume, low-latency machine learning applications at scale.

REFERENCES

[1] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine learning operations (mlops): Overview, definition, and architecture," *IEEE Access*, vol. 11, pp. 31 866–31 879, 2023.

[2] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe *et al.*, "Accelerating the machine learning lifecycle with mlflow." *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.

[3] J. George and A. Saha, "End-to-end machine learning using kubeflow," in *Proceedings of the 5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD)*, 2022, pp. 336–338.

[4] D. Baylor, K. Haas, K. Katsiapis, S. Leong, R. Liu, C. Menwald, H. Miao, N. Polyzotis, M. Trott, and M. Zinkevich, "Continuous training for production {ML} in the {TensorFlow} extended ({{{{{TFX}}}}}) platform," in *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, 2019, pp. 51–53.

[5] S. Wazir, G. S. Kashyap, and P. Saxena, "Mlops: A review," *arXiv preprint arXiv:2308.10908*, 2023.

[6] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, 2015.

[7] A. Raj, J. Bosch, H. H. Olsson, and T. J. Wang, "Modelling data pipelines," in *2020 46th Euromicro conference on software engineering and advanced applications (SEAA)*. IEEE, 2020, pp. 13–20.

[8] M. P. LaValley, "Logistic regression," *Circulation*, vol. 117, no. 18, pp. 2395–2399, 2008.

[9] R. C. Díaz Alonso, "Software containerization with docker," 2017.

[10] N. Badisa, J. K. Grandhi, L. Kallam, M. R. Eda, S. Bulla *et al.*, "Efficient docker image optimization using multi-stage builds and nginx for enhanced application deployment," 2023.

[11] I. Miell and A. Sayers, *Docker in practice*. Simon and Schuster, 2019.

[12] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE access*, vol. 5, pp. 3909–3943, 2017.

[13] F. Zhengxin, Y. Yi, Z. Jingyu, L. Yue, M. Yuechen, L. Qinghua, X. Xiwei, W. Jeff, W. Chen, Z. Shuai *et al.*, "Mlops spanning whole machine learning life cycle: A survey," *arXiv preprint arXiv:2304.07296*, 2023.

[14] I. Konev, I. Nikiforov, and S. Ustinov, "Algorithm for containers' persistent volumes auto-scaling in kubernetes," in *2022 31st Conference of Open Innovations Association (FRUCT)*. IEEE, 2022, pp. 89–95.

ABOUT THE AUTHORS

The authors are affiliated with the Open Source Technology Lab at the International Institute of Information Technology, Bengaluru. Our collective expertise spans open-source systems, distributed computing, and advanced software engineering.

- Dr. Thangaraju B – Professor with extensive contributions in open-source technologies and system architecture.
- Tanish Pathania – Student focused on scalable system design and community-driven development.
- Siddharth Vikram – Student with a specialization in distributed systems and open-source toolchains.